Unknown date | Unknown author

Python __Underscore__ Methods

Contents

1 Intro

My intention for this article is to be a quick-reference guide: all the information you might need about a method or property condensed into a few lines. This might not be possible for something like ___slots___, perhaps we should link to the Python Docs for details on the more complex ones. I know this stuff is documented in other places (for example, http://docs.python.org/ref/specialnames.html), but it's all over the place and hard to mentally parse. I want to teach while at the same time to be brief, if that is possible.

Anyways feel free to add, don't worry about messing stuff up, I can always revert your edits if I really hate them ;) A list of underscore methods that I haven't written about yet is at the bottom; there might be more too that I don't know about. And things I'm not sure about have question marks next to them, feel free to search for question marks and correct any uncertainties.

2 Class and Instance Methods

For consistency's sake, let's say we have a class called 'Class', instances called 'x' and 'y'. Keeping with the Python docstrings, '<==>' can be read as 'is equivalent to'.

2.1 Creation, Calling, and Destruction

`__init__(self, ...)`
`Class.__init__(...)` <==> `x.__init__(...)` <==> `Class(...)`
Called when **class** is called, or in other words when you instantiate the class. Shouldn't return any value; the created instance will be returned automatically.
If not present, nothing is called and life goes on.

`__call__(self, ...)`
`x.__call__(...)` <==> `x(...)`
Called when **instance** is called.
If not present, instance is not callable and a `TypeError` (object is not callable) or an `AttributeError` is raised. (?)

`__del__(self)`
Called when instance is about to be destroyed. This happens when the reference count reaches zero, probably because the instance was `del`'ed.
If not present, nothing is called and life goes on.

2.2 Conversion to Strings

`__repr__(self)`
`x.__repr__()` <==> `repr(x)` <==> `` `x` ``
Should return a string representation of the class or instance with as much information as possible, preferably something that can be passed to `eval` to recreate the object. Return value must be a string.
If not present, returns something like '<class ___main___.Class at 0x2e6d50>' for classes and '<___main___.Class instance at 0xbfb70>' for instances.

`__str__(self)`

`x.__str__() <==> str(x) <==> print x`

Should return an informal, user-friendly string describing the class or instance. Return value must be a string.

If not present, `__repr__` is tried. If that is not present, returns something like '___main___.Class' for classes and '<___main___.Class instance at 0xbfb70>' for instances.

`__unicode__(self)`

`x.__unicode__() <==> unicode(x)`

Should return an informal, user-friendly unicode string describing the class or instance. Return value must be unicode string, or a string or buffer (which will be automatically converted into a unicode string).

If not present, `__str__` is called and the result is converted into a unicode string.

2.3 Truth Testing

`__nonzero__(self)`

`bool(x) <==> x.__nonzero__()`

Called by the built-in function `bool`, or whenever any truth-value test occurs.

Should return `False`, `True`, `0`, or `1`.

If not defined, `__len__` is called; if both are not defined the instance always evaluates to `True`.

2.4 Comparisons

For instances to be meaningfully sorted, either `__cmp__`, `__lt__`, or `__gt__` must be defined. If more than one is defined, `__lt__` will be tried first, followed by `__gt__` and then finally `__cmp__`.

`__cmp__(self, other)`

`x.__cmp__(other) <==> cmp(x, other)`

Should return a negative integer if instance is **less than** `other`, `0` if instance is **equal to** `other`, or a positive integer if instance is

greater than `other`.
If not present (and other comparisons aren't present?) instances are compared by object identity ('address') (?)

2.4.1 "Rich Comparison" Operators

The following functions customarily return `True` or `False`, but can return any value. If in a boolean context, Python will call bool() on the returned value to determine truthfulness. `NotImplemented` should be raised if no valid comparison can be made.

If one of these methods is not defined, Python will try the opposite method with swapped arguments. For example, if you call `x < other` but `x.__lt__` is not defined, Python will try the equivalent `other.__gt__(x)`. If neither is not defined, `x.__cmp__(other)` will be tried, followed by other.___cmp___(x). If all four methods are undefined, instances are compared by object identity (address?).

