Mako Documentation

Release 1.0.7

Mike Bayer

Sep 28, 2017

Contents

CHAPTER 1

Usage

Basic Usage

This section describes the Python API for Mako templates. If you are using Mako within a web framework such as Pylons, the work of integrating Mako's API is already done for you, in which case you can skip to the next section, *Syntax*.

The most basic way to create a template and render it is through the `Template` class:

```
from mako.template import Template

mytemplate = Template("hello world!")
print(mytemplate.render())
```

Above, the text argument to `Template` is **compiled** into a Python module representation. This module contains a function called `render_body()`, which produces the output of the template. When `mytemplate.render()` is called, Mako sets up a runtime environment for the template and calls the `render_body()` function, capturing the output into a buffer and returning its string contents.

The code inside the `render_body()` function has access to a namespace of variables. You can specify these variables by sending them as additional keyword arguments to the `render()` method:

```
from mako.template import Template

mytemplate = Template("hello, ${name}!")
print(mytemplate.render(name="jack"))
```

The `render()` method calls upon Mako to create a `Context` object, which stores all the variable names accessible to the template and also stores a buffer used to capture output. You can create this `Context` yourself and have the template render with it, using the `render_context()` method:

```
from mako.template import Template
from mako.runtime import Context
from StringIO import StringIO
```

```
mytemplate = Template("hello, ${name}!")
buf = StringIO()
ctx = Context(buf, name="jack")
mytemplate.render_context(ctx)
print(buf.getvalue())
```

Using File-Based Templates

A *Template* can also load its template source code from a file, using the `filename` keyword argument:

```
from mako.template import Template

mytemplate = Template(filename='/docs/mytmpl.txt')
print(mytemplate.render())
```

For improved performance, a *Template* which is loaded from a file can also cache the source code to its generated module on the filesystem as a regular Python module file (i.e. a `.py` file). To do this, just add the `module_directory` argument to the template:

```
from mako.template import Template

mytemplate = Template(filename='/docs/mytmpl.txt', module_directory='/tmp/mako_modules
↪')
print(mytemplate.render())
```

When the above code is rendered, a file `/tmp/mako_modules/docs/mytmpl.txt.py` is created containing the source code for the module. The next time a *Template* with the same arguments is created, this module file will be automatically re-used.

Using `TemplateLookup`

All of the examples thus far have dealt with the usage of a single *Template* object. If the code within those templates tries to locate another template resource, it will need some way to find them, using simple URI strings. For this need, the resolution of other templates from within a template is accomplished by the *TemplateLookup* class. This class is constructed given a list of directories in which to search for templates, as well as keyword arguments that will be passed to the *Template* objects it creates:

```
from mako.template import Template
from mako.lookup import TemplateLookup

mylookup = TemplateLookup(directories=['/docs'])
mytemplate = Template("""<%include file="header.txt"/> hello world!""",␣
↪lookup=mylookup)
```

Above, we created a textual template which includes the file `"header.txt"`. In order for it to have somewhere to look for `"header.txt"`, we passed a *TemplateLookup* object to it, which will search in the directory `/docs` for the file `"header.txt"`.

Usually, an application will store most or all of its templates as text files on the filesystem. So far, all of our examples have been a little bit contrived in order to illustrate the basic concepts. But a real application would get most or all of its templates directly from the *TemplateLookup*, using the aptly named *get_template()* method, which accepts the URI of the desired template:

```
from mako.template import Template
from mako.lookup import TemplateLookup

mylookup = TemplateLookup(directories=['/docs'], module_directory='/tmp/mako_modules')

def serve_template(templatename, **kwargs):
    mytemplate = mylookup.get_template(templatename)
    print(mytemplate.render(**kwargs))
```

In the example above, we create a `TemplateLookup` which will look for templates in the `/docs` directory, and will store generated module files in the `/tmp/mako_modules` directory. The lookup locates templates by appending the given URI to each of its search directories; so if you gave it a URI of `/etc/beans/info.txt`, it would search for the file `/docs/etc/beans/info.txt`, else raise a `TopLevelNotFound` exception, which is a custom Mako exception.

When the lookup locates templates, it will also assign a `uri` property to the `Template` which is the URI passed to the `get_template()` call. `Template` uses this URI to calculate the name of its module file. So in the above example, a `templatename` argument of `/etc/beans/info.txt` will create a module file `/tmp/mako_modules/etc/beans/info.txt.py`.

Setting the Collection Size

The `TemplateLookup` also serves the important need of caching a fixed set of templates in memory at a given time, so that successive URI lookups do not result in full template compilations and/or module reloads on each request. By default, the `TemplateLookup` size is unbounded. You can specify a fixed size using the `collection_size` argument:

```
mylookup = TemplateLookup(directories=['/docs'],
                module_directory='/tmp/mako_modules', collection_size=500)
```

The above lookup will continue to load templates into memory until it reaches a count of around 500. At that point, it will clean out a certain percentage of templates using a least recently used scheme.

Setting Filesystem Checks

Another important flag on `TemplateLookup` is `filesystem_checks`. This defaults to `True`, and says that each time a template is returned by the `get_template()` method, the revision time of the original template file is checked against the last time the template was loaded, and if the file is newer will reload its contents and recompile the template. On a production system, setting `filesystem_checks` to `False` can afford a small to moderate performance increase (depending on the type of filesystem used).

Using Unicode and Encoding

Both `Template` and `TemplateLookup` accept `output_encoding` and `encoding_errors` parameters which can be used to encode the output in any Python supported codec:

```
from mako.template import Template
from mako.lookup import TemplateLookup

mylookup = TemplateLookup(directories=['/docs'], output_encoding='utf-8', encoding_
↪errors='replace')
```

```
mytemplate = mylookup.get_template("foo.txt")
print(mytemplate.render())
```

When using Python 3, the *render()* method will return a `bytes` object, **if** `output_encoding` is set. Otherwise it returns a `string`.

Additionally, the *render_unicode()* method exists which will return the template output as a Python `unicode` object, or in Python 3 a `string`:

```
print(mytemplate.render_unicode())
```

The above method disregards the output encoding keyword argument; you can encode yourself by saying:

```
print(mytemplate.render_unicode().encode('utf-8', 'replace'))
```

Note that Mako's ability to return data in any encoding and/or `unicode` implies that the underlying output stream of the template is a Python unicode object. This behavior is described fully in *The Unicode Chapter*.

Handling Exceptions

Template exceptions can occur in two distinct places. One is when you **lookup, parse and compile** the template, the other is when you **run** the template. Within the running of a template, exceptions are thrown normally from whatever Python code originated the issue. Mako has its own set of exception classes which mostly apply to the lookup and lexer/compiler stages of template construction. Mako provides some library routines that can be used to help provide Mako-specific information about any exception's stack trace, as well as formatting the exception within textual or HTML format. In all cases, the main value of these handlers is that of converting Python filenames, line numbers, and code samples into Mako template filenames, line numbers, and code samples. All lines within a stack trace which correspond to a Mako template module will be converted to be against the originating template file.

To format exception traces, the *text_error_template()* and *html_error_template()* functions are provided. They make usage of `sys.exc_info()` to get at the most recently thrown exception. Usage of these handlers usually looks like:

```
from mako import exceptions

try:
    template = lookup.get_template(uri)
    print(template.render())
except:
    print(exceptions.text_error_template().render())
```

Or for the HTML render function:

```
from mako import exceptions

try:
    template = lookup.get_template(uri)
    print(template.render())
except:
    print(exceptions.html_error_template().render())
```

The *html_error_template()* template accepts two options: specifying `full=False` causes only a section of an HTML document to be rendered. Specifying `css=False` will disable the default stylesheet from being rendered.

E.g.:

```
print(exceptions.html_error_template().render(full=False))
```

The HTML render function is also available built-in to *Template* using the `format_exceptions` flag. In this case, any exceptions raised within the **render** stage of the template will result in the output being substituted with the output of *html_error_template()*:

```
template = Template(filename="/foo/bar", format_exceptions=True)
print(template.render())
```

Note that the compile stage of the above template occurs when you construct the *Template* itself, and no output stream is defined. Therefore exceptions which occur within the lookup/parse/compile stage will not be handled and will propagate normally. While the pre-render traceback usually will not include any Mako-specific lines anyway, it will mean that exceptions which occur previous to rendering and those which occur within rendering will be handled differently... so the `try/except` patterns described previously are probably of more general use.

The underlying object used by the error template functions is the *RichTraceback* object. This object can also be used directly to provide custom error views. Here's an example usage which describes its general API:

```
from mako.exceptions import RichTraceback

try:
    template = lookup.get_template(uri)
    print(template.render())
except:
    traceback = RichTraceback()
    for (filename, lineno, function, line) in traceback.traceback:
        print("File %s, line %s, in %s" % (filename, lineno, function))
        print(line, "\n")
    print("%s: %s" % (str(traceback.error.__class__.__name__), traceback.error))
```

Common Framework Integrations

The Mako distribution includes a little bit of helper code for the purpose of using Mako in some popular web framework scenarios. This is a brief description of what's included.

WSGI

A sample WSGI application is included in the distribution in the file `examples/wsgi/run_wsgi.py`. This runner is set up to pull files from a *templates* as well as an *htdocs* directory and includes a rudimental two-file layout. The WSGI runner acts as a fully functional standalone web server, using `wsgiutils` to run itself, and propagates GET and POST arguments from the request into the *Context*, can serve images, CSS files and other kinds of files, and also displays errors using Mako's included exception-handling utilities.

Pygments

A Pygments-compatible syntax highlighting module is included under `mako.ext.pygmentplugin`. This module is used in the generation of Mako documentation and also contains various *setuptools* entry points under the heading `pygments.lexers`, including `mako`, `html+mako`, `xml+mako` (see the `setup.py` file for all the entry points).

Babel

Mako provides support for extracting *gettext* messages from templates via a Babel extractor entry point under `mako.ext.babelplugin`.

Gettext messages are extracted from all Python code sections, including those of control lines and expressions embedded in tags.

Translator comments may also be extracted from Mako templates when a comment tag is specified to Babel (such as with the `-c` option).

For example, a project `"myproj"` contains the following Mako template at `myproj/myproj/templates/name.html`:

```
<div id="name">
  Name:
  ## TRANSLATORS: This is a proper name. See the gettext
  ## manual, section Names.
  ${_('Francois Pinard')}
</div>
```

To extract gettext messages from this template the project needs a Mako section in its Babel Extraction Method Mapping file (typically located at `myproj/babel.cfg`):

```
# Extraction from Python source files

[python: myproj/**.py]

# Extraction from Mako templates

[mako: myproj/templates/**.html]
input_encoding = utf-8
```

The Mako extractor supports an optional `input_encoding` parameter specifying the encoding of the templates (identical to *`Template`*/*`TemplateLookup`*'s `input_encoding` parameter).

Invoking Babel's extractor at the command line in the project's root directory:

```
myproj$ pybabel extract -F babel.cfg -c "TRANSLATORS:" .
```

will output a *gettext* catalog to *stdout* including the following:

```
#. TRANSLATORS: This is a proper name. See the gettext
#. manual, section Names.
#: myproj/templates/name.html:5
msgid "Francois Pinard"
msgstr ""
```

This is only a basic example: Babel can be invoked from `setup.py` and its command line options specified in the accompanying `setup.cfg` via Babel Distutils/Setuptools Integration.

Comments must immediately precede a *gettext* message to be extracted. In the following case the `TRANSLATORS:` comment would not have been extracted:

```
<div id="name">
  ## TRANSLATORS: This is a proper name. See the gettext
  ## manual, section Names.
  Name: ${_('Francois Pinard')}
</div>
```

See the Babel User Guide for more information.

API Reference

class `mako.template.`**`Template`**(*text=None*, *filename=None*, *uri=None*, *format_exceptions=False*, *error_handler=None*, *lookup=None*, *output_encoding=None*, *encoding_errors='strict'*, *module_directory=None*, *cache_args=None*, *cache_impl='beaker'*, *cache_enabled=True*, *cache_type=None*, *cache_dir=None*, *cache_url=None*, *module_filename=None*, *input_encoding=None*, *disable_unicode=False*, *module_writer=None*, *bytestring_passthrough=False*, *default_filters=None*, *buffer_filters=()*, *strict_undefined=False*, *imports=None*, *future_imports=None*, *enable_loop=True*, *preprocessor=None*, *lexer_cls=None*, *include_error_handler=None*)

Bases: `object`

Represents a compiled template.

`Template` includes a reference to the original template source (via the *`source`* attribute) as well as the source code of the generated Python module (i.e. the *`code`* attribute), as well as a reference to an actual Python module.

`Template` is constructed using either a literal string representing the template text, or a filename representing a filesystem path to a source file.

Parameters

- **`text`** – textual template source. This argument is mutually exclusive versus the `filename` parameter.
- **`filename`** – filename of the source template. This argument is mutually exclusive versus the `text` parameter.
- **`buffer_filters`** – string list of filters to be applied to the output of `%defs` which are buffered, cached, or otherwise filtered, after all filters defined with the `%def` itself have been applied. Allows the creation of default expression filters that let the output of return-valued `%defs` "opt out" of that filtering via passing special attributes or objects.
- **`bytestring_passthrough`** – When `True`, and `output_encoding` is set to `None`, and *`Template.render()`* is used to render, the *StringIO* or *cStringIO* buffer will be used instead of the default "fast" buffer. This allows raw bytestrings in the output stream, such as in expressions, to pass straight through to the buffer. This flag is forced to `True` if `disable_unicode` is also configured.

  New in version 0.4: Added to provide the same behavior as that of the previous series.
- **`cache_args`** – Dictionary of cache configuration arguments that will be passed to the *`CacheImpl`*. See *Caching*.
- **`cache_dir`** – Deprecated since version 0.6: Use the `'dir'` argument in the `cache_args` dictionary. See *Caching*.
- **`cache_enabled`** – Boolean flag which enables caching of this template. See *Caching*.
- **`cache_impl`** – String name of a *`CacheImpl`* caching implementation to use. Defaults to `'beaker'`.
- **`cache_type`** – Deprecated since version 0.6: Use the `'type'` argument in the `cache_args` dictionary. See *Caching*.

- **cache_url** – Deprecated since version 0.6: Use the `'url'` argument in the `cache_args` dictionary. See *Caching*.
- **default_filters** – List of string filter names that will be applied to all expressions. See *The default_filters Argument*.
- **disable_unicode** – Disables all awareness of Python Unicode objects. See *Saying to Heck with It: Disabling the Usage of Unicode Entirely*.
- **enable_loop** – When `True`, enable the `loop` context variable. This can be set to `False` to support templates that may be making usage of the name "`loop`". Individual templates can re-enable the "loop" context by placing the directive `enable_loop="True"` inside the `<%page>` tag – see *Migrating Legacy Templates that Use the Word "loop"*.
- **encoding_errors** – Error parameter passed to `encode()` when string encoding is performed. See *Using Unicode and Encoding*.
- **error_handler** – Python callable which is called whenever compile or runtime exceptions occur. The callable is passed the current context as well as the exception. If the callable returns `True`, the exception is considered to be handled, else it is re-raised after the function completes. Is used to provide custom error-rendering functions.

  See also:

  `Template.include_error_handler` - include-specific error handler function
- **format_exceptions** – if `True`, exceptions which occur during the render phase of this template will be caught and formatted into an HTML error page, which then becomes the rendered result of the *`render()`* call. Otherwise, runtime exceptions are propagated outwards.
- **imports** – String list of Python statements, typically individual "import" lines, which will be placed into the module level preamble of all generated Python modules. See the example in *The default_filters Argument*.
- **future_imports** – String list of names to import from *__future__*. These will be concatenated into a comma-separated string and inserted into the beginning of the template, e.g. `futures_imports=['FOO', 'BAR']` results in `from __future__ import FOO, BAR`. If you're interested in using features like the new division operator, you must use future_imports to convey that to the renderer, as otherwise the import will not appear as the first executed statement in the generated code and will therefore not have the desired effect.
- **include_error_handler** – An error handler that runs when this template is included within another one via the `<%include>` tag, and raises an error. Compare to the *`Template.error_handler`* option.

  New in version 1.0.6.

  See also:

  `Template.error_handler` - top-level error handler function
- **input_encoding** – Encoding of the template's source code. Can be used in lieu of the coding comment. See *Using Unicode and Encoding* as well as *The Unicode Chapter* for details on source encoding.
- **lookup** – a *`TemplateLookup`* instance that will be used for all file lookups via the `<%namespace>`, `<%include>`, and `<%inherit>` tags. See *Using TemplateLookup*.
- **module_directory** – Filesystem location where generated Python module files will be placed.

- **module_filename** – Overrides the filename of the generated Python module file. For advanced usage only.
- **module_writer** – A callable which overrides how the Python module is written entirely. The callable is passed the encoded source content of the module and the destination path to be written to. The default behavior of module writing uses a tempfile in conjunction with a file move in order to make the operation atomic. So a user-defined module writing function that mimics the default behavior would be:

  ```
  import tempfile
  import os
  import shutil

  def module_writer(source, outputpath):
      (dest, name) = \\
          tempfile.mkstemp(
              dir=os.path.dirname(outputpath)
          )

      os.write(dest, source)
      os.close(dest)
      shutil.move(name, outputpath)

  from mako.template import Template
  mytemplate = Template(
                  filename="index.html",
                  module_directory="/path/to/modules",
                  module_writer=module_writer
              )
  ```

  The function is provided for unusual configurations where certain platform-specific permissions or other special steps are needed.
- **output_encoding** – The encoding to use when `render()` is called. See *Using Unicode and Encoding* as well as *The Unicode Chapter*.
- **preprocessor** – Python callable which will be passed the full template source before it is parsed. The return result of the callable will be used as the template source code.
- **lexer_cls** – A `Lexer` class used to parse the template. The `Lexer` class is used by default.

  New in version 0.7.4.
- **strict_undefined** – Replaces the automatic usage of `UNDEFINED` for any undeclared variables not located in the `Context` with an immediate raise of `NameError`. The advantage is immediate reporting of missing variables which include the name.

  New in version 0.3.6.
- **uri** – string URI or other identifier for this template. If not provided, the `uri` is generated from the filesystem path, or from the in-memory identity of a non-file-based template. The primary usage of the `uri` is to provide a key within `TemplateLookup`, as well as to generate the file path of the generated Python module file, if `module_directory` is specified.

code

Return the module source code for this `Template`.

get_def(*name*)

Return a def of this template as a `DefTemplate`.

list_defs()
return a list of defs in the template.

New in version 1.0.4.

render(**args*, ***data*)
Render the output of this template as a string.

If the template specifies an output encoding, the string will be encoded accordingly, else the output is raw (raw output uses *cStringIO* and can't handle multibyte characters). A `Context` object is created corresponding to the given data. Arguments that are explicitly declared by this template's internal rendering method are also pulled from the given `*args`, `**data` members.

render_context(*context*, **args*, ***kwargs*)
Render this `Template` with the given context.

The data is written to the context's buffer.

render_unicode(**args*, ***data*)
Render the output of this template as a unicode object.

source
Return the template source code for this `Template`.

class `mako.template.`**DefTemplate**(*parent*, *callable_*)
Bases: `mako.template.Template`

A `Template` which represents a callable def in a parent template.

class `mako.lookup.`**TemplateCollection**
Bases: `object`

Represent a collection of `Template` objects, identifiable via URI.

A `TemplateCollection` is linked to the usage of all template tags that address other templates, such as `<%include>`, `<%namespace>`, and `<%inherit>`. The `file` attribute of each of those tags refers to a string URI that is passed to that `Template` object's `TemplateCollection` for resolution.

`TemplateCollection` is an abstract class, with the usual default implementation being `TemplateLookup`.

adjust_uri(*uri*, *filename*)
Adjust the given `uri` based on the calling `filename`.

