SimpleTALSix Documentation

Release 6.2.1

Jan Brohl

Dec 23, 2017

Contents

This is an implementation of TAL/METAL and TALES for Python 2.7 and 3.2+ (possibly working on other versions too)

SimpleTALSix is based on SimpleTAL 4.3.

CHAPTER 1

Known limitations

- When using `not:` on an empty expression the result will be true rather than an error.
- Path type `python:` is not supported by default. You can enable this by passing `allowPythonPath=1` to the Context constructor. Note that this should only be used when the authors of the templates are completely trusted, the code included in the `python:` path can do anything.
- TAL markup `on-error` is not yet supported.
- HTML Templates might have duplicate attributes if an attribute is added using `tal:attributes` and the name is in upper case. The cause of this is the HTMLParser implementation, which always converts attributes to lower case.

CHAPTER 2

Contents

2.1 TAL/TALES and METAL Reference Guide

A guide to using TAL, TALES, and METAL.

2.1.1 Introduction

This is a simple reference guide to the TAL and TALES languages. Formal language specifications are hosted by Zope: TAL , TALES , METAL .

2.1.2 TAL Commands

TAL consists of seven different commands (highest priority first):

- *tal:define*
- *tal:condition*
- *tal:repeat*
- *tal:content*
- *tal:replace*
- *tal:attributes*
- *tal:omit-tag*

Commands are attributes on HTML or XML tags, e.g. `<div tal:content="article">Article goes here</div>`

tal:define

Syntax: `tal:define="[local | global] name expression [; define-expression...]"`

Description: Sets the value of "name" to "expression". By default the name will be applicable in the "local" scope, which consists of this tag, and all other tags nested inside this tag. If the "global" keyword is used then this name will keep its value for the rest of the document.

Example:

```
<div tal:define="global title book/theTitle; local chapterTitle book/chapter/theTitle
↪">
```

tal:condition

Syntax: `tal:condition="expression"`

Description: If the expression evaluates to true then this tag and all its children will be output. If the expression evaluates to false then this tag and all its children will not be included in the output.

Example:

```
<h1 tal:condition="user/firstLogin">Welcome to this page!</h1>
```

tal:repeat

Syntax: `tal:repeat="name expression"`

Description: Evaluates "expression", and if it is a sequence, repeats this tag and all children once for each item in the sequence. The "name" will be set to the value of the item in the current iteration, and is also the name of the repeat variable. The repeat variable is accessible using the TAL path: repeat/name and has the following properties:

- index - Iteration number starting from zero
- number - Iteration number starting from one
- even - True if this is an even iteration
- odd - True if this is an odd iteration
- start - True if this is the first item in the sequence
- end - True if this is the last item in the sequence. For iterators this is never true
- length - The length of the sequence. For iterators this is maxint as the length of an iterator is unknown
- letter - The lower case letter for this iteration, starting at "a"
- Letter - Upper case version of letter
- roman - Iteration number in Roman numerals, starting at i
- Roman - Upper case version of roman

Example:

```
<table>
  <tr tal:repeat="fruit basket">
    <td tal:content="repeat/fruit/number"></td>
    <td tal:content="fruit/name"></td>
```

```
  </tr>
</table>
```

tal:content

Syntax: `tal:content="[text | structure] expression"`

Description: Replaces the contents of the tag with the value of "expression". By default, and if the "text" keyword is present, then the value of the expression will be escaped as required (i.e. characters "&<> will be escaped). If the "structure" keyword is present then the value will be output with no escaping performed.

Example:

```
<h1 tal:content="user/firstName"></h1>
```

tal:replace

Syntax: `tal:replace="[text | structure] expression"`

Description: Behaves identically to tal:content, except that the tag is removed from the output (as if tal:omit-tag had been used).

Example:

```
<h1>Welcome <b tal:replace="user/firstName"></b></h1>
```

tal:attributes

Syntax: `tal:attributes="name expression[;attributes-expression]"`

Description: Evaluates each "expression" and replaces the tag's attribute "name". If the expression evaluates to nothing then the attribute is removed from the tag. If the expression evaluates to default then the original tag's attribute is kept. If the "expression" requires a semi-colon then it must be escaped by using ";;".

