1. INTRODUCTION

This document aims to be the authoritative source of information about Spyce, usable as a comprehensive refence, a user guide and a tutorial. It should be at least skimmed from beginning to end, so you have at least an idea of the functionality available and can refer back for more details as needed.

Spyce is a server-side language that supports elegant and efficient Python-based dynamic HTML generation. Spyce allows embedding Python in pages similar to how JSP embeds Java, but Spyce is far more than a JSP clone. Out of the box, Spyce provides development as rapid as other modern frameworks like Rails, but with an cohesive design rather than a morass of special cases.

Spyce's modular design makes it very flexible and extensible. It can also be used as a command-line utility for static text pre-processing or as a web-server proxy.

Spyce's performance is comparable to the other solutions in its class.

Note: This manual assumes a knowledge of Python and focusses exclusively on Spyce. If you do not already know Python, it is easy to learn via this short tutorial, and has extensive documentation.

1.1. Rationale / competitive analysis

This section is somewhat dated. We plan to update it soon.

A natural question to ask is why one would choose Spyce over JSP, ASP, PHP, or any of the other HTML scripting languages that perform a similar function. We compare Spyce with an array of exising tools:

- ***Java Server Pages, JSP,*** is a widely popular, effective and well-supported solution based on Java Servlet technology. Spyce differs from JSP in that it embeds Python code among the HTML, thus providing a number of advantages over Java.
    - Python is a high-level scripting language, where rapid prototyping is syntactically easier to perform.
    - There is no need for a separate "expression langauge" in Spyce; Python is well-suited for both larger modules and active tag scripting.
    - Python is interpreted and latently typed, which can be advantageous for prototyping, especially in avoiding unnecessary binary incompatibility of classes for minor changes.
    - Spyce code is of first-order in the Spyce language, unlike JSP, which allows you to create useful Spyce lambda functions.
    - Creating new active tags and modules is simpler in Spyce than in JSP.
    - Spyce is better-integrated than JSP; to get similar functionality in JSP, you have to add JSF (Java Server Faces) and Tiles, or equivalents.
- ***PHP*** is another popular webserver module for dynamic content generation. The PHP interpreter engine and the language itself were explicitly designed for the task of dynamic HTML generation, while Python is a general-purpose scripting language.
    - Spyce leverages from the extensive development effort in Python: since any Python library can be imported and reused, Spyce does not need to rebuild

    many of the core function libraries that have been implemented by the PHP project.
    - Use of Python often simplifies integration of Spyce with existing system environments.
    - Spyce code is also first-order in the Spyce language and Spyce supports active tags.
    - Spyce is modular in its design, allowing users to easily extend its base functinality with add-on modules.
    - The Spyce engine can be run from the command-line, which allows Spyce to be used as an HTML preprocessor.

  Spyce, like PHP, can run entirely within the process space of a webserver or via CGI (as well as other web server adapters), and has been benchmarked to be competitive in performance.

- ***ASP.NET*** is a Microsoft technology popular with Microsoft Internet Information Server (IIS) users. Visual Basic .NET and C# are both popular implementation languages.
    - Spyce provides the power of the ASP.NET "component" development style without trying to pretend that web applications live in a stateful, event-driven environment. This is a leaky abstraction that causes ASP.NET to have a steep learning curve while the user learns where the rough edges are.
    - ASP.NET is not well-supported outside the IIS environment. Spyce can currently run as a standalone or proxy server, under mod_python (Apache), or under CGI and FastCGI, which are supported in the majority of web server environments. Adapters have also been written for Xitami, Coil, Cheetah -- other web servers and frameworks.
    - Spyce is open-source, and free.
- ***WebWare with Python Server Pages, PSP,*** is another Python-based open-source development. PSP is similar in design to the Spyce language, and shares many of the same benefits. Some important differences include
    - Spyce supports both Python chunks (indented Python) as well as PSP-style statements (braced Python).
    - Spyce supports active tags and component-based development
    - Spyce code is first-order in the Spyce language

  PSP is also an integral part of WebWare, an application-server framework similar to Tomcat Java-based application server of the Apache Jakarta project. Spyce is to WebWare as JSP is to Tomcat. Spyce is far simpler to install and run than WebWare (in the author's humble opinion), and does not involve notions such as application contexts. It aims to do only one thing well: provide a preprocessor and runtime engine for the dynamic generation of HTML using embedded Python.

- ***Zope*** is an object-oriented open-source application server, specializing in "content management, portals, and custom applications." Zope is the most mature Python web application development environment, but to a large degree suffers from second-system syndrome. In the author's opinion, Zope is to a large degree responsible for the large number of python web environments: a few years ago, it was de rigeur for talented programmers to try Zope, realize it was a mess, and go off to write their own framework.

Zope provides a scripting language called DHTML and can call extensions written in Perl or Python. Spyce embeds Python directly in the HTML, and only Python. It is an HTML-embedded language, not an application server.

Spyce strikes a unique balance between power and simplicity. Many users have said that this is "exactly what they have been waiting for". Hopefully, this is the correct point in the design space for your project as well.

1.2. Design Goals

As a Spyce user, it helps to understand the broad design goals of this tool. Spyce is designed to be:

- **Minimalist:** The philosophy behind the design of Spyce is only to include features that particularly enhance its functionality over the wealth that is already available from within Python. One can readily import and use Python modules for many functions, and there is no need to recode large bodies of functionality.
- **Powerful:** Spyce aims to free the programmer from as much "plumbing"-style drudgery as possible through features such as Active Handlers and reusable Active Tags.
- **Modular:** Spyce is built to be extended with Spyce modules and Active Tags that provide additional functionality over the core engine capabilities and standard Python modules. New features in the core engine and language are rationalised against the option of creating a new module or a new tag library. Standard Spyce modules and tag libraries are those that are considered useful in a general setting and are included in the default Spyce distribution. Users and third-parties are encouraged to develop their own Spyce modules.
- **Intuitive:** Obey user expectations. Part of this is avoiding special cases.
- **Convenient:** Using Spyce should be made as efficient as possible. This, for example, is the reason behind the choice of [[as delimeters over alternatives such as <? (php) and <% (jsp). (However, ASP/JSP-style delimeters are also supported, so if you're used to that style and like it, feel free to continue using it with Spyce.) Functions and modules are also designed with as many defaults as possible. There are no XML configuration files in Spyce.
- **Single-purpose:** To be the best, most versatile, wildly-popular Python-based dynamic HTML engine. Nothing more; nothing less.
- **Fast:** Performance is important. It is expected that Spyce will perform comparably with any other dynamic, scripting solutions available.

Now, let's start using Spyce...

2. LANGUAGE

The basic structure of a Spyce script is an HTML file with embeddings. There are six types of possible embeddings among the plain HTML text:

- <taglib:name attr1=val1 ...>
  Active tags may include presentation and action code.
- **[[--** Spyce comment **--]]**
  Enclosed code is elided from the compiled Spyce class.

- **[[\** Python chunk **]]**
  Embed python in your code.
- **[[!** Python class chunk **]]**
  Like chunks, but at the class level rather than the spyceProcess method.
- **[[** Python statement(s) **]]**
  Like chunks, but may include braces to indicate that the block should continue after the]].
- **[[=** Python expression **]]**
  Output the result of evaluating the given expression.
- **[[.** Spyce directive **]]**
  Pass options to the Spyce compiler.
- **[[spy** lambda **]]**
  Allows dynamic compilation of Spyce code to a python function.

Each Spyce tag type has a unique beginning delimeter, namely **[[**, **[[\**, **[[=**, **[[.** or **[[--**. All tags end with **]]**, except comment tags, which end with **--]]**.

Since **[[** and **]]** are special Spyce delimeters, one would escape them as \**[[** and \**]]** for use in HTML text. They can not be escaped within Python code, but the string expressions **("["*2)** and **("]"*2)**, or equivalent expressions, can be used instead, or the brackets can be conveniently separated with a space in the case of list or slicing expressions.