When this method is called from the runtime, the `filename` parameter is taken directly to the `filename` attribute of the calling template. Therefore a custom `TemplateCollection` subclass can place any string identifier desired in the `filename` parameter of the `Template` objects it constructs and have them come back here.

filename_to_uri(*uri*, *filename*)
Convert the given `filename` to a URI relative to this `TemplateCollection`.

get_template(*uri*, *relativeto=None*)
Return a `Template` object corresponding to the given `uri`.

The default implementation raises `NotImplementedError`. Implementations should raise `TemplateLookupException` if the given `uri` cannot be resolved.

Parameters

- **uri** – String URI of the template to be resolved.
- **relativeto** – if present, the given `uri` is assumed to be relative to this URI.

has_template(*uri*)
Return `True` if this *TemplateLookup* is capable of returning a *Template* object for the given `uri`.

Parameters **uri** – String URI of the template to be resolved.

class mako.lookup.**TemplateLookup**(*directories=None*, *module_directory=None*, *filesystem_checks=True*, *collection_size=-1*, *format_exceptions=False*, *error_handler=None*, *disable_unicode=False*, *bytestring_passthrough=False*, *output_encoding=None*, *encoding_errors='strict'*, *cache_args=None*, *cache_impl='beaker'*, *cache_enabled=True*, *cache_type=None*, *cache_dir=None*, *cache_url=None*, *modulename_callable=None*, *module_writer=None*, *default_filters=None*, *buffer_filters=()*, *strict_undefined=False*, *imports=None*, *future_imports=None*, *enable_loop=True*, *input_encoding=None*, *preprocessor=None*, *lexer_cls=None*, *include_error_handler=None*)
Bases: *mako.lookup.TemplateCollection*

Represent a collection of templates that locates template source files from the local filesystem.

The primary argument is the `directories` argument, the list of directories to search:

```
lookup = TemplateLookup(["/path/to/templates"])
some_template = lookup.get_template("/index.html")
```

The *TemplateLookup* can also be given *Template* objects programatically using *put_string()* or *put_template()*:

```
lookup = TemplateLookup()
lookup.put_string("base.html", '''
    <html><body>${self.next()}</body></html>
''')
lookup.put_string("hello.html", '''
    <%include file='base.html'/>

    Hello, world !
''')
```

Parameters

- **directories** – A list of directory names which will be searched for a particular template URI. The URI is appended to each directory and the filesystem checked.
- **collection_size** – Approximate size of the collection used to store templates. If left at its default of `-1`, the size is unbounded, and a plain Python dictionary is used to relate URI strings to *Template* instances. Otherwise, a least-recently-used cache object is used which will maintain the size of the collection approximately to the number given.
- **filesystem_checks** – When at its default value of `True`, each call to *TemplateLookup.get_template()* will compare the filesystem last modified time to the time in which an existing *Template* object was created. This allows the *TemplateLookup* to regenerate a new *Template* whenever the original source has been updated. Set this to `False` for a very minor performance increase.
- **modulename_callable** – A callable which, when present, is passed the path of the source file as well as the requested URI, and then returns the full path of the generated Python module file. This is used to inject alternate schemes for Python module location. If left at its default of `None`, the built in system of generation based on `module_directory` plus `uri` is used.

All other keyword parameters available for `Template` are mirrored here. When new `Template` objects are created, the keywords established with this `TemplateLookup` are passed on to each new `Template`.

`adjust_uri`(*uri*, *relativeto*)
Adjust the given `uri` based on the given relative URI.

`filename_to_uri`(*filename*)
Convert the given `filename` to a URI relative to this `TemplateCollection`.

`get_template`(*uri*)
Return a `Template` object corresponding to the given `uri`.

Note: The `relativeto` argument is not supported here at the moment.

`put_string`(*uri*, *text*)
Place a new `Template` object into this `TemplateLookup`, based on the given string of `text`.

`put_template`(*uri*, *template*)
Place a new `Template` object into this `TemplateLookup`, based on the given `Template` object.

class `mako.exceptions.`**`RichTraceback`**(*error=None*, *traceback=None*)
Bases: `object`

Pull the current exception from the `sys` traceback and extracts Mako-specific template information.

See the usage examples in *Handling Exceptions*.

`error`
the exception instance.

`message`
the exception error message as unicode.

`source`
source code of the file where the error occurred. If the error occurred within a compiled template, this is the template source.

`lineno`
line number where the error occurred. If the error occurred within a compiled template, the line number is adjusted to that of the template source.

`records`
a list of 8-tuples containing the original python traceback elements, plus the filename, line number, source line, and full template source for the traceline mapped back to its originating source template, if any for that traceline (else the fields are `None`).

`reverse_records`
the list of records in reverse traceback – a list of 4-tuples, in the same format as a regular python traceback, with template-corresponding traceback records replacing the originals.

`reverse_traceback`
the traceback list in reverse.

`mako.exceptions.`**`html_error_template`**()
Provides a template that renders a stack trace in an HTML format, providing an excerpt of code as well as substituting source template filenames, line numbers and code for that of the originating source template, as applicable.

The template's default `encoding_errors` value is `'htmlentityreplace'`. The template has two options. With the `full` option disabled, only a section of an HTML document is returned. With the `css` option disabled, the default stylesheet won't be included.

`mako.exceptions.`**`text_error_template`**`(`*lookup=None*`)`

Provides a template that renders a stack trace in a similar format to the Python interpreter, substituting source template filenames, line numbers and code for that of the originating source template, as applicable.

CHAPTER 2

Syntax

A Mako template is parsed from a text stream containing any kind of content, XML, HTML, email text, etc. The template can further contain Mako-specific directives which represent variable and/or expression substitutions, control structures (i.e. conditionals and loops), server-side comments, full blocks of Python code, as well as various tags that offer additional functionality. All of these constructs compile into real Python code. This means that you can leverage the full power of Python in almost every aspect of a Mako template.

Expression Substitution

The simplest expression is just a variable substitution. The syntax for this is the `${}` construct, which is inspired by Perl, Genshi, JSP EL, and others:

```
this is x: ${x}
```

Above, the string representation of `x` is applied to the template's output stream. If you're wondering where `x` comes from, it's usually from the *`Context`* supplied to the template's rendering function. If `x` was not supplied to the template and was not otherwise assigned locally, it evaluates to a special value `UNDEFINED`. More on that later.

The contents within the `${}` tag are evaluated by Python directly, so full expressions are OK:

```
pythagorean theorem:  ${pow(x,2) + pow(y,2)}
```

The results of the expression are evaluated into a string result in all cases before being rendered to the output stream, such as the above example where the expression produces a numeric result.

Expression Escaping

Mako includes a number of built-in escaping mechanisms, including HTML, URI and XML escaping, as well as a "trim" function. These escapes can be added to an expression substitution using the `|` operator:

```
${"this is some text" | u}
```

The above expression applies URL escaping to the expression, and produces `this+is+some+text`. The `u` name indicates URL escaping, whereas `h` represents HTML escaping, `x` represents XML escaping, and `trim` applies a trim function.

Read more about built-in filtering functions, including how to make your own filter functions, in *Filtering and Buffering*.

Control Structures

A control structure refers to all those things that control the flow of a program – conditionals (i.e. `if`/`else`), loops (like `while` and `for`), as well as things like `try`/`except`. In Mako, control structures are written using the `%` marker followed by a regular Python control expression, and are "closed" by using another `%` marker with the tag "`end<name>`", where "`<name>`" is the keyword of the expression:

```
% if x==5:
    this is some output
% endif
```

The `%` can appear anywhere on the line as long as no text precedes it; indentation is not significant. The full range of Python "colon" expressions are allowed here, including `if`/`elif`/`else`, `while`, `for`, and even `def`, although Mako has a built-in tag for defs which is more full-featured.

```
% for a in ['one', 'two', 'three', 'four', 'five']:
    % if a[0] == 't':
    its two or three
    % elif a[0] == 'f':
    four/five
    % else:
    one
    % endif
% endfor
```

The `%` sign can also be "escaped", if you actually want to emit a percent sign as the first non whitespace character on a line, by escaping it as in `%%`:

```
%% some text

    %% some more text
```

The Loop Context

The **loop context** provides additional information about a loop while inside of a `% for` structure:

```
<ul>
% for a in ("one", "two", "three"):
    <li>Item ${loop.index}: ${a}</li>
% endfor
</ul>
```

See *The Loop Context* for more information on this feature.

New in version 0.7.

Comments

Comments come in two varieties. The single line comment uses `##` as the first non-space characters on a line:

```
## this is a comment.
...text ...
```

A multiline version exists using `<%doc> ...text... </%doc>`:

```
<%doc>
    these are comments
    more comments
</%doc>
```

Newline Filters

The backslash ("\") character, placed at the end of any line, will consume the newline character before continuing to the next line:

```
here is a line that goes onto \
another line.
```

The above text evaluates to:

```
here is a line that goes onto another line.
```

Python Blocks

Any arbitrary block of python can be dropped in using the `<% %>` tags:

```
this is a template
<%
    x = db.get_resource('foo')
    y = [z.element for z in x if x.frobnizzle==5]
%>
% for elem in y:
    element: ${elem}
% endfor
```

Within `<% %>`, you're writing a regular block of Python code. While the code can appear with an arbitrary level of preceding whitespace, it has to be consistently formatted with itself. Mako's compiler will adjust the block of Python to be consistent with the surrounding generated Python code.

Module-level Blocks

A variant on `<% %>` is the module-level code block, denoted by `<%! %>`. Code within these tags is executed at the module level of the template, and not within the rendering function of the template. Therefore, this code does not have access to the template's context and is only executed when the template is loaded into memory (which can be only once per application, or more, depending on the runtime environment). Use the `<%! %>` tags to declare your template's imports, as well as any pure-Python functions you might want to declare:

```
<%!
    import mylib
    import re

    def filter(text):
        return re.sub(r'^@', '', text)
%>
```

Any number of `<%! %>` blocks can be declared anywhere in a template; they will be rendered in the resulting module in a single contiguous block above all render callables, in the order in which they appear in the source template.

Tags

The rest of what Mako offers takes place in the form of tags. All tags use the same syntax, which is similar to an XML tag except that the first character of the tag name is a `%` character. The tag is closed either by a contained slash character, or an explicit closing tag:

```
<%include file="foo.txt"/>

<%def name="foo" buffered="True">
    this is a def
</%def>
```

All tags have a set of attributes which are defined for each tag. Some of these attributes are required. Also, many attributes support **evaluation**, meaning you can embed an expression (using `${}`) inside the attribute text:

```
<%include file="/foo/bar/${myfile}.txt"/>
```

Whether or not an attribute accepts runtime evaluation depends on the type of tag and how that tag is compiled into the template. The best way to find out if you can stick an expression in is to try it! The lexer will tell you if it's not valid.

Heres a quick summary of all the tags:

`<%page>`

This tag defines general characteristics of the template, including caching arguments, and optional lists of arguments which the template expects when invoked.

```
<%page args="x, y, z='default'"/>
```

Or a page tag that defines caching characteristics:

```
<%page cached="True" cache_type="memory"/>
```

Currently, only one `<%page>` tag gets used per template, the rest get ignored. While this will be improved in a future release, for now make sure you have only one `<%page>` tag defined in your template, else you may not get the results you want. The details of what `<%page>` is used for are described further in *The body() Method* as well as *Caching*.

`<%include>`

A tag that is familiar from other template languages, `%include` is a regular joe that just accepts a file argument and calls in the rendered result of that file:

```
<%include file="header.html"/>

    hello world

<%include file="footer.html"/>
```

Include also accepts arguments which are available as `<%page>` arguments in the receiving template:

```
<%include file="toolbar.html" args="current_section='members', username='ed'"/>
```

<%def>

The `%def` tag defines a Python function which contains a set of content, that can be called at some other point in the template. The basic idea is simple:

```
<%def name="myfunc(x)">
    this is myfunc, x is ${x}
</%def>

${myfunc(7)}
```

The `%def` tag is a lot more powerful than a plain Python `def`, as the Mako compiler provides many extra services with `%def` that you wouldn't normally have, such as the ability to export defs as template "methods", automatic propagation of the current *`Context`*, buffering/filtering/caching flags, and def calls with content, which enable packages of defs to be sent as arguments to other def calls (not as hard as it sounds). Get the full deal on what `%def` can do in *Defs and Blocks*.

<%block>

`%block` is a tag that is close to a `%def`, except executes itself immediately in its base-most scope, and can also be anonymous (i.e. with no name):

```
<%block filter="h">
    some <html> stuff.
</%block>
```

Inspired by Jinja2 blocks, named blocks offer a syntactically pleasing way to do inheritance:

```
<html>
    <body>
    <%block name="header">
        <h2><%block name="title"/></h2>
    </%block>
    ${self.body()}
    </body>
</html>
```

Blocks are introduced in *Using Blocks* and further described in *Inheritance*.

New in version 0.4.1.

`<%namespace>`

`%namespace` is Mako's equivalent of Python's `import` statement. It allows access to all the rendering functions and metadata of other template files, plain Python modules, as well as locally defined "packages" of functions.

```
<%namespace file="functions.html" import="*"/>
```

The underlying object generated by `%namespace`, an instance of *`mako.runtime.Namespace`*, is a central construct used in templates to reference template-specific information such as the current URI, inheritance structures, and other things that are not as hard as they sound right here. Namespaces are described in *Namespaces*.

`<%inherit>`

Inherit allows templates to arrange themselves in **inheritance chains**. This is a concept familiar in many other template languages.

```
<%inherit file="base.html"/>
```

When using the `%inherit` tag, control is passed to the topmost inherited template first, which then decides how to handle calling areas of content from its inheriting templates. Mako offers a lot of flexibility in this area, including dynamic inheritance, content wrapping, and polymorphic method calls. Check it out in *Inheritance*.

`<%nsname:defname>`

Any user-defined "tag" can be created against a namespace by using a tag with a name of the form `<%<namespacename>:<defname>>`. The closed and open formats of such a tag are equivalent to an inline expression and the `<%call>` tag, respectively.

```
<%mynamespace:somedef param="some value">
    this is the body
</%mynamespace:somedef>
```

To create custom tags which accept a body, see *Calling a Def with Embedded Content and/or Other Defs*.

New in version 0.2.3.

`<%call>`

The call tag is the "classic" form of a user-defined tag, and is roughly equivalent to the `<%namespacename:defname>` syntax described above. This tag is also described in *Calling a Def with Embedded Content and/or Other Defs*.

`<%doc>`

The `%doc` tag handles multiline comments:

```
<%doc>
    these are comments
    more comments
</%doc>
```

Also the `##` symbol as the first non-space characters on a line can be used for single line comments.

`<%text>`

This tag suspends the Mako lexer's normal parsing of Mako template directives, and returns its entire body contents as plain text. It is used pretty much to write documentation about Mako:

```
<%text filter="h">
    heres some fake mako ${syntax}
    <%def name="x()">${x}</%def>
</%text>
```

Exiting Early from a Template

Sometimes you want to stop processing a template or `<%def>` method in the middle and just use the text you've accumulated so far. This is accomplished by using `return` statement inside a Python block. It's a good idea for the `return` statement to return an empty string, which prevents the Python default return value of `None` from being rendered by the template. This return value is for semantic purposes provided in templates via the `STOP_RENDERING` symbol:

```
% if not len(records):
    No records found.
    <% return STOP_RENDERING %>
% endif
```

Or perhaps:

```
<%
    if not len(records):
        return STOP_RENDERING
%>
```

In older versions of Mako, an empty string can be substituted for the `STOP_RENDERING` symbol:

```
<% return '' %>
```

New in version 1.0.2: - added the `STOP_RENDERING` symbol which serves as a semantic identifier for the empty string `""` used by a Python `return` statement.

CHAPTER 3

Defs and Blocks

`<%def>` and `<%block>` are two tags that both demarcate any block of text and/or code. They both exist within generated Python as a callable function, i.e., a Python `def`. They differ in their scope and calling semantics. Whereas `<%def>` provides a construct that is very much like a named Python `def`, the `<%block>` is more layout oriented.

Using Defs

The `<%def>` tag requires a `name` attribute, where the `name` references a Python function signature:

```
<%def name="hello()">
    hello world
</%def>
```

To invoke the `<%def>`, it is normally called as an expression:

```
the def:  ${hello()}
```

If the `<%def>` is not nested inside of another `<%def>`, it's known as a **top level def** and can be accessed anywhere in the template, including above where it was defined.

All defs, top level or not, have access to the current contextual namespace in exactly the same way their containing template does. Suppose the template below is executed with the variables `username` and `accountdata` inside the context:

```
Hello there ${username}, how are ya.  Lets see what your account says:

${account()}

<%def name="account()">
    Account for ${username}:<br/>

    % for row in accountdata:
        Value: ${row}<br/>
    % endfor
</%def>
```

The `username` and `accountdata` variables are present within the main template body as well as the body of the `account()` def.

Since defs are just Python functions, you can define and pass arguments to them as well:

```
${account(accountname='john')}

<%def name="account(accountname, type='regular')">
    account name: ${accountname}, type: ${type}
</%def>
```

When you declare an argument signature for your def, they are required to follow normal Python conventions (i.e., all arguments are required except keyword arguments with a default value). This is in contrast to using context-level variables, which evaluate to `UNDEFINED` if you reference a name that does not exist.

Calling Defs from Other Files

Top level `<%def>`s are **exported** by your template's module, and can be called from the outside; including from other templates, as well as normal Python code. Calling a `<%def>` from another template is something like using an `<%include>` – except you are calling a specific function within the template, not the whole template.

The remote `<%def>` call is also a little bit like calling functions from other modules in Python. There is an "import" step to pull the names from another template into your own template; then the function or functions are available.

To import another template, use the `<%namespace>` tag:

```
<%namespace name="mystuff" file="mystuff.html"/>
```

The above tag adds a local variable `mystuff` to the current scope.

Then, just call the defs off of `mystuff`:

```
${mystuff.somedef(x=5,y=7)}
```

The `<%namespace>` tag also supports some of the other semantics of Python's `import` statement, including pulling names into the local variable space, or using `*` to represent all names, using the `import` attribute:

```
<%namespace file="mystuff.html" import="foo, bar"/>
```

This is just a quick intro to the concept of a **namespace**, which is a central Mako concept that has its own chapter in these docs. For more detail and examples, see *Namespaces*.

Calling Defs Programmatically

You can call defs programmatically from any *`Template`* object using the *`get_def()`* method, which returns a *`DefTemplate`* object. This is a *`Template`* subclass which the parent *`Template`* creates, and is usable like any other template:

```
from mako.template import Template

template = Template("""
    <%def name="hi(name)">
        hi ${name}!
    </%def>

    <%def name="bye(name)">
        bye ${name}!
    </%def>
""")

print(template.get_def("hi").render(name="ed"))
print(template.get_def("bye").render(name="ed"))
```

Defs within Defs

The def model follows regular Python rules for closures. Declaring `<%def>` inside another `<%def>` declares it within the parent's **enclosing scope**:

```
<%def name="mydef()">
    <%def name="subdef()">
        a sub def
    </%def>

    i'm the def, and the subcomponent is ${subdef()}
</%def>
```

Just like Python, names that exist outside the inner `<%def>` exist inside it as well:

```
<%
    x = 12
%>
<%def name="outer()">
    <%
        y = 15
    %>
    <%def name="inner()">
        inner, x is ${x}, y is ${y}
    </%def>

    outer, x is ${x}, y is ${y}
</%def>
```

Assigning to a name inside of a def declares that name as local to the scope of that def (again, like Python itself). This means the following code will raise an error:

```
<%
    x = 10
%>
<%def name="somedef()">
    ## error !
    somedef, x is ${x}
    <%
        x = 27
    %>
</%def>
```

...because the assignment to `x` declares `x` as local to the scope of `somedef`, rendering the "outer" version unreachable in the expression that tries to render it.

Calling a Def with Embedded Content and/or Other Defs

A flip-side to def within def is a def call with content. This is where you call a def, and at the same time declare a block of content (or multiple blocks) that can be used by the def being called. The main point of such a call is to create custom, nestable tags, just like any other template language's custom-tag creation system – where the external tag controls the execution of the nested tags and can communicate state to them. Only with Mako, you don't have to use any external Python modules, you can define arbitrarily nestable tags right in your templates.

