Genshi

Release 0.7dev

May 27, 2015

Contents


GENSHI

CHAPTER 1

Toolkit for generation of output for the web

Genshi is a Python library that provides an integrated set of components for parsing, generating, and processing HTML, XML or other textual content for output generation on the web. The major feature is a template language, which is heavily inspired by Kid.

1.1 Installation

1.1.1 Installing Genshi

Prerequisites

- Python 2.4 or later
- Optional: Setuptools 0.6c3 or later

Setuptools is only required for the template engine plugin, which can be used to integrate Genshi with Python web application frameworks such as Pylons or TurboGears. Genshi also provides a Setuptools-based plugin that integrates its internationalization support with the Babel library, but that support can also be used without Setuptools being available (although in a slightly less convenient fashion).

Installing via `easy_install`

If you have a recent version of Setuptools installed, you can directly install Genshi using the easy_install command-line tool:

```
$ easy_install Genshi
```

This downloads and installs the latest version of the Genshi package.

If you have an older Genshi release installed and would like to upgrade, add the `-U` option to the above command.

Installing from a Binary Installer

Binary packages for Windows and Mac OS X are provided for Genshi. To install from such a package, simply download and open it.

Installing from a Source Tarball

Once you've downloaded and unpacked a Genshi source release, enter the directory where the archive was unpacked, and run:

```
$ python setup.py install
```

Note that you may need administrator/root privileges for this step, as this command will by default attempt to install Genshi to the Python `site-packages` directory on your system.

Genshi comes with an optional extension module written in C that is used to improve performance in some areas. This extension is automatically compiled when you run the `setup.py` script as shown above. In the case that the extension can not be compiled, possibly due to a missing or incompatible C compiler, the compilation is skipped. If you'd prefer Genshi to not use this native extension module, you can explicitly bypass the compilation using the `--without-speedups` option:

```
$ python setup.py --without-speedups install
```

For other build and installation options, please consult the easy_install and/or the Python distutils documentation.

Support

If you encounter any problems with Genshi, please don't hesitate to ask questions on the Genshi mailing list or IRC channel.

1.1.2 Upgrading Genshi

Upgrading from Genshi 0.6.x to the development version

The Genshi development version now supports both Python 2 and Python 3.

The most noticable API change in the Genshi development version is that the default encoding in numerous places is now None (i.e. unicode) instead of UTF-8. This change was made in order to ease the transition to Python 3 where strings are unicode strings by default.

If your application relies on the default UTF-8 encoding a simple way to have it work both with Genshi 0.6.x and the development version is to specify the encoding explicitly in calls to the following classes, methods and functions:

- genshi.HTML
- genshi.Stream.render
- genshi.input.HTMLParser
- genshi.template.MarkupTemplate
- genshi.template.TemplateLoader
- genshi.template.TextTemplate (and genshi.template.NewTextTemplate)
- genshi.template.OldTextTemplate

Whether you explicitly specify UTF-8 or explicitly specify None (unicode) is a matter of personal taste, although working with unicode may make porting your own application to Python 3 easier.

Upgrading from Genshi 0.5.x to 0.6.x

Required Python Version

Support for Python 2.3 has been dropped in this release. Python 2.4 is now the minimum version of Python required to run Genshi.

The XPath engine has been completely overhauled for this version. Some patterns that previously matched incorrectly will no longer match, and the other way around. In such cases, the XPath expressions need to be fixed in your application and templates.

Upgrading from Genshi 0.4.x to 0.5.x

Error Handling

The default error handling mode has been changed to "strict". This means that accessing variables not defined in the template data will now generate an immediate exception, as will accessing object attributes or dictionary keys that don't exist. If your templates rely on the old lenient behavior, you can configure Genshi to use that instead. See the documentation for details on how to do that. But be warned that lenient error handling may be removed completely in a future release.

Match Template Processing

There has also been a subtle change to how `py:match` templates are processed: in previous versions, all match templates would be applied to the content generated by the matching template, and only the matching template itself was applied recursively to the original content. This behavior resulted in problems with many kinds of recursive matching, and hence was changed for 0.5: now, all match templates declared before the matching template are applied to the original content, and match templates declared after the matching template are applied to the generated content. This change should not have any effect on most applications, but you may want to check your use of match templates to make sure.

Text Templates

Genshi 0.5 introduces a new, alternative syntax for text templates, which is more flexible and powerful compared to the old syntax. For backwards compatibility, this new syntax is not used by default, though it will be in a future version. It is recommended that you migrate to using this new syntax. To do so, simply rename any references in your code to `TextTemplate` to `NewTextTemplate`. To explicitly use the old syntax, use `OldTextTemplate` instead, so that you can be sure you'll be using the same language when the default in Genshi is changed (at least until the old implementation is completely removed).

`Markup` Constructor

The `Markup` class no longer has a specialized constructor. The old (undocumented) constructor provided a shorthand for doing positional substitutions. If you have code like this:

```
Markup('<b>%s</b>', name)
```

You must replace it by the more explicit:

```
Markup('<b>%s</b>') % name
```

`Template` Constructor

The constructor of the `Template` class and its subclasses has changed slightly: instead of the optional `basedir` parameter, it now expects an (also optional) `filepath` parameter, which specifies the absolute path to the template. You probably aren't using those constructors directly, anyway, but using the `TemplateLoader` API instead.

Upgrading from Genshi 0.3.x to 0.4.x

The modules `genshi.filters` and `genshi.template` have been refactored into packages containing multiple modules. While code using the regular APIs should continue to work without problems, you should make sure to remove any leftover traces of the files `filters.py` and `template.py` in the `genshi` package on the installation path (including the corresponding `.pyc` files). This is not necessary when Genshi was installed as a Python egg.

Results of evaluating template expressions are no longer implicitly called if they are callable. If you have been using that feature, you will need to add the parenthesis to actually call the function.

Instances of `genshi.core.Attrs` are now immutable. Filters manipulating the attributes in a stream may need to be updated. Also, the `Attrs` class no longer automatically wraps all attribute names in `QName` objects, so users of the `Attrs` class need to do this themselves. See the documentation of the `Attrs` class for more information.

Upgrading from Markup

Prior to version 0.3, the name of the Genshi project was "Markup". The name change means that you will have to adjust your import statements and the namespace URI of XML templates, among other things:

- The package name was changed from "markup" to "genshi". Please adjust any import statements referring to the old package name.
- The namespace URI for directives in Genshi XML templates has changed from `http://markup.edgewall.org/` to `http://genshi.edgewall.org/`. Please update the `xmlns:py` declaration in your template files accordingly.

Furthermore, due to the inclusion of a text-based template language, the class:

```
markup.template.Template
```

has been renamed to:

```
genshi.template.MarkupTemplate
```

If you've been using the Template class directly, you'll need to update your code (a simple find/replace should do—the API itself did not change).

1.2 Usage

1.2.1 Markup Streams

A stream is the common representation of markup as a *stream of events*.

Basics

A stream can be attained in a number of ways. It can be:

- the result of parsing XML or HTML text, or

- the result of selecting a subset of another stream using XPath, or
- programmatically generated.

For example, the functions `XML()` and `HTML()` can be used to convert literal XML or HTML text to a markup stream:

```
>>> from genshi import XML
>>> stream = XML('<p class="intro">Some text and '
...              '<a href="http://example.org/">a link</a>.'
...              '<br/></p>')
>>> stream
<genshi.core.Stream object at ...>
```

The stream is the result of parsing the text into events. Each event is a tuple of the form `(kind, data, pos)`, where:

- `kind` defines what kind of event it is (such as the start of an element, text, a comment, etc).
- `data` is the actual data associated with the event. How this looks depends on the event kind (see *event kinds*)
- `pos` is a `(filename, lineno, column)` tuple that describes where the event "comes from".

```
>>> for kind, data, pos in stream:
...     print('%s %r %r' % (kind, data, pos))
...
START (QName('p'), Attrs([(QName('class'), u'intro')])) (None, 1, 0)
TEXT u'Some text and ' (None, 1, 17)
START (QName('a'), Attrs([(QName('href'), u'http://example.org/')])) (None, 1, 31)
TEXT u'a link' (None, 1, 61)
END QName('a') (None, 1, 67)
TEXT u'.' (None, 1, 71)
START (QName('br'), Attrs()) (None, 1, 72)
END QName('br') (None, 1, 77)
END QName('p') (None, 1, 77)
```

Filtering

One important feature of markup streams is that you can apply *filters* to the stream, either filters that come with Genshi, or your own custom filters.

A filter is simply a callable that accepts the stream as parameter, and returns the filtered stream:

```
def noop(stream):
    """A filter that doesn't actually do anything with the stream."""
    for kind, data, pos in stream:
        yield kind, data, pos
```

Filters can be applied in a number of ways. The simplest is to just call the filter directly:

```
stream = noop(stream)
```

The `Stream` class also provides a `filter()` method, which takes an arbitrary number of filter callables and applies them all:

```
stream = stream.filter(noop)
```

Finally, filters can also be applied using the *bitwise or* operator (`|`), which allows a syntax similar to pipes on Unix shells:

```
stream = stream | noop
```

One example of a filter included with Genshi is the `HTMLSanitizer` in `genshi.filters`. It processes a stream of HTML markup, and strips out any potentially dangerous constructs, such as Javascript event handlers. `HTMLSanitizer` is not a function, but rather a class that implements `__call__`, which means instances of the class are callable:

```
stream = stream | HTMLSanitizer()
```

Both the `filter()` method and the pipe operator allow easy chaining of filters:

```
from genshi.filters import HTMLSanitizer
stream = stream.filter(noop, HTMLSanitizer())
```

That is equivalent to:

```
stream = stream | noop | HTMLSanitizer()
```

For more information about the built-in filters, see Stream Filters.

Serialization

Serialization means producing some kind of textual output from a stream of events, which you'll need when you want to transmit or store the results of generating or otherwise processing markup.

The `Stream` class provides two methods for serialization: `serialize()` and `render()`. The former is a generator that yields chunks of `Markup` objects (which are basically unicode strings that are considered safe for output on the web). The latter returns a single string, by default UTF-8 encoded.

Here's the output from `serialize()`:

```
>>> for output in stream.serialize():
...     print(repr(output))
...
<Markup u'<p class="intro">'>
<Markup u'Some text and '>
<Markup u'<a href="http://example.org/">'>
<Markup u'a link'>
<Markup u'</a>'>
<Markup u'.'>
<Markup u'<br/>'>
<Markup u'</p>'>
```

And here's the output from `render()`:

```
>>> print(stream.render())
<p class="intro">Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

Both methods can be passed a `method` parameter that determines how exactly the events are serialized to text. This parameter can be either a string or a custom serializer class:

```
>>> print(stream.render('html'))
<p class="intro">Some text and <a href="http://example.org/">a link</a>.<br></p>
```

Note how the <*br*> element isn't closed, which is the right thing to do for HTML. See *serialization methods* for more details.

In addition, the `render()` method takes an `encoding` parameter, which defaults to "UTF-8". If set to `None`, the result will be a unicode string.

The different serializer classes in `genshi.output` can also be used directly:

```
>>> from genshi.filters import HTMLSanitizer
>>> from genshi.output import TextSerializer
>>> print(''.join(TextSerializer()(HTMLSanitizer()(stream))))
Some text and a link.
```

The pipe operator allows a nicer syntax:

```
>>> print(stream | HTMLSanitizer() | TextSerializer())
Some text and a link.
```

Serialization Methods

Genshi supports the use of different serialization methods to use for creating a text representation of a markup stream.

xml The `XMLSerializer` is the default serialization method and results in proper XML output including namespace support, the XML declaration, CDATA sections, and so on. It is not generally not suitable for serving HTML or XHTML web pages (unless you want to use true XHTML 1.1), for which the `xhtml` and `html` serializers described below should be preferred.

xhtml The `XHTMLSerializer` is a specialization of the generic `XMLSerializer` that understands the peculiarities of producing XML-compliant output that can also be parsed without problems by the HTML parsers found in modern web browsers. Thus, the output by this serializer should be usable whether sent as "text/html" or "application/xhtml+html" (although there are a lot of subtle issues to pay attention to when switching between the two, in particular with respect to differences in the DOM and CSS).

For example, instead of rendering a script tag as `<script/>` (which confuses the HTML parser in many browsers), it will produce `<script></script>`. Also, it will normalize any boolean attributes values that are minimized in HTML, so that for example `<hr noshade="1"/>` becomes `<hr noshade="noshade" />`.

This serializer supports the use of namespaces for compound documents, for example to use inline SVG inside an XHTML document.

html The `HTMLSerializer` produces proper HTML markup. The main differences compared to `xhtml` serialization are that boolean attributes are minimized, empty tags are not self-closing (so it's `<br>` instead of `<br />`), and that the contents of `<script>` and `<style>` elements are not escaped.

text The `TextSerializer` produces plain text from markup streams. This is useful primarily for text templates, but can also be used to produce plain text output from markup templates or other sources.

Serialization Options

Both `serialize()` and `render()` support additional keyword arguments that are passed through to the initializer of the serializer class. The following options are supported by the built-in serializers:

strip_whitespace Whether the serializer should remove trailing spaces and empty lines. Defaults to `True`.

(This option is not available for serialization to plain text.)

doctype A `(name, pubid, sysid)` tuple defining the name, publid identifier, and system identifier of a `DOCTYPE` declaration to prepend to the generated output. If provided, this declaration will override any `DOCTYPE` declaration in the stream.

The parameter can also be specified as a string to refer to commonly used doctypes:

Shorthand	DOCTYPE
`html` or `html-strict`	HTML 4.01 Strict
`html-transitional`	HTML 4.01 Transitional
`html-frameset`	HTML 4.01 Frameset
`html5`	DOCTYPE proposed for the work-in-progress HTML5 standard
`xhtml` or `xhtml-strict`	XHTML 1.0 Strict
`xhtml-transitional`	XHTML 1.0 Transitional
`xhtml-frameset`	XHTML 1.0 Frameset
`xhtml11`	XHTML 1.1
`svg` or `svg-full`	SVG 1.1
`svg-basic`	SVG 1.1 Basic
`svg-tiny`	SVG 1.1 Tiny

(This option is not available for serialization to plain text.)

namespace_prefixes The namespace prefixes to use for namespace that are not bound to a prefix in the stream itself.

(This option is not available for serialization to HTML or plain text.)

drop_xml_decl Whether to remove the XML declaration (the `<?xml ?>` part at the beginning of a document) when serializing. This defaults to `True` as an XML declaration throws some older browsers into "Quirks" rendering mode.

(This option is only available for serialization to XHTML.)

strip_markup Whether the text serializer should detect and remove any tags or entity encoded characters in the text.

(This option is only available for serialization to plain text.)

Using XPath

XPath can be used to extract a specific subset of the stream via the `select()` method:

```
>>> substream = stream.select('a')
>>> substream
<genshi.core.Stream object at ...>
>>> print(substream)
<a href="http://example.org/">a link</a>
```

Often, streams cannot be reused: in the above example, the sub-stream is based on a generator. Once it has been serialized, it will have been fully consumed, and cannot be rendered again. To work around this, you can wrap such a stream in a `list`:

```
>>> from genshi import Stream
>>> substream = Stream(list(stream.select('a')))
>>> substream
<genshi.core.Stream object at ...>
>>> print(substream)
<a href="http://example.org/">a link</a>
>>> print(substream.select('@href'))
http://example.org/
>>> print(substream.select('text()'))
a link
```

See Using XPath in Genshi for more information about the XPath support in Genshi.

Event Kinds

Every event in a stream is of one of several *kinds*, which also determines what the `data` item of the event tuple looks like. The different kinds of events are documented below.

Note: The `data` item is generally immutable. If the data is to be modified when processing a stream, it must be replaced by a new tuple. Effectively, this means the entire event tuple is immutable.

START

The opening tag of an element.

For this kind of event, the `data` item is a tuple of the form `(tagname, attrs)`, where `tagname` is a `QName` instance describing the qualified name of the tag, and `attrs` is an `Attrs` instance containing the attribute names and values associated with the tag (excluding namespace declarations):

```
START, (QName('p'), Attrs([(QName('class'), u'intro')])), pos
```

END

The closing tag of an element.

The `data` item of end events consists of just a `QName` instance describing the qualified name of the tag:

```
END, QName('p'), pos
```

TEXT

Character data outside of elements and comments.

For text events, the `data` item should be a unicode object:

```
TEXT, u'Hello, world!', pos
```

START_NS

The start of a namespace mapping, binding a namespace prefix to a URI.

The `data` item of this kind of event is a tuple of the form `(prefix, uri)`, where `prefix` is the namespace prefix and `uri` is the full URI to which the prefix is bound. Both should be unicode objects. If the namespace is not bound to any prefix, the `prefix` item is an empty string:

```
START_NS, (u'svg', u'http://www.w3.org/2000/svg'), pos
```

END_NS

The end of a namespace mapping.

The `data` item of such events consists of only the namespace prefix (a unicode object):

```
END_NS, u'svg', pos
```

DOCTYPE

A document type declaration.

For this type of event, the `data` item is a tuple of the form `(name, pubid, sysid)`, where `name` is the name of the root element, `pubid` is the public identifier of the DTD (or `None`), and `sysid` is the system identifier of the DTD (or `None`):

```
DOCTYPE, (u'html', u'-//W3C//DTD XHTML 1.0 Transitional//EN', \
          u'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'), pos
```

COMMENT

A comment.

For such events, the `data` item is a unicode object containing all character data between the comment delimiters:

```
COMMENT, u'Commented out', pos
```

PI

A processing instruction.

The `data` item is a tuple of the form `(target, data)` for processing instructions, where `target` is the target of the PI (used to identify the application by which the instruction should be processed), and `data` is text following the target (excluding the terminating question mark):

```
PI, (u'php', u'echo "Yo" '), pos
```

START_CDATA

Marks the beginning of a `CDATA` section.

The `data` item for such events is always `None`:

```
START_CDATA, None, pos
```

END_CDATA

Marks the end of a `CDATA` section.

The `data` item for such events is always `None`:

```
END_CDATA, None, pos
```

1.2.2 Genshi Templating Basics

Genshi provides a template engine that can be used for generating either markup (such as HTML or XML) or plain text. While both share some of the syntax (and much of the underlying implementation) they are essentially separate languages.

This document describes the common parts of the template engine and will be most useful as reference to those developing Genshi templates. Templates are XML or plain text files that include processing *directives* that affect how the template is rendered, and template *expressions* that are dynamically substituted by variable data.

Synopsis

A Genshi *markup template* is a well-formed XML document with embedded Python used for control flow and variable substitution. Markup templates should be used to generate any kind of HTML or XML output, as they provide a number of advantages over simple text-based templates (such as automatic escaping of variable data).

The following is a simple Genshi markup template:

```
<?python
  title = "A Genshi Template"
  fruits = ["apple", "orange", "kiwi"]
?>
<html xmlns:py="http://genshi.edgewall.org/">
  <head>
    <title py:content="title">This is replaced.</title>
  </head>

  <body>
    <p>These are some of my favorite fruits:</p>
    <ul>
      <li py:for="fruit in fruits">
        I like ${fruit}s
      </li>
    </ul>
  </body>
</html>
```

This example shows:

1. a Python code block in a processing instruction
2. the Genshi namespace declaration
3. usage of templates directives (`py:content` and `py:for`)
4. an inline Python expression (`${fruit}`).

The template would generate output similar to this:

```
<html>
  <head>
    <title>A Genshi Template</title>
  </head>

  <body>
    <p>These are some of my favorite fruits:</p>
    <ul>
      <li>I like apples</li>
      <li>I like oranges</li>
      <li>I like kiwis</li>
    </ul>
  </body>
</html>
```

A *text template* is a simple plain text document that can also contain embedded Python code. Text templates are intended to be used for simple *non-markup* text formats, such as the body of an plain text email. For example:

```
Dear $name,

These are some of my favorite fruits:
#for fruit in fruits
 * $fruit
#end
```

Python API

The Python code required for templating with Genshi is generally based on the following pattern:

- Attain a `MarkupTemplate` or `TextTemplate` object from a string or file-like object containing the template source. This can either be done directly, or through a `TemplateLoader` instance.
- Call the `generate()` method of the template, passing any data that should be made available to the template as keyword arguments.
- Serialize the resulting stream using its `render()` method.

For example:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<h1>Hello, $name!</h1>')
>>> stream = tmpl.generate(name='world')
>>> print(stream.render('xhtml'))
<h1>Hello, world!</h1>
```

Note: See the Serialization section of the Markup Streams page for information on configuring template output options.