2.1. Plain HTML and Active Tags

Static plain HTML strings are printed as they are encountered. Depending on the compacting mode of the Spyce compiler, some whitespace may be eliminated. The Spyce transform module, for example, may further pre-processes this string, by inserting transformations into the output pipe. This is useful, for example, for dynamic compression of the script result.

The Spyce language supports tag libraries. Once a tag library is imported under some name, mytags, then all static HTML tags of the form <mytags:foo ... > become "active". That is, code from the tag library is executed at that point in the document. Tags can control their output, conditionally skip or loop the execution of their bodies, and can interact with other active tags in the document. They are similar, in spirit and functionality, to JSP tags. Tag libraries and modules (discussed later) can both considerably reduce the amount of code on a Spyce page, and increase code reuse and modularity.

2.2. Spyce Comments

Syntax: [[-- comment --]]

Spyce comments are ignored, and do not produce any output, meaning that they will not appear at the browser even in the HTML source. The first line of a Spyce file, if it begins with the characters **#!**, is also considered a comment, by Unix scripting convention. Spyce comments do *not* nest.

2.3. Spyce Directives

Syntax: [[. directive]]

Spyce directives directly affect the operation of the Spyce compiler. There is a limited set of directives, listed and explained below:

- **[[.compact mode=*mode*]] :**
  Spyce can output the static HTML strings in various modes of compaction, which can both save bandwidth and improve download times without visibly affecting the output. Compaction of static HTML strings is performed once when the input Spyce file is compiled, and there is no additional run-time overhead beyond that. Dynamically generated content from Python code tags and expressions is not compacted nor altered in any way. Spyce can operate in one of the compaction modes listed below. One can use the **compact** tag to change the compaction mode from that point in the file forwards.
  - **off**: No compaction is performed. Every space and newline in the static HTML strings is preserved.
  - **space**: Space compaction involves reducing any consecutive runs of spaces or tabs down to a single space. Any spaces or tabs at the beginning of a line are eliminated. These transformations will not affect HTML output, barring the <pre> tag, but can considerably reduce the size of the emitted text.
  - **line**: Line compaction eliminates any (invisible) trailing whitespace at the end of lines. More significantly it improves the indented presentation of HTML, by ignoring any lines that do not contain any static text or expression tags. Namely, it removes all the whitespace, including the line break, surrounding the code or directives on that line. This compaction method usually "does the right thing", and produces nice HTML without requiring tricky indentation tricks by the developer. It is, therefore, the **initial** compaction mode.
  - **full**: Full compaction applies both space and line compaction. If the optional mode attribute is omitted, full compaction mode is the **default**value assumed.
- **[[.import name=*name* from=*file* as=*name* args=*arguments*]] :**
  The import directive loads and defines a Spyce module into the global context. (The [[.module ...]]directive is synonymous.) A Spyce module is a Python file, written specifically to interact with Spyce. The name parameter is required, specifying the name of the Python class to load. The file parameter is optional, specifying the file where the named class is to be found. If omitted, file will equal name.py. The file path can be absolute or relative. Relative paths are scanned in the Spyce home, user-configurable server path directories and current script directory, in that order. Users are encouraged to name or prefix their modules uniquely so as not to be masked by system modules or tag libraries. The asparameter is optional, and specifies the name under which the module will be installed in the global context. If omitted, this parameter defaults to the name parameter. Lastly, the optional args parameter provides arguments to be passed to the module initialization function. All Spyce modules are **start()**ed before Spyce processing begins, **init()**ed at the point where the directive is placed in the code, and **finish()**ed after Spyce processing terminates. It is convention to place modules at, or near, the very top of the file unless the location of initialization is relevant for the functioning of the specific module.

  [[.import names="*name1,name2,...*"]] :
  An alternative syntax allows convenient loading of multiple Spyce modules. One can not specify non-standard module file locations, nor rename the modules using this syntax.

- **[[.taglib name=*name* from=*file* as=*name*]] :**
  The taglib directive loads a Spyce tag library. A Spyce tag library is a Python file, written specifically to interact with Spyce. The name parameter specifies the name of the Python class to load if using a 1.x-style taglib; otherwise it is ignored. The file parameter is optional, specifying the file where the named class is to be found. If omitted, file will equal name.py. The file path can be absolute or relative. Relative paths are scanned in the Spyce home, user-configurable server path directories and current script directory, in that order. Users are encouraged to name or prefix their tag libraries uniquely so as not to be masked by system tag libraries and modules.
  The **as** parameter is optional, and specifies the unique tag prefix that will be used to identify the tags from this library. If omitted, this parameter defaults to the name parameter. It is convention to place tag library directives at, or near, the very top of the file. The tags only become active after the point of the tag library directive.

  Also note that the configuration parameter *globaltags* allows you to set up tag libraries globally, freeing you from having to specify the taglib directive on each page that uses a tag. By default, globaltags installs core under the spy: prefix, and form under the f: prefix. (Tag libraries specified in globaltags are only loaded if the Spyce compiler determines they are actually used on the page, so there is no performance difference between globaltags and manually setting up taglib for each page.)

There are some additional directives that are only legal when defining an active tag library.It is important to note that Spyce directives are processed at *compile* time, not during the execution of the script, much like directives in C, and other languages. In other words, they are processed as the Python code for the Spyce script is being produced, not as it is being executed. Consequently, it is not possible to include runtime values as parameters to the various directives.

2.4. Python Statements

Syntax: [[statement(s)]]

The contents of a code tag is one or more Python statements. The statements are executed when the page is emitted. There will be no output unless the statements themselves generate output.

The statements are separated with semi-colons or new lines, as in regular Python scripts. However, unlike regular Python code, Python statements *do**not** nest based on their level of indentation*. This is because indenting code properly in the middle of HTML is difficult on the developer. To alleviate this problem, Spyce supports a slightly modifed Python syntax: proper nesting of Spyce statements is achieved using begin- and end-braces: **{** and **}**, respectively. These **MUST** be used, because the compiler regenerates the correct indentation based on these markers alone. Even single-statement blocks of code must be wrapped with begin and end braces. (If you prefer to use Python-like indentation, read about chunks).

The following Spyce code, from the Hello World! example above:

```
[[ for i in range(10): { ]]
  [[=i]]
[[ } ]]
```

produces the following indented Python code:

```
for i in range(10):
  response.writeStatic('  ')
  response.writeExpr(i)
  response.writeStatic('\n')
```

Without the braces, the code produced would be unindented and, in this case, also invalid:

```
for i in range(10):
response.writeStatic('  ')
response.writeExpr(i)
response.writeStatic('\n')
```

Note how the indentation of the expression does not affect the indentation of the Python code that is produced; it merely changes the number of spaces in the writeStatic string. Also note that unbalanced open and close braces within a single tag are allowed, as in the example above, and they modify the indentation level outside the code tag. However, the braces must be balanced across an entire file. Remember: inside the [[...]] delimiters, **braces are always required** to change the indentation level.

2.5. Python Chunks

Syntax: [[\ Python chunk]]

There are many Python users that experience anguish, disgust or dismay upon reading the previous section: "Braces!? Give me real, indented Python!". These intendation zealots will be more comfortable using Python chunks, which is why Spyce supports them. Feel free to use Spyce statements or chunks inter-changeably, as the need arises.

A Python chunk is straight Python code, and the *internal* indentation is preserved. The entire block is merely outdented (or indented) as a whole, such that the first non-empty line of the block matches the indentation level of the context into which the chunk was placed. Thus, a Python chunk can not affect the indentation level outside its scope, but internal indentation is fully respected, relative to the first line of code, and braces ({, }) are not required, nor expected for anything but Python dictionaries. Since the first line of code is used as an indentation reference, it is recommended that the start delimeter of the tag (i.e. the [[\) be placed on its own line, above the code chunk, as shown in the following example:

```
[[\
    def printHello(num):
      for i in range(num):
        response.write('hello<br>')

    printHello(5)
```

```
                    ]]
```

Naturally, one should *not* use braces here for purposes of indentation, only for Python dictionaries. Additional braces will merely generate Python syntax errors in the context of chunks. To recap: a Python statement tag should contain braced Python; A Python chunk tag should contain regular indented Python.