Note that no other fallbacks will be performed. For example, if `__le__` is not defined, Python does not call `lt` and `eq`. Similarly, Python will not fallback to `__ne__` if `__eq__` is not defined, or vice versa -- this can result in two instances that are 'equal' and 'not equal' at the same time.

There are no explicit 'reflected' (swapped-argument) versions of the comparison operators.

`__eq__(self, other)`
`x.__eq__(other)` `<==>` `x == other`
Must be implemented if two instances are to be perceived as identical, for example in a set.
Should return a value indicating if instance is **equal to** `other`.
If not defined or returns `NotImplemented`, `other.__eq__(x)` will be tried, followed by x.___cmp___(other) then other.___cmp___(x).

`__ne__(self, other)`
`x.__ne__(other)` <==> `x != other`
Should return a value indicating if instance is **not equal to** `other`.
If not defined or returns `NotImplemented`, `other.__ne__(x)` will be tried, followed by x.___cmp___(other) then other.___cmp___(x).

`__lt__(self, other)`
`x.__lt__(other)` <==> `x < other`
Should return a value indicating if instance is **less than** `other`.
If not defined or returns `NotImplemented`, `other.__gt__(x)` will be tried, followed by x.___cmp___(other) then other.___cmp___(x).

`__le__(self, other)`
`x.__le__(other)` <==> `x <= other`
Should return a value indicating if instance is **less than or equal to** `other`.
If not defined or returns `NotImplemented`, `other.__ge__(x)` will be tried, followed by x.___cmp___(other) then other.___cmp___(x).

`__ge__(self, other)`
`x.__ge__(other)` <==> `x >= other`
Should return a value indicating if instance is **greater than or equal to** `other`.
If not defined or returns `NotImplemented`, `other.__le__(x)` will be tried, followed by x.___cmp___(other) then other.___cmp___(x).

`__gt__(self, other)`
`x.__gt__(other)` <==> `x > other`
Should return a value indicating if instance is **greater than** `other`.

If not defined or returns `NotImplemented`, `other.__lt__(x)` will be tried, followed by x.___cmp___(other) then other.___cmp___(x).

2.5 Logical and Mathematical Operators

2.5.1 Regular Binary Operations

These methods implment the binary operators `+`, `-`, `, /, //, %, *`, `<<`, `>>`, &, |, and `^`, and the functions `divmod` and `pow`.

The method may raise `NotImplemented` if no operation is possible with the supplied with the supplied argument(s).

If not defined or if returns `NotImplemented`, and the two instances belong to **different classes**, Python will try to call the equivalent reflected binary operator method of the `other` instance, with this instance as the argument.

If that method is also not defined, the relevant operation is not possible and a TypeError ('unsupported operand type') is raised.

`__add__(self, other)`
`x.__add__(other) <==> x + other`
If not defined or returns `NotImplemented`, `other.__radd__(x)` is tried.

`__sub__(self, other)`
`x.__sub__(other) <==> x - other`
If not defined or returns `NotImplemented`, `other.__rsub__(x)` is tried.

`__mul__(self, other)`
`x.__mul__(other) <==> x * other`
If not defined or returns `NotImplemented`\`,
``other.__rmul__(x)` is tried.

`__div__(self, other)`
`x.__div__(other)` <==> `x / other`
If `from __future__ import division` is used, `__truediv__` is called instead.
If not defined or returns `NotImplemented`, `other.__rdiv__(x)` is tried.

`__truediv__(self, other)`
`x.__truediv__(other)` <==> `x / other`
Only called if `from __future__ import division` is used.
Otherwise, `__div__` is used.
If not defined or returns `NotImplemented`,
`other.__rtruediv__(x)` is tried.

`__floordiv__(self, other)`
`x.__floordiv__(other)` <==> `x // other`
If not defined or returns `NotImplemented`,
`other.__rfloordiv__(x)` is tried.

`__mod__(self, other)`
`x.__mod__(other)` <==> `x % other`
If not defined or returns `NotImplemented`, `other.__rmod__(x)` is tried.

`__divmod__(self, other)`
`x.__divmod__(other)` <==> `divmod(x, other)`
Should be equivelant to using `__floordiv__` and `__mod__`. Should not be related to `__truediv__`.
If not defined or returns `NotImplemented`, `other.__rdivmod__(x)` is tried.