Using a text template is similar:

```
>>> from genshi.template import TextTemplate
>>> tmpl = TextTemplate('Hello, $name!')
>>> stream = tmpl.generate(name='world')
>>> print(stream)
Hello, world!
```

Note: If you want to use text templates, you should consider using the `NewTextTemplate` class instead of simply `TextTemplate`. See the Text Template Language page.

Using a template loader provides the advantage that "compiled" templates are automatically cached, and only parsed again when the template file changes. In addition, it enables the use of a *template search path*, allowing template directories to be spread across different file-system locations. Using a template loader would generally look as follows:

```
from genshi.template import TemplateLoader
loader = TemplateLoader([templates_dir1, templates_dir2])
tmpl = loader.load('test.html')
stream = tmpl.generate(title='Hello, world!')
print(stream.render())
```

See the API documentation for details on using Genshi via the Python API.

Template Expressions and Code Blocks

Python expressions can be used in text and directive arguments. An expression is substituted with the result of its evaluation against the template data. Expressions in text (which includes the values of non-directive attributes) need to prefixed with a dollar sign (`$`) and usually enclosed in curly braces (`{...}`).

If the expression starts with a letter and contains only letters, digits, dots, and underscores, the curly braces may be omitted. In all other cases, the braces are required so that the template processor knows where the expression ends:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<em>${items[0].capitalize()} item</em>')
>>> print(tmpl.generate(items=['first', 'second']))
<em>First item</em>
```

Expressions support the full power of Python. In addition, it is possible to access items in a dictionary using "dotted notation" (i.e. as if they were attributes), and vice-versa (i.e. access attributes as if they were items in a dictionary):

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<em>${dict.foo}</em>')
>>> print(tmpl.generate(dict={'foo': 'bar'}))
<em>bar</em>
```

Because there are two ways to access either attributes or items, expressions do not raise the standard `AttributeError` or `IndexError` exceptions, but rather an exception of the type `UndefinedError`. The same kind of error is raised when you try to use a top-level variable that is not in the context data. See *Error Handling* below for details on how such errors are handled.

Escaping

If you need to include a literal dollar sign in the output where Genshi would normally detect an expression, you can simply add another dollar sign:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<em>$foo</em>') # Wanted "$foo" as literal output
>>> print(tmpl.generate())
Traceback (most recent call last):
  ...
UndefinedError: "foo" not defined
>>> tmpl = MarkupTemplate('<em>$$foo</em>')
>>> print(tmpl.generate())
<em>$foo</em>
```

But note that this is not necessary if the characters following the dollar sign do not qualify as an expression. For example, the following needs no escaping:

```
>>> tmpl = MarkupTemplate('<script>$(function() {})</script>')
>>> print(tmpl.generate())
<script>$(function() {})</script>
```

On the other hand, Genshi will always replace two dollar signs in text with a single dollar sign, so you'll need to use three dollar signs to get two in the output:

```
>>> tmpl = MarkupTemplate('<script>$$$("div")</script>')
>>> print(tmpl.generate())
<script>$$("div")</script>
```

Code Blocks

Templates also support full Python code blocks, using the `<?python ?>` processing instruction in XML templates:

```
<div>
  <?python
      from genshi.builder import tag
      def greeting(name):
          return tag.b('Hello, %s!' % name) ?>
  ${greeting('world')}
</div>
```

This will produce the following output:

```
<div>
  <b>Hello, world!</b>
</div>
```

In text templates (although only those using the new syntax introduced in Genshi 0.5), code blocks use the special `{% python %}` directive:

```
{% python
    from genshi.builder import tag
    def greeting(name):
        return 'Hello, %s!' % name
%}
${greeting('world')}
```

This will produce the following output:

```
Hello, world!
```

Code blocks can import modules, define classes and functions, and basically do anything you can do in normal Python code. What code blocks can *not* do is to produce content that is emitted directly tp the generated output.

Note: Using the `print` statement will print to the standard output stream, just as it does for other Python code in your application.

Unlike expressions, Python code in `<?python ?>` processing instructions can not use item and attribute access in an interchangeable manner. That means that "dotted notation" is always attribute access, and vice-versa.

The support for Python code blocks in templates is not supposed to encourage mixing application code into templates, which is generally considered bad design. If you're using many code blocks, that may be a sign that you should move such code into separate Python modules.

If you'd rather not allow the use of Python code blocks in templates, you can simply set the `allow_exec` parameter (available on the `Template` and the `TemplateLoader` initializers) to `False`. In that case Genshi will raise a syntax error when a `<?python ?>` processing instruction is encountered. But please note that disallowing code blocks in templates does not turn Genshi into a sandboxable template engine; there are sufficient ways to do harm even using plain expressions.

Error Handling

By default, Genshi raises an `UndefinedError` if a template expression attempts to access a variable that is not defined:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<p>${doh}</p>')
>>> tmpl.generate().render('xhtml')
Traceback (most recent call last):
  ...
UndefinedError: "doh" not defined
```

You can change this behavior by setting the variable lookup mode to "lenient". In that case, accessing undefined variables returns an *Undefined* object, meaning that the expression does not fail immediately. See below for details.

If you need to check whether a variable exists in the template context, use the *defined* or the *value_of* function described below. To check for existence of attributes on an object, or keys in a dictionary, use the `hasattr()`, `getattr()` or `get()` functions, or the `in` operator, just as you would in regular Python code:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<p>${defined("doh")}</p>')
>>> print(tmpl.generate().render('xhtml'))
<p>False</p>
```

Note: Lenient error handling was the default in Genshi prior to version 0.5. Strict mode was introduced in version 0.4, and became the default in 0.5. The reason for this change was that the lenient error handling was masking actual errors in templates, thereby also making it harder to debug some problems.

Lenient Mode If you instruct Genshi to use the lenient variable lookup mode, it allows you to access variables that are not defined, without raising an `UndefinedError`.

This mode can be chosen by passing the `lookup='lenient'` keyword argument to the template initializer, or by passing the `variable_lookup='lenient'` keyword argument to the `TemplateLoader` initializer:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<p>${doh}</p>', lookup='lenient')
>>> print(tmpl.generate().render('xhtml'))
<p></p>
```

You *will* however get an exception if you try to call an undefined variable, or do anything else with it, such as accessing its attributes:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<p>${doh.oops}</p>', lookup='lenient')
>>> print(tmpl.generate().render('xhtml'))
Traceback (most recent call last):
  ...
UndefinedError: "doh" not defined
```

If you need to know whether a variable is defined, you can check its type against the `Undefined` class, for example in a conditional directive:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('<p>${type(doh) is not Undefined}</p>',
...                       lookup='lenient')
>>> print(tmpl.generate().render('xhtml'))
<p>False</p>
```

Alternatively, the built-in functions *defined* or *value_of* can be used in this case.

Custom Modes In addition to the built-in "lenient" and "strict" modes, it is also possible to use a custom error handling mode. For example, you could use lenient error handling in a production environment, while also logging a warning when an undefined variable is referenced.

See the API documentation of the `genshi.template.eval` module for details.

Built-in Functions & Types

The following functions and types are available by default in template code, in addition to the standard built-ins that are available to all Python code.

`defined(name)` This function determines whether a variable of the specified name exists in the context data, and returns `True` if it does.

`value_of(name, default=None)` This function returns the value of the variable with the specified name if such a variable is defined, and returns the value of the `default` parameter if no such variable is defined.

`Markup(text)` The `Markup` type marks a given string as being safe for inclusion in markup, meaning it will *not* be escaped in the serialization stage. Use this with care, as not escaping a user-provided string may allow malicious users to open your web site to cross-site scripting attacks.

`Undefined` The `Undefined` type can be used to check whether a reference variable is defined, as explained in *error handling*.

Template Directives

Directives provide control flow functionality for templates, such as conditions or iteration. As the syntax for directives depends on whether you're using markup or text templates, refer to the XML Template Language or Text Template Language pages for information.

1.2.3 Genshi XML Template Language

Genshi provides a XML-based template language that is heavily inspired by Kid, which in turn was inspired by a number of existing template languages, namely XSLT, TAL, and PHP.

This document describes the template language and will be most useful as reference to those developing Genshi XML templates. Templates are XML files of some kind (such as XHTML) that include processing *directives* (elements or attributes identified by a separate namespace) that affect how the template is rendered, and template expressions that are dynamically substituted by variable data.

See Genshi Templating Basics for general information on embedding Python code in templates.

Template Directives

Directives are elements and/or attributes in the template that are identified by the namespace `http://genshi.edgewall.org/`. They can affect how the template is rendered in a number of ways: Genshi provides directives for conditionals and looping, among others.

To use directives in a template, the namespace must be declared, which is usually done on the root element:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      lang="en">
  ...
</html>
```

In this example, the default namespace is set to the XHTML namespace, and the namespace for Genshi directives is bound to the prefix "py".

All directives can be applied as attributes, and some can also be used as elements. The `if` directives for conditionals, for example, can be used in both ways:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      lang="en">
  ...
  <div py:if="foo">
    <p>Bar</p>
  </div>
  ...
</html>
```

This is basically equivalent to the following:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      lang="en">
  ...
  <py:if test="foo">
    <div>
      <p>Bar</p>
    </div>
  </py:if>
  ...
</html>
```

The rationale behind the second form is that directives do not always map naturally to elements in the template. In such cases, the `py:strip` directive can be used to strip off the unwanted element, or the directive can simply be used as an element.

Conditional Sections

`py:if` The element and its content is only rendered if the expression evaluates to a truth value:

```
<div>
  <b py:if="foo">${bar}</b>
</div>
```

Given the data `foo=True` and `bar='Hello'` in the template context, this would produce:

```
<div>
  <b>Hello</b>
</div>
```

But setting `foo=False` would result in the following output:

```
<div>
</div>
```

This directive can also be used as an element:

```
<div>
  <py:if test="foo">
    <b>${bar}</b>
  </py:if>
</div>
```

py:choose The `py:choose` directive, in combination with the directives `py:when` and `py:otherwise` provides advanced conditional processing for rendering one of several alternatives. The first matching `py:when` branch is rendered, or, if no `py:when` branch matches, the `py:otherwise` branch is rendered.

If the `py:choose` directive is empty the nested `py:when` directives will be tested for truth:

```
<div py:choose="">
  <span py:when="0 == 1">0</span>
  <span py:when="1 == 1">1</span>
  <span py:otherwise="">2</span>
</div>
```

This would produce the following output:

```
<div>
  <span>1</span>
</div>
```

If the `py:choose` directive contains an expression the nested `py:when` directives will be tested for equality to the parent `py:choose` value:

```
<div py:choose="1">
  <span py:when="0">0</span>
  <span py:when="1">1</span>
  <span py:otherwise="">2</span>
</div>
```

This would produce the following output:

```
<div>
  <span>1</span>
</div>
```

These directives can also be used as elements:

```
<py:choose test="1">
  <py:when test="0">0</py:when>
  <py:when test="1">1</py:when>
  <py:otherwise>2</py:otherwise>
</py:choose>
```

Looping

py:for The element is repeated for every item in an iterable:

```
<ul>
  <li py:for="item in items">${item}</li>
</ul>
```

Given `items=[1, 2, 3]` in the context data, this would produce:

```
<ul>
  <li>1</li><li>2</li><li>3</li>
</ul>
```

This directive can also be used as an element:

```
<ul>
  <py:for each="item in items">
    <li>${item}</li>
  </py:for>
</ul>
```

Snippet Reuse

py:def The `py:def` directive can be used to create macros, i.e. snippets of template code that have a name and optionally some parameters, and that can be inserted in other places:

```
<div>
  <p py:def="greeting(name)" class="greeting">
    Hello, ${name}!
  </p>
  ${greeting('world')}
  ${greeting('everyone else')}
</div>
```

The above would be rendered to:

```
<div>
  <p class="greeting">
    Hello, world!
  </p>
  <p class="greeting">
    Hello, everyone else!
  </p>
</div>
```

If a macro doesn't require parameters, it can be defined without the parenthesis. For example:

```
<div>
  <p py:def="greeting" class="greeting">
    Hello, world!
  </p>
  ${greeting()}
</div>
```

The above would be rendered to:

```
<div>
  <p class="greeting">
    Hello, world!
  </p>
</div>
```

This directive can also be used as an element:

```
<div>
  <py:def function="greeting(name)">
    <p class="greeting">Hello, ${name}!</p>
```

```
  </py:def>
</div>
```

py:match This directive defines a *match template*: given an XPath expression, it replaces any element in the template that matches the expression with its own content.

For example, the match template defined in the following template matches any element with the tag name "greeting":

```
<div>
  <span py:match="greeting">
    Hello ${select('@name')}
  </span>
  <greeting name="Dude" />
</div>
```

This would result in the following output:

```
<div>
  <span>
    Hello Dude
  </span>
</div>
```

Inside the body of a `py:match` directive, the `select(path)` function is made available so that parts or all of the original element can be incorporated in the output of the match template. See Using XPath for more information about this function.

Match templates are applied both to the original markup as well to the generated markup. The order in which they are applied depends on the order they are declared in the template source: a match template defined after another match template is applied to the output generated by the first match template. The match templates basically form a pipeline.

This directive can also be used as an element:

```
<div>
  <py:match path="greeting">
    <span>Hello ${select('@name')}</span>
  </py:match>
  <greeting name="Dude" />
</div>
```

When used this way, the `py:match` directive can also be annotated with a couple of optimization hints. For example, the following informs the matching engine that the match should only be applied once:

```
<py:match path="body" once="true">
  <body py:attrs="select('@*')">
    <div id="header">...</div>
    ${select("*|text()")}
    <div id="footer">...</div>
  </body>
</py:match>
```

The following optimization hints are recognized:

Attribute	Default	Description
`buffer`	`true`	Whether the matched content should be buffered in memory. Buffering can improve performance a bit at the cost of needing more memory during rendering. Buffering is ''required" for match templates that contain more than one invocation of the `select()` function. If there is only one call, and the matched content can potentially be very long, consider disabling buffering to avoid excessive memory use.
`once`	`false`	Whether the engine should stop looking for more matching elements after the first match. Use this on match templates that match elements that can only occur once in the stream, such as the `<head>` or `<body>` elements in an HTML template, or elements with a specific ID.
`recursive`	`true`	Whether the match template should be applied to its own output. Note that `once` implies non-recursive behavior, so this attribute only needs to be set for match templates that don't also have `once` set.

Note: The `py:match` optimization hints were added in the 0.5 release. In earlier versions, the attributes have no effect.

Variable Binding

`py:with` The `py:with` directive lets you assign expressions to variables, which can be used to make expressions inside the directive less verbose and more efficient. For example, if you need use the expression `author.posts` more than once, and that actually results in a database query, assigning the results to a variable using this directive would probably help.

For example:

```
<div>
  <span py:with="y=7; z=x+10">$x $y $z</span>
</div>
```

Given `x=42` in the context data, this would produce:

```
<div>
  <span>42 7 52</span>
</div>
```

This directive can also be used as an element:

```
<div>
  <py:with vars="y=7; z=x+10">$x $y $z</py:with>
</div>
```

Note that if a variable of the same name already existed outside of the scope of the `py:with` directive, it will **not** be overwritten. Instead, it will have the same value it had prior to the `py:with` assignment. Effectively, this means that variables are immutable in Genshi.

Structure Manipulation

`py:attrs` This directive adds, modifies or removes attributes from the element:

```
<ul>
  <li py:attrs="foo">Bar</li>
</ul>
```

Given `foo={'class': 'collapse'}` in the template context, this would produce:

```
<ul>
  <li class="collapse">Bar</li>
</ul>
```

Attributes with the value `None` are omitted, so given `foo={'class': None}` in the context for the same template this would produce:

```
<ul>
  <li>Bar</li>
</ul>
```

This directive can only be used as an attribute.

py:content This directive replaces any nested content with the result of evaluating the expression:

```
<ul>
  <li py:content="bar">Hello</li>
</ul>
```

Given `bar='Bye'` in the context data, this would produce:

```
<ul>
  <li>Bye</li>
</ul>
```

This directive can only be used as an attribute.

py:replace This directive replaces the element itself with the result of evaluating the expression:

```
<div>
  <span py:replace="bar">Hello</span>
</div>
```

Given `bar='Bye'` in the context data, this would produce:

```
<div>
  Bye
</div>
```

This directive can also be used as an element (since version 0.5):

```
<div>
  <py:replace value="title">Placeholder</py:replace>
</div>
```

py:strip This directive conditionally strips the top-level element from the output. When the value of the `py:strip` attribute evaluates to `True`, the element is stripped from the output:

```
<div>
  <div py:strip="True"><b>foo</b></div>
</div>
```

This would be rendered as:

```
<div>
  <b>foo</b>
</div>
```

As a shorthand, if the value of the `py:strip` attribute is empty, that has the same effect as using a truth value (i.e. the element is stripped).

Processing Order

It is possible to attach multiple directives to a single element, although not all combinations make sense. When multiple directives are encountered, they are processed in the following order:

1. *py:def*
2. *py:match*
3. *py:when*
4. *py:otherwise*
5. *py:for*
6. *py:if*
7. *py:choose*
8. *py:with*
9. *py:replace*
10. *py:content*
11. *py:attrs*
12. *py:strip*

Includes

To reuse common snippets of template code, you can include other files using XInclude.

For this, you need to declare the XInclude namespace (commonly bound to the prefix "xi") and use the `<xi:include>` element where you want the external file to be pulled in:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="base.html" />
  ...
</html>
```

Include paths are relative to the filename of the template currently being processed. So if the example above was in the file "`myapp/index.html`" (relative to the template search path), the XInclude processor would look for the included file at "`myapp/base.html`". You can also use Unix-style relative paths, for example "`../base.html`" to look in the parent directory.

Any content included this way is inserted into the generated output instead of the `<xi:include>` element. The included template sees the same context data. *Match templates* and *macros* in the included template are also available to the including template after the point it was included.

By default, an error will be raised if an included file is not found. If that's not what you want, you can specify fallback content that should be used if the include fails. For example, to to make the include above fail silently, you'd write:

```
<xi:include href="base.html"><xi:fallback /></xi:include>
```

See the XInclude specification for more about fallback content. Note though that Genshi currently only supports a small subset of XInclude.

Dynamic Includes

Incudes in Genshi are fully dynamic: Just like normal attributes, the *href* attribute accepts expressions, and *directives* can be used on the `<xi:include />` element just as on any other element, meaning you can do things like conditional includes:

```
<xi:include href="${name}.html" py:if="not in_popup"
            py:for="name in ('foo', 'bar', 'baz')" />
```

Including Text Templates

The `parse` attribute of the `<xi:include>` element can be used to specify whether the included template is an XML template or a text template (using the new syntax added in Genshi 0.5):

```
<xi:include href="myscript.js" parse="text" />
```

This example would load the `myscript.js` file as a `NewTextTemplate`. See text templates for details on the syntax of text templates.

Comments

Normal XML/HTML comment syntax can be used in templates:

```
<!-- this is a comment -->
```

However, such comments get passed through the processing pipeline and are by default included in the final output. If that's not desired, prefix the comment text with an exclamation mark:

```
<!-- !this is a comment too, but one that will be stripped from the output -->
```

Note that it does not matter whether there's whitespace before or after the exclamation mark, so the above could also be written as follows:

```
<!--! this is a comment too, but one that will be stripped from the output -->
```

1.2.4 Genshi Text Template Language

In addition to the XML-based template language, Genshi provides a simple text-based template language, intended for basic plain text generation needs. The language is similar to the Django template language.

This document describes the template language and will be most useful as reference to those developing Genshi text templates. Templates are text files of some kind that include processing *directives* that affect how the template is rendered, and template expressions that are dynamically substituted by variable data.

See Genshi Templating Basics for general information on embedding Python code in templates.

Note: Actually, Genshi currently has two different syntaxes for text templates languages: One implemented by the class `OldTextTemplate` and another implemented by `NewTextTemplate`. This documentation concentrates on the latter, which is planned to completely replace the older syntax. The older syntax is briefly described under *legacy*.

Template Directives

Directives are template commands enclosed by `{% ... %}` characters. They can affect how the template is rendered in a number of ways: Genshi provides directives for conditionals and looping, among others.