2.6. Python Class Chunks

Syntax: [[! Python class chunk]]

Behind the scenes, your Spyce files are compiled into a class called spyceImpl. Your Spyce script runs in a method of this class called spyceProcess. Class chunks allow you to specify code to be placed inside the class, but outside the main method, analogously to the "<%!" token in JSP code. (If you would like to see your Spyce file in compiled Python form, use the following command-line: **spyce.py -c myfile.spy**.)

This is primarily useful when defining active handlers without using a separate .py file: active handlers are the first thing that the spyceProcess calls, even before any "python chunks." For a handler callback to be visible at this stage, it needs to be defined at the class level. Class chunks to the rescue:

examples/handlerintro.spy

```
[[!
def calculate(self, api, x, y):
   self.result = x * y
]]

<spy:parent title="Active Handler example" />
<f:form>
   <f:text name="x:float" default="2" label="first value" />
   <f:text name="y:float" default="3" label="second value" />

   <f:submit handler="self.calculate" value="Multiply" />
</f:form>

<p>
Result: [[= hasattr(self, 'result') and self.result or '(no result yet)' ]]
</p>
```

Run this code

2.7. Python Expressions

Syntax: [[= expression]]

The contents of an expression tag is a Python expression. The result of that expression evaluation is printed using the its string representation. The Python object None is special cased to output as the empty string just as it is in the Python interactive shell. This is almost always more convenient when working with HTML. (If you really want a literal string 'None' emitted instead, use response.write in a statement or chunk.)

The Spyce transform module, can pre-processes this result, to assist with mundane tasks such as ensuring that the string is properly HTML-encoded, or formatted.

2.8. Spyce Lambdas

Syntax: [[spy *[params]* : *spyce lambda code*]]
or: [[spy! *[params]* : *spyce lambda code*]]

A nice feature of Spyce is that Spyce scripts are first-class members of the language. In other words, you can create a Spyce lambda (or function) in any of the Spyce Python elements (statements, chunks and expressions). These can then be invoked like regular Python functions, stored in variables for later use, or be passed around as paramaters. This feature is often very useful for templating (example shown below), and can also be used to implement more esoteric processing functionality, such as internationalization, multi-modal component frameworks and other kinds of polymorphic renderers.

It is instructive to understand how these functions are generated. The [[spy ... : ...]] syntax is first translated during compilation into a call to the define() function of the spylambda module. At runtime, this call compiles the Spyce code at the point of its definition, and returns a function. While the invocation of a Spyce lambda is reasonably efficient, it is certainly not as fast as a regular Python function invocation. The spycelambda can be memoized (explained in the spylambda module section) by using the [[spy! ... : ...]] syntax. However, even with this optimization one should take care to use Python lambdas and functions when the overhead of Spyce parsing and invocation is not needed.

Note that Spyce lambdas do not currently support nested variable scoping, nor default parameters. The global execution context (specifically, Spyce modules) of the Spyce lambda is defined at the point of its execution.

examples/spylambda.spy

```
[[\
 # table template
 table = [[spy! title, data:
   <table>
     <tr>
       [[for cell in title: {]]
         <td><b>[[=cell]]</b></td>
       [[}]]
     </tr>
     [[for row in data: {]]
       <tr>
         [[for cell in row: {]]
           <td>[[=cell]]</td>
         [[}]]
       </tr>
```

```
    [[}]]
  </table>
 ]]

 # table information
 title = ['Country', 'Size', 'Population', 'GDP per capita']
 data = [
  [ 'USA', '9,158,960', '280,562,489', '$36,300' ],
  [ 'Canada', '9,220,970', '31,902,268', '$27,700' ],
  [ 'Mexico', '1,923,040', '103,400,165', '$9,000' ],
 ]
]]

[[-- emit web page --]]
<html><body>
 [[ table(title, data) ]]
</body></html>
```

Run this code

2.9. ASP/JSP syntax

Finally, due to popular demand, because of current editor support and people who actually enjoy pains in their wrists, the Spyce engine will respect ASP/JSP-like delimeters. In other words, it will also recognize the following syntax:

- <%-- Spyce comment --%>
- <%@ Spyce directive %>
- <% Python statement(s) %>
- <%\ Python chunk %>
- <%! Python class chunk %>
- <%= Python expression %>
- <%**spy** [parameters] **:** spyce lambda code %>

The two sets of delimeters may be used interchangeably within the same file, though for the sake of consistency this is not recommended.

3. RUNTIME

Having covered the Spyce language syntax, we now move to describing the runtime processing. Each time a request comes in, the cache of compiled Spyce files is checked for the compiled version of the requisite Spyce file. If one is not found, the Spyce file is quickly read, transformed, compiled and cached for future use.

The compiled Spyce is initialized, then processed, then finalized. The initialization consists of initializing all the Spyce modules. The Spyce file is executed top-down, until the end is reached or an exception is thrown, whichever comes first. The finalization step then finalizes each module in reverse order of initialization, and any buffered output is automatically flushed.

3.1. Exceptions

The Spyce file is executed top-down, until the end of the file is reached, a valued is returned, or an exception is thrown, whichever comes first. If the code terminates via an unhandled exception, then it is caught by the Spyce engine. Depending on the exception type, different actions are taken:

- **spyceDone** can be raised at any time to stop the Spyce processing (without error) at that point. It is often used to stop further output, as in the example below that emits a binary image file. The spyceDone exception, however, is more useful for modules writers. In regular Spyce code one could simply issue a return statement, with the same effect.
- **spyceRedirect** is used by the redirect module. It causes the Spyce engine to immediately redirect the request to another Spyce file *internally*. Internally means that we do not send back a redirect to the browser, but merely clear the output buffer and start processing a new script.
- **All other exceptions** that occur at runtime will be processed via the Spyce error module. This module will emit a default error message, unless the user has installed some other error handler.

Note that non-runtime exceptions, such as exceptions caused by compile errors, missing files, access restrictions and the like, are handled by the server. The default server error handler can be configured via the server configuration file.

examples/gif.spy

```
[[.import name=include ]]
[[\
  # Spyce can also generate other content types
  # The following code displays the Spyce logo
  response.setContentType('image/gif')
  import os.path, spyce
  path = os.path.join(spyce.getServer().config.SPYCE_HOME, 'www', 'spyce.gif')
  response.write(include.dump(path, 1))
  raise spyceDone
]]
```

Run this code

3.2. Code Transformation

While the minutia of the code transformation that produces Python code from the Spyce sources is of no interest to the casual user, it has some slight, but important, ramifications on certain aspects of the Python language semantics when used inside a Spyce file.

The result of the Spyce compilation is some Python code, wherein the majority of the Spyce code actually resides in a single function called**spyceProcess**. If you are curious to see the result of a Spyce compilation, execute: "spyce -c".

It follows from the compilation transformation that:

- Any functions defined within the Spyce file are actually nested functions within the spyceProcess function.
- The use of **global** variables within Spyce code is not supported, but also not needed. If nested scoping is available (Python versions >2.1) then these variables will simply be available. If not, then you will need to pass variables into functions as default parameters, and will not be able to update them by value (standard Python limitations). It is good practice to store constants and other globals in a single class, or to to place them in a single, included file, or both.
- The global Spyce namespace is reserved for special variables, such as Spyce and Python modules. While the use of the keyword global is not explicitly checked, it will pollute this space and may result in unexpected behaviour or runtime errors.
- The lifetime of variables is the duration of a request. Variables with lifetimes longer than a single request can be stored using the pool module.

3.3. Dynamic Content

The most common use of Spyce is to serve dynamic HTML content, but it should be noted that Spyce can be used as a general purpose text engine. It can be used to generate XML, text and other output, as easily as HTML. In fact, the engine can also be used to generate dynamic binary data, such as images, PDF files, etc., if needed.

The Spyce engine can be installed in a number of different configurations that can produce dynamic output. Proxy server, mod_python, and FastCGI exhibit high performance; the CGI approach is slower, since a new engine must be created for each request. See the configuration section for details.**3.4. Static Content**

A nice feature of Spyce is that it can be invoked both from within a web server to process a web request dynamically and also from the command-line. The processing engine itself is the same in both cases. The command-line option is actually just a modified CGI client, and is often used to pre-process static content, such as this manual.

