

Building and filling out templates with Python and Cheetah

Generate HTML, XML, plain text, and more with this powerful templating engine for Python

Level: Introductory

Leonard Richardson (leonardr@segfault.org), Software Engineer, CollabNet

09 Aug 2005

After reading this article, you'll be able to generate any kind of text-based content with Python scripts and Cheetah templates. Cheetah templates are easy to understand and maintain, and they help you separate the static parts of a document from the dynamic parts.

A plethora of templating systems

"[Connecting databases to Python with SQLAlchemy](#)" mentioned the wide variety of open source object-relational mapping libraries for Python. Python programmers like to do things their own way, which leads to a lot of duplication of effort. Out of all that effort, though, often comes one package that's good enough for just about everyone.

The same pattern has played out for templating systems: ways of representing static text as forms to be filled out, so that dynamic elements can be plugged in later. The official Python Wiki links to nearly 20 templating systems, and those are just the major ones. What's more, Python comes packaged with several basic templating systems that will work in simple cases.

This article describes the problems that templating systems solve. It also introduces Cheetah, the best Python templating system yet devised. The article assumes you have a basic knowledge of Python, but no knowledge of templating systems or what they're used for.

Basic templating concepts

Suppose you're writing a Web application for an online store. You need classes to represent aspects of the store and the purchasing process: items of stock, customers and the orders they place, and so on. The instances of these classes probably correspond to rows in a database, and they're used to represent the state of your store and your customers. In order for you and your customers to use the application, these objects need to be used to generate human-readable HTML pages and e-mail messages, like the following e-mail:

Listing 1. Hello code

```
Hel l o, Leonard.

Your order (#98765) has shi pped:
  Wi dget, green: 50 uni t(s)
  Wi dget, blue: 1 uni t(s)

Your tracking number is 1234567890AB.
```

This e-mail body has a static portion and a dynamic portion. The static portion is the abstract *form* of the message, which is always the same. It can be described in pseudocode like this:

Listing 2. Abstract form of the template

```
Hello, [customer's first name].

Your order ([order's ID]) has shipped:
[list items in the order, with quantity, appending "unit(s)" to each]

Your tracking number is [order's tracking number].
```

The dynamic portion of the e-mail body is all the information pertaining to a specific order from a specific customer. This portion is represented by the application objects and the data members associated with them: a customer, an order, and the list of items and quantities for that order.

The sections that follow use dummy `User` and `Order` objects to simulate the dynamic portion of the template. In a real application, these objects would probably be obtained from a database.

A templating system lets you represent the static portion of a text as a template definition, stored and managed separately from the dynamic portion. The static portion can be combined with different dynamic values to produce custom text.

The sections that follow explain how Python's built-in facilities fall short, introduce the use of Cheetah as an alternative, and show you how to implement this e-mail message as a Cheetah template.

Why use a templating system?

Without a templating system, you'd use Python code to generate pieces of text like the example e-mail message. You'd write logic that appends to a list of strings or writes to a file-like object. For instance, the following code generates the e-mail described above:

Listing 3. Generate the e-mail

```
from DummyObjects import dummyUser, dummyOrder
l = []
l.append(' Hello, ')
l.append(dummyUser.firstName)
l.append(' \n\nYour order (#')
l.append(str(dummyOrder.id))
l.append(') has shipped: \n')
for purchased, quantity in dummyOrder.purchased.items():
    l.append(' ')
    l.append(purchased.name)
    l.append(': ')
    l.append(str(quantity))
    l.append(' unit(s)\n')
l.append('\nYour tracking number is ')
l.append(dummyOrder.trackingNumber)
l.append('.')
print ''.join(l)
```

Unfortunately, this code isn't as nice as its output. It's hard to see the structure of the e-mail message by looking at the code. A lot of code is repeated (for instance, the many calls to the list `append()` method), which creates room for error. Finally, the UI designer on your team would probably rather edit something more like the abstract template described above than edit this Python code. Developers turn to template systems for all these reasons.

Are Python's built-in templating systems enough?

Python comes with a few built-in templating systems, which work well in simple cases. For a long time, Python had simple template systems that understood formats reminiscent of C's `printf()` string formatting:

Listing 4. Python's built-in template systems

```
from DummyObjects import dummyUser, dummyOrder
print 'Hello, %s.\n\nYour order (%#d) has shipped:' % (dummyUser.firstName,
                                                    dummyOrder.id)

print 'Hello, %(firstName)s.\n\nYour order (%%(orderId)d) has shipped:' % \
{'firstName' : dummyUser.firstName, 'orderId' : dummyOrder.id}
```

Python V2.4 introduced a template system with a more modern-looking format. Variable names are designated as such by prefixing them with a dollar sign (\$); this is similar to Perl, PHP, most shell languages, and Cheetah:

Listing 5. Python V2.4's built-in template system

```
from string import Template
from DummyObjects import dummyUser, dummyOrder
t = Template('Hello, $firstName.\n\nYour order ($orderId) has shipped:')
t.substitute({'firstName' : dummyUser.firstName, 'orderId' : dummyOrder.id})
```

These template systems share two major shortcomings.