Each directive must be terminated using an `{% end %}` marker. You can add a string inside the `{% end %}` marker, for example to document which directive is being closed, or even the expression associated with that directive. Any text after `end` inside the delimiters is ignored, and effectively treated as a comment.

If you want to include a literal delimiter in the output, you need to escape it by prepending a backslash character (`\`).

Conditional Sections

`{% if %}` The content is only rendered if the expression evaluates to a truth value:

```
{% if foo %}
  ${bar}
{% end %}
```

Given the data `foo=True` and `bar='Hello'` in the template context, this would produce:

```
Hello
```

`{% choose %}` The `choose` directive, in combination with the directives `when` and `otherwise`, provides advanced contional processing for rendering one of several alternatives. The first matching `when` branch is rendered, or, if no `when` branch matches, the `otherwise` branch is be rendered.

If the `choose` directive has no argument the nested `when` directives will be tested for truth:

```
The answer is:
{% choose %}
  {% when 0 == 1 %}0{% end %}
  {% when 1 == 1 %}1{% end %}
  {% otherwise %}2{% end %}
{% end %}
```

This would produce the following output:

```
The answer is:
  1
```

If the `choose` does have an argument, the nested `when` directives will be tested for equality to the parent `choose` value:

```
The answer is:
{% choose 1 %}\
  {% when 0 %}0{% end %}\
  {% when 1 %}1{% end %}\
  {% otherwise %}2{% end %}\
{% end %}
```

This would produce the following output:

```
The answer is:
    1
```

Looping

{% for %} The content is repeated for every item in an iterable:

```
Your items:
{% for item in items %}\
  * ${item}
{% end %}
```

Given `items=[1, 2, 3]` in the context data, this would produce:

```
Your items
  * 1
  * 2
  * 3
```

Snippet Reuse

{% def %} The `def` directive can be used to create macros, i.e. snippets of template text that have a name and optionally some parameters, and that can be inserted in other places:

```
{% def greeting(name) %}
  Hello, ${name}!
{% end %}
${greeting('world')}
${greeting('everyone else')}
```

The above would be rendered to:

```
Hello, world!
Hello, everyone else!
```

If a macro doesn't require parameters, it can be defined without the parenthesis. For example:

```
{% def greeting %}
  Hello, world!
{% end %}
${greeting()}
```

The above would be rendered to:

```
Hello, world!
```

{% include %} To reuse common parts of template text across template files, you can include other files using the `include` directive:

```
{% include base.txt %}
```

Any content included this way is inserted into the generated output. The included template sees the context data as it exists at the point of the include. *Macros* in the included template are also available to the including template after the point it was included.

Include paths are relative to the filename of the template currently being processed. So if the example above was in the file "`myapp/mail.txt`" (relative to the template search path), the include directive would look for the included file at "`myapp/base.txt`". You can also use Unix-style relative paths, for example "`../base.txt`" to look in the parent directory.

Just like other directives, the argument to the `include` directive accepts any Python expression, so the path to the included template can be determined dynamically:

```
{% include ${'%s.txt' % filename} %}
```

Note that a `TemplateNotFound` exception is raised if an included file can't be found.

Note: The include directive for text templates was added in Genshi 0.5.

Variable Binding

`{% with %}` The `{% with %}` directive lets you assign expressions to variables, which can be used to make expressions inside the directive less verbose and more efficient. For example, if you need use the expression `author.posts` more than once, and that actually results in a database query, assigning the results to a variable using this directive would probably help.

For example:

```
Magic numbers!
{% with y=7; z=x+10 %}
  $x $y $z
{% end %}
```

Given `x=42` in the context data, this would produce:

```
Magic numbers!
  42 7 52
```

Note that if a variable of the same name already existed outside of the scope of the `with` directive, it will **not** be overwritten. Instead, it will have the same value it had prior to the `with` assignment. Effectively, this means that variables are immutable in Genshi.

White-space and Line Breaks

Note that space or line breaks around directives is never automatically removed. Consider the following example:

```
{% for item in items %}
  {% if item.visible %}
    ${item}
  {% end %}
{% end %}
```

This will result in two empty lines above and beneath every item, plus the spaces used for indentation. If you want to supress a line break, simply end the line with a backslash:

```
{% for item in items %}\
  {% if item.visible %}\
    ${item}
  {% end %}\
{% end %}\
```

Now there would be no empty lines between the items in the output. But you still get the spaces used for indentation, and because the line breaks are removed, they actually continue and add up between lines. There are numerous ways to control white-space in the output while keeping the template readable, such as moving the indentation into the delimiters, or moving the end delimiter on the next line, and so on.

Comments

Parts in templates can be commented out using the delimiters `{# ... #}`. Any content in comments are removed from the output.

```
{# This won't end up in the output #}
This will.
```

Just like directive delimiters, these can be escaped by prefixing with a backslash.

```
\{# This *will* end up in the output, including delimiters #}
This too.
```

Legacy Text Template Syntax

The syntax for text templates was redesigned in version 0.5 of Genshi to make the language more flexible and powerful. The older syntax is based on line starting with dollar signs, similar to e.g. Cheetah or Velocity.

A simple template using the old syntax looked like this:

```
Dear $name,

We have the following items for you:
#for item in items
 * $item
#end

All the best,
Foobar
```

Beyond the requirement of putting directives on separate lines prefixed with dollar signs, the language itself is very similar to the new one. Except that comments are lines that start with two # characters, and a line-break at the end of a directive is removed automatically.

Note: If you're using this old syntax, it is strongly recommended to migrate to the new syntax. Simply replace any references to `TextTemplate` by `NewTextTemplate` (and also change the text templates, of course). On the other hand, if you want to stick with the old syntax for a while longer, replace references to `TextTemplate` by `OldTextTemplate`; while `TextTemplate` is still an alias for the old language at this point, that will change in a future release. But also note that the old syntax may be dropped entirely in a future release.

1.2.5 Loading Templates

Genshi comes with a simple but flexible implementation of a template loader in the `genshi.template.loader` module. The loader provides caching of templates so they do not need to be reparsed when used, support for multiple template directories that together form a virtual search path, as well as support for different template loading strategies.

Usage

The basic usage pattern is simple: instantiate one `TemplateLoader` object and keep it around, then ask it to load a template whenever you need to load one:

```
from genshi.template import TemplateLoader

loader = TemplateLoader(['/path/to/dir1', '/path/to/dir2'],
```

```
                            auto_reload=True)
tmpl = loader.load('test.html')
```

When you try to load a template that can't be found, you get a `TemplateNotFound` error.

The default template class used by the loader is `MarkupTemplate`, but that can be overridden both with a different default (as a keyword argument to the `TemplateLoader` constructor), as well as on invocation of the `load()` method:

```
from genshi.template.text import NewTextTemplate

tmpl = loader.load('mail.txt', cls=NewTextTemplate)
```

Caching

The `TemplateLoader` class provides a simple in-memory cache for parsed template objects. This improves performance, because templates do not need to be reparsed every time they are rendered.

The size of this cache can be adjusted using the *max_cache_size* option on the `TemplateLoader` constructor. The value of that option determines the maximum number of template objects kept in the cache. When this limit is reached, any templates that haven't been used in a while get purged. Technically, this is a least-recently-used (LRU) cache, the default limit is set to 25 templates.

Automatic Reloading

Once a template has been cached, it will normally not get reparsed until it has been purged from the cache. This means that any changes to the template file are not taken into consideration as long as it is still found in the cache. As this is inconvenient in development scenarios, the `auto_reload` option allows for automatic cache invalidation based on whether the template source has changed.

```
from genshi.template import TemplateLoader

loader = TemplateLoader('templates', auto_reload=True, max_cache_size=100)
```

In production environments, automatic reloading should be disabled, as it does affect performance negatively.

Callback Interface

Sometimes you need to make sure that templates get properly configured after they have been loaded, but you only want to do that when the template is actually loaded and parsed, not when it is returned from the cache.

For such cases, the `TemplateLoader` provides a way to specify a callback function that gets invoked whenever a template is loaded. You can specify that callback by passing it into the loader constructor via the `callback` keyword argument, or later by setting the attribute of the same name. The callback function should expect a single argument, the template object.

For example, to properly inject the translation filter into any loaded template, you'd use code similar to this:

```
from genshi.filters import Translator
from genshi.template import TemplateLoader

def template_loaded(template):
    Translator(translations.ugettext).setup(template)

loader = TemplateLoader('templates', callback=template_loaded)
```

Template Search Path

The template loader can be configured with a list of multiple directories to search for templates. The loader maps these directories to a single logical directory for locating templates by file name.

The order of the directories making up the search path is significant: the loader will first try to locate a requested template in the first directory on the path, then in the second, and so on. If there are two templates with the same file name in multiple directories on the search path, whatever file is found first gets used.

Based on this design, an application could, for example, configure a search path consisting of a directory containing the default templates, as well as a directory where site-specific templates can be stored that will override the default templates.

Load Functions

Usually the search path consists of strings representing directory paths, but it may also contain "load functions": functions that are basically invoked with the file name, and return the template content.

Genshi comes with three builtin load functions:

`directory(path)` The equivalent of just using a string containing the directory path: looks up the file name in a specific directory.

```
from genshi.template import TemplateLoader, loader
tl = TemplateLoader([loader.directory('/path/to/dir/')])
```

That is the same as:

```
tl = TemplateLoader(['/path/to/dir/'])
```

`package(name, path)` Uses the `pkg_resources` API to locate files in Python package data (which may be inside a ZIP archive).

```
from genshi.template import TemplateLoader, loader
tl = TemplateLoader([loader.package('myapp', 'templates')])
```

This will look for templates in the `templates` directory of the Python package `myapp`.

`prefixed(**delegates)` Delegates load requests to different load functions based on the path prefix.

```
from genshi.template import TemplateLoader, loader
tl = TemplateLoader(loader.prefixed(
  core = '/tmp/dir1',
  plugin1 = loader.package('plugin1', 'templates'),
  plugin2 = loader.package('plugin2', 'templates'),
))
tmpl = tl.load('core/index.html')
```

This example sets up a loader with three delegates, under the prefixes "core", "plugin1", and "plugin2". When a template is requested, the `prefixed` load function looks for a delegate with a corresponding prefix, removes the prefix from the path and asks the delegate to load the template.

In this case, assuming the directory `/path/to/dir` contains a file named `index.html`, that file will be used when we load `core/index.html`. The other delegates are not checked as their prefix does not match.

Note: These builtin load functions are available both as class methods of the `TemplateLoader` class as well as on

the module level

Custom Load Functions You can easily use your own load function with the template loader, for example to load templates from a database. All that is needed is a callable object that accepts a `filename` (a string) and returns a tuple of the form `(filepath, filename, fileobj, uptodate_fun)`, where:

`filepath` is the absolute path to the template. This is primarily used for output in tracebacks, and does not need to map to an actual path on the file system.

`filename` is the base name of the template file

`fileobj` is a readable file-like object that provides the content of the template

`uptodate_fun` is a function that the loader can invoke to check whether the cached version of the template is still up-to-date, or `None` if the load function is not able to provide such a check. If provided, the function should not expect any parameters (so you'll definitely want to use a closure here), and should return `True` if the template has not changed since it was last loaded.

When the requested template can not be found, the function should raise an `IOError` or `TemplateNotFound` exception.

Customized Loading

If you require a completely different implementation of template loading, you can extend or even replace the builtin `TemplateLoader` class.

Protocol

The protocol between the template loader and the `Template` class is simple and only used for processing includes. The only required part of that protocol is that the object assigned to `Template.loader` implements a `load` method compatible to that of the `TemplateLoader` class, at the minimum with the signature `load(filename, relative_to=None, cls=None)`.

In addition, templates currently check for the existence and value of a boolean `auto_reload` property. If the property does not exist or evaluates to a non-truth value, inlining of included templates is disabled. Inlining is a small optimization that removes some overhead in the processing of includes.

Subclassing `TemplateLoader`

You can also adjust the behavior of the `TemplateLoader` class by subclassing it. You can of course override anything needed, but the class also provides the `_instantiate()` hook, which is intended for use by subclasses to customize the creation of the template object from the file name and content. Please consult the code and the API documentation for more detail.

1.2.6 Stream Filters

Markup Streams showed how to write filters and how they are applied to markup streams. This page describes the features of the various filters that come with Genshi itself.

HTML Form Filler

The filter `genshi.filters.html.HTMLFormFiller` can automatically populate an HTML form from values provided as a simple dictionary. When using this filter, you can basically omit any `value`, `selected`, or `checked` attributes from form controls in your templates, and let the filter do all that work for you.

`HTMLFormFiller` takes a dictionary of data to populate the form with, where the keys should match the names of form elements, and the values determine the values of those controls. For example:

```
>>> from genshi.filters import HTMLFormFiller
>>> from genshi.template import MarkupTemplate

>>> template = MarkupTemplate("""<form>
...   <p>
...     <label>User name:
...       <input type="text" name="username" />
...     </label><br />
...     <label>Password:
...       <input type="password" name="password" />
...     </label><br />
...     <label>
...       <input type="checkbox" name="remember" /> Remember me
...     </label>
...   </p>
... </form>""")
>>> filler = HTMLFormFiller(data=dict(username='john', remember=True))
>>> print(template.generate() | filler)
<form>
  <p>
    <label>User name:
      <input type="text" name="username" value="john"/>
    </label><br/>
    <label>Password:
      <input type="password" name="password"/>
    </label><br/>
    <label>
      <input type="checkbox" name="remember" checked="checked"/> Remember me
    </label>
  </p>
</form>
```

Note: This processing is done without in any way reparsing the template output. As any stream filter it operates after the template output is generated but *before* that output is actually serialized.

The filter will of course also handle radio buttons as well as `<select>` and `<textarea>` elements. For radio buttons to be marked as checked, the value in the data dictionary needs to match the `value` attribute of the `<input>` element, or evaluate to a truth value if the element has no such attribute. For options in a `<select>` box to be marked as selected, the value in the data dictionary needs to match the `value` attribute of the `<option>` element, or the text content of the option if it has no `value` attribute. Password and file input fields are not populated, as most browsers would ignore that anyway for security reasons.

You'll want to make sure that the values in the data dictionary have already been converted to strings. While the filter may be able to deal with non-string data in some cases (such as check boxes), in most cases it will either not attempt any conversion or not produce the desired results.

You can restrict the form filler to operate only on a specific `<form>` by passing either the `id` or the `name` keyword argument to the initializer. If either of those is specified, the filter will only apply to form tags with an attribute matching the specified value.

HTML Sanitizer

The filter `genshi.filters.html.HTMLSanitizer` filter can be used to clean up user-submitted HTML markup, removing potentially dangerous constructs that could be used for various kinds of abuse, such as cross-site scripting (XSS) attacks:

```
>>> from genshi.filters import HTMLSanitizer
>>> from genshi.input import HTML

>>> html = HTML("""<div>
...   <p>Innocent looking text.</p>
...   <script>alert("Danger: " + document.cookie)</script>
... </div>""")
>>> sanitize = HTMLSanitizer()
>>> print(html | sanitize)
<div>
  <p>Innocent looking text.</p>
</div>
```

In this example, the `<script>` tag was removed from the output.

You can determine which tags and attributes should be allowed by initializing the filter with corresponding sets. See the API documentation for more information.

Inline `style` attributes are forbidden by default. If you allow them, the filter will still perform sanitization on the contents any encountered inline styles: the proprietary `expression()` function (supported only by Internet Explorer) is removed, and any property using an `url()` which a potentially dangerous URL scheme (such as `javascript:`) are also stripped out:

```
>>> from genshi.filters import HTMLSanitizer
>>> from genshi.input import HTML

>>> html = HTML("""<div>
...   <br style="background: url(javascript:alert(document.cookie); color: #000" />
... </div>""")
>>> sanitize = HTMLSanitizer(safe_attrs=HTMLSanitizer.SAFE_ATTRS | set(['style']))
>>> print(html | sanitize)
<div>
  <br style="color: #000"/>
</div>
```

Warning: You should probably not rely on the `style` filtering, as sanitizing mixed HTML, CSS, and Javascript is very complicated and suspect to various browser bugs. If you can somehow get away with not allowing inline styles in user-submitted content, that would definitely be the safer route to follow.

Transformer

The filter `genshi.filters.transform.Transformer` provides a convenient way to transform or otherwise work with markup event streams. It allows you to specify which parts of the stream you're interested in with XPath expressions, and then attach a variety of transformations to the parts that match:

```
>>> from genshi.builder import tag
>>> from genshi.core import TEXT
>>> from genshi.filters import Transformer
>>> from genshi.input import HTML

>>> html = HTML('''<html>
```

```
...   <head><title>Some Title</title></head>
...   <body>
...     Some <em>body</em> text.
...   </body>
... </html>''')

>>> print(html | Transformer('body/em').map(unicode.upper, TEXT)
...                                    .unwrap().wrap(tag.u).end()
...                                    .select('body/u')
...                                    .prepend('underlined '))
<html>
  <head><title>Some Title</title></head>
  <body>
    Some <u>underlined BODY</u> text.
  </body>
</html>
```

This example sets up a transformation that:

1. matches any <*em*> element anywhere in the body,
2. uppercases any text nodes in the element,
3. strips off the <*em*> start and close tags,
4. wraps the content in a <*u*> tag, and
5. inserts the text *underlined* inside the <*u*> tag.

A number of commonly useful transformations are available for this filter. Please consult the API documentation a complete list.

In addition, you can also perform custom transformations. For example, the following defines a transformation that changes the name of a tag:

```
>>> from genshi import QName
>>> from genshi.filters.transform import ENTER, EXIT

>>> class RenameTransformation(object):
...     def __init__(self, name):
...         self.name = QName(name)
...     def __call__(self, stream):
...         for mark, (kind, data, pos) in stream:
...             if mark is ENTER:
...                 data = self.name, data[1]
...             elif mark is EXIT:
...                 data = self.name
...             yield mark, (kind, data, pos)
```

A transformation can be any callable object that accepts an augmented event stream. In this case we define a class, so that we can initialize it with the tag name.

Custom transformations can be applied using the *apply()* method of a transformer instance:

```
>>> xform = Transformer('body//em').map(unicode.upper, TEXT) \
>>> xform = xform.apply(RenameTransformation('u'))
>>> print(html | xform)
<html>
  <head><title>Some Title</title></head>
  <body>
    Some <u>BODY</u> text.
```

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

Note: The transformation filter was added in Genshi 0.5.

Translator

The `genshi.filters.i18n.Translator` filter implements basic support for internationalizing and localizing templates. When used as a filter, it translates a configurable set of text nodes and attribute values using a `gettext`-style translation function.

The `Translator` class also defines the `extract` class method, which can be used to extract localizable messages from a template.

Please refer to the API documentation for more information on this filter.

Note: The translation filter was added in Genshi 0.4.

1.2.7 Using XPath in Genshi

Genshi provides basic XPath support for matching and querying event streams.

Limitations

Due to the streaming nature of the processing model, Genshi uses only a subset of the XPath 1.0 language.

In particular, only the following axes are supported:

- `attribute`
- `child`
- `descendant`
- `descendant-or-self`
- `self`

This means you can't use the `parent`, ancestor, or sibling axes in Genshi (the `namespace` axis isn't supported either, but what you'd ever need that for I don't know). Basically, any path expression that would require buffering of the stream is not supported.

Predicates are of course supported, but path expressions *inside* predicates are restricted to attribute lookups (again due to the lack of buffering).

Most of the XPath functions and operators are supported, however they (currently) only work inside predicates. The following functions are **not** supported:

- `count()`
- `id()`
- `lang()`
- `last()`
- `position()`

- `string()`
- `sum()`

The mathematical operators (`+`, `-`, `*`, `div`, and `mod`) are not yet supported, whereas sub-expressions and the various comparison and logical operators should work as expected.

You can also use XPath variable references (`$var`) inside predicates.

Querying Streams

The `Stream` class provides a `select(path)` function that can be used to retrieve subsets of the stream:

```
>>> from genshi.input import XML

>>> doc = XML('''<doc>
...  <items count="4">
...       <item status="new">
...         <summary>Foo</summary>
...       </item>
...       <item status="closed">
...         <summary>Bar</summary>
...       </item>
...       <item status="closed" resolution="invalid">
...         <summary>Baz</summary>
...       </item>
...       <item status="closed" resolution="fixed">
...         <summary>Waz</summary>
...       </item>
...   </items>
... </doc>''')

>>> print(doc.select('items/item[@status="closed" and '
...     '(@resolution="invalid" or not(@resolution))]/summary/text()'))
BarBaz
```

Matching in Templates

See the directive `py:match` in the XML Template Language Specification.

