Cheetah Developers' Guide

Release 0.9.15a1

Mike Orr with assistance from Tavis Rudd

October 6, 2002

iron@mso.oz.net

Contents

©Copyright 2002, Mike Orr. This document may be copied and modified under the terms of the **Open Publication License** http://www.opencontent.org/openpub/

1 Introduction

1.1 Who should read this Guide?

The Cheetah Developers' Guide is for those who want to learn how Cheetah works internally, or wish to modify or extend Cheetah. It is assumed that you've read the Cheetah Users' Guide and have an intermediate knowledge of Python.

1.2 Contents

This Guide takes a behaviorist approach. First we'll look at what the Cheetah compiler generates when it compiles a template definition, and how it compiles the various $placeholder features and #directives. Then we'll stroll through the files in the Cheetah source distribution and show how each file contributes to the compilation and/or filling of templates. Then we'll list every method/attribute inherited by a template object. Finally, we'll describe how to submit bugfixes/enhancements to Cheetah, and how to add to the documentation.

Appendix A will contain a BNF syntax of the Cheetah template language.

2 .py Template Modules

This chapter examines the structure of a .py template module. The following few chapters will then show how each placeholder and directive affects the generated Python code.

2.1 An example

Our first template follows a long noble tradition in computer tutorials. It produces a familiar, friendly greeting. Here's the template:

```
Hello, world!
```

... the output:

```
Hello, world!
```

... and the .py template module cheetah-compile produced, with line numbers added:

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

 2 """
 3 Autogenerated by CHEETAH: The Python-Powered Template Engine
 4  CHEETAH VERSION: 0.9.12
 5  Generation time: Sat Apr 20 14:27:47 2002
 6    Source file: x.tmpl
 7    Source file last modified: Wed Apr 17 22:10:59 2002
 8 """

 9 __CHEETAH_genTime__ = 'Sat Apr 20 14:27:47 2002'
10 __CHEETAH_src__ = 'x.tmpl'
11 __CHEETAH_version__ = '0.9.12'

12 ##################################################
13 ## DEPENDENCIES

14 import sys
15 import os
16 import os.path
17 from os.path import getmtime, exists
18 import time
19 import types
20 from Cheetah.Template import Template
21 from Cheetah.DummyTransaction import DummyTransaction
22 from Cheetah.NameMapper import NotFound, valueForName,
          valueFromSearchList
23 import Cheetah.Filters as Filters
24 import Cheetah.ErrorCatchers as ErrorCatchers

25 ##################################################
26 ## MODULE CONSTANTS

27 try:
28     True, False
29 except NameError:
30     True, False = (1==1), (1==0)

31 ##################################################
32 ## CLASSES

33 class x(Template):
34     """
35
36     Autogenerated by CHEETAH: The Python-Powered Template Engine
37     """
```

```
38      ##################################################
39      ## GENERATED METHODS


40      def __init__(self, *args, **KWs):
41          """
42
43          """

44          Template.__init__(self, *args, **KWs)
45          self._filePath = 'x.tmpl'
46          self._fileMtime = 1019106659

47      def respond(self,
48              trans=None,
49              dummyTrans=False,
50              VFS=valueFromSearchList,
51              VFN=valueForName,
52              getmtime=getmtime,
53              currentTime=time.time):


54          """
55          This is the main method generated by Cheetah
56          """

57          if not trans:
58              trans = DummyTransaction()
59              dummyTrans = True
60          write = trans.response().write
61          SL = self._searchList
62          filter = self._currentFilter
63          globalSetVars = self._globalSetVars
64
65          ########################################
66          ## START - generated method body
67
68          if exists(self._filePath) and getmtime(self._filePath) > \
                    self._fileMtime:
69              self.compile(file=self._filePath)
70              write(getattr(self, self._mainCheetahMethod_for_x)
                        (trans=trans))
71              if dummyTrans:
72                  return trans.response().getvalue()
73              else:
74                  return ""
75          write('Hello, world!\n')
76
77          ########################################
78          ## END - generated method body
79
80          if dummyTrans:
81              return trans.response().getvalue()
82          else:
83              return ""
```

```
84
85      ##################################################
86      ## GENERATED ATTRIBUTES


87      __str__ = respond

88      _mainCheetahMethod_for_x= 'respond'


89  # CHEETAH was developed by Tavis Rudd, Chuck Esterbrook, Ian Bicking
        #     and Mike Orr;
90  # with code, advice and input from many other volunteers.
91  # For more information visit http://www.CheetahTemplate.org

92  ##################################################
93  ## if run from command line:
94  if __name__ == '__main__':
95      x().runAsMainProgram()
```

(I added the line numbers for this Guide, and split a few lines to fit the page width. The continuation lines don't have line numbers, and I added indentation, backslashes and '#' as necessary to make the result a valid Python program.)

The examples were generated from CVS versions of Cheetah between 0.9.12 and 0.9.14.

2.2 A walk through the example

Lines 20-24 are the Cheetah-specific imports. Line 33 introduces our generated class, `x`, a subclass of `Template`. It's called x because the source file was x.tmpl.

Lines 40-46 are the `._init__` method called when the template is instantiated or used as a Webware servlet, or when the module is run as a standalone program. We can see it calling its superclass constructor and setting `._filePath` and `._fileMtime` to the filename and modification time (in Unix ticks) of the source .tmpl file.

Lines 47-84 are the main method `.respond`, the one that fills the template. Normally you call it without arguments, but Webware calls it with a Webware `Transaction` object representing the current request. Lines 57-59 set up the `trans` variable. If a real or dummy transaction is passed in, the method uses it. Otherwise (if the `trans` argument is `None`), the method creates a `DummyTransaction` instance. `dummyTrans` is a flag that just tells whether a dummy transaction is in effect; it'll be used at the end of the method.

The other four `.respond` arguments aren't anything you'd ever want to pass in; they exist solely to speed up access to these frequently-used global functions. This is a standard Python trick described in question 4.7 of the Python FAQ (http://www.python.org/cgi-bin/faqw.py). `VFS` and `VFN` are the functions that give your template the benefits of NameMapper lookup, such as the ability to use the searchList.

Line 60 initializes the `write` variable. This important variable is discussed below.

Lines 60-63 initialize a few more local variables. `SL` is the searchList. `filter` is the current output filter. `globalSetVars` are the variables that have been defined with `#set global`.

The comments at lines 65 and 78 delimit the start and end of the code that varies with each template. The code outside this region is identical in all template modules. That's not quite true – `#import` for instance generates additional `import` statements at the top of the module – but it's true enough for the most part.

Lines 68-74 exist only if the template source was a named file rather than a string or file object. The stanza recompiles the template if the source file has changed. Lines 70-74 seem to be redundant with 75-83: both fill the template and

send the output. The reason the first set of lines exists is because the second set may become invalid when the template is recompiled. (This is for *re* compilation only. The initial compilation happened in the `.__init__` method if the template wasn't precompiled.)

Line 75 is the most interesting line in this module. It's a direct translation of what we put in the template definition, "Hello, world!" Here the content is a single string literal. `write` looks like an ordinary function call, but remember that line 60 made it an alias to `trans.response().write`, a method in the transaction. The next few chapters describe how the different placeholders and directives influence this portion of the generated class.

Lines 80-83 finish the template filling. If `trans` is a real Webware transaction, `write` has already sent the output to Webware for handling, so we return `""`. If `trans` is a dummy transaction, `write` has been accumulating the output in a Python `StringIO` object rather than sending it anywhere, so we have to return it.

Line 83 is the end of the `.respond` method.

Line 87 makes code.__str__ an alias for the main method, so that you can `print` it or apply `str` to it and it will fill the template. Line 88 gives the name of the main method, because sometimes it's not `.respond`.

Lines 94-95 allow the module to be run directly as a script. Essentially, they process the command-line arguments and them make the template fill itself.

3 Placeholders

3.1 Simple placeholders

Let’s add a few $placeholders to our template:

```
>>> from Cheetah.Template import Template
>>> values = {'what': 'surreal', 'punctuation': '?'}
>>> t = Template("""\
... Hello, $what world$punctuation
... One of Python's least-used functions is $xrange.
... """, [values])
>>> print t
Hello, surreal world?
One of Python's least-used functions is <built-in function xrange>.