Some remarks regarding command-line execution specifics are in order. The request and response objects for a command-line request are connected to standard input and output, as expected. A minimal CGI-like environment is created among the other shell environment variables. Header and cookie lookups will return None and the engine will accept input on stdin for POST information, if requested. There is also no compiler cache, since the process memory is lost at the end of every execution.

Most commonly, Spyce is invoked from the command-line to generate static .html ouput. Spyce then becomes a rather handy and powerful .html preprocessing tool. It was used on this documentation to produce the consistent headers and footers, to include and highlight the example code snippets, etc...

The following makefile rule comes in handy:

```
%.html: %.spy
  spyce -o $@ $<
```

3.5. Command line

The full command-line syntax is:

```
Spyce 2.1
Command-line usage:
  spyce -c [-o filename.html] <filename.spy>
  spyce -w <filename.spy>                <-- CGI
  spyce -O filename(s).spy               <-- batch process
  spyce -l [-d file ]                    <-- proxy server
  spyce -h | -v
    -h, -?, --help        display this help information
    -v, --version         display version
    -o, --output          send output to given file
    -O                    send outputs of multiple files to *.html
    -c, --compile         compile only; do not execute
    -w, --web             cgi mode: emit headers (or use run_spyceCGI.py)
    -q, --query           set QUERY_STRING environment variable
    -l, --listen          run in HTTP server mode
    -d, --daemon          run as a daemon process with given pidfile
    --conf [file]         Spyce configuration file
To configure Apache, please refer to: spyceApache.conf
For more details, refer to the documentation.
  http://spyce.sourceforge.net
Send comments, suggestions and bug reports to <rimon-AT-acm.org>.
```

3.6. Configuration

Since there are a variety of very different installation alternatives for the Spyce engine, effort has been invested in consolidating all the various runtime configuration options. By default, the Spyce engine will search for a file called **spyceconf.py** in its installation directory. An alternative file location may be specified via the **--conf** command-line option.

The spyce configuration file is a valid python module; any python code may be used. To avoid duplication, we recommend starting with "from spyceconf import *" and override only select settings. One thing you cannot do from the config module is access the Spyce server object spyce.getServer(), since it has not been initialized yet.

You may access the loaded configuration module from Spyce scripts and from Python modules using the config attribute of the server object. Or, simply as the spyceConfig module, regardless of it actual file name. For example:

examples/config.spy
[[\

```
  import spyceConfig
  home = spyceConfig.SPYCE_HOME
]]

[[= home ]]
```

