Cheetah Users' Guide

Release 0.9.16a1

Edited by Mike Orr and Tavis Rudd

January 6, 2005

cheetahtemplate-discuss@lists.sourceforge.net

Contents

---

1 Introduction

1.1 Who should read this Guide?

This Users' Guide provides a technical overview and reference for the Cheetah template system. Knowledge of Python and object-orientated programming is assumed. The emphasis in this Guide is on features useful in a wide variety of situations. Information on less common situations and troubleshooting tips are gradually being moved to the Cheetah FAQ. There is also a Cheetah Developer's Guide for those who want to know what goes on under the hood.

This Guide also contains examples of integrating Cheetah with Webware for Python. You will have to learn Webware from its own documentation in order to build a Webware + Cheetah site.

1.2 What is Cheetah?

Cheetah is a Python-powered template engine and code generator. It may be used as a standalone utility or combined with other tools. Cheetah has many potential uses, but web developers looking for a viable alternative to ASP, JSP, PHP and PSP are expected to be its principle user group.

Cheetah:

- generates HTML, SGML, XML, SQL, Postscript, form email, LaTeX, or any other text-based format. It has also been used to produce Python, Java and PHP source code.
- cleanly separates content, graphic design, and program code. This leads to highly modular, flexible, and reusable site architectures; faster development time; and HTML and program code that is easier to understand and maintain. It is particularly well suited for team efforts.
- blends the power and flexibility of Python with a simple template language that non-programmers can understand.
- gives template writers full access in their templates to any Python data structure, module, function, object, or method.
- makes code reuse easy by providing an object-oriented interface to templates that is accessible from Python code or other Cheetah templates. One template can subclass another and selectively reimplement sections of it. A compiled template *is* a Python class, so it can subclass a pure Python class and vice-versa.
- provides a simple yet powerful caching mechanism

Cheetah integrates tightly with **Webware for Python** (http://webware.sourceforge.net/): a Python-powered application server and persistent servlet framework. Webware provides automatic session, cookie, and user management and can be used with almost any operating-system, web server, or database. Through Python, it works with XML, SOAP, XML-RPC, CORBA, COM, DCOM, LDAP, IMAP, POP3, FTP, SSL, etc.. Python supports structured exception handling, threading, object serialization, unicode, string internationalization, advanced cryptography and more. It can also be extended with code and libraries written in C, C++, Java and other languages.

Like Python, Cheetah and Webware are Open Source software and are supported by active user communities. Together, they are a powerful and elegant framework for building dynamic web sites.

Like its namesake, Cheetah is fast, flexible and powerful.

1.3 What is the philosophy behind Cheetah?

Cheetah's design was guided by these principles:

- Python for the back end, Cheetah for the front end. Cheetah was designed to complement Python, not replace it.

- Cheetah's core syntax should be easy for non-programmers to learn.
- Cheetah should make code reuse easy by providing an object-oriented interface to templates that is accessible from Python code or other Cheetah templates.
- Python objects, functions, and other data structures should be fully accessible in Cheetah.
- Cheetah should provide flow control and error handling. Logic that belongs in the front end shouldn't be relegated to the back end simply because it's complex.
- It should be easy to **separate** content, graphic design, and program code, but also easy to **integrate** them.

  A clean separation makes it easier for a team of content writers, HTML/graphic designers, and programmers to work together without stepping on each other's toes and polluting each other's work. The HTML framework and the content it contains are two separate things, and analytical calculations (program code) is a third thing. Each team member should be able to concentrate on their specialty and to implement their changes without having to go through one of the others (i.e., the dreaded "webmaster bottleneck").

  While it should be easy to develop content, graphics and program code separately, it should be easy to integrate them together into a website. In particular, it should be easy:

  – for **programmers** to create reusable components and functions that are accessible and understandable to designers.
  – for **designers** to mark out placeholders for content and dynamic components in their templates.
  – for **designers** to soft-code aspects of their design that are either repeated in several places or are subject to change.
  – for **designers** to reuse and extend existing templates and thus minimize duplication of effort and code.
  – and, of course, for **content writers** to use the templates that designers have created.

Why Cheetah doesn't use HTML-style tags

Cheetah does not use HTML/XML-style tags like some other template languages for the following reasons: Cheetah is not limited to HTML, HTML-style tags are hard to distinguish from real HTML tags, HTML-style tags are not visible in rendered HTML when something goes wrong, HTML-style tags often lead to invalid HTML (e.g., `<img src="<template-directive>">`), Cheetah tags are less verbose and easier to understand than HTML-style tags, and HTML-style tags aren't compatible with most WYSIWYG editors

Besides being much more compact, Cheetah also has some advantages over languages that put information inside the HTML tags, such as Zope Page Templates or PHP: HTML or XML-bound languages do not work well with other languages, While ZPT-like syntaxes work well in many ways with WYSIWYG HTML editors, they also give up a significant advantage of those editors – concrete editing of the document. When logic is hidden away in (largely inaccessible) tags it is hard to understand a page simply by viewing it, and it is hard to confirm or modify that logic.

1.4 Give me an example!

Here's a very simple example that illustrates some of Cheetah's basic syntax:

```
<HTML>
<HEAD><TITLE>$title</TITLE></HEAD>
<BODY>

<TABLE>
#for $client in $clients
<TR>
<TD>$client.surname, $client.firstname</TD>
<TD><A HREF="mailto:$client.email">$client.email</A></TD>
</TR>
#end for
</TABLE>

</BODY>
</HTML>
```

Compare this with PSP:

```
<HTML>
<HEAD><TITLE><%=title%></TITLE></HEAD>
<BODY>

<TABLE>
<% for client in clients: %>
<TR>
<TD><%=client['surname']%>, <%=client['firstname']%></TD>
<TD><A HREF="mailto:<%=client['email']%>"><%=client['email']%></A></TD>
</TR>
<%end%>
</TABLE>

</BODY>
</HTML>
```

Section 3.7 has a more typical example that shows how to get the plug-in values *into* Cheetah, and section 4.2 explains how to turn your template definition into an object-oriented Python module.

1.5 Give me an example of a Webware servlet!

This example uses an HTML form to ask the user's name, then invokes itself again to display a *personalized* friendly greeting.

```
<HTML><HEAD><TITLE>My Template-Servlet</TITLE></HEAD><BODY>
#set $name = $request.field('name', None)
#if $name
Hello $name
#else
<FORM ACTION="" METHOD="GET">
Name: <INPUT TYPE="text" NAME="name"><BR>
<INPUT TYPE="submit">
</FORM>
#end if
</BODY></HTML>
```

To try it out for yourself on a Webware system:

1. copy the template definition to a file **test.tmpl** in your Webware servlet directory.
2. Run "`cheetah compile test.tmpl`". This produces **test.py** (a .py template module) in the same directory.
3. In your web browser, go to **test.py**, using whatever site and directory is appropriate. Depending on your Webware configuration, you may also be able to go to **test**.

At the first request, field 'name' will be blank (false) so the "#else" portion will execute and present a form. You type your name and press submit. The form invokes the same page. Now 'name' is true so the "#if" portion executes, which displays the greeting. The "#set" directive creates a local variable that lasts while the template is being filled.

1.6 How mature is Cheetah?

Cheetah is stable, production quality, post-beta code. Cheetah's syntax, semantics and performance have been generally stable since a performance overhaul in mid 2001. Most of the changes since October 2001 have been in response to specific requests by production sites, things they need that we hadn't considered.

As of summer 2003, we are putting in the final touches before the 1.0 release.

The **TODO** and **BUGS** files in the Cheetah distribution show what we're working on now or planning to work on. There's also a **ToDo** page on the wiki (see below), which is updated less often. The **WishList** page on the wiki shows requested features we're considering but haven't commited to.

1.7 Where can I get news?

Cheetah releases and other stuff can be obtained from the the Cheetah **Web site**: http://CheetahTemplate.sourceforge.net

Cheetah discussions take place on the **mailing list** cheetahtemplate-discuss@lists.sourceforge.net. This is where to hear the latest news first.

The Cheetah **wiki** is becoming an increasingly popular place to list examples of Cheetah in use, provide cookbook tips for solving various problems, and brainstorm ideas for future versions of Cheetah. http://www.cheetahtemplate.org/wiki (The wiki is actually hosted at http://cheetah.colorstudy.net/twiki/bin/view/Cheetah/WebHome, but the other URL is easier to remember.) For those unfamiliar with a wiki, it's a type of Web site that readers can edit themselves to make additions or corrections to. Try it. Examples and tips from the wiki will also be considered for inclusion in future versions of this Users' Guide.

If you encounter difficulties, or are unsure about how to do something, please post a detailed message to the list.

1.8 How can I contribute?

Cheetah is the work of many volunteers. If you use Cheetah please share your experiences, tricks, customizations, and frustrations.

Bug reports and patches

If you think there is a bug in Cheetah, send a message to the e-mail list with the following information:

1. a description of what you were trying to do and what happened
2. all tracebacks and error output
3. your version of Cheetah
4. your version of Python
5. your operating system
6. whether you have changed anything in the Cheetah installation

Example sites and tutorials

If you're developing a website with Cheetah, please put a link on the wiki on the **WhoIsUsingCheetah** page, and mention it on the list. Also, if you discover new and interesting ways to use Cheetah, please put a quick tutorial (HOWTO) about your technique on the **CheetahRecipies** page on the wiki.

Template libraries and function libraries

We hope to build up a framework of Template libraries (see section 17.5) to distribute with Cheetah and would appreciate any contributions.

Test cases

Cheetah is packaged with a regression testing suite that is run with each new release to ensure that everything is working as expected and that recent changes haven't broken anything. The test cases are in the Cheetah.Tests module. If you find a reproduceable bug please consider writing a test case that will pass only when the bug is fixed. Send any new test cases to the email list with the subject-line "new test case for Cheetah."

Publicity

Help spread the word ... recommend it to others, write articles about it, etc.

1.9 Acknowledgements

Cheetah is one of several templating frameworks that grew out of a 'templates' thread on the Webware For Python email list. Tavis Rudd, Mike Orr, Chuck Esterbrook and Ian Bicking are the core developers.

We'd like to thank the following people for contributing valuable advice, code and encouragement: Geoff Talvola, Jeff Johnson, Graham Dumpleton, Clark C. Evans, Craig Kattner, Franz Geiger, Geir Magnusson, Tom Schwaller, Rober Kuzelj, Jay Love, Terrel Shumway, Sasa Zivkov, Arkaitz Bitorika, Jeremiah Bellomy, Baruch Even, Paul Boddie, Stephan Diehl, Chui Tey, Michael Halle, Edmund Lian and Aaron Held.

The Velocity, WebMacro and Smarty projects provided inspiration and design ideas. Cheetah has benefitted from the creativity and energy of their developers. Thank you.

1.10 License

The gist Cheetah is open source, but products developed with Cheetah or derived from Cheetah may be open source or closed source.

Cheetah.Utils.optik is based on a third-party package Optik by Gregory P Ward. Optik's license is in appendix D.

2 Vocabulary

Template is an informal term meaning a template definition, a template instance or a template class. A **template definition** is what the human **template maintainer** writes: a string consisting of text, placeholders and directives. **Placeholders** are variables that will be looked up when the template is filled. **Directives** are commands to be executed when the template is filled, or instructions to the Cheetah compiler. The conventional suffix for a file containing a template definition is **.tmpl**.

There are two things you can do with a template: compile it or fill it. **Filling** is the reason you have a template in the first place: to get a finished string out of it. Compiling is a necessary prerequisite: the **Cheetah compiler** takes a template definition and produces Python code to create the finished string. Cheetah provides several ways to compile and fill templates, either as one step or two.

Cheetah's compiler produces a subclass of `Cheetah.Template` specific to that template definition; this is called the **generated class**. A **template instance** is an instance of a generated class.

If the user calls the `Template` constructor directly (rather than a subclass constructor), s/he will get what appears to be an instance of `Template` but is actually a subclass created on-the-fly.

The user can make the subclass explicit by using the "cheetah compile" command to write the template class to a Python module. Such a module is called a **.py template module**.

The **Template Definition Language** – or the "Cheetah language" for short – is the syntax rules governing placeholders and directives. These are discussed in sections 5 and following in this Guide.

To fill a template, you call its **main method**. This is normally `.respond()`, but it may be something else, and you can use the `#implements` directive to choose the method name. (Section 8.3.

A **template-servlet** is a .py template module in a Webware servlet directory. Such templates can be filled directly through the web by requesting the URL. "Template-servlet" can also refer to the instance being filled by a particular web request. If a Webware servlet that is not a template-servlet invokes a template, that template is not a template-servlet either.

A **placeholder tag** is the substring in the template definition that is the placeholder, including the start and end delimeters (if there is an end delimeter). The **placeholder name** is the same but without the delimeters.

Placeholders consist of one or more **identifiers** separated by periods (e.g., `a.b`). Each identifier must follow the same rules as Python identifiers; that is, a letter or underscore followed by one or more letters, digits or underscores. (This is the regular expression `[A-Za-z_][A-Za-z0-9_]*`.)

The first (or only) identifier of a placeholder name represents a **variable** to be looked up. Cheetah looks up variables in various **namespaces**: the searchList, local variables, and certain other places. The searchList is a list of objects (**containers**) with attributes and/or keys: each container is a namespace. Every template instance has exactly one searchList. Identifiers after the first are looked up only in the parent object. The final value after all lookups have been performed is the **placeholder value**.

Placeholders may occur in three positions: top-level, expression and LVALUE. **Top-level** placeholders are those in ordinary text ("top-level text"). **Expression** placeholders are those in Python expressions. **LVALUE** placeholders are those naming a variable to receive a value. (LVALUE is computerese for "the left side of the equal sign".) Section 5.3 explains the differences between these three positions.

The routine that does the placeholder lookups is called the **NameMapper**. Cheetah's NameMapper supports universal dotted notation and autocalling. **Universal dotted notation** means that keys may be written as if they were attributes: `a.b` instead of `a['b']`. **Autocalling** means that if any identifier's value is found to be a function or method, Cheetah will call it without arguments if there is no `()` following. More about the NameMapper is in section 5.5.

Some directives are multi-line, meaning they have a matching **#end** tag. The lines of text between the start and end tags is the **body** of the directive. Arguments on the same line as the start tag, in contrast, are considered part of the directive tag. More details are in section 5.8 (Directive Syntax Rules).

3 Getting Started

3.1 Requirements

Cheetah requires Python release 2.0 or greater, and has been tested with Python 2.0, 2.1 and 2.2. It is known to run on Linux, Windows NT/98/XP, FreeBSD and Solaris, and should run anywhere Python runs.

99% of Cheetah is written in Python. There is one small C module (`_namemapper.so`) for speed, but Cheetah automatically falls back to a Python equivalent (`NameMapper.py`) if the C module is not available.

3.2 Installation

To install Cheetah in your system-wide Python library:

1. Login as a user with privileges to install system-wide Python packages. On POSIX systems (AIX, Solaris, Linux, IRIX, etc.), the command is normally 'su root'. On non-POSIX systems such as Windows NT, login as an administrator.
2. Run `python setup.py install` at the command prompt.
3. The setup program will install the wrapper script **cheetah** to wherever it usually puts Python binaries ("/usr/bin/", "bin/" in the Python install directory, etc.)

Cheetah's installation is managed by Python's Distribution Utilities ('distutils'). There are many options for customization. Type `''python setup.py help''` for more information.

To install Cheetah in an alternate location – someplace outside Python's `site-packages/` directory, use one of these options:

```
python setup.py install --home /home/tavis
python setup.py install --install-lib /home/tavis/lib/python
```

Either way installs to /home/tavis/lib/python/Cheetah/ . Of course, /home/tavis/lib/python must be in your Python path in order for Python to find Cheetah.

3.3 Files

If you do the systemwide install, all Cheetah modules are installed in the **site-packages/Cheetah/** subdirectory of your standard library directory; e.g., /opt/Python2.2/lib/python2.2/site-packages/Cheetah.

Two commands are installed in Python's `bin/` directory or a system bin directory: `cheetah` (section 3.5) and `cheetah-compile` (section 4.2).

3.4 Uninstalling

To uninstall Cheetah, merely delete the site-packages/Cheetah/ directory. Then delete the "cheetah" and "cheetah-compile" commands from whichever bin/ directory they were put in.

3.5 The 'cheetah' command

Cheetah comes with a utility `cheetah` that provides a command-line interface to various housekeeping tasks. The command's first argument is the name of the task. The following commands are currently supported:

```
cheetah compile [options] [FILES ...]     : Compile template definitions
cheetah fill [options] [FILES ...]        : Fill template definitions
cheetah help                              : Print this help message
cheetah options                           : Print options help message
cheetah test                              : Run Cheetah's regression tests
cheetah version                           : Print Cheetah version number
```

You only have to type the first letter of the command: `cheetah c` is the same as `cheetah compile`.

The test suite is described in the next section. The `compile` command will be described in section 4.2, and the `fill` command in section 4.3.

The depreciated `cheetah-compile` program does the same thing as `cheetah compile`.

3.6 Testing your installation

After installing Cheetah, you can run its self-test routine to verify it's working properly on your system. Change directory to any directory you have write permission in (the tests write temporary files). Do not run the tests in the directory you installed Cheetah from, or you'll get unnecessary errors. Type the following at the command prompt:

```
cheetah test
```

The tests will run for about three minutes and print a success/failure message. If the tests pass, start Python in interactive mode and try the example in the next section.

Certain test failures are insignificant:

AssertionError: Template output mismatch: Expected Output = 0(end) Actual Output = False(end) Python 2.3 changed the string representation of booleans, and the tests haven't yet been updated to reflect this.

AssertionError: subcommand exit status 127 Certain tests run "cheetah" as a subcommand. The failure may mean the command wasn't found in your system path. (What happens if you run "cheetah" on the command line?) The failure also happens on some Windows systems for unknown reasons. This failure has never been observed outside the test suite. Long term, we plan to rewrite the tests to do a function call rather than a subcommand, which will also make the tests run significantly faster.

ImportError: No module named SampleBaseClass The test tried to write a temporary module in the current directory and `import` it. Reread the first paragraph in this section about the current directory.

ImportError: No module named tmp May be the same problem as SampleBaseClass; let us know if changing the current directory doesn't work.

If any other tests fail, please send a message to the e-mail list with a copy of the test output and the following details about your installation:

1. your version of Cheetah
2. your version of Python
3. your operating system
4. whether you have changed anything in the Cheetah installation

3.7 Quickstart tutorial

This tutorial briefly introduces how to use Cheetah from the Python prompt. The following chapters will discuss other ways to use templates and more of Cheetah's features.

The core of Cheetah is the `Template` class in the `Cheetah.Template` module. The following example shows how to use the `Template` class in an interactive Python session. `t` is the Template instance. Lines prefixed with `>>>` and `...` are user input. The remaining lines are Python output.

```
>>> from Cheetah.Template import Template
>>> templateDef = """
... <HTML>
... <HEAD><TITLE>$title</TITLE></HEAD>
... <BODY>
... $contents
... ## this is a single-line Cheetah comment and won't appear in the output
... #* This is a multi-line comment and won't appear in the output
...    blah, blah, blah
... *#
... </BODY>
... </HTML>"""
>>> nameSpace = {'title': 'Hello World Example', 'contents': 'Hello World!'}
>>> t = Template(templateDef, searchList=[nameSpace])
>>> print t

<HTML>
<HEAD><TITLE>Hello World Example</TITLE></HEAD>
<BODY>
Hello World!
</BODY>
</HTML>
>>> print t    # print it as many times as you want
      [ ... same output as above ... ]
>>> nameSpace['title'] = 'Example #2'
>>> nameSpace['contents'] = 'Hiya Planet Earth!'
>>> print t   # Now with different plug-in values.
<HTML>
<HEAD><TITLE>Example #2</TITLE></HEAD>
<BODY>
Hiya Planet Earth!
</BODY>
</HTML>
```

Since Cheetah is extremely flexible, you can achieve the same result this way:

```
>>> t2 = Template(templateDef)
>>> t2.title = 'Hello World Example!'
>>> t2.contents = 'Hello World'
>>> print t2
      [ ... same output as the first example above ... ]
>>> t2.title = 'Example #2'
>>> t2.contents = 'Hello World!'
>>> print t2
     [ ... same as Example #2 above ... ]
```

Or this way:

```
>>> class Template3(Template):
>>>     title = 'Hello World Example!'
>>>     contents = 'Hello World!'
>>> t3 = Template3(templateDef)
>>> print t3
     [ ... you get the picture ... ]
```

The template definition can also come from a file instead of a string, as we will see in section 4.1.