>>> print t.generatedModuleCode()
  1 #!/usr/bin/env python

  2 """
  3 Autogenerated by CHEETAH: The Python-Powered Template Engine
  4  CHEETAH VERSION: 0.9.12
  5  Generation time: Sun Apr 21 00:53:01 2002
  6 """

  7 __CHEETAH_genTime__ = 'Sun Apr 21 00:53:01 2002'
  8 __CHEETAH_version__ = '0.9.12'

  9 ##################################################
 10 ## DEPENDENCIES

 11 import sys
 12 import os
 13 import os.path
 14 from os.path import getmtime, exists
 15 import time
 16 import types
 17 from Cheetah.Template import Template
 18 from Cheetah.DummyTransaction import DummyTransaction
 19 from Cheetah.NameMapper import NotFound, valueForName,
           valueFromSearchList
 20 import Cheetah.Filters as Filters
 21 import Cheetah.ErrorCatchers as ErrorCatchers

 22 ##################################################
 23 ## MODULE CONSTANTS

 24 try:
 25     True, False
 26 except NameError:
 27     True, False = (1==1), (1==0)

 28 ##################################################
 29 ## CLASSES

 30 class GenTemplate(Template):
 31     """
 32
 33     Autogenerated by CHEETAH: The Python-Powered Template Engine
 34     """

 35     ##################################################
 36     ## GENERATED METHODS
```

```
37      def __init__(self, *args, **KWs):
38          """
39
40          """

41          Template.__init__(self, *args, **KWs)

42      def respond(self,
43              trans=None,
44              dummyTrans=False,
45              VFS=valueFromSearchList,
46              VFN=valueForName,
47              getmtime=getmtime,
48              currentTime=time.time):


49          """
50          This is the main method generated by Cheetah
51          """

52          if not trans:
53              trans = DummyTransaction()
54              dummyTrans = True
55          write = trans.response().write
56          SL = self._searchList
57          filter = self._currentFilter
58          globalSetVars = self._globalSetVars
59
60          ########################################
61          ## START - generated method body
62
63          write('Hello, ')
64          write(filter(VFS(SL,"what",1))) # generated from '$what' at
                                                # line 1, col 8.
65          write(' world')
66          write(filter(VFS(SL,"punctuation",1))) # generated from
                                    # '$punctuation' at line 1, col 19.
67          write("\nOne of Python's least-used methods is ")
68          write(filter(xrange)) # generated from '$xrange' at line 2,
                                     # col 39.
69          write('.\n')
70
71          ########################################
72          ## END - generated method body
73
74          if dummyTrans:
75              return trans.response().getvalue()
76          else:
77              return ""
```

```
78
79      ##################################################
80      ## GENERATED ATTRIBUTES

81      __str__ = respond
82      _mainCheetahMethod_for_GenTemplate= 'respond'

83 # CHEETAH was developed by Tavis Rudd, Chuck Esterbrook, Ian Bicking
        # and Mike Orr;
84 # with code, advice and input from many other volunteers.
85 # For more information visit http://www.CheetahTemplate.org

86 ##################################################
87 ## if run from command line:
88 if __name__ == '__main__':
89      GenTemplate().runAsMainProgram()
```

(Again, I have added line numbers and split the lines as in the previous chapter.)

This generated template module is different from the previous one in several trivial respects and one important respect. Trivially, `._filePath` and `._fileMtime` are not updated in `.__init__`, so they inherit the value `None` from `Template`. Also, that if-stanza in `.respond` that recompiles the template if the source file changes is missing – because there is no source file. So this module is several lines shorter than the other one.

But the important way this module is different is that instead of the one `write` call outputting a string literal, this module has a series of `write` calls (lines 63-69) outputting successive chunks of the template. Regular text has been translated into a string literal, and placeholders into function calls. Every placeholder is wrapped inside a `filter` call to apply the current output filter. (The default output filter converts all objects to strings, and `None` to `""`.)

Placeholders referring to a Python builtin like `xrange` (line 68) generate a bare variable name. Placeholders to be looked up in the searchList have a nested function call; e.g.,

```
write(filter(VFS(SL,"what",1))) # generated from '$what' at line 1, col 8.
```

`VFS`, remember, is a function imported from `Cheetah.NameMapper` that looks up a value in a searchList. So we pass it the searchList, the name to look up, and a boolean (1) indicating we want autocalling. (It's `1` rather than `True` because it's generated from an `and` expression, and that's what Python 2.2 outputs for true `and` expressions.)

3.2 Complex placeholders

Placeholders can get far more complicated than that. This example shows what kind of code the various NameMapper features produce. The formulas are taken from Cheetah's test suite, in the `Cheetah.Tests.SyntaxAndOutput.Placeholders` class.

```
1 placeholder: $aStr
2 placeholders: $aStr $anInt
2 placeholders, back-to-back: $aStr$anInt
1 placeholder enclosed in {}: ${aStr}
1 escaped placeholder: \$var
func placeholder - with (): $aFunc()
func placeholder - with (int): $aFunc(1234)
func placeholder - with (string): $aFunc('aoeu')
func placeholder - with ('''\nstring'\n'''): $aFunc('''\naoeu'\n''')
func placeholder - with (string*int): $aFunc('aoeu'*2)
func placeholder - with (int*float): $aFunc(2*2.0)
Python builtin values: $None $True $False
func placeholder - with ($arg=float): $aFunc($arg=4.0)
deeply nested argstring: $aFunc(  $arg = $aMeth( $arg = $aFunc( 1 ) ) ):
function with None: $aFunc(None)
autocalling: $aFunc! $aFunc().
nested autocalling: $aFunc($aFunc).
list subscription: $aList[0]
list slicing: $aList[:2]
list slicing and subcription combined: $aList[:2][0]
dict - NameMapper style: $aDict.one
dict - Python style: $aDict['one']
dict combined with autocalled string method: $aDict.one.upper
dict combined with string method: $aDict.one.upper()
nested dict - NameMapper style: $aDict.nestedDict.two
nested dict - Python style: $aDict['nestedDict']['two']
nested dict - alternating style: $aDict['nestedDict'].two
nested dict - NameMapper style + method: $aDict.nestedDict.two.upper
nested dict - alternating style + method: $aDict['nestedDict'].two.upper
nested dict - NameMapper style + method + slice: $aDict.nestedDict.two.upper[:4]
nested dict - Python style, variable key: $aDict[$anObj.meth('nestedDict')].two
object method: $anObj.meth1
object method + complex slice: $anObj.meth1[0: ((4/4*2)*2)/$anObj.meth1(2) ]
very complex slice: $( anObj.meth1[0: ((4/4*2)*2)/$anObj.meth1(2) ] )
```

We'll need a big program to set up the placeholder values. Here it is:

```
#!/usr/bin/env python
from ComplexExample import ComplexExample

try:   # Python >= 2.2.1
    True, False
except NameError:  # Older Python
    True, False = (1==1), (1==0)

class DummyClass:
    _called = False
    def __str__(self):
        return 'object'

    def meth(self, arg="arff"):
        return str(arg)

    def meth1(self, arg="doo"):
        return arg

    def meth2(self, arg1="a1", arg2="a2"):
        return str(arg1) + str(arg2)

    def callIt(self, arg=1234):
        self._called = True
        self._callArg = arg

def dummyFunc(arg="Scooby"):
    return arg

defaultTestNameSpace = {
    'aStr':'blarg',
    'anInt':1,
    'aFloat':1.5,
    'aList': ['item0','item1','item2'],
    'aDict': {'one':'item1',
              'two':'item2',
              'nestedDict':{1:'nestedItem1',
                            'two':'nestedItem2'
                            },
              'nestedFunc':dummyFunc,
              },
    'aFunc': dummyFunc,
    'anObj': DummyClass(),
    'aMeth': DummyClass().meth1,
}

print ComplexExample( searchList=[defaultTestNameSpace] )
```

Here's the output:

```
1 placeholder: blarg
2 placeholders: blarg 1
2 placeholders, back-to-back: blarg1
1 placeholder enclosed in {}: blarg
1 escaped placeholder: $var
func placeholder - with (): Scooby
func placeholder - with (int): 1234
func placeholder - with (string): aoeu
func placeholder - with (’’’\nstring’\n’’’):
aoeu’

func placeholder - with (string*int): aoeuaoeu
func placeholder - with (int*float): 4.0
Python builtin values:  1 0
func placeholder - with ($arg=float): 4.0
deeply nested argstring: 1:
function with None:
autocalling: Scooby! Scooby.
nested autocalling: Scooby.
list subscription: item0
list slicing: [’item0’, ’item1’]
list slicing and subcription combined: item0
dict - NameMapper style: item1
dict - Python style: item1
dict combined with autocalled string method: ITEM1
dict combined with string method: ITEM1
nested dict - NameMapper style: nestedItem2
nested dict - Python style: nestedItem2
nested dict - alternating style: nestedItem2
nested dict - NameMapper style + method: NESTEDITEM2
nested dict - alternating style + method: NESTEDITEM2
nested dict - NameMapper style + method + slice: NEST
nested dict - Python style, variable key: nestedItem2
object method: doo
object method + complex slice: do
very complex slice: do
```

And here – tada! – is the generated module. To save space, I’ve included only the lines containing the `write` calls. The rest of the module is the same as in the first example, chapter 2.1. I’ve split some of the lines to make them fit on the page.

