Chameleon Documentation

Release 3.3

Malthe Borch et. al

May 23, 2018

Contents

Chameleon is an HTML/XML template engine for Python.

It's designed to generate the document output of a web application, typically HTML markup or XML.

The language used is *page templates*, originally a Zope invention[1], but available here as a *standalone library* that you can use in any script or application running Python 2.5 and up (including 3.x and pypy). It comes with a set of *new features*, too.

The template engine compiles templates into Python byte-code and is optimized for speed. For a complex template language, the performance is *very good*.

Found a bug? Please report issues to the issue tracker.

Need help? Post to the Pylons discussion list or join the `#pyramid` channel on Freenode IRC.

[1] The template language specifications and API for the Page Templates engine are based on Zope Page Templates (see in particular zope.pagetemplate). However, the Chameleon compiler and Page Templates engine is an entirely new codebase, packaged as a standalone distribution. It does not require a Zope software environment.

CHAPTER 1

Getting the code

You can download the package from the Python package index or install the latest release using setuptools or the newer distribute (required for Python 3.x):

```
$ easy_install Chameleon
```

There are no required library dependencies on Python 2.7 and up[2]. On 2.5 and 2.6, the ordereddict and unittest2 packages are set as dependencies.

The project is hosted in a GitHub repository. Code contributions are welcome. The easiest way is to use the pull request interface.

[2] The translation system in Chameleon is pluggable and based on gettext. There is built-in support for the zope.i18n package. If this package is installed, it will be used by default. The translationstring package offers some of the same helper and utility classes, without the Zope application interface.

CHAPTER 2

Introduction

The *page templates* language is used within your document structure as special element attributes and text markup. Using a set of simple language constructs, you control the document flow, element repetition, text replacement and translation.

Note: If you've used page templates in a Zope environment previously, note that Chameleon uses Python as the default expression language (instead of *path* expressions).

The basic language (known as the *template attribute language* or TAL) is simple enough to grasp from an example:

```
<html>
  <body>
    <h1>Hello, ${'world'}!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
           ${row.capitalize()} ${col}
        </td>
      </tr>
    </table>
  </body>
</html>
```

The `${...}` notation is short-hand for text insertion[3]. The Python-expression inside the braces is evaluated and the result included in the output. By default, the string is escaped before insertion. To avoid this, use the `structure:` prefix:

```
<div>${structure: ...}</div>
```

Note that if the expression result is an object that implements an `__html__()` method[4], this method will be called and the result treated as "structure". An example of such an object is the `Markup` class that's included as a utility:

[3] This syntax was taken from Genshi.

[4] See the WebHelpers library which provide a simple wrapper around this method.

```
from chameleon.utils import Markup
username = Markup("<tt>%s</tt>" % username)
```

The macro language (known as the *macro expansion language* or METAL) provides a means of filling in portions of a generic template.

On the left, the macro template; on the right, a template that loads and uses the macro, filling in the "content" slot:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example &mdash; ${document.title}</title>
  </head>
  <body>
    <h1>${document.title}</h1>

    <div id="content">
      <metal:content define-slot="content" />
    </div>
  </body>
</html>
```

```
<metal:main use-macro="load:main.pt">

   <p metal:fill-slot="content">${structure: document.body}</p>

</metal:main>
```

In the example, the expression type *load* is used to retrieve a template from the file system using a path relative to the calling template.

The METAL system works with TAL such that you can for instance fill in a slot that appears in a `tal:repeat` loop, or refer to variables defined using `tal:define`.

The third language subset is the translation system (known as the *internationalization language* or I18N):

```
<html i18n:domain="example">

  ...

  <div i18n:translate="">
     You have <span i18n:name="amount">${round(amount, 2)}</span> dollars in your account.
  </div>

  ...

</html>
```

Each translation message is marked up using `i18n:translate` and values can be mapped using `i18n:name`. Attributes are marked for translation using `i18n:attributes`. The template engine generates gettext translation strings from the markup:

```
"You have ${amount} dollars in your account."
```

If you use a web framework such as Pyramid, the translation system is set up automatically and will negotiate on a *target language* based on the HTTP request or other parameter. If not, then you need to configure this manually.

CHAPTER 3

Next steps

This was just an introduction. There are a number of other basic statements that you need to know in order to use the language. This is all covered in the *language reference*.

If you're already familiar with the page template language, you can skip ahead to the *getting started* section to learn how to use the template engine in your code.

To learn about integration with your favorite web framework see the section on *framework integration*.

CHAPTER 4

License

This software is made available under a BSD-like license.

CHAPTER 5

Contents

5.1 Library Documentation

This section documents the package as a Python library. To learn about the page template language, consult the *language reference*.

5.1.1 Getting started

There are several template constructor classes available, one for each of the combinations *text* or *xml*, and *string* or *file*.

The file-based constructor requires an absolute path. To set up a templates directory *once*, use the template loader class:

```
import os

path = os.path.dirname(__file__)

from chameleon import PageTemplateLoader
templates = PageTemplateLoader(os.path.join(path, "templates"))
```

Then, to load a template relative to the provided path, use dictionary syntax:

```
template = templates['hello.pt']
```

Alternatively, use the appropriate template class directly. Let's try with a string input:

```
from chameleon import PageTemplate
template = PageTemplate("<div>Hello, ${name}.</div>")
```

All template instances are callable. Provide variables by keyword argument:

```
>>> template(name='John')
'<div>Hello, John.</div>'
```

5.1.2 Performance

The template engine compiles (or *translates*) template source code into Python byte-code. In simple templates this yields an increase in performance of about 7 times in comparison to the reference implementation.

In benchmarks for the content management system Plone, switching to Chameleon yields a request to response improvement of 20-50%.

5.1.3 Extension

You can extend the language through the expression engine by writing your own expression compiler.

Let's try and write an expression compiler for an expression type that will simply uppercase the supplied value. We'll call it `upper`.

You can write such a compiler as a closure:

```
import ast

def uppercase_expression(string):
    def compiler(target, engine):
        uppercased = self.string.uppercase()
        value = ast.Str(uppercased)
        return [ast.Assign(targets=[target], value=value)]
    return compiler
```

To make it available under a certain prefix, we'll add it to the expression types dictionary.