Example:

```
<a tal:attributes="href user/homepage;title user/fullname">Your Homepage</a>
```

tal:omit-tag

Syntax: `tal:omit-tag="expression"`

Description: Removes the tag (leaving the tags content) if the expression evaluates to true. If expression is empty then it is taken as true.

Example:

```
<h1>
  <b tal:omit-tag="not:user/firstVisit">Welcome</b> to this page!
</h1>
```

2.1.3 TALES Expressions

The expressions used in TAL are called TALES expressions. The simplest TALES expression is a path which references a value, e.g. page/body references the body property of the page object.

path

Syntax: `[path:]string[|TALES Expression]`

Description: A path, optionally starting with the modifier 'path:', references a property of an object. The '/' delimiter is used to end the name of an object and the start of the property name. Properties themselves may be objects that in turn have properties. The 'I' ("or") character is used to find an alternative value to a path if the first path evaluates to 'Nothing' or does not exist.

Example:

```
<p tal:content="book/chapter/title | string:Untitled"></p>
```

There are several built in paths that can be used in paths:

- nothing - acts as None in Python
- default - keeps the existing value of the node (tag content or attribute value)
- options - the dictionary of values passed to the template (through the Context __init__ method)
- repeat - access the current repeat variable (see tal:repeat)
- attrs - a dictionary of original attributes of the current tag
- CONTEXTS - a dictionary containing all of the above

exists

Syntax: `exists:path`

Description: Returns true if the path exists, false otherwise. This is particularly useful for removing tags from output when the tags will have no content.

Example:

```
<p tal:omit-tag="not:exists:book/chapter/title" tal:content="book/chapter/title"></p>
```

nocall

Syntax: `nocall:path`

Description: Returns a reference to a path, but without evaluating the path. Useful when you wish to define a new name to reference a function, not the current value of a function.

Example:

```
<p tal:define="title nocall:titleFunction" tal:content="title"></p>
```

not

Syntax: `not:tales-path`

Description: Returns the inverse of the tales-path. If the path returns true, `not:path` will return false.

Example:

```
<p tal:condition="not: user/firstLogin">Welcome to the site!</p>
```

string

Syntax: `string:text`

Description: Evaluates to a literal string with value text while substituting variables with the form `${pathName}` and `$pathName`

Example:

```
<b tal:content="string:Welcome ${user/name}!"></b>
```

python

Syntax: `python:python-code`

Description: Evaluates the python-code and returns the result. The python code must be properly escaped, e.g. "python: 1 < 2" must be written as "python: 1 < 2". The python code has access to all Python functions, including four extra functions that correspond to their TALES commands: path (string), string (string), exists (string), and nocall (string)

Example:

```
<div tal:condition="python: path (basket/items) &gt; 1">Checkout!</div>
```

2.1.4 METAL Macro Language

METAL is a macro language commonly used with TAL and TALES. METAL allows part of a template to be used as a macro in later parts of a template, or a separate template altogether.

metal:define-macro

Syntax: `metal:define-macro="name"`

Description: Defines a new macro that can be reference later as "name".

Example:

```
<div metal:define-macro="footer">
  Copyright <span tal:content="page/lastModified">2004</span>
</div>
```

metal:use-macro

Syntax: `metal:use-macro="expression"`

Description: Evaluates "expression" and uses this as a macro.

Example:

```
<div metal:use-macro="footer"></div>
```

metal:define-slot

Syntax: `metal:define-slot="name"`

Description: Defines a customisation point in a macro with the given name.

Example:

```
<div metal:define-macro="footer">
  <b>Standard disclaimer for the site.</b>
  <i metal:define-slot="Contact">Contact admin@site.com</i>
</div>
```

metal:fill-slot

Syntax: `metal:fill-slot="name"`

Description: Replaces the content of a slot with this element.