```
 1  write('1 placeholder: ')
 2  write(filter(VFS(SL,"aStr",1))) # generated from '$aStr' at line 1, col 16.
 3  write('\n2 placeholders: ')
 4  write(filter(VFS(SL,"aStr",1))) # generated from '$aStr' at line 2, col 17.
 5  write(' ')
 6  write(filter(VFS(SL,"anInt",1)))
        # generated from '$anInt' at line 2, col 23.
 7  write('\n2 placeholders, back-to-back: ')
 8  write(filter(VFS(SL,"aStr",1))) # generated from '$aStr' at line 3, col 31.
 9  write(filter(VFS(SL,"anInt",1)))
        # generated from '$anInt' at line 3, col 36.
10  write('\n1 placeholder enclosed in {}: ')
11  write(filter(VFS(SL,"aStr",1))) # generated from '${aStr}' at line 4,
        # col 31.
12  write('\n1 escaped placeholder: $var\nfunc placeholder - with (): ')
13  write(filter(VFS(SL,"aFunc",0)())) # generated from '$aFunc()' at line 6,
        # col 29.
14  write('\nfunc placeholder - with (int): ')
15  write(filter(VFS(SL,"aFunc",0)(1234))) # generated from '$aFunc(1234)' at
        # line 7, col 32.
16  write('\nfunc placeholder - with (string): ')
17  write(filter(VFS(SL,"aFunc",0)('aoeu'))) # generated from "$aFunc('aoeu')"
        # at line 8, col 35.
18  write("\nfunc placeholder - with ('''\\nstring'\\n'''): ")
19  write(filter(VFS(SL,"aFunc",0)('''\naoeu'\n'''))) # generated from
        # "$aFunc('''\\naoeu'\\n''')" at line 9, col 46.
20  write('\nfunc placeholder - with (string*int): ')
21  write(filter(VFS(SL,"aFunc",0)('aoeu'*2))) # generated from
        # "$aFunc('aoeu'*2)" at line 10, col 39.
22  write('\nfunc placeholder - with (int*float): ')
23  write(filter(VFS(SL,"aFunc",0)(2*2.0))) # generated from '$aFunc(2*2.0)'
        # at line 11, col 38.
24  write('\nPython builtin values: ')
25  write(filter(None)) # generated from '$None' at line 12, col 24.
26  write(' ')
27  write(filter(True)) # generated from '$True' at line 12, col 30.
28  write(' ')
29  write(filter(False)) # generated from '$False' at line 12, col 36.
30  write('\nfunc placeholder - with ($arg=float): ')
31  write(filter(VFS(SL,"aFunc",0)(arg=4.0))) # generated from
        # '$aFunc($arg=4.0)' at line 13, col 40.
32  write('\ndeeply nested argstring: ')
33  write(filter(VFS(SL,"aFunc",0)(
        arg = VFS(SL,"aMeth",0)( arg = VFS(SL,"aFunc",0)( 1 ) ) )))
# generated from '$aFunc(  $arg = $aMeth( $arg = $aFunc( 1 ) ) )'
# at line 14, col 26.
34  write(':\nfunction with None: ')
35  write(filter(VFS(SL,"aFunc",0)(None))) # generated from '$aFunc(None)' at
        # line 15, col 21.
36  write('\nautocalling: ')
37  write(filter(VFS(SL,"aFunc",1))) # generated from '$aFunc' at line 16,
        # col 14.
38  write('! ')
39  write(filter(VFS(SL,"aFunc",0)())) # generated from '$aFunc()' at line 16,
        # col 22.
```

```
write('.\nnested autocalling: ')
write(filter(VFS(SL,"aFunc",0)(VFS(SL,"aFunc",1)))) # generated from
    # '$aFunc($aFunc)' at line 17, col 21.
write('.\nlist subscription: ')
write(filter(VFS(SL,"aList",1)[0])) # generated from '$aList[0]' at line
    # 18, col 20.
write('\nlist slicing: ')
write(filter(VFS(SL,"aList",1)[:2])) # generated from '$aList[:2]' at
    # line 19, col 15.
write('\nlist slicing and subcription combined: ')
write(filter(VFS(SL,"aList",1)[:2][0])) # generated from '$aList[:2][0]'
    # at line 20, col 40.
write('\ndict - NameMapper style: ')
write(filter(VFS(SL,"aDict.one",1))) # generated from '$aDict.one' at line
    # 21, col 26.
write('\ndict - Python style: ')
write(filter(VFS(SL,"aDict",1)['one'])) # generated from "$aDict['one']"
    # at line 22, col 22.
write('\ndict combined with autocalled string method: ')
write(filter(VFS(SL,"aDict.one.upper",1))) # generated from
    # '$aDict.one.upper' at line 23, col 46.
write('\ndict combined with string method: ')
write(filter(VFN(VFS(SL,"aDict.one",1),"upper",0)())) # generated from
    # '$aDict.one.upper()' at line 24, col 35.
write('\nnested dict - NameMapper style: ')
write(filter(VFS(SL,"aDict.nestedDict.two",1))) # generated from
    # '$aDict.nestedDict.two' at line 25, col 33.
write('\nnested dict - Python style: ')
write(filter(VFS(SL,"aDict",1)['nestedDict']['two'])) # generated from
    # "$aDict['nestedDict']['two']" at line 26, col 29.
write('\nnested dict - alternating style: ')
write(filter(VFN(VFS(SL,"aDict",1)['nestedDict'],"two",1))) # generated
    # from "$aDict['nestedDict'].two" at line 27, col 34.
write('\nnested dict - NameMapper style + method: ')
write(filter(VFS(SL,"aDict.nestedDict.two.upper",1))) # generated from
    # '$aDict.nestedDict.two.upper' at line 28, col 42.
write('\nnested dict - alternating style + method: ')
write(filter(VFN(VFS(SL,"aDict",1)['nestedDict'],"two.upper",1)))
    # generated from "$aDict['nestedDict'].two.upper" at line 29, col 43.
write('\nnested dict - NameMapper style + method + slice: ')
```

```
67  write(filter(VFN(VFS(SL,"aDict.nestedDict.two",1),"upper",1)[:4]))
        # generated from '$aDict.nestedDict.two.upper[:4]' at line 30, col 50.
68  write('\nnested dict - Python style, variable key: ')
69  write(filter(VFN(VFS(SL,"aDict",1)
        [VFN(VFS(SL,"anObj",1),"meth",0)('nestedDict')],"two",1)))
# generated from "$aDict[$anObj.meth('nestedDict')].two" at line 31,
# col 43.
70  write('\nobject method: ')
71  write(filter(VFS(SL,"anObj.meth1",1))) # generated from '$anObj.meth1' at
        # line 32, col 16.
72  write('\nobject method + complex slice: ')
73  write(filter(VFN(VFS(SL,"anObj",1),"meth1",1)
        [0: ((4/4*2)*2)/VFN(VFS(SL,"anObj",1),"meth1",0)(2) ]))
# generated from '$anObj.meth1[0: ((4/4*2)*2)/$anObj.meth1(2) ]'
# at line 33, col 32.
74  write('\nvery complex slice: ')
75  write(filter(VFN(VFS(SL,"anObj",1),"meth1",1)
        [0: ((4/4*2)*2)/VFN(VFS(SL,"anObj",1),"meth1",0)(2) ] ))
# generated from '$( anObj.meth1[0: ((4/4*2)*2)/$anObj.meth1(2) ] )'
# at line 34, col 21.
76  write('\n')
```

For each placeholder lookup, the the innermost level of nesting is a VFS call, which looks up the first (leftmost) placeholder component in the searchList. This is wrapped by zero or more VFN calls, which perform Universal Dotted Notation lookup on the next dotted component of the placeholder, looking for an attribute or key by that name within the previous object (not in the searchList). Autocalling is performed by VFS and VFN: that's the reason for their third argument.

Explicit function/method arguments, subscripts and keys (which are all expressions) are left unchanged, besides expanding any embedded $placeholders in them. This means they must result in valid Python expressions, following the standard Python quoting rules.

Built-in Python values (`None`, `True` and `False`) are converted to `filter(None)`, etc. They use normal Python variable lookup rather than VFS. (Cheetah emulates `True` and `False` using global variables for Python ¡ 2.2.1, when they weren't builtins yet.)

4 Caching placeholders and #cache

4.1 Dynamic placeholder – no cache

The template:

```
Dynamic variable:  $voom
```

The command line and the output:

```
% voom='Voom!' python x.py --env
Dynamic variable:  Voom!
```

The generated code:

```
write('Dynamic variable:  ')
write(filter(VFS(SL,"voom",1))) # generated from '$voom' at line 1, col 20.
write('\n')
```

Just what we expected, like any other dynamic placeholder.

4.2 Static placeholder

The template:

```
Cached variable:  $*voom
```

The command line and output:

```
% voom='Voom!' python x.py --env
Cached variable:  Voom!
```

The generated code, with line numbers:

```
 1  write('Cached variable:  ')
 2  ## START CACHE REGION: at line, col (1, 19) in the source.
 3  RECACHE = True
 4  if not self._cacheData.has_key('19760169'):
 5      pass
 6  else:
 7      RECACHE = False
 8  if RECACHE:
 9      orig_trans = trans
10      trans = cacheCollector = DummyTransaction()
11      write = cacheCollector.response().write
12      write(filter(VFS(SL,"voom",1))) # generated from '$*voom' at line 1,
            # col 19.
13      trans = orig_trans
14      write = trans.response().write
15      self._cacheData['19760169'] = cacheCollector.response().getvalue()
16      del cacheCollector
17  write(self._cacheData['19760169'])
18  ## END CACHE REGION