1.2.8 Internationalization and Localization

Genshi provides comprehensive supporting infrastructure for internationalizing and localizing templates. That includes functionality for extracting localizable strings from templates, as well as a template filter and special directives that can apply translations to templates as they get rendered.

This support is based on gettext message catalogs and the gettext Python module. The extraction process can be used from the API level, or through the front-ends implemented by the Babel project, for which Genshi provides a plugin.

Basics

The simplest way to internationalize and translate templates would be to wrap all localizable strings in a `gettext()` function call (which is often aliased to `_()` for brevity). In that case, no extra template filter is required.

```
<p>${_("Hello, world!")}</p>
```

However, this approach results in significant "character noise" in templates, making them harder to read and preview.

The `genshi.filters.Translator` filter allows you to get rid of the explicit gettext function calls, so you can (often) just continue to write:

```
<p>Hello, world!</p>
```

This text will still be extracted and translated as if you had wrapped it in a `_()` call.

Note: For parameterized or pluralizable messages, you need to use the special *template directives* described below, or use the corresponding `gettext` function in embedded Python expressions.

You can control which tags should be ignored by this process; for example, it doesn't really make sense to translate the content of the HTML `<script></script>` element. Both `<script>` and `<style>` are excluded by default.

Attribute values can also be automatically translated. The default is to consider the attributes `abbr`, `alt`, `label`, `prompt`, `standby`, `summary`, and `title`, which is a list that makes sense for HTML documents. Of course, you can tell the translator to use a different set of attribute names, or none at all.

Language Tagging

You can control automatic translation in your templates using the `xml:lang` attribute. If the value of that attribute is a literal string, the contents and attributes of the element will be ignored:

```
<p xml:lang="en">Hello, world!</p>
```

On the other hand, if the value of the `xml:lang` attribute contains a Python expression, the element contents and attributes are still considered for automatic translation:

```
<html xml:lang="$locale">
  ...
</html>
```

Template Directives

Sometimes localizable strings in templates may contain dynamic parameters, or they may depend on the numeric value of some variable to choose a proper plural form. Sometimes the strings contain embedded markup, such as tags for emphasis or hyperlinks, and you don't want to rely on the people doing the translations to know the syntax and escaping rules of HTML and XML.

In those cases the simple text extraction and translation process described above is not sufficient. You could just use `gettext` API functions in embedded Python expressions for parameters and pluralization, but that does not help when messages contain embedded markup. Genshi provides special template directives for internationalization that attempt to provide a comprehensive solution for this problem space.

To enable these directives, you'll need to register them with the templates they are used in. You can do this by adding them manually via the `Template.add_directives(namespace, factory)` (where `namespace` would be "http://genshi.edgewall.org/i18n" and `factory` would be an instance of the `Translator` class). Or you can just call the `Translator.setup(template)` class method, which both registers the directives and adds the translation filter.

After the directives have been registered with the template engine on the Python side of your application, you need to declare the corresponding directive namespace in all markup templates that use them. For example:

```
<html xmlns:py="http://genshi.edgewall.org/"
      xmlns:i18n="http://genshi.edgewall.org/i18n">
  ...
</html>
```

These directives only make sense in the context of markup templates. For text templates, you can just use the corresponding `gettext` API calls as needed.

Note: The internationalization directives are still somewhat experimental and have some known issues. However, the attribute language they implement should be stable and is not subject to change substantially in future versions.

Messages

`i18n:msg` This is the basic directive for defining localizable text passages that contain parameters and/or markup.

For example, consider the following template snippet:

```
<p>
  Please visit <a href="${site.url}">${site.name}</a> for help.
</p>
```

Without further annotation, the translation filter would treat this sentence as two separate messages ("Please visit" and "for help"), and the translator would have no control over the position of the link in the sentence.

However, when you use the Genshi internationalization directives, you simply add an `i18n:msg` attribute to the enclosing `<p>` element:

```
<p i18n:msg="name">
  Please visit <a href="${site.url}">${site.name}</a> for help.
</p>
```

Genshi is then able to identify the text in the `<p>` element as a single message for translation purposes. You'll see the following string in your message catalog:

```
Please visit [1:%(name)s] for help.
```

The *<a>* element with its attribute has been replaced by a part in square brackets, which does not include the tag name or the attributes of the element.

The value of the `i18n:msg` attribute is a comma-separated list of parameter names, which serve as simplified aliases for the actual Python expressions the message contains. The order of the paramer names in the list must correspond to the order of the expressions in the text. In this example, there is only one parameter: its alias for translation is "name", while the corresponding expression is `${site.name}`.

The translator now has complete control over the structure of the sentence. He or she certainly does need to make sure that any bracketed parts are not removed, and that the `name` parameter is preserved correctly. But those are things that can be easily checked by validating the message catalogs. The important thing is that the translator can change the sentence structure, and has no way to break the application by forgetting to close a tag, for example.

So if the German translator of this snippet decided to translate it to:

```
Um Hilfe zu erhalten, besuchen Sie bitte [1:%(name)s]
```

The resulting output might be:

```
<p>
  Um Hilfe zu erhalten, besuchen Sie bitte
  <a href="http://example.com/">Example</a>
</p>
```

Messages may contain multiple tags, and they may also be nested. For example:

```
<p i18n:msg="name">
  <i>Please</i> visit <b>the site <a href="${site.url}">${site.name}</a></b>
  for help.
</p>
```

This would result in the following message ID:

```
[1:Please] visit [2:the site [3:%(name)s]] for help.
```

Again, the translator has full control over the structure of the sentence. So the German translation could actually look like this:

```
Um Hilfe zu erhalten besuchen Sie [1:bitte]
[3:%(name)s], [2:das ist eine Web-Site]
```

Which Genshi would recompose into the following outout:

```
<p>
  Um Hilfe zu erhalten besuchen Sie <i>bitte</i>
  <a href="http://example.com/">Example</a>, <b>das ist eine Web-Site</b>
</p>
```

Note how the translation has changed the order and even the nesting of the tags.

Warning: Please note that `i18n:msg` directives do not support other nested directives. Directives commonly change the structure of the generated markup dynamically, which often would result in the structure of the text changing, thus making translation as a single message ineffective.

`i18n:choose`, `i18n:singular`, `i18n:plural` Translatable strings that vary based on some number of objects, such as "You have 1 new message" or "You have 3 new messages", present their own challenge, in particular when you consider that different languages have different rules for pluralization. For example, while English and most western languages have two plural forms (one for `n=1` and an other for `n<>1`), Welsh has five different plural forms, while Hungarian only has one.

The `gettext` framework has long supported this via the `ngettext()` family of functions. You specify two default messages, one singular and one plural, and the number of items. The translations however may contain any number of plural forms for the message, depending on how many are commonly used in the language. `ngettext` will choose the correct plural form of the translated message based on the specified number of items.

Genshi provides a variant of the `i18n:msg` directive described above that allows choosing the proper plural form based on the numeric value of a given variable. The pluralization support is implemented in a set of three directives that must be used together: `i18n:choose`, `i18n:singular`, and `i18n:plural`.

The `i18n:choose` directive is used to set up the context of the message: it simply wraps the singular and plural variants.

The value of this directive is split into two parts: the first is the *numeral*, a Python expression that evaluates to a number to determine which plural form should be chosen. The second part, separated by a semicolon, lists the parameter names. This part is equivalent to the value of the `i18n:msg` directive.

For example:

```
<p i18n:choose="len(messages); num">
  <i18n:singular>You have <b>${len(messages)}</b> new message.</i18n:singular>
  <i18n:plural>You have <b>${len(messages)}</b> new messages.</i18n:plural>
</p>
```

All three directives can be used either as elements or attribute. So the above example could also be written as follows:

```
<i18n:choose numeral="len(messages)" params="num">
  <p i18n:singular="">You have <b>${len(messages)}</b> new message.</p>
  <p i18n:plural="">You have <b>${len(messages)}</b> new messages.</p>
</i18n:choose>
```

When used as an element, the two parts of the `i18n:choose` value are split into two different attributes: `numeral` and `params`. The `i18n:singular` and `i18n:plural` directives do not require or support any value (or any extra attributes).

Comments and Domains

`i18n:comment` The `i18n:comment` directive can be used to supply a comment for the translator. For example, if a template snippet is not easily understood outside of its context, you can add a translator comment to help the translator understand in what context the message will be used:

```
<p i18n:msg="name" i18n:comment="Link to the relevant support site">
  Please visit <a href="${site.url}">${site.name}</a> for help.
</p>
```

This comment will be extracted together with the message itself, and will commonly be placed along the message in the message catalog, so that it is easily visible to the person doing the translation.

This directive has no impact on how the template is rendered, and is ignored outside of the extraction process.

`i18n:domain` In larger projects, message catalogs are commonly split up into different *domains*. For example, you might have a core application domain, and then separate domains for extensions or libraries.

Genshi provides a directive called `i18n:domain` that lets you choose the translation domain for a particular scope. For example:

```
<div i18n:domain="examples">
  <p>Hello, world!</p>
</div>
```

Extraction

The `Translator` class provides a class method called `extract`, which is a generator yielding all localizable strings found in a template or markup stream. This includes both literal strings in text nodes and attribute values, as well as strings in `gettext()` calls in embedded Python code. See the API documentation for details on how to use this method directly.

Babel Integration

This functionality is integrated with the message extraction framework provided by the Babel project. Babel provides a command-line interface as well as commands that can be used from `setup.py` scripts using Setuptools or Distutils.

The first thing you need to do to make Babel extract messages from Genshi templates is to let Babel know which files are Genshi templates. This is done using a "mapping configuration", which can be stored in a configuration file, or specified directly in your `setup.py`.

In a configuration file, the mapping may look like this:

```
# Python souce
[python:**.py]

# Genshi templates
[genshi:**/templates/**.html]
include_attrs = title

[genshi:**/templates/**.txt]
template_class = genshi.template.TextTemplate
encoding = latin-1
```

Please consult the Babel documentation for details on configuration.

If all goes well, running the extraction with Babel should create a POT file containing the strings from your Genshi templates and your Python source files.

Configuration Options

The Genshi extraction plugin for Babel supports the following options:

`template_class` The concrete `Template` class that the file should be loaded with. Specify the package/module name and the class name, separated by a colon.

The default is to use `genshi.template:MarkupTemplate`, and you'll want to set it to `genshi.template:TextTemplate` for text templates.

`encoding` The encoding of the template file. This is only used for text templates. The default is to assume "utf-8".

`include_attrs` Comma-separated list of attribute names that should be considered to have localizable values. Only used for markup templates.

`ignore_tags` Comma-separated list of tag names that should be ignored. Only used for markup templates.

`extract_text` Whether text outside explicit `gettext` function calls should be extracted. By default, any text nodes not inside ignored tags, and values of attribute in the `include_attrs` list are extracted. If this option is disabled, only strings in `gettext` function calls are extracted.

Note: If you disable this option, and do not make use of the internationalization directives, it's not necessary to add the translation filter as described above. You only need to make sure that the template has access to the `gettext` functions it uses.

Translation

If you have prepared MO files for use with Genshi using the appropriate tools, you can access the message catalogs with the gettext Python module. You'll probably want to create a `gettext.GNUTranslations` instance, and make the translation functions it provides available to your templates by putting them in the template context.

The `Translator` filter needs to be added to the filters of the template (applying it as a stream filter will likely not have the desired effect). Furthermore it needs to be the first filter in the list, including the internal filters that Genshi adds itself:

```
from genshi.filters import Translator
from genshi.template import MarkupTemplate

template = MarkupTemplate("...")
template.filters.insert(0, Translator(translations.ugettext))
```

The `Translator` class also provides the convenience method `setup()`, which will both add the filter and register the i18n directives:

```
from genshi.filters import Translator
from genshi.template import MarkupTemplate

template = MarkupTemplate("...")
translator = Translator(translations.ugettext)
translator.setup(template)
```

Warning: If you're using `TemplateLoader`, you should specify a callback function in which you add the filter. That ensures that the filter is not added everytime the template is rendered, thereby being applied multiple times.

Related Considerations

If you intend to produce an application that is fully prepared for an international audience, there are a couple of other things to keep in mind:

Unicode

Use `unicode` internally, not encoded bytestrings. Only encode/decode where data enters or exits the system. This means that your code works with characters and not just with bytes, which is an important distinction for example when calculating the length of a piece of text. When you need to decode/encode, it's probably a good idea to use UTF-8.

Date and Time

If your application uses datetime information that should be displayed to users in different timezones, you should try to work with UTC (universal time) internally. Do the conversion from and to "local time" when the data enters or exits the system. Make use the Python datetime module and the third-party pytz package.

Formatting and Locale Data

Make sure you check out the functionality provided by the Babel project for things like number and date formatting, locale display strings, etc.

1.2.9 Using the Templating Plugin

While you can easily use Genshi templating through the APIs provided directly by Genshi, in some situations you may want to use Genshi through the template engine plugin API. Note though that this considerably limits the power and flexibility of Genshi templates (for example, there's no good way to use filters such as Genshi's *HTML Form Filler* when the plugin API is sitting between your code and Genshi).

Introduction

Some Python web frameworks support a variety of different templating engines through the Template Engine Plugin API, which was first developed by the Buffet and TurboGears projects.

Genshi supports this API out of the box, so you can use it in frameworks like TurboGears or Pylons without installing any additional packages. A small example TurboGears application is included in the `examples` directory of source distributions of Genshi.

Usage

The way you use Genshi through the plugin API depends very much on the framework you're using. In general, the approach will look something like the following:

1. Configure Genshi as the default (or an additional) template engine
2. Optionally specify Genshi-specific *configuration options*
3. For any given *view* or *controller* (or whatever those are called in your framework of choice), specify the name of the template to use and which data should be made available to it.

For point 1, you'll have to specify the *name* of the template engine plugin. For Genshi, this is **"genshi"**. However, because Genshi supports both markup and text templates, it also provides two separate plugins, namely **"genshi-markup"** and **"genshi-text"** (the "genshi" name is just an alias for "genshi-markup").

Usually, you can choose a default template engine, but also use a different engine on a per-request basis. So to use markup templates in general, but a text template in a specific controller, you'd configure "genshi" as the default template engine, and specify "genshi-text" for the controllers that should use text templates. How exactly this works depends on the framework in use.

When rendering a specific template in a controller (point 3 above), you may also be able to pass additional options to the plugin. This includes the `format` keyword argument, which Genshi will use to override the configured default serialization method. In combination with specifying the "genshi-text" engine name as explained above, you would use this to specify the "text" serialization method when you want to use a text template. Or you'd specify "xml" for the format when you want to produce an Atom feed or other XML content.

Template Paths

How you specify template paths depends on whether you have a *search path* set up or not. The search path is a list of directories that Genshi should load templates from. Now when you request a template using a relative path such as `mytmpl.html` or `foo/mytmpl.html`, Genshi will look for that file in the directories on the search path.

For mostly historical reasons, the Genshi template engine plugin uses a different approach when you **haven't** configured the template search path: you now load templates using *dotted notation*, for example `mytmpl` or `foo.mytmpl`. Note how you've lost the ability to explicitly specify the file extension: you now have to use `.html` for markup templates, and `.txt` for text templates.

Using the search path is recommended for a number of reasons: First, it's the native Genshi model and is thus more robust and better supported. Second, a search path gives you much more flexibility for organizing your application templates. And as noted above, you aren't forced to use hardcoded filename extensions for your template files.

Extra Implicit Objects

The "genshi-markup" template engine plugin adds some extra functions that are made available to all templates implicitly, namely:

HTML(string) Parses the given string as HTML and returns a markup stream.

XML(string) Parses the given string as XML and returns a markup stream.

ET(tree) Adapts the given ElementTree object to a markup stream.

The framework may make additional objects available by default. Consult the documentation of your framework for more information.

Configuration Options

The plugin API allows plugins to be configured using a dictionary of strings. The following is a list of configuration options that Genshi supports. These may or may not be made available by your framework. TurboGears 1.0, for example, only passes a fixed set of options to all plugins.

genshi.allow_exec

Whether the Python code blocks should be permitted in templates. Specify "yes" to allow code blocks (which is the default), or "no" otherwise. Please note that disallowing code blocks in templates does not turn Genshi into a sandboxable template engine; there are sufficient ways to do harm even using plain expressions.

genshi.auto_reload

Whether the template loader should check the last modification time of template files, and automatically reload them if they have been changed. Specify "yes" to enable this reloading (which is the default), or "no" to turn it off.

You probably want to disable reloading in a production environment to improve performance of both templating loading and the processing of includes. But remember that you'll then have to manually restart the server process anytime the templates are updated.

genshi.default_doctype

The default `DOCTYPE` declaration to use in generated markup. Valid values are:

html-strict (or just html) HTML 4.01 Strict

html-transitional HTML 4.01 Transitional

xhtml-strict (or just xhtml) XHTML 1.0 Strict

xhtml-transitional XHTML 1.0 Transitional

html5 HTML5 (as proposed by the WHAT-WG)

Note: While using the Genshi API directly allows you to specify document types not in that list, the *dictionary-of-strings* based configuration utilized by the plugin API unfortunately limits your choices to those listed above.

The default behavior is to not do any prepending/replacing of a `DOCTYPE`, but rather pass through those defined in the templates (if any). If this option is set, however, any `DOCTYPE` declarations in the templates are replaced by the specified document type.

Note that with (X)HTML, the presence and choice of the `DOCTYPE` can have a more or less dramatic impact on how modern browsers render pages that use CSS style sheets. In particular, browsers may switch to *quirks rendering mode* for certain document types, or when the `DOCTYPE` declaration is missing completely.

For more information on the choice of the appropriate `DOCTYPE`, see:

- Recommended DTDs to use in your Web document
- Choosing a DOCTYPE

genshi.default_encoding

The default output encoding to use when serializing a template. By default, Genshi uses UTF-8. If you need to, you can choose a different charset by specifying this option, although that rarely makes sense.

As Genshi is not in control over what HTTP headers are being sent together with the template output, make sure that you (or the framework you're using) specify the chosen encoding as part of the outgoing `Content-Type` header. For example:

```
Content-Type: text/html; charset=utf-8
```

Note: Browsers commonly use ISO-8859-1 by default for `text/html`, so even if you use Genshi's default UTF-8 encoding, you'll have to let the browser know about that explicitly

genshi.default_format

Determines the default serialization method to use. Valid options are:

xml Serialization to XML

xhtml Serialization to XHTML in a way that should be compatible with HTML (i.e. the result can be sent using the `text/html` MIME type, but can also be handled by XML parsers if you're careful).

html Serialization to HTML

text Plain text serialization

See Understanding HTML, XML and XHTML for an excellent description of the subtle differences between the three different markup serialization options. As a general recommendation, if you don't have a special requirement to produce well-formed XML, you should probably use the **html** option for your web sites.

genshi.loader_callback

The callback function that should be invoked whenever the template loader loads a new template.

Note: Unlike the other options, this option can **not** be passed as a string value, but rather must be a reference to the actual function. That effectively means it can not be set from (non-Python) configuration files.

genshi.lookup_errors

The error handling style to use in template expressions. Can be either **lenient** (the default) or **strict**. See the Error Handling section for detailled information on the differences between these two modes.

genshi.max_cache_size

The maximum number of templates that the template loader will cache in memory. The default value is **25**. You may want to choose a higher value if your web site uses a larger number of templates, and you have enough memory to spare.

genshi.new_text_syntax

Whether the new syntax for text templates should be used. Specify "yes" to enable the new syntax, or "no" to use the old syntax.

In the version of Genshi, the default is to use the old syntax for backwards-compatibility, but that will change in a future release.

genshi.search_path

A colon-separated list of file-system path names that the template loader should use to search for templates.

1.3 API Documentation

1.3.1 genshi.builder

Support for programmatically generating markup streams from Python code using a very simple syntax. The main entry point to this module is the *tag* object (which is actually an instance of the `ElementFactory` class). You should rarely (if ever) need to directly import and use any of the other classes in this module.