1. Their template definitions can't call any method or access any members of the `dummyUser` and `dummyOrder` objects. You can't put `dummyUser.firstName` in the template definition. You have to put it into the map that gets applied to the template definition. All the dynamic information to be inserted into the static template definition must first be broken down into basic Python data types.
2. These template systems have no flow control -- no loops or conditionals. The previous examples stop right before they have to iterate over the items in the order, and for good reason: The templating systems they use can't do that iteration within a template definition. You need to write Python code to iterate over the items in the order, concatenating multiple strings together (possibly using intermediate templates) and providing the end result to the template system as a single string called something like `itemsOrdered`. The loop itself is part of the static portion of the e-mail body -- it works the same way no matter what user and order are being processed -- but there's no way to factor it out into the static template definition.

Most of the add-on template systems available for Python intend to address these two shortcomings. The best of this crowded field is Cheetah.

Getting started with Cheetah

Cheetah has a long pedigree. It's inspired by a Java™ templating system called Velocity, an improved version of the Webmacro templating system, which is itself an attempt to improve on JavaServer Pages. Cheetah provides a simple language for defining templates that provides basic flow control and object access constructs. It borrows its basic template syntax from Velocity, but adds features that give Cheetah templates access to the convenient constructs of Python.

Here's some Cheetah code that interprets the "easy" portion of the template definition -- the part with no flow control, which Python's built-in templating systems can handle:

Listing 6. A first Cheetah example

```
from Cheetah.Template import Template
from DummyObjects import dummyUser, dummyOrder
definition = """Hello, $user.firstName.

Your order (#$order.id) has shipped: """
print Template(definition, searchList=[{'user' : dummyUser,
                                       'order' : dummyOrder}])
```

The `definition` string contains the template definition (the static portion of the e-mail), which can make references to outside variables (the dynamic portion). The `Template` constructor is used here to bind the template definition to a `searchList` of

namespaces: ways of looking up objects corresponding to the variables used in the definition. For instance, `$user` in the template definition gets mapped here to the `dummyUser` variable. You can also run the `Template` constructor ahead of time and set its `searchList` member when you're ready to interpret the template with specific objects.

For Velocity users

A *namespace* in Cheetah is what Velocity calls a *context*.

You should already see the advantages of Cheetah over Python's built-in templating systems. The dynamic portions of the message (the `dummyUser` and `dummyOrder` objects) are the only things left out of the template definition. Everything else, including which members of the objects to access, doesn't change between messages and so goes into the template definition.

Suppose you needed to change the e-mail template so it printed the user's full name instead of first name. Assuming the `dummyUser` object already provides that information (for instance, with a `getFullName()` method or a `fullname` member), you could make this change solely by changing the template definition. With the built-in Python templating systems, you'd have to change the Python code.

Compiling template definitions into Python classes

What happens when you turn a template definition into a `Template` object? Cheetah generates a custom Python class that implements code for merging the template definition with dynamic data. You can see this for yourself by saving a template definition to a file and running the `cheetah compile` command on it. Here's `Greeting.tmpl`, a template file containing the same Cheetah template used previously:

```
Hello, $user.firstName.

Your order (#$order.id) has shipped:
```

Running `cheetah compile Greeting.tmpl` on this file generates a module called `Greeting.py`. This class contains a class called `Greeting` that implements code very similar to the manually written code featured at the beginning of this article:

Manual code

```
l.append('Hello, ')
l.append(
    dummyUser.firstName)

l.append('.\n\nYour
```

Cheetah-generated code

```
write('Hello, ')
write(filter(VFFSL(SL,
    "user.firstName",
    True), rawExpr=
    '$user.firstName'))
write('.\n\nYour
    order (#')
```

```

order (#')
l.append(str(
    dummyOrder.id))
l.append(') has
    shipped:\n')

write(filter(VFFSL(SL,
    "order.id",True),
    rawExpr=
    '$order.id'))
write(') has
    shipped:')

```

You can then use the generated Greeting class in Python code, just as if you had defined a generic Template with the contents of Greeting.tmpl:

```

from Greeting import Greeting
print Greeting(searchList=[{'user' : dummyUser, 'order' : dummyOrder}])

```

Because Cheetah can compile template files to Python code, you can do all the template parsing up front and get the benefits of compiled code when you fill out the templates with dynamic data.