Example:

```
<div metal:use-macro="footer">
  <i metal:fill-slot="Contact">Contact someone else</i>
</div>
```

2.2 Examples

2.2.1 Basic usage

As simple as it gets:

1. Create a context
2. Compile a template
3. Expand the template

Code

```
from simpletal import simpleTAL, simpleTALES
import sys

# Creat the context that is used by the template
```

```
context = simpleTALES.Context(allowPythonPath=1)

# Add a string to the context under the variable title
context.addGlobal("title", "Colours of the rainbow")

# A list of strings
colours = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]
# Add the list to the context under the variable rainbow
context.addGlobal("rainbow", colours)

# Open the template file
templateFile = open("basic.html", 'r')

# Compile a template
template = simpleTAL.compileHTMLTemplate(templateFile)

# Close the template file
templateFile.close()

# Expand the template as HTML using this context
template.expand(context, sys.stdout, outputEncoding="utf-8")
```

basic.html

```
<html>
        <body>
                <h1 tal:content="title">The title</h1>
                <ul tal:repeat="colour rainbow">
                  <li tal:content="colour">Colour of the rainbow</li>
                  One Two
                  <p tal:content="python: str (colour) + ' is the colour'"></p>
                </ul>
  </body>
</html>
```

2.2.2 CGI example

A demonstration of how TAL/TALES can be used from a cgi program. Quick instructions:

1. Copy this file and the two templates ("fields.html" and "results.html") to the cgi-bin directory on your webserver
2. Ensure that simpleTAL, simpleTALES and DummyLogger are installed in your site-packages directory
3. Go to http://servername/cgi-bin/simple-cgi.py

Code

```
from simpletal import simpleTAL, simpleTALES
import cgi
import sys
```

```
class ExampleCGI:

    def __init__(self):
        self.missingFields = {}
        self.fieldValues = {}
        self.form = cgi.FieldStorage()
        self.formValid = 1
        self.context = simpleTALES.Context()

    def buildContext(self, title):
        self.context.addGlobal("missingFields", self.missingFields)
        self.context.addGlobal("fieldValues", self.fieldValues)
        self.context.addGlobal("title", title)

    def getValue(self, name, mandatory=1):
        if (self.form.has_key(name)):
            self.fieldValues[name] = self.form[name].value
        elif (mandatory):
            self.missingFields[name] = 1
            self.formValid = 0

    def main(self):
        if (self.form.has_key("submit")):
            # Recieved the posting, get the name, occupation, and (optional)
            # age
            self.getValue("username")
            self.getValue("occupation")
            self.getValue("age", mandatory=0)

            if (self.formValid):
                # Valid form, show the results
                self.buildContext("Valid Results")
                self.expandTemplate("results.html")
            else:
                self.buildContext("Missing fields")
                self.expandTemplate("fields.html")
        else:
            self.buildContext("Enter data")
            self.expandTemplate("fields.html")

    def expandTemplate(self, templateName):
        # Print out the headers
        sys.stdout.write("Content-Type: text/html\n")      # HTML is following
        sys.stdout.write("\n")                            # blank line, end of headers

        # Expand the template and print it out
        templateFile = open(templateName, 'r')
        template = simpleTAL.compileHTMLTemplate(templateFile)

        # Close the template file
        templateFile.close()

        # Expand the template as HTML using this context
        template.expand(self.context, sys.stdout)

        sys.exit(0)
```

```
# Entry point for the cgi
cgiInstance = ExampleCGI()
cgiInstance.main()
```

fields.html

```
<!-- Example TAL HTML Template - this is used to gather data -->
<html>
  <head>
  <title tal:content="title">Title</title>
  </head>

  <body>
    <h1 tal:content="title">Title</h1>
    <form action="simple-cgi.py">
    <div>
        Name: <input name="username" tal:attributes="value fieldValues/username"></
↪input>
                <b tal:condition="missingFields/username">(Please fill in)</b><br>

        Occupation: <input name="occupation" tal:attributes="value fieldValues/
↪occupation"></input>
          <b tal:condition="missingFields/occupation">(Please fill in)</b><br>

        Age: <input name="age" tal:attributes="value fieldValues/age"></input><br>

    </div>
    <input type="submit" name="submit" value="submit">
    </form>
  </body>
</html>
```

results.html

```
<!-- Example TAL HTML Template - this is used to show results -->
<html>
  <head>
  <title tal:content="title">Title</title>
  </head>

  <body>
    <h1 tal:content="title">Title</h1>
    <div>
        Name: <b tal:content="fieldValues/username">Your name</b><br>
        Occupation: <b tal:content="fieldValues/occupation">Your occupation</b><br>
        <div tal:condition="fieldValues/age">Age:
                <b tal:content="fieldValues/age">Your age</b></div><br>
    </div>
  </body>
</html>
```

2.2.3 METAL example

An example of how to use METAL.