Elements can be created using the *tag* object using attribute access. For example:

```
>>> doc = tag.p('Some text and ', tag.a('a link', href='http://example.org/'), '.')
>>> doc
<Element "p">
```

This produces an *Element* instance which can be further modified to add child nodes and attributes. This is done by "calling" the element: positional arguments are added as child nodes (alternatively, the *Element.append* method can be used for that purpose), whereas keywords arguments are added as attributes:

```
>>> doc(tag.br)
<Element "p">
>>> print(doc)
<p>Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

If an attribute name collides with a Python keyword, simply append an underscore to the name:

```
>>> doc(class_='intro')
<Element "p">
>>> print(doc)
<p class="intro">Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

As shown above, an *Element* can easily be directly rendered to XML text by printing it or using the Python `str()` function. This is basically a shortcut for converting the *Element* to a stream and serializing that stream:

```
>>> stream = doc.generate()
>>> stream
<genshi.core.Stream object at ...>
>>> print(stream)
<p class="intro">Some text and <a href="http://example.org/">a link</a>.<br/></p>
```

The *tag* object also allows creating "fragments", which are basically lists of nodes (elements or text) that don't have a parent element. This can be useful for creating snippets of markup that are attached to a parent element later (for example in a template). Fragments are created by calling the *tag* object, which returns an object of type *Fragment*:

```
>>> fragment = tag('Hello, ', tag.em('world'), '!')
>>> fragment
<Fragment>
>>> print(fragment)
Hello, <em>world</em>!
```

class genshi.builder.**Fragment**

Represents a markup fragment, which is basically just a list of element or text nodes.

append(*node*)

Append an element or string as child node.

Parameters **node** – the node to append; can be an *Element*, *Fragment*, or a *Stream*, or a Python string or number

generate()

Return a markup event stream for the fragment.

Return type *Stream*

class genshi.builder.**Element**(*tag_*, ***attrib*)

Simple XML output generator based on the builder pattern.

Construct XML elements by passing the tag name to the constructor:

```
>>> print(Element('strong'))
<strong/>
```

Attributes can be specified using keyword arguments. The values of the arguments will be converted to strings and any special XML characters escaped:

```
>>> print(Element('textarea', rows=10, cols=60))
<textarea rows="10" cols="60"/>
>>> print(Element('span', title='1 < 2'))
<span title="1 &lt; 2"/>
>>> print(Element('span', title='"baz"'))
<span title="&#34;baz&#34;"/>
```

The " character is escaped using a numerical entity. The order in which attributes are rendered is undefined.

If an attribute value evaluates to *None*, that attribute is not included in the output:

```
>>> print(Element('a', name=None))
<a/>
```

Attribute names that conflict with Python keywords can be specified by appending an underscore:

```
>>> print(Element('div', class_='warning'))
<div class="warning"/>
```

Nested elements can be added to an element using item access notation. The call notation can also be used for this and for adding attributes using keyword arguments, as one would do in the constructor.

```
>>> print(Element('ul')(Element('li'), Element('li')))
<ul><li/><li/></ul>
>>> print(Element('a')('Label'))
<a>Label</a>
>>> print(Element('a')('Label', href="target"))
<a href="target">Label</a>
```

Text nodes can be nested in an element by adding strings instead of elements. Any special characters in the strings are escaped automatically:

```
>>> print(Element('em')('Hello world'))
<em>Hello world</em>
>>> print(Element('em')(42))
<em>42</em>
>>> print(Element('em')('1 < 2'))
<em>1 &lt; 2</em>
```

This technique also allows mixed content:

```
>>> print(Element('p')('Hello ', Element('b')('world')))
<p>Hello <b>world</b></p>
```

Quotes are not escaped inside text nodes: >>> print(Element('p')("'Hello"')) "Hello"

Elements can also be combined with other elements or strings using the addition operator, which results in a *Fragment* object that contains the operands:

```
>>> print(Element('br') + 'some text' + Element('br'))
<br/>some text<br/>
```

Elements with a namespace can be generated using the *Namespace* and/or *QName* classes:

```
>>> from genshi.core import Namespace
>>> xhtml = Namespace('http://www.w3.org/1999/xhtml')
>>> print(Element(xhtml.html, lang='en'))
<html xmlns="http://www.w3.org/1999/xhtml" lang="en"/>
```

generate()
Return a markup event stream for the fragment.

Return type *Stream*

class genshi.builder.**ElementFactory**(*namespace=None*)
Factory for *Element* objects.

A new element is created simply by accessing a correspondingly named attribute of the factory object:

```
>>> factory = ElementFactory()
>>> print(factory.foo)
<foo/>
>>> print(factory.foo(id=2))
<foo id="2"/>
```

Markup fragments (lists of nodes without a parent element) can be created by calling the factory:

```
>>> print(factory('Hello, ', factory.em('world'), '!'))
Hello, <em>world</em>!
```

A factory can also be bound to a specific namespace:

```
>>> factory = ElementFactory('http://www.w3.org/1999/xhtml')
>>> print(factory.html(lang="en"))
<html xmlns="http://www.w3.org/1999/xhtml" lang="en"/>
```

The namespace for a specific element can be altered on an existing factory by specifying the new namespace using item access:

```
>>> factory = ElementFactory()
>>> print(factory.html(factory['http://www.w3.org/2000/svg'].g(id=3)))
<html><g xmlns="http://www.w3.org/2000/svg" id="3"/></html>
```

Usually, the *ElementFactory* class is not be used directly. Rather, the *tag* instance should be used to create elements.

1.3.2 genshi.core

Core classes for markup processing.

class `genshi.core.`**`Stream`**(*events*, *serializer=None*)
Represents a stream of markup events.

This class is basically an iterator over the events.

Stream events are tuples of the form:

```
(kind, data, position)
```

where `kind` is the event kind (such as *START*, *END*, *TEXT*, etc), `data` depends on the kind of event, and `position` is a `(filename, line, offset)` tuple that contains the location of the original element or text in the input. If the original location is unknown, `position` is `(None, -1, -1)`.

Also provided are ways to serialize the stream to text. The *serialize()* method will return an iterator over generated strings, while *render()* returns the complete generated text at once. Both accept various parameters that impact the way the stream is serialized.

`filter`(**filters*)
Apply filters to the stream.

This method returns a new stream with the given filters applied. The filters must be callables that accept the stream object as parameter, and return the filtered stream.

The call:

```
stream.filter(filter1, filter2)
```

is equivalent to:

```
stream | filter1 | filter2
```

Parameters **`filters`** – one or more callable objects that should be applied as filters

Returns the filtered stream

Return type *Stream*

`render`(*method=None*, *encoding=None*, *out=None*, ***kwargs*)
Return a string representation of the stream.

Any additional keyword arguments are passed to the serializer, and thus depend on the *method* parameter value.

Parameters

- **`method`** – determines how the stream is serialized; can be either "xml", "xhtml", "html", "text", or a custom serializer class; if *None*, the default serialization method of the stream is used
- **`encoding`** – how the output string should be encoded; if set to *None*, this method returns a *unicode* object

- **out** – a file-like object that the output should be written to instead of being returned as one big string; note that if this is a file or socket (or similar), the *encoding* must not be *None* (that is, the output must be encoded)

Returns a *str* or *unicode* object (depending on the *encoding* parameter), or *None* if the *out* parameter is provided

Return type *basestring*

See XMLSerializer, XHTMLSerializer, HTMLSerializer, TextSerializer

Note Changed in 0.5: added the *out* parameter

select(*path*, *namespaces=None*, *variables=None*)

Return a new stream that contains the events matching the given XPath expression.

```
>>> from genshi import HTML
>>> stream = HTML('<doc><elem>foo</elem><elem>bar</elem></doc>', encoding='utf-8')
>>> print(stream.select('elem'))
<elem>foo</elem><elem>bar</elem>
>>> print(stream.select('elem/text()'))
foobar
```

Note that the outermost element of the stream becomes the *context node* for the XPath test. That means that the expression "doc" would not match anything in the example above, because it only tests against child elements of the outermost element:

```
>>> print(stream.select('doc'))
```

You can use the "." expression to match the context node itself (although that usually makes little sense):

```
>>> print(stream.select('.'))
<doc><elem>foo</elem><elem>bar</elem></doc>
```

Parameters

- **path** – a string containing the XPath expression
- **namespaces** – mapping of namespace prefixes used in the path
- **variables** – mapping of variable names to values

Returns the selected substream

Return type *Stream*

Raises PathSyntaxError if the given path expression is invalid or not supported

serialize(*method='xml'*, ***kwargs*)

Generate strings corresponding to a specific serialization of the stream.

Unlike the *render()* method, this method is a generator that returns the serialized output incrementally, as opposed to returning a single string.

Any additional keyword arguments are passed to the serializer, and thus depend on the *method* parameter value.

Parameters **method** – determines how the stream is serialized; can be either "xml", "xhtml", "html", "text", or a custom serializer class; if *None*, the default serialization method of the stream is used

Returns an iterator over the serialization results (*Markup* or *unicode* objects, depending on the serialization method)

Return type iterator

See XMLSerializer, XHTMLSerializer, HTMLSerializer, TextSerializer

class genshi.core.**Markup**

Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped.

classmethod escape(*text*, *quotes=True*)

Create a Markup instance from a string and escape special characters it may contain (<, >, & and ").

```
>>> escape('"1 < 2"')
<Markup u'&#34;1 &lt; 2&#34;'>
```

If the *quotes* parameter is set to *False*, the " character is left as is. Escaping quotes is generally only required for strings that are to be used in attribute values.

```
>>> escape('"1 < 2"', quotes=False)
<Markup u'"1 &lt; 2"'>
```

Parameters

- **text** – the text to escape
- **quotes** – if True, double quote characters are escaped in addition to the other special characters

Returns the escaped *Markup* string

Return type *Markup*

join(*seq*, *escape_quotes=True*)

Return a *Markup* object which is the concatenation of the strings in the given sequence, where this *Markup* object is the separator between the joined elements.

Any element in the sequence that is not a *Markup* instance is automatically escaped.

Parameters

- **seq** – the sequence of strings to join
- **escape_quotes** – whether double quote characters in the elements should be escaped

Returns the joined *Markup* object

Return type *Markup*

See *escape*

stripentities(*keepxmlentities=False*)

Return a copy of the text with any character or numeric entities replaced by the equivalent UTF-8 characters.

If the *keepxmlentities* parameter is provided and evaluates to *True*, the core XML entities (&, ', >, < and ") are not stripped.

Returns a *Markup* instance with entities removed

Return type *Markup*

See *genshi.util.stripentities*

striptags()

Return a copy of the text with all XML/HTML tags removed.

Returns a *Markup* instance with all tags removed

Return type *Markup*

See *genshi.util.striptags*

unescape()
Reverse-escapes &, <, >, and " and returns a *unicode* object.

```
>>> Markup('1 &lt; 2').unescape()
u'1 < 2'
```

Returns the unescaped string

Return type *unicode*

See *genshi.core.unescape*

genshi.core.**unescape**(*text*)
Reverse-escapes &, <, >, and " and returns a *unicode* object.

```
>>> unescape(Markup('1 &lt; 2'))
u'1 < 2'
```

If the provided *text* object is not a *Markup* instance, it is returned unchanged.

```
>>> unescape('1 &lt; 2')
'1 &lt; 2'
```

Parameters **text** – the text to unescape

Returns the unescsaped string

Return type *unicode*

class genshi.core.**Attrs**
Immutable sequence type that stores the attributes of an element.

Ordering of the attributes is preserved, while access by name is also supported.

```
>>> attrs = Attrs([('href', '#'), ('title', 'Foo')])
>>> attrs
Attrs([('href', '#'), ('title', 'Foo')])
```

```
>>> 'href' in attrs
True
>>> 'tabindex' in attrs
False
>>> attrs.get('title')
'Foo'
```

Instances may not be manipulated directly. Instead, the operators | and − can be used to produce new instances that have specific attributes added, replaced or removed.

To remove an attribute, use the − operator. The right hand side can be either a string or a set/sequence of strings, identifying the name(s) of the attribute(s) to remove:

```
>>> attrs - 'title'
Attrs([('href', '#')])
>>> attrs - ('title', 'href')
Attrs()
```

The original instance is not modified, but the operator can of course be used with an assignment:

```
>>> attrs
Attrs([('href', '#'), ('title', 'Foo')])
>>> attrs -= 'title'
>>> attrs
Attrs([('href', '#')])
```

To add a new attribute, use the `|` operator, where the right hand value is a sequence of `(name, value)` tuples (which includes *Attrs* instances):

```
>>> attrs | [('title', 'Bar')]
Attrs([('href', '#'), ('title', 'Bar')])
```

If the attributes already contain an attribute with a given name, the value of that attribute is replaced:

```
>>> attrs | [('href', 'http://example.org/')]
Attrs([('href', 'http://example.org/')])
```

get(*name*, *default=None*)
Return the value of the attribute with the specified name, or the value of the *default* parameter if no such attribute is found.

Parameters

- **name** – the name of the attribute
- **default** – the value to return when the attribute does not exist

Returns the attribute value, or the *default* value if that attribute does not exist

Return type *object*

totuple()
Return the attributes as a markup event.

The returned event is a *TEXT* event, the data is the value of all attributes joined together.

```
>>> Attrs([('href', '#'), ('title', 'Foo')]).totuple()
('TEXT', '#Foo', (None, -1, -1))
```

Returns a *TEXT* event

Return type *tuple*

class genshi.core.**Namespace**(*uri*)
Utility class creating and testing elements with a namespace.

Internally, namespace URIs are encoded in the *QName* of any element or attribute, the namespace URI being enclosed in curly braces. This class helps create and test these strings.

A *Namespace* object is instantiated with the namespace URI.

```
>>> html = Namespace('http://www.w3.org/1999/xhtml')
>>> html
Namespace('http://www.w3.org/1999/xhtml')
>>> html.uri
u'http://www.w3.org/1999/xhtml'
```

The *Namespace* object can than be used to generate *QName* objects with that namespace:

```
>>> html.body
QName('http://www.w3.org/1999/xhtml}body')
>>> html.body.localname
```

```
u'body'
>>> html.body.namespace
u'http://www.w3.org/1999/xhtml'
```

The same works using item access notation, which is useful for element or attribute names that are not valid Python identifiers:

```
>>> html['body']
QName('http://www.w3.org/1999/xhtml}body')
```

A *Namespace* object can also be used to test whether a specific *QName* belongs to that namespace using the `in` operator:

```
>>> qname = html.body
>>> qname in html
True
>>> qname in Namespace('http://www.w3.org/2002/06/xhtml2')
False
```

class genshi.core.**QName**

A qualified element or attribute name.

The unicode value of instances of this class contains the qualified name of the element or attribute, in the form `{namespace-uri}local-name`. The namespace URI can be obtained through the additional *namespace* attribute, while the local name can be accessed through the *localname* attribute.

```
>>> qname = QName('foo')
>>> qname
QName('foo')
>>> qname.localname
u'foo'
>>> qname.namespace
```

```
>>> qname = QName('http://www.w3.org/1999/xhtml}body')
>>> qname
QName('http://www.w3.org/1999/xhtml}body')
>>> qname.localname
u'body'
>>> qname.namespace
u'http://www.w3.org/1999/xhtml'
```

1.3.3 genshi.filters

Implementation of a number of stream filters.

genshi.filters.html

Implementation of a number of stream filters.

class genshi.filters.html.**HTMLFormFiller**(*name=None*, *id=None*, *data=None*, *passwords=False*)

A stream filter that can populate HTML forms from a dictionary of values.

```
>>> from genshi.input import HTML
>>> html = HTML('''<form>
...   <p><input type="text" name="foo" /></p>
... </form>''', encoding='utf-8')
```

```
>>> filler = HTMLFormFiller(data={'foo': 'bar'})
>>> print(html | filler)
<form>
  <p><input type="text" name="foo" value="bar"/></p>
</form>
```

class genshi.filters.html.**HTMLSanitizer**(*safe_tags=frozenset(['em', 'pre', 'code', 'p', 'h2', 'h3', 'h1', 'h6', 'h4', 'h5', 'table', 'font', 'u', 'select', 'kbd', 'strong', 'span', 'sub', 'img', 'area', 'menu', 'tt', 'tr', 'tbody', 'label', 'hr', 'dfn', 'tfoot', 'th', 'sup', 'strike', 'input', 'td', 'samp', 'cite', 'thead', 'map', 'dl', 'blockquote', 'fieldset', 'option', 'form', 'acronym', 'big', 'dd', 'var', 'ol', 'abbr', 'br', 'address', 'optgroup', 'li', 'dt', 'ins', 'legend', 'a', 'b', 'center', 'textarea', 'colgroup', 'i', 'button', 'q', 'caption', 's', 'del', 'small', 'div', 'col', 'dir', 'ul']), safe_attrs=frozenset(['rev', 'prompt', 'color', 'colspan', 'accesskey', 'usemap', 'cols', 'accept', 'datetime', 'char', 'accept-charset', 'shape', 'href', 'hreflang', 'selected', 'frame', 'type', 'alt', 'nowrap', 'border', 'id', 'axis', 'compact', 'rows', 'checked', 'for', 'start', 'hspace', 'charset', 'ismap', 'label', 'target', 'bgcolor', 'readonly', 'rel', 'valign', 'scope', 'size', 'cellspacing', 'cite', 'media', 'multiple', 'src', 'rules', 'nohref', 'action', 'rowspan', 'abbr', 'span', 'method', 'height', 'class', 'enctype', 'lang', 'disabled', 'name', 'charoff', 'clear', 'summary', 'value', 'longdesc', 'headers', 'vspace', 'noshade', 'coords', 'width', 'maxlength', 'cellpadding', 'title', 'align', 'dir', 'tabindex']), safe_schemes=frozenset(['mailto', 'ftp', 'http', 'file', 'https', None]), uri_attrs=frozenset(['src', 'lowsrc', 'href', 'dynsrc', 'background', 'action']))*

A filter that removes potentially dangerous HTML tags and attributes from the stream.

```
>>> from genshi import HTML
>>> html = HTML('<div><script>alert(document.cookie)</script></div>', encoding='utf-8')
>>> print(html | HTMLSanitizer())
<div/>
```

The default set of safe tags and attributes can be modified when the filter is instantiated. For example, to allow inline `style` attributes, the following instantation would work:

```
>>> html = HTML('<div style="background: #000"></div>', encoding='utf-8')
>>> sanitizer = HTMLSanitizer(safe_attrs=HTMLSanitizer.SAFE_ATTRS | set(['style']))
>>> print(html | sanitizer)
<div style="background: #000"/>
```

Note that even in this case, the filter *does* attempt to remove dangerous constructs from style attributes:

```
>>> html = HTML('<div style="background: url(javascript:void); color: #000"></div>', encoding='u
>>> print(html | sanitizer)
<div style="color: #000"/>
```

This handles HTML entities, unicode escapes in CSS and Javascript text, as well as a lot of other things. However, the style tag is still excluded by default because it is very hard for such sanitizing to be completely safe, especially considering how much error recovery current web browsers perform.

It also does some basic filtering of CSS properties that may be used for typical phishing attacks. For more sophisticated filtering, this class provides a couple of hooks that can be overridden in sub-classes.

Warn Note that this special processing of CSS is currently only applied to style attributes, **not** style elements.

is_safe_css(*propname*, *value*)

Determine whether the given css property declaration is to be considered safe for inclusion in the output.

Parameters

- **propname** – the CSS property name
- **value** – the value of the property

Returns whether the property value should be considered safe

Return type bool

Since version 0.6

is_safe_elem(*tag*, *attrs*)

Determine whether the given element should be considered safe for inclusion in the output.

Parameters

- **tag** (QName) – the tag name of the element
- **attrs** (Attrs) – the element attributes

Returns whether the element should be considered safe

Return type bool

Since version 0.6

is_safe_uri(*uri*)

Determine whether the given URI is to be considered safe for inclusion in the output.

The default implementation checks whether the scheme of the URI is in the set of allowed URIs (*safe_schemes*).

```
>>> sanitizer = HTMLSanitizer()
>>> sanitizer.is_safe_uri('http://example.org/')
True
>>> sanitizer.is_safe_uri('javascript:alert(document.cookie)')
False
```

Parameters **uri** – the URI to check

Returns *True* if the URI can be considered safe, *False* otherwise

Return type *bool*

Since version 0.4.3

sanitize_css(*text*)

Remove potentially dangerous property declarations from CSS code.