`__pow__(self, other [, mod])`
`x.__pow__(other [, mod])` <==> `x ** other [% mod]` <==> pow(x, other [, mod])
If no `mod` argument is accepted, the ternary version of `pow` cannot be used.

If not defined or returns `NotImplemented` and `mod` argument *is not* given, `other.__rpow__(x)` is tried.
If not defined or returns `NotImplemented` and `mod` argument *is* given, no other methods will be tried and a TypeError (?) is raised.

`__lshift__(self, other)`
`x.__lshift__(other)` `<==>` `x << other`
If not defined or returns `NotImplemented`, `other.__rlshift__(x)` is tried.

`__rshift__(self, other)`
`x.__rshift__(other)` `<==>` `x >> other`
If not defined or returns `NotImplemented`, `other.__rrshift__(x)` is tried.

`__and__(self, other)`
`x.__and__(other)` `<==>` `x & other`
If not defined or returns `NotImplemented`, `other.__rand__(x)` is tried.

`__or__(self, other)`
`x.__or__(other)` `<==>` `x | other`
If not defined or returns `NotImplemented`, `other.__ror__(x)` is tried.

`__xor__(self, other)`
`x.__xor__(other)` `<==>` `x ^ other`
If not defined or returns `NotImplemented`, `other.__rxor__(x)` is tried.

2.5.2 Reversed Binary Operations

These methods implement, with swapped commands, the binary operators +, -, , /, //, %, *, <<, >>, &, |, and ^, and the functions `divmod` and `pow`. They are only called if the target's

equivalent regular binary operator is not defined or returns `NotImplemented`.

In other words, if `y + x` is executed, and `y.__add__` is not defined, and y and x are of different types, Python will attempt to call `x.__radd__(y)`.

The method may raise `NotImplemented` if no operation is possible with the supplied with the supplied argument(s).

If not defined, the relevant operation is not possible and a TypeError ('unsupported operand type') is raised.

`__radd__(self, other)`
`x.__radd__(other)` <==> `other + x`
Only called if `other.__add__` is not defined or `other.__add__(x)` returns `NotImplemented`.

`__rsub__(self, other)`
`x.__rsub__(other)` <==> `other - x`
Only called if `other.__sub__` is not defined or `other.__sub__(x)` returns `NotImplemented`.

`__rmul__(self, other)`
`x.__rmul__(other)` <==> `other * x`
Only called if `other.__mul__` is not defined or `other.__mul__(x)` returns `NotImplemented`.

`__rdiv__(self, other)`
`x.__rdiv__(other)` <==> `other / x`
Only called if `other.__div__` is not defined or `other.__div__(x)` returns `NotImplemented`.
If `from __future__ import division` is used, `__rtruediv__` is called instead.

`__rtruediv__(self, other)`
`x.__rtruediv__(other)` <==> `other / x`

Only called if `other.__truediv__` is not defined or `other.__truediv__(x)` returns `NotImplemented`.
Also only called if `from __future__ import division` is used.
Otherwise, `__rdiv__` is called.

`__rfloordiv__(self, other)`
`x.__rfloordiv__(other)` <==> `other // x`
Only called if `other.__floordiv__` is not defined or `other.__floordiv__(x)` returns `NotImplemented`.

`__rmod__(self, other)`
`x.__mod__(other)` <==> `other % x`
Only called if `other.__mod__` is not defined or `other.__mod__(x)` returns `NotImplemented`.

`__rdivmod__(self, other)`
`x.__rdivmod__(other)` <==> `divmod(other, x)`
Should be equivelant to using `__rfloordiv__` and `__rmod__`.
Should not be related to `__rtruediv__`.
Only called if `other.__divmod__` is not defined or `other.__divmod__(x)` returns `NotImplemented`.

`__rpow__(self, other)`
`x.__rpow__(other)` <==> `other ** x` <==> pow(x, other)
Ternary `pow` will not try to call `__rpow__`, as coersion rules would be too complicated. So, `__rpow__` only needs to support binary operation.
Only called if `other.__pow__` is not defined or `other.__pow__(x)` returns `NotImplemented`.