Code

```
from simpletal import simpleTAL, simpleTALES
import sys

# Creat the context that is used by the template
context = simpleTALES.Context()

# Add a string to the context under the variable title
context.addGlobal("title", "Simple METAL Example")

# Compile the macro pages
templateFile = open("macro.html", 'r')
macros = simpleTAL.compileHTMLTemplate(templateFile)
templateFile.close()

# Add the macros page to the Context
context.addGlobal("sitemacros", macros)

# Now compile the page which will use the macros
templateFile = open("page.html", 'r')
page = simpleTAL.compileHTMLTemplate(templateFile)
templateFile.close()

# Expand the page using this context
page.expand(context, sys.stdout)
```

macro.html

```
<html>
        <body>
                <h1 tal:content="title">The title</h1>
                <p metal:define-macro="notice">
                This macro is brought to you by <b metal:define-slot="source">the␣
↪colour
                blue</b>.
                Now you know.
                </p>
  </body>
</html>
```

page.html

```
<html>
        <body>
                <h1 tal:content="title">The title</h1>
                <div metal:use-macro="sitemacros/macros/notice">
                  <i metal:fill-slot="source">SimpleTAL!</i>
```

```
                </div>
  </body>
</html>
```

2.2.4 Structure example

This shows how to include structure into a template, and how multiple templates can be embedded within each other.

Code

```
from simpletal import simpleTAL, simpleTALES
import sys

# Create the context that is used by the template
context = simpleTALES.Context()
context.addGlobal("title", "Hello World")
context.addGlobal("author", "Colin Stewart")

# A list that contains a dictionary
chapters = [{"heading": "Introduction", "text": "Some <b>text</b> here"}, {"heading":
↪"Details", "text": "Notice tags are preserved."}
           ]

advancedText = 'Structured text can contain other templates like this - written by <b␣
↪tal:replace="author">Me</b>'

chapters.append(
    {"heading": "Advanced", "text": simpleTAL.compileHTMLTemplate(advancedText)})

context.addGlobal("doc", chapters)

templateFile = open("structure.html", 'r')
template = simpleTAL.compileHTMLTemplate(templateFile)

templateFile.close()

template.expand(context, sys.stdout)
```

structure.html

```
<html>
        <body>
                <h1 tal:content="title">The title</h1>
                <div tal:repeat="chapters doc">
                  <h2 tal:content="chapters/heading">Chapter Heading</h2>
                  <p tal:content="structure chapters/text">Text</p>
                </div>
  </body>
</html>
```

2.3 simpletal package

2.3.1 Submodules

2.3.2 simpletal.simpleTAL module

simpleTAL Interpreter

The classes in this module implement the TAL language, expanding both XML and HTML templates.

class simpletal.simpleTAL.**LexicalHandler**
Bases: object

Dummy lexical handdler class

class simpletal.simpleTAL.**TemplateInterpreter**
Bases: object

tagAsText(*tag_atts*, *singletonFlag=0*)
This returns a tag as text.

initialise(*context*, *outputFile*)

cleanState()

popProgram()

pushProgram()

execute(*template*)

cmdDefine(*command*, *args*)
args: [(isLocalFlag (Y/n), variableName, variablePath),...] Define variables in either the local or global context

cmdCondition(*command*, *args*)
args: expression, endTagSymbol Conditionally continues with execution of all content contained by it.