The above is all fine for short templates, but for long templates or for an application that depends on many templates in a hierarchy, it's easier to store the templates in separate *.tmpl files and use the **cheetah compile** program to convert them into Python classes in their own modules. This will be covered in section 4.2.

As an appetizer, we'll just briefly mention that you can store constant values *inside* the template definition, and they will be converted to attributes in the generated class. You can also create methods the same way. You can even use inheritance to arrange your templates in a hierarchy, with more specific templates overriding certain parts of more general templates (e.g., a "page" template overriding a sidebar in a "section" template).

For the minimalists out there, here's a template definition, instantiation and filling all in one Python statement:

```
>>> print Template("Templates are pretty useless without placeholders.")
Templates are pretty useless without placeholders.
```

You use a precompiled template the same way, except you don't provide a template definition since it was already established:

```
from MyPrecompiledTemplate import MyPrecompiledTemplate
t = MyPrecompiledTemplate()
t.name = "Fred Flintstone"
t.city = "Bedrock City"
print t
```

4 How Cheetah Works

As mentioned before, you can do two things with templates: compile them and fill them. (Actually you can query them too, to see their attributes and method values.) Using templates in a Python program was shown in section 3.7 (Quickstart tutorial). Here we'll focus on compiling and filling templates from the shell command line, and how to make .py template modules. The compiling information here is also important for template-servlets, which will be otherwise covered in chapter 14 (Webware).

4.1 Constructing Template Objects

The heart of Cheetah is the `Template` class in the `Cheetah.Template` module. You can use it directly if you have a template definition in hand, or indirectly through a precompiled template, which is a subclass. The constructor accepts the following keyword arguments. (If you're a beginner, learn the first three arguments now; the others are much less frequent.)

source The template definition as a string. You may omit the `source=` prefix *if it's the first argument*, as in all the examples below. The source can be a string literal in your module, or perhaps a string you read from a database or other data structure.

file A filename or file object containing the template definition. A filename must be a string, and a file object must be open for reading. By convention, template definition files have the extension **.tmpl**.

searchList A list of objects to add to the searchList. The attributes/keys of these objects will be consulted for `$placeholder` lookup.

filter A class that will format every `$placeholder` value. You may specify a class object or string. If a class object, it must be a subclass of `Cheetah.Filters.Filter`. If a string, it must be the name of one of the filters in filtersLib module (see next item). (You may also use the `#filter` directive (section 7.9) to switch filters at runtime.)

filtersLib A module containing the filters Cheetah should look up by name. The default is `Cheetah.Filters`. All classes in this module that are subclasses of `Cheetah.Filters.Filter` are considered filters.

errorCatcher A class to handle `$placeholder` errors. You may specify a class object or string. If a class object, it must be a subclass of `Cheetah.ErrorCatchers.ErrorCatcher`. If a string, it must be the name of one of the error catchers in `Cheetah.ErrorCatchers`. This is similar to the `#errorCatcher` directive (section 10.2).

compilerSettings A dictionary (or dictionary hierarchy) of settings that change Cheetah's behavior. Not yet documented.

To use `Template` directly, you *must* specify either `source` or `file`, but not both. To use a precompiled template, you *must not* specify either one, because the template definition is already built into the class. The other arguments, however, may be used in either case. Here are typical ways to create a template instance:

`t = Template("The king is a $placeholder1.")`
Pass the template definition as a string.

`t = Template(file="fink.tmpl")`
Read the template definition from a file named "fink.tmpl".

`t = Template(file=f)`
Read the template definition from file-like object 'f'.

`t = Template("The king is a $placeholder1.", searchList=[dict, obj])`
Pass the template definition as a string. Also pass two namespaces for the searchList: a dictionary 'dict' and an instance 'obj'.

`t = Template(file="fink.txt", searchList=[dict, obj])`
Same, but pass a filename instead of a string.

`t = Template(file=f, searchList=[dict, obj])`
Same with a file object.

If you use `Template` directly, the template definition will be compiled the first time it's filled. Compilation creates a template-specific class called the **generated class**, which is a subclass of `Template`. It then dynamically switches the instance so it's now an instance of this class. Don't worry if you don't understand this; it works.

When you precompile a template using the "cheetah compile" command, it writes the generated class to a file. Actually, what it writes is the source code for a Python module that contains the generated class. Again, the generated class is a subclass of `Template`. We call the generated module a **.py template module**. Thus, if you always use precompiled templates (as many people do), you can view Cheetah as a convenient front-end for writing certain kinds of Python modules, the way you might use a graphical dialog builder to make a dialog module.

Precompiled templates provide a slight performance boost because the compilation happens only once rather than every time it's instantiated. Also, once you import the .py template module and allow Python to create a .pyc or .pyo file, you skip the Python compiler too. The speed advantage of all this is negligable, but it may make a difference in programs that use templates many times a second.

`Template` subclasses Webware's `Servlet` class when available, so the generated class can be used as a Webware servlet. This is practical only with precompiled templates.

To fill a template, you call its **main method**. This is normally `.respond()`, but under certain circumstances it's `.writeBody()` or a user-defined name. (Section 8.3 explains why the method name is not always the same.) However, `.__str__()` is always an alias for the main method, so you can always use `print myTemplateInstance` or `str(myTempateInstance)` to fill it. You can also call any `#def` or `#block` method and it will fill just that portion of the template, although this feature is not often used.

4.2 "cheetah compile" and .py template modules

To create a .py template module, do either of these:

```
cheetah compile [options] [FILES ...]
cheetah c [options] [FILES ...]
```

The following options are supported:

```
--idir DIR, --odir DIR : input/output directories (default: current dir)
--iext EXT, --oext EXT : input/output filename extensions
  (default input: tmpl,  default output:  py)
-R : recurse subdirectories looking for input files
--debug : print lots of diagnostic output to standard error
--flat : no destination subdirectories
--nobackup : don't make backups
--stdout, -p : output to standard output (pipe)
```

Note: If Cheetah can't find your input files, or if it puts output files in the wrong place, use the `--debug` option to see what Cheetah thinks of your command line.

The most basic usage is:

```
cheetah compile a.tmpl           : writes a.py
cheetah compile a.tmpl b.tmpl    : writes a.py and b.py
```

Cheetah will automatically add the default input extension (.tmpl) if the file is not found. So the following two examples are the same as above (provided files "a" and "b" don't exist):

```
cheetah compile a                : writes a.py (from a.tmpl)
cheetah compile a b              : writes a.py and b.py
```

You can override the default input extension and output extension (`py`) using `--iext` and `--oext`, although there's little reason to do so. Cheetah assumes the extension has a leading dot (.) even if you don't specify it.

Use the `-R` option to recurse subdirectories:

```
cheetah compile dir1             : error, file is a directory
cheetah compile -R dir1          : look in 'dir1' for files to compile
cheetah compile                  : error, no file specified
cheetah compile -R               : look in current directory for files
                                   to compile
cheetah compile -R a b dir1      : compile files and recurse
```

When recursing, only regular files that end in the input extension (.tmpl) are considered source files. All other filenames are ignored.

The options `--idir` and `--odir` allow you to specify that the source (and/or destination) paths are relative to a certain directory rather than to the current directory. This is useful if you keep your *.tmpl and *.py files in separate directory hierarchies. After editing a source file, just run one of these (or put the command in a script or Makefile):

```
cheetah compile --odir /var/webware a.tmpl
cheetah compile -R --odir /var/webware
cheetah c --odir /var/webware sub/a.tmpl
                                 : writes /var/webware/sub/a.py
```

"cheetah compile" overwrites any existing `.py` file it finds, after backing it up to FILENAME.py_bak (unless you specify `--nobackup`). For this reason, you should make changes to the `.tmpl` version of the template rather than to the `.py` version.

For the same reason, if your template requires custom Python methods or other Python code, don't put it in the `FILENAME.py` file. Instead, put it in a separate base class and use the `#extends` directive to inherit from it.

Because FILENAME will be used as a class and module name, it must be a valid Python identifier. For instance, `cheetah compile spam-eggs.tmpl` is illegal because of the hyphen ("-"). This is sometimes inconvenient when converting a site of HTML files into Webware servlets. Fortunately, the *directory* it's in does not have to be an identifier. (*Hint:* for date-specific files, try converting 2002/04/12.html to 2002/04/12/index.tmpl. This also gives you a directory to store images or supplemental files.)

Occasionally you may want output files put directly into the output directory (or current directory), rather than into a subdirectory parallel to the input file. The `--flat` option does this. Note that this introduces the possibility that several input files might map to one output file. Cheetah checks for output file collisions before writing any files, and aborts if there are any collisions.

```
cheetah c sub/a.py            : writes sub/a.py
cheetah c --flat sub/a.py     : writes a.py
cheetah c --odir DEST sub/a.tmpl
                              : writes DEST/sub/a.py
cheetah c --flat --odir DEST sub/a.tmpl
                              : writes DEST/a.py
cheetah c --idir /home/henry sub/rollins.tmpl
                              : writes sub/rollins.py
cheetah c --flat --idir /home/henry sub/rollins.tmpl
                              : writes rollins.py
cheetah c --idir /home/henry --odir /home/henry sub/rollins.tmpl
                              : writes /home/henry/sub/rollins.py
cheetah c --flat --idir /home/henry --odir /home/henry sub/rollins.tmpl
                              : writes /home/henry/rollins.py
```

Whenever "cheetah compile" has to create an output directory or subdirectory, it also creates an __init__.py file in it. This file is necessary in order to make Python treat the directory as a Python package.

One of the advantages of .py template modules is that you don't lose any flexibility. The generated class contains all `#attr` values and `#def`/`#block` values as ordinary attributes and methods, so you can read the values individually from other Python tools for any kind of custom processing you want. For instance, you can extract the titles of all your templates into a database, or find all the servlets with a certain `$author` value.

4.3 "cheetah fill"

You can compile and fill a large number of template definitions from the command line in one step using `cheetah fill`. This compiles the template in memory; it does *not* save the .py template module to disk. Instead it writes a finished output file, which has the extension `.html` by default. All the options to `cheetah compile` work the same way here, and there are also a couple additional options:

```
--env : put the environment in the searchList
--pickle FILE : unpickle FILE and put that object in the searchList
```

Because you can't provide a searchList on the command line, the templates must either contain or inherit all the variables it needs, or use the `--env` and `--pickle` options to provide additional variables.

Examples:

```
cheetah fill a.tmpl              : writes a.html
cheetah fill a.tmpl b.tmpl       : writes a.html and b.html
cheetah f --oext txt a           : writes a.txt (from a.tmpl)
```

Using `--env` may have security or reliability implications because the environment normally contains lots of variables you inherited rather than defining yourself. If any of these variables override any of yours (say a `#def`), you will get incorrect output, may reveal private information, and may get an exception due to the variable being an unexpected type (environmental variables are always strings). Your calling program may wish to clear out the environment before setting environmental variables for the template.

There are two other differences between "cheetah compile" and "cheetah fill". Cheetah doesn't create __init__.py files when creating directories in fill mode. Also, the source filenames don't have to be identifiers.

4.4 Some trivia about .py template modules

We won't look inside .py template modules in this Guide except to note that they are very different from template definitions. The following template definition fragment:

```
The number is $Test.unittest.main.
```

compiles to this:

```
write("The number is ")
write(filter(VFN(VFS(SL,"Test.unittest",1),"main",0)
write(".")
```

The Cheetah Developers' Guide looks at .py template modules in depth, and shows what the various directives compile to. But you are welcome to take a peek at some .py template modules yourself if you're curious about what Cheetah does under the hood. It's all regular Python code: writing strings and function calls to a file-like object.

Looking at a .py template module may also help you see why something doesn't work, by seeing what Cheetah thought you meant. It also helps discourage you from modifying the .py file yourself, because who wants to keep all those function calls and arguments straight? Let the computer do the drudgery work.

4.5 Running a .py template module as a standalone program

In addition to importing your .py template module file into a Python script or using it as a Webware servlet, you can also run it from the command line as a standalone program. The program will print the filled template on standard output. This is useful while debugging the template, and for producing formatted output in shell scripts.

When running the template as a program, you cannot provide a searchList or set `self.` attributes in the normal way, so you must take alternative measures to ensure that every placeholder has a value. Otherwise, you will get the usual `NameMapper.NotFound` exception at the first missing value. You can either set default values in the template itself (via the `#attr` or `#def` directives) or in a Python superclass, or use the `--env` or `--pickle` command-line options, which work just like their "cheetah fill" counterparts.

Run `python FILENAME.py --help` to see all the command-line options your .py template module accepts.

4.6 Object-Oriented Documents

Because Cheetah documents are actually class definitions, templates may inherit from one another in a natural way, using regular Python semantics. For instance, consider this template, FrogBase.tmpl:

```
#def title
This document has not defined its title
#end def
#def htTitle
$title
#end def
<HTML><HEAD>
<TITLE>$title</TITLE>
</HEAD><BODY>
<H1>$htTitle</H1>
$body
</BODY></HTML>
```

And its subclassed document, Frog1.tmpl:

```
#from FrogBase import FrogBase
#extends FrogBase
#def title
The Frog Page
#end def
#def htTitle
The <IMG SRC="Frog.png"> page
#end def
#def body
... lots of info about frogs ...
#end def
```

This is a classic use of inheritance. The parent "template" is simply an abstract superclass. Each document specializes the output of its parent. For instance, here the parent defines `$htTitle` so that by default it's identical to whatever the `$title` is, but it can also be customized.

In many other templating systems, you'd have to use case statements or if-elseif blocks of some sort, repeated in many different sections of code.

While we show another Cheetah document inheriting from this parent, a Python class can inherit from it just as easily. This Python class could define its programmatically-driven value for `$body` and `$title`, simply by defining body() and title() methods that return a string. (Actually they can return anything, but we'll get into that later.)

```
from FrogBase import FrogBase
class Frog2(FrogBase):
def title(self):
return "Frog 2 Page"
# We don't override .htTitle, so it defaults to "Frog 2 Page" too.
def body(self):
return " ... more info about frogs ..."
```

Similarly, the Cheetah document can inherit from an arbitrary class. That's how Cheetah makes templates usable as Webware servlets, by subclassing `Servlet`. This technique should be possible for non-Webware systems too.

(*Note:* `FrogBase.tmpl` could be improved by using the `#block` directive, section 8.8.)

5 Language Overview

Cheetah's basic syntax was inspired by the Java-based template engines Velocity and WebMacro. It has two types of tags: **$placeholders** and **#directives**. Both types are case-sensitive.

Placeholder tags begin with a dollar sign (`$varName`) and are similar to data fields in a form letter or to the `%(key)s` fields on the left side of Python's `%` operator. When the template is filled, the placeholders are replaced with the values they refer to.

Directive tags begin with a hash character (#) and are used for comments, loops, conditional blocks, includes, and all other advanced features. (*Note:* you can customize the start and end delimeters for placeholder and directive tags, but in this Guide we'll assume you're using the default.)

Placeholders and directives can be escaped by putting a backslash before them. `\$var` and `\#if` will be output as literal text.

A placeholder or directive can span multiple physical lines, following the same rules as Python source code: put a backslash (\) at the end of all lines except the last line. However, if there's an unclosed parenthesis, bracket or brace pending, you don't need the backslash.

```
#if $this_is_a_very_long_line and $has_lots_of_conditions \
    and $more_conditions:
<H1>bla</H1>
#end if

#if $country in ('Argentina', 'Uruguay', 'Peru', 'Colombia',
    'Costa Rica', 'Venezuela', 'Mexico')
<H1>Hola, senorita!</H1>
#else
<H1>Hey, baby!</H1>
#end if
```

5.1 Language Constructs – Summary

1. Comments and documentation strings
   (a) `##single line`
   (b) `#* multi line *#`
2. Generation, caching and filtering of output
   (a) plain text
   (b) look up a value: `$placeholder`
   (c) evaluate an expression: `#echo` ...
   (d) same but discard the output: `#silent` ...
   (e) one-line if: `#if EXPR then EXPR else EXPR`
   (f) gobble the EOL: `#slurp`
   (g) parsed file includes: `#include` ...
   (h) raw file includes: `#include raw` ...
   (i) verbatim output of Cheetah code: `#raw` ... `#end raw`
   (j) cached placeholders: `$*var`, `$*<interval>*var`
   (k) cached regions: `#cache` ... `#end cache`

(l) set the output filter: `#filter` ...
(m) control output indentation: `#indent` ... (*not implemented yet*)

3. Importing Python modules and objects: `#import` ..., `#from` ...
4. Inheritance
   (a) set the base class to inherit from: `#extends`
   (b) set the name of the main method to implement: `#implements` ...
5. Compile-time declaration
   (a) define class attributes: `#attr` ...
   (b) define class methods: `#def` ...`#end def`
   (c) `#block` ...`#end block` provides a simplified interface to `#def` ...`#end def`
6. Run-time assignment
   (a) local vars: `#set` ...
   (b) global vars: `#set global` ...
   (c) deleting local vars: `#del` ...
7. Flow control
   (a) `#if` ...`#else` ...`#else if` (aka `#elif`) ...`#end if`
   (b) `#unless` ...`#end unless`
   (c) `#for` ...`#end for`
   (d) `#repeat` ...`#end repeat`
   (e) `#while` ...`#end while`
   (f) `#break`
   (g) `#continue`
   (h) `#pass`
   (i) `#stop`
8. error/exception handling
   (a) `#assert`
   (b) `#raise`
   (c) `#try` ...`#except` ...`#else` ...`#end try`
   (d) `#try` ...`#finally` ...`#end try`
   (e) `#errorCatcher` ... set a handler for exceptions raised by $placeholder calls.
9. Instructions to the parser/compiler
   (a) `#breakpoint`
   (b) `#compiler-settings` ...`#end compiler-settings`
10. Escape to pure Python code
    (a) evalute expression and print the output: `<%=` ... `%>`
    (b) execute code and discard output: `<%` ... `%>`
11. Fine control over Cheetah-generated Python modules
    (a) set the source code encoding of compiled template modules: `#encoding`
    (b) set the sh-bang line of compiled template modules: `#shBang`

The use of all these constructs will be covered in the next several chapters.

5.2 Placeholder Syntax Rules

- Placeholders follow the same syntax rules as Python variables except that they are preceded by `$` (the short form) or enclosed in `${}` (the long form). Examples:

```
$var
${var}
$var2.abc['def']('gh', $subplaceholder, 2)
${var2.abc['def']('gh', $subplaceholder, 2)}
```

  We recommend `$` in simple cases, and `${}` when followed directly by a letter or when Cheetah or a human template maintainer might get confused about where the placeholder ends. You may alternately use `$()` or `$[]`, although this may confuse the (human) template maintainer:

```
$(var)
$[var]
$(var2.abc['def']('gh', $subplaceholder, 2))
$[var2.abc['def']('gh', $subplaceholder, 2)]
```

  Note: Advanced users can change the delimiters to anything they want via the `#compiler` directive.

  Note 2: The long form can be used only with top-level placeholders, not in expressions. See section 5.3 for an elaboration on this.

