Python: Magic Methods

An introduction to python's magic methods and more

Outline

1. We'll cover some common uses of magic methods and talk about python's objects
2. Lyrics Engine Features:
3. Introduction to Pickle
4. Suggestions for next time

Magic Methods

First of all, it's hard to beat the following resource: http://www.rafekettler.com/magicmethods.pdf

We'll go over Rafe Kettler's document and answer any questions that you may have

What are magic methods?

Unfortunately the magic methods aren't documented very well, and I always have to look them up.

Fortunately the PDF document linked on the previous page is incredibly verbose

Magic Methods

A magic method is a function (method) on a python object that has a special name.

The python interpreter knows to look for these functions and will attempt to use them when appropriate

Often, a magic method is all that separates one kind of object (executable) from another (list-based)

Everything in python is an object

One of the most powerful features of python is that functions, classes, and even integers are all objects.

We can modify them in runtime to our liking, something that is powerful but can also allow you to shoot yourself in the foot.

Open Save Undo

*magic_methods.py

```
'''
    Some examples of magic methods

    Blaine Booher / Ken Hill 2012
'''

# Here's a basic class
class computer():
    def __init__(self, platform, memory):
        '''
            This is our first magic method...
            __init__ is called every time ani nstance of this class
            is created by the interpreter
        '''
        self.platform=platform
        self.memory=memory

    def __str__(self):
        '''
            Another magic method. This is called when we wish to
            print this class. Simply return a string that represents
            a pretty notation for printing
        '''
        return "computer object; platform: %s" % self.platform

    def __repr__(self):
        '''
            Very similar to __str__(). If __str__() does not exist,
            python treats any request to __str__() as a request to
            __repr__().

            rule of thumb: __repr__() is for developers, __str__()
            is for customers.

            Some developers try to make repr() return something that
            could be evaluated into an object, like this:
                computer(platform, memory)

            So let's do that.
```

Examples

```
$ python
>>> import magic_methods
>>> from magic_methods import *
>>> pc = computer('x86', 512)
>>> print pc
computer object; platform: x86
>>> repr(pc)
"computer(platform='x86', memory=512)"
>>> eval(repr(pc))
computer(platform='x86', memory=512)
```

__cmp__(): compare (==, >, <)

```
    def __cmp__(self, other):
        '''
            Compare this computer to another instance of a computer.
            ex: computer1 > computer2

            We return -1, 0, or 1:
                -1 if this instance is "less than" the other instance
                0  if this instance is "equivalent" to the other instance
                1  if this instance is "greater than" the other instance

            For this example, let's compare RAM (self.memory)
        '''
        if self.memory == other.memory:
            return 0 # they're equal!
        elif self.memory < other.memory:
            return -1 # we're smaller than their memory
        else:
            return 1 # we're greater than the other memory

if __name__ == "__main__":
    '''
        Technically another magic method. If this file is called
        directly, the __name__ variable will contain the value
        '__main__', otherwise it will be empty. This allows us
        to distinguish between a library and an application
    '''
    mac = computer('mac', 512)
    pc = computer('x86', 1024)
    print mac # call __str__()
    print pc  # call __str__()
    print "mac == pc: ", mac == pc # call __cmp__
    print "mac > pc:  ", mac > pc  # call __cmp__
    print "mac < pc:  ", mac < pc  # call __cmp__
```

__cmp__ continued

We just saw an example of __cmp__, which allows us to use ==, <, and > to compare to instances of the same object.

34$ python magic_methods.py
computer object; platform: mac; memory: 512
computer object; platform: x86; memory: 1024
mac == pc: False
mac > pc: False
mac < pc: True

Now let's look at __index__

__getitem__: use object[x]

__getitem__(self, number) will let us get information from the object through an index. The index is arbitrary and can be any value. We can use it to grab from an internal list, or do something crazy like always return 0, or return some other object, or return the number after a math operation...