`__rlshift__(self, other)`
`x.__rlshift__(other)` <==> `other << x`
Only called if `other.__lshift__` is not defined or `other.__lshift__(x)` returns `NotImplemented`.

`__rrshift__(self, other)`

`x.__rrshift__(other)` <==> `other >> x`
Only called if `other.__rshift__` is not defined or `other.__rshift__(x)` returns `NotImplemented`..

`__rand__(self, other)`
`x.__rand__(other)` <==> `other & x`
Only called if `other.__and__` is not defined or `other.__and__(x)` returns `NotImplemented`.

`__ror__(self, other)`
`x.__ror__(other)` <==> `other | x`
Only called if `other.__or__` is not defined or `other.__or__(x)` returns `NotImplemented`.

`__rxor__(self, other)`
`x.__rxor__(other)` <==> `other ^ x`
Only called if `other.__xor__` is not defined or `other.__xor__(x)` returns `NotImplemented`.

2.5.3 In-Place Binary Operations

These methods implement the in-place binary operators `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `|=`, and `^=`. Note that there is no in-place equivalent for `divmod`, while `pow` is only available as `**=`.

The method should attempt to modify the instance (*self*) **in place** and return the result. In any case, Python assigns the result as the new value of the instance.

It may raise `NotImplemented` if no operation is possible with the supplied with the supplied argument(s).

If method is not defined or returns `NotImplemented`, the equivalent operation is attempted with the the equivalent regular binary operator. If that is not defined or returns `NotImplemented`, and the two instances are of different types,

the equivalent reflected binary operator method of the `other` instance is called.

If none of the three is defined or don't raise `Not Implemented`, the operation is not possible and a TypeError ('unsupported operand type') is raised.

For example, if x += other is attempted, x.___iadd___(other) is tried first, followed by x.___add___(other) and then other.___radd___(x).

`__iadd__(self, other)`
`x.__iadd__(other) <==> x += other`
If not defined or returns `NotImplemented`, `x.__add__(other)` is tried, followed by `other.__radd__(x)`.

`__isub__(self, other)`
`x.__isub__(other) <==> x -= other`
If not defined or returns `NotImplemented`, `x.__sub__(other)` is tried, followed by `other.__rsub__(x)`.

`__imul__(self, other)`
`x.__imul__(other) <==> x *= other`
If not defined or returns `NotImplemented`, `x.__mul__(other)` is tried, followed by `other.__rmul__(x)`.

`__idiv__(self, other)`
`x.__idiv__(other) <==> x /= other`
If `from __future__ import division` is used, `__itruediv__` is called instead.
If not defined or returns `NotImplemented`, `x.__div__(other)` is tried, followed by `other.__rdiv__(x)`.

`__itruediv__(self, other)`
`x.__itruediv__(other) <==> x /= other`
Only called if `from __future__ import division` is used.
Otherwise, `__idiv__` is used.

If not defined or returns `NotImplemented`, `x.__truediv__(other)` is tried, followed by `other.__rtruediv__(x)`.

`__ifloordiv__(self, other)`
`x.__ifloordiv__(other) <==> x //= other`
If not defined or returns `NotImplemented`, `x.__floordiv__(other)` is tried, followed by `other.__rfloordiv__(x)`.

`__imod__(self, other)`
`x.__imod__(other) <==> x %= other`
If not defined or returns `NotImplemented`, `x.__mod__(other)` is tried, followed by `other.__rmod__(x)`.

`__ipow__(self, other)`
`x.__ipow__(other) <==> x **= other`
If not defined or returns `NotImplemented`, `x.__pow__(other)` is tried, followed by `other.__rpow__(x)`.

`__ilshift__(self, other)`
`x.__ilshift__(other) <==> x <<= other`
If not defined or returns `NotImplemented`, `x.__lshift__(other)` is tried, followed by `other.__rlshift__(x)`.

`__irshift__(self, other)`
`x.__irshift__(other) <==> x >>= other`
If not defined or returns `NotImplemented`, `x.__rshift__(other)` is tried, followed by `other.__rrshift__(x)`.

`__iand__(self, other)`
`x.__iand__(other) <==> x &= other`
If not defined or returns `NotImplemented`, `x.__and__(other)` is tried, followed by `other.__radd__(x)`.