- To reiterate Python's rules, placeholders consist of one or more identifiers separated by periods. Each identifier must start with a letter or an underscore, and the subsequent characters must be letters, digits or underscores. Any identifier may be followed by arguments enclosed in `()` and/or keys/subscripts in `[]`.
- Identifiers are case sensitive. `$var` does not equal `$Var` or `$vAr` or `$VAR`.
- Arguments inside `()` or `[]` are just like in Python. Strings may be quoted using any Python quoting style. Each argument is an expression and may use any of Python's expression operators. Variables used in argument expressions are placeholders and should be prefixed with `$`. This also applies to the *arg and **kw forms. However, you do *not* need the `$` with the special Python constants `None`, `True` and `False`. Examples:

```
$hex($myVar)
$func($arg=1234)
$func2($*args, $**kw)
$func3(3.14159, $arg2, None, True)
$myList[$mySubscript]
```

- Trailing periods are ignored. Cheetah will recognize that the placeholder name in `$varName.` is `varName`, and the period will be left alone in the template output.
- The syntax `${placeholderName, arg1="val1"}` passes arguments to the output filter (see `#filter`, section 7.9. The braces and comma are required in this case. It's conventional to omit the `$` before the keyword arguments (i.e. `arg1`) in this case.
- Cheetah ignores all dollar signs ($) that are not followed by a letter or an underscore.

The following are valid $placeholders:

```
$a $_ $var $_var $var1 $_1var $var2_ $dict.key $list[3]
$object.method $object.method() $object.method
$nest($nest($var))
```

These are not $placeholders but are treated as literal text:

```
$@var $^var $15.50 $$
```

5.3 Where can you use placeholders?

There are three places you can use placeholders: top-level position, expression position and LVALUE position. Each has slightly different syntax rules.

Top-level position means interspersed in text. This is the only place you can use the placeholder long form: `${var}`.

Expression position means inside a Cheetah expression, which is the same as a Python expression. The placeholder names a searchList or other variable to be read. Expression position occurs inside () and [] arguments within placeholder tags (i.e., a placeholder inside a placeholder), and in several directive tags.

LVALUE position means naming a variable that will be written to. LVALUE is a computer science term meaning "the left side of an assignment statement". The first argument of directives `#set`, `#for`, `#def`, `#block` and `#attr` is an LVALUE.

This stupid example shows the three positions. Top-level position is shown in `courier`, expression position is *italic*, and LVALUE position is **bold**.

#for **$count** in *$range*(*$ninetyNine*, 0, -1)
#set **$after** = *$count* - 1
`$count` bottles of beer on the wall. `$count` bottles of beer!
Take one down, pass it around. `$after` bottles of beer on the wall.
#end for
`$hex`(*$myVar*, **$default**=*None*)

The output of course is:

```
99 bottles of beer on the wall.  99 bottles of beer!
    Take one down, pass it around.  98 bottles of beer on the wall.
98 bottles of beer on the wall.  98 bottles of beer!
    Take one down, pass it around.  97 bottles of beer on the wall.
...
```

5.4 Are all those dollar signs really necessary?

$ is a "smart variable prefix". When Cheetah sees $, it determines both the variable's position and whether it's a searchList value or a non-searchList value, and generates the appropriate Python code.

In top-level position, the $ is *required*. Otherwise there's nothing to distinguish the variable from ordinary text, and the variable name is output verbatim.

In expression position, the $ is *required* if the value comes from the searchList or a "#set global" variable, *recommended* for local/global/builtin variables, and *not necessary* for the special constants `None`, `True` and `False`. This works because Cheetah generates a function call for a searchList placeholder, but a bare variable name for a local/global/builtin variable.

In LVALUE position, the $ is *recommended*. Cheetah knows where an LVALUE is expected, so it can handle your variable name whether it has $ or not.

EXCEPTION: Do *not* use the $ prefix for intermediate variables in a Python list comprehensions. This is a limitation of Cheetah's parser; it can't tell which variables in a list comprehension are the intermediate variables, so you have to help it. For example:

```
#set $theRange = [x ** 2 for x in $range(10)]
```

`$theRange` is a regular `#set` variable. `$range` is a Python built-in function. But `x` is a scratch variable internal to the list comprehension: if you type `$x`, Cheetah will miscompile it.

5.5 NameMapper Syntax

One of our core aims for Cheetah was to make it easy for non-programmers to use. Therefore, Cheetah uses a simplified syntax for mapping placeholders in Cheetah to values in Python. It's known as the **NameMapper syntax** and allows for non-programmers to use Cheetah without knowing (a) the difference between an instance and a dictionary, (b) what functions and methods are, and (c) what 'self' is. A side benefit is that you can change the underlying data structure (e.g., instance to dictionary or vice-versa) without having to modify the templates.

NameMapper syntax is used for all variables in Cheetah placeholders and directives. If desired, it can be turned off via the `Template` class' `'useNameMapper'` compiler setting. But it's doubtful you'd ever want to turn it off.

Example

Consider this scenario:

You are building a customer information system. The designers with you want to use information from your system on the client's website –AND– they want to understand the display code and so they can maintian it themselves.

You write a UI class with a 'customers' method that returns a dictionary of all the customer objects. Each customer object has an 'address' method that returns the a dictionary with information about the customer's address. The designers want to be able to access that information.

Using PSP, the display code for the website would look something like the following, assuming your servlet subclasses the class you created for managing customer information:

```
<%= self.customer()[ID].address()['city'] %>   (42 chars)
```

With Cheetah's NameMapper syntax, you can use any of the following:

```
$self.customers()[$ID].address()['city']       (39 chars)
--OR--
$customers()[$ID].address()['city']
--OR--
$customers()[$ID].address().city
--OR--
$customers()[$ID].address.city
--OR--
$customers[$ID].address.city                   (27 chars)
```

Which of these would you prefer to explain to the designers, who have no programming experience? The last form is 15 characters shorter than the PSP version and – conceptually – far more accessible. With PHP or ASP, the code would be even messier than with PSP.

This is a rather extreme example and, of course, you could also just implement `$getCustomer($ID).city` and obey the Law of Demeter (search Google for more on that). But good object orientated design isn't the point of this example.

Dictionary Access

NameMapper syntax allows access to dictionary items with the same dotted notation used to access object attributes in Python. This aspect of NameMapper syntax is known as 'Unified Dotted Notation'. For example, with Cheetah it is possible to write:

```
$customers()['kerr'].address()  --OR--  $customers().kerr.address()
```

where the second form is in NameMapper syntax.

This works only with dictionary keys that also happen to be valid Python identifiers.

Autocalling

Cheetah automatically detects functions and methods in Cheetah $variables and calls them if the parentheses have been left off. Our previous example can be further simplified to:

```
$customers.kerr.address
```

As another example, if 'a' is an object, 'b' is a method

```
$a.b
```

is equivalent to

```
$a.b()
```

If b returns a dictionary, then following variations are possible

```
$a.b.c  --OR--  $a.b().c  --OR--  $a.b()['c']
```

where 'c' is a key in the dictionary that a.b() returns.

Further notes:

- When Cheetah autocalls a function/method, it calls it without any arguments. Thus, the function/method must have been declared without arguments (except `self` for methods) or to provide default values for all arguments. If the function requires arguments, you must use the `()`.
- Cheetah autocalls only functions and methods. Classes and other callable objects are not autocalled. The reason is that the primary purpose of a function/method is to call it, whereas the primary purpose of an instance is to look up its attributes or call its methods, not to call the instance itself. And calling a class may allocate large sums of memory uselessly or have other side effects, depending on the class. For instance, consider `$myInstance.fname`. Do we want to look up `fname` in the namespace of `myInstance` or in the namespace of whatever `myinstance` returns? It could go either way, so Cheetah follows the principle of

least surprise. If you *do* want to call the instance, put the `()` on, or rename the `.__call__()` method to `.__str__`.

- Autocalling can be disabled via Cheetah's 'useAutocalling' compiler setting. You can also disable it for one placeholder by using the syntax `$getVar('varName', 'default value', False)`. (`.getvar()` works only with searchList values.)

5.6 Namespace cascading and the searchList

When Cheetah maps a variable name in a template to a Python value, it searches several namespaces in order:

1. **Local variables:** created by `#set`, `#for`, or predefined by Cheetah.
2. The **searchList**, consisting of:
   (a) `#set global` variables.
   (b) The **searchList** containers you passed to the `Template` constructor, if any.
   (c) The **Template instance** ("self"). This contains any attributes you assigned, `#def` methods and `#block methods`, attributes/methods inherited via `#extends`, and other attributes/methods built into `Template` or inherited by it (there's a list of all these methods in section 13.5).
3. **Python globals:** created by `#import`, `#from ... import`, or otherwise predefined by Cheetah.
4. **Python builtins:** `None`, `max`, etc.

The first matching name found is used.

Remember, these namespaces apply only to the *first* identifier after the `$`. In a placeholder like `$a.b`, only 'a' is looked up in the searchList and other namespaces. 'b' is looked up only inside 'a'.

A searchList container can be any Python object with attributes or keys: dictionaries, instances, classes or modules. If an instance contains both attributes and keys, its attributes are searched first, then its keys.

Because the `Template` instance is part of the searchList, you can access its attributes/methods without 'self': `$myAttr`. However, use the 'self' if you want to make sure you're getting the `Template` attribute and not a same-name variable defined in a higher namespace: `$self.myAttr`. This works because "self" itself is a local variable.

The final resulting value, after all lookups and function calls (but before the filter is applied) is called the *placeholder value*, no matter which namespace it was found in.

Note carefully: if you put an object 'myObject' in the searchList, you *cannot* look up `$myObject`! You can look up only the attributes/keys *inside* 'myObject'.

Earlier versions of Cheetah did not allow you to override Python builtin names, but this was fixed in Cheetah 0.9.15.

If your template will be used as a Webware servlet, do not override methods 'name' and 'log' in the `Template` instance or it will interfere with Webware's logging. However, it *is* OK to use those variables in a higher namespace, since Webware doesn't know about Cheetah namespaces.

5.7 Missing Values

If NameMapper can not find a Python value for a Cheetah variable name, it will raise the NameMapper.NotFound exception. You can use the `#errorCatcher` directive (section 10.2) or **errorCatcher** Template constructor argument (section 4.1) to specify an alternate behaviour. BUT BE AWARE THAT errorCatcher IS ONLY INTENDED FOR DEBUGGING!

To provide a default value for a placeholder, write it like this: `$getVar('varName', 'default value')`. If you don't specify the default; i.e., `$getVar('varName')`, the default is `None`, which is converted to '' by the output filter.

5.8 Directive Syntax Rules

Directive tags begin with a hash character (#) and are used for comments, loops, conditional blocks, includes, and all other advanced features. Cheetah uses a Python-like syntax inside directive tags and understands any valid Python expression. **However, unlike Python, Cheetah does not use colons (:) and indentation to mark off multi-line directives.** That doesn't work in an environment where whitespace is significant as part of the text. Instead, multi-line directives like `#for` have corresponding closing tags (`#end for`). Most directives are direct mirrors of Python statements.

Many directives have arguments after the opening tag, which must be in the specified syntax for the tag. All end tags have the following syntax:

```
#end TAG_NAME [EXPR]
```

The expression is ignored, so it's essentially a comment.

Directive closures and whitespace handling

Directive tags can be closed explicitly with #, or implicitly with the end of the line if you're feeling lazy.

```
#block testBlock #
Text in the body of the
block directive
#end block testBlock #
```

is identical to:

```
#block testBlock
Text in the body of the
block directive
#end block testBlock
```

When a directive tag is closed explicitly, it can be followed with other text on the same line:

```
bah, bah, #if $sheep.color == 'black'# black#end if # sheep.
```

When a directive tag is closed implicitly with the end of the line, all trailing whitespace is gobbled, including the newline character:

```
"""
foo #set $x = 2
bar
"""
outputs
"""
foo bar
"""

while
"""
foo #set $x = 2 #
bar
"""
outputs
"""
foo
bar
"""
```

When a directive tag is closed implicitly AND there is no other text on the line, the ENTIRE line is gobbled up including any preceeding whitespace:

```
"""
foo
   #set $x = 2
bar
"""
outputs
"""
foo
bar
"""

while
"""
foo
 - #set $x = 2
bar
"""
outputs
"""
foo
 - bar
"""
```

The `#slurp` directive (section 7.7) also gobbles up whitespace.

Spaces outside directives are output *exactly* as written. In the black sheep example, there's a space before "black" and another before "sheep". So although it's legal to put multiple directives on one line, it can be hard to read.

```
#if $a# #echo $a + 1# #end if
      - There's a space between each directive,
        or two extra spaces total.
#if $a##echo $a + 1##end if
      - No spaces, but you have to look closely
        to verify none of the ''##'' are comment markers.
#if $a##echo $a + 1##end if     ### A comment.
      - In ''###'', the first ''#'' ends the directive,
        the other two begin the comment.  (This also shows
how you can add extra whitespace in the directive
tag without affecting the output.)
#if $a##echo $a + 1##end if     # ## A comment.
      - More readable, but now there's a space before the
        comment.
```

6 Comments

Comments are used to mark notes, explanations, and decorative text that should not appear in the output. Cheetah maintains the comments in the Python module it generates from the Cheetah source code. There are two forms of the comment directive: single-line and multi-line.

All text in a template definition that lies between two hash characters (##) and the end of the line is treated as a single-line comment and will not show up in the output, unless the two hash characters are escaped with a backslash.

```
##============================  this is a decorative comment-bar
$var    ## this is an end-of-line comment
##============================
```

Any text between `#*` and `*#` will be treated as a multi-line comment.

```
#*
   Here is some multiline
   comment text
*#
```

If you put blank lines around method definitions or loops to separate them, be aware that the blank lines will be output as is. To avoid this, make sure the blank lines are enclosed in a comment. Since you normally have a comment before the next method definition (right?), you can just extend that comment to include the blank lines after the previous method definition, like so:

```
#def method1
... lines ...
#end def
#*


   Description of method2.
   $arg1, string, a phrase.
*#
#def method2($arg1)
... lines ...
#end def
```

6.1 Docstring Comments

Python modules, classes, and methods can be documented with inline 'documentation strings' (aka 'docstrings'). Docstrings, unlike comments, are accesible at run-time. Thus, they provide a useful hook for interactive help utilities.

Cheetah comments can be transformed into doctrings by adding one of the following prefixes:

```
##doc: This text will be added to the method docstring
#*doc: If your template file is MyTemplate.tmpl, running "cheetah compile"
       on it will produce MyTemplate.py, with a class MyTemplate in it,
       containing a method .respond().  This text will be in the .respond()
       method's docstring. *#

##doc-method: This text will also be added to .respond()'s docstring
#*doc-method: This text will also be added to .respond()'s docstring *#

##doc-class: This text will be added to the MyTemplate class docstring
#*doc-class: This text will be added to the MyTemplate class docstring *#

##doc-module: This text will be added to the module docstring MyTemplate.py
#*doc-module: This text will be added to the module docstring MyTemplate.py*#
```

6.2 Header Comments

Cheetah comments can also be transformed into module header comments using the following syntax:

```
##header: This text will be added to the module header comment
#*header: This text will be added to the module header comment *#
```

Note the difference between `##doc-module:` and `header:` : "cheetah-compile" puts `##doc-module:` text inside the module docstring. `header:` makes the text go *above* the docstring, as a set of #-prefixed comment lines.

7 Generating, Caching and Filtering Output

7.1 Output from complex expressions: #echo

Syntax:

```
#echo EXPR
```

The `#echo` directive is used to echo the output from expressions that can't be written as simple $placeholders.

```
Here is my #echo ', '.join(['silly']*5) # example
```

This produces:

```
Here is my silly, silly, silly, silly, silly example.
```

7.2 Executing expressions without output: #silent

Syntax:

```
#silent EXPR
```

`#silent` is the opposite of `#echo`. It executes an expression but discards the output.

```
#silent $myList.reverse()
#silent $myList.sort()
Here is #silent $covertOperation() # nothing
```

If your template requires some Python code to be executed at the beginning; (e.g., to calculate placeholder values, access a database, etc), you can put it in a "doEverything" method you inherit, and call this method using `#silent` at the top of the template.

7.3 One-line #if

Syntax:

```
#if EXPR1 then EXPR2 else EXPR3#
```

The `#if` flow-control directive (section 9.4) has a one-line counterpart akin to Perl's and C's `?:` operator. If `EXPR1` is true, it evaluates `EXPR2` and outputs the result (just like `#echo EXPR2#`). Otherwise it evaluates `EXPR3` and outputs that result. This directive is short-circuiting, meaning the expression that isn't needed isn't evaluated.

You MUST include both 'then' and 'else'. If this doesn't work for you or you don't like the style use multi-line `#if` directives (section 9.4).

The trailing # is the normal end-of-directive character. As usual it may be omitted if there's nothing after the directive on the same line.

7.4 Caching Output

Caching individual placeholders

By default, the values of each $placeholder is retrieved and interpolated for every request. However, it's possible to cache the values of individual placeholders if they don't change very often, in order to speed up the template filling.

To cache the value of a single `$placeholder`, add an asterisk after the $; e.g., `$*var`. The first time the template is filled, `$var` is looked up. Then whenever the template is filled again, the cached value is used instead of doing another lookup.

The `$*` format caches "forever"; that is, as long as the template instance remains in memory. It's also possible to cache for a certain time period using the form `$*<interval>*variable`, where `<interval>` is the interval. The time interval can be specified in seconds (5s), minutes (15m), hours (3h), days (2d) or weeks (1.5w). The default is minutes.

```
<HTML>
<HEAD><TITLE>$title</TITLE></HEAD>
<BODY>

$var ${var}           ## dynamic - will be reinterpolated for each request
$*var2 $*{var2}       ## static - will be interpolated only once at start-up
$*5*var3 $*5*{var3}   ## timed refresh - will be updated every five minutes.

</BODY>
</HTML>
```

Note that "every five minutes" in the example really means every five minutes: the variable is looked up again when the time limit is reached, whether the template is being filled that frequently or not. Keep this in mind when setting refresh times for CPU-intensive or I/O intensive operations.

If you're using the long placeholder syntax, `${}`, the braces go only around the placeholder name: `$*.5h*{var.func('arg')}`.

Sometimes it's preferable to explicitly invalidate a cached item whenever you say so rather than at certain time intervals. You can't do this with individual placeholders, but you can do it with cached regions, which will be described next.

Caching entire regions

Syntax:

```
#cache [id=EXPR] [timer=EXPR] [test=EXPR]
#end cache
```

The `#cache` directive is used to cache a region of content in a template. The region is cached as a single unit, after placeholders and directives inside the region have been evaluated. If there are any `$*<interval>*var` placholders inside the cache region, they are refreshed only when *both* the cache region *and* the placeholder are simultaneously due for a refresh.

Caching regions offers more flexibility than caching individual placeholders. You can specify the refresh interval using a placeholder or expression, or refresh according to other criteria rather than a certain time interval.

`#cache` without arguments caches the region statically, the same way as `$*var`. The region will not be automatically refreshed.

To refresh the region at an interval, use the `timer=EXPRESSION` argument, equivalent to `$*<interval>*`. The expression should evaluate to a number or string that is a valid interval (e.g., 0.5, '3m', etc).

To refresh whenever an expression is true, use `test=EXPRESSION`. The expression can be a method/function returning true or false, a boolean placeholder, several of these joined by `and` and/or `or`, or any other expression. If the expression contains spaces, it's easier to read if you enclose it in `()`, but this is not required.

To refresh whenever you say so, use `id=EXPRESSION`. Your program can then call `.refreshCache(ID)` whenever it wishes. This is useful if the cache depends on some external condition that changes infrequently but has just changed now.