In particular, properties using the CSS `url()` function with a scheme that is not considered safe are removed:

```
>>> sanitizer = HTMLSanitizer()
>>> sanitizer.sanitize_css(u'''
...   background: url(javascript:alert("foo"));
```

```
...   color: #000;
... ''')
[u'color: #000']
```

Also, the proprietary Internet Explorer function `expression()` is always stripped:

```
>>> sanitizer.sanitize_css(u'''
...   background: #fff;
...   color: #000;
...   width: e/**/xpression(alert("foo"));
... ''')
[u'background: #fff', u'color: #000']
```

Parameters **text** – the CSS text; this is expected to be *unicode* and to not contain any character or numeric references

Returns a list of declarations that are considered safe

Return type *list*

Since version 0.4.3

genshi.filters.i18n

Directives and utilities for internationalization and localization of templates.

since version 0.4

note Directives support added since version 0.6

class genshi.filters.i18n.**Translator**(*translate=<gettext.NullTranslations instance>*, *ignore_tags=frozenset([QName('http://www.w3.org/1999/xhtml}style'), QName('http://www.w3.org/1999/xhtml}script'), QName('style'), QName('script')])*, *include_attrs=frozenset(['prompt', 'title', 'standby', 'summary', 'abbr', 'alt', 'label'])*, *extract_text=True*)

Can extract and translate localizable strings from markup streams and templates.

For example, assume the following template:

```
>>> tmpl = MarkupTemplate('''<html xmlns:py="http://genshi.edgewall.org/">
...   <head>
...     <title>Example</title>
...   </head>
...   <body>
...     <h1>Example</h1>
...     <p>${_("Hello, %(name)s") % dict(name=username)}</p>
...   </body>
... </html>''', filename='example.html')
```

For demonstration, we define a dummy `gettext`-style function with a hard-coded translation table, and pass that to the *Translator* initializer:

```
>>> def pseudo_gettext(string):
...     return {
...         'Example': 'Beispiel',
...         'Hello, %(name)s': 'Hallo, %(name)s'
...     }[string]
>>> translator = Translator(pseudo_gettext)
```

Next, the translator needs to be prepended to any already defined filters on the template:

```
>>> tmpl.filters.insert(0, translator)
```

When generating the template output, our hard-coded translations should be applied as expected:

```
>>> print(tmpl.generate(username='Hans', _=pseudo_gettext))
<html>
  <head>
    <title>Beispiel</title>
  </head>
  <body>
    <h1>Beispiel</h1>
    <p>Hallo, Hans</p>
  </body>
</html>
```

Note that elements defining `xml:lang` attributes that do not contain variable expressions are ignored by this filter. That can be used to exclude specific parts of a template from being extracted and translated.

extract(*stream*, *gettext_functions=('_', 'gettext', 'ngettext', 'dgettext', 'dngettext', 'ugettext', 'ungettext')*, *search_text=True*, *comment_stack=None*)

Extract localizable strings from the given template stream.

For every string found, this function yields a `(lineno, function, message, comments)` tuple, where:

- `lineno` is the number of the line on which the string was found,
- `function` is the name of the `gettext` function used (if the string was extracted from embedded Python code), and
- `message` is the string itself (a `unicode` object, or a tuple of `unicode` objects for functions with multiple string arguments).
- `comments` is a list of comments related to the message, extracted from `i18n:comment` attributes found in the markup

```
>>> tmpl = MarkupTemplate('''<html xmlns:py="http://genshi.edgewall.org/">
...   <head>
...     <title>Example</title>
...   </head>
...   <body>
...     <h1>Example</h1>
...     <p>${_("Hello, %(name)s") % dict(name=username)}</p>
...     <p>${ngettext("You have %d item", "You have %d items", num)}</p>
...   </body>
... </html>''', filename='example.html')
>>> for line, func, msg, comments in Translator().extract(tmpl.stream):
...     print('%d, %r, %r' % (line, func, msg))
3, None, u'Example'
6, None, u'Example'
7, '_', u'Hello, %(name)s'
8, 'ngettext', (u'You have %d item', u'You have %d items', None)
```

Parameters

- **stream** – the event stream to extract strings from; can be a regular stream or a template stream
- **gettext_functions** – a sequence of function names that should be treated as gettext-style localization functions

- **search_text** – whether the content of text nodes should be extracted (used internally)

Note Changed in 0.4.1: For a function with multiple string arguments (such as `ngettext`), a single item with a tuple of strings is yielded, instead an item for each string argument.

Note Changed in 0.6: The returned tuples now include a fourth element, which is a list of comments for the translator.

setup(*template*)

Convenience function to register the *Translator* filter and the related directives with the given template.

Parameters **template** – a *Template* instance

genshi.filters.i18n.**extract**(*fileobj*, *keywords*, *comment_tags*, *options*)

Babel extraction method for Genshi templates.

Parameters

- **fileobj** – the file-like object the messages should be extracted from
- **keywords** – a list of keywords (i.e. function names) that should be recognized as translation functions
- **comment_tags** – a list of translator tags to search for and include in the results
- **options** – a dictionary of additional options (optional)

Returns an iterator over `(lineno, funcname, message, comments)` tuples

Return type `iterator`

genshi.filters.transform

A filter for functional-style transformations of markup streams.

The *Transformer* filter provides a variety of transformations that can be applied to parts of streams that match given XPath expressions. These transformations can be chained to achieve results that would be comparitively tedious to achieve by writing stream filters by hand. The approach of chaining node selection and transformation has been inspired by the jQuery Javascript library.

For example, the following transformation removes the `<title>` element from the `<head>` of the input document:

```
>>> from genshi.builder import tag
>>> html = HTML('''<html>
...  <head><title>Some Title</title></head>
...  <body>
...    Some <em>body</em> text.
...  </body>
... </html>''',
... encoding='utf-8')
>>> print(html | Transformer('body/em').map(unicode.upper, TEXT)
...                                    .unwrap().wrap(tag.u))
<html>
  <head><title>Some Title</title></head>
  <body>
    Some <u>BODY</u> text.
  </body>
</html>
```

The `Transformer` support a large number of useful transformations out of the box, but custom transformations can be added easily.

since version 0.5

class `genshi.filters.transform.`**`Transformer`**(*path='.'*)

Stream filter that can apply a variety of different transformations to a stream.

This is achieved by selecting the events to be transformed using XPath, then applying the transformations to the events matched by the path expression. Each marked event is in the form (mark, (kind, data, pos)), where mark can be any of *ENTER*, *INSIDE*, *EXIT*, *OUTSIDE*, or *None*.

The first three marks match *START* and *END* events, and any events contained *INSIDE* any selected XML/HTML element. A non-element match outside a *START*/*END* container (e.g. `text()`) will yield an *OUTSIDE* mark.

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
```

Transformations act on selected stream events matching an XPath expression. Here's an example of removing some markup (the title, in this case) selected by an expression:

```
>>> print(html | Transformer('head/title').remove())
<html><head/><body>Some <em>body</em> text.</body></html>
```

Inserted content can be passed in the form of a string, or a markup event stream, which includes streams generated programmatically via the *builder* module:

```
>>> from genshi.builder import tag
>>> print(html | Transformer('body').prepend(tag.h1('Document Title')))
<html><head><title>Some Title</title></head><body><h1>Document
Title</h1>Some <em>body</em> text.</body></html>
```

Each XPath expression determines the set of tags that will be acted upon by subsequent transformations. In this example we select the `<title>` text, copy it into a buffer, then select the `<body>` element and paste the copied text into the body as `<h1>` enclosed text:

```
>>> buffer = StreamBuffer()
>>> print(html | Transformer('head/title/text()').copy(buffer)
...      .end().select('body').prepend(tag.h1(buffer)))
<html><head><title>Some Title</title></head><body><h1>Some Title</h1>Some
<em>body</em> text.</body></html>
```

Transformations can also be assigned and reused, although care must be taken when using buffers, to ensure that buffers are cleared between transforms:

```
>>> emphasis = Transformer('body//em').attr('class', 'emphasis')
>>> print(html | emphasis)
<html><head><title>Some Title</title></head><body>Some <em
class="emphasis">body</em> text.</body></html>
```

`after`(*content*)

Insert content after selection.

Here, we insert some text after the closing tag:

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//em').after(' rock'))
```

```
<html><head><title>Some Title</title></head><body>Some <em>body</em>
rock text.</body></html>
```

Parameters **content** – Either a callable, an iterable of events, or a string to insert.

Return type *Transformer*

append(*content*)

Insert content before the END event of the selection.

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//body').append(' Some new body text.'))
<html><head><title>Some Title</title></head><body>Some <em>body</em>
text. Some new body text.</body></html>
```

Parameters **content** – Either a callable, an iterable of events, or a string to insert.

Return type *Transformer*

apply(*function*)

Apply a transformation to the stream.

Transformations can be chained, similar to stream filters. Any callable accepting a marked stream can be used as a transform.

As an example, here is a simple *TEXT* event upper-casing transform:

```
>>> def upper(stream):
...     for mark, (kind, data, pos) in stream:
...         if mark and kind is TEXT:
...             yield mark, (kind, data.upper(), pos)
...         else:
...             yield mark, (kind, data, pos)
>>> short_stream = HTML('<body>Some <em>test</em> text</body>',
...                     encoding='utf-8')
>>> print(short_stream | Transformer('.//em/text()').apply(upper))
<body>Some <em>TEST</em> text</body>
```

attr(*name*, *value*)

Add, replace or delete an attribute on selected elements.

If *value* evaulates to *None* the attribute will be deleted from the element:

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em class="before">body</em> <em>text</em>.</body>'
...             '</html>', encoding='utf-8')
>>> print(html | Transformer('body/em').attr('class', None))
<html><head><title>Some Title</title></head><body>Some <em>body</em>
<em>text</em>.</body></html>
```

Otherwise the attribute will be set to *value*:

```
>>> print(html | Transformer('body/em').attr('class', 'emphasis'))
<html><head><title>Some Title</title></head><body>Some <em
class="emphasis">body</em> <em class="emphasis">text</em>.</body></html>
```

If *value* is a callable it will be called with the attribute name and the *START* event for the matching element. Its return value will then be used to set the attribute:

```
>>> def print_attr(name, event):
...     attrs = event[1][1]
...     print(attrs)
...     return attrs.get(name)
>>> print(html | Transformer('body/em').attr('class', print_attr))
Attrs([(QName('class'), u'before')])
Attrs()
<html><head><title>Some Title</title></head><body>Some <em
class="before">body</em> <em>text</em>.</body></html>
```

Parameters

- **name** – the name of the attribute
- **value** – the value that should be set for the attribute.

Return type *Transformer*

before(*content*)

Insert content before selection.

In this example we insert the word 'emphasised' before the opening tag:

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//em').before('emphasised '))
<html><head><title>Some Title</title></head><body>Some emphasised
<em>body</em> text.</body></html>
```

Parameters **content** – Either a callable, an iterable of events, or a string to insert.

Return type *Transformer*

buffer()

Buffer the entire stream (can consume a considerable amount of memory).

Useful in conjunction with copy(accumulate=True) and cut(accumulate=True) to ensure that all marked events in the entire stream are copied to the buffer before further transformations are applied.

For example, to move all <note> elements inside a <notes> tag at the top of the document:

```
>>> doc = HTML('<doc><notes></notes><body>Some <note>one</note> '
...            'text <note>two</note>.</body></doc>',
...            encoding='utf-8')
>>> buffer = StreamBuffer()
>>> print(doc | Transformer('body/note').cut(buffer, accumulate=True)
...     .end().buffer().select('notes').prepend(buffer))
<doc><notes><note>one</note><note>two</note></notes><body>Some  text
.</body></doc>
```

copy(*buffer*, *accumulate=False*)

Copy selection into buffer.

The buffer is replaced by each *contiguous* selection before being passed to the next transformation. If accumulate=True, further selections will be appended to the buffer rather than replacing it.

```
>>> from genshi.builder import tag
>>> buffer = StreamBuffer()
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('head/title/text()').copy(buffer)
...     .end().select('body').prepend(tag.h1(buffer)))
<html><head><title>Some Title</title></head><body><h1>Some
Title</h1>Some <em>body</em> text.</body></html>
```

This example illustrates that only a single contiguous selection will be buffered:

```
>>> print(html | Transformer('head/title/text()').copy(buffer)
...     .end().select('body/em').copy(buffer).end().select('body')
...     .prepend(tag.h1(buffer)))
<html><head><title>Some Title</title></head><body><h1>Some
Title</h1>Some <em>body</em> text.</body></html>
>>> print(buffer)
<em>body</em>
```

Element attributes can also be copied for later use:

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body><em>Some</em> <em class="before">body</em>'
...             '<em>text</em>.</body></html>',
...             encoding='utf-8')
>>> buffer = StreamBuffer()
>>> def apply_attr(name, entry):
...     return list(buffer)[0][1][1].get('class')
>>> print(html | Transformer('body/em[@class]/@class').copy(buffer)
...     .end().buffer().select('body/em[not(@class)]')
...     .attr('class', apply_attr))
<html><head><title>Some Title</title></head><body><em
class="before">Some</em> <em class="before">body</em><em
class="before">text</em>.</body></html>
```

Parameters **buffer** – the *StreamBuffer* in which the selection should be stored

Return type *Transformer*

Note Copy (and cut) copy each individual selected object into the buffer before passing to the next transform. For example, the XPath `*|text()` will select all elements and text, each instance of which will be copied to the buffer individually before passing to the next transform. This has implications for how `StreamBuffer` objects can be used, so some experimentation may be required.

cut(*buffer*, *accumulate=False*)

Copy selection into buffer and remove the selection from the stream.

```
>>> from genshi.builder import tag
>>> buffer = StreamBuffer()
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//em/text()').cut(buffer)
...     .end().select('.//em').after(tag.h1(buffer)))
<html><head><title>Some Title</title></head><body>Some
<em/><h1>body</h1> text.</body></html>
```

Specifying accumulate=True, appends all selected intervals onto the buffer. Combining this with the .buffer() operation allows us operate on all copied events rather than per-segment. See the documentation on buffer() for more information.

Parameters **buffer** – the *StreamBuffer* in which the selection should be stored

Return type *Transformer*

Note this transformation will buffer the entire input stream

empty()

Empty selected elements of all content.

Example:

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//em').empty())
<html><head><title>Some Title</title></head><body>Some <em/>
text.</body></html>
```

Return type *Transformer*

end()

End current selection, allowing all events to be selected.

Example:

```
>>> html = HTML('<body>Some <em>test</em> text</body>', encoding='utf-8')
>>> print(html | Transformer('//em').end().trace())
('OUTSIDE', ('START', (QName('body'), Attrs()), (None, 1, 0)))
('OUTSIDE', ('TEXT', u'Some ', (None, 1, 6)))
('OUTSIDE', ('START', (QName('em'), Attrs()), (None, 1, 11)))
('OUTSIDE', ('TEXT', u'test', (None, 1, 15)))
('OUTSIDE', ('END', QName('em'), (None, 1, 19)))
('OUTSIDE', ('TEXT', u' text', (None, 1, 24)))
('OUTSIDE', ('END', QName('body'), (None, 1, 29)))
<body>Some <em>test</em> text</body>
```

Returns the stream augmented by transformation marks

Return type *Transformer*

filter(*filter*)

Apply a normal stream filter to the selection. The filter is called once for each contiguous block of marked events.

```
>>> from genshi.filters.html import HTMLSanitizer
>>> html = HTML('<html><body>Some text<script>alert(document.cookie)'
...             '</script> and some more text</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('body/*').filter(HTMLSanitizer()))
<html><body>Some text and some more text</body></html>
```

Parameters **filter** – The stream filter to apply.

Return type *Transformer*

invert()
Invert selection so that marked events become unmarked, and vice versa.

Specificaly, all marks are converted to null marks, and all null marks are converted to OUTSIDE marks.

```
>>> html = HTML('<body>Some <em>test</em> text</body>', encoding='utf-8')
>>> print(html | Transformer('//em').invert().trace())
('OUTSIDE', ('START', (QName('body'), Attrs()), (None, 1, 0)))
('OUTSIDE', ('TEXT', u'Some ', (None, 1, 6)))
(None, ('START', (QName('em'), Attrs()), (None, 1, 11)))
(None, ('TEXT', u'test', (None, 1, 15)))
(None, ('END', QName('em'), (None, 1, 19)))
('OUTSIDE', ('TEXT', u' text', (None, 1, 24)))
('OUTSIDE', ('END', QName('body'), (None, 1, 29)))
<body>Some <em>test</em> text</body>
```

Return type *Transformer*

map(*function*, *kind*)
Applies a function to the `data` element of events of `kind` in the selection.

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...              '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('head/title').map(unicode.upper, TEXT))
<html><head><title>SOME TITLE</title></head><body>Some <em>body</em>
text.</body></html>
```

Parameters

- **function** – the function to apply
- **kind** – the kind of event the function should be applied to

Return type *Transformer*

prepend(*content*)
Insert content after the ENTER event of the selection.

Inserting some new text at the start of the <body>:

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//body').prepend('Some new body text. '))
<html><head><title>Some Title</title></head><body>Some new body text.
Some <em>body</em> text.</body></html>
```

Parameters **content** – Either a callable, an iterable of events, or a string to insert.

Return type *Transformer*

remove()
Remove selection from the stream.

Example:

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
```

```
>>> print(html | Transformer('.//em').remove())
<html><head><title>Some Title</title></head><body>Some
text.</body></html>
```

Return type *Transformer*

rename(*name*)

Rename matching elements.

```
>>> html = HTML('<html><body>Some text, some more text and '
...             '<b>some bold text</b></body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('body/b').rename('strong'))
<html><body>Some text, some more text and <strong>some bold text</strong></body></html>
```

replace(*content*)

Replace selection with content.

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//title/text()').replace('New Title'))
<html><head><title>New Title</title></head><body>Some <em>body</em>
text.</body></html>
```

Parameters **content** – Either a callable, an iterable of events, or a string to insert.

Return type *Transformer*

select(*path*)

Mark events matching the given XPath expression, within the current selection.

```
>>> html = HTML('<body>Some <em>test</em> text</body>', encoding='utf-8')
>>> print(html | Transformer().select('.//em').trace())
(None, ('START', (QName('body'), Attrs()), (None, 1, 0)))
(None, ('TEXT', u'Some ', (None, 1, 6)))
('ENTER', ('START', (QName('em'), Attrs()), (None, 1, 11)))
('INSIDE', ('TEXT', u'test', (None, 1, 15)))
('EXIT', ('END', QName('em'), (None, 1, 19)))
(None, ('TEXT', u' text', (None, 1, 24)))
(None, ('END', QName('body'), (None, 1, 29)))
<body>Some <em>test</em> text</body>
```

Parameters **path** – an XPath expression (as string) or a *Path* instance

Returns the stream augmented by transformation marks

Return type *Transformer*

substitute(*pattern*, *replace*, *count=1*)

Replace text matching a regular expression.

Refer to the documentation for `re.sub()` for details.

```
>>> html = HTML('<html><body>Some text, some more text and '
...             '<b>some bold text</b>\n'
...             '<i>some italicised text</i></body></html>',
...             encoding='utf-8')
```

```
>>> print(html | Transformer('body/b').substitute('(?i)some', 'SOME'))
<html><body>Some text, some more text and <b>SOME bold text</b>
<i>some italicised text</i></body></html>
>>> tags = tag.html(tag.body('Some text, some more text and\n',
...      Markup('<b>some bold text</b>')))
>>> print(tags.generate() | Transformer('body').substitute(
...     '(?i)some', 'SOME'))
<html><body>SOME text, some more text and
<b>SOME bold text</b></body></html>
```

Parameters

- **pattern** – A regular expression object or string.
- **replace** – Replacement pattern.
- **count** – Number of replacements to make in each text fragment.

Return type *Transformer*

trace(*prefix=''*, *fileobj=None*)

Print events as they pass through the transform.