cmdRepeat(*command*, *args*)
args: (varName, expression, endTagSymbol) Repeats anything in the cmndList

cmdContent(*command*, *args*)
args: (replaceFlag, structureFlag, expression, endTagSymbol) Expands content

cmdAttributes(*command*, *args*)
args: [(attributeName, expression)] Add, leave, or remove attributes from the start tag

cmdOmitTag(*command*, *args*)
args: expression Conditionally turn off tag output

cmdOutputStartTag(*command*, *args*)

cmdEndTagEndScope(*command*, *args*)

cmdOutput(*command*, *args*)

cmdStartScope(*command*, *args*)
args: (originalAttributes, currentAttributes) Pushes the current state onto the stack, and sets up the new state

cmdNoOp(*command*, *args*)

cmdUseMacro(*command*, *args*)
args: (macroExpression, slotParams, endTagSymbol) Evaluates the expression, if it resolves to a SubTemplate it then places the slotParams into currentSlots and then jumps to the end tag

cmdDefineSlot(*command*, *args*)
args: (slotName, endTagSymbol) If the slotName is filled then that is used, otherwise the original conent is used.

class simpletal.simpleTAL.**HTMLTemplateInterpreter**(*minimizeBooleanAtts=0*)
Bases: *simpletal.simpleTAL.TemplateInterpreter*

tagAsTextMinimizeAtts(*tag_atts*, *singletonFlag=0*)
This returns a tag as text.

class simpletal.simpleTAL.**Template**(*commands*, *macros*, *symbols*, *doctype=None*)
Bases: object

expand(*context*, *outputFile*, *outputEncoding=None*, *interpreter=None*)
This method will write to the outputFile, using the encoding specified, the expanded version of this template. The context passed in is used to resolve all expressions with the template.

expandInline(*context*, *outputFile*, *interpreter=None*)
Internally used when expanding a template that is part of a context.

getProgram()
Returns a tuple of (commandList, startPoint, endPoint, symbolTable)

class simpletal.simpleTAL.**SubTemplate**(*startRange*, *endRangeSymbol*)
Bases: *simpletal.simpleTAL.Template*

A SubTemplate is part of another template, and is used for the METAL implementation. The two uses for this class are:

1 - metal:define-macro results in a SubTemplate that is the macro 2 - metal:fill-slot results in a SubTemplate that is a parameter to metal:use-macro

setParentTemplate(*parentTemplate*)

getProgram()
Returns a tuple of (commandList, startPoint, endPoint, symbolTable)

class simpletal.simpleTAL.**HTMLTemplate**(*commands*, *macros*, *symbols*, *doctype=None*, *minimizeBooleanAtts=0*)
Bases: *simpletal.simpleTAL.Template*

A specialised form of a template that knows how to output HTML

expand(*context*, *outputFile*, *outputEncoding=u'utf-8'*, *interpreter=None*)
This method will write to the outputFile, using the encoding specified, the expanded version of this template. The context passed in is used to resolve all expressions with the template.

expandInline(*context*, *outputFile*, *interpreter=None*)
Ensure we use the HTMLTemplateInterpreter

class simpletal.simpleTAL.**XMLTemplate**(*commands*, *macros*, *symbols*, *doctype=None*)
Bases: *simpletal.simpleTAL.Template*

A specialised form of a template that knows how to output XML

expand(*context*, *outputFile*, *outputEncoding=u'utf-8'*, *docType=None*, *suppressXMLDeclaration=0*, *interpreter=None*)
This method will write to the outputFile, using the encoding specified, the expanded version of this template. The context passed in is used to resolve all expressions with the template.

class simpletal.simpleTAL.**TemplateCompiler**
Bases: object

setTALPrefix(*prefix*)

setMETALPrefix(*prefix*)

popTALNamespace()

popMETALNamespace()

tagAsText(*tag_atts*, *singletonFlag=0*)
This returns a tag as text.

getTemplate()

addCommand(*command*)

addTag(*tag*, *tagProperties={}*)
Used to add a tag to the stack. Various properties can be passed in the dictionary as being information required by the tag. Currently supported properties are:

'command' - The (command,args) tuple associated with this command 'originalAtts' - The original attributes that include any metal/tal attributes 'endTagSymbol' - The symbol associated with the end tag for this element 'popFunctionList' - A list of functions to execute when this tag is popped