You can combine arguments by separating them with commas. For instance, you can specify both `id=` and `interval=`, or `id=` and `test=`. (You can also combine interval and test although it's not very useful.) However, repeating an argument is undefined.

```
#cache
This is a static cache.  It will not be refreshed.
$a $b $c
#end cache

#cache timer='30m', id='cache1'
#for $cust in $customers
$cust.name:
$cust.street - $cust.city
#end for
#end cache

#cache id='sidebar', test=$isDBUpdated
... left sidebar HTML ...
#end cache

#cache id='sidebar2', test=($isDBUpdated or $someOtherCondition)
... right sidebar HTML ...
#end cache
```

The `#cache` directive cannot be nested.

We are planning to add a `'varyBy'` keyword argument in the future that will allow a separate cache instances to be created for a variety of conditions, such as different query string parameters or browser types. This is inspired by ASP.net's varyByParam and varyByBrowser output caching keywords.

7.5 #raw

Syntax:

```
#raw
#end raw
```

Any section of a template definition that is inside a `#raw ...#end raw` tag pair will be printed verbatim without any parsing of $placeholders or other directives. This can be very useful for debugging, or for Cheetah examples and

tutorials.

#raw is conceptually similar to HTML's <PRE> tag and LaTeX's
verbatim{} tag, but unlike those tags, #raw does not cause the body to appear in a special font or typeface. It can't, because Cheetah doesn't know what a font is.

7.6 #include

Syntax:

```
#include [raw] FILENAME_EXPR
#include [raw] source=STRING_EXPR
```

The #include directive is used to include text from outside the template definition. The text can come from an external file or from a $placeholder variable. When working with external files, Cheetah will monitor for changes to the included file and update as necessary.

This example demonstrates its use with external files:

```
#include "includeFileName.txt"
```

The content of "includeFileName.txt" will be parsed for Cheetah syntax.

And this example demonstrates use with $placeholder variables:

```
#include source=$myParseText
```

The value of $myParseText will be parsed for Cheetah syntax. This is not the same as simply placing the $placeholder tag "$myParseText" in the template definition. In the latter case, the value of $myParseText would not be parsed.

By default, included text will be parsed for Cheetah tags. The argument "raw" can be used to suppress the parsing.

```
#include raw "includeFileName.txt"
#include raw source=$myParseText
```

Cheetah wraps each chunk of #include text inside a nested Template object. Each nested template has a copy of the main template's searchList. However, #set variables are visible across includes only if the defined using the #set global keyword.

All directives must be balanced in the include file. That is, if you start a #for or #if block inside the include, you must end it in the same include. (This is unlike PHP, which allows unbalanced constructs in include files.)

7.7 #slurp

Syntax:

```
#slurp
```

The #slurp directive eats up the trailing newline on the line it appears in, joining the following line onto the current

line.

It is particularly useful in `#for` loops:

```
#for $i in range(5)
$i #slurp
#end for
```

outputs:

```
0 1 2 3 4
```

7.8 #indent

This directive is not implemented yet. When/if it's completed, it will allow you to

1. indent your template definition in a natural way (e.g., the bodies of `#if` blocks) without affecting the output
2. add indentation to output lines without encoding it literally in the template definition. This will make it easier to use Cheetah to produce indented source code programmatically (e.g., Java or Python source code).

There is some experimental code that recognizes the `#indent` directive with options, but the options are purposely undocumented at this time. So pretend it doesn't exist. If you have a use for this feature and would like to see it implemented sooner rather than later, let us know on the mailing list.

The latest specification for the future `#indent` directive is in the TODO file in the Cheetah source distribution.

7.9 Ouput Filtering and #filter

Syntax:

```
#filter FILTER_CLASS_NAME
#filter $PLACEHOLDER_TO_A_FILTER_INSTANCE
#filter None
```

Output from $placeholders is passed through an ouput filter. The default filter merely returns a string representation of the placeholder value, unless the value is `None`, in which case the filter returns an empty string. Only top-level placeholders invoke the filter; placeholders inside expressions do not.

Certain filters take optional arguments to modify their behaviour. To pass arguments, use the long placeholder syntax and precede each filter argument by a comma. By convention, filter arguments don't take a `$` prefix, to avoid clutter in the placeholder tag which already has plenty of dollar signs. For instance, the MaxLen filter takes an argument 'maxlen':

```
${placeholderName, maxlen=20}
${functionCall($functionArg), maxlen=$myMaxLen}
```

To change the output filter, use the `'filter'` keyword to the `Template` class constructor, or the `#filter` directive at runtime (details below). You may use `#filter` as often as you wish to switch between several filters, if certain `$placeholders` need one filter and other `$placeholders` need another.

The standard filters are in the module `Cheetah.Filters`. Cheetah currently provides:

`Filter`
: The default filter, which converts None to '' and everything else to `str(whateverItIs)`. This is the base class for all other filters, and the minimum behaviour for all filters distributed with Cheetah.

`ReplaceNone`
: Same.

`MaxLen`
: Same, but truncate the value if it's longer than a certain length. Use the 'maxlen' filter argument to specify the length, as in the examples above. If you don't specify 'maxlen', the value will not be truncated.

`Pager`
: Output a "pageful" of a long string. After the page, output HTML hyperlinks to the previous and next pages. This filter uses several filter arguments and environmental variables, which have not been documented yet.

`WebSafe`
: Same as default, but convert HTML-sensitive characters ('<', '&', '>') to HTML entities so that the browser will display them literally rather than interpreting them as HTML tags. This is useful with database values or user input that may contain sensitive characters. But if your values contain embedded HTML tags you want to preserve, you do not want this filter.

  The filter argument 'also' may be used to specify additional characters to escape. For instance, say you want to ensure a value displays all on one line. Escape all spaces in the value with ' ', the non-breaking space:

```
${$country, also=' '}
```

To switch filters using a class object, pass the class using the **filter** argument to the Template constructor, or via a placeholder to the `#filter` directive: `#filter $myFilterClass`. The class must be a subclass of `Cheetah.Filters.Filter`. When passing a class object, the value of **filtersLib** does not matter, and it does not matter where the class was defined.

To switch filters by name, pass the name of the class as a string using the **filter** argument to the Template constructor, or as a bare word (without quotes) to the `#filter` directive: `#filter TheFilter`. The class will be looked up in the **filtersLib**.

The filtersLib is a module containing filter classes, by default `Cheetah.Filters`. All classes in the module that are subclasses of `Cheetah.Filters.Filter` are considered filters. If your filters are in another module, pass the module object as the **filtersLib** argument to the Template constructor.

Writing a custom filter is easy: just override the `.filter` method.

```
def filter(self, val, **kw):     # Returns a string.
```

Return the *string* that should be output for 'val'. 'val' may be any type. Most filters return '' for `None`. Cheetah passes one keyword argument: `kw['rawExpr']` is the placeholder name as it appears in the template definition, including all subscripts and arguments. If you use the long placeholder syntax, any options you pass appear as keyword arguments. Again, the return value must be a string.

You can always switch back to the default filter this way: `#filter None`. This is easy to remember because "no filter" means the default filter, and because None happens to be the only object the default filter treats specially.

We are considering additional filters; see http://webware.colorstudy.net/twiki/bin/view/Cheetah/MoreFilters for the latest ideas.

8 Import, Inheritance, Declaration and Assignment

8.1 #import and #from directives

Syntax:

```
#import MODULE_OR_OBJECT [as NAME] [, ...]
#from MODULE import MODULE_OR_OBJECT [as NAME] [, ...]
```

The `#import` and `#from` directives are used to make external Python modules or objects available to placeholders. The syntax is identical to the import syntax in Python. Imported modules are visible globally to all methods in the generated Python class.

```
#import math
#import math as mathModule
#from math import sin, cos
#from math import sin as _sin
#import random, re
#from mx import DateTime         # ## Part of Egenix's mx package.
```

After the above imports, `$math`, `$mathModule`, `$sin`, `$cos` and `$_sin`, `$random`, `$re` and `$DateTime` may be used in `$placeholders` and expressions.

8.2 #extends

Syntax:

```
#extends CLASS
```

All templates are subclasses of `Cheetah.Template.Template`. However, it's possible for a template to subclass another template or a pure Python class. This is where `#extends` steps in: it specifies the parent class. It's equivalent to PSP's `''@page extends=''` directive.

Cheetah imports the class mentioned in an `#extends` directive automatically if you haven't imported it yet. The implicit importing works like this:

```
#extends Superclass
## Implicitly does '#from Superclass import Superclass'.

#extends Cheetah.Templates.SkeletonPage
## Implicitly does '#from Cheetah.Templates.SkeletonPage import SkeletonPage'.
```

If your superclass is in an unusual location or in a module named differently than the class, you must import it explicitly. There is no support for extending from a class that is not imported; e.g., from a template dynamically created from a string. Since the most practical way to get a parent template into a module is to precompile it, all parent templates essentially have to be precompiled.

There can be only one `#extends` directive in a template and it may list only one class. In other words, templates don't do multiple inheritance. This is intentional: it's too hard to initialize multiple base classes correctly from inside

a template. However, you can do multiple inheritance in your pure Python classes.

If your pure Python class overrides any of the standard `Template` methods such as `.__init__` or `.awake`, be sure to call the superclass method in your method or things will break. Examples of calling the superclass method are in section 13.4. A list of all superclass methods is in section 13.5.

In all cases, the root superclass must be `Template`. If your bottommost class is a template, simply omit the `#extends` in it and it will automatically inherit from `Template`. *If your bottommost class is a pure Python class, it must inherit from `Template` explicitly:*

```
from Cheetah.Template import Template
class MyPurePythonClass(Template):
```

If you're not keen about having your Python classes inherit from `Template`, create a tiny glue class that inherits both from your class and from `Template`.

Before giving any examples we'll stress that Cheetah does *not* dictate how you should structure your inheritance tree. As long as you follow the rules above, many structures are possible.

Here's an example for a large web site that has not only a general site template, but also a template for this section of the site, and then a specific template-servlet for each URL. (This is the "inheritance approach" discussed in the Webware chapter.) Each template inherits from a pure Python class that contains methods/attributes used by the template. We'll begin with the bottommost superclass and end with the specific template-servlet:

```
1.  SiteLogic.py (pure Python class containing methods for the site)
        from Cheetah.Template import Template
        class SiteLogic(Template):

2.  Site.tmpl/py  (template containing the general site framework;
                   this is the template that controls the output,
                   the one that contains "<HTML><HEAD>...", the one
                   that contains text outside any #def/#block.)
        #from SiteLogic import SiteLogic
        #extends SiteLogic
        #implements respond

3.  SectionLogic.py  (pure Python class with helper code for the section)
        from Site import Site
        class SectionLogic(Site)

4.  Section.tmpl/py  (template with '#def' overrides etc. for the section)
        #from SectionLogic import SectionLogic
        #extends SectionLogic

5.  page1Logic.py  (pure Python class with helper code for the template-servlet)
        from Section import Section
        class indexLogic(Section):

6.  page1.tmpl/py  (template-servlet for a certain page on the site)
        #from page1Logic import page1Logic
        #extends page1Logic
```

A pure Python classes might also contain methods/attributes that aren't used by their immediate child template, but are available for any descendant template to use if it wishes. For instance, the site template might have attributes for the name and e-mail address of the site administrator, ready to use as $placeholders in any template that wants it.

Whenever you use `#extends`, you often need `#implements` too, as in step 2 above. Read the next section to understand what `#implements` is and when to use it.

8.3 #implements

Syntax:

```
#implements METHOD
```

You can call any `#def` or `#block` method directly and get its outpt. The top-level content – all the text/placeholders/directives outside any `#def`/`#block` – gets concatenated and wrapped in a "main method", by default `.respond()`. So if you call `.respond()`, you get the "whole template output". When Webware calls `.respond()`, that's what it's doing. And when you do 'print t' or 'str(t)' on a template instance, you're taking advantage of the fact that Cheetah makes `.__str__()` an alias for the main method.

That's all fine and dandy, but what if your application prefers to call another method name rather than `.respond()`? What if it wants to call, say, `.send_output()` instead? That's where `#implements` steps in. It lets you choose the name for the main method. Just put this in your template definition:

```
#implements send_output
```

When one template extends another, every template in the inheritance chain has its own main method. To fill the template, you invoke exactly one of these methods and the others are ignored. The method you call may be in any of the templates in the inheritance chain: the base template, the leaf template, or any in between, depending on how you structure your application. So you have two problems: (1) calling the right method name, and (2) preventing an undesired same-name subclass method from overriding the one you want to call.

Cheetah assumes the method you will call is `.respond()` because that's what Webware calls. It further assumes the desired main method is the one in the lowest-level base template, because that works well with `#block` as described in the Inheritance Approach for building Webware servlets (section 14.2), which was originally the principal use for Cheetah. So when you use `#extends`, Cheetah changes that template's main method to `.writeBody()` to get it out of the way and prevent it from overriding the base template's `.respond()`.

Unfortunately this assumption breaks down if the template is used in other ways. For instance, you may want to use the main method in the highest-level leaf template, and treat the base template(s) as merely a library of methods/attributes. In that case, the leaf template needs `#implements respond` to change its main method name back to `.respond()` (or whatever your application desires to call). Likewise, if your main method is in one of the intermediate templates in an inheritance chain, that template needs `#implements respond`.

The other way the assumption breaks down is if the main method *is* in the base template but that template extends a pure Python class. Cheetah sees the `#extends` and dutifully but incorrectly renames the method to `.writeBody()`, so you have to use `#implements respond` to change it back. Otherwise the dummy `.respond()` in `Cheetah.Template` is found, which outputs... nothing. **So if you're using `#extends` and get no output, the *first* thing you should think is, "Do I need to add `#implements respond` somewhere?"**

8.4 #set

Syntax:

```
#set [global] $var = EXPR
```

`#set` is used to create and update local variables at run time. The expression may be any Python expression. Remember to preface variable names with $ unless they're part of an intermediate result in a list comprehension.

Here are some examples:

```
#set $size = $length * 1096
#set $buffer = $size + 1096
#set $area = $length * $width
#set $namesList = ['Moe','Larry','Curly']
#set $prettyCountry = $country.replace(' ', ' ')
```

`#set` variables are useful to assign a short name to a `$deeply.nested.value`, to a calculation, or to a printable version of a value. The last example above converts any spaces in the 'country' value into HTML non-breakable-space entities, to ensure the entire value appears on one line in the browser.

`#set` variables are also useful in `#if` expressions, but remember that complex logical routines should be coded in Python, not in Cheetah!

```
#if $size > 1500
  #set $adj = 'large'
#else
  #set $adj = 'small'
#end if
```

Or Python's one-line equivalent, "A and B or C". Remember that in this case, B must be a true value (not None, '', 0, [] or).

```
#set $adj = $size > 1500 and 'large' or 'small'
```

(Note: Cheetah's one-line `#if` will not work for this, since it produces output rather than setting a variable.

You can also use the augmented assignment operators:

```
## Increment $a by 5.
#set $a += 5
```

By default, `#set` variables are not visible in method calls or include files unless you use the `global` attribute: `#set global $var = EXPRESSION`. Global variables are visible in all methods, nested templates and included files. Use this feature with care to prevent surprises.

8.5 #del

Syntax:

```
#del $var
```

`#del` is the opposite of `#set`. It deletes a *local* variable. Its usage is just like Python's `del` statement:

```
#del $myVar
#del $myVar, $myArray[5]
```

Only local variables can be deleted. There is no directive to delete a `#set global` variable, a searchList variable, or any other type of variable.

8.6 #attr

Syntax:

```
#attr $var = EXPR
```

The `#attr` directive creates class attributes in the generated Python class. It should be used to assign simple Python literals such as numbers or strings. In particular, the expression must *not* depend on searchList values or `#set` variables since those are not known at compile time.

```
#attr $title = "Rob Roy"
#attr $author = "Sir Walter Scott"
#attr $version = 123.4
```

This template or any child template can output the value thus:

```
$title, by $author, version $version
```

If you have a library of templates derived from etexts (http://www.gutenberg.org/), you can extract the titles and authors and put them in a database (assuming the templates have been compiled into .py template modules):

8.7 #def

Syntax:

```
#def METHOD[(ARGUMENTS)]
#end def
```

Or the one-line variation:

```
#def METHOD[(ARGUMENTS)] : TEXT_AND_PLACEHOLDERS
```

The `#def` directive is used to define new methods in the generated Python class, or to override superclass methods. It is analogous to Python's `def` statement. The directive is silent, meaning it does not itself produce any output. However, the content of the method will be inserted into the output (and the directives executed) whenever the method is later called by a $placeholder.

```
#def myMeth()
This is the text in my method
$a $b $c(123)  ## these placeholder names have been defined elsewhere
#end def

## and now use it...
$myMeth()
```

The arglist and parentheses can be omitted:

```
#def myMeth
This is the text in my method
$a $b $c(123)
#end def

## and now use it...
$myMeth
```

Methods can have arguments and have defaults for those arguments, just like in Python. Remember the $ before variable names:

```
#def myMeth($a, $b=1234)
This is the text in my method
$a - $b
#end def

## and now use it...
$myMeth(1)
```

The output from this last example will be:

```
This is the text in my method
1 - 1234
```

There is also a single line version of the `#def` directive. **Unlike the multi-line directives, it uses a colon (:) to delimit the method signature and body**:

```
#attr $adj = 'trivial'
#def myMeth: This is the $adj method
$myMeth
```

Leading and trailing whitespace is stripped from the method. This is in contrast to:

```
#def myMeth2
This is the $adj method
#end def
```

where the method includes a newline after "method". If you don't want the newline, add `#slurp`:

```
#def myMeth3
This is the $adj method#slurp
#end def
```

Because `#def` is handled at compile time, it can appear above or below the placeholders that call it. And if a superclass placeholder calls a method that's overridden in a subclass, it's the subclass method that will be called.

8.8 #block ... #end block

The #block directive allows you to mark a section of your template that can be selectively reimplemented in a subclass. It is very useful for changing part of a template without having to copy-paste-and-edit the entire thing. The output from a template definition that uses blocks will be identical to the output from the same template with the #block ...#end block tags removed.

(*Note:* don't be confused by the generic word 'block" in this Guide, which means a section of code inside *any* #TAG ...#end TAG pair. Thus, an if-block, for-block, def-block, block-block etc. In this section we are talking only of block-blocks.)

To reimplement the block, use the #def directive. The magical effect is that it appears to go back and change the output text *at the point the original block was defined* rather than at the location of the reimplementation.

```
#block testBlock
Text in the contents
area of the block directive
#if $testIt
$getFoo()
#end if
#end block testBlock
```

You can repeat the block name in the #end block directive or not, as you wish.

#block directives can be nested to any depth.

```
#block outerBlock
Outer block contents

#block innerBlock1
inner block1 contents
#end block innerBlock1

#block innerBlock2
inner block2 contents
#end block innerBlock2

#end block outerBlock
```

Note that the name of the block is optional for the #end block tag.

Technically, #block directive is equivalent to a #def directive followed immediately by a #placeholder for the same name. In fact, that's what Cheetah does. Which means you can use $theBlockName elsewhere in the template to output the block content again.

There is a one-line #block syntax analagous to the one-line #def.

The block must not require arguments because the implicit placeholder that's generated will call the block without arguments.

9 Flow Control

9.1 #for ... #end for

Syntax:

```
#for $var in EXPR
#end for
```

The `#for` directive iterates through a sequence. The syntax is the same as Python, but remember the `$` before variables.