19  write('\n')
```

That one little star generated a whole lotta code. First, instead of an ordinary `VFS` lookup (searchList) lookup, it converted the placeholder to a lookup in the `._cacheData` dictionary. Cheetah also generated a unique key (`'19760169'`) for our cached item – this is its cache ID.

Second, Cheetah put a pair of if-blocks before the `write`. The first (lines 3-7) determine whether the cache value is missing or out of date, and sets local variable `RECHARGE` true or false. This stanza may look unnecessarily verbose – lines 3-7 could be eliminated if line 8 was changed to

```
if not self._cacheData.has_key('19760169'):
```

– but this model is expandable for some of the cache features we'll see below.

The second if-block, lines 8-16, do the cache updating if necessary. Clearly, the programmer is trying to stick as close to normal (dynamic) workflow as possible. Remember that `write`, even though it looks like a local function, is actually a method of a file-like object. So we create a temporary file-like object to divert the `write` object into, then read the result and stuff it into the cache.

4.3 Timed-refresh placeholder

The template:

```
Timed cache:  $*.5m*voom
```

The command line and the output:

```
% voom='Voom!' python x.py --env
Timed cache:  Voom!
```

The generated method's docstring:

```
"""
This is the main method generated by Cheetah
This cache will be refreshed every 30.0 seconds.
"""
```

The generated code:

```
 1  write('Timed cache:  ')
 2  ## START CACHE REGION: at line, col (1, 15) in the source.
 3  RECACHE = True
 4  if not self._cacheData.has_key('55048032'):
 5      self.__cache55048032__refreshTime = currentTime() + 30.0
 6  elif currentTime() > self.__cache55048032__refreshTime:
 7      self.__cache55048032__refreshTime = currentTime() + 30.0
 8  else:
 9      RECACHE = False
10  if RECACHE:
11      orig_trans = trans
12      trans = cacheCollector = DummyTransaction()
13      write = cacheCollector.response().write
14      write(filter(VFS(SL,"voom",1))) # generated from '$*.5m*voom' at
            # line 1, col 15.
15      trans = orig_trans
16      write = trans.response().write
17      self._cacheData['55048032'] = cacheCollector.response().getvalue()
18      del cacheCollector
19  write(self._cacheData['55048032'])
20  ## END CACHE REGION

21  write('\n')
```

This code is identical to the static cache example except for the docstring and the first if-block. (OK, so the cache ID is different and the comment on line 14 is different too. Big deal.)

Each timed-refresh cache item has a corrsponding private attribute `.__cache########__refreshTime` giving the refresh time in ticks (=seconds since January 1, 1970). The first if-block (lines 3-9) checks whether the cache value is missing or its update time has passed, and if so, sets `RECHARGE` to true and also schedules another refresh at the next interval.

The method docstring reminds the user how often the cache will be refreshed. This information is unfortunately not as robust as it could be. Each timed-cache placeholder blindly generates a line in the docstring. If all refreshes are at the same interval, there will be multiple identical lines in the docstring. If the refreshes are at different intervals, you get a situation like this:

```
"""
This is the main method generated by Cheetah
This cache will be refreshed every 30.0 seconds.
This cache will be refreshed every 60.0 seconds.
This cache will be refreshed every 120.0 seconds.
"""
```

The docstring tells only that "something" will be refreshed every 60.0 seconds, but doesn't reveal *which* placeholder that is. Only if you know the relative order of the placeholders in the template can you figure that out.

4.4 Timed-refresh placeholder with braces

This example is the same but with the long placeholder syntax. It's here because it's a Cheetah FAQ whether to put the cache interval inside or outside the braces. (It's also here so I can look it up because I frequently forget.) The answer is: outside. The braces go around only the placeholder name (and perhaps some output-filter arguments.)

The template:

```
Timed with {}:  $*.5m*{voom}
```

The output:

```
Timed with {}:  Voom!
```

The generated code differs only in the comment. Inside the cache-refresh if-block:

```
write(filter(VFS(SL,"voom",1))) # generated from '$*.5m*{voom}' at line 1,
    #col 17.
```

The reason this example is here is because it's a Cheetah FAQ whether to put the cache interval inside or outside the `{}`. (Also so I can look it up when I forget, as I frequently do.) The answer is: outside. The `{}` go around only the placeholder name and arguments. If you do it this way:

```
Timed with {}:  ${*.5m*voom}      ## Wrong!
```

you get:

```
Timed with {}:  ${*.5m*voom}
```

because `${` is not a valid placeholder, so it's treated as ordinary text.

4.5 #cache

The template:

```
#cache
This is a cached region.  $voom
#end cache
```

The output:

```
This is a cached region.  Voom!
```

The generated code:

```
 1  ## START CACHE REGION: at line, col (1, 1) in the source.
 2  RECACHE = True
 3  if not self._cacheData.has_key('23711421'):
 4      pass
 5  else:
 6      RECACHE = False
 7  if RECACHE:
 8      orig_trans = trans
 9      trans = cacheCollector = DummyTransaction()
10      write = cacheCollector.response().write
11      write('This is a cached region.  ')
12      write(filter(VFS(SL,"voom",1))) # generated from '$voom' at line 2,
            # col 27.
13      write('\n')
14      trans = orig_trans
15      write = trans.response().write
16      self._cacheData['23711421'] = cacheCollector.response().getvalue()
17      del cacheCollector
18  write(self._cacheData['23711421'])
19  ## END CACHE REGION
```

This is the same as the `$*voom` example, except that the plain text around the placeholder is inside the second if-block.

4.6 #cache with timer and id

The template:

```
#cache timer='.5m', id='cache1'
This is a cached region.  $voom
#end cache
```

The output:

```
This is a cached region.  Voom!
```

The generated code is the same as the previous example except the first if-block:

```
RECACHE = True
if not self._cacheData.has_key('13925129'):
    self._cacheIndex['cache1'] = '13925129'
    self.__cache13925129__refreshTime = currentTime() + 30.0
elif currentTime() > self.__cache13925129__refreshTime:
    self.__cache13925129__refreshTime = currentTime() + 30.0
else:
    RECACHE = False
```

4.7 #cache with test: expression and method conditions

The template:

```
#cache test=$isDBUpdated
This is a cached region.  $voom
#end cache
```

(Analysis postponed: bug in Cheetah produces invalid Python.)

The template:

```
#cache id='cache1', test=($isDBUpdated or $someOtherCondition)
This is a cached region.  $voom
#end cache
```

The output:

```
This is a cached region.  Voom!
```

The first if-block in the generated code:

```
RECACHE = True
if not self._cacheData.has_key('36798144'):
    self._cacheIndex['cache1'] = '36798144'
elif (VFS(SL,"isDBUpdated",1) or VFS(SL,"someOtherCondition",1)):
    RECACHE = True
else:
    RECACHE = False
```

The second if-block is the same as in the previous example. If you leave out the `()` around the test expression, the result is the same, although it may be harder for the template maintainer to read.

You can even combine arguments, although this is of questionable value.

The template:

```
#cache id='cache1', timer='30m', test=$isDBUpdated or $someOtherCondition
This is a cached region.  $voom
#end cache
```

The output:

```
This is a cached region.  Voom!
```

The first if-block:

```
RECACHE = True
if not self._cacheData.has_key('88939345'):
    self._cacheIndex['cache1'] = '88939345'
    self.__cache88939345__refreshTime = currentTime() + 1800.0
elif currentTime() > self.__cache88939345__refreshTime:
    self.__cache88939345__refreshTime = currentTime() + 1800.0
elif VFS(SL,"isDBUpdated",1) or VFS(SL,"someOtherCondition",1):
    RECACHE = True
else:
    RECACHE = False
```

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. Since this is not implemented yet, I cannot provide examples here.

5 Directives: Comments

The template:

```
Text before the comment.
## The comment.
Text after the comment.
#* A multi-line comment spanning several lines.
   It spans several lines, too.
*#
Text after the multi-line comment.
```

The output:

```
Text before the comment.
Text after the comment.

Text after the multi-line comment.
```

The generated code:

```
write('Text before the comment.\n')
#  The comment.
write('Text after the comment.\n')
#  A multi-line comment spanning several lines.
#    It spans several lines, too.
write('\nText after the multi-line comment.\n')
```

5.1 Docstring and header comments

The template:

```
##doc: .respond() method comment.
##doc-method: Another .respond() method comment.
##doc-class: A class comment.
##doc-module: A module comment.
##header: A header comment.
```

The output:

The beginning of the generated `.respond` method:

```
def respond(self,
        trans=None,
        dummyTrans=False,
        VFS=valueFromSearchList,
        VFN=valueForName,
        getmtime=getmtime,
        currentTime=time.time):

    """
    This is the main method generated by Cheetah
    .respond() method comment.
    Another .respond() method comment.
    """
```

The class docstring:

```
"""
A class comment.

Autogenerated by CHEETAH: The Python-Powered Template Engine
"""
```

The top of the module:

```
#!/usr/bin/env python
# A header comment.

"""A module comment.