To achieve this, the target def is invoked using the form `<%namespacename:defname>` instead of the normal `${}` syntax. This syntax, introduced in Mako 0.2.3, is functionally equivalent to another tag known as `%call`, which takes the form `<%call expr='namespacename.defname(args)'>`. While `%call` is available in all versions of Mako, the newer style is probably more familiar looking. The `namespace` portion of the call is the name of the **namespace** in which the def is defined – in the most simple cases, this can be `local` or `self` to reference the current template's namespace (the difference between `local` and `self` is one of inheritance – see *Built-in Namespaces* for details).

When the target def is invoked, a variable `caller` is placed in its context which contains another namespace containing the body and other defs defined by the caller. The body itself is referenced by the method `body()`. Below, we build a `%def` that operates upon `caller.body()` to invoke the body of the custom tag:

```
<%def name="buildtable()">
    <table>
        <tr><td>
            ${caller.body()}
        </td></tr>
    </table>
</%def>

<%self:buildtable>
    I am the table body.
</%self:buildtable>
```

This produces the output (whitespace formatted):

```
<table>
    <tr><td>
        I am the table body.
    </td></tr>
</table>
```

Using the older `%call` syntax looks like:

```
<%def name="buildtable()">
    <table>
        <tr><td>
            ${caller.body()}
        </td></tr>
    </table>
</%def>

<%call expr="buildtable()">
    I am the table body.
</%call>
```

The `body()` can be executed multiple times or not at all. This means you can use def-call-with-content to build iterators, conditionals, etc:

```
<%def name="lister(count)">
    % for x in range(count):
        ${caller.body()}
    % endfor
</%def>

<%self:lister count="${3}">
    hi
</%self:lister>
```

Produces:

```
hi
hi
hi
```

Notice above we pass `3` as a Python expression, so that it remains as an integer.

A custom "conditional" tag:

```
<%def name="conditional(expression)">
    % if expression:
        ${caller.body()}
    % endif
</%def>

<%self:conditional expression="${4==4}">
    i'm the result
</%self:conditional>
```

Produces:

```
i'm the result
```

But that's not all. The `body()` function also can handle arguments, which will augment the local namespace of the body callable. The caller must define the arguments which it expects to receive from its target def using the `args` attribute, which is a comma-separated list of argument names. Below, our `<%def>` calls the `body()` of its caller, passing in an element of data from its argument:

```
<%def name="layoutdata(somedata)">
    <table>
    % for item in somedata:
        <tr>
        % for col in item:
            <td>${caller.body(col=col)}</td>
        % endfor
        </tr>
    % endfor
    </table>
</%def>

<%self:layoutdata somedata="${[[1,2,3],[4,5,6],[7,8,9]]}" args="col">\
Body data: ${col}\
</%self:layoutdata>
```

Produces:

```
<table>
    <tr>
        <td>Body data: 1</td>
        <td>Body data: 2</td>
        <td>Body data: 3</td>
    </tr>
    <tr>
        <td>Body data: 4</td>
        <td>Body data: 5</td>
        <td>Body data: 6</td>
    </tr>
    <tr>
        <td>Body data: 7</td>
        <td>Body data: 8</td>
        <td>Body data: 9</td>
    </tr>
</table>
```

You don't have to stick to calling just the `body()` function. The caller can define any number of callables, allowing the `<%call>` tag to produce whole layouts:

```
<%def name="layout()">
    ## a layout def
    <div class="mainlayout">
        <div class="header">
            ${caller.header()}
        </div>

        <div class="sidebar">
            ${caller.sidebar()}
        </div>

        <div class="content">
            ${caller.body()}
        </div>
    </div>
</%def>

## calls the layout def
<%self:layout>
    <%def name="header()">
        I am the header
    </%def>
    <%def name="sidebar()">
        <ul>
            <li>sidebar 1</li>
            <li>sidebar 2</li>
        </ul>
    </%def>

        this is the body
</%self:layout>
```

The above layout would produce:

```
<div class="mainlayout">
    <div class="header">
    I am the header
    </div>

    <div class="sidebar">
    <ul>
        <li>sidebar 1</li>
        <li>sidebar 2</li>
    </ul>
    </div>

    <div class="content">
    this is the body
    </div>
</div>
```

The number of things you can do with `<%call>` and/or the `<%namespacename:defname>` calling syntax is enormous. You can create form widget libraries, such as an enclosing `<FORM>` tag and nested HTML input elements, or portable wrapping schemes using `<div>` or other elements. You can create tags that interpret rows of data, such as from a database, providing the individual columns of each row to a `body()` callable which lays out the row any way it wants. Basically anything you'd do with a "custom tag" or tag library in some other system, Mako provides via `<%def>` tags and plain Python callables which are invoked via `<%namespacename:defname>` or `<%call>`.

Using Blocks

The `<%block>` tag introduces some new twists on the `<%def>` tag which make it more closely tailored towards layout.

New in version 0.4.1.

An example of a block:

```
<html>
    <body>
        <%block>
            this is a block.
        </%block>
    </body>
</html>
```

In the above example, we define a simple block. The block renders its content in the place that it's defined. Since the block is called for us, it doesn't need a name and the above is referred to as an **anonymous block**. So the output of the above template will be:

```
<html>
    <body>
            this is a block.
    </body>
</html>
```

So in fact the above block has absolutely no effect. Its usefulness comes when we start using modifiers. Such as, we can apply a filter to our block:

```
<html>
    <body>
        <%block filter="h">
            <html>this is some escaped html.</html>
        </%block>
    </body>
</html>
```

or perhaps a caching directive:

```
<html>
    <body>
        <%block cached="True" cache_timeout="60">
            This content will be cached for 60 seconds.
        </%block>
    </body>
</html>
```

Blocks also work in iterations, conditionals, just like defs:

```
% if some_condition:
    <%block>condition is met</%block>
% endif
```

While the block renders at the point it is defined in the template, the underlying function is present in the generated Python code only once, so there's no issue with placing a block inside of a loop or similar. Anonymous blocks are defined as closures in the local rendering body, so have access to local variable scope:

```
% for i in range(1, 4):
    <%block>i is ${i}</%block>
% endfor
```

Using Named Blocks

Possibly the more important area where blocks are useful is when we do actually give them names. Named blocks are tailored to behave somewhat closely to Jinja2's block tag, in that they define an area of a layout which can be overridden by an inheriting template. In sharp contrast to the `<%def>` tag, the name given to a block is global for the entire template regardless of how deeply it's nested:

```
<html>
<%block name="header">
    <head>
        <title>
            <%block name="title">Title</%block>
        </title>
    </head>
</%block>
<body>
    ${next.body()}
</body>
</html>
```

The above example has two named blocks "`header`" and "`title`", both of which can be referred to by an inheriting template. A detailed walkthrough of this usage can be found at *Inheritance*.

Note above that named blocks don't have any argument declaration the way defs do. They still implement themselves as Python functions, however, so they can be invoked additional times beyond their initial definition:

```
<div name="page">
    <%block name="pagecontrol">
        <a href="">previous page</a> |
        <a href="">next page</a>
    </%block>

    <table>
        ## some content
    </table>

    ${pagecontrol()}
</div>
```

The content referenced by `pagecontrol` above will be rendered both above and below the `<table>` tags.

To keep things sane, named blocks have restrictions that defs do not:

- The `<%block>` declaration cannot have any argument signature.
- The name of a `<%block>` can only be defined once in a template – an error is raised if two blocks of the same name occur anywhere in a single template, regardless of nesting. A similar error is raised if a top level def shares the same name as that of a block.
- A named `<%block>` cannot be defined within a `<%def>`, or inside the body of a "call", i.e. `<%call>` or `<%namespacename:defname>` tag. Anonymous blocks can, however.

Using Page Arguments in Named Blocks

A named block is very much like a top level def. It has a similar restriction to these types of defs in that arguments passed to the template via the `<%page>` tag aren't automatically available. Using arguments with the `<%page>` tag is described in the section *The body() Method*, and refers to scenarios such as when the `body()` method of a template is called from an inherited template passing arguments, or the template is invoked from an `<%include>` tag with arguments. To allow a named block to share the same arguments passed to the page, the `args` attribute can be used:

```
<%page args="post"/>

<a name="${post.title}" />

<span class="post_prose">
    <%block name="post_prose" args="post">
        ${post.content}
    </%block>
</span>
```

Where above, if the template is called via a directive like `<%include file="post.mako" args="post=post" />`, the `post` variable is available both in the main body as well as the `post_prose` block.

Similarly, the `**pageargs` variable is present, in named blocks only, for those arguments not explicit in the `<%page>` tag:

```
<%block name="post_prose">
    ${pageargs['post'].content}
</%block>
```

The `args` attribute is only allowed with named blocks. With anonymous blocks, the Python function is always rendered in the same scope as the call itself, so anything available directly outside the anonymous block is available inside as well.

CHAPTER 4

The Mako Runtime Environment

This section describes a little bit about the objects and built-in functions that are available in templates.

Context

The `Context` is the central object that is created when a template is first executed, and is responsible for handling all communication with the outside world. Within the template environment, it is available via the *reserved name* `context`. The `Context` includes two major components, one of which is the output buffer, which is a file-like object such as Python's `StringIO` or similar, and the other a dictionary of variables that can be freely referenced within a template; this dictionary is a combination of the arguments sent to the `render()` function and some built-in variables provided by Mako's runtime environment.

The Buffer

The buffer is stored within the `Context`, and writing to it is achieved by calling the `write()` method – in a template this looks like `context.write('some string')`. You usually don't need to care about this, as all text within a template, as well as all expressions provided by `${}`, automatically send everything to this method. The cases you might want to be aware of its existence are if you are dealing with various filtering/buffering scenarios, which are described in *Filtering and Buffering*, or if you want to programmatically send content to the output stream, such as within a `<% %>` block.

```
<%
    context.write("some programmatic text")
%>
```

The actual buffer may or may not be the original buffer sent to the `Context` object, as various filtering/caching scenarios may "push" a new buffer onto the context's underlying buffer stack. For this reason, just stick with `context.write()` and content will always go to the topmost buffer.

Context Variables

When your template is compiled into a Python module, the body content is enclosed within a Python function called `render_body`. Other top-level defs defined in the template are defined within their own function bodies which are named after the def's name with the prefix `render_` (i.e. `render_mydef`). One of the first things that happens within these functions is that all variable names that are referenced within the function which are not defined in some other way (i.e. such as via assignment, module level imports, etc.) are pulled from the `Context` object's dictionary of variables. This is how you're able to freely reference variable names in a template which automatically correspond to what was passed into the current `Context`.

- **What happens if I reference a variable name that is not in the current context?** - The value you get back is a special value called `UNDEFINED`, or if the `strict_undefined=True` flag is used a `NameError` is raised. `UNDEFINED` is just a simple global variable with the class `mako.runtime.Undefined`. The `UNDEFINED` object throws an error when you call `str()` on it, which is what happens if you try to use it in an expression.
- **UNDEFINED makes it hard for me to find what name is missing** - An alternative is to specify the option `strict_undefined=True` to the `Template` or `TemplateLookup`. This will cause any non-present variables to raise an immediate `NameError` which includes the name of the variable in its message when `render()` is called – `UNDEFINED` is not used.

  New in version 0.3.6.
- **Why not just return None?** Using `UNDEFINED`, or raising a `NameError` is more explicit and allows differentiation between a value of `None` that was explicitly passed to the `Context` and a value that wasn't present at all.
- **Why raise an exception when you call str() on it ? Why not just return a blank string?** - Mako tries to stick to the Python philosophy of "explicit is better than implicit". In this case, it's decided that the template author should be made to specifically handle a missing value rather than experiencing what may be a silent failure. Since `UNDEFINED` is a singleton object just like Python's `True` or `False`, you can use the `is` operator to check for it:

```
% if someval is UNDEFINED:
    someval is: no value
% else:
    someval is: ${someval}
% endif
```

Another facet of the `Context` is that its dictionary of variables is **immutable**. Whatever is set when `render()` is called is what stays. Of course, since its Python, you can hack around this and change values in the context's internal dictionary, but this will probably will not work as well as you'd think. The reason for this is that Mako in many cases creates copies of the `Context` object, which get sent to various elements of the template and inheriting templates used in an execution. So changing the value in your local `Context` will not necessarily make that value available in other parts of the template's execution. Examples of where Mako creates copies of the `Context` include within top-level def calls from the main body of the template (the context is used to propagate locally assigned variables into the scope of defs; since in the template's body they appear as inlined functions, Mako tries to make them act that way), and within an inheritance chain (each template in an inheritance chain has a different notion of `parent` and `next`, which are all stored in unique `Context` instances).

- **So what if I want to set values that are global to everyone within a template request?** - All you have to do is provide a dictionary to your `Context` when the template first runs, and everyone can just get/set variables from that. Lets say its called `attributes`.

  Running the template looks like:

```
output = template.render(attributes={})
```

  Within a template, just reference the dictionary:

```
<%
    attributes['foo'] = 'bar'
%>
'foo' attribute is: ${attributes['foo']}
```

- **Why can't "attributes" be a built-in feature of the Context?** - This is an area where Mako is trying to make as few decisions about your application as it possibly can. Perhaps you don't want your templates to use this technique of assigning and sharing data, or perhaps you have a different notion of the names and kinds of data structures that should be passed around. Once again Mako would rather ask the user to be explicit.

Context Methods and Accessors

Significant members of *Context* include:

- `context[key]` / `context.get(key, default=None)` - dictionary-like accessors for the context. Normally, any variable you use in your template is automatically pulled from the context if it isn't defined somewhere already. Use the dictionary accessor and/or `get` method when you want a variable that *is* already defined somewhere else, such as in the local arguments sent to a `%def` call. If a key is not present, like a dictionary it raises `KeyError`.
- `keys()` - all the names defined within this context.
- `kwargs` - this returns a **copy** of the context's dictionary of variables. This is useful when you want to propagate the variables in the current context to a function as keyword arguments, i.e.:

```
${next.body(**context.kwargs)}
```

- `write(text)` - write some text to the current output stream.
- `lookup` - returns the *TemplateLookup* instance that is used for all file-lookups within the current execution (even though individual *Template* instances can conceivably have different instances of a *TemplateLookup*, only the *TemplateLookup* of the originally-called *Template* gets used in a particular execution).

The Loop Context

Within `% for` blocks, the *reserved name* `loop` is available. `loop` tracks the progress of the `for` loop and makes it easy to use the iteration state to control template behavior:

```
<ul>
% for a in ("one", "two", "three"):
    <li>Item ${loop.index}: ${a}</li>
% endfor
</ul>
```

New in version 0.7.

Iterations

Regardless of the type of iterable you're looping over, `loop` always tracks the 0-indexed iteration count (available at `loop.index`), its parity (through the `loop.even` and `loop.odd` bools), and `loop.first`, a bool indicating whether the loop is on its first iteration. If your iterable provides a `__len__` method, `loop` also provides access to a count of iterations remaining at `loop.reverse_index` and `loop.last`, a bool indicating whether the loop is on its last iteration; accessing these without `__len__` will raise a `TypeError`.

Cycling

Cycling is available regardless of whether the iterable you're using provides a `__len__` method. Prior to Mako 0.7, you might have generated a simple zebra striped list using `enumerate`:

```
<ul>
% for i, item in enumerate(('spam', 'ham', 'eggs')):
  <li class="${'odd' if i % 2 else 'even'}">${item}</li>
% endfor
</ul>
```

With `loop.cycle`, you get the same results with cleaner code and less prep work:

```
<ul>
% for item in ('spam', 'ham', 'eggs'):
  <li class="${loop.cycle('even', 'odd')}">${item}</li>
% endfor
</ul>
```

Both approaches produce output like the following:

```
<ul>
  <li class="even">spam</li>
  <li class="odd">ham</li>
  <li class="even">eggs</li>
</ul>
```

Parent Loops

Loop contexts can also be transparently nested, and the Mako runtime will do the right thing and manage the scope for you. You can access the parent loop context through `loop.parent`.

This allows you to reach all the way back up through the loop stack by chaining `parent` attribute accesses, i.e. `loop.parent.parent....` as long as the stack depth isn't exceeded. For example, you can use the parent loop to make a checkered table:

```
<table>
% for consonant in 'pbj':
  <tr>
  % for vowel in 'iou':
    <td class="${'black' if (loop.parent.even == loop.even) else 'red'}">
      ${consonant + vowel}t
    </td>
  % endfor
  </tr>
% endfor
</table>
```

```
<table>
  <tr>
    <td class="black">
      pit
    </td>
    <td class="red">
      pot
    </td>
    <td class="black">
      put
    </td>
  </tr>
  <tr>
    <td class="red">
      bit
    </td>
    <td class="black">
      bot
    </td>
    <td class="red">
      but
    </td>
  </tr>
  <tr>
    <td class="black">
      jit
    </td>
    <td class="red">
      jot
    </td>
    <td class="black">
      jut
    </td>
  </tr>
</table>
```

Migrating Legacy Templates that Use the Word "loop"

Changed in version 0.7: The `loop` name is now *reserved* in Mako, which means a template that refers to a variable named `loop` won't function correctly when used in Mako 0.7.

To ease the transition for such systems, the feature can be disabled across the board for all templates, then re-enabled on a per-template basis for those templates which wish to make use of the new system.

First, the `enable_loop=False` flag is passed to either the *`TemplateLookup`* or *`Template`* object in use:

```
lookup = TemplateLookup(directories=['/docs'], enable_loop=False)
```

or:

```
template = Template("some template", enable_loop=False)
```

An individual template can make usage of the feature when `enable_loop` is set to `False` by switching it back on within the `<%page>` tag:

```
<%page enable_loop="True"/>

% for i in collection:
    ${i} ${loop.index}
% endfor
```

Using the above scheme, it's safe to pass the name `loop` to the *`Template.render()`* method as well as to freely make usage of a variable named `loop` within a template, provided the `<%page>` tag doesn't override it. New templates that want to use the `loop` context can then set up `<%page enable_loop="True"/>` to use the new feature without affecting old templates.

All the Built-in Names

A one-stop shop for all the names Mako defines. Most of these names are instances of *`Namespace`*, which are described in the next section, *Namespaces*. Also, most of these names other than `context`, `UNDEFINED`, and `loop` are also present *within* the *`Context`* itself. The names `context`, `loop` and `UNDEFINED` themselves can't be passed to the context and can't be substituted – see the section *Reserved Names*.

- `context` - this is the *`Context`* object, introduced at *Context*.
- `local` - the namespace of the current template, described in *Built-in Namespaces*.
- `self` - the namespace of the topmost template in an inheritance chain (if any, otherwise the same as `local`), mostly described in *Inheritance*.
- `parent` - the namespace of the parent template in an inheritance chain (otherwise undefined); see *Inheritance*.
- `next` - the namespace of the next template in an inheritance chain (otherwise undefined); see *Inheritance*.
- `caller` - a "mini" namespace created when using the `<%call>` tag to define a "def call with content"; described in *Calling a Def with Embedded Content and/or Other Defs*.
- `loop` - this provides access to *`LoopContext`* objects when they are requested within `% for` loops, introduced at *The Loop Context*.
- `capture` - a function that calls a given def and captures its resulting content into a string, which is returned. Usage is described in *Filtering and Buffering*.
- `UNDEFINED` - a global singleton that is applied to all otherwise uninitialized template variables that were not located within the *`Context`* when rendering began, unless the *`Template`* flag `strict_undefined` is set to `True`. `UNDEFINED` is an instance of *`Undefined`*, and raises an exception when its `__str__()` method is called.
- `pageargs` - this is a dictionary which is present in a template which does not define any `**kwargs` section in its `<%page>` tag. All keyword arguments sent to the `body()` function of a template (when used via namespaces) go here by default unless otherwise defined as a page argument. If this makes no sense, it shouldn't; read the section *The body() Method*.