'singletonTag' - A boolean to indicate that this is a singleton flag

popTag(*tag*, *omitTagFlag=0*)
omitTagFlag is used to control whether the end tag should be included in the output or not. In HTML 4.01 there are several tags which should never have end tags, this flag allows the template compiler to specify that these should not be output.

parseStartTag(*tag*, *attributes*, *singletonElement=0*)

parseEndTag(*tag*)
Just pop the tag and related commands off the stack.

parseData(*data*)

compileCmdDefine(*argument*)

compileCmdCondition(*argument*)

compileCmdRepeat(*argument*)

compileCmdContent(*argument*, *replaceFlag=0*)

compileCmdReplace(*argument*)

compileCmdAttributes(*argument*)

compileCmdOmitTag(*argument*)

compileMetalUseMacro(*argument*)

compileMetalDefineMacro(*argument*)

compileMetalFillSlot(*argument*)

compileMetalDefineSlot(*argument*)

exception simpletal.simpleTAL.**TemplateParseException**(*location*, *errorDescription*)
Bases: exceptions.Exception

class simpletal.simpleTAL.**HTMLTemplateCompiler**
Bases: *simpletal.simpleTAL.TemplateCompiler*, HTMLParser.HTMLParser

parseTemplate(*file*, *encoding=u'UTF-8-SIG'*, *minimizeBooleanAtts=0*)

tagAsText(*tag_atts*, *singletonFlag=0*)
This returns a tag as text.

handle_startendtag(*tag*, *attributes*)

handle_starttag(*tag*, *attributes*)

handle_endtag(*tag*)

handle_data(*data*)

handle_charref(*ref*)

handle_entityref(*ref*)

handle_decl(*data*)

handle_comment(*data*)

handle_pi(*data*)

report_unbalanced(*tag*)

getTemplate()

class simpletal.simpleTAL.**XMLTemplateCompiler**
Bases: *simpletal.simpleTAL.TemplateCompiler*, xml.sax.handler.ContentHandler, xml.sax.handler.DTDHandler, *simpletal.simpleTAL.LexicalHandler*

parseTemplate(*file*)

parseDOM(*dom*)

startDTD(*name*, *public_id*, *system_id*)

startElement(*tag*, *attributes*)

endElement(*tag*)

skippedEntity(*name*)

characters(*data*)

processingInstruction(*target*, *data*)

comment(*data*)

getTemplate()

simpletal.simpleTAL.**compileHTMLTemplate**(*template*, *inputEncoding=u'UTF-8-SIG'*, *minimizeBooleanAtts=0*)
Reads the templateFile and produces a compiled template. To use the resulting template object call:

template.expand (context, outputFile)

simpletal.simpleTAL.**compileXMLTemplate**(*template*)
Reads the templateFile and produces a compiled template. To use the resulting template object call:

template.expand (context, outputFile)

simpletal.simpleTAL.**compileDOMTemplate**(*template*)
Traverses the DOM and produces a compiled template. To use the resulting template object call:

template.expand (context, outputFile)

2.3.3 simpletal.simpleTALConstants module

simpletal.simpleTALConstants.**METAL_NAME_URI = u'http://xml.zope.org/namespaces/metal'**
METAL namespace URI

simpletal.simpleTALConstants.**TAL_NAME_URI = u'http://xml.zope.org/namespaces/tal'**
TAL namespace URI

simpletal.simpleTALConstants.**TAL_DEFINE = 1**
Argument – [(isLocalFlag (Y/n), variableName, variablePath),. . .]

simpletal.simpleTALConstants.**TAL_CONDITION = 2**
Argument – expression, endTagSymbol

simpletal.simpleTALConstants.**TAL_REPEAT = 3**
Argument – (varname, expression, endTagSymbol)

simpletal.simpleTALConstants.**TAL_CONTENT = 4**
Argument – (replaceFlag, type, expression)

simpletal.simpleTALConstants.**TAL_REPLACE = 5**
Not used in byte code, only ordering.