Autogenerated by CHEETAH: The Python-Powered Template Engine
 CHEETAH VERSION: 0.9.13a1
 Generation time: Fri Apr 26 22:39:23 2002
   Source file: x.tmpl
   Source file last modified: Fri Apr 26 22:36:23 2002
"""
```

6 Directives: Output

6.1 #echo

The template:

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

The output:

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

The generated code:

```
write(’Here is my ’)
write(filter(’, ’.join([’silly’]*5) ))
write(’ example\n’)
```

6.2 #silent

The template:

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

The output:

```
Here is my  example
```

The generated code:

```
            write(’Here is my ’)
            ’, ’.join([’silly’]*5)
            write(’ example\n’)
```

OK, it’s not quite covert because that extra space gives it away, but it almost succeeds.

6.3 #raw

The template:

```
Text before raw.
#raw
Text in raw.  $alligator.  $croc.o['dile'].  #set $a = $b + $c.
#end raw
Text after raw.
```

The output:

```
Text before raw.
Text in raw.  $alligator.  $croc.o['dile'].  #set $a = $b + $c.
Text after raw.
```

The generated code:

```
        write('''Text before raw.
Text in raw.  $alligator.  $croc.o['dile'].  #set $a = $b + $c.
Text after raw.
''')
```

So we see that `#raw` is really like a quoting mechanism. It says that anything inside it is ordinary text, and Cheetah joins a `#raw` section with adjacent string literals rather than generating a separate `write` call.

6.4 #include

The main template:

```
#include "y.tmpl"
```

The included template y.tmpl:

```
Let's go $voom!
```

The shell command and output:

```
% voom="VOOM" x.py --env
Let's go VOOM!
```

The generated code:

```
write(self._includeCheetahSource("y.tmpl", trans=trans, includeFrom="file",
    raw=0))
```

#include raw

The main template:

```
#include raw "y.tmpl"
```

The shell command and output:

```
% voom="VOOM" x.py --env
Let’s go $voom!
```

The generated code:

```
write(self._includeCheetahSource("y.tmpl", trans=trans, includeFrom="fil
e", raw=1))
```

That last argument, `raw`, makes the difference.

#include from a string or expression (eval)

The template:

```
#attr $y = "Let’s go $voom!"
#include source=$y
#include raw source=$y
#include source="Bam!  Bam!"
```

The output:

```
% voom="VOOM" x.py --env
Let’s go VOOM!Let’s go $voom!Bam!  Bam!
```

The generated code:

```
write(self._includeCheetahSource(VFS(SL,"y",1), trans=trans,
    includeFrom="str", raw=0, includeID="481020889808.74"))
write(self._includeCheetahSource(VFS(SL,"y",1), trans=trans,
    includeFrom="str", raw=1, includeID="711020889808.75"))
write(self._includeCheetahSource("Bam!  Bam!", trans=trans,
    includeFrom="str", raw=0, includeID="1001020889808.75"))
```

Later in the generated class:

```
y = "Let’s go $voom!"
```

6.5 #slurp

The template:

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

The output:

```
0
1
2
3
4
0 1 2 3 4 Line after slurp.
```

The generated code:

```
for i in range(5):
    write(filter(i)) # generated from '$i' at line 2, col 1.
    write('\n')
for i in range(5):
    write(filter(i)) # generated from '$i' at line 5, col 1.
    write(' ')
write('Line after slurp.\n')
```

The space after each number is because of the space before `#slurp` in the template definition.

6.6 #filter

The template:

```
#attr $ode = ">> Rubber Ducky, you're the one!  You make bathtime so much fun! <<"
$ode
#filter WebSafe
$ode
#filter MaxLen
${ode, maxlen=13}
#filter None
${ode, maxlen=13}
```

The output:

```
>> Rubber Ducky, you're the one!  You make bathtime so much fun! <<
&gt;&gt; Rubber Ducky, you're the one!  You make bathtime so much fun! &lt;&lt;
>> Rubber Duc
>> Rubber Ducky, you're the one!  You make bathtime so much fun! <<
```

The `WebSafe` filter escapes characters that have a special meaning in HTML. The `MaxLen` filter chops off values at the specified length. `#filter None` returns to the default filter, which ignores the `maxlen` argument.

The generated code:

```
 1  write(filter(VFS(SL,"ode",1))) # generated from '$ode' at line 2, col 1.
 2  write('\n')
 3  filterName = 'WebSafe'
 4  if self._filters.has_key("WebSafe"):
 5      filter = self._currentFilter = self._filters[filterName]
 6  else:
 7      filter = self._currentFilter = \
 8                  self._filters[filterName] = getattr(self._filtersLib,
                       filterName)(self).filter
 9  write(filter(VFS(SL,"ode",1))) # generated from '$ode' at line 4, col 1.
10  write('\n')
11  filterName = 'MaxLen'
12  if self._filters.has_key("MaxLen"):
13      filter = self._currentFilter = self._filters[filterName]
14  else:
15      filter = self._currentFilter = \
16                  self._filters[filterName] = getattr(self._filtersLib,
                       filterName)(self).filter
17  write(filter(VFS(SL,"ode",1), maxlen=13)) # generated from
        #'${ode, maxlen=13}' at line 6, col 1.
18  write('\n')
19  filter = self._initialFilter
20  write(filter(VFS(SL,"ode",1), maxlen=13)) # generated from
       #'${ode, maxlen=13}' at line 8, col 1.
21  write('\n')
```

As we've seen many times, Cheetah wraps all placeholder lookups in a `filter` call. (This also applies to non-searchList lookups: local, global and builtin variables.) The `filter` "function" is actually an alias to the current filter object:

```
filter = self._currentFilter
```

as set at the top of the main method. Here in lines 3-8 and 11-16 we see the filter being changed. Whoops, I lied. `filter` is not an alias to the filter object itself but to that object's `.filter` method. Line 19 switches back to the default filter.

In line 17 we see the `maxlen` argument being passed as a keyword argument to `filter` (not to VFS). In line 20 the same thing happens although the default filter ignores the argument.

7 Directives: Import, Inheritance, Declaration and Assignment

7.1 #import and #from

The template:

```
#import math
```

This construct does not produce any output.

The generated module, at the bottom of the import section:

```
import math
```

7.2 #extends

The template:

```
#extends SomeClass
```

The generated import (skipped if `SomeClass` has already been imported):

```
from SomeClass import SomeClass
```

The generated class:

```
class x(SomeClass):
```

7.3 #implements

The template:

```
#implements doOutput
```

In the generated class, the main method is `.doOutput` instead of `.respond`, and the attribute naming this method is:

```
_mainCheetahMethod_for_x2= 'doOutput'
```

7.4 #set and #set global

The template:

```
#set $namesList = ['Moe','Larry','Curly']
$namesList
#set global $toes = ['eeny', 'meeny', 'miney', 'moe']
$toes
```

The output:

```
['Moe', 'Larry', 'Curly']
['eeny', 'meeny', 'miney', 'moe']
```

The generated code:

```
1  namesList = ['Moe','Larry','Curly']
2  write(filter(namesList)) # generated from '$namesList' at line 2, col 1.
3  write('\n')
4  globalSetVars["toes"] = ['eeny', 'meeny', 'miney', 'moe']
5  write(filter(VFS(SL,"toes",1))) # generated from '$toes' at line 4, col 1.
6  write('\n')
```

`globalSetVars` is a local variable shadowing `._globalSetVars`. Writes go into it directly, but reads take advantage of the fact that `._globalSetVars` is on the searchList. (In fact, it's the very first namespace.)

7.5 #del

The template:

```
#set $a = 1
#del $a
#set $a = 2
#set $arr = [0, 1, 2]
#del $a, $arr[1]
```

In the generated class:

```
1  a = 1
2  del a
3  a = 2
4  arr = [0, 1, 2]
5  del a, arr[1]
```

7.6 #attr

The template:

```
#attr $namesList = ['Moe', 'Larry', 'Curly']
```

In the generated class:

```
## GENERATED ATTRIBUTES

namesList = ['Moe', 'Larry', 'Curly']
```

7.7 #def

The template:

```
#def printArg($arg)
The argument is $arg.
#end def
My method returned $printArg(5).
```

The output:

```
My method returned The argument is 5.
.
```

Hmm, not exactly what we expected. The method returns a trailing newline because we didn't end the last line with `#slurp`. So the second period (outside the method) appears on a separate line.

The `#def` generates a method `.printArg` whose structure is similar to the main method:

```
def printArg(self,
        arg,
        trans=None,
        dummyTrans=False,
        VFS=valueFromSearchList,
        VFN=valueForName,
        getmtime=getmtime,
        currentTime=time.time):


    """
    Generated from #def printArg($arg) at line 1, col 1.
    """

    if not trans:
        trans = DummyTransaction()
        dummyTrans = True
    write = trans.response().write
    SL = self._searchList
    filter = self._currentFilter
    globalSetVars = self._globalSetVars

    ########################################
    ## START - generated method body

    write('The argument is ')
    write(filter(arg)) # generated from '$arg' at line 2, col 17.
    write('.\n')

    ########################################
    ## END - generated method body

    if dummyTrans:
        return trans.response().getvalue()
    else:
        return ""
```

When `.printArg` is called from a placeholder, only the arguments the user supplied are passed. The other arguments retain their default values.

7.8 #block

The template:

```
#block content
This page is under construction.
#end block
```

The output:

```
This page is under construction.
```

This construct generates a method `.content` in the same structure as `.printArg` above, containing the write code:

```
write('This page is under construction.\n')
```

In the main method, the write code is:

```
self.content(trans=trans) # generated from ('content', '#block content')
    # at line 1, col 1.
```

So a block placeholder implicitly passes the current transaction to the method.

7.9 #settings

This directive is undocumented because it's likely to disappear in Cheetah 0.9.14.

8 Directives: Flow Control

8.1 #for

The template:

```
#for $i in $range(10)
$i #slurp
#end for
```

The output:

```
0 1 2 3 4 5 6 7 8 9
```

The generated code:

```
for i in range(10):
    write(filter(i)) # generated from '$i' at line 2, col 1.
    write(' ')
```

8.2 #repeat

The template:

```
#repeat 3
My bonnie lies over the ocean
#end repeat
O, bring back my bonnie to me!
```

The output:

```
My bonnie lies over the ocean
My bonnie lies over the ocean
My bonnie lies over the ocean
O, bring back my bonnie to me!
```

(OK, so the second line should be “sea” instead of “ocean”.)

The generated code:

```
for i in range( 3):
    write('My bonnie lies over the ocean\n')
write('O, bring back my bonnie to me!\n')
```

8.3 #while

The template:

```
#set $alive = True
#while $alive
I am alive!
#set $alive = False
#end while
```

The output:

```
I am alive!
```

The generated code:

```
alive = True
while alive:
    write('I am alive!\n')
    alive = False
```

8.4 #if

The template:

```
#set $size = 500
#if $size >= 1500
It's big
#else if $size < 1500 and $size > 0
It's small
#else
It's not there
#end if
```

The output:

```
It's small
```

The generated code:

```
size = 500
if size >= 1500:
    write("It's big\n")
elif size < 1500 and size > 0:
    write("It's small\n")
else:
    write("It's not there\n")
```

8.5 #unless

The template:

```
#set $count = 9
#unless $count + 5 > 15
Count is in range.
#end unless
```

The output:

```
Count is in range.
```

The generated code:

```
count = 9
if not (count + 5 > 15):
    write('Count is in range.\n')
```

Note: There is a bug in Cheetah 0.9.13. It's forgetting the parentheses in the `if` expression, which could lead to it calculating something different than it should.

8.6 #break and #continue

The template:

```
#for $i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 'James', 'Joe', 'Snow']
#if $i == 10
  #continue
#end if
#if $i == 'Joe'
  #break
#end if
$i - #slurp
#end for
```

The output:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 11 - 12 - James -
```

The generated code:

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 'James', 'Joe', 'Snow']:
    if i == 10:
        write('')
        continue
    if i == 'Joe':
        write('')
        break
    write(filter(i)) # generated from '$i' at line 8, col 1.
    write(' - ')
```

8.7 #pass

The template:

```
Let's check the number.
#set $size = 500
#if $size >= 1500
It's big
#elif $size > 0
#pass
#else
Invalid entry
#end if
Done checking the number.
```

The output:

```
Let's check the number.
Done checking the number.
```

The generated code:

```
write("Let's check the number.\n")
size = 500
if size >= 1500:
    write("It's big\n")
elif size > 0:
    pass
else:
    write('Invalid entry\n')
write('Done checking the number.\n')
```

8.8 #stop

The template:

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

The output:

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

The generated code:

```
write('A cat\n')
if 1:
    write('  sat on a mat\n')
    if dummyTrans:
        return trans.response().getvalue()
    else:
        return ""
    write('  watching a rat\n')
write('in a flat.\n')
```

8.9 #return

The template:

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

The output:

```
1
2
3
```

The generated code:

```
        def test(self,
                 trans=None,
                 dummyTrans=False,
                 VFS=valueFromSearchList,
                 VFN=valueForName,
                 getmtime=getmtime,
                 currentTime=time.time):


            """
            Generated from #def test at line 5, col 1.
            """

            if not trans:
                trans = DummyTransaction()
                dummyTrans = True
            write = trans.response().write
            SL = self._searchList
            filter = self._currentFilter
            globalSetVars = self._globalSetVars

            ########################################
            ## START - generated method body

            write('1.5\n')
            if 1:
                return '123'
            else:
                write('99999\n')

            ########################################
            ## END - generated method body

            if dummyTrans:
                return trans.response().getvalue()
            else:
                return ""
```

```
def respond(self,
        trans=None,
        dummyTrans=False,
        VFS=valueFromSearchList,
        VFN=valueForName,
        getmtime=getmtime,
        currentTime=time.time):


    """
    This is the main method generated by Cheetah
    """

    if not trans:
        trans = DummyTransaction()
        dummyTrans = True
    write = trans.response().write
    SL = self._searchList
    filter = self._currentFilter
    globalSetVars = self._globalSetVars

    ########################################
    ## START - generated method body

    write('\n1\n')
    write(filter(VFS(SL,"test",1)[1])) # generated from '$test[1]' at line 3, col 1.
    write('\n3\n')

    ########################################
    ## END - generated method body

    if dummyTrans:
        return trans.response().getvalue()
    else:
        return ""
```

9 Directives: Error Handling

9.1 #try and #raise

The template:

```
#import traceback
#try
#raise RuntimeError
#except RuntimeError
A runtime error occurred.
#end try

#try
#raise RuntimeError("Hahaha!")
#except RuntimeError
#echo $sys.exc_info()[1]
#end try

#try
#echo 1/0
#except ZeroDivisionError
You can’t divide by zero, idiot!
#end try
```

The output:

```
A runtime error occurred.

Hahaha!

You can’t divide by zero, idiot!
```

The generated code:

```
try:
    raise RuntimeError
except RuntimeError:
    write(’A runtime error occurred.\n’)
write(’\n’)
try:
    raise RuntimeError("Hahaha!")
except RuntimeError:
    write(filter(VFN(sys,"exc_info",0)()[1]))
    write(’\n’)
write(’\n’)
try:
    write(filter(1/0))
    write(’\n’)
except ZeroDivisionError:
    write("You can’t divide by zero, idiot!\n")
```

`#finally` works just like in Python.

9.2 #assert

The template:

```
#assert False, "You lose, buster!"
```

The output:

```
Traceback (most recent call last):
  File "x.py", line 117, in ?
    x().runAsMainProgram()
  File "/local/opt/Python/lib/python2.2/site-packages/Webware/Cheetah/
Template.py", line 331, in runAsMainProgram
    CmdLineIface(templateObj=self).run()
  File "/local/opt/Python/lib/python2.2/site-packages/Webware/Cheetah/
TemplateCmdLineIface.py", line 59, in run
    print self._template
  File "x.py", line 91, in respond
    assert False, "You lose, buster!"
AssertionError: You lose, buster!
```

The generated code:

```
assert False, "You lose, buster!"
```

9.3 #errorCatcher

No error catcher

The template:

```
$noValue
```

The output:

```
Traceback (most recent call last):
  File "x.py", line 118, in ?
    x().runAsMainProgram()
  File "/local/opt/Python/lib/python2.2/site-packages/Webware/Cheetah/
Template.py", line 331, in runAsMainProgram
    CmdLineIface(templateObj=self).run()
  File "/local/opt/Python/lib/python2.2/site-packages/Webware/Cheetah/
TemplateCmdLineIface.py", line 59, in run
    print self._template
  File "x.py", line 91, in respond
    write(filter(VFS(SL,"noValue",1))) # generated from '$noValue' at line
1, col 1.
NameMapper.NotFound: noValue
```

The generated code:

```
write(filter(VFS(SL,"noValue",1))) # generated from '$noValue' at line 1,
    # col 1.
write('\n')
```

Echo and BigEcho

The template:

```
#errorCatcher Echo
$noValue
#errorCatcher BigEcho
$noValue
```

The output:

```
$noValue
===============&lt;$noValue could not be found&gt;===============
```

The generated code:

```
if self._errorCatchers.has_key("Echo"):
    self._errorCatcher = self._errorCatchers["Echo"]
else:
    self._errorCatcher = self._errorCatchers["Echo"] = ErrorCatchers.Echo(self)
write(filter(self.__errorCatcher1(localsDict=locals())))
    # generated from '$noValue' at line 2, col 1.
write('\n')
if self._errorCatchers.has_key("BigEcho"):
    self._errorCatcher = self._errorCatchers["BigEcho"]
else:
    self._errorCatcher = self._errorCatchers["BigEcho"] = \
        ErrorCatchers.BigEcho(self)
write(filter(self.__errorCatcher1(localsDict=locals())))
        # generated from '$noValue' at line 4, col 1.
write('\n')
```

ListErrors

The template:

```
#import pprint
#errorCatcher ListErrors
$noValue
$anotherMissingValue.really
$pprint.pformat($errorCatcher.listErrors)
## This is really self.errorCatcher().listErrors()
```

The output:

```
$noValue
$anotherMissingValue.really
[{'code': 'VFS(SL,"noValue",1)',
  'exc_val': <NameMapper.NotFound instance at 0x8170ecc>,
  'lineCol': (3, 1),
  'rawCode': '$noValue',
  'time': 'Wed May 15 00:38:23 2002'},
 {'code': 'VFS(SL,"anotherMissingValue.really",1)',
  'exc_val': <NameMapper.NotFound instance at 0x816d0fc>,
  'lineCol': (4, 1),
  'rawCode': '$anotherMissingValue.really',
  'time': 'Wed May 15 00:38:23 2002'}]
```

The generated import:

```
import pprint
```

Then in the generated class, we have our familiar `.respond` method and several new methods:

```
def __errorCatcher1(self, localsDict={}):
    """
    Generated from $noValue at line, col (3, 1).
    """

    try:
        return eval('''VFS(SL,"noValue",1)''', globals(), localsDict)
    except self._errorCatcher.exceptions(), e:
        return self._errorCatcher.warn(exc_val=e, code= 'VFS(SL,"noValue",1)' ,
    rawCode= '$noValue' , lineCol=(3, 1))

def __errorCatcher2(self, localsDict={}):
    """
    Generated from $anotherMissingValue.really at line, col (4, 1).
    """

    try:
        return eval('''VFS(SL,"anotherMissingValue.really",1)''', globals(),
    localsDict)
    except self._errorCatcher.exceptions(), e:
        return self._errorCatcher.warn(exc_val=e,
    code= 'VFS(SL,"anotherMissingValue.really",1)' ,
    rawCode= '$anotherMissingValue.really' , lineCol=(4, 1))

def __errorCatcher3(self, localsDict={}):
    """
    Generated from $pprint.pformat($errorCatcher.listErrors) at line, col
    (5, 1).
    """

    try:
        return eval('''VFN(pprint,"pformat",0)(VFS(SL,
    "errorCatcher.listErrors",1))''', globals(), localsDict)
    except self._errorCatcher.exceptions(), e:
        return self._errorCatcher.warn(exc_val=e, code=
    'VFN(pprint,"pformat",0)(VFS(SL,"errorCatcher.listErrors",1))' ,
    rawCode= '$pprint.pformat($errorCatcher.listErrors)' ,
    lineCol=(5, 1))
```

```
def respond(self,
        trans=None,
        dummyTrans=False,
        VFS=valueFromSearchList,
        VFN=valueForName,
        getmtime=getmtime,
        currentTime=time.time):


    """
    This is the main method generated by Cheetah
    """

    if not trans:
        trans = DummyTransaction()
        dummyTrans = True
    write = trans.response().write
    SL = self._searchList
    filter = self._currentFilter
    globalSetVars = self._globalSetVars

    ########################################
    ## START - generated method body

    if exists(self._filePath) and getmtime(self._filePath) > self._fileMtime:
        self.compile(file=self._filePath)
        write(getattr(self, self._mainCheetahMethod_for_x)(trans=trans))
        if dummyTrans:
            return trans.response().getvalue()
        else:
            return ""
    if self._errorCatchers.has_key("ListErrors"):
        self._errorCatcher = self._errorCatchers["ListErrors"]
    else:
        self._errorCatcher = self._errorCatchers["ListErrors"] = \
    ErrorCatchers.ListErrors(self)
    write(filter(self.__errorCatcher1(localsDict=locals())))
        # generated from '$noValue' at line 3, col 1.
    write('\n')
    write(filter(self.__errorCatcher2(localsDict=locals())))
        # generated from '$anotherMissingValue.really' at line 4, col 1.
    write('\n')
    write(filter(self.__errorCatcher3(localsDict=locals())))
        # generated from '$pprint.pformat($errorCatcher.listErrors)' at line
# 5, col 1.
    write('\n')
    #  This is really self.errorCatcher().listErrors()

    ########################################
    ## END - generated method body

    if dummyTrans:
        return trans.response().getvalue()
    else:
        return ""
```

So whenever an error catcher is active, each placeholder gets wrapped in its own method. No wonder error catchers

slow down the system!

10 Directives: Parser Instructions

10.1 #breakpoint

The template:

```
Text before breakpoint.
#breakpoint
Text after breakpoint.
#raise RuntimeError
```

The output:

```
Text before breakpoint.
```

The generated code:

```
write('Text before breakpoint.\n')
```

Nothing after the breakpoint was compiled.

10.2 #compiler

The template:

```
// Not a comment
#compiler commentStartToken = '///'
// A comment
#compiler reset
// Not a comment
```

The output:

```
// Not a comment
// Not a comment
```

The generated code:

```
write('// Not a comment\n')
#  A comment
write('// Not a comment\n')
```

So this didn't affect the generated program, it just affected how the template definition was read.

11 Files

This chapter will be an overview of the files in the Cheetah package, and how they interrelate in compiling and filling a template. We'll also look at files in the Cheetah tarball that don't get copied into the package.

12 Template

This chapter will mainly walk through the `Cheetah.Template` constructor and not at what point the template is compiled.

(Also need to look at Transaction,py and Servlet.py .)

13 The parser

How templates are compiled: a walk through Parser.py's source. (Also need to look at Lexer.py, but not too closely.)

14 The compiler

How templates are compiled: a walk through Compiler.py .

15 History of Cheetah

In spring 2001, several members of the webware-discuss mailing list expressed the need for a template engine. Webware like Python is great for organizing analytical logic, but they both suffer when you need to do extensive variable interpolation into large pieces of text, or to build up a text string from its nested parts. Python's % operator gets you only so far, the syntax is cumbersome, and you have to use a separate format string for each nested part. Most of us had used template systems from other platforms–chiefly Zope's DTML, PHPLib's Template object and Java's Velocity–and wanted to port something like those so it could be used both in Webware servlets and in standalone Python programs.

Since I (Mike Orr) am writing this history, I'll describe how I encountered Cheetah. I had written a template module called PlowPlate based on PHPLib's Template library. Like PHPLib, it used regular expressions to search and destroy–er, replace–placeholders, behaved like a dictionary to specify placeholder values, contained no directives, but did have BEGIN and END markers which could be used to extract a named block (subtemplate). Meanwhile, Tavis Rudd was also on webware-discuss and interested in templates, and he lived just a few hours away. So 12 May 12, 2001 we met in Vancouver at a gelato shop on Denman Street and discussed Webware, and he drew on a napkin the outline of a template system he was working on.

Instead of filling the template by search-and-replace, he wanted to break it up into parts. This was a primitive form of template compiling: do the time-consuming work once and put it to a state where you can fill the template quickly multiple times. A template without directives happens to break down naturally into a list of alternating text/placeholder pairs. The odd subscript values are literal strings; the even subscripts are string keys into a dictionary of placeholder values. The project was called TemplateServer.

In a couple months, Tavis decided that instead of compiling to a list, he wanted to compile to Python source code: a series of `write` calls that would output onto a file-like object. This was the nucleus that became Cheetah. I thought that idea was stupid, but it turned out that this not-so-stupid idea blew the others out of the water in terms of performance.

Another thing Tavis pushed hard for from near the beginning was "display logic", or simple directives like `#for`, `#if` and `#echo`. (OK, `#echo` came later, but conceptually it belongs here. I thought display logic was even stupider than compiling to Python source code because it would just lead to "DTML hell"–complicated templates that are hard to read and maintain, and for which you have to learn (and debug) a whole new language when Python does it just fine. But others (hi Chuck!) had templates that were maintained by secretaries who didn't know Python, and the secretaries needed display logic, so that was that. Finally, after working with Cheetah templates (with display logic) and PlowPlate templates (with just blocks rather than display logic), I realized Tavis was smarter than I was and display logic really did belong in the template.

The next step was making directives for all the Python flow-control statements: `#while`, `#try`, `#assert`, etc. Some of them we couldn't think of a use for. Nevertheless, they were easy to code, and "somebody" would probably need them "someday", so we may as well implement them now.

During all this, Chuck Esterbrook, Ian Bicking and others offered (and still offer) their support and suggestions, and Chuck gave us feedback about his use of Cheetah–its first deployment in a commercial production environment. Later, Edmund Lian became our #1 bug reporter and suggester as he used Cheetah in his web applications.

A breakthrough came in fall 2001 when Tavis figured out how to implement the name mapper in C. The name mapper is what gives Cheetah its Autocalling and Uniform Dotted Notation features. This raised performance sufficiently to rewrite Cheetah in a totally "late binding" manner like Python is. More about this is in the next chapter.

We were going to release 1.0 in January 2002, but we decided to delay it until more people used it in real-world situations and gave us feedback about what is still needed. This has led to many refinements, and we have added (and removed) features according to this feedback. Nevertheless, Cheetah has been changing but stable since the late-binding rewrite in fall 2001, and anybody who keeps up with the cheetah-discuss mailing list will know when changes occur that require modifying one's template, and since most people use point releases rather than CVS, they generally have a few week's warning about any significant changes.

More detail on Cheetah's history and evolution, and why it is the way it is, can be found in our paper for the Python10

conference, http://www.cheetahtemplate.org/Py10.html.

16 Design Decisions and Tradeoffs

16.1 Delimiters

One of the first decisions we encountered was which delimiter syntax to use. We decided to follow Velocity's `$placeholder` and `#directive` syntax because the former is widely used in other languages for the same purpose, and the latter stands out in an HTML or text document. We also implemented the `${longPlaceholder}` syntax like the shells for cases where Cheetah or you might be confused where a placeholder ends. Tavis went ahead and made `$(longPlaceholder}` and `$[longPlaceholder]` interchangeable with it since it was trivial to implement. Finally, the `#compiler` directive allows you to change the delimiters if you don't like them or if they conflict with the text in your document. (Obviously, if your document contains a Perl program listing, you don't necessarily want to backslash each and every $ and #, do you?)

The choice of comment delimiters was more arbitrary. ## and `#* ...*#` doesn't match any language, but it's reminiscent of Python and C while also being consistent with our "# is for directives" convention.

We specifically chose *not* to use pseudo HTML tags for placeholders and directives, as described more thoroughly in the Cheetah Users' Guide introduction. Pseudo HTML tags may be easier to see in a visual editor (supposedly), but in text editors they're hard to distinguish from "real" HTML tags unless you look closely, and they're many more keystrokes to type. Also, if you make a mistake, the tag will show up as literal text in the rendered HTML page where it will be easy to notice and eradicate, rather than disappearing as bogus HTML tags do in browsers.

16.2 Late binding

One of Cheetah's unique features is the name mapper, which lets you write `$a.$b` without worrying much about the type of `a` or `b`. Prior to version 0.9.7, Cheetah did the entire NameMapper lookup at runtime. This provided maximum flexibility at the expense of speed. Doing a NameMapper lookup is intrinsically more expensive than an ordinary Python expression because Cheetah has to decide what type of container `a` is, whether the the value is a function (autocall it), issue the appropriate Python incantation to look up `b` in it, autocall again if necessary, and then convert the result to a string.

To maximize run-time (filling-time) performance, Cheetah 0.9.7 pushed much of this work back into the compiler. The compiler looked up `a` in the searchList at compile time, noted its type, and generated an eval'able Python expression based on that.

This approach had two significant drawbacks. What if `a` later changes type before a template filling? Answer: unpredictable exceptions occur. What if `a` does not exist in the searchList at compile time? Answer: the template can't compile.

To prevent these catastrophes, users were required to prepopulate the searchList before instantiating the template instance, and then not to change `a`'s type. Static typing is repugnant in a dynamic language like Python, and having to prepopulate the searchList made certain usages impossible. For example, you couldn't instantiate the template object without a searchList and then set `self` attributes to specify the values.

After significant user complaints about the fragility of this system, Tavis rewrote placeholder handling, and in version 0.9.8a3 (August 2001), Tavis moved the name mapper lookup back into runtime. Performance wasn't crippled because he discovered that writing a C version of the name mapper was easier than anticipated, and the C version completed the lookup quickly. Now Cheetah had "late binding", meaning the compiler does not look up `a` or care whether it exists. This allows users to create `a` or change its type anytime before a template filling.

The lesson we learned is that it's better to decide what you want and then figure out how to do it, rather than assuming that certain goals are unattainable due to performance considerations.

16.3 Caching framework

17 Patching Cheetah

How to commit changes to CVS or submit patches, how to run the test suite. Describe distutils and how the regression tests work.

17.1 File Requirements

The codeTemplate class contains not only the Cheetah infrastructure, but also some convenience methods useful in all templates. More methods may be added if it's generally agreed among Cheetah developers that the method is sufficiently useful to all types of templates, or at least to all types of HTML-output templates. If a method is too long to fit into `Template` – especially if it has helper methods – put it in a mixin class under `Cheetah.Utils` and inherit it.

Routines for a specific problem domain should be put under `Cheetah.Tools`, so that it doesn't clutter the namespace unless the user asks for it.

Remember: `Cheetah.Utils` is for objects required by any part of Cheetah's core. `Cheetah.Tools` is for completely optional objects. It should always be possible to delete `Cheetah.Tools` without breaking Cheetah's core services.

If a core method needs to look up an attribute defined under `Cheetah.Tools`, it should use `hasattr()` and gracefully provide a default if the attribute does not exist (meaning the user has not imported that subsystem).

17.2 Testing Changes and Building Regression Tests

Cheetah ships with a regression test suite. To run the built-in tests, execute at the shell prompt:

```
cheetah test
```

Before checking any changes in, run the tests and verify they all pass. That way, users can check out the CVS version of Cheetah at any time with a fairly high confidence that it will work. If you fix a bug or add a feature, please take the time to add a test that exploits the bug/feature. This will help in the future, to prevent somebody else from breaking it again without realizing it. Users can also run the test suite to verify all the features work on their particular platform and computer.

The general procedure for modifying Cheetah is as follows:

1. Write a simple Python program that exploits the bug/feature you're working on. You can either write a regression test (see below), or a separate program that writesthe template output to one file and put the expected output in another file; then you can run `diff` on the two outputs. (`diff` is a utility included on all Unix-like systems. It shows the differences between two files line by line. A precompiled Windows version is at http://gnuwin32.sourceforge.net/packages/diffutils.htm, and MacOS sources at http://perso.wanadoo.fr/gilles.depeyrot/DevTools_en.html.)
2. Make the change in your Cheetah CVS sandbox or in your installed version of Cheetah. If you make it in the sandbox, you'll have to run `python setup.py install` before testing it. If you make it in the installed version, do *not* run the installer or it will overwrite your changes!
3. Run `cheetah test` to verify you didn't break anything. Then run your little test program.
4. Repeat steps 2-3 until everything is correct.
5. Turn your little program into a regression test as described below.

6. When `cheetah test` runs cleanly with your regression test included, update the `CHANGES` file and check in your changes. If you made the changes in your installed copy of Cheetah, you'll have to copy them back into the CVS sandbox first. If you added any files that must be distributed, *be sure to* `cvs add` them before committing. Otherwise Cheetah will run fine on your computer but fail on anybody else's, and the test suite can't check for this.

7. Announce the change on the cheetahtemplate-discuss list and provide a tutorial if necessary. The documentation maintainer will update the Users' Guide and Developers' Guide based on this message and on the changelog.

If you add a directory to Cheetah, you have to mention it in `setup.py` or it won't be installed.

The tests are in the `Cheetah.Tests` package, aka the `src/Tests/` directory of your CVS sandbox. Most of the tests are in `SyntaxAndOutput.py`. You can either run all the tests or choose which to run:

`python Test.py` Run all the tests. (Equivalent to `cheetah test`.)

`python SyntaxAndOutput.py` Run only the tests in that module.

`python SyntaxAndOutput.py CGI` Run only the tests in the class `CGI` inside the module. The class must be a direct or indirect subclass of `unittest_local_copy.TestCase`.

`python SyntaxAndOutput.py CGI Indenter` Run the tests in classes `CGI` and `Indenter`.

`python SyntaxAndOutput.py CGI.test1` Run only test `test1`, which is a method in the `CGI` class.

etc...

To make a SyntaxAndOutput test, first see if your test logically fits into one of the existing classes. If so, simply add a method; e.g., `test16`. The method should not require any arguments except `self`, and should call `.verify(source, expectedOutput)`, where the two arguments are a template definition string and a control string. The tester will complain if the template output does not match the control string. You have a wide variety of placeholder variables to choose from, anything that's included in the `defaultTestNameSpace` global dictionary. If that's not enough, add items to the dictionary, but please keep it from being cluttered with wordy esoteric items for a single test).

If your test logically belongs in a separate class, create a subclass of `OutputTest`. You do not need to do anything else; the test suite will automatically find your class in the module. Having a separate class allows you to define state variables needed by your tests (see the `CGI` class) or override `.searchList()` (see the `Indenter` class) to provide your own searchList.

To modify another test module or create your own test module, you'll have to study the existing modules, the `unittest_local_copy` source, and the `unittest` documentation in the Python Library Reference. Note that we are using a hacked version of `unittest` to make a more convenient test structure for Cheetah. The differences between `unittest_local_copy` and Python's standard `unittest` are documented at the top of the module.

18 Documenting Cheetah

How to build the documentation. Why LaTeX, a minimum LaTeX reference, etc.

A A BNF Grammar of Cheetah

B Safe Delegation

Safe delegation, as provided by Zope and Allaire's Spectra, is not implemented in Cheetah. The core aim has been to help developers and template maintainers get things done, without throwing unnecessary complications in their way. So you should give write access to your templates only to those whom you trust. However, several hooks have been built into Cheetah so that safe delegation can be implemented at a later date.

It should be possible to implement safe delegation via a future configuration Setting `safeDelegationLevel` (0=none, 1=semi-secure, 2-alcatraz). This is not implemented but the steps are listed here in case somebody wants to try them out and test them.

Of course, you would also need to benchmark your code and verify it does not impact performance when safe delegation is off, and impacts it only modestly when it is on." All necessary changes can be made at compile time, so there should be no performance impact when filling the same TO multiple times.

1. Only give untrusted developers access to the .tmpl files. (Verifying what this means. Why can't trusted developers access them?)
2. Disable the `#attr` directive and maybe the `#set` directive.
3. Use Cheetah's directive validation hooks to disallow references to `self`, etc (e.g. `#if $steal(self.thePrivateVar)`)
4. Implement a validator for the $placeholders and use it to disallow '`__`' in $placeholders so that tricks like `$obj.__class__.__dict__` are not possible.