Reserved Names

Mako has a few names that are considered to be "reserved" and can't be used as variable names.

Changed in version 0.7: Mako raises an error if these words are found passed to the template as context arguments, whereas in previous versions they'd be silently ignored or lead to other error messages.

- `context` - see *Context*.
- `UNDEFINED` - see *Context Variables*.
- `loop` - see *The Loop Context*. Note this can be disabled for legacy templates via the `enable_loop=False` argument; see *Migrating Legacy Templates that Use the Word "loop"*.

API Reference

class `mako.runtime.`**`Context`**(*buffer*, ***data*)

Bases: `object`

Provides runtime namespace, output buffer, and various callstacks for templates.

See *The Mako Runtime Environment* for detail on the usage of `Context`.

`get`(*key*, *default=None*)
Return a value from this `Context`.

`keys`()
Return a list of all names established in this `Context`.

`kwargs`
Return the dictionary of top level keyword arguments associated with this `Context`.

This dictionary only includes the top-level arguments passed to `Template.render()`. It does not include names produced within the template execution such as local variable names or special names such as `self`, `next`, etc.

The purpose of this dictionary is primarily for the case that a `Template` accepts arguments via its `<%page>` tag, which are normally expected to be passed via `Template.render()`, except the template is being called in an inheritance context, using the `body()` method. `Context.kwargs` can then be used to propagate these arguments to the inheriting template:

```
${next.body(**context.kwargs)}
```

`lookup`
Return the `TemplateLookup` associated with this `Context`.

`pop_caller`()
Pop a `caller` callable onto the callstack for this `Context`.

`push_caller`(*caller*)
Push a `caller` callable onto the callstack for this `Context`.

`write`(*string*)
Write a string to this `Context` object's underlying output buffer.

`writer`()
Return the current writer function.

class `mako.runtime.`**`LoopContext`**(*iterable*)

Bases: `object`

A magic loop variable. Automatically accessible in any `% for` block.

See the section *The Loop Context* for usage notes.

`parent` -> `LoopContext` or `None` The parent loop, if one exists.

`index` -> *int* The 0-based iteration count.

`reverse_index` -> *int* The number of iterations remaining.

`first` -> *bool* `True` on the first iteration, `False` otherwise.

`last` -> *bool* `True` on the last iteration, `False` otherwise.

`even` -> *bool* `True` when `index` is even.

`odd` -> *bool* `True` when `index` is odd.

`cycle``(`**values*`)`
Cycle through values as the loop progresses.

class `mako.runtime.`**`Undefined`**
Bases: `object`

Represents an undefined value in a template.

All template modules have a constant value `UNDEFINED` present which is an instance of this object.

CHAPTER 5

Namespaces

Namespaces are used to organize groups of defs into categories, and also to "import" defs from other files.

If the file `components.html` defines these two defs:

```
## components.html
<%def name="comp1()">
    this is comp1
</%def>

<%def name="comp2(x)">
    this is comp2, x is ${x}
</%def>
```

you can make another file, for example `index.html`, that pulls those two defs into a namespace called `comp`:

```
## index.html
<%namespace name="comp" file="components.html"/>

Here's comp1:  ${comp.comp1()}
Here's comp2:  ${comp.comp2(x=5)}
```

The `comp` variable above is an instance of *`Namespace`*, a **proxy object** which delivers method calls to the underlying template callable using the current context.

`<%namespace>` also provides an `import` attribute which can be used to pull the names into the local namespace, removing the need to call it via the "`.`" operator. When `import` is used, the `name` attribute is optional.

```
<%namespace file="components.html" import="comp1, comp2"/>

Heres comp1:  ${comp1()}
Heres comp2:  ${comp2(x=5)}
```

`import` also supports the "`*`" operator:

```
<%namespace file="components.html" import="*"/>

Heres comp1:  ${comp1()}
Heres comp2:  ${comp2(x=5)}
```

The names imported by the `import` attribute take precedence over any names that exist within the current context.

Note: In current versions of Mako, usage of `import='*'` is known to decrease performance of the template. This will be fixed in a future release.

The `file` argument allows expressions – if looking for context variables, the `context` must be named explicitly:

```
<%namespace name="dyn" file="${context['namespace_name']}"/>
```

Ways to Call Namespaces

There are essentially four ways to call a function from a namespace.

The "expression" format, as described previously. Namespaces are just Python objects with functions on them, and can be used in expressions like any other function:

```
${mynamespace.somefunction('some arg1', 'some arg2', arg3='some arg3', arg4='some arg4
↪')}
```

Synonymous with the "expression" format is the "custom tag" format, when a "closed" tag is used. This format, introduced in Mako 0.2.3, allows the usage of a "custom" Mako tag, with the function arguments passed in using named attributes:

```
<%mynamespace:somefunction arg1="some arg1" arg2="some arg2" arg3="some arg3" arg4=
↪"some arg4"/>
```

When using tags, the values of the arguments are taken as literal strings by default. To embed Python expressions as arguments, use the embedded expression format:

```
<%mynamespace:somefunction arg1="${someobject.format()}" arg2="${somedef(5, 12)}"/>
```

The "custom tag" format is intended mainly for namespace functions which recognize body content, which in Mako is known as a "def with embedded content":

```
<%mynamespace:somefunction arg1="some argument" args="x, y">
    Some record: ${x}, ${y}
</%mynamespace:somefunction>
```

The "classic" way to call defs with embedded content is the `<%call>` tag:

```
<%call expr="mynamespace.somefunction(arg1='some argument')" args="x, y">
    Some record: ${x}, ${y}
</%call>
```

For information on how to construct defs that embed content from the caller, see *Calling a Def with Embedded Content and/or Other Defs*.

Namespaces from Regular Python Modules

Namespaces can also import regular Python functions from modules. These callables need to take at least one argument, `context`, an instance of *`Context`*. A module file `some/module.py` might contain the callable:

```
def my_tag(context):
    context.write("hello world")
    return ''
```

A template can use this module via:

```
<%namespace name="hw" module="some.module"/>

${hw.my_tag()}
```

Note that the `context` argument is not needed in the call; the *`Namespace`* tag creates a locally-scoped callable which takes care of it. The `return ''` is so that the def does not dump a `None` into the output stream – the return value of any def is rendered after the def completes, in addition to whatever was passed to *`Context.write()`* within its body.

If your def is to be called in an "embedded content" context, that is as described in *Calling a Def with Embedded Content and/or Other Defs*, you should use the *`supports_caller()`* decorator, which will ensure that Mako will ensure the correct "caller" variable is available when your def is called, supporting embedded content:

```
from mako.runtime import supports_caller

@supports_caller
def my_tag(context):
    context.write("<div>")
    context['caller'].body()
    context.write("</div>")
    return ''
```

Capturing of output is available as well, using the outside-of-templates version of the *`capture()`* function, which accepts the "context" as its first argument:

```
from mako.runtime import supports_caller, capture

@supports_caller
def my_tag(context):
    return "<div>%s</div>" % \
            capture(context, context['caller'].body, x="foo", y="bar")
```

Declaring Defs in Namespaces

The `<%namespace>` tag supports the definition of `<%def>`s directly inside the tag. These defs become part of the namespace like any other function, and will override the definitions pulled in from a remote template or module:

```
## define a namespace
<%namespace name="stuff">
    <%def name="comp1()">
        comp1
    </%def>
</%namespace>
```

```
## then call it
${stuff.comp1()}
```

The `body()` Method

Every namespace that is generated from a template contains a method called `body()`. This method corresponds to the main body of the template, and plays its most important roles when using inheritance relationships as well as def-calls-with-content.

Since the `body()` method is available from a namespace just like all the other defs defined in a template, what happens if you send arguments to it? By default, the `body()` method accepts no positional arguments, and for usefulness in inheritance scenarios will by default dump all keyword arguments into a dictionary called `pageargs`. But if you actually want to get at the keyword arguments, Mako recommends you define your own argument signature explicitly. You do this via using the `<%page>` tag:

```
<%page args="x, y, someval=8, scope='foo', **kwargs"/>
```

A template which defines the above signature requires that the variables `x` and `y` are defined, defines default values for `someval` and `scope`, and sets up `**kwargs` to receive all other keyword arguments. If `**kwargs` or similar is not present, the argument `**pageargs` gets tacked on by Mako. When the template is called as a top-level template (i.e. via *`render()`*) or via the `<%include>` tag, the values for these arguments will be pulled from the `Context`. In all other cases, i.e. via calling the `body()` method, the arguments are taken as ordinary arguments from the method call. So above, the body might be called as:

```
${self.body(5, y=10, someval=15, delta=7)}
```

The *`Context`* object also supplies a *`kwargs`* accessor, for cases when you'd like to pass along the top level context arguments to a `body()` callable:

```
${next.body(**context.kwargs)}
```

The usefulness of calls like the above become more apparent when one works with inheriting templates. For more information on this, as well as the meanings of the names `self` and `next`, see *Inheritance*.

Built-in Namespaces

The namespace is so great that Mako gives your template one (or two) for free. The names of these namespaces are `local` and `self`. Other built-in namespaces include `parent` and `next`, which are optional and are described in *Inheritance*.

`local`

The `local` namespace is basically the namespace for the currently executing template. This means that all of the top level defs defined in your template, as well as your template's `body()` function, are also available off of the `local` namespace.

The `local` namespace is also where properties like `uri`, `filename`, and `module` and the `get_namespace` method can be particularly useful.

`self`

The `self` namespace, in the case of a template that does not use inheritance, is synonymous with `local`. If inheritance is used, then `self` references the topmost template in the inheritance chain, where it is most useful for providing the ultimate form of various "method" calls which may have been overridden at various points in an inheritance chain. See *Inheritance*.

Inheritable Namespaces

The `<%namespace>` tag includes an optional attribute `inheritable="True"`, which will cause the namespace to be attached to the `self` namespace. Since `self` is globally available throughout an inheritance chain (described in the next section), all the templates in an inheritance chain can get at the namespace imported in a super-template via `self`.

```
## base.html
<%namespace name="foo" file="foo.html" inheritable="True"/>

${next.body()}

## somefile.html
<%inherit file="base.html"/>

${self.foo.bar()}
```

This allows a super-template to load a whole bunch of namespaces that its inheriting templates can get to, without them having to explicitly load those namespaces themselves.

The `import="*"` part of the `<%namespace>` tag doesn't yet interact with the `inheritable` flag, so currently you have to use the explicit namespace name off of `self`, followed by the desired function name. But more on this in a future release.

Namespace API Usage Example - Static Dependencies

The `<%namespace>` tag at runtime produces an instance of *`Namespace`*. Programmatic access of *`Namespace`* can be used to build various kinds of scaffolding in templates and between templates.

A common request is the ability for a particular template to declare "static includes" - meaning, the usage of a particular set of defs requires that certain Javascript/CSS files are present. Using *`Namespace`* as the object that holds together the various templates present, we can build a variety of such schemes. In particular, the *`Context`* has a `namespaces` attribute, which is a dictionary of all *`Namespace`* objects declared. Iterating the values of this dictionary will provide a *`Namespace`* object for each time the `<%namespace>` tag was used, anywhere within the inheritance chain.

Version One - Use `Namespace.attr`

The *`Namespace.attr`* attribute allows us to locate any variables declared in the `<%! %>` of a template.

```
## base.mako
## base-most template, renders layout etc.
<html>
<head>
## traverse through all namespaces present,
## look for an attribute named 'includes'
```

```
% for ns in context.namespaces.values():
    % for incl in getattr(ns.attr, 'includes', []):
        ${incl}
    % endfor
% endfor
</head>
<body>
${next.body()}
</body
</html>

## library.mako
## library functions.
<%!
    includes = [
        '<link rel="stylesheet" type="text/css" href="mystyle.css"/>',
        '<script type="text/javascript" src="functions.js"></script>'
    ]
%>

<%def name="mytag()">
    <form>
        ${caller.body()}
    </form>
</%def>

## index.mako
## calling template.
<%inherit file="base.mako"/>
<%namespace name="foo" file="library.mako"/>

<%foo:mytag>
    a form
</%foo:mytag>
```

Above, the file `library.mako` declares an attribute `includes` inside its global `<%! %>` section. `index.mako` includes this template using the `<%namespace>` tag. The base template `base.mako`, which is the inherited parent of `index.mako` and is responsible for layout, then locates this attribute and iterates through its contents to produce the includes that are specific to `library.mako`.

Version Two - Use a specific named def

In this version, we put the includes into a `<%def>` that follows a naming convention.

```
## base.mako
## base-most template, renders layout etc.
<html>
<head>
## traverse through all namespaces present,
## look for a %def named 'includes'
% for ns in context.namespaces.values():
    % if hasattr(ns, 'includes'):
        ${ns.includes()}
    % endif
% endfor
</head>
```

```
<body>
${next.body()}
</body
</html>

## library.mako
## library functions.

<%def name="includes()">
    <link rel="stylesheet" type="text/css" href="mystyle.css"/>
    <script type="text/javascript" src="functions.js"></script>
</%def>

<%def name="mytag()">
    <form>
        ${caller.body()}
    </form>
</%def>


## index.mako
## calling template.
<%inherit file="base.mako"/>
<%namespace name="foo" file="library.mako"/>

<%foo:mytag>
    a form
</%foo:mytag>
```

In this version, `library.mako` declares a `<%def>` named `includes`. The example works identically to the previous one, except that `base.mako` looks for defs named `include` on each namespace it examines.

API Reference

class `mako.runtime.`**`Namespace`**(*name*, *context*, *callables=None*, *inherits=None*, *populate_self=True*, *calling_uri=None*)

Bases: `object`

Provides access to collections of rendering methods, which can be local, from other templates, or from imported modules.

To access a particular rendering method referenced by a *`Namespace`*, use plain attribute access:

```
${some_namespace.foo(x, y, z)}
```

`Namespace` also contains several built-in attributes described here.

`attr`

Access module level attributes by name.

This accessor allows templates to supply "scalar" attributes which are particularly handy in inheritance relationships.

See also:

Inheritable Attributes

Version One - Use Namespace.attr

`cache`
Return the *`Cache`* object referenced by this *`Namespace`* object's *`Template`*.

`context` = None
The *`Context`* object for this *`Namespace`*.

Namespaces are often created with copies of contexts that contain slightly different data, particularly in inheritance scenarios. Using the *`Context`* off of a *`Namespace`* one can traverse an entire chain of templates that inherit from one-another.

`filename` = None
The path of the filesystem file used for this *`Namespace`*'s module or template.

If this is a pure module-based *`Namespace`*, this evaluates to `module.__file__`. If a template-based namespace, it evaluates to the original template file location.

`get_cached`(*key*, ***kwargs*)
Return a value from the *`Cache`* referenced by this *`Namespace`* object's *`Template`*.

The advantage to this method versus direct access to the *`Cache`* is that the configuration parameters declared in `<%page>` take effect here, thereby calling up the same configured backend as that configured by `<%page>`.

`get_namespace`(*uri*)
Return a *`Namespace`* corresponding to the given `uri`.

If the given `uri` is a relative URI (i.e. it does not contain a leading slash `/`), the `uri` is adjusted to be relative to the `uri` of the namespace itself. This method is therefore mostly useful off of the built-in `local` namespace, described in *local*.

In most cases, a template wouldn't need this function, and should instead use the `<%namespace>` tag to load namespaces. However, since all `<%namespace>` tags are evaluated before the body of a template ever runs, this method can be used to locate namespaces using expressions that were generated within the body code of the template, or to conditionally use a particular namespace.

`get_template`(*uri*)
Return a *`Template`* from the given `uri`.

The `uri` resolution is relative to the `uri` of this *`Namespace`* object's *`Template`*.

`include_file`(*uri*, ***kwargs*)
Include a file at the given `uri`.

`module` = None
The Python module referenced by this *`Namespace`*.

If the namespace references a *`Template`*, then this module is the equivalent of `template.module`, i.e. the generated module for the template.

`template` = None
The *`Template`* object referenced by this *`Namespace`*, if any.

`uri` = None
The URI for this *`Namespace`*'s template.

I.e. whatever was sent to *`TemplateLookup.get_template()`*.

This is the equivalent of `Template.uri`.

class `mako.runtime.`**`TemplateNamespace`**(*name*, *context*, *template=None*, *templateuri=None*, *callables=None*, *inherits=None*, *populate_self=True*, *calling_uri=None*)
Bases: *`mako.runtime.Namespace`*

A `Namespace` specific to a `Template` instance.

`filename`
The path of the filesystem file used for this `Namespace`'s module or template.

`module`
The Python module referenced by this `Namespace`.

If the namespace references a `Template`, then this module is the equivalent of `template.module`, i.e. the generated module for the template.

`uri`
The URI for this `Namespace`'s template.

I.e. whatever was sent to `TemplateLookup.get_template()`.

This is the equivalent of `Template.uri`.

class `mako.runtime.`**`ModuleNamespace`**(*name*, *context*, *module*, *callables=None*, *inherits=None*, *populate_self=True*, *calling_uri=None*)
Bases: `mako.runtime.Namespace`

A `Namespace` specific to a Python module instance.

`filename`
The path of the filesystem file used for this `Namespace`'s module or template.

`mako.runtime.`**`supports_caller`**(*func*)
Apply a caller_stack compatibility decorator to a plain Python function.

See the example in *Namespaces from Regular Python Modules*.

`mako.runtime.`**`capture`**(*context*, *callable_*, **args*, ***kwargs*)
Execute the given template def, capturing the output into a buffer.

See the example in *Namespaces from Regular Python Modules*.

CHAPTER 6

Inheritance

Note: Most of the inheritance examples here take advantage of a feature that's new in Mako as of version 0.4.1 called the "block". This tag is very similar to the "def" tag but is more streamlined for usage with inheritance. Note that all of the examples here which use blocks can also use defs instead. Contrasting usages will be illustrated.

Using template inheritance, two or more templates can organize themselves into an **inheritance chain**, where content and functions from all involved templates can be intermixed. The general paradigm of template inheritance is this: if a template `A` inherits from template `B`, then template `A` agrees to send the executional control to template `B` at runtime (`A` is called the **inheriting** template). Template `B`, the **inherited** template, then makes decisions as to what resources from `A` shall be executed.

In practice, it looks like this. Here's a hypothetical inheriting template, `index.html`:

```
## index.html
<%inherit file="base.html"/>

<%block name="header">
    this is some header content
</%block>

this is the body content.
```

And `base.html`, the inherited template:

```
## base.html
<html>
    <body>
        <div class="header">
            <%block name="header"/>
        </div>

        ${self.body()}

        <div class="footer">
```

```
            <%block name="footer">
                this is the footer
            </%block>
        </div>
    </body>
</html>
```

Here is a breakdown of the execution:

1. When `index.html` is rendered, control immediately passes to `base.html`.
2. `base.html` then renders the top part of an HTML document, then invokes the `<%block name="header">` block. It invokes the underlying `header()` function off of a built-in namespace called `self` (this namespace was first introduced in the *Namespaces chapter* in *self*). Since `index.html` is the topmost template and also defines a block called `header`, it's this `header` block that ultimately gets executed – instead of the one that's present in `base.html`.
3. Control comes back to `base.html`. Some more HTML is rendered.
4. `base.html` executes `self.body()`. The `body()` function on all template-based namespaces refers to the main body of the template, therefore the main body of `index.html` is rendered.
5. When `<%block name="header">` is encountered in `index.html` during the `self.body()` call, a conditional is checked – does the current inherited template, i.e. `base.html`, also define this block? If yes, the `<%block>` is **not** executed here – the inheritance mechanism knows that the parent template is responsible for rendering this block (and in fact it already has). In other words a block only renders in its *basemost scope*.
6. Control comes back to `base.html`. More HTML is rendered, then the `<%block name="footer">` expression is invoked.
7. The `footer` block is only defined in `base.html`, so being the topmost definition of `footer`, it's the one that executes. If `index.html` also specified `footer`, then its version would **override** that of the base.
8. `base.html` finishes up rendering its HTML and the template is complete, producing:

```
<html>
    <body>
        <div class="header">
            this is some header content
        </div>

        this is the body content.

        <div class="footer">
            this is the footer
        </div>
    </body>
</html>
```

...and that is template inheritance in a nutshell. The main idea is that the methods that you call upon `self` always correspond to the topmost definition of that method. Very much the way `self` works in a Python class, even though Mako is not actually using Python class inheritance to implement this functionality. (Mako doesn't take the "inheritance" metaphor too seriously; while useful to setup some commonly recognized semantics, a textual template is not very much like an object-oriented class construct in practice).