```
>>> html = HTML('<body>Some <em>test</em> text</body>', encoding='utf-8')
>>> print(html | Transformer('em').trace())
(None, ('START', (QName('body'), Attrs()), (None, 1, 0)))
(None, ('TEXT', u'Some ', (None, 1, 6)))
('ENTER', ('START', (QName('em'), Attrs()), (None, 1, 11)))
('INSIDE', ('TEXT', u'test', (None, 1, 15)))
('EXIT', ('END', QName('em'), (None, 1, 19)))
(None, ('TEXT', u' text', (None, 1, 24)))
(None, ('END', QName('body'), (None, 1, 29)))
<body>Some <em>test</em> text</body>
```

Parameters

- **prefix** – a string to prefix each event with in the output
- **fileobj** – the writable file-like object to write to; defaults to the standard output stream

Return type *Transformer*

unwrap()

Remove outermost enclosing elements from selection.

Example:

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//em').unwrap())
<html><head><title>Some Title</title></head><body>Some body
text.</body></html>
```

Return type *Transformer*

wrap(*element*)

Wrap selection in an element.

```
>>> html = HTML('<html><head><title>Some Title</title></head>'
...             '<body>Some <em>body</em> text.</body></html>',
...             encoding='utf-8')
>>> print(html | Transformer('.//em').wrap('strong'))
<html><head><title>Some Title</title></head><body>Some
<strong><em>body</em></strong> text.</body></html>
```

Parameters **element** – either a tag name (as string) or an *Element* object

Return type *Transformer*

class genshi.filters.transform.**StreamBuffer**
Stream event buffer used for cut and copy transformations.

append(*event*)
Add an event to the buffer.

Parameters **event** – the markup event to add

reset()
Empty the buffer of events.

class genshi.filters.transform.**InjectorTransformation**(*content*)
Abstract base class for transformations that inject content into a stream.

```
>>> class Top(InjectorTransformation):
...     def __call__(self, stream):
...         for event in self._inject():
...             yield event
...         for event in stream:
...             yield event
>>> html = HTML('<body>Some <em>test</em> text</body>', encoding='utf-8')
>>> print(html | Transformer('.//em').apply(Top('Prefix ')))
Prefix <body>Some <em>test</em> text</body>
```

1.3.4 genshi.input

Support for constructing markup streams from files, strings, or other sources.

genshi.input.**ET**(*element*)
Convert a given ElementTree element to a markup stream.

Parameters **element** – an ElementTree element

Returns a markup stream

exception genshi.input.**ParseError**(*message*, *filename=None*, *lineno=-1*, *offset=-1*)
Exception raised when fatal syntax errors are found in the input being parsed.

class genshi.input.**XMLParser**(*source*, *filename=None*, *encoding=None*)
Generator-based XML parser based on roughly equivalent code in Kid/ElementTree.

The parsing is initiated by iterating over the parser object:

```
>>> parser = XMLParser(StringIO('<root id="2"><child>Foo</child></root>'))
>>> for kind, data, pos in parser:
...     print('%s %s' % (kind, data))
START (QName('root'), Attrs([(QName('id'), u'2')]))
START (QName('child'), Attrs())
TEXT Foo
```

```
    END child
    END root
```

parse()
Generator that parses the XML source, yielding markup events.

Returns a markup event stream

Raises ParseError if the XML text is not well formed

genshi.input.**XML**(*text*)
Parse the given XML source and return a markup stream.

Unlike with *XMLParser*, the returned stream is reusable, meaning it can be iterated over multiple times:

```
    >>> xml = XML('<doc><elem>Foo</elem><elem>Bar</elem></doc>')
    >>> print(xml)
    <doc><elem>Foo</elem><elem>Bar</elem></doc>
    >>> print(xml.select('elem'))
    <elem>Foo</elem><elem>Bar</elem>
    >>> print(xml.select('elem/text()'))
    FooBar
```

Parameters **text** – the XML source

Returns the parsed XML event stream

Raises ParseError if the XML text is not well-formed

class genshi.input.**HTMLParser**(*source*, *filename=None*, *encoding=None*)
Parser for HTML input based on the Python *HTMLParser* module.

This class provides the same interface for generating stream events as *XMLParser*, and attempts to automatically balance tags.

The parsing is initiated by iterating over the parser object:

```
    >>> parser = HTMLParser(BytesIO(u'<UL compact><LI>Foo</UL>'.encode('utf-8')), encoding='utf-8')
    >>> for kind, data, pos in parser:
    ...     print('%s %s' % (kind, data))
    START (QName('ul'), Attrs([(QName('compact'), u'compact')]))
    START (QName('li'), Attrs())
    TEXT Foo
    END li
    END ul
```

parse()
Generator that parses the HTML source, yielding markup events.

Returns a markup event stream

Raises ParseError if the HTML text is not well formed

genshi.input.**HTML**(*text*, *encoding=None*)
Parse the given HTML source and return a markup stream.

Unlike with *HTMLParser*, the returned stream is reusable, meaning it can be iterated over multiple times:

```
    >>> html = HTML('<body><h1>Foo</h1></body>', encoding='utf-8')
    >>> print(html)
    <body><h1>Foo</h1></body>
    >>> print(html.select('h1'))
```

```
    <h1>Foo</h1>
    >>> print(html.select('h1/text()'))
    Foo
```

Parameters **text** – the HTML source

Returns the parsed XML event stream

Raises ParseError if the HTML text is not well-formed, and error recovery fails

1.3.5 genshi.output

This module provides different kinds of serialization methods for XML event streams.

genshi.output.**encode**(*iterator*, *method='xml'*, *encoding=None*, *out=None*)

Encode serializer output into a string.

Parameters

- **iterator** – the iterator returned from serializing a stream (basically any iterator that yields unicode objects)
- **method** – the serialization method; determines how characters not representable in the specified encoding are treated
- **encoding** – how the output string should be encoded; if set to *None*, this method returns a *unicode* object
- **out** – a file-like object that the output should be written to instead of being returned as one big string; note that if this is a file or socket (or similar), the *encoding* must not be *None* (that is, the output must be encoded)

Returns a *str* or *unicode* object (depending on the *encoding* parameter), or *None* if the *out* parameter is provided

Since version 0.4.1

Note Changed in 0.5: added the *out* parameter

genshi.output.**get_serializer**(*method='xml'*, ***kwargs*)

Return a serializer object for the given method.

Parameters **method** – the serialization method; can be either "xml", "xhtml", "html", "text", or a custom serializer class

Any additional keyword arguments are passed to the serializer, and thus depend on the *method* parameter value.

See *XMLSerializer*, *XHTMLSerializer*, *HTMLSerializer*, *TextSerializer*

Since version 0.4.1

class genshi.output.**DocType**

Defines a number of commonly used DOCTYPE declarations as constants.

classmethod **get**(*name*)

Return the `(name, pubid, sysid)` tuple of the `DOCTYPE` declaration for the specified name.

The following names are recognized in this version:

- "html" or "html-strict" for the HTML 4.01 strict DTD
- "html-transitional" for the HTML 4.01 transitional DTD
- "html-frameset" for the HTML 4.01 frameset DTD

- "html5" for the DOCTYPE proposed for HTML5
- "xhtml" or "xhtml-strict" for the XHTML 1.0 strict DTD
- "xhtml-transitional" for the XHTML 1.0 transitional DTD
- "xhtml-frameset" for the XHTML 1.0 frameset DTD
- "xhtml11" for the XHTML 1.1 DTD
- "svg" or "svg-full" for the SVG 1.1 DTD
- "svg-basic" for the SVG Basic 1.1 DTD
- "svg-tiny" for the SVG Tiny 1.1 DTD

Parameters **name** – the name of the DOCTYPE

Returns the (name, pubid, sysid) tuple for the requested DOCTYPE, or None if the name is not recognized

Since version 0.4.1

class genshi.output.**XMLSerializer**(*doctype=None*, *strip_whitespace=True*, *namespace_prefixes=None*, *cache=True*)

Produces XML text from an event stream.

```
>>> from genshi.builder import tag
>>> elem = tag.div(tag.a(href='foo'), tag.br, tag.hr(noshade=True))
>>> print(''.join(XMLSerializer()(elem.generate())))
<div><a href="foo"/><br/><hr noshade="True"/></div>
```

class genshi.output.**XHTMLSerializer**(*doctype=None*, *strip_whitespace=True*, *namespace_prefixes=None*, *drop_xml_decl=True*, *cache=True*)

Produces XHTML text from an event stream.

```
>>> from genshi.builder import tag
>>> elem = tag.div(tag.a(href='foo'), tag.br, tag.hr(noshade=True))
>>> print(''.join(XHTMLSerializer()(elem.generate())))
<div><a href="foo"></a><br /><hr noshade="noshade" /></div>
```

class genshi.output.**HTMLSerializer**(*doctype=None*, *strip_whitespace=True*, *cache=True*)

Produces HTML text from an event stream.

```
>>> from genshi.builder import tag
>>> elem = tag.div(tag.a(href='foo'), tag.br, tag.hr(noshade=True))
>>> print(''.join(HTMLSerializer()(elem.generate())))
<div><a href="foo"></a><br><hr noshade></div>
```

class genshi.output.**TextSerializer**(*strip_markup=False*)

Produces plain text from an event stream.

Only text events are included in the output. Unlike the other serializer, special XML characters are not escaped:

```
>>> from genshi.builder import tag
>>> elem = tag.div(tag.a('<Hello!>', href='foo'), tag.br)
>>> print(elem)
<div><a href="foo">&lt;Hello!&gt;</a><br/></div>
>>> print(''.join(TextSerializer()(elem.generate())))
<Hello!>
```

If text events contain literal markup (instances of the *Markup* class), that markup is by default passed through unchanged:

```
>>> elem = tag.div(Markup('<a href="foo">Hello &amp; Bye!</a><br/>'))
>>> print(elem.generate().render(TextSerializer, encoding=None))
<a href="foo">Hello &amp; Bye!</a><br/>
```

You can use the `strip_markup` to change this behavior, so that tags and entities are stripped from the output (or in the case of entities, replaced with the equivalent character):

```
>>> print(elem.generate().render(TextSerializer, strip_markup=True,
...                              encoding=None))
Hello & Bye!
```

1.3.6 genshi.path

Basic support for evaluating XPath expressions against streams.

```
>>> from genshi.input import XML
>>> doc = XML('''<doc>
...  <items count="4">
...       <item status="new">
...         <summary>Foo</summary>
...       </item>
...       <item status="closed">
...         <summary>Bar</summary>
...       </item>
...       <item status="closed" resolution="invalid">
...         <summary>Baz</summary>
...       </item>
...       <item status="closed" resolution="fixed">
...         <summary>Waz</summary>
...       </item>
...   </items>
... </doc>''')
>>> print(doc.select('items/item[@status="closed" and '
...     '(@resolution="invalid" or not(@resolution))]/summary/text()'))
BarBaz
```

Because the XPath engine operates on markup streams (as opposed to tree structures), it only implements a subset of the full XPath 1.0 language.

class `genshi.path.`**`Path`**(*text*, *filename=None*, *lineno=-1*)

Implements basic XPath support on streams.

Instances of this class represent a "compiled" XPath expression, and provide methods for testing the path against a stream, as well as extracting a substream matching that path.

`select`(*stream*, *namespaces=None*, *variables=None*)

Returns a substream of the given stream that matches the path.

If there are no matches, this method returns an empty stream.

```
>>> from genshi.input import XML
>>> xml = XML('<root><elem><child>Text</child></elem></root>')
```

```
>>> print(Path('.//child').select(xml))
<child>Text</child>
```

```
>>> print(Path('.//child/text()').select(xml))
Text
```

Parameters

- **stream** – the stream to select from
- **namespaces** – (optional) a mapping of namespace prefixes to URIs
- **variables** – (optional) a mapping of variable names to values

Returns the substream matching the path, or an empty stream

Return type *Stream*

test(*ignore_context=False*)

Returns a function that can be used to track whether the path matches a specific stream event.

The function returned expects the positional arguments `event`, `namespaces` and `variables`. The first is a stream event, while the latter two are a mapping of namespace prefixes to URIs, and a mapping of variable names to values, respectively. In addition, the function accepts an `updateonly` keyword argument that default to `False`. If it is set to `True`, the function only updates its internal state, but does not perform any tests or return a result.

If the path matches the event, the function returns the match (for example, a *START* or *TEXT* event.) Otherwise, it returns `None`.

```
>>> from genshi.input import XML
>>> xml = XML('<root><elem><child id="1"/></elem><child id="2"/></root>')
>>> test = Path('child').test()
>>> namespaces, variables = {}, {}
>>> for event in xml:
...     if test(event, namespaces, variables):
...         print('%s %r' % (event[0], event[1]))
START (QName('child'), Attrs([(QName('id'), u'2')]))
```

Parameters **ignore_context** – if *True*, the path is interpreted like a pattern in XSLT, meaning for example that it will match at any depth

Returns a function that can be used to test individual events in a stream against the path

Return type `function`

exception genshi.path.**PathSyntaxError**(*message*, *filename=None*, *lineno=-1*, *offset=-1*)

Exception raised when an XPath expression is syntactically incorrect.

1.3.7 genshi.template

Implementation of the template engine.

genshi.template.base

Basic templating functionality.

class genshi.template.base.**Context**(***data*)

Container for template input data.

A context provides a stack of scopes (represented by dictionaries).

Template directives such as loops can push a new scope on the stack with data that should only be available inside the loop. When the loop terminates, that scope can get popped off the stack again.

```
>>> ctxt = Context(one='foo', other=1)
>>> ctxt.get('one')
'foo'
>>> ctxt.get('other')
1
>>> ctxt.push(dict(one='frost'))
>>> ctxt.get('one')
'frost'
>>> ctxt.get('other')
1
>>> ctxt.pop()
{'one': 'frost'}
>>> ctxt.get('one')
'foo'
```

get(*key*, *default=None*)
Get a variable's value, starting at the current scope and going upward.

Parameters

- **key** – the name of the variable
- **default** – the default value to return when the variable is not found

has_key(*key*)
Return whether a variable exists in any of the scopes.

Parameters **key** – the name of the variable

items()
Return a list of `(name, value)` tuples for all variables in the context.

Returns a list of variables

keys()
Return the name of all variables in the context.

Returns a list of variable names

pop()
Pop the top-most scope from the stack.

push(*data*)
Push a new scope on the stack.

Parameters **data** – the data dictionary to push on the context stack.

update(*mapping*)
Update the context from the mapping provided.

class genshi.template.base.**DirectiveFactory**
Base for classes that provide a set of template directives.

Since version 0.6

get_directive(*name*)
Return the directive class for the given name.

Parameters **name** – the directive name as used in the template

Returns the directive class

See *Directive*

get_directive_index(*dir_cls*)
Return a key for the given directive class that should be used to sort it among other directives on the same *SUB* event.

The default implementation simply returns the index of the directive in the *directives* list.

Parameters **dir_cls** – the directive class

Returns the sort key

class genshi.template.base.**Template**(*source*, *filepath=None*, *filename=None*, *loader=None*, *encoding=None*, *lookup='strict'*, *allow_exec=True*)
Abstract template base class.

This class implements most of the template processing model, but does not specify the syntax of templates.

generate(**args*, ***kwargs*)
Apply the template to the given context data.

Any keyword arguments are made available to the template as context data.

Only one positional argument is accepted: if it is provided, it must be an instance of the *Context* class, and keyword arguments are ignored. This calling style is used for internal processing.

Returns a markup event stream representing the result of applying the template to the context data.

exception genshi.template.base.**TemplateError**(*message*, *filename=None*, *lineno=-1*, *offset=-1*)
Base exception class for errors related to template processing.

exception genshi.template.base.**TemplateRuntimeError**(*message*, *filename=None*, *lineno=-1*, *offset=-1*)
Exception raised when an the evaluation of a Python expression in a template causes an error.

exception genshi.template.base.**TemplateSyntaxError**(*message*, *filename=None*, *lineno=-1*, *offset=-1*)
Exception raised when an expression in a template causes a Python syntax error, or the template is not well-formed.

exception genshi.template.base.**BadDirectiveError**(*name*, *filename=None*, *lineno=-1*)
Exception raised when an unknown directive is encountered when parsing a template.

An unknown directive is any attribute using the namespace for directives, with a local name that doesn't match any registered directive.

genshi.template.directives

Implementation of the various template directives.

class genshi.template.directives.**AttrsDirective**(*value*, *template=None*, *namespaces=None*, *lineno=-1*, *offset=-1*)
Implementation of the `py:attrs` template directive.

The value of the `py:attrs` attribute should be a dictionary or a sequence of `(name, value)` tuples. The items in that dictionary or sequence are added as attributes to the element:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<ul xmlns:py="http://genshi.edgewall.org/">
...   <li py:attrs="foo">Bar</li>
... </ul>''')
>>> print(tmpl.generate(foo={'class': 'collapse'}))
<ul>
  <li class="collapse">Bar</li>
```

```
</ul>
>>> print(tmpl.generate(foo=[('class', 'collapse')]))
<ul>
  <li class="collapse">Bar</li>
</ul>
```

If the value evaluates to `None` (or any other non-truth value), no attributes are added:

```
>>> print(tmpl.generate(foo=None))
<ul>
  <li>Bar</li>
</ul>
```

class genshi.template.directives.**ChooseDirective**(*value*, *template=None*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:choose` directive for conditionally selecting one of several body elements to display.

If the `py:choose` expression is empty the expressions of nested `py:when` directives are tested for truth. The first true `py:when` body is output. If no `py:when` directive is matched then the fallback directive `py:otherwise` will be used.

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/"
...   py:choose="">
...   <span py:when="0 == 1">0</span>
...   <span py:when="1 == 1">1</span>
...   <span py:otherwise="">2</span>
... </div>''')
>>> print(tmpl.generate())
<div>
  <span>1</span>
</div>
```

If the `py:choose` directive contains an expression, the nested `py:when` directives are tested for equality to the `py:choose` expression:

```
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/"
...   py:choose="2">
...   <span py:when="1">1</span>
...   <span py:when="2">2</span>
... </div>''')
>>> print(tmpl.generate())
<div>
  <span>2</span>
</div>
```

Behavior is undefined if a `py:choose` block contains content outside a `py:when` or `py:otherwise` block. Behavior is also undefined if a `py:otherwise` occurs before `py:when` blocks.

class genshi.template.directives.**ContentDirective**(*value*, *template=None*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:content` template directive.

This directive replaces the content of the element with the result of evaluating the value of the `py:content` attribute:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<ul xmlns:py="http://genshi.edgewall.org/">
...   <li py:content="bar">Hello</li>
... </ul>''')
```

```
>>> print(tmpl.generate(bar='Bye'))
<ul>
  <li>Bye</li>
</ul>
```

class genshi.template.directives.**DefDirective**(*args*, *template*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:def` template directive.

This directive can be used to create "Named Template Functions", which are template snippets that are not actually output during normal processing, but rather can be expanded from expressions in other places in the template.

A named template function can be used just like a normal Python function from template expressions:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <p py:def="echo(greeting, name='world')" class="message">
...     ${greeting}, ${name}!
...   </p>
...   ${echo('Hi', name='you')}
... </div>''')
>>> print(tmpl.generate(bar='Bye'))
<div>
  <p class="message">
    Hi, you!
  </p>
</div>
```

If a function does not require parameters, the parenthesis can be omitted in the definition:

```
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <p py:def="helloworld" class="message">
...     Hello, world!
...   </p>
...   ${helloworld()}
... </div>''')
>>> print(tmpl.generate(bar='Bye'))
<div>
  <p class="message">
    Hello, world!
  </p>
</div>
```

class genshi.template.directives.**ForDirective**(*value*, *template*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:for` template directive for repeating an element based on an iterable in the context data.

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<ul xmlns:py="http://genshi.edgewall.org/">
...   <li py:for="item in items">${item}</li>
... </ul>''')
>>> print(tmpl.generate(items=[1, 2, 3]))
<ul>
  <li>1</li><li>2</li><li>3</li>
</ul>
```

class genshi.template.directives.**IfDirective**(*value*, *template=None*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:if` template directive for conditionally excluding elements from being output.

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <b py:if="foo">${bar}</b>
... </div>''')
>>> print(tmpl.generate(foo=True, bar='Hello'))
<div>
  <b>Hello</b>
</div>
```

class genshi.template.directives.**MatchDirective**(*value*, *template*, *hints=None*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:match` template directive.

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <span py:match="greeting">
...     Hello ${select('@name')}
...   </span>
...   <greeting name="Dude" />
... </div>''')
>>> print(tmpl.generate())
<div>
  <span>
    Hello Dude
  </span>
</div>
```

class genshi.template.directives.**OtherwiseDirective**(*value*, *template*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:otherwise` directive for nesting in a parent with the `py:choose` directive.

See the documentation of *ChooseDirective* for usage.

class genshi.template.directives.**ReplaceDirective**(*value*, *template=None*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:replace` template directive.

This directive replaces the element with the result of evaluating the value of the `py:replace` attribute:

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <span py:replace="bar">Hello</span>
... </div>''')
>>> print(tmpl.generate(bar='Bye'))
<div>
  Bye
</div>
```

This directive is equivalent to `py:content` combined with `py:strip`, providing a less verbose way to achieve the same effect:

```
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <span py:content="bar" py:strip="">Hello</span>
... </div>''')
>>> print(tmpl.generate(bar='Bye'))
<div>
  Bye
</div>
```

class genshi.template.directives.**StripDirective**(*value*, *template=None*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:strip` template directive.