```
from chameleon import PageTemplate
PageTemplate.expression_types['upper'] = uppercase_expression
```

Alternatively, you could subclass the template class and set the attribute `expression_types` to a dictionary that includes your expression:

```
from chameleon import PageTemplateFile
from chameleon.tales import PythonExpr

class MyPageTemplateFile(PageTemplateFile):
    expression_types = {
        'python': PythonExpr,
        'upper': uppercase_expression
        }
```

You can now uppercase strings *natively* in your templates:

```
<div tal:content="upper: hello, world" />
```

It's probably best to stick with a Python expression:

```
<div tal:content="'hello, world'.upper()" />
```

5.1.4 API reference

This section describes the documented API of the library.

Templates

Use the `PageTemplate*` template classes to define a template from a string or file input:

class chameleon.**PageTemplate**(*body*, ***config*)

Constructor for the page template language.

Takes a string input as the only positional argument:

```
template = PageTemplate("<div>Hello, ${name}.</div>")
```

Configuration (keyword arguments):

`auto_reload`

Enables automatic reload of templates. This is mostly useful in a development mode since it takes a significant performance hit.

`default_expression`

Set the default expression type. The default setting is `python`.

`encoding`

The default text substitution value is a unicode string on Python 2 or simply string on Python 3.

Pass an encoding to allow encoded byte string input (e.g. UTF-8).

`literal_false`

Attributes are not dropped for a value of `False`. Instead, the value is coerced to a string.

This setting exists to provide compatibility with the reference implementation.

`boolean_attributes`

Attributes included in this set are treated as booleans: if a true value is provided, the attribute value is the attribute name, e.g.:

```
boolean_attributes = {"selected"}
```

If we insert an attribute with the name "selected" and provide a true value, the attribute will be rendered:

```
selected="selected"
```

If a false attribute is provided (including the empty string), the attribute is dropped.

The special return value `default` drops or inserts the attribute based on the value element attribute value.

`translate`

Use this option to set a translation function.

Example:

```
def translate(msgid, domain=None, mapping=None, default=None,␣
↪context=None):
    ...
    return translation
```

Note that if `target_language` is provided at render time, the translation function must support this argument.

`implicit_i18n_translate`

Enables implicit translation for text appearing inside elements. Default setting is `False`.

While implicit translation does work for text that includes expression interpolation, each expression must be simply a variable name (e.g. `${foo}`); otherwise, the text will not be marked for translation.

`implicit_i18n_attributes`

Any attribute contained in this set will be marked for implicit translation. Each entry must be a lowercase string.

Example:

```
implicit_i18n_attributes = set(['alt', 'title'])
```

`strict`

Enabled by default. If disabled, expressions are only required to be valid at evaluation time.

This setting exists to provide compatibility with the reference implementation which compiles expressions at evaluation time.

`trim_attribute_space`

If set, additional attribute whitespace will be stripped.

`restricted_namespace`

True by default. If set False, ignored all namespace except chameleon default namespaces. It will be useful working with attributes based javascript template renderer like VueJS.

Example:

 <button v-on:click="greet">Greet</button> <button @click="greet">Greet</button>

`tokenizer`

None by default. If provided, this tokenizer is used instead of the default (which is selected based on the template mode parameter.)

Output is unicode on Python 2 and string on Python 3.

Note: The remaining classes take the same general configuration arguments.

render(*encoding=None*, ***_kw*)

Render template to string.

If providd, the `encoding` argument overrides the template default value.

Additional keyword arguments are passed as template variables.

In addition, some also have a special meaning:

`translate`

This keyword argument will override the default template translate function.

`target_language`

This will be used as the default argument to the translate function if no *i18n:target* value is provided.

If not provided, the *translate* function will need to negotiate a language based on the provided context.

class chameleon.**PageTemplateFile**(*filename*, ***config*)

File-based constructor.

Takes a string input as the only positional argument:

```
template = PageTemplateFile(absolute_path)
```

Note that the file-based template class comes with the expression type `load` which loads templates relative to the provided filename.

Below are listed the configuration arguments specific to file-based templates; see the string-based template class for general options and documentation:

Configuration (keyword arguments):

`loader_class`

The provided class will be used to create the template loader object. The default implementation supports relative and absolute path specs.

The class must accept keyword arguments `search_path` (sequence of paths to search for relative a path spec) and `default_extension` (if provided, this should be added to any path spec).

`prepend_relative_search_path`

Inserts the path relative to the provided template file path into the template search path.

The default setting is `True`.

`search_path`

If provided, this is used as the search path for the `load:` expression. It must be a string or an iterable yielding a sequence of strings.

class chameleon.**PageTextTemplate**(*body*, ***config*)

Text-based template class.

Takes a non-XML input:

```
template = PageTextTemplate("Hello, ${name}.")
```

This is similar to the standard library class `string.Template`, but uses the expression engine to substitute variables.

class chameleon.**PageTextTemplateFile**(*filename*, *search_path=None*, *loader_class=<class 'chameleon.loader.TemplateLoader'>*, ***config*)

File-based constructor.

Loader

Some systems have framework support for loading templates from files. The following loader class is directly compatible with the Pylons framework and may be adapted to other frameworks:

class chameleon.**PageTemplateLoader**(*search_path=None*, *default_extension=None*, ***config*)

Load templates from `search_path` (must be a string or a list of strings):

```
templates = PageTemplateLoader(path)
example = templates['example.pt']
```

If `default_extension` is provided, this will be added to inputs that do not already have an extension:

```
templates = PageTemplateLoader(path, ".pt")
example = templates['example']
```

Any additional keyword arguments will be passed to the template constructor:

```
templates = PageTemplateLoader(path, debug=True, encoding="utf-8")
```

PageTemplateLoader.**load**(*filename*, *format=None*)

Load and return a template file.

The format parameter determines will parse the file. Valid options are *xml* and *text*.

Exceptions

Chameleon may raise exceptions during both the cooking and the rendering phase, but those raised during the cooking phase (parse and compile) all inherit from a single base class:

class chameleon.**TemplateError**(*msg*, *token*)

This exception is the base class of all exceptions raised by the template engine in the case where a template has an error.

It may be raised during rendering since templates are processed lazily (unless eager loading is enabled).

An error that occurs during the rendering of a template is wrapped in an exception class to disambiguate the two cases:

class chameleon.**RenderError**(**args*)

Indicates an exception that resulted from the evaluation of an expression in a template.

A complete traceback is attached to the exception beginning with the expression that resulted in the error. The traceback includes a string representation of the template variable scope for further reference.

Expressions

For advanced integration, the compiler module provides support for dynamic expression evaluation:

class chameleon.compiler.**ExpressionEvaluator**(*engine*, *builtins*)

Evaluates dynamic expression.

This is not particularly efficient, but supported for legacy applications.

```
>>> from chameleon import tales
>>> parser = tales.ExpressionParser({'python': tales.PythonExpr}, 'python')
>>> engine = functools.partial(ExpressionEngine, parser)
```

```
>>> evaluate = ExpressionEvaluator(engine, {
...     'foo': 'bar',
...     })
```

The evaluation function is passed the local and remote context, the expression type and finally the expression.