Here's a simple client listing:

```
<TABLE>
#for $client in $service.clients
<TR>
<TD>$client.surname, $client.firstname</TD>
<TD><A HREF="mailto:$client.email" >$client.email</A></TD>
</TR>
#end for
</TABLE>
```

Here's how to loop through the keys and values of a dictionary:

```
<PRE>
#for $key, $value in $dict.items()
$key: $value
#end for
</PRE>
```

Here's how to create list of numbers separated by hyphens. This "#end for" tag shares the last line to avoid introducing a newline character after each hyphen.

```
#for $i in range(15)
$i - #end for
```

If the location of the `#end for` offends your sense of indentational propriety, you can do this instead:

```
#for $i in $range(15)
$i - #slurp
#end for
```

The previous two examples will put an extra hyphen after last number. Here's how to get around that problem, using the `#set` directive, which will be dealt with in more detail below.

```
#set $sep = ''
#for $name in $names
$sep$name
#set $sep = ', '
#end for
```

Although to just put a separator between strings, you don't need a for loop:

```
#echo ', '.join($names)
```

9.2 #repeat ... #end repeat

Syntax:

```
#repeat EXPR
#end repeat
```

Do something a certain number of times. The argument may be any numeric expression. If it's zero or negative, the loop will execute zero times.

```
#repeat $times + 3
She loves me, she loves me not.
#repeat
She loves me.
```

Inside the loop, there's no way to tell which iteration you're on. If you need a counter variable, use `#for` instead with Python's `range` function. Since Python's ranges are base 0 by default, there are two ways to start counting at 1. Say we want to count from 1 to 5, and that `$count` is 5.

```
#for $i in $range($count)
#set $step = $i + 1
$step.  Counting from 1 to $count.
#end for


#for $i in $range(1, $count + 1)
$i.  Counting from 1 to $count.
#end for
```

A previous implementation used a local variable `$i` as the repeat counter. However, this prevented instances of `#repeat` from being nested. The current implementation does not have this problem as it uses a new local variable for every instance of `#repeat`.

9.3 #while ... #end while

Syntax:

```
#while EXPR
#end while
```

`#while` is the same as Python's `while` statement. It may be followed by any boolean expression:

```
#while $someCondition('arg1', $arg2)
The condition is true.
#end while
```

Be careful not to create an infinite loop. `#while 1` will loop until the computer runs out of memory.

9.4 #if ... #else if ... #else ... #end if

Syntax:

```
#if EXPR
#else if EXPR
#elif EXPR
#else
#end if
```

The `#if` directive and its kin are used to display a portion of text conditionally. `#if` and `#else if` should be followed by a true/false expression, while `#else` should not. Any valid Python expression is allowed. As in Python, the expression is true unless it evaluates to 0, '', None, an empty list, or an empty dictionary. In deference to Python, `#elif` is accepted as a synonym for `#else if`.

Here are some examples:

```
#if $size >= 1500
It's big
#else if $size < 1500 and $size > 0
It's small
#else
It's not there
#end if
```

```
#if $testItem($item)
The item $item.name is OK.
#end if
```

Here's an example that combines an `#if` tag with a `#for` tag.

```
#if $people
<table>
<tr>
<th>Name</th>
<th>Address</th>
<th>Phone</th>
</tr>
#for $p in $people
<tr>
<td>$p.name</td>
<td>$p.address</td>
<td>$p.phone</td>
</tr>
#end for
</table>
#else
<p> Sorry, the search did not find any people. </p>
#end if
```

See section 7.3 for the one-line `#if` directive, which is equivalent to Perl's and C's `?:` operator.

9.5 #unless ... #end unless

Syntax:

```
#unless EXPR
#end unless
```

`#unless` is the opposite of `#if`: the text is executed if the condition is **false**. Sometimes this is more convenient. `#unless EXPR` is equivalent to `#if not (EXPR)`.

```
#unless $alive
This parrot is no more!  He has ceased to be!
'E's expired and gone to meet 'is maker! ...
THIS IS AN EX-PARROT!!
#end unless
```

You cannot use `#else if` or `#else` inside an `#unless` construct. If you need those, use `#if` instead.

9.6 #break and #continue

Syntax:

```
#break
#continue
```

These directives are used as in Python. `#break` will exit a `#for` loop prematurely, while `#continue` will immediately jump to the next iteration in the `#for` loop.

In this example the output list will not contain "10 - ".

```
#for $i in range(15)
#if $i == 10
  #continue
#end if
$i - #slurp
#end for
```

In this example the loop will exit if it finds a name that equals 'Joe':

```
#for $name in $names
#if $name == 'Joe'
  #break
#end if
$name - #slurp
#end for
```

9.7 #pass

Syntax:

```
#pass
```

The `#pass` directive is identical to Python `pass` statement: it does nothing. It can be used when a statement is required syntactically but the program requires no action.

The following example does nothing if only $A is true

```
#if $A and $B
   do something
#elif $A
  #pass
#elif $B
  do something
#else
  do something
#end if
```

9.8 #stop

Syntax:

```
#stop
```

The `#stop` directive is used to stop processing of a template at a certain point. The output will show *only* what has been processed up to that point.

When `#stop` is called inside an `#include` it skips the rest of the included code and continues on from after the

`#include` directive. stop the processing of the included code. Likewise, when `#stop` is called inside a `#def` or `#block`, it stops only the `#def` or `#block`.

```
A cat
#if 1
  sat on a mat
  #stop
  watching a rat
#end if
in a flat.
```

will print

```
A cat
  sat on a mat
```

And

```
A cat
#block action
  sat on a mat
  #stop
  watching a rat
#end block
in a flat.
```

will print

```
A cat
  sat on a mat
in a flat.
```

9.9 #return

Syntax:

```
#return
```

This is used as in Python. `#return` will exit the current method with a default return value of `None` or the value specified. It may be used only inside a `#def` or a `#block`.

Note that `#return` is different from the `#stop` directive, which returns the sum of all text output from the method in which it is called. The following examples illustrate this point:

```
1
$test[1]
3
#def test
1.5
#if 1
#return '123'
#else
99999
#end if
#end def
```

will produce

```
1
2
3
```

while

```
1
$test
3
#def test
1.5
#if 1
#stop
#else
99999
#end if
#end def
```

will produce

```
1
1.5
3
```

10 Error Handling

There are two ways to handle runtime errors (exceptions) in Cheetah. The first is with the Cheetah directives that mirror Python's structured exception handling statements. The second is with Cheetah's `ErrorCatcher` framework. These are described below.

10.1 #try ... #except ... #end try, #finally, and #assert

Cheetah's exception-handling directives are exact mirrors Python's exception-handling statements. See Python's documentation for details. The following Cheetah code demonstrates their use:

```
#try
  $mightFail()
#except
  It failed
#end try

#try
  #assert $x == $y
#except AssertionError
  They're not the same!
#end try

#try
  #raise ValueError
#except ValueError
  #pass
#end try


#try
  $mightFail()
#except ValueError
  Hey, it raised a ValueError!
#except NameMapper.NotFound
  Hey, it raised a NameMapper.NotFound!
#else
  It didn't raise anything!
#end try

#try
  $mightFail()
#finally
  $cleanup()
#end try
```

Like Python, `#except` and `#finally` cannot appear in the same try-block, but can appear in nested try-blocks.

10.2 #errorCatcher and ErrorCatcher objects

Syntax:

```
#errorCatcher CLASS
#errorCatcher $PLACEHOLDER_TO_AN_ERROR_CATCHER_INSTANCE
```

`ErrorCatcher` is a debugging tool that catches exceptions that occur inside `$placeholder` tags and provides a customizable warning to the developer. Normally, the first missing namespace value raises a `NameMapper.NotFound` error and halts the filling of the template. This requires the developer to resolve the exceptions in order without seeing the subsequent output. When an `ErrorCatcher` is enabled, the developer can see all the exceptions at once as well as the template output around them.

The `Cheetah.ErrorCatchers` module defines the base class for ErrorCatchers:

```
class ErrorCatcher:
    _exceptionsToCatch = (NameMapper.NotFound,)

    def __init__(self, templateObj):
        pass

    def exceptions(self):
        return self._exceptionsToCatch

    def warn(self, exc_val, code, rawCode, lineCol):
        return rawCode
```

This ErrorCatcher catches `NameMapper.NotFound` exceptions and leaves the offending placeholder visible in its raw form in the template output. If the following template is executed:

```
#errorCatcher Echo
#set $iExist = 'Here I am!'
Here's a good placeholder: $iExist
Here's bad placeholder: $iDontExist
```

the output will be:

```
Here's a good placeholder: Here I am!
Here's bad placeholder: $iDontExist
```

The base class shown above is also accessible under the alias `Cheetah.ErrorCatchers.Echo`. `Cheetah.ErrorCatchers` also provides a number of specialized subclasses that warn about exceptions in different ways. `Cheetah.ErrorCatchers.BigEcho` will output

```
Here's a good placeholder: Here I am!
Here's bad placeholder: ==========&lt;$iDontExist could not be found&gt;==========
```

ErrorCatcher has a significant performance impact and is turned off by default. It can also be turned on with the `Template` class' `'errorCatcher'` keyword argument. The value of this argument should either be a string specifying which of the classes in `Cheetah.ErrorCatchers` to use, or a class that subclasses `Cheetah.ErrorCatchers.ErrorCatcher`. The `#errorCatcher` directive can also be used to change the errorCatcher part way through a template.

`Cheetah.ErrorCatchers.ListErrors` will produce the same ouput as `Echo` while maintaining a list of the errors that can be retrieved later. To retrieve the list, use the `Template` class' `'errorCatcher'` method to retrieve the errorCatcher and then call its `listErrors` method.

ErrorCatcher doesn't catch exceptions raised inside directives.

11 Instructions to the Parser/Compiler

11.1 #breakpoint

Syntax:

```
#breakpoint
```

`#breakpoint` is a debugging tool that tells the parser to stop parsing at a specific point. All source code from that point on will be ignored.

The difference between `#breakpoint` and `#stop` is that `#stop` occurs in normal templates (e.g., inside an `#if`) but `#breakpoint` is used only when debugging Cheetah. Another difference is that `#breakpoint` operates at compile time, while `#stop` is executed at run time while filling the template.

11.2 #compiler-settings

Syntax:

```
#compiler-settings
key = value    (no quotes)
#end compiler-settings

#compiler-settings reset
```

The `#compiler-settings` directive overrides Cheetah's standard settings, changing how it parses source code and generates Python code. This makes it possible to change the behaviour of Cheetah's parser/compiler for a certain template, or within a portion of the template.

The `reset` argument reverts to the default settings. With `reset`, there's no end tag.

Here are some examples of what you can do:

```
$myVar
#compiler-settings
cheetahVarStartToken = @
#end compiler-settings
@myVar
#compiler-settings reset
$myVar


## normal comment
#compiler-settings
commentStartToken = //
#end compiler-settings

// new style of comment

#compiler-settings reset

## back to normal comments
```

```
#slurp
#compiler-settings
directiveStartToken = %
#end compiler-settings

%slurp
%compiler-settings reset

#slurp
```

Here’s a partial list of the settings you can change:

1. syntax settings
   (a) cheetahVarStartToken
   (b) commentStartToken
   (c) multilineCommentStartToken
   (d) multilineCommentEndToken
   (e) directiveStartToken
   (f) directiveEndToken
2. code generation settings
   (a) commentOffset
   (b) outputRowColComments
   (c) defDocStrMsg
   (d) useNameMapper
   (e) useAutocalling
   (f) reprShortStrConstants
   (g) reprNewlineThreshold

The meaning of these settings and their default values will be documented in the future.

12 Fine Control over Cheetah-generated Python modules

12.1 Setting the source code encoding: #encoding

Including

```
#encoding UTF-8
```

in your Cheetah `.tmpl` file will result in

```
# -*- coding: UTF-8 -*-
```

being appended to the top of the `.py` module file that Cheetah's compiler generates.
See http://www.python.org/doc/2.3/whatsnew/section-encodings.html for more details.

12.2 Setting the sh-bang: #shBang

Including

```
#shBang #!/usr/local/bin/python2.3
```

in your Cheetah `.tmpl` file will result in

```
#!/usr/local/bin/python2.3
```

being appended to the top of the `.py` module file that Cheetah's compiler generates.
The default sh-bang is

```
#!/usr/bin/env python
```

13 Tips, Tricks and Troubleshooting

This chapter contains short stuff that doesn't fit anywhere else.

See the Cheetah FAQ for more specialized issues and for troubleshooting tips. Check the wiki periodically for recent tips contributed by users. If you get stuck and none of these resources help, ask on the mailing list.

13.1 Placeholder Tips

Here's how to do certain important lookups that may not be obvious. For each, we show first the Cheetah expression and then the Python equivalent, because you can use these either in templates or in pure Python subclasses. The Cheetah examples use NameMapper shortcuts (uniform dotted notation, autocalling) as much as possible.

To verify whether a variable exists in the searchList:

```
$varExists('theVariable')
self.varExists('theVariable')
```

This is useful in `#if` or `#unless` constructs to avoid a `#NameMapper.NotFound` error if the variable doesn't exist. For instance, a CGI GET parameter that is normally supplied but in this case the user typed the URL by hand and forgot the parameter (or didn't know about it). (`.hasVar` is a synonym for `.varExists`.)

To look up a variable in the searchList from a Python method:

```
self.getVar('theVariable')
self.getVar('theVariable', None)
self.getVar('theVariable', myDefault)
```

This is the equivalent to `$theVariable` in the template. `getVar` returns the second argument (`None` or `myDefault` if the variable is missing; or, if there is no second argument, it raises raises `NameMapper.NotFound`. However, it usually easier to write your method so that all needed searchList values come in as method arguments. That way the caller can just use a $placeholder to specify the argument, which is less verbose than you writing a getVar call.

To do a "safe" placeholder lookup that returns a default value if the variable is missing:

```
$getVar('theVariable', None)
$getVar('theVariable', $myDefault)
```

To get an environmental variable, put `os.environ` on the searchList as a container. Or read the envvar in Python code and set a placeholder variable for it.

Remember that variables found earlier in the searchList override same-name variables located in a later searchList object. Be careful when adding objects containing other variables besides the ones you want (e.g., `os.environ`, CGI parameters). The "other" variables may override variables your application depends on, leading to hard-to-find bugs. Also, users can inadvertently or maliciously set an environmental variable or CGI parameter you didn't expect, screwing up your program. To avoid all this, know what your namespaces contain, and place the namespaces you have the most control over first. For namespaces that could contain user-supplied "other" variables, don't put the namespace itself in the searchList; instead, copy the needed variables into your own "safe" namespace.

13.2 Diagnostic Output

If you need send yourself some debugging output, you can use `#silent` to output it to standard error:

```
#silent $sys.stderr.write("Incorrigible var is '$incorrigible'.\n")
#silent $sys.stderr.write("Is 'unknown' in the searchList? " +
    $getVar("unknown", "No.") + "\n" )
```

(Tip contributed by Greg Czajkowski.)

13.3 When to use Python methods

You always have a choice whether to code your methods as Cheetah `#def` methods or Python methods (the Python methods being located in a class your template inherits). So how do you choose?

Generally, if the method consists mostly of text and placeholders, use a Cheetah method (a `#def` method). That's why `#def` exists, to take the tedium out of writing those kinds of methods. And if you have a couple `#if` stanzas to `#set` some variables, followed by a `#for` loop, no big deal. But if your method consists mostly of directives and only a little text, you're better off writing it in Python. Especially be on the watch for extensive use of `#set`, `#echo` and `#silent` in a Cheetah method–it's a sure sign you're probably using the wrong language. Of course, though, you are free to do so if you wish.

Another thing that's harder to do in Cheetah is adjacent or nested multiline stanzas (all those directives with an accompanying `#end` directive). Python uses indentation to show the beginning and end of nested stanzas, but Cheetah can't do that because any indentation shows up in the output, which may not be desired. So unless all those extra spaces and tabs in the output are acceptable, you have to keep directives flush with the left margin or the preceding text.

The most difficult decisions come when you have conflicting goals. What if a method generates its output in parts (i.e., output concatenation), contains many searchList placeholders and lots of text, *and* requires lots of `#if ...#set ...#else #set ...#end if` stanzas. A Cheetah method would be more advantageous in some ways, but a Python method in others. You'll just have to choose, perhaps coding groups of methods all the same way. Or maybe you can split your method into two, one Cheetah and one Python, and have one method call the other. Usually this means the Cheetah method calling the Python method to calculate the needed values, then the Cheetah method produces the output. One snag you might run into though is that `#set` currently can set only one variable per statement, so if your Python method needs to return multiple values to your Cheetah method, you'll have to do it another way.

13.4 Calling superclass methods, and why you have to

If your template or pure Python class overrides a standard method or attribute of `Template` or one of its base classes, you should call the superclass method in your method to prevent various things from breaking. The most common methods to override are `.awake` and `.__init__`. `.awake` is called automatically by Webware early during the web transaction, so it makes a convenient place to put Python initialization code your template needs. You'll definitely want to call the superclass `.awake` because it sets up many wonderful attributes and methods, such as those to access the CGI input fields.

There's nothing Cheetah-specific to calling superclass methods, but because it's vital, we'll recap the standard Python techniques here.

In Python ¿= 2.2, you can simply do:

```
from Cheetah.Template import Template
class MyClass(Template):
def awake(self, trans):
super(MyClass, self).awake(trans)
... add your own great and exciting features here ...
```

For Python ¡ 2.2, you have to explicitly name the superclass and call the method as an unbound method:

```
from Cheetah.Template import Template
from Cheetah.Servlet import Servlet
class MyClass(Template):
def awake(self, trans):
Servlet.awake(self, trans)
... great and exciting features written by me ...
```

[@@MO: Need to test this. .awake is in Servlet, which is a superclass of Template. Do we really need both imports? Can we call Template.awake?]

To avoid hardcoding the superclass name in older Python, you can use this function `callbase()`, which emulates `super()` for older versions of Python. It also works even `super()` does exist, so you don't have to change your servlets immediately when upgrading. Note that the argument sequence is different than `super` uses.

```
================================================================================
# Place this in a module SOMEWHERE.py .  Contributed by Edmund Lian.
class CallbaseError(AttributeError):
    pass

def callbase(obj, base, methodname='__init__', args=(), kw={},
    raiseIfMissing=None):
    try: method = getattr(base, methodname)
    except AttributeError:
        if raiseIfMissing:
            raise CallbaseError, methodname
        return None
    if args is None: args = ()
    return method(obj, *args, **kw)
================================================================================
# Place this in your class that's overriding .awake (or any method).
from SOMEWHERE import callbase
class MyMixin:
        def awake(self, trans):
                args = (trans,)
                callbase(self, MyMixin, 'awake', args)
                ... everything else you want to do ...
================================================================================
```

13.5 All methods

Here is a list of all the standard methods and attributes that can be accessed from a placeholder. Some of them exist for you to call, others are mainly used by Cheetah internally but you can call them if you wish, and others are only for internal use by Cheetah or Webware. Do not use these method names in mixin classes (`#extends`, section 8.2) unless you intend to override the standard method.

Variables with a star prefix (*****) are frequently used in templates or in pure Python classes.

Inherited from Cheetah.Template

compile(source=None, file=None, moduleName=None, mainMethodName='respond') Compile the template. Automatically called by `.__init__`.

generatedModuleCode() Return the module code the compiler generated, or `None` if no compilation took place.

generatedClassCode() Return the class code the compiler generated, or `None` if no compilation took place.

*** searchList()** Return a reference to the underlying search list. (a list of objects). Use this to print out your searchList for debugging. Modifying the returned list will affect your placeholder searches!

*** errorCatcher()** Return a reference to the current error catcher.

*** refreshCache(cacheKey=None)** If 'cacheKey' is not `None`, refresh that item in the cache. If `None`, delete all items in the cache so they will be recalculated the next time they are encountered.

*** shutdown()** Break reference cycles before discarding a servlet.

*** getVar(varName, default=NoDefault, autoCall=True)** Look up a variable in the searchList. Same as `$varName` but allows you to specify a default value and control whether autocalling occurs.