Nesting Blocks

The named blocks defined in an inherited template can also be nested within other blocks. The name given to each block is globally accessible via any inheriting template. We can add a new block `title` to our `header` block:

```
## base.html
<html>
    <body>
        <div class="header">
            <%block name="header">
                <h2>
                    <%block name="title"/>
                </h2>
            </%block>
        </div>

        ${self.body()}

        <div class="footer">
            <%block name="footer">
                this is the footer
            </%block>
        </div>
    </body>
</html>
```

The inheriting template can name either or both of `header` and `title`, separately or nested themselves:

```
## index.html
<%inherit file="base.html"/>

<%block name="header">
    this is some header content
    ${parent.header()}
</%block>

<%block name="title">
    this is the title
</%block>

this is the body content.
```

Note when we overrode `header`, we added an extra call `${parent.header()}` in order to invoke the parent's `header` block in addition to our own. That's described in more detail below, in *Using the parent Namespace to Augment Defs*.

Rendering a Named Block Multiple Times

Recall from the section *Using Blocks* that a named block is just like a `<%def>`, with some different usage rules. We can call one of our named sections distinctly, for example a section that is used more than once, such as the title of a page:

```
<html>
    <head>
        <title>${self.title()}</title>
    </head>
    <body>
    <%block name="header">
        <h2><%block name="title"/></h2>
    </%block>
    ${self.body()}
    </body>
</html>
```

Where above an inheriting template can define `<%block name="title">` just once, and it will be used in the base template both in the `<title>` section as well as the `<h2>`.

But what about Defs?

The previous example used the `<%block>` tag to produce areas of content to be overridden. Before Mako 0.4.1, there wasn't any such tag – instead there was only the `<%def>` tag. As it turns out, named blocks and defs are largely interchangeable. The def simply doesn't call itself automatically, and has more open-ended naming and scoping rules that are more flexible and similar to Python itself, but less suited towards layout. The first example from this chapter using defs would look like:

```
## index.html
<%inherit file="base.html"/>

<%def name="header()">
    this is some header content
</%def>

this is the body content.
```

And `base.html`, the inherited template:

```
## base.html
<html>
    <body>
        <div class="header">
            ${self.header()}
        </div>

        ${self.body()}

        <div class="footer">
            ${self.footer()}
        </div>
    </body>
</html>

<%def name="header()"/>
<%def name="footer()">
    this is the footer
</%def>
```

Above, we illustrate that defs differ from blocks in that their definition and invocation are defined in two separate places, instead of at once. You can *almost* do exactly what a block does if you put the two together:

```
<div class="header">
    <%def name="header()"></%def>${self.header()}
</div>
```

The `<%block>` is obviously more streamlined than the `<%def>` for this kind of usage. In addition, the above "inline" approach with `<%def>` does not work with nesting:

```
<head>
    <%def name="header()">
        <title>
        ## this won't work !
        <%def name="title()">default title</%def>${self.title()}
        </title>
    </%def>${self.header()}
</head>
```

Where above, the `title()` def, because it's a def within a def, is not part of the template's exported namespace and will not be part of `self`. If the inherited template did define its own `title` def at the top level, it would be called, but the "default title" above is not present at all on `self` no matter what. For this to work as expected you'd instead need to say:

```
<head>
    <%def name="header()">
        <title>
        ${self.title()}
        </title>
    </%def>${self.header()}

    <%def name="title()"/>
</head>
```

That is, `title` is defined outside of any other defs so that it is in the `self` namespace. It works, but the definition needs to be potentially far away from the point of render.

A named block is always placed in the `self` namespace, regardless of nesting, so this restriction is lifted:

```
## base.html
<head>
    <%block name="header">
        <title>
        <%block name="title"/>
        </title>
    </%block>
</head>
```

The above template defines `title` inside of `header`, and an inheriting template can define one or both in **any** configuration, nested inside each other or not, in order for them to be used:

```
## index.html
<%inherit file="base.html"/>
<%block name="title">
    the title
</%block>
<%block name="header">
    the header
</%block>
```

So while the `<%block>` tag lifts the restriction of nested blocks not being available externally, in order to achieve this it *adds* the restriction that all block names in a single template need to be globally unique within the template, and additionally that a `<%block>` can't be defined inside of a `<%def>`. It's a more restricted tag suited towards a more specific use case than `<%def>`.

Using the `next` Namespace to Produce Content Wrapping

Sometimes you have an inheritance chain that spans more than two templates. Or maybe you don't, but you'd like to build your system such that extra inherited templates can be inserted in the middle of a chain where they would be smoothly integrated. If each template wants to define its layout just within its main body, you can't just call `self.body()` to get at the inheriting template's body, since that is only the topmost body. To get at the body of the *next* template, you call upon the namespace `next`, which is the namespace of the template **immediately following** the current template.

Lets change the line in `base.html` which calls upon `self.body()` to instead call upon `next.body()`:

```
## base.html
<html>
    <body>
        <div class="header">
            <%block name="header"/>
        </div>

        ${next.body()}

        <div class="footer">
            <%block name="footer">
                this is the footer
            </%block>
        </div>
    </body>
</html>
```

Lets also add an intermediate template called `layout.html`, which inherits from `base.html`:

```
## layout.html
<%inherit file="base.html"/>
<ul>
    <%block name="toolbar">
        <li>selection 1</li>
        <li>selection 2</li>
        <li>selection 3</li>
    </%block>
</ul>
<div class="mainlayout">
    ${next.body()}
</div>
```

And finally change `index.html` to inherit from `layout.html` instead:

```
## index.html
<%inherit file="layout.html"/>

## .. rest of template
```

In this setup, each call to `next.body()` will render the body of the next template in the inheritance chain (which can be written as `base.html -> layout.html -> index.html`). Control is still first passed to the bottommost template `base.html`, and `self` still references the topmost definition of any particular def.

The output we get would be:

```
<html>
    <body>
        <div class="header">
            this is some header content
        </div>

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

        <div class="mainlayout">
        this is the body content.
        </div>

        <div class="footer">
            this is the footer
        </div>
    </body>
</html>
```

So above, we have the `<html>`, `<body>` and `header`/`footer` layout of `base.html`, we have the `<ul>` and `mainlayout` section of `layout.html`, and the main body of `index.html` as well as its overridden `header` def. The `layout.html` template is inserted into the middle of the chain without `base.html` having to change anything. Without the `next` namespace, only the main body of `index.html` could be used; there would be no way to call `layout.html`'s body content.

Using the `parent` Namespace to Augment Defs

Lets now look at the other inheritance-specific namespace, the opposite of `next` called `parent`. `parent` is the namespace of the template **immediately preceding** the current template. What's useful about this namespace is that defs or blocks can call upon their overridden versions. This is not as hard as it sounds and is very much like using the `super` keyword in Python. Lets modify `index.html` to augment the list of selections provided by the `toolbar` function in `layout.html`:

```
## index.html
<%inherit file="layout.html"/>

<%block name="header">
    this is some header content
</%block>

<%block name="toolbar">
    ## call the parent's toolbar first
    ${parent.toolbar()}
    <li>selection 4</li>
    <li>selection 5</li>
</%block>

this is the body content.
```

Above, we implemented a `toolbar()` function, which is meant to override the definition of `toolbar` within the inherited template `layout.html`. However, since we want the content from that of `layout.html` as well, we call it via the `parent` namespace whenever we want it's content, in this case before we add our own selections. So the output for the whole thing is now:

```
<html>
    <body>
        <div class="header">
            this is some header content
        </div>

        <ul>
            <li>selection 1</li>
            <li>selection 2</li>
            <li>selection 3</li>
            <li>selection 4</li>
            <li>selection 5</li>
        </ul>

        <div class="mainlayout">
        this is the body content.
        </div>

        <div class="footer">
            this is the footer
        </div>
    </body>
</html>
```

and you're now a template inheritance ninja!

Using `<%include>` with Template Inheritance

A common source of confusion is the behavior of the `<%include>` tag, often in conjunction with its interaction within template inheritance. Key to understanding the `<%include>` tag is that it is a *dynamic*, e.g. runtime, include, and not a static include. The `<%include>` is only processed as the template renders, and not at inheritance setup time. When encountered, the referenced template is run fully as an entirely separate template with no linkage to any current inheritance structure.

If the tag were on the other hand a *static* include, this would allow source within the included template to interact within the same inheritance context as the calling template, but currently Mako has no static include facility.

In practice, this means that `<%block>` elements defined in an `<%include>` file will not interact with corresponding `<%block>` elements in the calling template.

A common mistake is along these lines:

```
## partials.mako
<%block name="header">
    Global Header
</%block>

## parent.mako
<%include file="partials.mako">

## child.mako
<%inherit file="parent.mako">
```

```
<%block name="header">
    Custom Header
</%block>
```

Above, one might expect that the "header" block declared in child.mako might be invoked, as a result of it overriding the same block present in parent.mako via the include for partials.mako. But this is not the case. Instead, parent.mako will invoke partials.mako, which then invokes "header" in partials.mako, and then is finished rendering. Nothing from child.mako will render; there is no interaction between the "header" block in child.mako and the "header" block in partials.mako.

Instead, parent.mako must explicitly state the inheritance structure. In order to call upon specific elements of partials.mako, we will call upon it as a namespace:

```
## partials.mako
<%block name="header">
    Global Header
</%block>

## parent.mako
<%namespace name="partials" file="partials.mako"/>
<%block name="header">
    ${partials.header()}
</%block>

## child.mako
<%inherit file="parent.mako">
<%block name="header">
    Custom Header
</%block>
```

Where above, parent.mako states the inheritance structure that child.mako is to participate within. partials.mako only defines defs/blocks that can be used on a per-name basis.

Another scenario is below, which results in both "SectionA" blocks being rendered for the child.mako document:

```
## base.mako
${self.body()}
<%block name="SectionA">
    base.mako
</%block>

## parent.mako
<%inherit file="base.mako">
<%include file="child.mako">

## child.mako
<%block name="SectionA">
    child.mako
</%block>
```

The resolution is similar; instead of using <%include>, we call upon the blocks of child.mako using a namespace:

```
## parent.mako
<%inherit file="base.mako">
<%namespace name="child" file="child.mako">
```

```
<%block name="SectionA">
    ${child.SectionA()}
</%block>
```

Inheritable Attributes

The *attr* accessor of the *Namespace* object allows access to module level variables declared in a template. By accessing `self.attr`, you can access regular attributes from the inheritance chain as declared in `<%! %>` sections. Such as:

```
<%!
    class_ = "grey"
%>

<div class="${self.attr.class_}">
    ${self.body()}
</div>
```

If an inheriting template overrides `class_` to be `"white"`, as in:

```
<%!
    class_ = "white"
%>
<%inherit file="parent.html"/>

This is the body
```

you'll get output like:

```
<div class="white">
    This is the body
</div>
```

See also:

Version One - Use Namespace.attr - a more sophisticated example using *Namespace.attr*.

CHAPTER 7

Filtering and Buffering

Expression Filtering

As described in the chapter *Syntax*, the "`|`" operator can be applied to a "`${}`" expression to apply escape filters to the output:

```
${"this is some text" | u}
```

The above expression applies URL escaping to the expression, and produces `this+is+some+text`.

The built-in escape flags are:

- `u` : URL escaping, provided by `urllib.quote_plus(string.encode('utf-8'))`
- `h` : HTML escaping, provided by `markupsafe.escape(string)`

  New in version 0.3.4: Prior versions use `cgi.escape(string, True)`.
- `x` : XML escaping
- `trim` : whitespace trimming, provided by `string.strip()`
- `entity` : produces HTML entity references for applicable strings, derived from `htmlentitydefs`
- `unicode` (`str` on Python 3): produces a Python unicode string (this function is applied by default)
- `decode.<some encoding>`: decode input into a Python unicode with the specified encoding
- `n` : disable all default filtering; only filters specified in the local expression tag will be applied.

To apply more than one filter, separate them by a comma:

```
${" <tag>some value</tag> " | h,trim}
```

The above produces `&lt;tag&gt;some value&lt;/tag&gt;`, with no leading or trailing whitespace. The HTML escaping function is applied first, the "trim" function second.

Naturally, you can make your own filters too. A filter is just a Python function that accepts a single string argument, and returns the filtered result. The expressions after the | operator draw upon the local namespace of the template in which they appear, meaning you can define escaping functions locally:

```
<%!
    def myescape(text):
        return "<TAG>" + text + "</TAG>"
%>

Here's some tagged text: ${"text" | myescape}
```

Or from any Python module:

```
<%!
    import myfilters
%>

Here's some tagged text: ${"text" | myfilters.tagfilter}
```

A page can apply a default set of filters to all expression tags using the `expression_filter` argument to the `%page` tag:

```
<%page expression_filter="h"/>

Escaped text:  ${"<html>some html</html>"}
```

Result:

```
Escaped text: &lt;html&gt;some html&lt;/html&gt;
```

The `default_filters` Argument

In addition to the `expression_filter` argument, the `default_filters` argument to both *`Template`* and *`TemplateLookup`* can specify filtering for all expression tags at the programmatic level. This array-based argument, when given its default argument of `None`, will be internally set to `["unicode"]` (or `["str"]` on Python 3), except when `disable_unicode=True` is set in which case it defaults to `["str"]`:

```
t = TemplateLookup(directories=['/tmp'], default_filters=['unicode'])
```

To replace the usual `unicode`/`str` function with a specific encoding, the `decode` filter can be substituted:

```
t = TemplateLookup(directories=['/tmp'], default_filters=['decode.utf8'])
```

To disable `default_filters` entirely, set it to an empty list:

```
t = TemplateLookup(directories=['/tmp'], default_filters=[])
```

Any string name can be added to `default_filters` where it will be added to all expressions as a filter. The filters are applied from left to right, meaning the leftmost filter is applied first.

```
t = Template(templatetext, default_filters=['unicode', 'myfilter'])
```

To ease the usage of `default_filters` with custom filters, you can also add imports (or other code) to all templates using the `imports` argument:

```
t = TemplateLookup(directories=['/tmp'],
                   default_filters=['unicode', 'myfilter'],
                   imports=['from mypackage import myfilter'])
```

The above will generate templates something like this:

```
# ....
from mypackage import myfilter

def render_body(context):
    context.write(myfilter(unicode("some text")))
```

Turning off Filtering with the `n` Filter

In all cases the special `n` filter, used locally within an expression, will **disable** all filters declared in the `<%page>` tag as well as in `default_filters`. Such as:

```
${'myexpression' | n}
```

will render `myexpression` with no filtering of any kind, and:

```
${'myexpression' | n,trim}
```

will render `myexpression` using the `trim` filter only.

Filtering Defs and Blocks

The `%def` and `%block` tags have an argument called `filter` which will apply the given list of filter functions to the output of the `%def`:

```
<%def name="foo()" filter="h, trim">
    <b>this is bold</b>
</%def>
```

When the `filter` attribute is applied to a def as above, the def is automatically **buffered** as well. This is described next.

Buffering

One of Mako's central design goals is speed. To this end, all of the textual content within a template and its various callables is by default piped directly to the single buffer that is stored within the *`Context`* object. While this normally is easy to miss, it has certain side effects. The main one is that when you call a def using the normal expression syntax, i.e. `${somedef()}`, it may appear that the return value of the function is the content it produced, which is then delivered to your template just like any other expression substitution, except that normally, this is not the case; the return value of `${somedef()}` is simply the empty string `''`. By the time you receive this empty string, the output of `somedef()` has been sent to the underlying buffer.

You may not want this effect, if for example you are doing something like this:

```
${" results " + somedef() + " more results "}
```

If the `somedef()` function produced the content "`somedef's results`", the above template would produce this output:

```
somedef's results results more results
```

This is because `somedef()` fully executes before the expression returns the results of its concatenation; the concatenation in turn receives just the empty string as its middle expression.

Mako provides two ways to work around this. One is by applying buffering to the `%def` itself:

```
<%def name="somedef()" buffered="True">
    somedef's results
</%def>
```

The above definition will generate code similar to this:

```
def somedef():
    context.push_buffer()
    try:
        context.write("somedef's results")
    finally:
        buf = context.pop_buffer()
    return buf.getvalue()
```

So that the content of `somedef()` is sent to a second buffer, which is then popped off the stack and its value returned. The speed hit inherent in buffering the output of a def is also apparent.

Note that the `filter` argument on `%def` also causes the def to be buffered. This is so that the final content of the `%def` can be delivered to the escaping function in one batch, which reduces method calls and also produces more deterministic behavior for the filtering function itself, which can possibly be useful for a filtering function that wishes to apply a transformation to the text as a whole.

The other way to buffer the output of a def or any Mako callable is by using the built-in `capture` function. This function performs an operation similar to the above buffering operation except it is specified by the caller.

```
${" results " + capture(somedef) + " more results "}
```

Note that the first argument to the `capture` function is **the function itself**, not the result of calling it. This is because the `capture` function takes over the job of actually calling the target function, after setting up a buffered environment. To send arguments to the function, just send them to `capture` instead:

```
${capture(somedef, 17, 'hi', use_paging=True)}
```

The above call is equivalent to the unbuffered call:

```
${somedef(17, 'hi', use_paging=True)}
```

Decorating

New in version 0.2.5.

Somewhat like a filter for a `%def` but more flexible, the `decorator` argument to `%def` allows the creation of a function that will work in a similar manner to a Python decorator. The function can control whether or not the function executes. The original intent of this function is to allow the creation of custom cache logic, but there may be other uses as well.

`decorator` is intended to be used with a regular Python function, such as one defined in a library module. Here we'll illustrate the python function defined in the template for simplicities' sake:

```
<%!
    def bar(fn):
        def decorate(context, *args, **kw):
            context.write("BAR")
            fn(*args, **kw)
            context.write("BAR")
            return ''
        return decorate
%>

<%def name="foo()" decorator="bar">
    this is foo
</%def>

${foo()}
```

The above template will return, with more whitespace than this, `"BAR this is foo BAR"`. The function is the render callable itself (or possibly a wrapper around it), and by default will write to the context. To capture its output, use the *`capture()`* callable in the `mako.runtime` module (available in templates as just `runtime`):

```
<%!
    def bar(fn):
        def decorate(context, *args, **kw):
            return "BAR" + runtime.capture(context, fn, *args, **kw) + "BAR"
        return decorate
%>

<%def name="foo()" decorator="bar">
    this is foo
</%def>

${foo()}
```

The decorator can be used with top-level defs as well as nested defs, and blocks too. Note that when calling a top-level def from the *`Template`* API, i.e. `template.get_def('somedef').render()`, the decorator has to write the output to the `context`, i.e. as in the first example. The return value gets discarded.

CHAPTER 8

The Unicode Chapter

The Python language supports two ways of representing what we know as “strings”, i.e. series of characters. In Python 2, the two types are `string` and `unicode`, and in Python 3 they are `bytes` and `string`. A key aspect of the Python 2 `string` and Python 3 `bytes` types are that they contain no information regarding what **encoding** the data is stored in. For this reason they were commonly referred to as **byte strings** on Python 2, and Python 3 makes this name more explicit. The origins of this come from Python’s background of being developed before the Unicode standard was even available, back when strings were C-style strings and were just that, a series of bytes. Strings that had only values below 128 just happened to be **ASCII** strings and were printable on the console, whereas strings with values above 128 would produce all kinds of graphical characters and bells.