simpletal.simpleTALConstants.**TAL_ATTRIBUTES = 6**
Argument – [(attributeName, expression)]

simpletal.simpleTALConstants.**TAL_OMITTAG = 7**
Argument – expression

simpletal.simpleTALConstants.**TAL_START_SCOPE = 8**
Argument – (originalAttributeList, currentAttributeList)

simpletal.simpleTALConstants.**TAL_OUTPUT = 9**
Argument – String to output

simpletal.simpleTALConstants.**TAL_STARTTAG = 10**
Argument – None

simpletal.simpleTALConstants.**TAL_ENDTAG_ENDSCOPE = 11**
Argument – Tag, omitTagFlag

simpletal.simpleTALConstants.**TAL_NOOP = 13**
Argument – None

simpletal.simpleTALConstants.**METAL_USE_MACRO = 14**
Argument – expression, slotParams, endTagSymbol

simpletal.simpleTALConstants.**METAL_DEFINE_SLOT = 15**
Argument – macroName, endTagSymbol

simpletal.simpleTALConstants.**METAL_FILL_SLOT = 16**
Only used for parsing

simpletal.simpleTALConstants.**METAL_DEFINE_MACRO = 17**
Only used for parsing

simpletal.simpleTALConstants.**HTML4_VOID_ELEMENTS = frozenset([u'IMG', u'AREA', u'BASEFONT',**
The set of elements in HTML4 that can not have end tags

Source: http://www.w3.org/TR/html401/index/elements.html

simpletal.simpleTALConstants.**HTML5_VOID_ELEMENTS = frozenset([u'WBR', u'IMG', u'AREA', u'HR**
The set of elements in HTML5 that can not have end tags

Source: http://www.w3.org/TR/html-markup/syntax.html#void-element

simpletal.simpleTALConstants.**HTML_FORBIDDEN_ENDTAG = frozenset([u'WBR', u'IMG', u'BASEFONT**
The set of elements in HTML5 that can not have end tags

simpletal.simpleTALConstants.**HTML_BOOLEAN_ATTS = frozenset([(u'SCRIPT', u'DEFER'), (u'INPU**
Set of element – attribute pairs that can use minimized form in HTML

class simpletal.simpleTALConstants.**SignalValue**(*info*)
Bases: object

Helper class to make unique values with a useful __str__

2.3.4 simpletal.simpleTALES module

simpleTALES Implementation

The classes in this module implement the TALES specification, used by the simpleTAL module.

exception simpletal.simpleTALES.**PathNotFoundException**
Bases: exceptions.Exception

exception simpletal.simpleTALES.**ContextContentException**
Bases: exceptions.Exception

This is raised when invalid content has been placed into the Context object. For example using non-ascii characters instead of Unicode strings.

exception simpletal.simpleTALES.**ContextVariable**(*value=None*)
Bases: exceptions.Exception

value(*currentPath=None*)

rawValue()

exception simpletal.simpleTALES.**RepeatVariable**(*sequence*)
Bases: *simpletal.simpleTALES.ContextVariable*

To be written

value(*currentPath=None*)

rawValue()

getCurrentValue()

increment()

createMap()

getIndex()

getNumber()

getEven()

getOdd()

getStart()

getEnd()

getLowerLetter()

getUpperLetter()

getLowerRoman()

getUpperRoman()

exception simpletal.simpleTALES.**IteratorRepeatVariable**(*sequence*)
Bases: *simpletal.simpleTALES.RepeatVariable*

getCurrentValue()

increment()

createMap()

getLength()

getEnd()

exception simpletal.simpleTALES.**PathFunctionVariable**(*func*)
Bases: *simpletal.simpleTALES.ContextVariable*

This class wraps a callable object (e.g. function) so that it can receive part of a TAL path as it's argument.

To use this simply create a new instance of the PathFunctionVariable and then place this into the Context (see above). The path passed to the function is that part of the path not already used. For example if the function "helloFunc" is placed in the Context the path "helloFunc/an/example" results in the string "an/example" being passed to the function.

value(*currentPath=None*)

exception simpletal.simpleTALES.**CachedFuncResult**(*value=None*)
Bases: *simpletal.simpleTALES.ContextVariable*

This class wraps a callable object (e.g. function) so that the callable is only called once.