__getitem__

```
class computer():
    def __init__(self, platform, memory, peripherals=None):
        '''
            This is our first magic method...
            __init__ is called every time ani nstance of this class
            is created by the interpreter

            platform is a string, like 'x86'
            memory is an integer, like 512
            peripherals is a list, or nothing. like ['mouse', 'keyboard', 'hard drive']
        '''
        self.platform=platform
        self.memory=memory
        self.peripherals=peripherals

    def __getitem__(self, number):
        '''
            Magic method for grabbing data from this object, like obj[x]

            Let's use the perhipherals list.
            computer_obj[2] should return the third (index 2) peripheral in our
            internal peripheral list.
        '''
        if self.peripherals == None:
            return ''
        elif len(self.peripherals) < number:
            return ''
        else:
            return self.peripherals[number]
```

```
newpc = computer('x86', 1024, ['mouse', 'keyboard', 'monitor', 'bluetooth dongle', 'usb flash drive'])
print newpc[2]
print newpc[3]
print newpc[10] == '' # should return blank string
```

__getitem__ result

```
[blaine@macbook:/data/cliftonlabs/ken_hill/ppt/magic Tue May 29]
37$ python magic_methods.py
computer object; platform: mac; memory: 512
computer object; platform: x86; memory: 1024
mac == pc:  False
mac > pc:   False
mac < pc:   True
monitor
bluetooth dongle
True
```

__getitem__ magic

Just for fun, whose to say we can't have some fun? This prints out the 10th power of the index we pass in. Output: 10000000000

```
class fun():
    def __getitem__(self, number):
        return number**10
print fun()[10]
```

__setitem__

the compliment of __getitem__ is __setitem__, allows us to set things:
computer[5] = 'computer speakers'

We also must decide what to do if the index is larger than our internal peripheral list. I'm choosing to accomodate for it by extending the list with empty values.

```
        self.memory=memory
        self.peripherals=list(peripherals) # force to list since we now implement __setitem__

    def __setitem__(self, number, value):
        '''
            Magic method for saving data based on index like obj[x] = 'hello there'
        '''
        try:
            self.peripherals[number] = value
        except IndexError:
            # called when we're trying to set a value out of range...
            # we can either handle it by increasing the number of values,
            # or just notify the user that we can't make it happen.
            # i choose to make it work.
            n = len(self.peripherals)
            diff = number - n + 1 # how many slots to add to the list
            for i in range(diff):
                self.peripherals.append('empty slot')
            print len(self.peripherals)
            self.peripherals[number] = value
```

```
# test __setitem__
newpc[0]='headphones'
print newpc[0]
newpc[10]='zip drive'
print newpc[9], newpc[10]
```

output:
headphones
11
empty slot zip drive

__iter__: handling "for x in obj"

```
def __iter__(self):
    '''
        Magic Method: for x in obj: print x
        We can use our peripherals list. We also introduce 'yield'

        Yield is just like "return" except that the state of the function is saved
        and the next time the routine is called (aka the next iteration in the for loop)
        and the iteration is continued from where it left off.
    '''
    for p in self.peripherals:
        yield p
```

```
# test __iter__
# enumerate simply returns an incremented number for every iteration. it's cleaner than
# using k=0, k++ for every iteration.
for i, item in enumerate(newpc):
    print i, item
```

Output:
0 headphones
1 keyboard
2 monitor
3 bluetooth dongle
4 usb flash drive
5 empty slot
6 empty slot
7 empty slot
8 empty slot
9 empty slot
10 zip drive

__contains__

__contains__ lets us use the "in" keyword.

For example:

print "mouse" in newpc

__contains__

```
def __contains__(self, name):
    '''
        Magic method for "does object contain x"

        Since we're using our self.peripherals list, simply see if the
        string object is a peripheral. For more complex classes, you
        could do more interesting comparisons (like going to a database
        or something similar)
    '''
    return name in self.peripherals
```

```
# test __contains__
print "cd-rom in pc: ", "cd-rom" in newpc
newpc[6] = "cd-rom"
print "just added it... now is cd-rom in newpc: ", "cd-rom" in newpc
```

Output:

cd-rom in pc: False
just added it... now is cd-rom in newpc:
True

__call__: Make something executable

There are many other magic functions. But here's another one that makes a class instance "callable".