*** varExists(varName, autoCall=True)**

*** getFileContents(path)** Read the named file. If used as a placeholder, inserts the file's contents in the output without interpretation, like `#include raw`. If used in an expression, returns the file's content (e.g., to assign it to a variable).

runAsMainProgram() This is what happens if you run a .py template module as a standalone program.

Private methods: **_bindCompiledMethod**, **_bindFunctionAsMethod**, **_includeCheetahSource**, **_genTmpFilename**, **_importAsDummyModule**, **_makeDummyPackageForDir**, **_importFromDummyPackage**, **_importModuleFromDirectory**.

Other private attributes:

*** _fileMtime** Time the template definition was modified, in Unix ticks. `None` if the template definition came from a string or file handle rather than a named file, same for the next three variables.

*** _fileDirName** The directory containing the template definition.

*** _fileBaseName** The basename of the template definition file.

*** _filePath** The directory+filename of the template definition.

Inherited from Cheetah.Utils.WebInputMixin

nonNumericInputError Exception raised by `.webInput`.

*** webInput(...)** Convenience method to access GET/POST variables from a Webware servlet or CGI script, or Webware cookie or session variables. See section 14.7 for usage information.

Inherited from Cheetah.SettingsManager

setting(name, default=NoDefault) Get a compiler setting.

hasSetting(name) Does this compiler setting exist?

setSetting(name, value) Set setting 'name' to 'value'. See `#compiler-settings`, section 11.2.

settings() Return the underlying settings dictionary. (Warning: modifying this dictionary will change Cheetah's behavior.)

copySettings() Return a copy of the underlying settings dictionary.

deepcopySettings() Return a deep copy of the underlying settings dictionary. See Python's `copy` module.

updateSettings(newSettings, merge=True) Update Cheetah's compiler settings from the 'newSettings' dictionary. If 'merge' is true, update only the names in newSettings and leave the other names alone. (The SettingsManager is smart enough to update nested dictionaries one key at a time rather than overwriting the entire old dictionary.) If 'merge' is false, delete all existing settings so that the new ones are the only settings.

updateSettingsFromPySrcStr(theString, merge=True) Same, but pass a string of `name=value` pairs rather than a dictionary, the same as you would provide in a `#compiler-settings` directive, section 11.2.

updateSettingsFromPySrcFile(path, merge=True) Same, but exec a Python source file and use the variables it contains as the new settings. (e.g., `cheetahVarStartToken = "@"`).

updateSettingsFromConfigFile(path, **kw) Same, but get the new settings from a text file in ConfigParser format (similar to Windows' *.ini file format). See Python's `ConfigParser` module.

updateSettingsFromConfigFileObj Same, but read the open file object 'inFile' for the new settings.

updateSettingsFromConfigStr(configStr, convert=True, merge=True Same, but read the new settings from a string in ConfigParser format.

writeConfigFile(path) Write the current compiler settings to a file named 'path' in *.ini format.

getConfigString() Return a string containing the current compiler settings in *.ini format.

Private methods: **_createConfigFile**.

Private methods inherited from `_SettingsCollector` in same module: **normalizePath**, **readSettingsFromContainer**, **isContainer**, **_getAllAttrsFromContainer**, **readSettingsFromPySrcFile**, **readSettingsFromPySrcStr**, **readSettingsFromConfigFile**, **readSettingsFromConfigFileObj**.

Inherited from Cheetah.Servlet *Do not override these in a subclass or assign to them as attributes if your template will be used as a servlet*, otherwise Webware will behave unpredictably. However, it *is* OK to put same-name variables in the searchList, because Webware does not use the searchList.

EXCEPTION: It's OK to override **awake** and **sleep** as long as you call the superclass methods. (See section 13.4.)

*** isControlledByWebKit** True if this template instance is part of a live transaction in a running WebKit servlet.

*** isWebwareInstalled** True if Webware is installed and the template instance inherits from WebKit.Servlet. If not, it inherits from Cheetah.Servlet.DummyServlet.

*** awake(transaction)** Called by WebKit at the beginning of the web transaction.

*** sleep(transaction)** Called by WebKit at the end of the web transaction.

*** respond(transaction)** Called by WebKit to produce the web transaction content. For a template-servlet, this means filling the template.

shutdown() Break reference cycles before deleting instance.

*** serverSidePath()** The filesystem pathname of the template-servlet (as opposed to the URL path).

transaction The current Webware transaction.

application The current Webware application.

response The current Webware response.

request The current Webware request.

session The current Webware session.

write Call this method to insert text in the filled template output.

Several other goodies are available to template-servlets under the `request` attribute, see section 14.7.

`transaction`, `response`, `request` and `session` are created from the current transaction when WebKit calls `awake`, and don't exist otherwise. Calling `awake` yourself (rather than letting WebKit call it) will raise an exception because the `transaction` argument won't have the right attributes.

Inherited from WebKit.Servlet These are accessible only if Cheetah knows Webware is installed. This listing is based on a CVS snapshot of Webware dated 22 September 2002, and may not include more recent changes.

The same caveats about overriding these methods apply.

name() The simple name of the class. Used by Webware's logging and debugging routines.

log() Used by Webware's logging and debugging routines.

canBeThreaded() True if the servlet can be multithreaded.

canBeReused() True if the servlet can be used for another transaction after the current transaction is finished.

serverSideDir() Depreciated by `.serverSidePath()`.

Private attributes: **_serverSitePath**. Also apparently `_application`, `_request`, `_response`, `_session`, `_servlet`, `_errorOccurred` although I don't see where they're defined.

13.6 Optimizing templates

Here are some things you can do to make your templates fill faster and user fewer CPU cycles. Before you put a lot of energy into this, however, make sure you really need to. In many situations, templates appear to initialize and fill instantaneously, so no optimization is necessary. If you do find a situation where your templates are filling slowly or taking too much memory or too many CPU cycles, we'd like to hear about it on the mailing list.

Cache $placeholders whose values don't change frequently. (Section 7.4).

Use `#set` for values that are very frequently used, especially if they come out of an expensive operation like a deeply.nested.structure or a database lookup. `#set` variables are set to Python local variables, which have a faster lookup time than Python globals or values from Cheetah's searchList.

Moving variable lookups into Python code may provide a speedup in certain circumstances. If you're just reading `self` attributes, there's no reason to use NameMapper lookup ($placeholders) for them. NameMapper does a lot more work than simply looking up a `self` attribute.

On the other hand, if you don't know exactly where the value will come from (maybe from `self`, maybe from the searchList, maybe from a CGI input variable, etc), it's easier to just make that an argument to your method, and then the template can handle all the NameMapper lookups for you:

```
#silent $myMethod($arg1, $arg2, $arg3)
```

Otherwise you'd have to call `self.getVar('arg1')` etc in your method, which is more wordy, and tedious.

13.7 PSP-style tags

`<%= ...%>` and `<%...%>` allow an escape to Python syntax inside the template. You do not need it to use Cheetah effectively, and we're hard pressed to think of a case to recommend it. Nevertheless, it's there in case you encounter a situation you can't express adequately in Cheetah syntax. For instance, to set a local variable to an elaborate initializer.

`<%= ...%>` encloses a Python expression whose result will be printed in the output.

`<%...%>` encloses a Python statement or expression (or set of statements or expressions) that will be included as-is into the generated method. The statements themselves won't produce any output, but you can use the local function `write(EXPRESSION)` to produce your own output. (Actually, it's a method of a file-like object, but it looks like a local function.) This syntax also may be used to set a local variable with a complicated initializer.

To access Cheetah services, you must use Python code like you would in an inherited Python class. For instance, use `self.getVar()` to look up something in the searchList.

Warning: **No error checking is done!** If you write:

```
<% break %>      ## Wrong!
```

you'll get a `SyntaxError` when you fill the template, but that's what you deserve.

Note that these are PSP-*style* tags, not PSP tags. A Cheetah template is not a PSP document, and you can't use PSP commands in it.

13.8 Makefiles

If your project has several templates and you get sick of typing "cheetah compile FILENAME.tmpl" all the time–much less remembering which commands to type when–and your system has the `make` command available, consider building a Makefile to make your life easier.

Here's a simple Makefile that controls two templates, ErrorsTemplate and InquiryTemplate. Two external commands, `inquiry` and `receive`, depend on ErrorsTemplate.py. Aditionally, InquiryTemplate itself depends on ErrorsTemplate.

```
all:  inquiry  receive

.PHONY:  all  receive  inquiry  printsource

printsource:
        a2ps InquiryTemplate.tmpl ErrorsTemplate.tmpl

ErrorsTemplate.py:  ErrorsTemplate.tmpl
        cheetah compile ErrorsTemplate.tmpl

InquiryTemplate.py:  InquiryTemplate.tmpl ErrorsTemplate.py
        cheetah compile InquiryTemplate.tmpl

inquiry: InquiryTemplate.py  ErrorsTemplate.py

receive: ErrorsTemplate.py
```

Now you can type `make` anytime and it will recompile all the templates that have changed, while ignoring the ones that haven't. Or you can recompile all the templates `receive` needs by typing `make receive`. Or you can recompile only ErrorsTemplate by typing `make ErrorsTemplate`. There's also another target, "printsource": this sends a Postscript version of the project's source files to the printer. The .PHONY target is explained in the `make` documentation; essentially, you have it depend on every target that doesn't produce an output file with the same name as the target.

13.9 Using Cheetah in a Multi-Threaded Application

Template classes may be shared freely between threads. However, template instances should not be shared unless you either:

- Use a lock (mutex) to serialize template fills, to prevent two threads from filling the template at the same time.
- Avoid thread-unsafe features:
  - Modifying searchList values or instance variables.
  - Caching (`$*var`, `#cache`, etc).
  - `#set global`, `#filter`, `#errorCatcher`.

  Any changes to these in one thread will be visible in other threads, causing them to give inconsistent output.

About the only advantage in sharing a template instance is building up the placeholder cache. But template instances are so low overhead that it probably wouldn't take perceptibly longer to let each thread instantiate its own template instance. Only if you're filling templates several times a second would the time difference be significant, or if some of the placeholders trigger extremely slow calculations (e.g., parsing a long text file each time). The biggest overhead in Cheetah is importing the `Template` module in the first place, but that has to be done only once in a long-running application.

You can use Python's `mutex` module for the lock, or any similar mutex. If you have to change searchList values or instance variables before each fill (which is usually the case), lock the mutex before doing this, and unlock it only after the fill is complete.

For Webware servlets, you're probably better off using Webware's servlet caching rather than Cheetah's caching. Don't override the servlet's `.canBeThreaded()` method unless you avoid the unsafe operations listed above.

13.10 Using Cheetah with gettext

gettext is a project for creating internationalized applications. For more details, visit http://docs.python.org/lib/module-gettext.html. gettext can be used with Cheetah to create internationalized applications, even for CJK character sets, but you must keep a couple things in mind:

- xgettext is used on compiled templates, not on the templates themselves.
- The way the NameMapper syntax gets compiled to Python gets in the way of the syntax that xgettext recognizes. Hence, a special case exists for the functions `_`, `N_`, and `ngettext`. If you need to use a different set of functions for marking strings for translation, you must set the Cheetah setting `gettextTokens` to a list of strings representing the names of the functions you are using to mark strings for translation.

14 Using Cheetah with Webware

Webware for Python is a 'Python-Powered Internet Platform' that runs servlets in a manner similar to Java servlets. **WebKit** is the name of Webware's application server. For more details, please visit http://webware.sourceforge.net/.

All comments below refer to the official version of Webware, the DamnSimple! offshoot at ?, and the now-abandoned WebwareExperimental implementation at http://sourceforge.net/projects/expwebware/, except where noted. All the implementations are 95% identical to the servlet writer: their differences lie in their internal structure and configuration files. One difference is that the executable you run to launch standard Webware is called `AppServer`, whereas in WebwareExperimental it's called `webkit`. But to servlets they're both "WebKit, Webware's application server", so it's one half dozen to the other. In this document, we generally use the term **WebKit** to refer to the currently-running application server.

14.1 Installing Cheetah on a Webware system

Install Cheetah after you have installed Webware, following the instructions in chapter 3.

The standard Cheetah test suite ('cheetah test') does not test Webware features. We plan to build a test suite that can run as a Webware servlet, containing Webware-specific tests, but that has not been built yet. In the meantime, you can make a simple template containing something like "This is a very small template.", compile it, put the *.py template module in a servlet directory, and see if Webware serves it up OK.

You must not have a Webware context called "Cheetah". If you do, Webware will mistake that directory for the Cheetah module directory, and all template-servlets will bomb out with a "ImportError: no module named Template". (This applies only to the standard Webware; WebwareExperimental does not have contexts.)

If Webware complains that it cannot find your servlet, make sure '.tmpl' is listed in 'ExtensionsToIgnore' in your 'Application.config' file.

14.2 Containment vs Inheritance

Because Cheetah's core is flexible, there are many ways to integrate it with Webware servlets. There are two broad strategies: the **Inheritance approach** and the **Containment approach**. The difference is that in the Inheritance approach, your template object *`is`* the servlet, whereas in the Containment approach, the servlet is not a template but merely *uses* template(s) for portion(s) of its work.

The Inheritance approach is recommended for new sites because it's simpler, and because it scales well for large sites with a site-¿section-¿subsection-¿servlet hierarchy. The Containment approach is better for existing servlets that you don't want to restructure. For instance, you can use the Containment approach to embed a discussion-forum table at the bottom of a web page.

However, most people who use Cheetah extensively seem to prefer the Inheritance approach because even the most analytical servlet needs to produce *some* output, and it has to fit the site's look and feel *anyway*, so you may as well use a template-servlet as the place to put the output. Especially since it's so easy to add a template-servlet to a site once the framework is established. So we recommend you at least evaluate the effort that would be required to convert your site framework to template superclasses as described below, vs the greater flexibility and manageability it might give the site over the long term. You don't necessarily have to convert all your existing servlets right away: just build common site templates that are visually and behaviorally compatible with your specification, and use them for new servlets. Existing servlets can be converted later, if at all.

Edmund Liam is preparing a section on a hybrid approach, in which the servlet is not a template, but still calls template(s) in an inheritance chain to produce the output. The advantage of this approach is that you aren't dealing with `Template` methods and Webware methods in the same object.

The Containment Approach

In the Containment approach, your servlet is not a template. Instead, it it makes its own arrangements to create and use template object(s) for whatever it needs. The servlet must explicitly call the template objects' `.respond()` (or `.__str__()`) method each time it needs to fill the template. This does not present the output to the user; it merely gives the output to the servlet. The servlet then calls its `#self.response().write()` method to send the output to the user.

The developer has several choices for managing her templates. She can store the template definition in a string, file or database and call `Cheetah.Template.Template` manually on it. Or she can put the template definition in a *.tmpl file and use **cheetah compile** (section 4.2) to convert it to a Python class in a *.py module, and then import it into her servlet.

Because template objects are not thread safe, you should not store one in a module variable and allow multiple servlets to fill it simultaneously. Instead, each servlet should instantiate its own template object. Template *classes*, however, are thread safe, since they don't change once created. So it's safe to store a template class in a module global variable.

The Inheritance Approach

In the Inheritance approach, your template object doubles as as Webware servlet, thus these are sometimes called **template-servlets**. **cheetah compile** (section 4.2) automatically creates modules containing valid Webware servlets. A servlet is a subclass of Webware's `WebKit.HTTPServlet` class, contained in a module with the same name as the servlet. WebKit uses the request URL to find the module, and then instantiates the servlet/template. The servlet must have a `.respond()` method (or `.respondToGet()`, `.respondToPut()`, etc., but the Cheetah default is `.respond()`). Servlets created by `cheetah compile` meet all these requirements.

(Cheetah has a Webware plugin that automatically converts a `.tmpl servlet file` into a `.py servlet file` when the `.tmpl servlet file` is requested by a browser. However, that plugin is currently unavailable because it's being redesigned. For now, use `cheetah compile` instead.)

What about logic code? Cheetah promises to keep content (the placeholder values), graphic design (the template definition and is display logic), and algorithmic logic (complex calculations and side effects) separate. How? Where do you do form processing?

The answer is that your template class can inherit from a pure Python class containing the analytical logic. You can either use the `#extends` directive in Cheetah to indicate the superclass(es), or write a Python `class` statement to do the same thing. See the template `Cheetah.Templates.SkeletonPage.tmpl` and its pure Python class `Cheetah.Templates._SkeletonPage.py` for an example of a template inheriting logic code. (See sections 8.2 and 8.3 for more information about `#extends` and `#implements`. They have to be used a certain right way.)

If `#WebKit.HTTPServlet` is not available, Cheetah fakes it with a dummy class to satisfy the dependency. This allows servlets to be tested on the command line even on systems where Webware is not installed. This works only with servlets that don't call back into WebKit for information about the current web transaction, since there is no web transaction. Trying to access form input, for instance, will raise an exception because it depends on a live web request object, and in the dummy class the request object is `None`.

Because Webware servlets must be valid Python modules, and "cheetah compile" can produce only valid module names, if you're converting an existing site that has .html filenames with hyphens (-), extra dots (.), etc, you'll have to rename them (and possibly use redirects).

14.3 Site frameworks

Web sites are normally arranged hierarchically, with certain features common to every page on the site, other features common to certain sections or subsections, and others unique to each page. You can model this easily with a hierarchy of classes, with specific servlets inheriting from their more general superclasses. Again, you can do this two ways, using Cheetah's **Containment** approach or **Inheritance** approach.

In the Inheritance approach, parents provide `#block`s and children override them using `#def`. Each child `#extend`s its immediate parent. Only the leaf servlets need to be under WebKit's document root directory. The superclass servlets can live anywhere in the filesystem that's in the Python path. (You may want to modify your WebKit startup script to add that library directory to your `PYTHONPATH` before starting WebKit.)

Section 17.7 contains information on a stock template that simplifies defining the basic HTML structure of your web page templates.

In the Containment approach, your hierarchy of servlets are not templates, but each uses one or more templates as it wishes. Children provide callback methods to to produce the various portions of the page that are their responsibility, and parents call those methods. Webware's `WebKit.Page` and `WebKit.SidebarPage` classes operate like this.

Note that the two approaches are not compatible! `WebKit.Page` was not designed to intermix with `Cheetah.Templates.SkeletonPage`. Choose either one or the other, or expect to do some integration work.

If you come up with a different strategy you think is worth noting in this chapter, let us know.

14.4 Directory structure

Here's one way to organize your files for Webware+Cheetah.

```
www/                        # Web root directory.
    site1.example.com/      # Site subdirectory.
        apache/             # Web server document root (for non-servlets).
        www/                # WebKit document root.
           index.py         # http://site1.example.com/
           index.tmpl       # Source for above.
           servlet2.py      # http://site1.example.com/servlet2
           servlet2.tmpl    # Source for above.
        lib/                # Directory for helper classes.
           Site.py          # Site superclass ("#extends Site").
           Site.tmpl        # Source for above.
           Logic.py         # Logic class inherited by some template.
        webkit.config       # Configuration file (for WebwareExperimental).
        Webware/            # Standard Webware's MakeAppWorkDir directory.
           AppServer        # Startup program (for standard Webware).
           Configs/         # Configuration directory (for standard Webware).
               Application.config
                            # Configuration file (for standard Webware).
    site2.example.org/      # Another virtual host on this computer....
```

14.5 Initializing your template-servlet with Python code

If you need a place to initialize variables or do calculations for your template-servlet, you can put it in an `.awake()` method because WebKit automatically calls that early when processing the web transaction. If you do override `.awake()`, be sure to call the superclass `.awake` method. You probably want to do that first so that you have access to the web transaction data `Servlet.awake` provides. You don't have to worry about whether your parent class has its own `.awake` method, just call it anyway, and somebody up the inheritance chain will respond, or at minimum `Servlet.awake` will respond. Section 13.4 gives examples of how to call a superclass method.

As an alternative, you can put all your calculations in your own method and call it near the top of your template. (`#silent`, section 7.2).

