
Templating with Cheetah

(2005-12-27) - Contributed by Peyton McCullough

Templating makes it easier to handle dynamic content on web pages. One of the better templating frameworks for Python is Cheetah. Keep reading to find out more.

Introduction

The creation of dynamic web content is made much easier by using templates. These templates include substitution variables that are replaced with the proper value when the content is delivered to the user. There are a number of languages and frameworks that offer templating to developers, but Cheetah is one of Python's frontrunners. It allows templates to be made and then converted into Python objects for use within Python applications, and it works with pretty much every Python framework. For example, Cheetah could start with a template that looks like this:

```
$header
Hello, $name. Your last visit was on $date.
$footer
```

Cheetah would then compile it into a module, and \$header, \$name, \$date and \$footer could be substituted for just about anything, complete with any bells and whistles. In this article, we'll examine Cheetah and what it has to offer to Python developers looking to simplify dynamic content generation.

Installing Cheetah

Cheetah may be obtained from SourceForge:

http://sourceforge.net/project/showfiles.php?group_id=28961

Simply unarchive Cheetah and install it:

```
python install setup.py
```

Windows users will likely want to take a few extra steps when installing Cheetah. The first is to rename the file cheetah since it does not come with any extension to identify it as a Python file. It's located in the Scripts directory of your Python installation. Simply rename it cheetah.py, and if you are dying to lose the extension, create a file in the same directory called cheetah.bat:

```
cheetah.py %*
```

Next, it is recommended that you obtain a file named `_namemapper.pyd` from SourceForge. It speeds up Cheetah on Windows:

http://prdownloads.sourceforge.net/cheetahtemplate/_namemapper.pyd?download

Place the file in the Lib/site-packages/Cheetah directory of your Python installation.

You'll probably want to add Cheetah to your Path environmental variable, too. If you've never done anything like this before (and since this article is intended to be useful to people of various levels of experience), simply go to the Properties of My Computer, click the Advanced tab, and then click the Environmental Variables button. Then, add a semicolon to Path followed by the path to cheetah's location. For example:

;C:\Python24\Scripts

You'll want to make sure Python is there, too.

```
{mospagebreak title=Basic Templating}
```

Let's start off with a basic template with which to test out Cheetah. A simple counter will work fine. Create a file named counter.tpl:

Hello, This page has been viewed \$counter times.

In the above example, we will obviously want to substitute the \$counter variable for the number of times the particular page has been viewed. It's pretty easy to turn our template into a real, working page in just a few lines of Python. We'll use a text file to store the count for simplicity's sake:

```
import os.path
from Cheetah.Template import Template

# Update the count
if os.path.exists ( 'count.txt' ):
    count = int ( file ( 'count.txt' ).read() )
    count = count + 1
else:
    count = 1
file ( 'count.txt', 'r' ).write ( str ( count ) )

print Template ( file = 'counter.tpl', searchList =
[{'counter': str ( count ) }])
```

After updating the count, we simply pass the file we wish to parse and a dictionary containing the replacement value. Cheetah then does the rest and returns the result, which is printed.

It's also possible to pass objects to Cheetah. Take a look at this template, contact.tpl, which display's a particular person's contact information:

```
Name: $profile.name<br />
E-Mail: $profile.email<br />
Phone: $profile.phone<br />
MSN: $profile.msn<br />
AIM: $profile.aim<br />
ICQ: $profile.icq<br />
YIM: $profile.yim
```

All we need to do is create an object with the above properties and pass it under the profile key:

```
from Cheetah.Template import Template
```

```
class Profile:
    def __init__ ( self, name, email, phone, msn, aim, icq, yim ):
        self.name = name
```

```
self.email = email
self.phone = phone
self.msn = msn
self.aim = aim
self.icq = icq
self.yim = yim
```

```
johnDoe = Profile ( 'John Doe', 'jdoe@google.com', '(555) 555-
5555', 'jdoe@google.com', 'JDoe', '0123456789', 'JDoe' )
print Template ( file = 'contact.tmpl', searchList =
[{'profile': johnDoe }])
```

Cheetah then substitutes the appropriate attributes.

```
{mospagebreak title=Compiling Templates}
```

Compiling a template converts it into Python code, which eliminates the need for Cheetah to go through a template and replace everything. This means that Cheetah can work faster. Compiling a template is very simple, and all it involves is a call to Cheetah. Take a look at this template, quote.tmpl, which provides a placeholder for a random quotation:

```
<center>
$quotation<br />
-- $speaker
</center>
```

To compile this, we pass a few arguments to cheetah:

```
cheetah compile quote.tmpl
```