`__ior__(self, other)`
`x.__ior__(other) <==> x |= other`

If not defined or returns `NotImplemented`, `x.__or__(other)` is tried, followed by `other.__ror__(x)`.

`__ixor__(self, other)`
`x.__ixor__(other) <==> x ^= other`
If not defined or returns `NotImplemented`, `x.__xor__(other)` is tried, followed by `other.__rxor__(x)`.

2.5.4 Unary Operations

`__neg__(self)`
`x.__neg__() <==> -x`
If not defined, this construct cannot be used and an `AttributeError` is raised.

`__pos__(self)`
`x.__pos__() <==> +x`
If not defined, this construct cannot be used and an `AttributeError` is raised.

`__invert__(self)`
`x.__invert__() <==> ~x`
If not defined, this construct cannot be used and an `AttributeError` is raised.

`__abs__(self)`
`x.__abs__() <==> abs(x)`
If not defined, the `abs` function cannot be used on this instance and an `AttributeError` is raised.

2.6 Casts

These methods implement conversion to various numeric types. One of these methods is called when the corresponding built-in method is called with the instance. If no conversion is possible, a `ValueError` can be raised (?) (not sure if this is

official, but that's what strings do). If not present, the specified conversion cannot be done and an `AttributeError` is raised.

2.6.1 Real Numbers

In these methods, if the incorrect type is returned, a `TypeError` is raised (e.g., "___float___ returned non-float").

`__float__(self)`
`x.__float__() <==> float(x)`
Should return a floating-point representation of `x`, as a `float` instance.

`__hex__(self)`
`x.__hex__() <==> hex(x)`
Should return a hexadecimal representation of `x`, preceded by "0x", as a **string**.

`__int__(self)`
`x.__int__() <==> int(x)`
Should return a integer representation of `x`, as an `int` instance.
Note: Although the built-in `int` accepts a *base* value, this appears to be only useable if a class inherits from `str` (?) and appears not to involve `str.__int__` (?).

`__long__(self)`
`x.__long__() <==> long(x)`
Should return a long-integer representation of `x`, as a `long` instance.
Note: Although the built-in `long` accepts a *base* value, this appears to be only useable if a class inherits from `str` (?) and appears not to involve `str.__long__` (?).

`__oct__(self)`
`x.__oct__() <==> oct(x)`
Should return a octal representation of `x`, preceded by "0", as a **string**.

2.6.1.1 Slice Indices

`__index__(self)`
`other[x.__index__()] <==> other[x]`
`x.__index()__ <==> operator.index(x)`
Called whenever another object is 'sliced' with the instance, and by the `operator` module function `index`.
Should return an integer representation (an int or a long) of the instance suitable for slicing.
New in version 2.5

2.6.2 Complex Numbers

`__complex__(self)`
`x.__complex__() <==> complex(x)`
`x.__complex__() + y.__float__() <==> complex(x,y)`
Should return a complex number. If another type is returned, Python will attempt to call its ___float___ method and assign the result to the real part of the complex number.
If a second argument (the imaginary part) is given to the built-in `complex` method, Python will call its `__float__` method and add the result to the imaginary part of the complex number.

2.7 Containing Items

2.7.1 Basics

Although not required, it's probably a good idea to implement these if your instances will contain user-accessible items.

`__len__(self)`
`x.__len__() <==> len(x)`
Should (must?) return an integer describing how may items are contained by the instance.
If `__nonzero__` is not defined and `__len__` returns `0`, instance is considered to evaluate to `False`. If neither is present instance will always evaluate to `True`.

If not present, `len` cannot be used and negative slice indexes cannot be used for `__getslice__`, `__setslice__`, and `__delslice__`. An `AttributeError` is raised.

`__contains__(self, item)`
`x.__contains__(item) <==> item in x`
Should return `True` if `item` is contained by the instance, `False` otherwise. Mapping objects should consider only keys, not values.
If not present, Python attempts to look for the item by iterating over all items using `__iter__` and then `__getitem__`, with integer keys starting at zero and ending when `IndexError` is raised. If neither is defined, a `TypeError` (?) is raised.