Run this code

Below is the configuration file that this server is running. The length of the file is primarily due to the thoroughness of the comments:

```
# NOTE: do note write code that directly imports this module
# (except when you are writing a custom configuration module.)
# This is a recipe for trouble, since spyce allows the user
# to specify the configuration module filename on the commandline.
# Instead, use import spyce; spyce.getServer().config.

import os, sys
import spycePreload

# Determine SPYCE_HOME dynamically.
# (you can hardcode SPYCE_HOME if you really want to, but it shouldn't be necessary.)
SPYCE_HOME = spycePreload.guessSpyceHome()

# The spyce path determines which directories are searched for when
# loading modules (with [[.import]]) and tag libraries (with [[.taglib]]
# and the globaltags configuration later in this file.
#
# By default, the Spyce installation directory is always searched
# first. Any directories in the SPYCE_PATH environment are also
# searched.
#
# If you need to import from .py modules in nonstandard locations
# (i.e., not in your python installation's library directory),
# you will want to add their directories to sys.path as well, as
# done here for the error module.  (However, Spyce automagically
# changes sys.path dynamically so you will always be able to import
# modules in the same directory as your currently-processing .spy file.)
#
# path += ['/usr/spyce/inc/myapplication', '/var/myapp/lib']
path = [os.path.join(SPYCE_HOME, 'modules'), os.path.join(SPYCE_HOME, 'contrib',
'modules')]
path.append(os.path.join(SPYCE_HOME, 'tags'))
path.append(os.path.join(SPYCE_HOME, 'contrib', 'tags'))
if os.environ.has_key('SPYCE_PATH'):
    path += os.environ['SPYCE_PATH'].split(os.pathsep)
# provide originalsyspath so if someone wants to maintain a config file via
# "from spyceconf import *"
# he can remove the above modifications if desired.
```

```
originalsyspath = list(sys.path)
sys.path.extend(path)

# The globaltags option specifies a group of tag libraries that will be autoloaded
# as if [[.taglib name=libname from=file as=prefix]] were specified in every .spy
# file.. (There is no performance hit if the library is not used on a page;
# the Spyce compiler optimizes it out.)
#
# The format is ('libname', 'file', 'prefix').
# For a 2.0-style tag library, the libname attribute is ignored.  Passing None is fine.
#
# globaltags.append(('mytag', 'mytaglib.py', 'my'))
# globaltags.append((None, 'taglib2.py', 'my2'))
globaltags = [
    ('core', 'core.py', 'spy'),
    ('form', 'form.py', 'f'),
    (None, 'render.spi', 'render'),
    ]

# The default parent template is the one that is used by <spy:parent>
# if no src attribute is given, specified as an absolute url.
defaultparent = "/parent.spi"

# The errorhandler option sets the server-level error handler.  These
# errors include spyce.spyceNotFound, spyce.spyceForbidden,
# spyce.spyceSyntaxError and spyce.pythonSyntaxError.  (file-level
# error handling is defined within Spyce scripts using the error module.)
#
# The server will call the error handler as errorhandler(request, response, error).
#
# Please look at the default function if you are considering writing your own
# server error handler.
import error
errorhandler = error.serverHandler

# The pageerror option sets the default page-level error handler.
# "Page-level" means all runtime errors that occur during the
# processing of a Spyce script (i.e. after the compilation phase has
# completed successfully)
#
# The format of this option is one of:
#   ('string', 'MODULE', 'VARIABLE')
#   ('file', 'URL')
# (This format is used since the error template must be transformed into
# compiled spyce code, which can't be done before the configuration file
# is completely loaded.)
#
# Please refer to the default template to see how to define your own
# page-level error handlers.
#
```

```
# pageerrortemplate = ('file', '/error.spy')
pageerrortemplate = ('string', 'error', 'defaultErrorTemplate')

# The cache option affects the underlying cache mechanism that the
# server uses to maintain compiled Spyce scripts. Currently, Spyce
# supports two cache handlers:
#
#   cache = 'memory'
#   OR
#   cache = 'file'
#   cachedir = '/tmp' # REQUIRED: directory in which to store compiled files
#
# Why store the cache in the filesystem instead of memory?  The main
# reason is if you are running under CGI or mod_python.  Under
# mod_python, Apache will kick off a number of separate spyce
# processes; each would have its own memory cache; using the
# filesystem avoids wasteful duplication.  (Pointing the cachedir to a
# ramdisk makes it almost as fast as a memory cache.)  Under CGI of
# course, the python process isn't persistent so file caching is the
# only option to avoid expensive recompilation with each request.
#
# If you are running multiple Spyce instances on the same machine,
# they cannot share the same cachedir.  Give each a different cachedir,
# or use the in-memory cache type.
cache = 'memory'

# The check_mtime option affects the caching of compiled Spyce code.
# When True, Spyce will check file timestamps with each request and
# recompile if they have been modified.  Setting this to False can
# speed up a "production server" but you will have to restart the
# server and (if using a file cache) clear out the cachedir
# to have changes made in the spyce code take effect.
check_mtime = True

# The debug option turns on a LOT of logging to stderr.
debug = False

# The globals section defines server-wide constants. The hashtable is
# accessible as "pool" within any Spyce file (with the pool
# method loaded), or as self._api.getServerGlobals() within any Spyce
# module.
#
# globals = {'name': "My Website", 'four': 2+2}
globals = {}

# You may wish to pre-load various Python modules during engine initialization.
# Once imported, they will be in the python module cache.
#
# (You may of course use normal imports at any time in this configuration script;
# however, imports specified here are run after the Spyce server is
# completely initialized, making it safe to access the server internals via
# import spyce; spyce.getServer()...)
#
# imports = ['myModule', 'myModule2']
imports = []

# The root option defines the path from which Spyce requests are processed.
# I.e., when a request for http://yourserver/path/foo.spy arrives,
# Spyce looks for foo.spy in <root>/path/.
#
# root = '/var/www/html'
root = os.path.join(SPYCE_HOME, 'www')
# feel free to comment this next line out if you don't have python modules (.py) in your web
root
sys.path.append(root)

# some parts of spyce may need to create temporary files; usually the default is fine.
# BUT if you do override this, be sure to also override other parts of the config
# that reference it.  (currently just session_store)
import tempfile
tmp = tempfile.gettempdir()

# active tag to render form validation errors
validation_render = 'render:validation'


#####
# database connection
#####

from sqlalchemy.ext.sqlsoup import SqlSoup

# Examples:
# db = SqlSoup('postgres://user:pass@localhost/dbname')
# db = SqlSoup('sqlite:///my.db')
# db = SqlSoup('mysql://user:pass@localhost/dbname')
#
# SqlSoup takes the same URLs as an SqlAlchemy Engine.  See
# http://www.sqlalchemy.org/docs/dbengine.myt#dbengine_establishing
# for more examples.
try:
  db = SqlSoup('sqlite:///www/demos/to-do/todo.db')
except:
  db = None

#####
# session options -- see docs/mod_session.html for details
#####

import session
```

```
session_store = session.DbmStore(tmp)
# session_store = session.MemoryStore()

session_path = '/'

session_expire = 24 * 60 * 60 # seconds

#####
# login options
#####

# The spyce login system uses the session storage
# defined above; a key called _spy_login will be added to each session.

# validators must be a function that takes login and password as
# arguments, and returns a pickle-able object (usually int or string)
# representing the ID of the logged in user, or None if login fails.
#
# You'll need to supply your own to hook into your database or other
# validation system; see the pyweboff config.py for an example of doing
# this.
def nevervalidator(login, password):
        return None

def testvalidator(login, password):
    if login == 'spyce' and password == 'spyce':
        return 2
    return None

login_defaultvalidator = testvalidator

# How to store login tokens.  FileStorage comes with Spyce,
# but it's easy to create your own Storage class if you want to put them
# in your database, for instance.
from _coreutil import FileStorage
# It's not a good idea to put login-tokens off of www/ in production!
# It's just done this way here to keep the spyce source tree relatively clean.
login_storage = FileStorage(os.path.join(SPYCE_HOME, 'www', 'login-tokens'))

# tags to render login form; must be visible in the globaltags search space.
# These come from tags/render.spi; to make your own, just create a tag
# that takes the same parameters and add the library to the globaltags list above.
login_render = 'render:login'
loginrequired_render = 'render:login_required'

######
# webserver options -- does not affect mod_python or *CGI configurations
######
```

```
# indexFiles specifies a list of files to look for
# in a directory if directory itself is requested.
# The first matching file will be selected.
# If empty, a directory listing will be served instead.
#
# indexFiles = ['index.spy', 'index.html', 'index.txt']
indexFiles = ['index.spy', 'index.html']

# The Spyce webserver uses a threaded concurrency model.  (Historically,
# it also offered "no concurrency" and forking.  If for some strange
# reason you really want no concurrency, set minthreads=maxthreads=1.
# Forking was removed entirely because it performed over 10x slower
# than threading on Linux and Windows.)
#
# Do note that because of the Python GIL (global interpeter lock),
# only one CPU of a multi-CPU machine can execute Python code at a time.
# If this is your situation, mod_python may be a better option for you.
minthreads = 5
maxthreads = 10
# number of pending requests to accept
maxqueuesize = 50

# Restart the webserver if a python module changes.
# Spyce does this by running the "real" server in a subprocess; when that
# server detects changed modules, it exits and Spyce starts another one.
#
# Spyce will check for changes every second, or when a request
# is made, whichever comes first.
#
# It's highly recommended to turn this off for "production" servers,
# since checking each module for each request is a performance hit.
check_modules_and_restart = True

# The ipaddr option defines which IP addresses the server will listen on.
# empty string for all.
ipaddr = ''

# The port option defines which TCP port the server will listen on.
port = 8000
# Port that provides an interactive Python console interface to
# the webserver's guts.  Currently no password protection is offered;
# don't expose this to the outside world!
adminport = None

# The mime option is a list of filenames. The files should
# be definitions of mime-types for common file extensions in the
# standard Apache format.
#
# mime: ['/etc/mime.types']
mime = [os.path.join(SPYCE_HOME, 'spyce.mime')]
```

```
# The www_handlers option defines the hander used for files of
# arbitrary extensions.  (The None key specifies the default.)
# The currently supported handlers are:
#   spyce     - process the file at the requested path as a spyce script
#   directory - display directory listing
#   dump      - transfer the file at the requested path verbatim,
#     providing an appropriate "Content-type" header, if it is known.
# (It's difficult to use the actual instance methods here since we
# don't have a handle to the WWW server object.  So, we use strings
# and let spyceWWW eval them later.)
www_handlers = {
  'spy': 'spyce',
  '/':   'directory',
  None:  'dump'
}


######
# (F)CGI options
######

# Forbid direct requests to the cgi script and discard command line arguments.
#
# Reason: http://www.cert.org/advisories/CA-1996-11.html
#
# You may need to disable this for
#  1) non-Apache servers that do not set the REDIRECT_STATUS environment
#     variable.
#  2) when using the alternative #! variant of CGI configuration
cgi_allow_only_redirect = False


######
# standard module customization
######

# (request module)

# param filters and file filters are lists of functions that are called
# for all GET and POST parameters or files, respectively.
# Each callable should expect two arguments: a reference to the
# request object/spyce module, and the dictionary being filtered.
# (Thus, each callable in param_filters will be called twice; once for the
# GET dict and once for POST. Each callable in file_filters will only be called once.)
param_filters = []
file_filters = []
```

3.7. Server utilities

Like any application server, the Spyce server provides several facilities that can aid development. **3.7.1.** ***The Spyce scheduler***

Spyce provides a scheduler that allows you to easily define tasks to run at specified times or intervals inside the Spyce server. This allows your tasks to leverage the tools Spyce gives you, as well as any global data your application maintains within Spyce, such as cached data or database connection pools. This also has the advantage (over, say, crontab entries) of keeping your application self-contained, making it easier to deploy changes from a development machine to production.

The Spyce scheduler is currently only useful if you are running in webserver mode. If you run under mod_python, CGI, or FastCGI, you could approximate scheduler behavior by storing tasks and checking to see if any are overdue with every request received; this would be an excellent project for someone wishing to get started in Spyce development.

scheduler index /var/spyce-2.1/scheduler.py

A module for scheduling arbitrary callables to run at given times
or intervals, modeled on the naviserver API. Scheduler runs in
its own thread; callables run in this same thread, so if you have
an unusually long callable to run you may wish to give it its own
thread, for instance,

```
schedule(3600, lambda: threading.Thread(target=longcallable).start())
```

Scheduler does not provide subsecond resolution.

Public functions are threadsafe.

Modules

atexit	threading	traceback
sys	time	

Classes

Task

class **Task**

Instantiated by the schedule methods.

Instance variables:
  nextrun: epoch seconds at which to run next
  interval: seconds before repeating
  callable: function to invoke

last: if True, will be unscheduled after nextrun

(Note that by manually setting last on a Task ins
tance, you
can cause it to run an arbitrary number of times.
)

Methods defined here:
__init__(self, firstrun, interval, callable, once)
__repr__(self)

examples/scheduling.py

```
import spyce, scheduler

def delete_unsubmitted():
    db = spyce.SPYCE_GLOBALS['dbpool'].connection()
    sql = "DELETE FROM alerts WHERE status = 'unsubmitted' AND created < now() - '1 week'::interval"
    db.execute(sql)

# delete alerts that were created over a week ago but but still not submitted
scheduler.schedule_daily(00, 10, delete_unsubmitted)
```

3.7.2. *spyceUtil*

Most of the spyceUtil module is interesting only to internal operations, but several functions are more generally applicable:

- **url2file**(url, relativeto=None)
  Returns the filesystem path of the file represented by url, relative to a given path. For example, url2file('/index.spy') or url2file('img/header.png', request.filename()).
- **exceptionString**( )
  Every python programmer writes this eventually: returns a string containing the description and stacktrace for the most recent exception.