Flow control: the #for directive

Cheetah is better than Python's built-in templating systems at generating the first part of the sample template. It also handles the rest of the template, which the built-in systems can't handle at all. In addition to variable references, Cheetah template definitions can contain directives to the Cheetah interpreter, including the #for directive that sets up loops:

Listing 7. Using the #for directive to iterate over a list

```

defini ti on = """Hello, $user.fi rstName.

Your order (#$order.id) has shipped:
#for $purchased, $quantity in $order.purchased.items():
    $purchased.name: $quantity uni t(s)
#end for

Your tracking number is $order.trackingNumber."""
print Template(defini ti on, searchLi st=[{'user' : dummyUser,
                                          'order' : dummyOrder}])

```

The code is exactly the same as before, but the template definition is different. The #for directive starts a loop, and the #end for directive ends it. Because it can be used to generate text where whitespace is significant (like an e-mail message), Cheetah can't use whitespace as a flow control mechanism the way Python does -- thus, the #end directives.

The #for iteration works like a Python iteration using Python's for keyword. This iteration works exactly like the hand-written Python iteration shown above:

```

for purchased, quantity in order.items():
    l.append(purchased.name)
    ...

```

In the hand-written Python code, each item of output had to be appended manually to the list `l` of output strings. Cheetah makes things easier: it evaluates the code inside a `#for` loop and automatically appends the output of each iteration to the output of the template.

Cheetah also provides a `#while/#end while` directive, which is equivalent to Python's `while` construct.

Flow control: the `#if` directive

You've created a Cheetah template that reproduces the e-mail described above. Now let's improve it a little. The e-mail says you ordered "1 unit(s)" of blue widgets. It shouldn't be difficult to change the template to say "1 unit" if you ordered only one of something or "x units" otherwise. Cheetah provides an `#if` directive that lets you set up if-then-else conditionals. Here's a Cheetah template that tries to handle the plural correctly. The Python code is the same as always, so the following just presents the new template definition:

Listing 8. Using the `#if` directive to handle plurals

```
Hello, $user.firstName.  
Your order (#$order.id) has shipped:  
#for $purchased, $quantity in $order.purchased.items():  
    $purchased.name: $quantity unit  
#if $quantity != 1  
s  
#end if  
#end for
```

The only problem with this template definition is that Cheetah prints *s* on a separate line from *unit*:

```
Widget, green: 50 unit  
s  
Widget, blue: 1 unit
```

You can avoid this embarrassing fate a couple of ways: suppress the troublesome new line, or set the appropriate string as a variable ahead of time.

Suppressing new lines with the `#slurp` directive

The Cheetah directive `#slurp` tells Cheetah not to print the newline at the end of a particular line:

Listing 9. Suppressing the newline at the end of a line

```
#for $purchased, $quantity in $order.purchased.items():  
    $purchased.name: $quantity unit#slurp  
#if $quantity != 1  
s  
#end if  
#end for
```

This code gives the output you want:

```
Widget, green: 50 units
Widget, blue: 1 unit
```

Setting variables with the #set directive

If the #slurp directive looks ugly to you, there's another way. You can use the #set directive to create a temporary variable in the scope of the Cheetah template:

Listing 10. Use #set to create a temporary variable

```
#for $purchased, $quantity in $order.purchased.items():
    #if $quantity == 1
        #set $units = 'unit'
    #else
        #set $units = 'units'
    #end if
    $purchased.name: $quantity $units
#end for
```

In this case, #set lets you move the conditional out of the text generation and into the code that defines the variable. #set is a generally useful directive. You can also use it to create intermediate values or to avoid calling a costly method multiple times.

Generating other types of files

For simplicity's sake, all the examples so far have generated a plain text e-mail, but you don't have to stop there. Cheetah was originally designed to generate HTML, and you can use it to generate any text-based format: XML, SQL, even Python or other programming language code.

Here's a Cheetah template that gives an HTML rendering of an order status page. It might go into a file called OrderStatus.tmpl. Note its basic similarity to the e-mail rendition of the same information:

Listing 11. An HTML rendering of the order status information

```
<html>
<head><title>Status for order #${order.id}</title></head>
<body>
<p>[You are logged in as ${user.getFullName().}]</p>
<p>
    #if ($order.hasShipped())
        Your order has shipped. Your tracking number is ${order.trackingNumber}.
    #else
        Your order has not yet shipped.
    #end if
</p>
<p>Order #${order.id} contains the following items:</p>
<ul>
    #for $purchased, $quantity in $order.purchased.items():
        <li>${purchased.name}: $quantity unit#slurp
        #if ($quantity != 1)
            s
        #end if
        </li>
    #end for
</ul>
```

```
</ul>
<hr />
Served by Online Store v1.0
</body>
</html>
```

Combining templates

The HTML order status page defined previously has certain elements that are (or should be) common to all pages of the online store Web application: an HTML header with a page-specific title, a notice at the top of the page telling you that you're logged in, and an HTML footer. It should be possible to factor out these common page elements into a separate master template. That way, the template definition in `OrderStatus.tmpl` will contain only the code specific to displaying order status, and all the other template definitions can use the common code in the master template, instead of defining the same code each time.