Cheetah then generates the file quote.py, which contains the class quote. The class can be imported into a Python script and then used similarly to the Template class:

```
import random
from quote import quote
```

```
# Define a list of quotations
quotations = [ [ 'It is easier to find people fit to govern
themselves than people fit to govern others.', 'Lord Acton' ],\
               [ 'A house divided against itself cannot stand.',
'Abraham Lincoln' ],\
               [ 'Before anything else, preparation is the key to
success.', 'Alexander Graham Bell' ] ]
```

```
# Pick a random quotation
quotation = random.choice ( quotations )
```

```
# Print the product
print quotation ( searchList = [{ 'quotation': quotation [0],
'author': quotation [1] }])
```

Adding Logic

Instead of having a Python script control all the logic for a certain template, it's possible to slip a bit of simple logic into a template. Cheetah makes this possible through special directives. For example, say that you want to display a certain message if one condition is met and a certain message if another condition is met. The `#if` directive can be used for this task, rather than doing work in Python. Take a look at `greeting.tmpl`, which displays different greetings for different times of day:

```
#if $hour < 12
Good morning!
#else if $hour >= 12 and $hour <=18
Good afternoon!
#else
Good evening!
#end if
```

All we need to do is pass the hour of the day to the script, and it will display the appropriate message:

```
import time
from Cheetah.Template import Template

hour = time.localtime() [3]
print Template ( file = 'greeting.tmpl', searchList = [{ 'hour':
hour }])
```

The `#unless` directive is similar to the `#if` directive, but it returns the opposite of what the expression returns. If the expression is true, then the contained code is not executed. If the expression is false, then the contained code executes. Take this template for example:

```
#unless $a
< Message >
#end unless
```

Unless `$a` returns a true value, "`< Message >`" is displayed to the user. If `$a` does return a true value, then the message is never displayed.

Cheetah also allows for loops to be used within templates. The `#for` directive acts as a for loop, as in `listPresidents.tmpl`:

```
Presidents:
#for $name in $names:
<br />$name
#end for
```

To see the template in action, we need to pass a list to take the place of the `$names` variable:

```
from Cheetah.Template import Template

presidents = [ 'George Washington', 'John Adams', 'Thomas
Jefferson', 'James Madison', 'James Monroe' ]

print Template ( file = 'listPresidents.tmpl', searchList =
[{ 'names': presidents }])
```

The `#repeat` directive simply repeats something a specified amount of times. For example, say I wanted to make a primitive horizontal bar graph using “|” characters. I could use the `#repeat` directive to do this in `graph.tmpl`:

```
#repeat $x
|
#end repeat
```

We can now substitute `$x` for the length of the graph:

```
from Cheetah.Template import Template
```

```
# Set the length of the bar graph
x = 10
```

```
print Template ( file = 'graph.tmpl', searchList = [{ 'x': x }] )
```

If you run the above script, however, you will see that each “|” character is put on a new line. This can be fixed using the `#slurp` directive:

```
#repeat $x
| #slurp
#end repeat
```

Lastly, the `#while` directive acts as a while loop:

```
#set $x = 1
#while $x <= 10
    $x
    #set $x = $x + 1
#end while
```

Notice that the above template uses the `#set` directive. This simply sets the value of a variable.

Cheetah also contains `#break` and `#continue` directives for use in loops. This template will not display “4” or anything past “5”:

```
#set $x = 1
#while $x <= 10
    #if $x == 4
        #continue
    #end if
    $x
    #if $x == 5
        #break
    #end if
    #set $x = $x + 1
#end while
```

```
{mospagebreak title=More Logic}
```

Functions can be declared in Cheetah templates, too, using the `#def` directive:

```
#def number($num)
Number: $num
#end def
#set $x = 1
#while $x <= 5
$number($x)
#set $x = $x + 1
#end while
```

Cheetah also contains `#return` and `#pass` directives for use in functions that behave just like their Python equivalents.

It's possible to import Python modules in a template using the `#import` directive:

```
#import time
$time.strftime('%m/%d/%Y')
```

A `#from` directive also exists that is identical to Python's `from`.

You can also import other compiled templates, and it's possible to subclass both compiled templates and Python modules as well. Here, we define `parent.tmpl`:

```
One: $one
Two: $two
Three: $three
```

Compile it, and then we are able to subclass it and define methods for `$one`, `$two` and `$three`:

```
#from parent import parent
#extends parent
#def one
1
#end def
#def two
2
#end def
#def three
3
#end def
```

Of course, just because you can play around with logic in Cheetah templates does not mean that you should. Combining presentation and logic wherever possible defeats the purpose of templating, which is to separate the two elements of dynamic content. Too many directives within a template make the template difficult to read, anyway. In my experience, a good rule of thumb is to use template logic where actual layout elements will vary as a result. Otherwise, if it's just a simple message or something that will vary, Python does the job a lot better. More complex layout elements are probably best left to Python, too.

Conclusion

Cheetah offers templating that can be used with just about any Python framework. I should also note that it doesn't have to be used with web pages. It can also be used for XML documents, e-mails and just about any other format. Beyond basic templating, it also allows for logic to be placed within templates, such as conditional statements, loops and

functions. It's an easy-to-use tool that's worth taking into consideration for projects that require dynamic content.