3.8. Modules

The Spyce language, as described above, is simple and small. Most functionality is provided at runtime through Spyce modules and Python modules.

The standard Spyce modules are documented here; some other modules are also distributed in the contrib/ directory.

Non-implicit modules are imported using the Spyce [[.import]] directive. Python modules are imported using the Python import keyword. Remember that modules need to have the same read permissions as regular files that you expect the web server to read.

Modules may be imported with a non-default name using the **as** attribute to the .import directive. This is discouraged for standard Spyce modules; the**session** module, for example, expects to find or otherwise load a module named **cookie** in the Spyce environment.

Once included, a Spyce module may be accessed anywhere in the Spyce code. **3.8.1.** ***DB (implicit)***

Spyce integrates an advanced database module to make reading and modifying your data faster and less repetitive.

Just initialize the **db** reference in your Spyce config file following the examples given there, and you're all set.

The general idea is, **db.tablename** represents the **tablename** table in your database, and provides hooks for reading and modifying data in that table in pure Python, no SQL required. We find this gives 90% of the benefits of an object-relational mapping layer, without making developers learn another complex tool. And since the Spyce db module is part of SQLAlchemy, probably the most advanced database toolkit in the world, the full ORM approach is available to those who want it.

Here's a quick example of reading and inserting data from a table called **todo_lists**:

examples/db.spy

```
<spy:parent title="To-do demo" />

[[!
def list_new(self, api, name):
   if api.db.todo_lists.selectfirst_by(name=name):
      raise HandlerError('New list', 'a list with that description already exists')
   api.db.todo_lists.insert(name=name)
   api.db.flush()
]]

(This is an self-contained example using the same database as the
<a href=/demos/to-do/index.spy>to-do demo</a>.)

<h2>To-do lists</h2>

[[ lists = db.todo_lists.select(order_by=db.todo_lists.c.name) ]]
<spy:ul data="[L.name for L in lists]" />

<h2>New list</h2>
<f:form>
<f:submit value="New list" handler=self.list_new />:
<f:text name=name value="" />
```

```
</f:form>
```

Run this code

Full SqlSoup documentation follows.

Loading objects

Loading objects is as easy as this:

```
   >>> users = db.users.select()
   >>> users.sort()
   >>> users
   [MappedUsers(name='Joe
Student',email='student@example.edu',password='student',classname=None,admin=0),
    MappedUsers(name='Bhargan
Basepair',email='basepair@example.edu',password='basepair',classname=None,admin=1)]
```

Of course, letting the database do the sort is better (".c" is short for ".columns"):

```
   >>> db.users.select(order_by=[db.users.c.name])
   [MappedUsers(name='Bhargan
Basepair',email='basepair@example.edu',password='basepair',classname=None,admin=1),
    MappedUsers(name='Joe
Student',email='student@example.edu',password='student',classname=None,admin=0)]
```

Field access is intuitive:

```
   >>> users[0].email
   u'student@example.edu'
```

Of course, you don't want to load all users very often. The common case is to select by a key or other field:

```
   >>> db.users.selectone_by(name='Bhargan Basepair')
   MappedUsers(name='Bhargan
Basepair',email='basepair@example.edu',password='basepair',classname=None,admin=1)
```

Select variants

All the SqlAlchemy Query select variants are available. Here's a quick summary of these methods: - get(PK): load a single object identified by its primary key (either a scalar, or a tuple) - select(Clause, **kwargs): perform a select restricted by the Clause argument; returns a list of objects. The most common clause argument takes the form "db.tablename.c.columname == value." The most common optional argument is order_by. - select_by(**params): the *_by selects allow using bare column names. (columname=value)This feels more natural to most Python programmers; the downside is you can't specify order_by or other select options. - selectfirst, selectfirst_by: returns only the first object found; equivalent to select(...)[0] or select_by(...)[0], except None is returned if no rows are selected. - selectone, selectone_by: like selectfirst or selectfirst_by, but raises if less or more than one object is selected. - count, count_by: returns an integer count of the rows selected. See the SqlAlchemy documentation for details: - general info and examples - details on constructing WHERE clauses

Modifying objects

Modifying objects is intuitive:

```
   >>> user = _
   >>> user.email = 'basepair+nospam@example.edu'
   >>> db.flush()
```

(SqlSoup leverages the sophisticated SqlAlchemy unit-of-work code, so multiple updates to a single object will be turned into a single UPDATE statement when you flush.)

To finish covering the basics, let's insert a new loan, then delete it:

```
   >>> db.loans.insert(book_id=db.books.selectfirst(db.books.c.title=='Regional Variation in
Moss').id, user_name=user.name)
   MappedLoans(book_id=2,user_name='Bhargan Basepair',loan_date=None)
   >>> db.flush()

   >>> loan = db.loans.selectone_by(book_id=2, user_name='Bhargan Basepair')
   >>> db.delete(loan)
   >>> db.flush()
```

You can also delete rows that have not been loaded as objects. Let's do our insert/delete cycle once more, this time using the loans table's delete method. (For SQLAlchemy experts: note that no flush() call is required since this delete acts at the SQL level, not at the Mapper level.) The same where-clause construction rules apply here as to the select methods:

```
   >>> db.loans.insert(book_id=book_id, user_name=user.name)
   MappedLoans(book_id=2,user_name='Bhargan Basepair',loan_date=None)
   >>> db.flush()
   >>> db.loans.delete(db.loans.c.book_id==2)
```

You can similarly update multiple rows at once. This will change the book_id to 1 in all loans whose book_id is 2:

```
   >>> db.loans.update(db.loans.c.book_id==2, book_id=1)
   >>> db.loans.select_by(db.loans.c.book_id==1)
   [MappedLoans(book_id=1,user_name='Joe Student',loan_date=datetime.datetime(2006, 7,
12, 0, 0))]
```

Joins

Occasionally, you will want to pull out a lot of data from related tables all at once. In this situation, it is far more efficient to have the database perform the necessary join. (Here we do not have "a lot of data," but hopefully the concept is still clear.) SQLAlchemy is smart enough to recognize that loans has a foreign key to users, and uses that as the join condition automatically.

```
   >>> join1 = db.join(db.users, db.loans, isouter=True)
   >>> join1.select_by(name='Joe Student')
   [MappedJoin(name='Joe
Student',email='student@example.edu',password='student',classname=None,admin=0,
    book_id=1,user_name='Joe Student',loan_date=datetime.datetime(2006, 7, 12, 0, 0))]
```

You can compose arbitrarily complex joins by combining Join objects with tables or other joins.

```
   >>> join2 = db.join(join1, db.books)
   >>> join2.select()
   [MappedJoin(name='Joe
Student',email='student@example.edu',password='student',classname=None,admin=0,
    book_id=1,user_name='Joe Student',loan_date=datetime.datetime(2006, 7, 12, 0, 0),
    id=1,title='Mustards I Have Known',published_year='1989',authors='Jones')]
```

If you join tables that have an identical column name, wrap your join with "with_labels", and all the columns will be prefixed with their table name:

```
   >>> db.with_labels(join1).select()
   [MappedUsersLoansJoin(users_name='Joe Student',users_email='student@example.edu',
                     users_password='student',users_classname=None,users_admin=0,
                     loans_book_id=1,loans_user_name='Joe Student',
                     loans_loan_date=datetime.datetime(2006, 7, 12, 0, 0))]
```

Advanced usage

You can access the SqlSoup's engine attribute to compose SQL directly. The engine's **execute** method corresponds to the one of a DBAPI cursor, and returns a ResultProxy that has **fetch** methods you would also see on a cursor.

```
   >>> rp = db.engine.execute('select name, email from users order by name')
   >>> for name, email in rp.fetchall(): print name, email
   Bhargan Basepair basepair+nospam@example.edu
   Joe Student student@example.edu
```

You can also pass this engine object to other SQLAlchemy constructs; see the SQLAlchemy documentation for details.