```
>>> evaluate({'boo': 'baz'}, {}, 'python', 'foo + boo')
'barbaz'
```

The cache is now primed:

```
>>> evaluate({'boo': 'baz'}, {}, 'python', 'foo + boo')
'barbaz'
```

Note that the call method supports currying of the expression argument:

```
>>> python = evaluate({'boo': 'baz'}, {}, 'python')
>>> python('foo + boo')
'barbaz'
```

5.2 Language Reference

The language reference is structured such that it can be read as a general introduction to the *page templates* language.

It's split into parts that correspond to each of the main language features.

5.2.1 Syntax

You can safely *skip this section* if you're familiar with how template languages work or just want to learn by example.

An *attribute language* is a programming language designed to render documents written in XML or HTML markup. The input must be a well-formed document. The output from the template is usually XML-like but isn't required to be well-formed.

The statements of the language are document tags with special attributes, and look like this:

```
<p namespace-prefix:command="argument"> ... </p>
```

In the above example, the attribute `namespace-prefix:command="argument"` is the statement, and the entire paragraph tag is the statement's element. The statement's element is the portion of the document on which this statement operates.

The namespace prefixes are typically declared once, at the top of a template (note that prefix declarations for the template language namespaces are omitted from the template output):

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n">
  ...
</html>
```

Thankfully, sane namespace prefix defaults are in place to let us skip most of the boilerplate:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
    <p tal:content="text"> ... </p>
  </body>
</html>
```

Note how `tal` is used without an explicit namespace declaration. Chameleon sets up defaults for `metal` and `i18n` as well.

Note: Default prefixes are a special feature of Chameleon.

If the `enable_data_attributes` option is set then you can use `data-prefix-command="argument"` in addition to the namespace prefix attributes.

5.2.2 Basics (TAL)

The *template attribute language* is used to create dynamic XML-like content. It allows elements of a document to be replaced, repeated, or omitted.

Statements

These are the available statements:

Statement	Description
`tal:define`	Define variables.
`tal:switch`	Defines a switch condition
`tal:condition`	Include element only if expression is true.
`tal:repeat`	Repeat an element.
`tal:case`	Includes element only if expression is equal to parent switch.
`tal:content`	Substitute the content of an element.
`tal:replace`	Replace the element with dynamic content.
`tal:omit-tag`	Omit the element tags, leaving only the inner content.
`tal:attributes`	Dynamically change or insert element attributes.
`tal:on-error`	Substitute the content of an element if processing fails.

When there is only one TAL statement per element, the order in which they are executed is simple. Starting with the root element, each element's statements are executed, then each of its child elements is visited, in order, to do the same:

```
<html>
  <meta>
    <title tal:content="context.title" />
  </meta>
  <body>
    <div tal:condition="items">
      <p>These are your items:</p>
      <ul>
        <li tal:repeat="item items" tal:content="item" />
      </ul>
    </div>
  </body>
</html>
```

Any combination of statements may appear on the same element, except that the `tal:content` and `tal:replace` statements may not be used on the same element.

Note: The `tal:case` and `tal:switch` statements are available in Chameleon only.

TAL does not use the order in which statements are written in the tag to determine the order in which they are executed. When an element has multiple statements, they are executed in the order printed in the table above.

There is a reasoning behind this ordering. Because users often want to set up variables for use in other statements contained within this element or subelements, `tal:define` is executed first. Then any switch statement. `tal:condition` follows, then `tal:repeat`, then `tal:case`. We are now rendering an element; first `tal:content` or `tal:replace`. Finally, before `tal:attributes`, we have `tal:omit-tag` (which is implied with `tal:replace`).

Note: *TALES* is used as the expression language for the "stuff in the quotes". The default syntax is simply Python, but other inputs are possible — see the section on *expressions*.

---

tal:attributes

Removes, updates or inserts element attributes.

```
tal:attributes="href request.url"
```

Syntax

`tal:attributes` syntax:

```
argument              ::= attribute_statement [';' attribute_statement]*
attribute_statement   ::= (attribute_name expression | expression)
attribute_name        ::= [namespace-prefix ':'] Name
namespace-prefix      ::= Name
```

Description

The `tal:attributes` statement replaces the value of an attribute (or drops, or creates an attribute) with a dynamic value. The value of each expression is converted to a string, if necessary.

Note: You can qualify an attribute name with a namespace prefix, for example `html:table`, if you are generating an XML document with multiple namespaces.

If an attribute expression evaluates to `None`, the attribute is deleted from the statement element (or simply not inserted).

If an attribute statement is just an expression, it must evaluate to a Python dict (or implement the methods `update()` and `items()` from the dictionary specification).

If the expression evaluates to the symbol `default` (a symbol which is always available when evaluating attributes), its value is defined as the default static attribute value. If there is no such default value, a return value of `default` will drop the attribute.

If you use `tal:attributes` on an element with an active `tal:replace` command, the `tal:attributes` statement is ignored.

If you use `tal:attributes` on an element with a `tal:repeat` statement, the replacement is made on each repetition of the element, and the replacement expression is evaluated fresh for each repetition.

Note: If you want to include a semicolon (";") in an expression, it must be escaped by doubling it (";;"). Similarly, you can escape expression interpolation using the "$" symbol by doubling it ("$$").

Examples

Replacing a link:

```
<a href="/sample/link.html"
   tal:attributes="href context.url()"
   >
   ...
</a>
```

Replacing two attributes:

```
<textarea rows="80" cols="20"
          tal:attributes="rows request.rows();cols request.cols()"
    />
```

A checkbox input:

```
<input type="checkbox" tal:attributes="checked True" />
```

tal:condition

Conditionally includes or omits an element:

```
<div tal:condition="comments">
  ...
</div>
```

Syntax

`tal:condition` syntax:

```
argument ::= expression
```

Description

> The `tal:condition` statement includes the statement element in the template only if the condition is met, and omits it otherwise. If its expression evaluates to a *true* value, then normal processing of the element continues, otherwise the statement element is immediately removed from the template. For these purposes, the value `nothing` is false, and `default` has the same effect as returning a true value.

Note: Like Python itself, ZPT considers None, zero, empty strings, empty sequences, empty dictionaries, and instances which return a nonzero value from `__len__` or which return false from `__nonzero__`; all other values are true, including `default`.

Examples

Test a variable before inserting it:

```
<p tal:condition="request.message" tal:content="request.message" />
```

Testing for odd/even in a repeat-loop:

```
<div tal:repeat="item range(10)">
  <p tal:condition="repeat.item.even">Even</p>
  <p tal:condition="repeat.item.odd">Odd</p>
</div>
```

tal:content

Replaces the content of an element.

Syntax

`tal:content` syntax:

```
argument ::= (['text'] | 'structure') expression
```

Description

Rather than replacing an entire element, you can insert text or structure in place of its children with the `tal:content` statement. The statement argument is exactly like that of `tal:replace`, and is interpreted in the same fashion. If the expression evaluates to `nothing`, the statement element is left childless. If the expression evaluates to `default`, then the element's contents are evaluated.

The default replacement behavior is `text`, which replaces angle-brackets and ampersands with their HTML entity equivalents. The `structure` keyword passes the replacement text through unchanged, allowing HTML/XML markup to be inserted. This can break your page if the text contains unanticipated markup (eg. text submitted via a web form), which is the reason that it is not the default.

Note: The `structure` keyword exists to provide backwards compatibility. In Chameleon, the `structure:` expression type provides the same functionality (also for inline expressions).

Examples

Inserting the user name:

```
<p tal:content="user.getUserName()">Fred Farkas</p>
```

Inserting HTML/XML:

```
<p tal:content="structure context.getStory()">
   Marked <b>up</b> content goes here.
</p>
```

tal:define

Defines local variables.

Syntax

`tal:define` syntax:

```
variable_name  ::= Name | '(' Name [',' Name]* ')'
define_var     ::= variable_name expression
define_scope   ::= (['local'] | 'global') define_var
argument       ::= define_scope [';' define_scope]*
```

Description

The `tal:define` statement defines variables. When you define a local variable in a statement element, you can use that variable in that element and the elements it contains. If you redefine a variable in a contained element, the new definition hides the outer element's definition within the inner element.

Note that valid variable names are any Python identifier string including underscore, although two or more leading underscores are disallowed (used internally by the compiler). Further, names are case-sensitive.

Variable names support basic iterable unpacking when surrounded by parenthesis. This also applies to the variable established by `tal:repeat`.

Note: This is a Chameleon-specific *language extension*.

Python builtins are always "in scope", but most of them may be redefined (such as `help`). Exceptions are:: `float`, `int`, `len`, `long`, `str`, `None`, `True` and `False`.

In addition, the following names are reserved: `econtext`, `rcontext`, `translate`, `decode` and `convert`.

If the expression associated with a variable evaluates to `nothing`, then that variable has the value `nothing`, and may be used as such in further expressions. Likewise, if the expression evaluates to `default`, then the variable has the value `default`, and may be used as such in further expressions.

You can define two different kinds of variables: *local* and *global*. When you define a local variable in a statement element, you can only use that variable in that element and the elements it contains. If you redefine a local variable in a contained element, the new definition hides the outer element's definition within the inner element. When you define a global variables, you can use it in any element processed after the defining element. If you redefine a global variable, you replace its definition for the rest of the template.

Tip: Global variables may be changed by the execution of a macro if that macro also declares the variable to be global.

To set the definition scope of a variable, use the keywords `local` or `global` in front of the assignment. The default setting is `local`; thus, in practice, only the `global` keyword is used.

Note: If you want to include a semicolon (";") in an expression, it must be escaped by doubling it (";;").

Examples

Defining a variable:

```
tal:define="company_name 'Zope Corp, Inc.'"
```

Defining two variables, where the second depends on the first:

```
tal:define="mytitle context.title; tlen len(mytitle)"
```

Defining a local and global variable:

```
tal:define="global mytitle context.title; tlen len(mytitle)"
```

Unpacking a sequence:

```
tal:define="(key,value) ('a', 42)"
```

`tal:switch` and `tal:case`

Defines a switch clause.

```
<ul tal:switch="len(items) % 2">
  <li tal:case="True">odd</li>
  <li tal:case="False">even</li>
</ul>
```

Syntax

`tal:case` and `tal:switch` syntax:

```
argument ::= expression
```

Description

The *switch* and *case* construct is a short-hand syntax for matching a set of expressions against a single parent.

The `tal:switch` statement is used to set a new parent expression and the contained `tal:case` statements are then matched in sequence such that only the first match succeeds.

Note that the symbol `default` affirms the case precisely when no previous case has been successful. It should therefore be placed last.

Note: These statements are only available in Chameleon 2.x and not part of the ZPT specification.

Examples

```
<ul tal:switch="item.type">
  <li tal:case="'document'">
    Document
  </li>
  <li tal:case="'folder'">
    Folder
  </li>
  <li tal:case="default">
    Other
  </li>
</ul>
```

tal:omit-tag

Removes an element, leaving its contents.

Syntax

`tal:omit-tag` syntax:

```
argument ::= [ expression ]
```

Description

The `tal:omit-tag` statement leaves the contents of an element in place while omitting the surrounding start and end tags.

If the expression evaluates to a *false* value, then normal processing of the element continues and the tags are not omitted. If the expression evaluates to a *true* value, or no expression is provided, the statement element is replaced with its contents.

Note: Like Python itself, ZPT considers None, zero, empty strings, empty sequences, empty dictionaries, and instances which return a nonzero value from __len__ or which return false from __nonzero__; all other values are true, including `default`.

Examples

Unconditionally omitting a tag:

```
<div tal:omit-tag="" comment="This tag will be removed">
  <i>...but this text will remain.</i>
</div>
```

Conditionally omitting a tag:

```
<b tal:omit-tag="not:bold">I may be bold.</b>
```

The above example will omit the `b` tag if the variable `bold` is false.

Creating ten paragraph tags, with no enclosing tag:

```
<span tal:repeat="n range(10)"
      tal:omit-tag="">
  <p tal:content="n">1</p>
</span>
```

tal:repeat

Repeats an element.

Syntax

`tal:repeat` syntax:

```
argument      ::= variable_name expression
variable_name ::= Name
```

Description

The `tal:repeat` statement replicates a sub-tree of your document once for each item in a sequence. The expression should evaluate to a sequence. If the sequence is empty, then the statement element is deleted, otherwise it is repeated for each value in the sequence. If the expression is `default`, then the element is left unchanged, and no new variables are defined.

The `variable_name` is used to define a local variable and a repeat variable. For each repetition, the local variable is set to the current sequence element, and the repeat variable is set to an iteration object.

Repeat variables

You use repeat variables to access information about the current repetition (such as the repeat index). The repeat variable has the same name as the local variable, but is only accessible through the built-in variable named `repeat`.

The following information is available from the repeat variable:

Attribute	Description
`index`	Repetition number, starting from zero.
`number`	Repetition number, starting from one.
`even`	True for even-indexed repetitions (0, 2, 4, ...).
`odd`	True for odd-indexed repetitions (1, 3, 5, ...).
`parity`	For odd-indexed repetitions, this is 'odd', else 'even'.
`start`	True for the starting repetition (index 0).
`end`	True for the ending, or final, repetition.
`length`	Length of the sequence, which will be the total number of repetitions.
`letter`	Repetition number as a lower-case letter: "a" - "z", "aa" - "az", "ba" - "bz", ..., "za" - "zz", "aaa" - "aaz", and so forth.
`Letter`	Upper-case version of *letter*.
`roman`	Repetition number as a lower-case roman numeral: "i", "ii", "iii", "iv", "v", etc.
`Roman`	Upper-case version of *roman*.

You can access the contents of the repeat variable using either dictionary- or attribute-style access, e.g. `repeat['item'].start` or `repeat.item.start`.

Note: For legacy compatibility, the attributes `odd`, `even`, `number`, `letter`, `Letter`, `roman`, and `Roman` are callable (returning `self`).

Note: Earlier versions of this document, and the Zope 2 Page Templates Reference, referred to `first` and `last` attributes for use with sorted sequences. These are not implemented in Chameleon or the Zope reference implementation zope.tales. Instead, you can use `itertools.groupby()`, as in the example below.

Examples

Iterating over a sequence of strings:

```
<p tal:repeat="txt ('one', 'two', 'three')">
   <span tal:replace="txt" />
</p>
```

Inserting a sequence of table rows, and using the repeat variable to number the rows:

```
<table>
  <tr tal:repeat="item here.cart">
      <td tal:content="repeat.item.number">1</td>
      <td tal:content="item.description">Widget</td>
      <td tal:content="item.price">$1.50</td>
  </tr>
</table>
```

Nested repeats:

```
<table border="1">
  <tr tal:repeat="row range(10)">
    <td tal:repeat="column range(10)">
      <span tal:define="x repeat.row.number;
                        y repeat.column.number;
                        z x * y"
            tal:replace="string:$x * $y = $z">1 * 1 = 1</span>
    </td>
  </tr>
</table>
```

Grouping objects by type, drawing a rule between elements of different types:

```
<div tal:repeat="(type,objects) list(map(lambda g: (g[0], list(g[1])), itertools.
↪groupby(objects, key=lambda o: o.meta_type)))"
     tal:define="itertools import:itertools">
  <h2 tal:content="type">Meta Type</h2>
  <p tal:repeat="object objects"
     tal:content="object.id">Object ID</p>
  <hr />
</div>
```

Caution: It is important to fully realize the iterator produced by `itertools.groupby()`, as well as the iterator produced for each group, in the expression passed to `tal:repeat`. This is because the implementation of certain repeat variables, such as `length` and `end` requires Chameleon to look ahead in the iterator, consuming it faster than is visible. The iterator returned by `itertools.groupby()` is shared among all of its subgroups, so without the full reification of all the iterators, incorrect results will be produced.

tal:replace

Replaces an element.

Syntax

`tal:replace` syntax:

```
argument ::= ['structure'] expression
```

Description

The `tal:replace` statement replaces an element with dynamic content. It replaces the statement element with either text or a structure (unescaped markup). The body of the statement is an expression with an optional type prefix. The value of the expression is converted into an escaped string unless you provide the 'structure' prefix. Escaping consists of converting `&` to `&amp;`, `<` to `&lt;`, and `>` to `&gt;`.

Note: If the inserted object provides an `__html__` method, that method is called with the result inserted as structure. This feature is not implemented by ZPT.

If the expression evaluates to `None`, the element is simply removed. If the value is `default`, then the element is left unchanged.

Examples

Inserting a title:

```
<span tal:replace="context.title">Title</span>
```

Inserting HTML/XML:

```
<div tal:replace="structure table" />
```

5.2.3 Expressions (TALES)

The *Template Attribute Language Expression Syntax* (TALES) standard describes expressions that supply *Basics (TAL)* and *Macros (METAL)* with data. TALES is *one* possible expression syntax for these languages, but they are not bound to this definition. Similarly, TALES could be used in a context having nothing to do with TAL or METAL.

TALES expressions are described below with any delimiter or quote markup from higher language layers removed. Here is the basic definition of TALES syntax:

```
Expression  ::= [type_prefix ':'] String
type_prefix ::= Name
```

Here are some simple examples:

```
1 + 2
None
string:Hello, ${view.user_name}
```

The optional *type prefix* determines the semantics and syntax of the *expression string* that follows it. A given implementation of TALES can define any number of expression types, with whatever syntax you like. It also determines which expression type is indicated by omitting the prefix.

Types

These are the available TALES expression types:

Prefix	Description
`exists`	Evaluate the result inside an exception handler; if one of the exceptions `AttributeError`, `LookupError`, `TypeError`, `NameError`, or `KeyError` is raised during evaluation, the result is `False`, otherwise `True`. Note that the original result is discarded in any case.
`import`	Import a global symbol using dotted notation.
`load`	Load a template relative to the current template or absolute.
`not`	Negate the expression result
`python`	Evaluate a Python expression
`string`	Format a string
`structure`	Wraps the expression result as *structure*.

Note: The default expression type is `python`.

Warning: The Zope reference engine defaults to a `path` expression type, which is closely tied to the Zope framework. This expression is not implemented in Chameleon (but it's available in a Zope framework compatibility package, z3c.pt).

There's a mechanism to allow fallback to alternative expressions, if one should fail (raise an exception). The pipe character ('|') is used to separate two expressions:

```
<div tal:define="page request.GET['page'] | 0">
```

This mechanism applies only to the `python` expression type, and by derivation `string`.

python

Evaluates a Python expression.

Syntax

Python expression syntax:

```
Any valid Python language expression
```

Description

Python expressions are executed natively within the translated template source code. There is no built-in security apparatus.

string

Syntax

String expression syntax:

```
string_expression ::= ( plain_string | [ varsub ] )*
varsub            ::= ( '$' Variable ) | ( '${ Expression }' )
plain_string      ::= ( '$$' | non_dollar )*
non_dollar        ::= any character except '$'
```

Description

String expressions interpret the expression string as text. If no expression string is supplied the resulting string is *empty*. The string can contain variable substitutions of the form `$name` or `${expression}`, where `name` is a variable name, and `expression` is a TALES-expression. The escaped string value of the expression is inserted into the string.

Note: To prevent a `$` from being interpreted this way, it must be escaped as `$$`.

Examples

Basic string formatting:

```
<span tal:replace="string:$this and $that">
  Spam and Eggs
</span>

<p tal:content="string:${request.form['total']}">
  total: 12
</p>
```

Including a dollar sign:

```
<p tal:content="string:$$$cost">
  cost: $42.00
</p>
```

import

Imports a module global.

structure

Wraps the expression result as *structure*: The replacement text is inserted into the document without escaping, allowing HTML/XML markup to be inserted. This can break your page if the text contains unanticipated markup (eg. text submitted via a web form), which is the reason that it is not the default.

load

Loads a template instance.

Syntax

Load expression syntax:

```
Relative or absolute file path
```

Description

The template will be loaded using the same template class as the calling template.

Examples

Loading a template and using it as a macro:

```
<div tal:define="master load: ../master.pt" metal:use-macro="master" />
```

Built-in names

These are the names always available in the TALES expression namespace:

- `default` - special value used to specify that existing text or attributes should not be replaced. See the documentation for individual TAL statements for details on how they interpret *default*.
- `repeat` - the *repeat* variables; see *tal:repeat* for more information.
- `template` - reference to the template which was first called; this symbol is carried over when using macros.
- `macros` - reference to the macros dictionary that corresponds to the current template.

5.2.4 Macros (METAL)

The *Macro Expansion Template Attribute Language* (METAL) standard is a facility for HTML/XML macro preprocessing. It can be used in conjunction with or independently of TAL and TALES.

Macros provide a way to define a chunk of presentation in one template, and share it in others, so that changes to the macro are immediately reflected in all of the places that share it. Additionally, macros are always fully expanded, even in a template's source text, so that the template appears very similar to its final rendering.

A single Page Template can accomodate multiple macros.

Namespace

The METAL namespace URI and recommended alias are currently defined as:

```
xmlns:metal="http://xml.zope.org/namespaces/metal"
```

Just like the TAL namespace URI, this URI is not attached to a web page; it's just a unique identifier. This identifier must be used in all templates which use METAL.

Note that elements that appear in a template with the METAL namespace are omitted from the output where they appear. This is useful when defining a macro:

```
<metal:block define-macro="hello">
  ...
</metal:block>
```

In the example above the element is named *block* but any name can be used to the same effect as long as it is qualified with the METAL namespace.

Statements

METAL defines a number of statements:

- `metal:define-macro` Define a macro.
- `metal:use-macro` Use a macro.
- `metal:extend-macro` Extend a macro.
- `metal:define-slot` Define a macro customization point.
- `metal:fill-slot` Customize a macro.

Although METAL does not define the syntax of expression non-terminals, leaving that up to the implementation, a canonical expression syntax for use in METAL arguments is described in TALES Specification.

define-macro

Defines a macro.

Syntax

`metal:define-macro` syntax:

```
argument ::= Name
```

Description

The `metal:define-macro` statement defines a macro. The macro is named by the statement expression, and is defined as the element and its sub-tree.

Examples

Simple macro definition:

```
<p metal:define-macro="copyright">
  Copyright 2011, <em>Foobar</em> Inc.
</p>
```

define-slot

Defines a macro customization point.

Syntax

`metal:define-slot` syntax:

```
argument ::= Name
```

Description

The `metal:define-slot` statement defines a macro customization point or *slot*. When a macro is used, its slots can be replaced, in order to customize the macro. Slot definitions provide default content for the slot. You will get the default slot contents if you decide not to customize the macro when using it.

The `metal:define-slot` statement must be used inside a `metal:define-macro` statement.

Slot names must be unique within a macro.

Examples

Simple macro with slot:

```
<p metal:define-macro="hello">
  Hello <b metal:define-slot="name">World</b>
</p>
```

This example defines a macro with one slot named `name`. When you use this macro you can customize the `b` element by filling the `name` slot.

fill-slot

Customize a macro.

Syntax

`metal:fill-slot` syntax:

```
argument ::= Name
```

Description

The `metal:fill-slot` statement customizes a macro by replacing a *slot* in the macro with the statement element (and its content).

The `metal:fill-slot` statement must be used inside a `metal:use-macro` statement.

Slot names must be unique within a macro.

If the named slot does not exist within the macro, the slot contents will be silently dropped.

Examples

Given this macro:

```
<p metal:define-macro="hello">
  Hello <b metal:define-slot="name">World</b>
</p>
```

You can fill the `name` slot like so:

```
<p metal:use-macro="container['master.html'].macros.hello">
  Hello <b metal:fill-slot="name">Kevin Bacon</b>
</p>
```

use-macro

Use a macro.

Syntax

`metal:use-macro` syntax:

```
argument ::= expression
```

Description

The `metal:use-macro` statement replaces the statement element with a macro. The statement expression describes a macro definition. The `macroname` variable will be bound to the defined name of the macro being used.

Note: In Chameleon the expression may point to a template instance; in this case it will be rendered in its entirety.

extend-macro

Extends a macro.

Syntax

`metal:extend-macro` syntax:

```
argument ::= expression
```

Description

To extend an existing macro, choose a name for the macro and add a define-macro attribute to a document element with the name as the argument. Add an extend-macro attribute to the document element with an expression referencing the base macro as the argument. The extend-macro must be used in conjunction with define-macro, and must not be used with use-macro. The element's subtree is the macro body.

Examples

```
<div metal:define-macro="page-header"
     metal:extend-macro="standard_macros['page-header']">
  <div metal:fill-slot="breadcrumbs">
    You are here:
    <div metal:define-slot="breadcrumbs"/>
  </div>
</div>
```

5.2.5 Translation (I18N)

Translation of template contents and attributes is supported via the `i18n` namespace and message objects.

Messages

The translation machinery defines a message as *any object* which is not a string or a number and which does not provide an `__html__` method.

When any such object is inserted into the template, the translate function is invoked first to see if it needs translation. The result is always coerced to a native string before it's inserted into the template.

Translation function

The simplest way to hook into the translation machinery is to provide a translation function to the template constructor or at render-time. In either case it should be passed as the keyword argument `translate`.

The function has the following signature:

```
def translate(msgid, domain=None, mapping=None, context=None, target_language=None,␣
↪default=None):
    ...
```

The result should be a string or `None`. If another type of object is returned, it's automatically coerced into a string.

If zope.i18n is available, the translation machinery defaults to using its translation function. Note that this function requires messages to conform to the message class from zope.i18nmessageid; specifically, messages must have attributes `domain`, `mapping` and `default`. Example use:

```
from zope.i18nmessageid import MessageFactory
_ = MessageFactory("food")

apple = _(u"Apple")
```

There's currently no further support for other translation frameworks.

Using Zope's translation framework

The translation function from `zope.i18n` relies on *translation domains* to provide translations.

These are components that are registered for some translation domain identifier and which implement a `translate` method that translates messages for that domain.

Note: To register translation domain components, the Zope Component Architecture must be used (see zope.component).

The easiest way to configure translation domains is to use the the `registerTranslations` ZCML-directive; this requires the use of the zope.configuration package. This will set up translation domains and gettext catalogs automatically:

```
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:i18n="http://xml.zope.org/namespaces/i18n">

   <i18n:registerTranslations directory="locales" />

</configure>
```

The `./locales` directory must follow a particular directory structure:

```
./locales/en/LC_MESSAGES
./locales/de/LC_MESSAGES
...
```

In each of the `LC_MESSAGES` directories, one GNU gettext file in the `.po` format must be present per translation domain:

```
# ./locales/de/LC_MESSAGES/food.po

msgid ""
msgstr ""
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

msgid "Apple"
msgstr "Apfel"
```

It may be necessary to compile the message catalog using the `msgfmt` utility. This will produce a `.mo` file.

Translation domains without gettext

The following example demonstrates how to manually set up and configure a translation domain for which messages are provided directly:

```
from zope import component
from zope.i18n.simpletranslationdomain import SimpleTranslationDomain

food = SimpleTranslationDomain("food", {
    ('de', u'Apple'): u'Apfel',
    })

component.provideUtility(food, food.domain)
```

An example of a custom translation domain class:

```
from zope import interface

class TranslationDomain(object):
     interface.implements(ITranslationDomain)

     def translate(self, msgid, mapping=None, context=None,
                   target_language=None, default=None):

         ...

component.provideUtility(TranslationDomain(), name="custom")
```

This approach can be used to integrate other translation catalog implementations.

Namespace

The `i18n` namespace URI and recommended prefix are currently defined as:

```
xmlns:i18n="http://xml.zope.org/namespaces/i18n"
```

This is not a URL, but merely a unique identifier. Do not expect a browser to resolve it successfully.

Statements

The allowable `i18n` statements are:

- `i18n:translate`
- `i18n:domain`
- `i18n:context`
- `i18n:source`
- `i18n:target`
- `i18n:name`
- `i18n:attributes`
- `i18n:data`
- `i18n:comment`

- `i18n:ignore`
- `i18n:ignore-attributes`

i18n:translate

This attribute is used to mark units of text for translation. If this attribute is specified with an empty string as the value, the message ID is computed from the content of the element bearing this attribute. Otherwise, the value of the element gives the message ID.

i18n:domain

The `i18n:domain` attribute is used to specify the domain to be used to get the translation. If not specified, the translation services will use a default domain. The value of the attribute is used directly; it is not a TALES expression.

i18n:context

The `i18n:context` attribute is used to specify the context to be used to get the translation. If not specified, the translation services will use a default context. The context is generally use to distinguish identical texts in different context (because in a translation this may not be the case.) The value of the attribute is used literally; it is not an expression.

i18n:source

The `i18n:source` attribute specifies the language of the text to be translated. The default is `nothing`, which means we don't provide this information to the translation services.

i18n:target

The `i18n:target` attribute specifies the language of the translation we want to get. If the value is `default`, the language negotiation services will be used to choose the destination language. If the value is `nothing`, no translation will be performed; this can be used to suppress translation within a larger translated unit. Any other value must be a language code.

The attribute value is a TALES expression; the result of evaluating the expression is the language code or one of the reserved values.

Note: `i18n:target` is primarily used for hints to text extraction tools and translation teams. If you had some text that should only be translated to e.g. German, then it probably shouldn't be wrapped in an `i18n:translate` span.

---

i18n:name

Name the content of the current element for use in interpolation within translated content. This allows a replaceable component in content to be re-ordered by translation. For example:

```
<span i18n:translate=''>
  <span tal:replace='context.name' i18n:name='name' /> was born in
  <span tal:replace='context.country_of_birth' i18n:name='country' />.
</span>
```

would cause this text to be passed to the translation service:

```
"${name} was born in ${country}."
```

i18n:attributes

This attribute will allow us to translate attributes of HTML tags, such as the `alt` attribute in the `img` tag. The `i18n:attributes` attribute specifies a list of attributes to be translated with optional message IDs for each; if multiple attribute names are given, they must be separated by semicolons. Message IDs used in this context must not include whitespace.

Note that the value of the particular attributes come either from the HTML attribute value itself or from the data inserted by `tal:attributes`.

If an attibute is to be both computed using `tal:attributes` and translated, the translation service is passed the result of the TALES expression for that attribute.

An example:

```
<img src="http://foo.com/logo" alt="Visit us"
     tal:attributes="alt context.greeting"
     i18n:attributes="alt"
     >
```

In this example, we let `tal:attributes` set the value of the `alt` attribute to the text "Stop by for a visit!". This text will be passed to the translation service, which uses the result of language negotiation to translate "Stop by for a visit!" into the requested language. The example text in the template, "Visit us", will simply be discarded.

Another example, with explicit message IDs:

```
<img src="../icons/uparrow.png" alt="Up"
     i18n:attributes="src up-arrow-icon; alt up-arrow-alttext"
     >
```

Here, the message ID `up-arrow-icon` will be used to generate the link to an icon image file, and the message ID 'up-arrow-alttext' will be used for the "alt" text.

i18n:data

Since TAL always returns strings, we need a way in ZPT to translate objects, one of the most obvious cases being `datetime` objects. The `data` attribute will allow us to specify such an object, and `i18n:translate` will provide us with a legal format string for that object. If `data` is used, `i18n:translate` must be used to give an explicit message ID, rather than relying on a message ID computed from the content.

i18n:comment

The `i18n:comment` attribute can be used to add extra comments for translators. It is not used by Chameleon for processing, but will be picked up by tools like lingua.

An example:

> **<h3 i18n:comment="Header for the news section"** i18n:translate="">News</h3>

i18n:ignore

The `i18n:ignore` attribute can be used to inform translation extraction tools like i18ndude to not give a warning/error on the given tag if there is no `i18n:translate` attribute.

An example:

> <h1 i18n:ignore="">News</h3>

i18n:ignore-attributes

The `i18n:ignore-attributes`, just like `i18n:ignore` is expected to be used by translation extraction tools like i18ndude. If `i18n:ignore` makes text within a tag to be ignored, `i18n:ignore-attributes` marks the given attributes as ignored.

An example:

> **Python website

Relation with TAL processing

The attributes defined in the `i18n` namespace modify the behavior of the TAL interpreter for the `tal:attributes`, `tal:content`, `tal:repeat`, and `tal:replace` attributes, but otherwise do not affect TAL processing.

Since these attributes only affect TAL processing by causing translations to occur at specific times, using these with a TAL processor which does not support the `i18n` namespace degrades well; the structural expectations for a template which uses the `i18n` support is no different from those for a page which does not. The only difference is that translations will not be performed in a legacy processor.

Relation with METAL processing

When using translation with METAL macros, the internationalization context is considered part of the specific documents that page components are retrieved from rather than part of the combined page. This makes the internationalization context lexical rather than dynamic, making it easier for a site builder to understand the behavior of each element with respect to internationalization.

Let's look at an example to see what this means:

```
<html i18n:translate='' i18n:domain='EventsCalendar'
      metal:use-macro="container['master.html'].macros.thismonth">

  <div metal:fill-slot='additional-notes'>
    <ol tal:condition="context.notes">
      <li tal:repeat="note context.notes">
         <tal:block tal:omit-tag=""
                    tal:condition="note.heading">
           <strong tal:content="note.heading">
             Note heading goes here
           </strong>
           <br />
```

(continues on next page)

(continued from previous page)

```
        </tal:block>
        <span tal:replace="note/description">
          Some longer explanation for the note goes here.
        </span>
      </li>
    </ol>
  </div>

</html>
```

And the macro source:

```
<html i18n:domain='CalendarService'>
  <div tal:replace='python:DateTime().Month()'
       i18n:translate=''>January</div>

  <!-- really hairy TAL code here ;-) -->

  <div define-slot="additional-notes">
    Place for the application to add additional notes if desired.
  </div>

</html>
```

Note that the macro is using a different domain than the application (which it should be). With lexical scoping, no special markup needs to be applied to cause the slot-filler in the application to be part of the same domain as the rest of the application's page components. If dynamic scoping were used, the internationalization context would need to be re-established in the slot-filler.

Extracting translatable message

Translators use PO files when translating messages. To create and update PO files you need to do two things: *extract* all messages from python and templates files and store them in a `.pot` file, and for each language *update* its `.po` file. Chameleon facilitates this by providing extractors for Babel. To use this you need modify `setup.py`. For example:

```
from setuptools import setup

setup(name="mypackage",
      install_requires = [
            "Babel",
            ],
      message_extractors = { "src": [
            ("**.py",   "chameleon_python", None ),
            ("**.pt",   "chameleon_xml", None ),
            ]},
      )
```

This tells Babel to scan the `src` directory while using the `chameleon_python` extractor for all `.py` files and the `chameleon_xml` extractor for all `.pt` files.

You can now use Babel to manage your PO files:

```
python setup.py extract_messages --output-file=i18n/mydomain.pot
python setup.py update_catalog \
          -l nl \
```

(continues on next page)

(continued from previous page)

```
          -i i18n/mydomain.pot \
          -o i18n/nl/LC_MESSAGES/mydomain.po
python setup.py compile_catalog \
          --directory i18n --locale nl
```

You can also configure default options in a `setup.cfg` file. For example:

```
[compile_catalog]
domain = mydomain
directory = i18n

[extract_messages]
copyright_holder = Acme Inc.
output_file = i18n/mydomain.pot
charset = UTF-8

[init_catalog]
domain = mydomain
input_file = i18n/mydomain.pot
output_dir = i18n

[update_catalog]
domain = mydomain
input_file = i18n/mydomain.pot
output_dir = i18n
previous = true
```

You can now use the Babel commands directly:

```
python setup.py extract_messages
python setup.py update_catalog
python setup.py compile_catalog
```

5.2.6 ${...} operator

The `${...}` notation is short-hand for text insertion. The Python-expression inside the braces is evaluated and the result included in the output (all inserted text is escaped by default):

```
<div id="section-${index + 1}">
  ${content}
</div>
```

To escape this behavior, prefix the notation with a backslash character: `\${...}`.

Note that if an object implements the `__html__` method, the result of this method will be inserted as-is (without XML escaping).

5.2.7 Code blocks

The `<?python ... ?>` notation allows you to embed Python code in templates:

```
<div>
  <?python numbers = map(str, range(1, 10)) ?>
```

(continues on next page)

(continued from previous page)

```
  Please input a number from the range ${", ".join(numbers)}.
</div>
```

The scope of name assignments is up to the nearest macro definition, or the template, if macros are not used.

Note that code blocks can span multiple line and start on the next line of where the processing instruction begins:

```
<?python
  foo = [1, 2, 3]
?>
```

You can use this to debug templates:

```
<div>
  <?python import pdb; pdb.set_trace() ?>
</div>
```

5.2.8 Markup comments

You can apply the "!" and "?" modifiers to change how comments are processed:

Drop

```
<!--! This comment will be dropped from output -->
```

Verbatim

```
<!--? This comment will be included verbatim -->
```

That is, evaluation of `${...}` expressions is disabled if the comment opens with the "?" character.

5.2.9 Language extensions

Chameleon extends the *page template* language with a new expression types and language features. Some take inspiration from Genshi.

New expression types

The *structure* expression wraps an expression result as *structure*:

```
<div>${structure: body.text}</div>
```

The *import* expression imports module globals:

```
<div tal:define="compile import: re.compile">
  ...
</div>
```

The *load* expression loads templates relative to the current template:

```
<div tal:define="compile load: main.pt">
  ...
</div>
```

Tuple unpacking

The `tal:define` and `tal:repeat` statements support tuple unpacking:

```
tal:define="(a, b, c) [1, 2, 3]"
```

Extended iterable unpacking using the asterisk character is not currently supported (even for versions of Python that support it natively).

Dictionary lookup as fallback after attribute error

If attribute lookup (using the `obj.<name>` syntax) raises an `AttributeError` exception, a secondary lookup is attempted using dictionary lookup — `obj['<name>']`.

Behind the scenes, this is done by rewriting all attribute-lookups to a custom lookup call:

```
def lookup_attr(obj, key):
    try:
        return getattr(obj, key)
    except AttributeError as exc:
        try:
            get = obj.__getitem__
        except AttributeError:
            raise exc
        try:
            return get(key)
        except KeyError:
            raise exc
```

Inline string substitution

In element attributes and in the text or tail of an element, string expression interpolation is available using the `${...}` syntax:

```
<span class="content-${item_type}">
   ${title or item_id}
</span>
```

Code blocks

Using `<?python ... ?>` notation, you can embed Python statements in your templates:

```
<div>
  <?python numbers = map(str, range(1, 10)) ?>
  Please input a number from the range ${", ".join(numbers)}.
</div>
```

Literal content

While the `tal:content` and `tal:repeat` attributes both support the `structure` keyword which inserts the content as a literal (without XML-escape), an object may also provide an `__html__` method to the same effect.

The result of the method will be inserted as *structure*.

This is particularly useful for content which is substituted using the expression operator: `"${...}"` since the `structure` keyword is not allowed here.

Switch statement

Two new attributes have been added: `tal:switch` and `tal:case`. A case attribute works like a condition and only allows content if the value matches that of the nearest parent switch value.

5.2.10 Incompatibilities and differences

There are a number of incompatibilities and differences between the Chameleon language implementation and the Zope reference implementation (ZPT):

Default expression

The default expression type is Python.

Template arguments

Arguments passed by keyword to the render- or call method are inserted directly into the template execution namespace. This is different from ZPT where these are only available through the `options` dictionary.

Zope:

```
<div tal:content="options/title" />
```

Chameleon:

```
<div tal:content="title" />
```

Special symbols

The `CONTEXTS` symbol is not available.

The z3c.pt package works as a compatibility layer. The template classes in this package provide a implementation which is fully compatible with ZPT.

5.3 Integration

Integration with Chameleon is available for a number of popular web frameworks. The framework will usually provide loading mechanisms and translation (internationalization) configuration.

5.3.1 Pyramid

pyramid_chameleon is a set of bindings that make templates written for the Chameleon templating system work under the Pyramid web framework.

5.3.2 Zope 2 / Plone

Install the five.pt package to replace the reference template engine (globally).

5.3.3 Zope Toolkit (ZTK)

Install the z3c.pt package for applications based on the Zope Toolkit (ZTK). Note that you need to explicit use the template classes from this package.

5.3.4 Grok

Support for the Grok framework is available in the grokcore.chameleon package.

This package will setup Grok's policy for templating integration and associate the Chameleon template components for the `.cpt` template filename extension.

5.3.5 Django

Install the django-chameleon-templates app to enable Chameleon as a template engine.

5.4 Configuration

Most settings can be provided as keyword-arguments to the template constructor classes.

There are certain settings which are required at environment level. Acceptable values are `"0"`, `"1"`, or the literals `"true"` or `"false"` (case-insensitive).

5.4.1 General usage

The following settings are useful in general.

`CHAMELEON_EAGER` Parse and compile templates on instantiation.

`CHAMELEON_CACHE`

When set to a file system path, the template compiler will write its output to files in this directory and use it as a cache.

This not only enables you to see the compiler output, but also speeds up startup.

`CHAMELEON_RELOAD` This setting controls the default value of the `auto_reload` parameter.

5.4.2 Development

The following settings are mostly useful during development or debugging of the library itself.

`CHAMELEON_DEBUG`

Enables a set of debugging settings which make it easier to discover and research issues with the engine itself.

This implicitly enables auto-reload for any template.

CHAPTER 6

Indices and Tables

- genindex
- modindex
- search

CHAPTER 7

Notes

Python Module Index

C

Index