Most templating systems (including Cheetah, with its `#include` directive) let you call one template from another. You could use this functionality to move the common template content into (for instance) `Header.tmpl` and `Footer.tmpl` files. However, Cheetah also lets you solve this problem in a more elegant way: by making it possible for templates to subclass each other.

Here's `Skeleton.tmpl`, a template definition that defines the skeleton of an HTML page, but leaves two items conspicuously unbound: `$title` and `$body`:

Listing 12. Skeleton.tmpl template definition

```
<html>
<head><title>$title</title></head>
<body>
<p>[You are logged in as $user.getFullName().]</p>
$body
<hr />
Served by Online Store v1.0.
</body>
</html>
```

Recall that running the `cheetah compile` command on the `Skeleton.tmpl` file generates `Skeleton.py`, a package containing a Python class called `Skeleton` that acts just like `Skeleton.tmpl`. Once that file is in place, you can write an order status template. `OrderStatusII.tmpl` can import the generated `Skeleton` class and subclass it. You'll also define values for `$title` and `$body`, variables referenced in the master template:

Listing 13. Subclassing Skeleton.tmpl

```
#from Skeleton import Skeleton
#extends Skeleton

#def title
Status for order #$order.id
#end def

#def body
<p>
#i f ($order.hasShipped())
    Your order has shipped. Your tracking number is $order.trackingNumber.
#e l s e
    Your order has not yet shipped.
#e n d i f
```



```

</p>
<p>Order #${order.id} contains the following items: </p>
<ul>
#for $purchased, $quantity in $order.purchased.items():
  <li>${purchased.name}: ${quantity} unit#slurp
#if ($quantity != 1)
  s
#end if
</li>
#end for
</ul>
#end def

```

OrderStatusII.tmpl uses an `#extends` directive to declare that its template is a specialization of the template defined in `Skeleton.tmpl`. It then uses `#def` directives to define functions called `title` and `body`. These correspond to the `$title` and `$body` variables used in the skeleton template. It won't do here to `#set` the value of `$title` and `$body`: the `#set` directive sets a value for a Python variable, but `title` and `body` are expected to correspond to Python functions.

As with the `def` keyword in Python, you can also use the Cheetah `#def` directive to define functions inside a template. You can then call the template multiple times, to avoid having to copy and paste code.

Conclusion

Cheetah offers many more features that aren't covered here. For instance, you can set up a filter that modifies the output of all variable references in a certain way. You can use the `#import` directive to import arbitrary Python modules into Cheetah templates and call their functions. In fact, almost anything you can do in Python you can do inside Cheetah.

However, I recommend that you keep it simple. Remember the goal behind templating systems: separate the dynamic parts of a document from its static description. Start putting application code into your Cheetah templates, and you'll find yourself with the same headaches that drive programmers and UI designers to choose templating systems in the first place. One aspect of Cheetah's philosophy: "Python for the back end, Cheetah for the front end." Follow this rule of thumb, and you should have no trouble reaping the benefits of this templating system.

Download

Description	Name	Size	Download method
Scripts and templates	os-c	de.zip	10 KB FTP

→ [Information about download methods](#)

→ [Get Adobe® Reader®](#)

Resources

Learn

- [The official Python wiki](#) lists many open-source templating systems. Perhaps the most popular alternatives to Cheetah are [Spyce](#) and [Zope Page Templates](#).

- [Cheetah](#) resources include a comprehensive [user's guide](#) and a quick [tutorial](#).
- Three Java templating systems form a major part of Cheetah's pedigree: [Velocity](#), [WebMacro](#), and [JavaServer Pages](#).
- Get started with Python with "[Python 101](#)."
- See "[Charming Python: Review of Python IDEs](#)" to learn more about Python tools.
- The article "[Discover Python, Part 1: Python's built-in numerical types](#)" is the first in a series written for Java developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Leonard Richardson is the author of many Python applications and libraries, including NewsBruiser and Beautiful Soup. He is a co-author of the new tome *Beginning Python*, from Wrox.