Contrast the “byte-string” type with the “unicode/string” type. Objects of this latter type are created whenever you say something like `u"hello world"` (or in Python 3, just `"hello world"`). In this case, Python represents each character in the string internally using multiple bytes per character (something similar to UTF-16). What’s important is that when using the `unicode`/`string` type to store strings, Python knows the data’s encoding; it’s in its own internal format. Whereas when using the `string`/`bytes` type, it does not.

When Python 2 attempts to treat a byte-string as a string, which means it’s attempting to compare/parse its characters, to coerce it into another encoding, or to decode it to a unicode object, it has to guess what the encoding is. In this case, it will pretty much always guess the encoding as `ascii`... and if the byte-string contains bytes above value 128, you’ll get an error. Python 3 eliminates much of this confusion by just raising an error unconditionally if a byte-string is used in a character-aware context.

There is one operation that Python *can* do with a non-ASCII byte-string, and it’s a great source of confusion: it can dump the byte-string straight out to a stream or a file, with nary a care what the encoding is. To Python, this is pretty much like dumping any other kind of binary data (like an image) to a stream somewhere. In Python 2, it is common to see programs that embed all kinds of international characters and encodings into plain byte-strings (i.e. using `"hello world"` style literals) can fly right through their run, sending reams of strings out to wherever they are going, and the programmer, seeing the same output as was expressed in the input, is now under the illusion that his or her program is Unicode-compliant. In fact, their program has no unicode awareness whatsoever, and similarly has no ability to interact with libraries that *are* unicode aware. Python 3 makes this much less likely by defaulting to unicode as the storage format for strings.

The “pass through encoded data” scheme is what template languages like Cheetah and earlier versions of Myghty do by default. Mako as of version 0.2 also supports this mode of operation when using Python 2, using the `disable_unicode=True` flag. However, when using Mako in its default mode of unicode-aware, it requires explicitness when dealing with non-ASCII encodings. Additionally, if you ever need to handle unicode strings and other kinds of encoding conversions more intelligently, the usage of raw byte-strings quickly becomes a nightmare, since you are sending the Python interpreter collections of bytes for which it can make no intelligent decisions with regards to encoding. In Python 3 Mako only allows usage of native, unicode strings.

In normal Mako operation, all parsed template constructs and output streams are handled internally as Python `unicode` objects. It's only at the point of `render()` that this unicode stream may be rendered into whatever the desired output encoding is. The implication here is that the template developer must :ensure that *the encoding of all non-ASCII templates is explicit* (still required in Python 3), that *all non-ASCII-encoded expressions are in one way or another converted to unicode* (not much of a burden in Python 3), and that *the output stream of the template is handled as a unicode stream being encoded to some encoding* (still required in Python 3).

Specifying the Encoding of a Template File

This is the most basic encoding-related setting, and it is equivalent to Python's "magic encoding comment", as described in pep-0263. Any template that contains non-ASCII characters requires that this comment be present so that Mako can decode to unicode (and also make usage of Python's AST parsing services). Mako's lexer will use this encoding in order to convert the template source into a `unicode` object before continuing its parsing:

```
## -*- coding: utf-8 -*-

Alors vous imaginez ma surprise, au lever du jour, quand
une drôle de petite voix m'a réveillé. Elle disait:
 « S'il vous plaît... dessine-moi un mouton! »
```

For the picky, the regular expression used is derived from that of the above mentioned pep:

```
#.*coding[:=]\s*([-\w.]+).*\n
```

The lexer will convert to unicode in all cases, so that if any characters exist in the template that are outside of the specified encoding (or the default of `ascii`), the error will be immediate.

As an alternative, the template encoding can be specified programmatically to either `Template` or `TemplateLookup` via the `input_encoding` parameter:

```
t = TemplateLookup(directories=['./'], input_encoding='utf-8')
```

The above will assume all located templates specify `utf-8` encoding, unless the template itself contains its own magic encoding comment, which takes precedence.

Handling Expressions

The next area that encoding comes into play is in expression constructs. By default, Mako's treatment of an expression like this:

```
${"hello world"}
```

looks something like this:

```
context.write(unicode("hello world"))
```

In Python 3, it's just:

```
context.write(str("hello world"))
```

That is, **the output of all expressions is run through the ‘‘unicode‘‘ built-in**. This is the default setting, and can be modified to expect various encodings. The `unicode` step serves both the purpose of rendering non-string expressions into strings (such as integers or objects which contain `__str()__` methods), and to ensure that the final output stream is constructed as a unicode object. The main implication of this is that **any raw byte-strings that contain an encoding other than ASCII must first be decoded to a Python unicode object**. It means you can't say this in Python 2:

```
${"voix m'a réveillé."}  ## error in Python 2!
```

You must instead say this:

```
${u"voix m'a réveillé."}  ## OK !
```

Similarly, if you are reading data from a file that is streaming bytes, or returning data from some object that is returning a Python byte-string containing a non-ASCII encoding, you have to explicitly decode to unicode first, such as:

```
${call_my_object().decode('utf-8')}
```

Note that filehandles acquired by `open()` in Python 3 default to returning "text", that is the decoding is done for you. See Python 3's documentation for the `open()` built-in for details on this.

If you want a certain encoding applied to *all* expressions, override the `unicode` builtin with the `decode` built-in at the `Template` or `TemplateLookup` level:

```
t = Template(templatetext, default_filters=['decode.utf8'])
```

Note that the built-in `decode` object is slower than the `unicode` function, since unlike `unicode` it's not a Python built-in, and it also checks the type of the incoming data to determine if string conversion is needed first.

The `default_filters` argument can be used to entirely customize the filtering process of expressions. This argument is described in *The default_filters Argument*.

Defining Output Encoding

Now that we have a template which produces a pure unicode output stream, all the hard work is done. We can take the output and do anything with it.

As stated in the *"Usage" chapter*, both `Template` and `TemplateLookup` accept `output_encoding` and `encoding_errors` parameters which can be used to encode the output in any Python supported codec:

```
from mako.template import Template
from mako.lookup import TemplateLookup

mylookup = TemplateLookup(directories=['/docs'], output_encoding='utf-8', encoding_
↪errors='replace')

mytemplate = mylookup.get_template("foo.txt")
print(mytemplate.render())
```

`render()` will return a `bytes` object in Python 3 if an output encoding is specified. By default it performs no encoding and returns a native string.

`render_unicode()` will return the template output as a Python `unicode` object (or `string` in Python 3):

```
print(mytemplate.render_unicode())
```

The above method disgards the output encoding keyword argument; you can encode yourself by saying:

```
print(mytemplate.render_unicode().encode('utf-8', 'replace'))
```

Buffer Selection

Mako does play some games with the style of buffering used internally, to maximize performance. Since the buffer is by far the most heavily used object in a render operation, it's important!

When calling `render()` on a template that does not specify any output encoding (i.e. it's `ascii`), Python's `cStringIO` module, which cannot handle encoding of non-ASCII `unicode` objects (even though it can send raw byte-strings through), is used for buffering. Otherwise, a custom Mako class called `FastEncodingBuffer` is used, which essentially is a super dumbed-down version of `StringIO` that gathers all strings into a list and uses `u''.join(elements)` to produce the final output – it's markedly faster than `StringIO`.

Saying to Heck with It: Disabling the Usage of Unicode Entirely

Some segments of Mako's userbase choose to make no usage of Unicode whatsoever, and instead would prefer the "pass through" approach; all string expressions in their templates return encoded byte-strings, and they would like these strings to pass right through. The only advantage to this approach is that templates need not use `u""` for literal strings; there's an arguable speed improvement as well since raw byte-strings generally perform slightly faster than unicode objects in Python. For these users, assuming they're sticking with Python 2, they can hit the `disable_unicode=True` flag as so:

```
# -*- coding:utf-8 -*-
from mako.template import Template

t = Template("drôle de petite voix m’a réveillé.", disable_unicode=True, input_
↪encoding='utf-8')
print(t.code)
```

The `disable_unicode` mode is strictly a Python 2 thing. It is not supported at all in Python 3.

The generated module source code will contain elements like these:

```
# -*- coding:utf-8 -*-
#  ...more generated code ...

def render_body(context,**pageargs):
    context.caller_stack.push_frame()
    try:
        __M_locals = dict(pageargs=pageargs)
        # SOURCE LINE 1
        context.write('dr\xc3\xb4le de petite voix m\xe2\x80\x99a␣
↪r\xc3\xa9veill\xc3\xa9.')
        return ''
    finally:
        context.caller_stack.pop_frame()
```

Where above that the string literal used within `Context.write()` is a regular byte-string.

When `disable_unicode=True` is turned on, the `default_filters` argument which normally defaults to `["unicode"]` now defaults to `["str"]` instead. Setting `default_filters` to the empty list `[]` can remove the overhead of the `str` call. Also, in this mode you **cannot** safely call *`render_unicode()`* – you'll get unicode/decode errors.

The `h` filter (HTML escape) uses a less performant pure Python escape function in non-unicode mode. This because MarkupSafe only supports Python unicode objects for non-ASCII strings.

Changed in version 0.3.4: In prior versions, it used `cgi.escape()`, which has been replaced with a function that also escapes single quotes.

Rules for using `disable_unicode=True`

- Don't use this mode unless you really, really want to and you absolutely understand what you're doing.
- Don't use this option just because you don't want to learn to use Unicode properly; we aren't supporting user issues in this mode of operation. We will however offer generous help for the vast majority of users who stick to the Unicode program.
- Python 3 is unicode by default, and the flag is not available when running on Python 3.

CHAPTER 9

Caching

Any template or component can be cached using the `cache` argument to the `<%page>`, `<%def>` or `<%block>` directives:

```
<%page cached="True"/>

template text
```

The above template, after being executed the first time, will store its content within a cache that by default is scoped within memory. Subsequent calls to the template's *`render()`* method will return content directly from the cache. When the *`Template`* object itself falls out of scope, its corresponding cache is garbage collected along with the template.

The caching system requires that a cache backend be installed; this includes either the Beaker package or the dogpile.cache, as well as any other third-party caching libraries that feature Mako integration.

By default, caching will attempt to make use of Beaker. To use dogpile.cache, the `cache_impl` argument must be set; see this argument in the section *Cache Arguments*.

In addition to being available on the `<%page>` tag, the caching flag and all its options can be used with the `<%def>` tag as well:

```
<%def name="mycomp" cached="True" cache_timeout="60">
    other text
</%def>
```

... and equivalently with the `<%block>` tag, anonymous or named:

```
<%block cached="True" cache_timeout="60">
    other text
</%block>
```

Cache Arguments

Mako has two cache arguments available on tags that are available in all cases. The rest of the arguments available are specific to a backend.

The two generic tags arguments are:

- `cached="True"` - enable caching for this `<%page>`, `<%def>`, or `<%block>`.
- `cache_key` - the "key" used to uniquely identify this content in the cache. Usually, this key is chosen automatically based on the name of the rendering callable (i.e. `body` when used in `<%page>`, the name of the def when using `<%def>`, the explicit or internally-generated name when using `<%block>`). Using the `cache_key` parameter, the key can be overridden using a fixed or programmatically generated value.

  For example, here's a page that caches any page which inherits from it, based on the filename of the calling template:

  ```
  <%page cached="True" cache_key="${self.filename}"/>

  ${next.body()}

  ## rest of template
  ```

On a *Template* or *TemplateLookup*, the caching can be configured using these arguments:

- `cache_enabled` - Setting this to `False` will disable all caching functionality when the template renders. Defaults to `True`. e.g.:

  ```
  lookup = TemplateLookup(
                  directories='/path/to/templates',
                  cache_enabled = False
                  )
  ```

- `cache_impl` - The string name of the cache backend to use. This defaults to `'beaker'`, indicating that the 'beaker' backend will be used.
- `cache_args` - A dictionary of cache parameters that will be consumed by the cache backend. See *Using the Beaker Cache Backend* and *Using the dogpile.cache Backend* for examples.

Backend-Specific Cache Arguments

The `<%page>`, `<%def>`, and `<%block>` tags accept any named argument that starts with the prefix `"cache_"`. Those arguments are then packaged up and passed along to the underlying caching implementation, minus the `"cache_"` prefix.

The actual arguments understood are determined by the backend.

- *Using the Beaker Cache Backend* - Includes arguments understood by Beaker.
- *Using the dogpile.cache Backend* - Includes arguments understood by dogpile.cache.

Using the Beaker Cache Backend

When using Beaker, new implementations will want to make usage of **cache regions** so that cache configurations can be maintained externally to templates. These configurations live under named "regions" that can be referred to within templates themselves.

New in version 0.6.0: Support for Beaker cache regions.

For example, suppose we would like two regions. One is a "short term" region that will store content in a memory-based dictionary, expiring after 60 seconds. The other is a Memcached region, where values should expire in five minutes. To configure our *TemplateLookup*, first we get a handle to a `beaker.cache.CacheManager`:

```
from beaker.cache import CacheManager

manager = CacheManager(cache_regions={
    'short_term':{
        'type': 'memory',
        'expire': 60
    },
    'long_term':{
        'type': 'ext:memcached',
        'url': '127.0.0.1:11211',
        'expire': 300
    }
})

lookup = TemplateLookup(
                directories=['/path/to/templates'],
                module_directory='/path/to/modules',
                cache_impl='beaker',
                cache_args={
                    'manager':manager
                }
        )
```

Our templates can then opt to cache data in one of either region, using the `cache_region` argument. Such as using `short_term` at the `<%page>` level:

```
<%page cached="True" cache_region="short_term">

## ...
```

Or, `long_term` at the `<%block>` level:

```
<%block name="header" cached="True" cache_region="long_term">
    other text
</%block>
```

The Beaker backend also works without regions. There are a variety of arguments that can be passed to the `cache_args` dictionary, which are also allowable in templates via the `<%page>`, `<%block>`, and `<%def>` tags specific to those sections. The values given override those specified at the *TemplateLookup* or *Template* level.

With the possible exception of `cache_timeout`, these arguments are probably better off staying at the template configuration level. Each argument specified as `cache_XYZ` in a template tag is specified without the `cache_` prefix in the `cache_args` dictionary:

- `cache_timeout` - number of seconds in which to invalidate the cached data. After this timeout, the content is re-generated on the next call. Available as `timeout` in the `cache_args` dictionary.
- `cache_type` - type of caching. `'memory'`, `'file'`, `'dbm'`, or `'ext:memcached'` (note that the string `memcached` is also accepted by the dogpile.cache Mako plugin, though not by Beaker itself). Available as `type` in the `cache_args` dictionary.
- `cache_url` - (only used for `memcached` but required) a single IP address or a semi-colon separated list of IP address of memcache servers to use. Available as `url` in the `cache_args` dictionary.
- `cache_dir` - in the case of the `'file'` and `'dbm'` cache types, this is the filesystem directory with which to store data files. If this option is not present, the value of `module_directory` is used (i.e. the directory

where compiled template modules are stored). If neither option is available an exception is thrown. Available as `dir` in the `cache_args` dictionary.

Using the dogpile.cache Backend

dogpile.cache is a new replacement for Beaker. It provides a modernized, slimmed down interface and is generally easier to use than Beaker. As of this writing it has not yet been released. dogpile.cache includes its own Mako cache plugin – see `dogpile.cache.plugins.mako_cache` in the dogpile.cache documentation.

Programmatic Cache Access

The `Template`, as well as any template-derived `Namespace`, has an accessor called `cache` which returns the `Cache` object for that template. This object is a facade on top of the underlying `CacheImpl` object, and provides some very rudimental capabilities, such as the ability to get and put arbitrary values:

```
<%
    local.cache.set("somekey", type="memory", "somevalue")
%>
```

Above, the cache associated with the `local` namespace is accessed and a key is placed within a memory cache.

More commonly, the `cache` object is used to invalidate cached sections programmatically:

```
template = lookup.get_template('/sometemplate.html')

# invalidate the "body" of the template
template.cache.invalidate_body()

# invalidate an individual def
template.cache.invalidate_def('somedef')

# invalidate an arbitrary key
template.cache.invalidate('somekey')
```

You can access any special method or attribute of the `CacheImpl` itself using the `impl` attribute:

```
template.cache.impl.do_something_special()
```

Note that using implementation-specific methods will mean you can't swap in a different kind of `CacheImpl` implementation at a later time.

Cache Plugins

The mechanism used by caching can be plugged in using a `CacheImpl` subclass. This class implements the rudimental methods Mako needs to implement the caching API. Mako includes the `BeakerCacheImpl` class to provide the default implementation. A `CacheImpl` class is acquired by Mako using a `pkg_resources` entrypoint, using the name given as the `cache_impl` argument to `Template` or `TemplateLookup`. This entry point can be installed via the standard *setuptools*/`setup()` procedure, underneath the *EntryPoint* group named `"mako.cache"`. It can also be installed at runtime via a convenience installer `register_plugin()` which accomplishes essentially the same task.

An example plugin that implements a local dictionary cache:

```
from mako.cache import Cacheimpl, register_plugin

class SimpleCacheImpl(CacheImpl):
    def __init__(self, cache):
        super(SimpleCacheImpl, self).__init__(cache)
        self._cache = {}

    def get_or_create(self, key, creation_function, **kw):
        if key in self._cache:
            return self._cache[key]
        else:
            self._cache[key] = value = creation_function()
            return value

    def set(self, key, value, **kwargs):
        self._cache[key] = value

    def get(self, key, **kwargs):
        return self._cache.get(key)

    def invalidate(self, key, **kwargs):
        self._cache.pop(key, None)

# optional - register the class locally
register_plugin("simple", __name__, "SimpleCacheImpl")
```

Enabling the above plugin in a template would look like:

```
t = Template("mytemplate",
             file="mytemplate.html",
             cache_impl='simple')
```

Guidelines for Writing Cache Plugins

- The `CacheImpl` is created on a per-`Template` basis. The class should ensure that only data for the parent `Template` is persisted or returned by the cache methods. The actual `Template` is available via the `self.cache.template` attribute. The `self.cache.id` attribute, which is essentially the unique modulename of the template, is a good value to use in order to represent a unique namespace of keys specific to the template.
- Templates only use the `CacheImpl.get_or_create()` method in an implicit fashion. The `CacheImpl.set()`, `CacheImpl.get()`, and `CacheImpl.invalidate()` methods are only used in response to direct programmatic access to the corresponding methods on the `Cache` object.
- `CacheImpl` will be accessed in a multithreaded fashion if the `Template` itself is used multithreaded. Care should be taken to ensure caching implementations are threadsafe.
- A library like Dogpile, which is a minimal locking system derived from Beaker, can be used to help implement the `CacheImpl.get_or_create()` method in a threadsafe way that can maximize effectiveness across multiple threads as well as processes. `CacheImpl.get_or_create()` is the key method used by templates.
- All arguments passed to `**kw` come directly from the parameters inside the `<%def>`, `<%block>`, or `<%page>` tags directly, minus the `"cache_"` prefix, as strings, with the exception of the argument `cache_timeout`, which is passed to the plugin as the name `timeout` with the value converted to an integer. Arguments present in `cache_args` on `Template` or `TemplateLookup` are passed directly, but are superseded by those present in the most specific template tag.

- The directory where *`Template`* places module files can be acquired using the accessor `self.cache.template.module_directory`. This directory can be a good place to throw cache-related work files, underneath a prefix like `_my_cache_work` so that name conflicts with generated modules don't occur.

API Reference

class `mako.cache.`**`Cache`**(*template*, **args*)

Bases: `object`

Represents a data content cache made available to the module space of a specific *`Template`* object.

New in version 0.6: *`Cache`* by itself is mostly a container for a *`CacheImpl`* object, which implements a fixed API to provide caching services; specific subclasses exist to implement different caching strategies. Mako includes a backend that works with the Beaker caching system. Beaker itself then supports a number of backends (i.e. file, memory, memcached, etc.)

The construction of a *`Cache`* is part of the mechanics of a *`Template`*, and programmatic access to this cache is typically via the `Template.cache` attribute.

`get`(*key*, ***kw*)

Retrieve a value from the cache.

Parameters

- **`key`** – the value's key.
- **`**kw`** – cache configuration arguments. The backend is configured using these arguments upon first request. Subsequent requests that use the same series of configuration values will use that same backend.