14.6 Form processing

There are many ways to display and process HTML forms with Cheetah. But basically, all form processing involves two steps.

1. Display the form.
2. In the next web request, read the parameters the user submitted, check for user errors, perform any side effects (e.g., reading/writing a database or session data) and present the user an HTML response or another form.

The second step may involve choosing between several templates to fill (or several servlets to redirect to), or a big if-elif-elif-else construct to display a different portion of the template depending on the situation.

In the oldest web applications, step 1 and step 2 were handled by separate objects. Step 1 was a static HTML file, and step 2 was a CGI script. Frequently, a better strategy is to have a single servlet handle both steps. That way, the servlet has better control over the entire situation, and if the user submits unacceptable data, the servlet can redisplay the form with a ”try again” error message at the top and and all the previous input filled in. The servlet can use the presence or absence of certain CGI parameters (e.g., the submit button, or a hidden mode field) to determine which step to take.

One neat way to build a servlet that can handle both the form displaying and form processing is like this:

1. Put your form HTML into an ordinary template-servlet. In each input field, use a placeholder for the value of the `VALUE=` attribue. Place another placeholder next to each field, for that field’s error message.
2. Above the form, put a `$processFormData` method call.
3. Define that method in a Python class your template `#extend`s. (Or if it’s a simple method, you can define it in a `#def`.) The method should:
   (a) Get the form input if any.
   (b) If the input variable corresponding to the submit field is empty, there is no form input, so we’re showing the form for the first time. Initialize all VALUE= variables to their default value (usually ””), and all error variables to ””. Return ””, which will be the value for `$processFormData`.
   (c) If the submit variable is not empty, fill the VALUE= variables with the input data the user just submitted.
   (d) Now check the input for errors and put error messages in the error placeholders.
   (e) If there were any user errors, return a general error message string; this will be the value for `$processFormData`.
   (f) If there were no errors, do whatever the form’s job is (e.g., update a database) and return a success message; this will be the value for `$processFormData`.
4. The top of the page will show your success/failure message (or nothing the first time around), with the form below. If there are errors, the user will have a chance to correct them. After a successful submit, the form will appear again, so the user can either review their entry, or change it and submit it again. Depending on the application, this may make the servlet update the same database record again, or it may generate a new record.

`FunFormKit` is a third-party Webware package that makes it easier to produce forms and handle their logic. It has been successfully been used with Cheetah. You can download FunFormKit from http://colorstudy.net/software/funformkit/ and try it out for yourself.

14.7 Form input, cookies, session variables and web server variables

General variable tips that also apply to servlets are in section 13.1.

To look up a CGI GET or POST parameter (with POST overriding):

```
$request.field('myField')
self.request().field('myField')
```

These will fail if Webware is not available, because `$request` (aka `self.request()` will be `None` rather than a Webware `WebKit.Request` object. If you plan to read a lot of CGI parameters, you may want to put the `.fields` method into a local variable for convenience:

```
#set $fields = $request.fields
$fields.myField
```

But remember to do complicated calculations in Python, and assign the results to simple variables in the searchList for display. These `$request` forms are useful only for occasions where you just need one or two simple request items that going to Python for would be overkill.

To get a cookie or session parameter, subsitute "cookie" or "session" for "field" above. To get a dictionary of all CGI parameters, substitute "fields" (ditto for "cookies"). To verify a field exists, substitute "hasField" (ditto for "hasCookie").

Other useful request goodies:

```
## Defined in WebKit.Request
$request.field('myField', 'default value')
$request.time              ## Time this request began in Unix ticks.
$request.timeStamp         ## Time in human-readable format ('asctime' format).
## Defined in WebKit.HTTPRequest
$request.hasField.myField  ## Is a CGI parameter defined?
$request.fields            ## Dictionary of all CGI parameters.
$request.cookie.myCookie   ## A cookie parameter (also .hasCookie, .cookies).
$request.value.myValue     ## A field or cookie variable (field overrides)
                           ## (also .hasValue).
$request.session.mySessionVar  # A session variable.
$request.extraURLPath      ## URL path components to right of servlet, if any.
$request.serverDictionary  ## Dict of environmental vars from web server.
$request.remoteUser        ## Authenticated username.  HTTPRequest.py source
                           ## suggests this is broken and always returns None.
$request.remoteAddress  ## User's IP address (string).
$request.remoteName     ## User's domain name, or IP address if none.
$request.urlPath        ## URI of this servlet.
$request.urlPathDir     ## URI of the directory containing this servlet.
$request.serverSidePath ## Absolute path of this servlet on local filesystem.
$request.serverURL      ## URL of this servlet, without "http://" prefix,
                        ## extra path info or query string.
$request.serverURLDir   ## URL of this servlet's directory, without "http://".
$log("message")         ## Put a message in the Webware server log.  (If you
                        ## define your own 'log' variable, it will override
                        ## this; use $self.log("message") in that case.
```

.webInput()

From the method docstring:

```
    def webInput(self, names, namesMulti=(), default='', src='f',
        defaultInt=0, defaultFloat=0.00, badInt=0, badFloat=0.00, debug=False):

This method places the specified GET/POST fields, cookies or session variables
into a dictionary, which is both returned and put at the beginning of the
searchList.  It handles:
    * single vs multiple values
    * conversion to integer or float for specified names
    * default values/exceptions for missing or bad values
    * printing a snapshot of all values retrieved for debugging
All the 'default*' and 'bad*' arguments have "use or raise" behavior, meaning
that if they're a subclass of Exception, they're raised.  If they're anything
else, that value is substituted for the missing/bad value.

The simplest usage is:

    #silent $webInput(['choice'])
    $choice

    dic = self.webInput(['choice'])
    write(dic['choice'])

Both these examples retrieves the GET/POST field 'choice' and print it.  If you
leave off the "#silent", all the values would be printed too.  But a better way
to preview the values is

    #silent $webInput(['name'], $debug=1)

because this pretty-prints all the values inside HTML <PRE> tags.

Since we didn't specify any coversions, the value is a string.  It's a "single"
value because we specified it in 'names' rather than 'namesMulti'.  Single
values work like this:
    * If one value is found, take it.
    * If several values are found, choose one arbitrarily and ignore the rest.
    * If no values are found, use or raise the appropriate 'default*' value.

Multi values work like this:
    * If one value is found, put it in a list.
    * If several values are found, leave them in a list.
    * If no values are found, use the empty list ([]).  The 'default*'
      arguments are *not* consulted in this case.

Example: assume 'days' came from a set of checkboxes or a multiple combo box
on a form, and the user chose "Monday", "Tuesday" and "Thursday".

    #silent $webInput([], ['days'])
    The days you chose are: #slurp
    #for $day in $days
    $day #slurp
    #end for

    dic = self.webInput([], ['days'])
    write("The days you chose are: ")
    for day in dic['days']:
        write(day + " ")

Both these examples print:  "The days you chose are: Monday Tuesday Thursday".

By default, missing strings are replaced by "" and missing/bad numbers by zero.
(A "bad number" means the converter raised an exception for it, usually because
of non-numeric characters in the value.)  This mimics Perl/PHP behavior, and
simplifies coding for many applications where missing/bad values *should* be
blank/zero.  In those relatively few cases where you must distinguish between
""/zero on the one hand and missing/bad on the other, change the appropriate
```

14.8 More examples

Example A – a standalone servlet

Example B – a servlet under a site framework

Example C – several servlets with a common template

14.9 Other Tips

If your servlet accesses external files (e.g., via an `#include` directive), remember that the current directory is not necessarily directory the servlet is in. It's probably some other directory WebKit chose. To find a file relative to the servlet's directory, prefix the path with whatever `self.serverSidePath()` returns (from `Servlet.serverSidePath()`.

If you don't understand how `#extends` and `#implements` work, and about a template's main method, read the chapter on inheritance (sections 8.2 and 8.3). This may help you avoid buggy servlets.

15 non-Webware HTML output

Cheetah can be used with all types of HTML output, not just with Webware.

15.1 Static HTML Pages

Some sites like Linux Gazette (http://www.linuxgazette.com/) require completely static pages because they are mirrored on servers running completely different software from the main site. Even dynamic sites may have one or two pages that are static for whatever reason, and the site administrator may wish to generate those pages from Cheetah templates.

There's nothing special here. Just create your templates as usual. Then compile and fill them whenever the template definition changes, and fill them again whenever the placeholder values change. You may need an extra step to copy the .html files to their final location. A Makefile (chapter 13.8) can help encapsulate these steps.

15.2 CGI scripts

Unlike Webware servlets, which don't have to worry about the HTTP headers, CGI scripts must emit their own headers. To make a template CGI aware, add this at the top:

```
#extends Cheetah.Tools.CGITemplate
#implements respond
$cgiHeaders#slurp
```

Or if your template is inheriting from a Python class:

```
#extends MyPythonClass
#implements respond
$cgiHeaders#slurp
```

A sample Python class:

```
from Cheetah.Tools import CGITemplate
class MyPythonClass(CGITemplate):
    def cgiHeadersHook(self):
        return "Content-Type: text/html; charset=koi8-r\n\n"
```

Compile the template as usual, put the .py template module in your cgi-bin directory and give it execute permission. `.cgiHeaders()` is a "smart" method that outputs the headers if the module is called as a CGI script, or outputs nothing if not. Being "called as a CGI script" means the environmental variable `REQUEST_METHOD` exists and `self.isControlledByWebKit` is false. If you don't agree with that definition, override `.isCgi()` and provide your own.

The default header is a simple `Content-type: text/html\n\n`, which works with all CGI scripts. If you want to customize the headers (e.g., to specify the character set), override `.cgiHeadersHook()` and return a string containing all the headers. Don't forget to include the extra newline at the end of the string: the HTTP protocol requires this empty line to mark the end of the headers.

To read GET/POST variables from form input, use the `.webInput()` method (section 14.7), or extract them yourself using Python's `cgi` module or your own function. Although `.webInput()` was originally written for Webware servlets, it now handles CGI scripts too. There are a couple behavioral differences between CGI scripts and Webware

servlets regarding input variables:

1. CGI scripts, using Python's `cgi` module, believe `REQUEST_METHOD` and recognize *either* GET variables *or* POST variables, not both. Webware servlets, doing additional processing, ignore `REQUEST_METHOD` and recognize both, like PHP does.
2. Webware servlets can ask for cookies or session variables instead of GET/POST variables, by passing the argument `src='c'` or `src='s'`. CGI scripts get a `RuntimeError` if they try to do this.

If you keep your .tmpl files in the same directory as your CGI scripts, make sure they don't have execute permission. Apache at least refuses to serve files in a `ScriptAlias` directory that don't have execute permission.

16 Non-HTML Output

Cheetah can also output any other text format besides HTML.

16.1 Python source code

To be written. We're in the middle of working on an autoindenter to make it easier to encode Python indentation in a Cheetah template.

17 Batteries included: templates and other libraries

Cheetah comes "batteries included" with libraries of templates, functions, classes and other objects you can use in your own programs. The different types are listed alphabetically below, followed by a longer description of the SkeletonPage framework. Some of the objects are classes for specific purposes (e.g., filters or error catchers), while others are standalone and can be used without Cheetah.

If you develop any objects which are generally useful for Cheetah sites, please consider posting them on the wiki with an announcement on the mailing list so we can incorporate them into the standard library. That way, all Cheetah users will benefit, and it will encourage others to contribute their objects, which might include something you want.

17.1 ErrorCatchers

Module `Cheetah.ErrorCatchers` contains error-handling classes suitable for the `#errorCatcher` directive. These are debugging tools that are not intended for use in production systems. See section 10.2 for a description of the error catchers bundled with Cheetah.

17.2 FileUtils

Module `Cheetah.FileUtils` contains generic functions and classes for doing bulk search-and-replace on several files, and for finding all the files in a directory hierarchy whose names match a glob pattern.

17.3 Filters

Module `Filters` contains filters suitable for the `#Filter` directive. See section 7.9 for a description of the filters bundled with Cheetah.

17.4 SettingsManager

The `SettingsManager` class in the `Cheetah.SettingsManager` module is a baseclass that provides facilities for managing application settings. It facilitates the use of user-supplied configuration files to fine tune an application. A setting is a key/value pair that an application or component (e.g., a filter, or your own servlets) looks up and treats as a configuration value to modify its (the component's) behaviour.

SettingsManager is designed to:

- work well with nested settings dictionaries of any depth
- read/write `.ini style config files` (or strings)
- read settings from Python source files (or strings) so that complex Python objects can be stored in the application's settings dictionary. For example, you might want to store references to various classes that are used by the application, and plugins to the application might want to substitute one class for another.
- allow sections in `.ini config files` to be extended by settings in Python src files. If a section contains a setting like "`importSettings=mySettings.py`", `SettingsManager` will merge all the settings defined in "`mySettings.py`" with the settings for that section that are defined in the `.ini config file`.
- maintain the case of setting names, unlike the ConfigParser module

Cheetah uses `SettingsManager` to manage its configuration settings. `SettingsManager` might also be useful in your own applications. See the source code and docstrings in the file `src/SettingsManager.py` for more information.

17.5 Templates

Package `Cheetah.Templates` contains stock templates that you can either use as is, or extend by using the `#def` directive to redefine specific **blocks**. Currently, the only template in here is SkeletonPage, which is described in detail below in section 17.7. (Contributed by Tavis Rudd.)

17.6 Tools

Package `Cheetah.Tools` contains functions and classes contributed by third parties. Some are Cheetah-specific but others are generic and can be used standalone. None of them are imported by any other Cheetah component; you can delete the Tools/ directory and Cheetah will function fine.

Some of the items in Tools/ are experimental and have been placed there just to see how useful they will be, and whether they attract enough users to make refining them worthwhile (the tools, not the users :).

Nothing in Tools/ is guaranteed to be: (A) tested, (B) reliable, (C) immune from being deleted in a future Cheetah version, or (D) immune from backwards-incompatable changes. If you depend on something in Tools/ on a production system, consider making a copy of it outside the Cheetah/ directory so that this version won't be lost when you upgrade Cheetah. Also, learn enough about Python and about the Tool so that you can maintain it and bugfix it if necessary.

If anything in Tools/ is found to be necessary to Cheetah's operation (i.e., if another Cheetah component starts importing it), it will be moved to the `Cheetah.Utils` package.

Current Tools include:

Cheetah.Tools.MondoReport an ambitious class useful when iterating over records of data (`#for` loops), displaying one pageful of records at a time (with previous/next links), and printing summary statistics about the records or the current page. See `MondoReportDoc.txt` in the same directory as the module. Some features are not implemented yet. `MondoReportTest.py` is a test suite (and it shows there are currently some errors in MondoReport, hmm). Contributed by Mike Orr.

Cheetah.Tools.RecursiveNull Nothing, but in a friendly way. Good for filling in for objects you want to hide. If `$form.f1` is a RecursiveNull object, then `$form.f1.anything["you"].might("use")` will resolve to the empty string. You can also put a `RecursiveNull` instance at the end of the searchList to convert missing values to " rather than raising a `NotFound` error or having a (less efficient) errorCatcher handle it. Of course, maybe you prefer to get a `NotFound` error... Contributed by Ian Bicking.

Cheetah.Tools.SiteHierarchy Provides navigational links to this page's parents and children. The constructor takes a recursive list of (url,description) pairs representing a tree of hyperlinks to every page in the site (or section, or application...), and also a string containing the current URL. Two methods 'menuList' and 'crumbs' return output-ready HTML showing an indented menu (hierarchy tree) or crumbs list (Yahoo-style bar: home ¿ grandparent ¿ parent ¿ currentURL). Contributed by Ian Bicking.

17.7 Utils

Package `Cheetah.Utils` contains non-Cheetah-specific functions and classes that are imported by other Cheetah components. Many of these utils can be used standalone in other applications too.

Current Utils include:

Cheetah.Utils.CGIImportMixin This is inherited by `Template` objects, and provides the method, `.cgiImport` method (section **??**).

Cheetah.Utils.Misc A catch-all module for small functions.

`UseOrRaise(thing, errmsg='')` Raise 'thing' if it's a subclass of Exception, otherwise return it. Useful when one argument does double duty as a default value or an exception to throw. Contribyted by Mike Orr.

`checkKeywords(dic, legalKeywords, what='argument'` Verifies the dictionary does not contain any keys not listed in 'legalKeywords'. If it does, raise TypeError. Useful for checking the keyword arguments to a function. Contributed by Mike Orr.

Cheetah.Utils.UploadFileMixin Not implemented yet, but will contain the `.uploadFile` method (or three methods) to "safely" copy a form-uploaded file to a local file, to a searchList variable, or return it. When finished, this will be inherited by `Template`, allowing all templates to do this. If you want this feature, read the docstring in the source and let us know on the mailing list what you'd like this method to do. Contributed by Mike Orr.

Cheetah.Utils.VerifyType Functions to verify the type of a user-supplied function argument. Contributed by Mike Orr.

Cheetah.Templates.SkeletonPage

A stock template class that may be useful for web developers is defined in the `Cheetah.Templates.SkeletonPage` module. The `SkeletonPage` template class is generated from the following Cheetah source code:

```
##doc-module: A Skeleton HTML page template, that provides basic structure and utility methods.
################################################################################
#extends Cheetah.Templates._SkeletonPage
#implements respond
################################################################################
#cache id='header'
$docType
$htmlTag
<!-- This document was autogenerated by Cheetah(http://CheetahTemplate.org).
Do not edit it directly!

Copyright $currentYr - $siteCopyrightName - All Rights Reserved.
Feel free to copy any javascript or html you like on this site,
provided you remove all links and/or references to $siteDomainName
However, please do not copy any content or images without permission.

$siteCredits

-->


#block writeHeadTag
<head>
<title>$title</title>
$metaTags
$stylesheetTags
$javascriptTags
</head>
#end block writeHeadTag

#end cache header
##################

$bodyTag

#block writeBody
This skeleton page has no flesh. Its body needs to be implemented.
#end block writeBody

</body>
</html>
```

You can redefine any of the blocks defined in this template by writing a new template that `#extends` SkeletonPage. (As you remember, using `#extends` makes your template implement the `.writeBody()` method instead of `.respond()` – which happens to be the same method SkeletonPage expects the page content to be (note the writeBody block in SkeletonPage).)

```
#def bodyContents
Here's my new body. I've got some flesh on my bones now.
#end def bodyContents
```

All of the $placeholders used in the `SkeletonPage` template definition are attributes or methods of the `SkeletonPage` class. You can reimplement them as you wish in your subclass. Please read the source code of the file `src/Templates/_SkeletonPage.py` before doing so.

You'll need to understand how to use the following methods of the `SkeletonPage` class: `$metaTags()`, `$stylesheetTags()`, `$javascriptTags()`, and `$bodyTag()`. They take the data you define in various attributes and renders them into HTML tags.

- **metaTags()** – Returns a formatted vesion of the self._metaTags dictionary, using the formatMetaTags function from `_SkeletonPage.py`.
- **stylesheetTags()** – Returns a formatted version of the `self._stylesheetLibs` and `self._stylesheets` dictionaries. The keys in `self._stylesheets` must be listed in the order that they should appear in the list `self._stylesheetsOrder`, to ensure that the style rules are defined in the correct order.
- **javascriptTags()** – Returns a formatted version of the `self._javascriptTags` and `self._javascriptLibs` dictionaries. Each value in `self._javascriptTags` should be a either a code string to include, or a list containing the JavaScript version number and the code string. The keys can be anything. The same applies for `self._javascriptLibs`, but the string should be the SRC filename rather than a code string.
- **bodyTag()** – Returns an HTML body tag from the entries in the dict `self._bodyTagAttribs`.