When the value of the `py:strip` attribute evaluates to `True`, the element is stripped from the output

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <div py:strip="True"><b>foo</b></div>
... </div>''')
>>> print(tmpl.generate())
<div>
  <b>foo</b>
</div>
```

Leaving the attribute value empty is equivalent to a truth value.

This directive is particulary interesting for named template functions or match templates that do not generate a top-level element:

```
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <div py:def="echo(what)" py:strip="">
...     <b>${what}</b>
...   </div>
...   ${echo('foo')}
... </div>''')
>>> print(tmpl.generate())
<div>
    <b>foo</b>
</div>
```

class genshi.template.directives.**WhenDirective**(*value*, *template*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:when` directive for nesting in a parent with the `py:choose` directive.

See the documentation of the *ChooseDirective* for usage.

class genshi.template.directives.**WithDirective**(*value*, *template*, *namespaces=None*, *lineno=-1*, *offset=-1*)

Implementation of the `py:with` template directive, which allows shorthand access to variables and expressions.

```
>>> from genshi.template import MarkupTemplate
>>> tmpl = MarkupTemplate('''<div xmlns:py="http://genshi.edgewall.org/">
...   <span py:with="y=7; z=x+10">$x $y $z</span>
... </div>''')
>>> print(tmpl.generate(x=42))
<div>
  <span>42 7 52</span>
</div>
```

genshi.template.eval

Support for "safe" evaluation of Python expressions.

class genshi.template.eval.**Code**(*source*, *filename=None*, *lineno=-1*, *lookup='strict'*, *xform=None*)

Abstract base class for the *Expression* and *Suite* classes.

class genshi.template.eval.**Expression**(*source*, *filename=None*, *lineno=-1*, *lookup='strict'*, *xform=None*)

Evaluates Python expressions used in templates.

```
>>> data = dict(test='Foo', items=[1, 2, 3], dict={'some': 'thing'})
>>> Expression('test').evaluate(data)
'Foo'
```

```
>>> Expression('items[0]').evaluate(data)
1
>>> Expression('items[-1]').evaluate(data)
3
>>> Expression('dict["some"]').evaluate(data)
'thing'
```

Similar to e.g. Javascript, expressions in templates can use the dot notation for attribute access to access items in mappings:

```
>>> Expression('dict.some').evaluate(data)
'thing'
```

This also works the other way around: item access can be used to access any object attribute:

```
>>> class MyClass(object):
...     myattr = 'Bar'
>>> data = dict(mine=MyClass(), key='myattr')
>>> Expression('mine.myattr').evaluate(data)
'Bar'
>>> Expression('mine["myattr"]').evaluate(data)
'Bar'
>>> Expression('mine[key]').evaluate(data)
'Bar'
```

All of the standard Python operators are available to template expressions. Built-in functions such as `len()` are also available in template expressions:

```
>>> data = dict(items=[1, 2, 3])
>>> Expression('len(items)').evaluate(data)
3
```

evaluate(*data*)

Evaluate the expression against the given data dictionary.

Parameters **data** – a mapping containing the data to evaluate against

Returns the result of the evaluation

class genshi.template.eval.**Suite**(*source*, *filename=None*, *lineno=-1*, *lookup='strict'*, *xform=None*)

Executes Python statements used in templates.

```
>>> data = dict(test='Foo', items=[1, 2, 3], dict={'some': 'thing'})
>>> Suite("foo = dict['some']").execute(data)
>>> data['foo']
'thing'
```

execute(*data*)

Execute the suite in the given data dictionary.

Parameters **data** – a mapping containing the data to execute in

class genshi.template.eval.**LenientLookup**

Default variable lookup mechanism for expressions.

When an undefined variable is referenced using this lookup style, the reference evaluates to an instance of the *Undefined* class:

```
>>> expr = Expression('nothing', lookup='lenient')
>>> undef = expr.evaluate({})
>>> undef
<Undefined 'nothing'>
```

The same will happen when a non-existing attribute or item is accessed on an existing object:

```
>>> expr = Expression('something.nil', lookup='lenient')
>>> expr.evaluate({'something': dict()})
<Undefined 'nil'>
```

See the documentation of the *Undefined* class for details on the behavior of such objects.

See *StrictLookup*

classmethod undefined(*key*, *owner=<object object>*)
Return an `Undefined` object.

class genshi.template.eval.**StrictLookup**
Strict variable lookup mechanism for expressions.

Referencing an undefined variable using this lookup style will immediately raise an `UndefinedError`:

```
>>> expr = Expression('nothing', lookup='strict')
>>> try:
...     expr.evaluate({})
... except UndefinedError, e:
...     print e.msg
"nothing" not defined
```

The same happens when a non-existing attribute or item is accessed on an existing object:

```
>>> expr = Expression('something.nil', lookup='strict')
>>> try:
...     expr.evaluate({'something': dict()})
... except UndefinedError, e:
...     print e.msg
{} has no member named "nil"
```

classmethod undefined(*key*, *owner=<object object>*)
Raise an `UndefinedError` immediately.

class genshi.template.eval.**Undefined**(*name*, *owner=<object object>*)
Represents a reference to an undefined variable.

Unlike the Python runtime, template expressions can refer to an undefined variable without causing a *NameError* to be raised. The result will be an instance of the *Undefined* class, which is treated the same as `False` in conditions, but raise an exception on any other operation:

```
>>> foo = Undefined('foo')
>>> bool(foo)
False
>>> list(foo)
[]
>>> print(foo)
undefined
```

However, calling an undefined variable, or trying to access an attribute of that variable, will raise an exception that includes the name used to reference that undefined variable.

```
>>> try:
...     foo('bar')
... except UndefinedError, e:
...     print e.msg
"foo" not defined
```

```
>>> try:
...     foo.bar
... except UndefinedError, e:
...     print e.msg
"foo" not defined
```

See *LenientLookup*

exception genshi.template.eval.**UndefinedError**(*name*, *owner=<object object>*)
Exception thrown when a template expression attempts to access a variable not defined in the context.

See *LenientLookup*, *StrictLookup*

genshi.template.interpolation

String interpolation routines, i.e. the splitting up a given text into some parts that are literal strings, and others that are Python expressions.

genshi.template.interpolation.**interpolate**(*text*, *filepath=None*, *lineno=-1*, *offset=0*, *lookup='strict'*)
Parse the given string and extract expressions.

This function is a generator that yields *TEXT* events for literal strings, and *EXPR* events for expressions, depending on the results of parsing the string.

```
>>> for kind, data, pos in interpolate("hey ${foo}bar"):
...     print('%s %r' % (kind, data))
TEXT 'hey '
EXPR Expression('foo')
TEXT 'bar'
```

Parameters

- **text** – the text to parse
- **filepath** – absolute path to the file in which the text was found (optional)
- **lineno** – the line number at which the text was found (optional)
- **offset** – the column number at which the text starts in the source (optional)
- **lookup** – the variable lookup mechanism; either "lenient" (the default), "strict", or a custom lookup class

Returns a list of *TEXT* and *EXPR* events

Raises TemplateSyntaxError when a syntax error in an expression is encountered

genshi.template.loader

Template loading and caching.

class genshi.template.loader.**TemplateLoader**(*search_path=None*, *auto_reload=False*, *default_encoding=None*, *max_cache_size=25*, *default_class=None*, *variable_lookup='strict'*, *allow_exec=True*, *callback=None*)

Responsible for loading templates from files on the specified search path.

```
>>> import tempfile
>>> fd, path = tempfile.mkstemp(suffix='.html', prefix='template')
>>> os.write(fd, u'<p>$var</p>'.encode('utf-8'))
11
>>> os.close(fd)
```

The template loader accepts a list of directory paths that are then used when searching for template files, in the given order:

```
>>> loader = TemplateLoader([os.path.dirname(path)])
```

The *load()* method first checks the template cache whether the requested template has already been loaded. If not, it attempts to locate the template file, and returns the corresponding *Template* object:

```
>>> from genshi.template import MarkupTemplate
>>> template = loader.load(os.path.basename(path))
>>> isinstance(template, MarkupTemplate)
True
```

Template instances are cached: requesting a template with the same name results in the same instance being returned:

```
>>> loader.load(os.path.basename(path)) is template
True
```

The *auto_reload* option can be used to control whether a template should be automatically reloaded when the file it was loaded from has been changed. Disable this automatic reloading to improve performance.

```
>>> os.remove(path)
```

static directory(*path*)

Loader factory for loading templates from a local directory.

Parameters **path** – the path to the local directory containing the templates

Returns the loader function to load templates from the given directory

Return type function

load(*filename*, *relative_to=None*, *cls=None*, *encoding=None*)

Load the template with the given name.

If the *filename* parameter is relative, this method searches the search path trying to locate a template matching the given name. If the file name is an absolute path, the search path is ignored.

If the requested template is not found, a *TemplateNotFound* exception is raised. Otherwise, a *Template* object is returned that represents the parsed template.

Template instances are cached to avoid having to parse the same template file more than once. Thus, subsequent calls of this method with the same template file name will return the same *Template* object (unless the auto_reload option is enabled and the file was changed since the last parse.)

If the *relative_to* parameter is provided, the *filename* is interpreted as being relative to that path.

Parameters

- **filename** – the relative path of the template file to load

- **relative_to** – the filename of the template from which the new template is being loaded, or `None` if the template is being loaded directly
- **cls** – the class of the template object to instantiate
- **encoding** – the encoding of the template to load; defaults to the `default_encoding` of the loader instance

Returns the loaded *Template* instance

Raises TemplateNotFound if a template with the given name could not be found

static package(*name*, *path*)

Loader factory for loading templates from egg package data.

Parameters

- **name** – the name of the package containing the resources
- **path** – the path inside the package data

Returns the loader function to load templates from the given package

Return type `function`

static prefixed(***delegates*)

Factory for a load function that delegates to other loaders depending on the prefix of the requested template path.

The prefix is stripped from the filename when passing on the load request to the delegate.

```
>>> load = prefixed(
...     app1 = lambda filename: ('app1', filename, None, None),
...     app2 = lambda filename: ('app2', filename, None, None)
... )
>>> print(load('app1/foo.html'))
('app1', 'app1/foo.html', None, None)
>>> print(load('app2/bar.html'))
('app2', 'app2/bar.html', None, None)
```

Parameters **delegates** – mapping of path prefixes to loader functions

Returns the loader function

Return type `function`

exception genshi.template.loader.**TemplateNotFound**(*name*, *search_path*)

Exception raised when a specific template file could not be found.

genshi.template.loader.**directory**(*path*)

Loader factory for loading templates from a local directory.

Parameters **path** – the path to the local directory containing the templates

Returns the loader function to load templates from the given directory

Return type `function`

genshi.template.loader.**package**(*name*, *path*)

Loader factory for loading templates from egg package data.

Parameters

- **name** – the name of the package containing the resources
- **path** – the path inside the package data

Returns the loader function to load templates from the given package

Return type function

genshi.template.loader.**prefixed**(***delegates*)

Factory for a load function that delegates to other loaders depending on the prefix of the requested template path.

The prefix is stripped from the filename when passing on the load request to the delegate.

```
>>> load = prefixed(
...     app1 = lambda filename: ('app1', filename, None, None),
...     app2 = lambda filename: ('app2', filename, None, None)
... )
>>> print(load('app1/foo.html'))
('app1', 'app1/foo.html', None, None)
>>> print(load('app2/bar.html'))
('app2', 'app2/bar.html', None, None)
```

Parameters **delegates** – mapping of path prefixes to loader functions

Returns the loader function

Return type function

genshi.template.markup

Markup templating engine.

class genshi.template.markup.**MarkupTemplate**(*source*, *filepath=None*, *filename=None*, *loader=None*, *encoding=None*, *lookup='strict'*, *allow_exec=True*)

Implementation of the template language for XML-based templates.

```
>>> tmpl = MarkupTemplate('''<ul xmlns:py="http://genshi.edgewall.org/">
...   <li py:for="item in items">${item}</li>
... </ul>''')
>>> print(tmpl.generate(items=[1, 2, 3]))
<ul>
  <li>1</li><li>2</li><li>3</li>
</ul>
```

add_directives(*namespace*, *factory*)

Register a custom *DirectiveFactory* for a given namespace.

Parameters

- **namespace** (*basestring*) – the namespace URI
- **factory** (*DirectiveFactory*) – the directive factory to register

Since version 0.6

genshi.template.plugin

Basic support for the template engine plugin API used by TurboGears and CherryPy/Buffet.

exception genshi.template.plugin.**ConfigurationError**

Exception raised when invalid plugin options are encountered.

class genshi.template.plugin.**AbstractTemplateEnginePlugin**(*extra_vars_func=None*, *options=None*)

Implementation of the plugin API.

load_template(*templatename*, *template_string=None*)
Find a template specified in python 'dot' notation, or load one from a string.

render(*info*, *format=None*, *fragment=False*, *template=None*)
Render the template to a string using the provided info.

transform(*info*, *template*)
Render the output to an event stream.

class genshi.template.plugin.**MarkupTemplateEnginePlugin**(*extra_vars_func=None*, *options=None*)

Implementation of the plugin API for markup templates.

template_class
alias of MarkupTemplate

transform(*info*, *template*)
Render the output to an event stream.

class genshi.template.plugin.**TextTemplateEnginePlugin**(*extra_vars_func=None*, *options=None*)

Implementation of the plugin API for text templates.

template_class
alias of OldTextTemplate

genshi.template.text

Plain text templating engine.

This module implements two template language syntaxes, at least for a certain transitional period. *OldTextTemplate* (aliased to just *TextTemplate*) defines a syntax that was inspired by Cheetah/Velocity. *NewTextTemplate* on the other hand is inspired by the syntax of the Django template language, which has more explicit delimiting of directives, and is more flexible with regards to white space and line breaks.

In a future release, *OldTextTemplate* will be phased out in favor of *NewTextTemplate*, as the names imply. Therefore the new syntax is strongly recommended for new projects, and existing projects may want to migrate to the new syntax to remain compatible with future Genshi releases.

class genshi.template.text.**NewTextTemplate**(*source*, *filepath=None*, *filename=None*, *loader=None*, *encoding=None*, *lookup='strict'*, *allow_exec=False*, *delims=('{%', '%}', '{#', '#}')*)

Implementation of a simple text-based template engine. This class will replace *OldTextTemplate* in a future release.

It uses a more explicit delimiting style for directives: instead of the old style which required putting directives on separate lines that were prefixed with a `#` sign, directives and commenbtsr are enclosed in delimiter pairs (by default `{% ... %}` and `{# ... #}`, respectively).

Variable substitution uses the same interpolation syntax as for markup languages: simple references are prefixed with a dollar sign, more complex expression enclosed in curly braces.

```
>>> tmpl = NewTextTemplate('''Dear $name,
...
... {# This is a comment #}
... We have the following items for you:
```

```
... {% for item in items %}
...  * ${'Item %d' % item}
... {% end %}
... ''')
>>> print(tmpl.generate(name='Joe', items=[1, 2, 3]).render(encoding=None))
Dear Joe,


We have the following items for you:

 * Item 1

 * Item 2

 * Item 3
```

By default, no spaces or line breaks are removed. If a line break should not be included in the output, prefix it with a backslash:

```
>>> tmpl = NewTextTemplate('''Dear $name,
...
... {# This is a comment #}\
... We have the following items for you:
... {% for item in items %}\
...  * $item
... {% end %}\
... ''')
>>> print(tmpl.generate(name='Joe', items=[1, 2, 3]).render(encoding=None))
Dear Joe,

We have the following items for you:
 * 1
 * 2
 * 3
```

Backslashes are also used to escape the start delimiter of directives and comments:

```
>>> tmpl = NewTextTemplate('''Dear $name,
...
... \{# This is a comment #}
... We have the following items for you:
... {% for item in items %}\
...  * $item
... {% end %}\
... ''')
>>> print(tmpl.generate(name='Joe', items=[1, 2, 3]).render(encoding=None))
Dear Joe,

{# This is a comment #}
We have the following items for you:
 * 1
 * 2
 * 3
```

Since version 0.5

class genshi.template.text.**OldTextTemplate**(*source*, *filepath=None*, *filename=None*, *loader=None*, *encoding=None*, *lookup='strict'*, *allow_exec=True*)

Legacy implementation of the old syntax text-based templates. This class is provided in a transition phase for backwards compatibility. New code should use the *NewTextTemplate* class and the improved syntax it provides.

```
>>> tmpl = OldTextTemplate('''Dear $name,
...
... We have the following items for you:
... #for item in items
...  * $item
... #end
...
... All the best,
... Foobar''')
>>> print(tmpl.generate(name='Joe', items=[1, 2, 3]).render(encoding=None))
Dear Joe,

We have the following items for you:
 * 1
 * 2
 * 3

All the best,
Foobar
```

genshi.template.text.**TextTemplate**
alias of *OldTextTemplate*

1.3.8 genshi.util

Various utility classes and functions.

class genshi.util.**LRUCache**(*capacity*)
A dictionary-like object that stores only a certain number of items, and discards its least recently used item when full.

```
>>> cache = LRUCache(3)
>>> cache['A'] = 0
>>> cache['B'] = 1
>>> cache['C'] = 2
>>> len(cache)
3
```

```
>>> cache['A']
0
```

Adding new items to the cache does not increase its size. Instead, the least recently used item is dropped:

```
>>> cache['D'] = 3
>>> len(cache)
3
>>> 'B' in cache
False
```

Iterating over the cache returns the keys, starting with the most recently used:

```
>>> for key in cache:
...     print(key)
D
A
C
```

This code is based on the LRUCache class from `myghtyutils.util`, written by Mike Bayer and released under the MIT license. See:

http://svn.myghty.org/myghtyutils/trunk/lib/myghtyutils/util.py

`genshi.util.`**`flatten`**(*items*)

Flattens a potentially nested sequence into a flat list.

Parameters **`items`** – the sequence to flatten

```
>>> flatten((1, 2))
[1, 2]
>>> flatten([1, (2, 3), 4])
[1, 2, 3, 4]
>>> flatten([1, (2, [3, 4]), 5])
[1, 2, 3, 4, 5]
```

`genshi.util.`**`plaintext`**(*text*, *keeplinebreaks=True*)

Return the text with all entities and tags removed.

```
>>> plaintext('<b>1 &lt; 2</b>')
u'1 < 2'
```

The *keeplinebreaks* parameter can be set to `False` to replace any line breaks by simple spaces:

```
>>> plaintext('''<b>1
... &lt;
... 2</b>''', keeplinebreaks=False)
u'1 < 2'
```

Parameters

- **`text`** – the text to convert to plain text
- **`keeplinebreaks`** – whether line breaks in the text should be kept intact

Returns the text with tags and entities removed

`genshi.util.`**`stripentities`**(*text*, *keepxmlentities=False*)

Return a copy of the given text with any character or numeric entities replaced by the equivalent UTF-8 characters.

```
>>> stripentities('1 &lt; 2')
u'1 < 2'
>>> stripentities('more &hellip;')
u'more \u2026'
>>> stripentities('&#8230;')
u'\u2026'
>>> stripentities('&#x2026;')
u'\u2026'
```

If the *keepxmlentities* parameter is provided and is a truth value, the core XML entities (&, ', >, < and ") are left intact.

```
>>> stripentities('1 &lt; 2 &hellip;', keepxmlentities=True)
u'1 &lt; 2 \u2026'
```

`genshi.util.`**`striptags`**(*text*)

Return a copy of the text with any XML/HTML tags removed.

```
>>> striptags('<span>Foo</span> bar')
'Foo bar'
>>> striptags('<span class="bar">Foo</span>')
'Foo'
>>> striptags('Foo<br />')
'Foo'
```

HTML/XML comments are stripped, too:

```
>>> striptags('<!-- <blub>hehe</blah> -->test')
'test'
```

Parameters **text** – the string to remove tags from

Returns the text with tags removed

b

c

f

i

o

p

t

u

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X