`get_or_create`(*key*, *creation_function*, ***kw*)

Retrieve a value from the cache, using the given creation function to generate a new value.

`id` = **None**

Return the 'id' that identifies this cache.

This is a value that should be globally unique to the *`Template`* associated with this cache, and can be used by a caching system to name a local container for data specific to this template.

`impl` = **None**

Provide the *`CacheImpl`* in use by this *`Cache`*.

This accessor allows a *`CacheImpl`* with additional methods beyond that of *`Cache`* to be used programmatically.

`invalidate`(*key*, ***kw*)

Invalidate a value in the cache.

Parameters

- **`key`** – the value's key.
- **`**kw`** – cache configuration arguments. The backend is configured using these arguments upon first request. Subsequent requests that use the same series of configuration values will use that same backend.

`invalidate_body`()

Invalidate the cached content of the "body" method for this template.

`invalidate_closure`(*name*)

Invalidate a nested `<%def>` within this template.

Caching of nested defs is a blunt tool as there is no management of scope – nested defs that use cache tags need to have names unique of all other nested defs in the template, else their content will be overwritten by each other.

invalidate_def(*name*)
Invalidate the cached content of a particular `<%def>` within this template.

put(*key*, *value*, ***kw*)
A synonym for `Cache.set()`.

This is here for backwards compatibility.

set(*key*, *value*, ***kw*)
Place a value in the cache.

Parameters

- **key** – the value's key.
- **value** – the value.
- ****kw** – cache configuration arguments.

starttime = None
Epochal time value for when the owning `Template` was first compiled.

A cache implementation may wish to invalidate data earlier than this timestamp; this has the effect of the cache for a specific `Template` starting clean any time the `Template` is recompiled, such as when the original template file changed on the filesystem.

class `mako.cache.`**CacheImpl**(*cache*)
Bases: `object`

Provide a cache implementation for use by `Cache`.

get(*key*, ***kw*)
Retrieve a value from the cache.

Parameters

- **key** – the value's key.
- ****kw** – cache configuration arguments.

get_or_create(*key*, *creation_function*, ***kw*)
Retrieve a value from the cache, using the given creation function to generate a new value.

This function *must* return a value, either from the cache, or via the given creation function. If the creation function is called, the newly created value should be populated into the cache under the given key before being returned.

Parameters

- **key** – the value's key.
- **creation_function** – function that when called generates a new value.
- ****kw** – cache configuration arguments.

invalidate(*key*, ***kw*)
Invalidate a value in the cache.

Parameters

- **key** – the value's key.
- ****kw** – cache configuration arguments.

`pass_context` = **False**

If `True`, the *`Context`* will be passed to *`get_or_create`* as the name `'context'`.

`set`(*key*, *value*, ***kw*)

Place a value in the cache.

Parameters

- **`key`** – the value's key.
- **`value`** – the value.
- **`**kw`** – cache configuration arguments.

`mako.cache.`**`register_plugin`**(*self*, *name*, *modulepath*, *objname*)

class `mako.ext.beaker_cache.`**`BeakerCacheImpl`**(*cache*)

Bases: *`mako.cache.CacheImpl`*

A *`CacheImpl`* provided for the Beaker caching system.

This plugin is used by default, based on the default value of `'beaker'` for the `cache_impl` parameter of the *`Template`* or *`TemplateLookup`* classes.

CHAPTER 10

Changelog

1.0

1.0.7

Released: Thu Jul 13 2017

- **[bug]** Changed the "print" in the mako-render command to sys.stdout.write(), avoiding the extra newline at the end of the template output. Pull request courtesy Yves Chevallier. ¶

1.0.6

Released: Wed Nov 9 2016

- **[feature]** Added new parameter `Template.include_error_handler`. This works like `Template.error_handler` but indicates the handler should take place when this template is included within another template via the `<%include>` tag. Pull request courtesy Huayi Zhang. ¶

1.0.5

Released: Wed Nov 2 2016

- **[bug]** Updated the Sphinx documentation builder to work with recent versions of Sphinx. ¶

1.0.4

Released: Thu Mar 10 2016

- **[test] [feature]** The default test runner is now py.test. Running "python setup.py test" will make use of py.test instead of nose. nose still works as a test runner as well, however. ¶

- **[lexer] [bug]** Major improvements to lexing of intricate Python sections which may contain complex backslash sequences, as well as support for the bitwise operator (e.g. pipe symbol) inside of expression sections distinct from the Mako "filter" operator, provided the operator is enclosed within parentheses or brackets. Pull request courtesy Daniel Martin. ¶ References: pull request github:19
- **[feature]** Added new method `Template.list_defs()`. Pull request courtesy Jonathan Vanasco. ¶ References: pull request bitbucket:16

1.0.3

Released: Tue Oct 27 2015

- **[babel] [bug]** Fixed an issue where the Babel plugin would not handle a translation symbol that contained non-ascii characters. Pull request courtesy Roman Imankulov. ¶ References: pull request bitbucket:21

1.0.2

Released: Wed Aug 26 2015

- **[installation] [bug]** The "universal wheel" marker is removed from setup.cfg, because our setup.py currently makes use of conditional dependencies. In #249, the discussion is ongoing on how to correct our setup.cfg / setup.py fully so that we can handle the per-version dependency changes while still maintaining optimal wheel settings, so this issue is not yet fully resolved. ¶ References: #249
- **[bug] [py3k]** Repair some calls within the ast module that no longer work on Python3.5; additionally replace the use of `inspect.getargspec()` under Python 3 (seems to be called from the TG plugin) to avoid deprecation warnings. ¶ References: #250
- **[bug]** Update the Lingua translation extraction plugin to correctly handle templates mixing Python control statements (such as if, for and while) with template fragments. Pull request courtesy Laurent Daverio. ¶ References: pull request bitbucket:18
- **[feature]** Added `STOP_RENDERING` keyword for returning/exiting from a template early, which is a synonym for an empty string `""`. Previously, the docs suggested a bare `return`, but this could cause `None` to appear in the rendered template result.

  See also:

  Exiting Early from a Template

  ¶ References: #236

1.0.1

Released: Thu Jan 22 2015

- **[feature]** Added support for Lingua, a translation extraction system as an alternative to Babel. Pull request courtesy Wichert Akkerman. ¶ References: pull request bitbucket:9
- **[bug] [py3k]** Modernized the examples/wsgi/run_wsgi.py file for Py3k. Pull requset courtesy Cody Taylor. ¶ References: pull request bitbucket:11

1.0.0

Released: Sun Jun 8 2014

- **[py2k] [bug]** Improved the error re-raise operation when a custom *`Template.error_handler`* is used that does not handle the exception; the original stack trace etc. is now preserved. Pull request courtesy Manfred Haltner. ¶ References: pull request bitbucket:8
- **[py2k] [bug] [filters]** Added an html_escape filter that works in "non unicode" mode. Previously, when using `disable_unicode=True`, the `u` filter would fail to handle non-ASCII bytes properly. Pull request courtesy George Xie. ¶ References: pull request bitbucket:7
- **[general]** Compatibility changes; in order to modernize the codebase, Mako is now dropping support for Python 2.4 and Python 2.5 altogether. The source base is now targeted at Python 2.6 and forwards. ¶
- **[feature]** Template modules now generate a JSON "metadata" structure at the bottom of the source file which includes parseable information about the templates' source file, encoding etc. as well as a mapping of module source lines to template lines, thus replacing the "# SOURCE LINE" markers throughout the source code. The structure also indicates those lines that are explicitly not part of the template's source; the goal here is to allow better integration with coverage and other tools. ¶
- **[bug] [py3k]** Fixed bug in `decode.<encoding>` filter where a non-string object would not be correctly interpreted in Python 3. ¶
- **[bug] [py3k]** Fixed bug in Python parsing logic which would fail on Python 3 when a "try/except" targeted a tuple of exception types, rather than a single exception. ¶ References: #227
- **[feature]** mako-render is now implemented as a setuptools entrypoint script; a standalone mako.cmd.cmdline() callable is now available, and the system also uses argparse now instead of optparse. Pull request courtesy Derek Harland. ¶ References: pull request bitbucket:5
- **[feature]** The mako-render script will now catch exceptions and run them into the text error handler, and exit with a non-zero exit code. Pull request courtesy Derek Harland. ¶ References: pull request bitbucket:4
- **[bug]** A rework of the mako-render script allows the script to run correctly when given a file pathname that is outside of the current directory, e.g. `mako-render ../some_template.mako`. In this case, the "template root" defaults to the directory in which the template is located, instead of ".". The script also accepts a new argument `--template-dir` which can be specified multiple times to establish template lookup directories. Standard input for templates also works now too. Pull request courtesy Derek Harland. ¶ References: pull request bitbucket:2
- **[feature] [py3k]** Support is added for Python 3 "keyword only" arguments, as used in defs. Pull request courtesy Eevee. ¶ References: pull request github:7

0.9

0.9.1

Released: Thu Dec 26 2013

- **[bug]** Fixed bug in Babel plugin where translator comments would be lost if intervening text nodes were encountered. Fix courtesy Ned Batchelder. ¶ References: #225
- **[bug]** Fixed TGPlugin.render method to support unicode template names in Py2K - courtesy Vladimir Magamedov. ¶
- **[bug]** Fixed an AST issue that was preventing correct operation under alpha versions of Python 3.4. Pullreq courtesy Zer0-. ¶
- **[bug]** Changed the format of the "source encoding" header output by the code generator to use the format `# -*- coding:%s -*-` instead of `# -*- encoding:%s -*-`; the former is more common and compatible with emacs. Courtesy Martin Geisler. ¶

- **[bug]** Fixed issue where an old lexer rule prevented a template line which looked like "#*" from being correctly parsed. ¶ References: #224

0.9.0

Released: Tue Aug 27 2013

- **[bug]** The Context.locals_() method becomes a private underscored method, as this method has a specific internal use. The purpose of Context.kwargs has been clarified, in that it only delivers top level keyword arguments originally passed to template.render(). ¶ References: #219
- **[bug]** Fixed the babel plugin to properly interpret ${} sections inside of a "call" tag, i.e. <%self:some_tag attr="${_('foo')}"/>. Code that's subject to babel escapes in here needs to be specified as a Python expression, not a literal. This change is backwards incompatible vs. code that is relying upon a _('') translation to be working within a call tag. ¶
- **[bug]** The Babel plugin has been repaired to work on Python 3. ¶ References: #187
- **[bug]** Using <%namespace import="*" module="somemodule"/> now skips over module elements that are not explcitly callable, avoiding TypeError when trying to produce partials. ¶ References: #207
- **[bug]** Fixed Py3K bug where a "lambda" expression was not interpreted correctly within a template tag; also fixed in Py2.4. ¶ References: #190

0.8

0.8.1

Released: Fri May 24 2013

- **[bug]** Changed setup.py to skip installing markupsafe if Python version is < 2.6 or is between 3.0 and less than 3.3, as Markupsafe now only supports 2.6->2.X, 3.3->3.X. ¶ References: #216
- **[bug]** Fixed regression where "entity" filter wasn't converted for py3k properly (added tests.) ¶ References: #214
- **[bug]** Fixed bug where mako-render script wasn't compatible with Py3k. ¶ References: #212
- **[bug]** Cleaned up all the various deprecation/ file warnings when running the tests under various Pythons with warnings turned on. ¶ References: #213

0.8.0

Released: Wed Apr 10 2013

- **[feature]** Performance improvement to the "legacy" HTML escape feature, used for XML escaping and when markupsafe isn't present, courtesy George Xie. ¶
- **[bug]** Fixed bug whereby an exception in Python 3 against a module compiled to the filesystem would fail trying to produce a RichTraceback due to the content being in bytes. ¶ References: #209
- **[bug]** Change default for compile()->reserved_names from tuple to frozenset, as this is expected to be a set by default. ¶ References: #208
- **[feature]** Code has been reworked to support Python 2.4-> Python 3.xx in place. 2to3 no longer needed. ¶

- **[feature]** Added lexer_cls argument to Template, TemplateLookup, allows alternate Lexer classes to be used. ¶
- **[feature]** Added future_imports parameter to Template and TemplateLookup, renders the __future__ header with desired capabilities at the top of the generated template module. Courtesy Ben Trofatter. ¶

0.7

0.7.3

Released: Wed Nov 7 2012

- **[bug]** legacy_html_escape function, used when Markupsafe isn't installed, was using an inline-compiled regexp which causes major slowdowns on Python 3.3; is now precompiled. ¶
- **[bug]** AST supporting now supports tuple-packed function arguments inside pure-python def or lambda expressions. ¶ References: #201
- **[bug]** Fixed Py3K bug in the Babel extension. ¶
- **[bug]** Fixed the "filter" attribute of the <%text> tag so that it pulls locally specified identifiers from the context the same way as that of <%block> and <%filter>. ¶
- **[bug]** Fixed bug in plugin loader to correctly raise exception when non-existent plugin is specified. ¶

0.7.2

Released: Fri Jul 20 2012

- **[bug]** Fixed regression in 0.7.1 where AST parsing for Py2.4 was broken. ¶ References: #193

0.7.1

Released: Sun Jul 8 2012

- **[feature]** Control lines with no bodies will now succeed, as "pass" is added for these when no statements are otherwise present. Courtesy Ben Trofatter ¶ References: #146
- **[bug]** Fixed some long-broken scoping behavior involving variables declared in defs and such, which only became apparent when the strict_undefined flag was turned on. ¶ References: #192
- **[bug]** Can now use strict_undefined at the same time args passed to def() are used by other elements of the <%def> tag. ¶ References: #191

0.7.0

Released: Fri Mar 30 2012

- **[feature]** Added new "loop" variable to templates, is provided within a % for block to provide info about the loop such as index, first/last, odd/even, etc. A migration path is also provided for legacy templates via the "enable_loop" argument available on Template, TemplateLookup, and <%page>. Thanks to Ben Trofatter for all the work on this ¶ References: #125
- **[feature]** Added a real check for "reserved" names, that is names which are never pulled from the context and cannot be passed to the template.render() method. Current names are "context", "loop", "UNDEFINED". ¶

- **[feature]** The html_error_template() will now apply Pygments highlighting to the source code displayed in the traceback, if Pygments if available. Courtesy Ben Trofatter ¶ References: #95
- **[feature]** Added support for context managers, i.e. "% with x as e:/ % endwith" support. Courtesy Ben Trofatter ¶ References: #147
- **[feature]** Added class-level flag to CacheImpl "pass_context"; when True, the keyword argument 'context' will be passed to get_or_create() containing the Mako Context object. ¶ References: #185
- **[bug]** Fixed some Py3K resource warnings due to filehandles being implicitly closed. ¶ References: #182
- **[bug]** Fixed endless recursion bug when nesting multiple def-calls with content. Thanks to Jeff Dairiki. ¶ References: #186
- **[feature]** Added Jinja2 to the example benchmark suite, courtesy Vincent Férotin ¶

Older Versions

0.6.2

Released: Thu Feb 2 2012

- **[bug]** The ${{"foo":"bar"}} parsing issue is fixed!! The legendary Eevee has slain the dragon!. Also fixes quoting issue at. ¶ References: #86, #20

0.6.1

Released: Sat Jan 28 2012

- **[bug]** Added special compatibility for the 0.5.0 Cache() constructor, which was preventing file version checks and not allowing Mako 0.6 to recompile the module files. ¶

0.6.0

Released: Sat Jan 21 2012

- **[feature]** Template caching has been converted into a plugin system, whereby the usage of Beaker is just the default plugin. Template and TemplateLookup now accept a string "cache_impl" parameter which refers to the name of a cache plugin, defaulting to the name 'beaker'. New plugins can be registered as pkg_resources entrypoints under the group "mako.cache", or registered directly using mako.cache.register_plugin(). The core plugin is the mako.cache.CacheImpl class. ¶
- **[feature]** Added support for Beaker cache regions in templates. Usage of regions should be considered as superseding the very obsolete idea of passing in backend options, timeouts, etc. within templates. ¶
- **[feature]** The 'put' method on Cache is now 'set'. 'put' is there for backwards compatibility. ¶
- **[feature]** The <%def>, <%block> and <%page> tags now accept any argument named "cache_*", and the key minus the "**cache_**" prefix will be passed as keyword arguments to the CacheImpl methods. ¶
- **[feature]** Template and TemplateLookup now accept an argument cache_args, which refers to a dictionary containing cache parameters. The cache_dir, cache_url, cache_type, cache_timeout arguments are deprecated (will probably never be removed, however) and can be passed now as cache_args={'url':<some url>, 'type':'memcached', 'timeout':50, 'dir':'/path/to/some/directory'} ¶

- **[feature/bug]** Can now refer to context variables within extra arguments to <%block>, <%def>, i.e. <%block name="foo" cache_key="${somekey}">. Filters can also be used in this way, i.e. <%def name="foo()" filter="myfilter"> then template.render(myfilter=some_callable) ¶ References: #180
- **[feature]** Added "–var name=value" option to the mako-render script, allows passing of kw to the template from the command line. ¶ References: #178
- **[feature]** Added module_writer argument to Template, TemplateLookup, allows a callable to be passed which takes over the writing of the template's module source file, so that special environment-specific steps can be taken. ¶ References: #181
- **[bug]** The exception message in the html_error_template is now escaped with the HTML filter. ¶ References: #142
- **[bug]** Added "white-space:pre" style to html_error_template() for code blocks so that indentation is preserved ¶ References: #173
- **[bug]** The "benchmark" example is now Python 3 compatible (even though several of those old template libs aren't available on Py3K, so YMMV) ¶ References: #175

0.5.0

Released: Wed Sep 28 2011

- A Template is explicitly disallowed from having a url that normalizes to relative outside of the root. That is, if the Lookup is based at /home/mytemplates, an include that would place the ultimate template at /home/mytemplates/../some_other_directory, i.e. outside of /home/mytemplates, is disallowed. This usage was never intended despite the lack of an explicit check. The main issue this causes is that module files can be written outside of the module root (or raise an error, if file perms aren't set up), and can also lead to the same template being cached in the lookup under multiple, relative roots. TemplateLookup instead has always supported multiple file roots for this purpose. ¶ References: #174

0.4.2

Released: Fri Aug 5 2011

- Fixed bug regarding <%call>/def calls w/ content whereby the identity of the "caller" callable inside the <%def> would be corrupted by the presence of another <%call> in the same block. ¶ References: #170
- Fixed the babel plugin to accommodate <%block> ¶ References: #169

0.4.1

Released: Wed Apr 6 2011

- New tag: <%block>. A variant on <%def> that evaluates its contents in-place. Can be named or anonymous, the named version is intended for inheritance layouts where any given section can be surrounded by the <%block> tag in order for it to become overrideable by inheriting templates, without the need to specify a top-level <%def> plus explicit call. Modified scoping and argument rules as well as a more strictly enforced usage scheme make it ideal for this purpose without at all replacing most other things that defs are still good for. Lots of new docs. ¶ References: #164
- a slight adjustment to the "highlight" logic for generating template bound stacktraces. Will stick to known template source lines without any extra guessing. ¶ References: #165

0.4.0

Released: Sun Mar 6 2011

- A 20% speedup for a basic two-page inheritance setup rendering a table of escaped data (see http://techspot.zzzeek.org/2010/11/19/quick-mako-vs.-jinja-speed-test/). A few configurational changes which affect those in the I-don't-do-unicode camp should be noted below. ¶
- The FastEncodingBuffer is now used by default instead of cStringIO or StringIO, regardless of whether output_encoding is set to None or not. FEB is faster than both. Only StringIO allows bytestrings of unknown encoding to pass right through, however - while it is of course not recommended to send bytestrings of unknown encoding to the output stream, this mode of usage can be re-enabled by setting the flag bytestring_passthrough to True. ¶
- disable_unicode mode requires that output_encoding be set to None - it also forces the bytestring_passthrough flag to True. ¶
- the <%namespace> tag raises an error if the 'template' and 'module' attributes are specified at the same time in one tag. A different class is used for each case which allows a reduction in runtime conditional logic and function call overhead. ¶ References: #156
- the keys() in the Context, as well as it's internal _data dictionary, now include just what was specified to render() as well as Mako builtins 'caller', 'capture'. The contents of __builtin__ are no longer copied. Thanks to Daniel Lopez for pointing this out. ¶ References: #159

0.3.6

Released: Sat Nov 13 2010

- Documentation is on Sphinx. ¶ References: #126
- Beaker is now part of "extras" in setup.py instead of "install_requires". This to produce a lighter weight install for those who don't use the caching as well as to conform to Pyramid deployment practices. ¶ References: #154
- The Beaker import (or attempt thereof) is delayed until actually needed; this to remove the performance penalty from startup, particularly for "single execution" environments such as shell scripts. ¶ References: #153
- Patch to lexer to not generate an empty '' write in the case of backslash-ended lines. ¶ References: #155
- Fixed missing **extra collection in setup.py which prevented setup.py from running 2to3 on install. ¶ References: #148
- New flag on Template, TemplateLookup - strict_undefined=True, will cause variables not found in the context to raise a NameError immediately, instead of defaulting to the UNDEFINED value. ¶
- The range of Python identifiers that are considered "undefined", meaning they are pulled from the context, has been trimmed back to not include variables declared inside of expressions (i.e. from list comprehensions), as well as in the argument list of lambdas. This to better support the strict_undefined feature. The change should be fully backwards-compatible but involved a little bit of tinkering in the AST code, which hadn't really been touched for a couple of years, just FYI. ¶

0.3.5

Released: Sun Oct 24 2010

- The <%namespace> tag allows expressions for the *file* argument, i.e. with ${}. The *context* variable, if needed, must be referenced explicitly. ¶ References: #141
- ${} expressions embedded in tags, such as <%foo:bar x="${...}">, now allow multiline Python expressions. ¶

- Fixed previously non-covered regular expression, such that using a ${} expression inside of a tag element that doesn't allow them raises a CompileException instead of silently failing. ¶
- Added a try/except around "import markupsafe". This to support GAE which can't run markupsafe. No idea whatsoever if the install_requires in setup.py also breaks GAE, couldn't get an answer on this. ¶ References: #151

0.3.4

Released: Tue Jun 22 2010

- Now using MarkupSafe for HTML escaping, i.e. in place of cgi.escape(). Faster C-based implementation and also escapes single quotes for additional security. Supports the __html__ attribute for the given expression as well.