In essence, a function is just an object with __call__ defined.

o = myObject() # call myObject.__init__()
o() # call o.__call__()

```
def __call__(self, memory):
    '''
        Magic method for turning an object into something callable

        For this, let's take in number and just add it to our memory, then return it.
        Do something special if we're passed in a string.

        see the example for something fun using map()
    '''
    if isinstance(memory, int):
        return self.memory + memory
    else:
        return str(self.memory) + "_" + str(memory)
```

```
# test __call__
print newpc(50)
# remember, map calls the function in parameter 1 for each object in parameter 2)
print map(newpc, (100, 1000, 5555))
# OK now we're getting funky.
# for each object in param2 (newpc: so each object is going to be an element of
# peripherals, probably because map will call "for obj in param2", meaning it
# will get a list of the peripherals from __iter__
# then newpc's __call__ routine will add each number to the string (concatonation)
# due to the behavior of the __call__ routine.
print map(newpc, newpc)
```

Output:
1074
[1124, 2024, 6579]
['1024_headphones', '1024_keyboard', '1024_monitor', '1024_bluetooth dongle', '1024_usb flash drive', '1024_empty slot', '1024_cd-rom', '1024_empty slot', '1024_empty slot', '1024_empty slot', '1024_zip drive']

Pickle

Pickle is a cheap and easy way to store data for future retrieval. It isn't good for transferring data between python versions, but it's perfect for something like a caching system.

It's built into python, and has a C version: cPickle that is equivalent to the pure python "pickle", but is more fast on systems that support it.

```
 1 from magic_methods import computer
 2 import cPickle as pickle # good to keep pickle as the name for less confusion and compatibility
 3
 4 # make a simple object
 5 obj = {'name': 'blaine',
 6        'city': 'cincinnati',
 7        'favorite food': 'pizza'}
 8
 9 # save it for later
10 pickle.dump(obj, open("blaine.pickle", "w"))
11
12 # then we can load it up at a later time quickly
13 loaded = pickle.load(open("blaine.pickle")) # open() is 'read' by default
14
15 # we can do this with classes, too, but we have to be careful that the
16 # class is available in the namespace that we load into. In other words,
17 # pickle will dump the instance but not the definition of the class.
18
19 comp = computer("Gameboy", 16, ['buttons', 'screen', 'speakers', 'battery', 'plastic case'])
20 print comp
21
22 # save it for later
23 print 'saving to computer.pickle'
24 pickle.dump(comp, open("computer.pickle", "w"))
25
26 # delete it
27 print 'deleting computer instance'
28 del comp
29
30 # load it back up!
31 comp = pickle.load(open("computer.pickle"))
32 print comp
```

Let's add features to the lyric engine

I have some ideas for the lyric engine.

1. Make the lyric engine into a class
2. Use pickle to dump the lyrics to a file for safe keeping that can be retrieved later
3. When asking for a new lyric, first see if a pickle file exists before going out to the internet
4. Create a function that lists all available cached lyrics

Lyrics: Making it into a class & dumping it to a pickle file

class lyric():

1. Instantiate it with a url that is stored into self.url
2. Create a getlyrics(self) routine that gets the lyrics, saves to self.lyrics in the class instance, and then returns it
3. Create a dumplyrics(self) routine that dumps self.lyrics to a pickle file. the name is based on self.url

Caching

Then, add the ability to getlyric(self) to check for a pickle file first, and if it exists: load it and return the contents instead of going to the internet to retrieve it.

List all lyrics

Create a function that displays which lyrics are cached. Hint:

```
import glob
files = glob.glob("lyrics/*.pickle")
for f in files...
```

Next up...

Once the lyrics engine is up and ready to go, I'd like to do the following:

1. Upgrade from pickle to SQL database (if you're so included, it's actually pretty easy)
2. Wrap it in a micro-web framework (flask) to display on your browser
3. Add an html input box so you can get lyrics from the web interface