`__iter__(self)`
`x.__iter__() <==> iter(x)`
Also called when iteration over the instance is requested, such as in a `for` loop.
Should return an iterator suitable for iterating over all the items contained in the instance. Mapping objects, again, should consider only keys.
If not present, Python attempts to call `__getitem__` with integer keys starting at zero and ending when `IndexError` is raised. If `__getitem__` is also not defined, a `TypeError` ('iteration over non-sequence') is raised.

`__reversed__(self)`
`x.__reversed__() <==> reversed(x)`
Should return an iterator suitable for iterating over all the items contained in the instance **in reverse order**.
If not present, Python attempts to call `__getitem__` with indices starting at `__len__ - 1` and ending at 0. In this case, if `__len__` is not defined, an `AttributeError` is raised. If `__getitem__` is not defined, a `TypeError` ('iteration over non-sequence') is raised.

2.7.2. Items and Slices

These methods are called when bracket notation is used.

Python will behave differently depending on the type of value inside of the brackets:

`x[key]`, where `key` is a single value
: Calls `x.__*item__(key)`

`x[start:end]` where `x.__*slice__` **exists**
: Calls `x.__*slice__(cooked_start, cooked_end)` where `start` and `end` are 'cooked' as described below in 'Old-Style Slices'

`x[start:end]` where `x.__*slice__` **does not exist**, or `x[extended_slice]`, where extended slice is any slice more complex than `start:end`
: Calls `x.__*item__` with slice object, `Ellipsis`, or list of these.

2.7.2.1 Items and New-Style Slices

In general, if key is of an inappropriate type, TypeError should be raised. If it is outside the sequence of keys in instance, IndexError should be raised. If instance is a mapping object and key cannot be found, KeyError should be raised. (What if neither of these is true? I don't know.)

`__getitem__(self, key)`
`x.__getitem__(key)` <==> `x[key]`
Should return item(s) referenced by `key`.
Not called if `__setslice__` exists and simple `start:end` slicing is used.
If not present, items cannot be evaluated using bracket notation, and an `AttributeError` is raised.

`__setitem__(self, key, value)`
`x.__setitem__(key, value)` <==> `x[key] = value`
Should set or replace item(s) referenced by `key`. `value` can be a single value or a sequence.

Not called if `__setslice__` exists and simple `start:end` slicing is used. Usage not dependent on presence of `__getitem__`.
If not present, items cannot be assigned using bracket notation, and an `AttributeError` is raised.

`__delitem__(self, key)`
`x.__delitem__(key) <==> del x[key]`
Should delete item(s) represented by `key`. Not dependent on presence of `__getitem__`.
Not called if `__delslice__` exists and simple `start:end` slicing is used. Usage not dependent on presence of `__getitem__`.
If not present, items cannot be deleted using bracket notation, and an `AttributeError` is raised.

2.7.2.2 Old-Style Slices

These methods are **depreciated** but still widely used. Furthermore, for simple slicing Python checks for their existence **first** before calling the `__*item__` methods.

For these methods, Python 'cooks' the indexes before passing them. Negative slice indexes are converted to (usually) positive ones by adding them to the value returned by `__len__`, and so if `__len__` is not defined, negative slice indexes **cannot** be used. Furthermore, an empty start index is replaced by `0`, and an empty end index by `sys.maxint`.

`__getslice__(self, start, end)`
`x.__getslice__(start, end) <==> x[start:end]`
Should return items in the slice represented by `start:end`.
If not present, or if extended slicing is used, a slice object is passed to `__getitem__`.

`__setslice__(self, start, end, sequence)`
`x.__setitem__(start, end, sequence) <==> x[start:end] = sequence`
Should assign sequence to the slice represented by `start:end`.

If not present, or if extended slicing is used, a slice object is passed to `__setitem__`.

`__delslice__(self, key)`
`x.__delslice__(start, end) <==> del x[start:end]`
Should delete items in the slice represented by `start:end`.
If not present, or if extended slicing is used, a slice object is passed to `__delitem__`.

2.8 Attribute Access

As an alternative to *Containing Items*, instance attributes can be directly read, written to, and deleted. Access can be customized using these methods.