  When using "disable_unicode" mode, a pure Python HTML escaper function is used which also quotes single quotes.

  Note that Pylons by default doesn't use Mako's filter - check your environment.py file. ¶
- Fixed call to "unicode.strip" in exceptions.text_error_template which is not Py3k compatible. ¶ References: #137

0.3.3

Released: Mon May 31 2010

- Added conditional to RichTraceback such that if no traceback is passed and sys.exc_info() has been reset, the formatter just returns blank for the "traceback" portion. ¶ References: #135
- Fixed sometimes incorrect usage of exc.__class__.__name__ in html/text error templates when using Python 2.4 ¶ References: #131
- Fixed broken @property decorator on template.last_modified ¶
- Fixed error formatting when a stacktrace line contains no line number, as in when inside an eval/exec-generated function. ¶ References: #132
- When a .py is being created, the tempfile where the source is stored temporarily is now made in the same directory as that of the .py file. This ensures that the two files share the same filesystem, thus avoiding cross-filesystem synchronization issues. Thanks to Charles Cazabon. ¶

0.3.2

Released: Thu Mar 11 2010

- Calling a def from the top, via template.get_def(...).render() now checks the argument signature the same way as it did in 0.2.5, so that TypeError is not raised. reopen of ¶ References: #116

0.3.1

Released: Sun Mar 7 2010

- Fixed incorrect dir name in setup.py ¶ References: #129

0.3.0

Released: Fri Mar 5 2010

- Python 2.3 support is dropped. ¶ References: #123
- Python 3 support is added ! See README.py3k for installation and testing notes. ¶ References: #119
- Unit tests now run with nose. ¶ References: #127
- Source code escaping has been simplified. In particular, module source files are now generated with the Python "magic encoding comment", and source code is passed through mostly unescaped, except for that code which is regenerated from parsed Python source. This fixes usage of unicode in <%namespace:defname> tags. ¶ References: #99
- RichTraceback(), html_error_template().render(), text_error_template().render() now accept "error" and "traceback" as optional arguments, and these are now actually used. ¶ References: #122
- The exception output generated when format_exceptions=True will now be as a Python unicode if it occurred during render_unicode(), or an encoded string if during render(). ¶
- A percent sign can be emitted as the first non-whitespace character on a line by escaping it as in "%%". ¶ References: #112
- Template accepts empty control structure, i.e. % if: %endif, etc. ¶ References: #94
- The <%page args> tag can now be used in a base inheriting template - the full set of render() arguments are passed down through the inherits chain. Undeclared arguments go into **pageargs as usual. ¶ References: #116
- defs declared within a <%namespace> section, an uncommon feature, have been improved. The defs no longer get doubly-rendered in the body() scope, and now allow local variable assignment without breakage. ¶ References: #109
- Windows paths are handled correctly if a Template is passed only an absolute filename (i.e. with c: drive etc.) and no URI - the URI is converted to a forward-slash path and module_directory is treated as a windows path. ¶ References: #128
- TemplateLookup raises TopLevelLookupException for a given path that is a directory, not a filename, instead of passing through to the template to generate IOError. ¶ References: #73

0.2.6

no release date

- Fix mako function decorators to preserve the original function's name in all cases. Patch from Scott Torborg. ¶
- Support the <%namespacename:defname> syntax in the babel extractor. ¶ References: #118
- Further fixes to unicode handling of .py files with the html_error_template. ¶ References: #88

0.2.5

Released: Mon Sep 7 2009

- Added a "decorator" kw argument to <%def>, allows custom decoration functions to wrap rendering callables. Mainly intended for custom caching algorithms, not sure what other uses there may be (but there may be). Examples are in the "filtering" docs. ¶

- When Mako creates subdirectories in which to store templates, it uses the more permissive mode of 0775 instead of 0750, helping out with certain multi-process scenarios. Note that the mode is always subject to the restrictions of the existing umask. ¶ References: #101
- Fixed namespace.__getattr__() to raise AttributeError on attribute not found instead of RuntimeError. ¶ References: #104
- Added last_modified accessor to Template, returns the time.time() when the module was created. ¶ References: #97
- Fixed lexing support for whitespace around '=' sign in defs. ¶ References: #102
- Removed errant "lower()" in the lexer which was causing tags to compile with case-insensitive names, thus messing up custom <%call> names. ¶ References: #108
- added "mako.__version__" attribute to the base module. ¶ References: #110

0.2.4

Released: Tue Dec 23 2008

- Fixed compatibility with Jython 2.5b1. ¶

0.2.3

Released: Sun Nov 23 2008

- the <%namespacename:defname> syntax described at http://techspot.zzzeek.org/?p=28 has now been added as a built in syntax, and is recommended as a more modern syntax versus <%call expr="expression">. The %call tag itself will always remain, with <%namespacename:defname> presenting a more HTML-like alternative to calling defs, both plain and nested. Many examples of the new syntax are in the "Calling a def with embedded content" section of the docs. ¶
- added support for Jython 2.5. ¶
- cache module now uses Beaker's CacheManager object directly, so that all cache types are included. memcached is available as both "ext:memcached" and "memcached", the latter for backwards compatibility. ¶
- added "cache" accessor to Template, Namespace. e.g. ${local.cache.get('somekey')} or template.cache.invalidate_body() ¶
- added "cache_enabled=True" flag to Template, TemplateLookup. Setting this to False causes cache operations to "pass through" and execute every time; this flag should be integrated in Pylons with its own cache_enabled configuration setting. ¶
- the Cache object now supports invalidate_def(name), invalidate_body(), invalidate_closure(name), invalidate(key), which will remove the given key from the cache, if it exists. The cache arguments (i.e. storage type) are derived from whatever has been already persisted for that template. ¶ References: #92
- For cache changes to work fully, Beaker 1.1 is required. 1.0.1 and up will work as well with the exception of cache expiry. Note that Beaker 1.1 is **required** for applications which use dynamically generated keys, since previous versions will permanently store state in memory for each individual key, thus consuming all available memory for an arbitrarily large number of distinct keys. ¶
- fixed bug whereby an <%included> template with <%page> args named the same as a __builtin__ would not honor the default value specified in <%page> ¶ References: #93
- fixed the html_error_template not handling tracebacks from normal .py files with a magic encoding comment ¶ References: #88

- RichTraceback() now accepts an optional traceback object to be used in place of sys.exc_info()[2]. html_error_template() and text_error_template() accept an optional render()-time argument "traceback" which is passed to the RichTraceback object. ¶
- added ModuleTemplate class, which allows the construction of a Template given a Python module generated by a previous Template. This allows Python modules alone to be used as templates with no compilation step. Source code and template source are optional but allow error reporting to work correctly. ¶
- fixed Python 2.3 compat. in mako.pyparser ¶ References: #90
- fix Babel 0.9.3 compatibility; stripping comment tags is now optional (and enabled by default). ¶

0.2.2

Released: Mon Jun 23 2008

- cached blocks now use the current context when rendering an expired section, instead of the original context passed in ¶ References: #87
- fixed a critical issue regarding caching, whereby a cached block would raise an error when called within a cache-refresh operation that was initiated after the initiating template had completed rendering. ¶

0.2.1

Released: Mon Jun 16 2008

- fixed bug where 'output_encoding' parameter would prevent render_unicode() from returning a unicode object. ¶
- bumped magic number, which forces template recompile for this version (fixes incompatible compile symbols from 0.1 series). ¶
- added a few docs for cache options, specifically those that help with memcached. ¶

0.2.0

Released: Tue Jun 3 2008

- Speed improvements (as though we needed them, but people contributed and there you go): ¶
- added "bytestring passthru" mode, via *disable_unicode=True* argument passed to Template or TemplateLookup. All unicode-awareness and filtering is turned off, and template modules are generated with the appropriate magic encoding comment. In this mode, template expressions can only receive raw bytestrings or Unicode objects which represent straight ASCII, and render_unicode() may not be used if multibyte characters are present. When enabled, speed improvement around 10-20%. (courtesy anonymous guest) ¶ References: #77
- inlined the "write" function of Context into a local template variable. This affords a 12-30% speedup in template render time. (idea courtesy same anonymous guest) ¶ References: #76
- New Features, API changes: ¶
- added "attr" accessor to namespaces. Returns attributes configured as module level attributes, i.e. within <%! %> sections. i.e.:

  # somefile.html <%!

      foo = 27

%>

# some other template <%namespace name="myns" file="somefile.html"/> ${myns.attr.foo}

The slight backwards incompatibility here is, you can't have namespace defs named "attr" since the "attr" descriptor will occlude it. ¶ References: #62

- cache_key argument can now render arguments passed directly to the %page or %def, i.e. <%def name="foo(x)" cached="True" cache_key="${x}"/> ¶ References: #78
- some functions on Context are now private: _push_buffer(), _pop_buffer(), caller_stack._push_frame(), caller_stack._pop_frame(). ¶
- added a runner script "mako-render" which renders standard input as a template to stdout ¶ References: #56, #81
- **[bugfixes]** can now use most names from __builtins__ as variable names without explicit declaration (i.e. 'id', 'exception', 'range', etc.) ¶ References: #83, #84
- **[bugfixes]** can also use builtin names as local variable names (i.e. dict, locals) (came from fix for) ¶ References: #84
- **[bugfixes]** fixed bug in python generation when variable names are used with identifiers like "else", "finally", etc. inside them ¶ References: #68
- **[bugfixes]** fixed codegen bug which occurred when using <%page> level caching, combined with an expression-based cache_key, combined with the usage of <%namespace import="*"/> - fixed lexer exceptions not cleaning up temporary files, which could lead to a maximum number of file descriptors used in the process ¶ References: #69
- **[bugfixes]** fixed issue with inline format_exceptions that was producing blank exception pages when an inheriting template is present ¶ References: #71
- **[bugfixes]** format_exceptions will apply the encoding options of html_error_template() to the buffered output ¶
- **[bugfixes]** rewrote the "whitespace adjuster" function to work with more elaborate combinations of quotes and comments ¶ References: #75

0.1.10

no release date

- fixed propagation of 'caller' such that nested %def calls within a <%call> tag's argument list propigates 'caller' to the %call function itself (propigates to the inner calls too, this is a slight side effect which previously existed anyway) ¶
- fixed bug where local.get_namespace() could put an incorrect "self" in the current context ¶
- fixed another namespace bug where the namespace functions did not have access to the correct context containing their 'self' and 'parent' ¶

0.1.9

no release date

- filters.Decode filter can also accept a non-basestring object and will call str() + unicode() on it ¶ References: #47
- comments can be placed at the end of control lines, i.e. if foo: # a comment,, thanks to Paul Colomiets ¶ References: #53

- fixed expressions and page tag arguments and with embedded newlines in CRLF templates, follow up to, thanks Eric Woroshow ¶ References: #16
- added an IOError catch for source file not found in RichTraceback exception reporter ¶ References: #51

0.1.8

Released: Tue Jun 26 2007

- variable names declared in render methods by internal codegen prefixed by "__M_" to prevent name collisions with user code ¶
- added a Babel (http://babel.edgewall.org/) extractor entry point, allowing extraction of gettext messages directly from mako templates via Babel ¶ References: #45
- fix to turbogears plugin to work with dot-separated names (i.e. load_template('foo.bar')). also takes file extension as a keyword argument (default is 'mak'). ¶
- more tg fix: fixed, allowing string-based templates with tgplugin even if non-compatible args were sent ¶ References: #35

0.1.7

Released: Wed Jun 13 2007

- one small fix to the unit tests to support python 2.3 ¶
- a slight hack to how cache.py detects Beaker's memcached, works around unexplained import behavior observed on some python 2.3 installations ¶

0.1.6

Released: Fri May 18 2007

- caching is now supplied directly by Beaker, which has all of MyghtyUtils merged into it now. The latest Beaker (0.7.1) also fixes a bug related to how Mako was using the cache API. ¶
- fix to module_directory path generation when the path is "./" ¶ References: #34
- TGPlugin passes options to string-based templates ¶ References: #35
- added an explicit stack frame step to template runtime, which allows much simpler and hopefully bug-free tracking of 'caller', fixes ¶ References: #28
- if plain Python defs are used with <%call>, a decorator @runtime.supports_callable exists to ensure that the "caller" stack is properly handled for the def. ¶
- fix to RichTraceback and exception reporting to get template source code as a unicode object ¶ References: #37
- html_error_template includes options "full=True", "css=True" which control generation of HTML tags, CSS ¶ References: #39
- added the 'encoding_errors' parameter to Template/TemplateLookup for specifying the error handler associated with encoding to 'output_encoding' ¶ References: #40
- the Template returned by html_error_template now defaults to output_encoding=sys.getdefaultencoding(), encoding_errors='htmlentityreplace' ¶ References: #37
- control lines, i.e. % lines, support backslashes to continue long lines (#32) ¶

- fixed codegen bug when defining <%def> within <%call> within <%call> ¶
- leading utf-8 BOM in template files is honored according to pep-0263 ¶

0.1.5

Released: Sat Mar 31 2007

- AST expression generation - added in just about everything expression-wise from the AST module ¶ References: #26
- AST parsing, properly detects imports of the form "import foo.bar" ¶ References: #27
- fix to lexing of <%docs> tag nested in other tags ¶
- fix to context-arguments inside of <%include> tag which broke during 0.1.4 ¶ References: #29
- added "n" filter, disables *all* filters normally applied to an expression via <%page> or default_filters (but not those within the filter) ¶
- added buffer_filters argument, defines filters applied to the return value of buffered/cached/filtered %defs, after all filters defined with the %def itself have been applied. allows the creation of default expression filters that let the output of return-valued %defs "opt out" of that filtering via passing special attributes or objects. ¶

0.1.4

Released: Sat Mar 10 2007

- got defs-within-defs to be cacheable ¶
- fixes to code parsing/whitespace adjusting where plain python comments may contain quote characters ¶ References: #23
- fix to variable scoping for identifiers only referenced within functions ¶
- added a path normalization step to lookup so URIs like "/foo/bar/../etc/../foo" pre-process the ".." tokens before checking the filesystem ¶
- fixed/improved "caller" semantics so that undefined caller is "UNDEFINED", propigates __nonzero__ method so it evaulates to False if not present, True otherwise. this way you can say % if caller:n ${caller.body()}n% endif ¶
- <%include> has an "args" attribute that can pass arguments to the called template (keyword arguments only, must be declared in that page's <%page> tag.) ¶
- <%include> plus arguments is also programmatically available via self.include_file(<filename>, **kwargs) ¶
- further escaping added for multibyte expressions in %def, %call attributes ¶ References: #24

0.1.3

Released: Wed Feb 21 2007

- ***Small Syntax Change*** - the single line comment character is now *two* hash signs, i.e. "## this is a comment". This avoids a common collection with CSS selectors. ¶
- the magic "coding" comment (i.e. # coding:utf-8) will still work with either one "#" sign or two for now; two is preferred going forward, i.e. ## coding:<someencoding>. ¶
- new multiline comment form: "<%doc> a comment </%doc>" ¶

- UNDEFINED evaluates to False ¶
- improvement to scoping of "caller" variable when using <%call> tag ¶
- added lexer error for unclosed control-line (%) line ¶
- added "preprocessor" argument to Template, TemplateLookup - is a single callable or list of callables which will be applied to the template text before lexing. given the text as an argument, returns the new text. ¶
- added mako.ext.preprocessors package, contains one preprocessor so far: 'convert_comments', which will convert single # comments to the new ## format ¶

0.1.2

Released: Thu Feb 1 2007

- fix to parsing of code/expression blocks to insure that non-ascii characters, combined with a template that indicates a non-standard encoding, are expanded into backslash-escaped glyphs before being AST parsed ¶ References: #11
- all template lexing converts the template to unicode first, to immediately catch any encoding issues and ensure internal unicode representation. ¶
- added module_filename argument to Template to allow specification of a specific module file ¶
- added modulename_callable to TemplateLookup to allow a function to determine module filenames (takes filename, uri arguments). used for ¶ References: #14
- added optional input_encoding flag to Template, to allow sending a unicode() object with no magic encoding comment ¶
- "expression_filter" argument in <%page> applies only to expressions ¶
- **["unicode"]** added "default_filters" argument to Template, TemplateLookup. applies only to expressions, gets prepended to "expression_filter" arg from <%page>. defaults to, so that all expressions get stringified into u" by default (this is what Mako already does). By setting to [], expressions are passed through raw. ¶
- added "imports" argument to Template, TemplateLookup. so you can predefine a list of import statements at the top of the template. can be used in conjunction with default_filters. ¶
- support for CRLF templates...whoops ! welcome to all the windows users. ¶ References: #16
- small fix to local variable propigation for locals that are conditionally declared ¶
- got "top level" def calls to work, i.e. template.get_def("somedef").render() ¶

0.1.1

Released: Sun Jan 14 2007

- buffet plugin supports string-based templates, allows ToscaWidgets to work ¶ References: #8
- AST parsing fixes: fixed TryExcept identifier parsing ¶
- removed textmate tmbundle from contrib and into separate SVN location; windows users cant handle those files, setuptools not very good at "pruning" certain directories ¶
- fix so that "cache_timeout" parameter is propigated ¶
- fix to expression filters so that string conversion (actually unicode) properly occurs before filtering ¶
- better error message when a lookup is attempted with a template that has no lookup ¶

- implemented "module" attribute for namespace ¶
- fix to code generation to correctly track multiple defs with the same name ¶
- "directories" can be passed to TemplateLookup as a scalar in which case it gets converted to a list ¶ References: #9

CHAPTER 11

Indices and Tables

- genindex
- search

Index