递归和迭代

所谓递归，简而言之就是应用程序自身调用自身，以实现层次数据结构的查询和访问。 递归的使用可以使代码更简洁清晰，可读性更好（对于初学者到不见得），但由于递归需要系统堆栈，所以空间消耗要比非递归代码要大很多，而且，如果递归深度太大，可能系统资源会不够用。  
往往有这样的观点：能不用递归就不用递归，递归都可以用迭代来代替。  
诚然，在理论上，递归和迭代在时间复杂度方面是等价的（在不考虑函数调用开销和函数调用产生的堆栈开销），但实际上递归确实效率比迭代低，既然这样，递归没有任何优势，那么是不是就，没有使用递归的必要了，那递归的存在有何意义呢？ 万物的存在是需要时间的检验的，递归没有被历史所埋没，即有存在的理由。  
从理论上说，所有的递归函数都可以转换为迭代函数，反之亦然，然而代价通常都是比较高的。但从算法结构来说，递归声明的结构并不总能够转换为迭代结构，原因在于结构的引申本身属于递归的概念，用迭代的方法在设计初期根本无法实现，这就像动多态的东西并不总是可以用静多态的方法实现一样。  
这也是为什么在结构设计时，通常采用递归的方式而不是采用迭代的方式的原因，一个极典型的例子类似于链表，使用递归定义及其简单，但对于内存定义(数组方式)其定义及调用处理说明就变得很晦涩，尤其是在遇到环链、图、网格等问题时，使用迭代方式从描述到实现上都变得不现实。 因而可以从实际上说，所有的迭代可以转换为递归，但递归不一定可以转换为迭代。  
采用递归算法需要的前提条件是，当且仅当一个存在预期的收敛时，才可采用递归算法，否则，就不能使用递归算法。  
递归其实是方便了程序员难为了机器，递归可以通过数学公式很方便的转换为程序。其优点就是易理解，容易编程。但递归是用栈机制实现的，每深入一层，都要占去一块栈数据区域，对嵌套层数深的一些算法，递归会力不从心，空间上会以内存崩溃而告终，而且递归也带来了大量的函数调用，这也有许多额外的时间开销。所以在深度大时，它的时空性就不好了。  
而迭代虽然效率高，运行时间只因循环次数增加而增加，没什么额外开销，空间上也没有什么增加，但缺点就是不容易理解，编写复杂问题时困难。  
因而，“能不用递归就不用递归，递归都可以用迭代来代替”这样的理解，还是辩证的来看待，不可一棍子打死。

**A Guide to Python's Magic Methods**

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

## [Introduction](http://www.rafekettler.com/magicmethods.html#intro)

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\_\_).

## [Construction and Initialization](http://www.rafekettler.com/magicmethods.html#construction)

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](http://www.python.org/download/releases/2.2/descrintro/#__new__).

\_\_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

### [Comparison magic methods](http://www.rafekettler.com/magicmethods.html#comparisons)

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 Words (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 modulefunctools 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.

## [Representing your Classes](http://www.rafekettler.com/magicmethods.html#representations)

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.