In normal SimpleTAL operation any function placed into a Context might be called multiple times during template expansion. To ensure that it is only called once simply wrap in the CachedFuncResult object first.

value(*currentPath=None*)

clearCache()
Clears the cache.

Use this to clear the cache between multiple template expansions if the callable should be executed once per template expansion.

class simpletal.simpleTALES.**PythonPathFunctions**(*context*)
Bases: object

path(*expr*)

string(*expr*)

exists(*expr*)

nocall(*expr*)

test(**arguments*)

class simpletal.simpleTALES.**Context**(*options=None*, *allowPythonPath=False*)
Bases: object

addRepeat(*name*, *var*, *initialValue*)

removeRepeat(*name*)

addGlobal(*name*, *value*)
Adds the value to the context under name.

Value can either be a fundamental python data type or a callable object.

pushLocals()

setLocal(*name*, *value*)

popLocals()

evaluate(*expr*, *originalAtts=None*)

evaluatePython(*expr*)

evaluatePath(*expr*)

evaluateExists(*expr*)

evaluateNoCall(*expr*)

evaluateNot(*expr*)

evaluateString(*expr*)

traversePath(*expr*, *canCall=1*)

populateDefaultVariables(*options*)

2.3.5 simpletal.simpleTALUtils module

simpleTALUtils

This module is holds utilities that make using SimpleTAL easier. Initially this is just the HTMLStructureCleaner class, used to clean up HTML that can then be used as 'structure' content.

class simpletal.simpleTALUtils.**TemplateCache**
Bases: object

A TemplateCache is a multi-thread safe object that caches compiled templates. This cache only works with file based templates, the mtime of the file is checked on each hit, if the file has changed the template is re-compiled.

isHTML(*name*)

getTemplate(*name*, *inputEncoding='UTF-8-SIG'*)
Name should be the path of a template file. If self.isHTML(name) it is treated as an HTML Template, otherwise it's treated as an XML Template. If the template file has changed since the last cache it will be re-compiled.

inputEncoding is only used for HTML templates, and should be the encoding that the template is stored in.

getXMLTemplate(*name*)
Name should be the path of an XML template file.

class simpletal.simpleTALUtils.**TemplateRoot**(*rootPath*, *loadFunc*, *templateExt='.html'*)
Bases: object

Simple-to-use templating. Interface not yet fully stable.

expand(*templatePath*, *options=None*, *addGlobals={}*)

expandMacros(*templatePath*, *options=None*, *addGlobals={}*)

get(*templatePath*)

resolvePath(*subpath*)

getForContext(*subpath=None*)

class simpletal.simpleTALUtils.**MacroExpansionInterpreter**
Bases: *simpletal.simpleTAL.TemplateInterpreter*

A MacroExpansionInterpreter only expands METAL macros but does not touch TAL.

popProgram()

pushProgram()

cmdOutputStartTag(*command*, *args*)

cmdUseMacro(*command*, *args*)

cmdEndTagEndScope(*command*, *args*)

simpletal.simpleTALUtils.**expandMacros**(*context*, *template*, *outputFile*, *outputEncoding='utf-8'*)
This function can be used to expand a template which contains METAL macros, while leaving in place all the TAL and METAL commands.

Doing this makes editing a template which uses METAL macros easier, because the results of the macro can be seen immediately. The macros referred to by the passed in template must be present in the context so that their contents can be referenced. The outputEncoding determines the encoding of the returned string, which will contain the expanded macro.

simpletal.simpleTALUtils.**ExpandMacros**(*context*, *template*, *outputEncoding='utf-8'*)
This legacy function does the same as expandMacros but returns a string instead of writing to a file.

2.3.6 Module contents

2.4 License

SimpleTALSix is based on SimpleTAL so scroll down and have a look at its license, too.

2.4.1 SimpleTALSix

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.

2.4.2 SimpleTAL 4.3

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. 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.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘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 AUTHOR 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.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

A

C

E

G

H

I

L

M

N

P

R

S

T

V

X