The class also provides some convenience methods that can be used as $placeholders in your template definitions:

- **imgTag(self, src, alt=", width=None, height=None, border=0)** – Dynamically generate an image tag. Cheetah will try to convert the "`src`" argument to a WebKit serverSidePath relative to the servlet's location. If width and height aren't specified they are calculated using PIL or ImageMagick if either of these tools are available. If all your images are stored in a certain directory you can reimplement this method to append that directory's path to the "`src`" argument. Doing so would also insulate your template definitions from changes in your directory structure.

18 Visual Editors

This chapter is about maintaining Cheetah templates with visual editors, and the tradeoffs between making it friendly to both text editors and visual editors.

Cheetah's main developers do not use visual editors. Tavis uses `emacs`; Mike uses `vim`. So our first priority is to make templates easy to maintain in text editors. In particular, we don't want to add features like Zope Page Template's placeholder-value-with-mock-text-for-visual-editors-all-in-an-XML-tag. The syntax is so verbose it makes for a whole lotta typing just to insert a simple placeholder, for the benefit of editors we never use. However, as users identify features which would help their visual editing without making it harder to maintain templates in a text editor, we're all for it.

As it said in the introduction, Cheetah purposely does not use HTML/XML tags for $placeholders or #directives. That way, when you preview the template in an editor that interprets HTML tags, you'll still see the placeholder and directive source definitions, which provides some "mock text" even if it's not the size the final values will be, and allows you to use your imagination to translate how the directive output will look visually in the final.

If your editor has syntax highlighting, turn it on. That makes a big difference in terms of making the template easier to edit. Since no "Cheetah mode" has been invented yet, set your highlighting to Perl mode, and at least the directives/placeholders will show up in different colors, although the editor won't reliably guess where the directive/placeholder ends and normal text begins.

A Useful Web Links

See the wiki for more links. (The wiki is also updated more often than this chapter is.)

A.1 Cheetah Links

Home Page – http:www.CheetahTemplate.org/

On-line Documentation – http:www.CheetahTemplate.org/learn.html

SourceForge Project Page – http:sf.net/projects/cheetahtemplate/

Mailing List Subscription Page – http://lists.sourceforge.net/lists/listinfo/cheetahtemplate-discuss

Mailing List Archive @ Geocrawler – http://www.geocrawler.com/lists/3/SourceForge/12986/0/

Mailing List Archive @ Yahoo – http://groups.yahoo.com/group/cheetah-archive/

CVS Repository – http://sourceforge.net/cvs/?group_id=28961

CVS-commits archive – http://www.geocrawler.com/lists/3/SourceForge/13091/0/

A.2 Third-party Cheetah Stuff

- Steve Howell has written a photo viewer using Python. http://mountainwebtools.com/PicViewer/install.htm

A.3 Webware Links

Home Page – http://webware.sf.net/

On-line Documentation – http://webware.sf.net/Webware/Docs/

SourceForge Project Page – http://sf.net/projects/webware/

Mailing List Subscription Page – http://lists.sourceforge.net/lists/listinfo/webware-discuss

A.4 Python Links

Home Page – http://www.python.org/

On-line Documentation – http://www.python.org/doc/

SourceForge Project Page – http://sf.net/projects/python/

The Vaults of Parnassus: Python Resources – http://www.vex.net/parnassus/

Python Cookbook – http://aspn.activestate.com/ASPN/Cookbook/Python

A.5 Other Useful Links

Python Database Modules and Open Source Databases

Python Database Topic Guide – http://python.org/topics/database/

PostgreSQL Database – http://www.postgresql.org/index.html

MySQL Database – http://www.mysql.com/

A comparison of PostgreSQL and MySQL – http://phpbuilder.com/columns/tim20001112.php3

Other Template Systems

Chuck's "Templates" Summary Page – http://webware.sf.net/Papers/Templates/

Other Internet development frameworks

ZOPE (Z Object Publishing Environment) – http://zope.org/

Server Side Java – http://jakarta.apache.org/

PHP – http://php.net/

IBM Websphere – http://www.ibm.com/websphere/

Coldfusion and Spectra – http://www.macromedia.com/

B Examples

The Cheetah distribution comes with an 'examples' directory. Browse the files in this directory and its subdirectories for examples of how Cheetah can be used.

B.1 Syntax examples

The `Cheetah.Tests` module contains a large number of test cases that can double as examples of how the Cheetah Language works. To view these cases go to the base directory of your Cheetah distribution and open the file `Cheetah/Tests/SyntaxAndOutput.py` in a text editor.

B.2 Webware Examples

For examples of Cheetah in use with Webware visit the Cheetah and Webware wikis or use google. We used to have more examples in the cheetah source tarball, but they were out of date and confused people.

C Cheetah vs. Other Template Engines

This appendix compares Cheetah with various other template/emdedded scripting languages and Internet development frameworks. As Cheetah is similar to Velocity at a superficial level, you may also wish to read comparisons between Velocity and other languages at http://jakarta.apache.org/velocity/ymtd/ymtd.html.

C.1 Which features are unique to Cheetah

- The **block framework** (section 8.8)
- Cheetah's powerful yet simple **caching framework** (section 7.4)
- Cheetah's **Unified Dotted Notation** and **autocalling** (sections 5.5 and 5.5)
- Cheetah's searchList (section 5.6) information.
- Cheetah's `#raw` directive (section 7.5)
- Cheetah's `#slurp` directive (section 7.7)
- Cheetah's tight integration with Webware for Python (section 14)
- Cheetah's **SkeletonPage framework** (section 17.7)
- Cheetah's ability to mix PSP-style code with Cheetah Language syntax (section 13.7) Because of Cheetah's design and Python's flexibility it is relatively easy to extend Cheetah's syntax with syntax elements from almost any other template or embedded scripting language.

C.2 Cheetah vs. Velocity

For a basic introduction to Velocity, visit http://jakarta.apache.org/velocity.

Velocity is a Java template engine. It's older than Cheetah, has a larger user base, and has better examples and docs at the moment. Cheetah, however, has a number of advantages over Velocity:

- Cheetah is written in Python. Thus, it's easier to use and extend.
- Cheetah's syntax is closer to Python's syntax than Velocity's is to Java's.
- Cheetah has a powerful caching mechanism. Velocity has no equivalent.
- It's far easier to add data/objects into the namespace where $placeholder values are extracted from in Cheetah. Velocity calls this namespace a 'context'. Contexts are dictionaries/hashtables. You can put anything you want into a context, BUT you have to use the .put() method to populate the context; e.g.,

  ```
  VelocityContext context1 = new VelocityContext();
  context1.put("name","Velocity");
  context1.put("project", "Jakarta");
  context1.put("duplicate", "I am in context1");
  ```

  Cheetah takes a different approach. Rather than require you to manually populate the 'namespace' like Velocity, Cheetah will accept any existing Python object or dictionary AS the 'namespace'. Furthermore, Cheetah allows you to specify a list namespaces that will be searched in sequence to find a varname-to-value mapping. This searchList can be extended at run-time.

If you add a ‘foo’ object to the searchList and the ‘foo’ has an attribute called ’bar’, you can simply type `$bar` in the template. If the second item in the searchList is dictionary ’foofoo’ containing `{'spam':1234, 'parrot':666}`, Cheetah will first look in the ‘foo’ object for a ‘spam’ attribute. Not finding it, Cheetah will then go to ‘foofoo’ (the second element in the searchList) and look among its dictionary keys for ‘spam’. Finding it, Cheetah will select `foofoo['spam']` as `$spam`’s value.

- In Cheetah, the tokens that are used to signal the start of $placeholders and #directives are configurable. You can set them to any character sequences, not just $ and #.

C.3 Cheetah vs. WebMacro

For a basic introduction to WebMacro, visit http://webmacro.org.

The points discussed in section C.2 also apply to the comparison between Cheetah and WebMacro. For further differences please refer to http://jakarta.apache.org/velocity/differences.html.

C.4 Cheetah vs. Zope’s DTML

For a basic introduction to DTML, visit http://www.zope.org/Members/michel/ZB/DTML.dtml.

- Cheetah is faster than DTML.
- Cheetah does not use HTML-style tags; DTML does. Thus, Cheetah tags are visible in rendered HTML output if something goes wrong.
- DTML can only be used with ZOPE for web development; Cheetah can be used as a standalone tool for any purpose.
- Cheetah’s documentation is more complete than DTML’s.
- Cheetah’s learning curve is shorter than DTML’s.
- DTML has no equivalent of Cheetah’s blocks, caching framework, unified dotted notation, and `#raw` directive.

Here are some examples of syntax differences between DTML and Cheetah:

```
<ul>
<dtml-in frogQuery>
 <li><dtml-var animal_name></li>
</dtml-in>
</ul>
```

```
<ul>
#for $animal_name in $frogQuery
 <li>$animal_name</li>
#end for
</ul>
```

```
<dtml-if expr="monkeys > monkey_limit">
  <p>There are too many monkeys!</p>
<dtml-elif expr="monkeys < minimum_monkeys">
  <p>There aren't enough monkeys!</p>
<dtml-else>
  <p>There are just enough monkeys.</p>
</dtml-if>
```

```
#if $monkeys > $monkey_limit
  <p>There are too many monkeys!</p>
#else if $monkeys < $minimum_monkeys
  <p>There aren't enough monkeys!</p>
#else
  <p>There are just enough monkeys.</p>
#end if
```

```
<table>
<dtml-in expr="objectValues('File')">
  <dtml-if sequence-even>
    <tr bgcolor="grey">
  <dtml-else>
    <tr>
  </dtml-if>
  <td>
  <a href="&dtml-absolute_url;"><dtml-var title_or_id></a>
  </td></tr>
</dtml-in>
</table>
```

```
<table>
#set $evenRow = 0
#for $file in $files('File')
  #if $evenRow
    <tr bgcolor="grey">
    #set $evenRow = 0
  #else
    <tr>
    #set $evenRow = 1
  #end if
  <td>
  <a href="$file.absolute_url">$file.title_or_id</a>
  </td></tr>
#end for
</table>
```

The last example changed the name of `$objectValues` to `$files` because that's what a Cheetah developer would write. The developer would be responsible for ensuring `$files` returned a list (or tuple) of objects (or dictionaries) containing the attributes (or methods or dictionary keys) 'absolute_url' and 'title_or_id'. All these names ('objectValues', 'absolute_url' and 'title_or_id') are standard parts of Zope, but in Cheetah the developer is in charge of writing them and giving them a reasonable behaviour.

Some of DTML's features are being ported to Cheetah, such as `Cheetah.Tools.MondoReport`, which is based on the `<dtml-in>` tag. We are also planning an output filter as flexible as the `<dtml-var>` formatting options. However, neither of these are complete yet.

C.5 Cheetah vs. Zope Page Templates

For a basic introduction to Zope Page Templates, please visit http://www.zope.org/Documentation/Articles/ZPT2.

C.6 Cheetah vs. PHP's Smarty templates

PHP (http://www.php.net/) is one of the few scripting languages expressly designed for web servlets. However, it's also a full-fledged programming language with libraries similar to Python's and Perl's. The syntax and functions are like a cross between Perl and C plus some original ideas (e.g.; a single array type serves as both a list and a dictionary, `$arr[]="value";` appends to an array).

Smarty (http://smarty.php.net/) is an advanced template engine for PHP. (*Note:* this comparision is based on Smarty's on-line documentation. The author has not used Smarty. Please send corrections or ommissions to the Cheetah mailing list.) Like Cheetah, Smarty:

- compiles to the target programming language (PHP).
- has configurable delimeters.
- passes if-blocks directly to PHP, so you can use any PHP expression in them.
- allows you to embed PHP code in a template.
- has a caching framework (although it works quite differently).
- can read the template definition from any arbitrary source.

Features Smarty has that Cheetah lacks:

- Preprocessors, postprocessors and output filters. You can emulate a preprocessor in Cheetah by running your template definition through a filter program or function before Cheetah sees it. To emulate a postprocessor, run a .py template module through a filter program/function. To emulate a Smarty output filter, run the template output through a filter program/function. If you want to use "cheetah compile" or "cheetah fill" in a pipeline, use `-` as the input file name and `--stdout` to send the result to standard output. Note that Cheetah uses the term "output filter" differently than Smarty: Cheetah output filters (`#filter`) operate on placeholders, while Smarty output filters operate on the entire template output. There has been a proposed `#sed` directive that would operate on the entire output line by line, but it has not been implemented.
- Variable modifiers. In some cases, Python has equivalent string methods (`.strip`, `.capitalize`, `.replace(SEARCH, REPL)`), but in other cases you must wrap the result in a function call or write a custom output filter (`#filter`).
- Certain web-specific functions, which can be emulated with third-party functions.
- The ability to "plug in" new directives in a modular way. Cheetah directives are tightly bound to the compiler. However, third-party *functions* can be freely imported and called from placeholders, and *methods* can be mixed in via `#extends`. Part of this is because Cheetah distinguishes between functions and directives, while Smarty treats them all as "functions". Cheetah's design does not allow functions to have flow control effect outside the function (e.g., `#if` and `#for`, which operate on template body lines), so directives like these cannot be encoded as functions.

- Configuration variables read from an .ini-style file. The `Cheetah.SettingsManager` module can parse such a file, but you'd have to invoke it manually. (See the docstrings in the module for details.) In Smarty, this feature is used for multilingual applications. In Cheetah, the developers maintain that everybody has their own preferred way to do this (such as using Python's `gettext` module), and it's not worth blessing one particular strategy in Cheetah since it's easy enough to integrate third-party code around the template, or to add the resulting values to the searchList.

Features Cheetah has that Smarty lacks:

- Saving the compilation result in a Python (PHP) module for quick reading later.
- Caching individual placeholders or portions of a template. Smarty caches only the entire template output as a unit.

Comparisions of various Smarty constructs:

```
{assign var="name" value="Bob"} (#set has better syntax in the author's opinion)
counter   (looks like equivalent to #for)
eval      (same as #include with variable)
fetch: insert file content into output   (#include raw)
fetch: insert URL content into output    (no euqivalent, user can write
     function calling urllib, call as $fetchURL('URL') )
fetch: read file into variable  (no equivalent, user can write function
     based on the 'open/file' builtin, or on .getFileContents() in
     Template.)
fetch: read URL content into variable  (no equivalent, use above
     function and call as:  #set $var = $fetchURL('URL')
html_options: output an HTML option list  (no equivalent, user can
     write custom function.  Maybe FunFormKit can help.)
html_select_date: output three dropdown controls to specify a date
     (no equivalent, user can write custom function)
html_select_time: output four dropdown controls to specify a time
     (no equvalent, user can write custom function)
math: eval calculation and output result   (same as #echo)
math: eval calculation and assign to variable  (same as #set)
popup_init: library for popup windows  (no equivalent, user can write
     custom method outputting Javascript)


Other commands:
capture   (no equivalent, collects output into variable.  A Python
     program would create a StringIO instance, set sys.stdout to
     it temporarily, print the output, set sys.stdout back, then use
     .getvalue() to get the result.)
config_load   (roughly analagous to #settings, which was removed
     from Cheetah.  Use Cheetah.SettingsManager manually or write
     a custom function.)
include   (same as #include, but can include into variable.
     Variables are apparently shared between parent and child.)
include_php: include a PHP script (e.g., functions)
     (use #extends or #import instead)
insert   (same as #include not in a #cache region)
{ldelim}{rdelim}   (escape literal $ and # with a backslash,
     use #compiler-settings to change the delimeters)
literal  (#raw)
php    (''<% %>'' tags)
section  (#for $i in $range(...) )
foreach  (#for)
strip   (like the #sed tag which was never implemented.  Strips
     leading/trailing whitespace from lines, joins several lines
     together.)


Variable modifiers:
capitalize    ( $STRING.capitalize() )
count_characters    (   $len(STRING)  )
count_paragraphs/sentances/words   (no equivalent, user can write function)
date_format    (use 'time' module or download Egenix's mx.DateTime)
default    ($getVar('varName', 'default value') )
escape: html encode    ($cgi.escape(VALUE) )
escape: url encode    ($urllib.quote_plus(VALUE) )
escape: hex encode   (no equivalent?  user can write function)
escape: hex entity encode  (no equivalent?  user can write function)
indent: indent all lines of a var's output  (may be part of future
     #indent directive)
lower    ($STRING.lower() )
regex_replace  (not my template)
replace    ($STRING.replace(OLD, NEW, MAXSPLIT) )
spacify   (#echo "SEPARATOR".join(SEQUENCE) )
string_format   (#echo "%.2f" % FLOAT , etc.)
strip_tags  (no equivalent, user can write function to strip HTML tags
```

Some of these modifiers could be added to the super output filter we want to write someday.

C.7 Cheetah vs. PHPLib's Template class

PHPLib ((http://phplib.netuse.de/) is a collection of classes for various web objects (authentication, shopping cart, sessions, etc), but what we're interested in is the `Template` object. It's much more primitive than Smarty, and was based on an old Perl template class. In fact, one of the precursors to Cheetah was based on it too. Differences from Cheetah:

- Templates consist of text with `{placeholders}` in braces.
- Instead of a searchList, there is one flat namespace. Every variable must be assigned via the `set_var` method. However, you can pass this method an array (dictionary) of several variables at once.
- You cannot embed lookups or calculations into the template. Every placeholder must be an exact variable name.
- There are no directives. You must do all display logic (if, for, etc) in the calling routine.
- There is, however, a "block" construct. A block is a portion of text between the comment markers `<!-- BEGIN blockName --> ...<!-- END blockName>`. The `set_block` method extracts this text into a namespace variable and puts a placeholder referring to it in the template. This has a few parallels with Cheetah's `#block` directive but is overall quite different.
- To do the equivalent of `#if`, extract the block. Then if true, do nothing. If false, assign the empty string to the namespace variable.
- To do the equivalent of `#for`, extract the block. Set any namespace variables needed inside the loop. To parse one iteration, use the `parse` method to fill the block variable (a mini-template) into another namespace variable, appending to it. Refresh the namespace variables needed inside the loop and parse again; repeat for each iteration. You'll end up with a mini-result that will be plugged into the main template's placeholder.
- To read a template definition from a file, use the `set_file` method. This places the file's content in a namespace variable. To read a template definition from a string, assign it to a namespace variable.
- Thus, for complicated templates, you are doing a lot of recursive block filling and file reading and parsing mini-templates all into one flat namespace as you finally build up values for the main template. In Cheetah, all this display logic can be embedded into the template using directives, calling out to Python methods for the more complicated tasks.
- Although you can nest blocks in the template, it becomes tedious and arguably hard to read, because all blocks have identical syntax. Unless you choose your block names carefully and put comments around them, it's hard to tell which blocks are if-blocks and which are for-blocks, or what their nesting order is.
- PHPLib templates do not have caching, output filters, etc.

C.8 Cheetah vs. PSP, PHP, ASP, JSP, Embperl, etc.

Webware's PSP Component – http://webware.sourceforge.net/Webware/PSP/Docs/

Tomcat JSP Information – http://jakarta.apache.org/tomcat/index.html

ASP Information at ASP101 – http://www.asp101.com/

Embperl – http://perl.apache.org/embperl/

Here’s a basic Cheetah example:

```
<TABLE>
#for $client in $service.clients
<TR>
<TD>$client.surname, $client.firstname</TD>
<TD><A HREF="mailto:$client.email" >$client.email</A></TD>
</TR>
#end for
</TABLE>
```

Compare this with PSP:

```
<TABLE>
<% for client in service.clients(): %>
<TR>
<TD><%=client.surname()%>, <%=client.firstname()%></TD>
<TD><A HREF="mailto:<%=client.email()%>"><%=client.email()%></A></TD>
</TR>
<%end%>
</TABLE>
```

D Optik license

The optik package (Cheetah.Utils.optik) is based on Optik 1.3, http://optik.sourceforge.net/, ©2001 Gregory P Ward <gward@python.net>. It's unmodified from the original version except the `import` statements, which have been adjusted to make them work in this location. The following license applies to optik:

> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
> - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
> - Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.