You can access SQLAlchemy Table and Mapper objects as db.tablename._table and db.tablename._mapper, respectively. **3.8.2.** ***Request (implicit)***

The request module is loaded implicitly into every Spyce environment.

The spyce configuration file gives two lists that affect the request module: param_filters and file_filters. param_filters is a list of functions to run against the GET and POST variables in the request; file filters is the same, only for files uploaded. Each function will be passed the request module and a dictionary when it is called; each will be called once for GET and once for POST with each new request.

These hooks exist because the request dictionaries should not be modified in an ad-hoc manner; these allow you to set an application-wide policy in a well-defined manner. You might, for instance, disallow all file uploads over 1 MB.

Here's an example that calls either Html.clean or Html.escape (not shown) to ensure that no potentially harmful html can be injected in user-editable areas of a site:

examples/filter.py

```
def htmlFilter(request, d):
  # note that spoofing __htmlfields doesn't help attacker get unsafe html in;
  # we always call either clean() or escape().
  try:
    # don't use request['__htmlfields'], or you will recurse infinitely
    toClean = request._post['__htmlfields'][0].split(',')
  except KeyError:
    toClean = []
  for key in d:
    if key in toClean:
      d[key] = [Html.clean(s) for s in d[key]]
    else:
      d[key] = [Html.escape(s) for s in d[key]]
```

The request module provides the following methods:

- **login_id**:
  Returns the id generated by your validator function if the user has logged in via spy:login or spy:login_required, or None if the user is unvalidated. (See the core tag library for details on the login tags.)
- **uri**([component]):
  Returns the request URI, or some component thereof. If the optional **component** parameter is specified, it should be one of the following strings:'scheme', 'location', 'path', 'parameters', 'query' or 'fragment'.
- **method**():
  Returns request method type (GET, POST, ...)
- **query**():
  Returns the request query string
- **get**([name], [default], [ignoreCase]):
  Returns request GET information. If **name** is specified then a single list of values is returned if the parameter exists, or **default**, which defaults to an empty list, if the parameter does not exist. Parameters without values are skipped, though empty string values are allowed. If name is omitted, then a dictionary of lists is returned.

If **ignoreCase** is true, then the above behaviour is performed in a case insensitive manner (all parameters are treated as lowercase).

- **get1**([name], [default], [ignoreCase]):
  Returns request GET information, similarly to (though slightly differently from) the function above. If **name** is specified then a single string is returned if the parameter exists, or **default**, which default to None, if the parameter does not exist. If there is more than one value for a parameter, then only one is returned. Parameters without values are skipped, though empty string values are allowed. If name is omitted, then a dictionary of strings is returned. If the optional **ignoreCase** flag is true, then the above behaviour is performed in a case insensitive manner (all parameters are treated as lowercase).
- **post**([name], [default], [ignoreCase]):
  Returns request POST information. If **name** is specified then a single list of values is returned if the parameter exists, or **default**, which defaults to an empty list, if the parameter does not exist. Parameters without values are skipped, though empty string values are allowed. If name is omitted, then a dictionary of lists is returned.
  If **ignoreCase** is true, then the above behaviour is performed in a case insensitive manner (all parameters are treated as lowercase). This function understands form information encoded either as 'application/x-www-form-urlencoded' or 'multipart/form-data'. Uploaded file parameters are not included in this dictionary; they can be accessed via the file method.
- **post1**([name], [default], [ignoreCase]):
  Returns request POST information, similarly to (though slightly differently from) the function above. If **name** is specified then a single string is returned if the parameter exists, or **default**, which defaults to None, if the parameter does not exist. If there is more than one value for a parameter, then only one is returned. Parameters without values are skipped, though empty string values are allowed. If name is omitted, then a dictionary of strings is returned. If the optional **ignoreCase** flag is true, then the above behaviour is performed in a case insensitive manner (all parameters are treated as lowercase). This function understands form information encoded either as 'application/x-www-form-urlencoded' or 'multipart/form-data'. Uploaded file parameters are not included in this dictionary; they can be accessed via the file method.
- **file**([name], [ignoreCase]):
  Returns files POSTed in the request. If **name** is specified then a single cgi.FieldStorage class is returned if such a file parameter exists, otherwise None. If name is omitted, then a dictionary of file entries is returned. If the optional **ignoreCase** flag is true, then the above behaviour is performed in a case insensitive manner (all parameters are treated as lowercase). The interesting fields of the FieldStorage class are:
  - **name:** the field name, if specified; otherwise None
  - **filename:** the filename, if specified; otherwise None; this is the client-side filename, not the filename in which the content is stored - a temporary file you don't deal with
  - **value:** the value as a string; for file uploads, this transparently reads the file every time you request the value
  - **file:** the file(-like) object from which you can read the data; None if the data is stored a simple string
  - **type:** the content-type, or None if not specified
  - **type_options:** dictionary of options specified on the content-type line

    - **disposition:** content-disposition, or None if not specified
    - **disposition_options:** dictionary of corresponding options
    - **headers:** a dictionary(-like) object (sometimes rfc822.Message or a subclass thereof) containing *all* headers
- **__getitem__**(key):
  The request module can be used as a dictionary: i.e. request['foo']. This method first calls the get1() method, then the post1() method and lastly the file() method trying to find the first non-None value to return. If no value is found, then this method returns None. Note: Throwing an exception seemed too strong a semantics, and so this is a break from Python. One can also iterate over the request object, as if over a dictionary of field names in the get1 and post1 dictionaries. In the case of overlap, the get1() dictionary takes precedence.
- **getpost**([name], [default], [ignoreCase]):
  Using given parameters, return get() result if not None, otherwise return post() result if not None, otherwise **default**.
- **getpost1**([name], [default], [ignoreCase]):
  Using given parameters, return get1() result if not None, otherwise return post1() result if not None, otherwise **default**.
- **postget**([name], [default], [ignoreCase]):
  Using given parameters, return post() result if not None, otherwise return get() result if not None, otherwise **default**.
- **postget1**([name], [default], [ignoreCase]):
  Using given parameters, return post1() result if not None, otherwise return get1() result if not None, otherwise **default**.
- **env**([name], [default]):
  Returns a dictionary with CGI-like environment information of this request. If **name** is specified then a single entry is returned if the parameter exists, otherwise **default**, which defaults to None, if omitted.
- **getHeader**([type]):
  Return a specific header sent by the browser. If optional **type** is omitted, a dictionary of all headers is returned.
- **filename**([path]):
  Return the Spyce filename of the request currently being processed. If an optional **path** parameter is provided, then that path is made relative to the Spyce filename of the request currently being processed.
- **stack**([i]):
  Returns a stack of files processed by the Spyce runtime. If **i** is provided, then a given frame is returned, with negative numbers wrapping from the back as per Python convention. The first (index zero) item on the stack is the filename corresponding to the URL originally requested. The last (index -1) item on the stack is the current filename being processed. Items are added to the stack by includes, Spyce lambdas, and internal redirects.
- **default**(value, value2):
  (convenience method) Return **value** if it is not None, otherwise return **value2**.