`__getattr__(self, attr)`
`x.__getattr__(self, attr) <==> x.attr`
Only called when `attr` isn't found in any of the usual places (it is not an instance attribute or a class attribute, and class is old-style or ___getattribute___ is not defined).
Should return the value of the key `attr` in the relevant mapping object.
If not implemented, the attribute cannot be found and an `AttributeError` is raised.

`__getattribute__(self, attr)`
`x.__getattribute__(self, attr) <==> x.attr`
New-style classes only.
Called whenever attribute assignment is attempted (unlike `__getattr__`).
Should return the value of the key `attr` in the relevant mapping object.
If not implemented, Python will find the attribute as normal (including calling ___getattr___).
If raises `AttributeError`, Python will call ___getattr___, but will *not look anywhere else* for the attribute.

`__setattr__(self, attr, value)`
`x.__setattr__(self, attr, value)` <==> `x.attr = value`
Called whenever attribute assignment is attempted.
Should give `attr` a value of `value` in the relevant mapping object.
If not implemented, Python will assign the attribute to the instance as normal: `self.__dict__[attr] = value` for old-style classes and `object.__setattr__(self, attr, value)` for new-style classes.

`__delattr__(self, attr)`
`x.__delattr__(self, attr)` <==> `del x.attr`
Called whenever attribute deletion is attempted.
Should delete the key `attr` in the relevant mapping object.
If not implemented, Python deletes the attribute as normal: `del x.__dict__[attr]` for old-style classes and `object.__delattr__(self, attr)` for new-style classes. (I think, can anyone verify?)

2.9 Being Contained By Another Object

`__hash__(self)`
`x.__hash__()` <==> `hash(x)`
Called by the built-in function `hash`, and to compare **keys** in dictionary operations.
Should return a 32-bit integer. Objects which evaluate as equal should have the same hash value. Starting in Python 2.5, may instead return a long integer; that object's ___hash___ will be used.
Only should be defined if `__cmp__` is defined; if `__cmp__` or `__eq__` are defined and `__hash__` is not, hashing of any sort will fail and the object **cannot be used as a dictionary key or in a set**.
If none of the three are defined, object **can** be used as a dictionary key or in a set (why?).
Hash value should be immutable (always?).

2.9.1 Descriptors

New-style classes may possess attributes called "descriptors", which are themselves **new-style** class instances. With the following functions, descriptors can change what will happen when the descriptor is accessed, set, or deleted.

Note: to work in this fashion, a descriptor must be possessed by a *class*, not by an *instance*.

For the following examples, assume we have a new-style container class called 'Container' with an instance called 'container'. Also note that `instance` is the container class instance that the descriptor was accessed through, or `None` if the descriptor was accessed through the class itself. `owner` is the actual owner of the descriptor (the container class itself).

`__get__(self, instance, owner)`
`x.__get__(self, container, Container)` <==> `container.x`
`x.__get__(self, None, Container)` <==> `Container.x`
Called when attribute lookup is attempted in a new-style instance **or class**.
Should return an appropriate value representing the instance, or raise `AttributeError` if this is not possible.
If not defined, the entire instance `x` will be returned.
If `AttributeError` is raised without a message, it will be caught and re-raised with the message "type object 'Container' has no attribute 'x'".

`__set__(self, instance, value)`
`x.__set__(self, container, value)` <==> `container.x = value`
Called when attribute assignment is attempted in a new-style instance.
Should somehow store `value`. (what to return?)
If not defined, the attribute `container.x` will be set to `value` instead of the current instance.

`__delete__(self, instance)`
`x.__delete__(self, container)` <==> `del container.x`
Called when attribute deletion is attempted in a new-style instance.
Should somehow delete oneself. (what to return?)
If not defined, the attribute `container.x` will be deleted.

2.10 Pickling

The pickle protocol provides a standard way to serialize and de-serialize objects. These functions customize the pickling of class instances.

For all the gooey details (trust me they're gooey), see PEP 307 -- Extensions to the Pickle Protocol

2.10.1 Serializing and Unserializing Data

These functions are only called if class is **old-style** or `__reduce__` and `__reduce_ex__` are undefined.

`__getstate__(self)`
Called during pickling.
Should return a pickleable value representing the instance's state.
If not defined, and class is **old-style** or `__slots__` is not defined, `self.__dict__` is used.
If not defined and if class is **new-style** and defines `__slots__`, a two-item tuple is returned. This tuple contains `self.__dict__` as its first argument and a dict that maps slot name to slot value as its second.

`__setstate__(self, state)`
Called during un-pickling. `state` is the serialized value -- that returned by `__getstate__` or `__dict__` during pickling.

For **new-style classes** only, **not called if state is a false value**.
Should recreate the original instance using the serialized data.
If not defined, `self.__dict__.update(state)` is tried. If `RuntimeError` is then raised, `self.__setattr__(self, key, value)` is performed for each `(key, value)` pair in `state`. If instead `state` is a two-item tuple as described in `__getstate__` above, something appropriate is done (what exactly?).

2.10.2 Customizing Object Creation upon Unpickling

`__getinitargs__(self)`
Called during pickling, if class is **old-style**.
Should return a tuple of arguments to be passed to `__init__` upon unpickling, if calling `__init__` is desired.
If not defined, ___init___ will not be called upon unpickling.

`__getnewargs__(self)`
Called during pickling if class is **new-style**, **pickling protocol is 2** and `__reduce__` and `__reduce_ex__` are undefined.
Should return a tuple of arguments (besides the class itself) to be passed to `__new__` upon unpickling.
If not defined, an empty tuple is used. Upon unpickling, the new object will be created by calling `__new__(Class)`.

2.10.3 Total Control

These methods work for **new-style** classes only.

`__reduce__(self)`
Called during pickling, if `__reduce_ex__` is not defined.
Overrides almost all aspects of un-pickling procedure.
Should return either a string, naming a global(?) variable whose contents are pickled as normal, or a tuple containing the following items:
* A callable object called to initially create the new instance upon un-pickling
* A tuple of arguments for this callable

* **(Optional)** The object's state, as described in `__getstate__` above.
* **(Optional)** An iterator (not a sequence) with sequential list items. Will be added to the instance upon creation with `append` or `extend`.
* **(Optional)** An iterator with key-value pairs. Will be added to the instance upon creation with `instance[key] = value`.

If not defined, `object.__reduce__` is used and pickling will occur normally. Note: the default `__reduce__` function will not work for protocols 0 or 1 for built-in types and new-style classes implemented in C.

`__reduce_ex__(self, protocol)`
Called during pickling.
Functions the same way as `__reduce__`, with the obvious exception that the pickling protocol is passed.
If not defined, `__reduce__` will be tried. If both are not defined, pickling will occur normally.

2.10.4 One More Thing

`__newobj__(cls, *args)`
This method does nothing special in itself.
In a **new-style** class being reduced under **protocol 2**, `__reduce__` or `__reduce_ex__` may specify `__newobj__` as the first argument in the returned tuple -- the callable to create the new instance upon unpickling. In this case something special is done to shrink the size of the pickle: the method is assumed to have the syntax `def __newobj__(cls, *args): return cls.__new__(cls, *args)`. The pickler does not store `__newobj__` as the callable, but instead stores an opcode to just call `cls.__new__` upon unpickling.
Protocols 0 and 1 fall back on the 'normal' approach, which is to store `__newobj__` as the callable; for this reason `__newobj__` should actually have the assumed syntax.

2.11 Copying

___copy___ ___deepcopy___

2.12 "With" Statements

___enter___ ___exit___

2.13 Really Complicated

___new___ ___coerce___

3 Class Methods

___subclasses___, see http://lucumr.pocoo.org/blogarchive/python-plugin-system

4 Class & Instance Properties

___dict___ & vars() ___class___ ___metaclass___ ___bases___ ___name___ ___slots___ ___weakref___

5 Class & Function Properties

___doc___

6 File Properties

___file___

7 Builtin Functions

___import___

8 Module Properties

___builtins___ (nonstandard, see http://docs.python.org/lib/module-builtin.html) ___all___

9 Modules

___builtin___ ___main___ (represents scope of main program, http://docs.python.org/lib/module-main.html) ___future___

10 Other Things I've Seen Around

___requires___ ___traceback_hide___ ___debug___

11 Talk about in Intro/Conclusion?

- dir()
- object
- property/fget/fset/fdel