The example below presents the results of all the method calls list above. Run it to understand the information available.

examples/request.spy
<html><body>

```
 Using the Spyce request object, we can obtain
 information sent along with the request. The
 table below shows some request methods and their
 return values. Use the form below to post form
 data via GET or POST. <br>
 <hr>
 [[-- input forms --]]
 <form action="[[=request.uri('path')]]" method=get>
   get: <input type=text name=name>
   <input type=submit value=ok>
 </form>
 <form action="[[=request.uri('path')]]" method=post>
   post: <input type=text name=name>
   <input type=submit value=ok>
 </form>
 <hr>
 [[-- tabulate response information --]]
 <table border=1>
   <tr>
     <td><b>Method</b></td>
     <td><b>Return value</b></td>
   </tr>
   [[ for method in ['uri()', 'uri("path")',
     'uri("query")', 'method()','query()',
     'get()','get1()', 'post()','post1()',
     'getHeader()','env()', 'filename()']: {
   ]]
     <tr>
       <td valign=top>request.[[=method]]</td>
       <td>[[=eval('request.%s' % method)]]</td>
     </tr>
   [[ } ]]
 </table>
</body></html>
```

Run this code

Lastly, the following example shows how to deal with uploaded files.

examples/fileupload.spy

```
[[\
if request.post('ct'):
  response.setContentType(request.post1('ct'))
  if request.file('upfile')!=None:
    response.write(request.file('upfile').value)
  else:
    print 'file not properly uploaded'
  raise spyceDone
]]
```

```
<html><body>
 Upload a file and it will be sent back to you.<br>
 [[-- input forms --]]
 <hr>
 <table>
  <form action="[[=request.uri('path')]]" method=post
    enctype="multipart/form-data">
   <tr>
    <td>file:</td>
    <td><input type=file name=upfile></td>
   </tr><tr>
    <td>content-type:</td>
    <td><input type=text name=ct value="text/html"></td>
   </tr><tr>
    <td><input type=submit value=ok></td>
   </tr>
  </form>
 </table>
</body></html>
```

Run this code

3.8.3. *Response (implicit)*

Like the request module, the response module is also loaded implicitly into every Spyce environment. It provides the following methods:

- **write**(string):
  Sends a **string** to the client. All writes are buffered by default and sent at the end of Spyce processing to allow appending headers, setting cookies and exception handling. Note that using the print statement is often easier, and stdout is implicitly redirected to the browser.
- **writeln**(string):
  Sends a **string** to the client, and appends a newline.
- **writeStatic**(string):
  All static HTML **strings** are emitted to the client via this method, which (by default) simply calls write(). This method is *not* commonly invoked by the user.
- **writeExpr**(object):
  All expression results are emitted to the client via this method, which (by default) calls write() with the str() of the result **object**. This method is *not*commonly invoked by the user.
- **clear**( ): Clears the output buffer.
- **flush**( ): Sends buffered output to the client immediately. This is a blocking call, and can incur a performance hit.
- **setContentType**(contentType):
  Sets the MIME **content type** of the response.
- **setReturnCode**(code):
  Set the HTTP return code for this response. This **return code** may be overriden if an error occurs or by functions in other modules (such as redirects).

- **addHeader**(type, data, [replace]):
  Adds the header line "type: data" to the outgoing response. The optional **replace** flag determines whether any previous headers of the same type are first removed.
- **unbuffer**():
  Turns off buffering on the output stream. In other words, each write is followed by a flush(). An unbuffered output stream should be used only when sending large amounts of data (ie. file transfers) that would take up server memory unnecessarily, and involve consistently large writes. Note that using an unbuffered response stream will not allow the output to be cleared if an exception occurs. It will also immediately send any headers.
- **isCancelled**():
  Returns true if it has been detected that the client is no longer connected. This flag will turn on, and remain on, after the first client output failure. However, the detection is best-effort, and may never turn on in certain configurations (such as CGI) due to buffering.
- **timestamp**([t]):
  Timestamps the response with an HTTP Date: header, using the optional **t** parameter, which may be either be the number of seconds since the epoch (see Python time module), or a properly formatted HTTP date string. If t is omitted, the current time is used.
- **expires**([t]):
  Sets the expiration time of the response with an HTTP Expires: header, using the optional **t** parameter, which may be either the number of seconds since the epoch (see Python time module), or a properly formatted HTTP date string. If t is omitted, the current time is used.
- **expiresRel**([secs]):
  Sets the expiration time of the response *relative to the current time* with an HTTP Expires: header. The optional **secs** (which may also be negative) indicates the number of seconds to add to the current time to compute the expiration time. If secs is omitted, it defaults to zero.
- **lastModified**([t]):
  Sets the last modification time of the response with an HTTP Last-Modified: header, using the optional **t** parameter, which can be either the number of seconds since the epoch (see Python time module), or a properly formatted HTTP date string, or None indicating the current time. If t is omitted, this function will default to the last modification time of the Spyce file for this request, and raise an exception if this time can not be determined. Note that, as per the HTTP specification, you should not set a last modification time that is beyond the response timestamp.
- **uncacheable**():
  Sets the HTTP/1.1 Cache-Control: and HTTP/1.0 Pragma: headers to inform clients and proxies that this content should not be cached.

The methods are self-explanatory. One of the more interesting things that one could do is to emit non-HTML content types. The example below emits the Spyce logo as a GIF.

examples/gif.spy

```
[[.import name=include ]]
[[\
  # Spyce can also generate other content types
  # The following code displays the Spyce logo
  response.setContentType('image/gif')
  import os.path, spyce
  path = os.path.join(spyce.getServer().config.SPYCE_HOME, 'www', 'spyce.gif')
  response.write(include.dump(path, 1))
  raise spyceDone
]]
```

Run this code

3.8.4. *Redirect*

The redirect module allows requests to be redirected to different pages, by providing the following methods:

- **internal**(uri):
  Performs an internal redirect. All processing on the current page ends, the output buffer is cleared and processing continues at the named **uri**. The browser URI remains unchanged, and does not realise that a redirect has even occurred during processing.
- **external**(uri, [permanent]):
  Performs an external redirect using the HTTP Location header to a new **uri**. **Processing of the current file continues** unless you raise spyceDone, but the content is ignored (ie. the buffer is cleared at the end). The status of the document is set to 301 MOVED PERMANENTLY or 302 MOVED TEMPORARILY, depending on the **permanent** boolean parameter, which defaults to false or temporary. The redirect document is sent to the browser, which requests the new relative uri.
- **externalRefresh**(uri, [seconds]):
  Performs an external redirect using the HTTP Refresh header a new **uri**. Processing of the current file continues, and will be displayed on the browser as a regular document. Unless interrupted by the user, the browser will request the new URL after the specified number of **seconds**, which defaults to zero if omitted. Many websites use this functionality to show some page, while a file is being downloaded. To do this, one would show the page using Spyce, and redirect with an externalRefresh to the download URI. Remember to set the Content-Type on the target download file page to be something that the browser can not display, only download.

The example below, shows the possible redirects in use:

examples/redirect.spy

```
[[.import name=redirect]]
<html><body>
  [[ type = request['type']
     url = request['url']
     if url and not type: {
       ]]
       <font color=red><b>
         please select a redirect type
       </b></font><br>
       [[
     }
     if type and url: {
```

```
    if type=='internal': redirect.internal(url)
    if type=='external': redirect.external(url)
    if type=='externalRefresh': redirect.externalRefresh(url, 3)
    ]] Received POST info: [[=request.post1()]] [[
  }
]]
<form action="[[=request.uri('path')]]" method=post>
  Redirection url:
  <input type=text name=url value=hello.spy><br>
  Redirection type:
  <table border=0>
    <tr><td>
      <input type=radio name=type value=internal>
      internal
    </td></tr>
    <tr><td>
      <input type=radio name=type value=external>
      external
    </td></tr>
    <tr><td>
      <input type=radio name=type value=externalRefresh>
      externalRefresh (3 seconds)
    </td></tr>
  </table>
  <input type=submit value=redirect>
</form>
</body></html>
```

Run this code

3.8.5. *Cookie*

This module provides cookie functionality. Its methods are:

- **get**([key]):
  Return a specific cookie string sent by the browser. If the optional cookie **key** is omitted, a dictionary of all cookies is returned. The cookie module may also be accessed as an associative array to achieve the same result as calling:
  namely, cookie['foo'] and cookie.get('foo') are equivalent.
- **set**(key, value, [expire], [domain], [path], [secure]):
  Sends a cookie to the browser. The cookie will be sent back on *subsequent* requests and can be retreived using the get function. The **key** and **value** parameters are required; the rest are optional. The **expire** parameter determines how long this cookie information will remain valid. It is specified in seconds from the current time. If expire is omitted, no expiration value will be provided along with the cookie header, meaning that the cookie will expire when the browser is closed.
  The **domain** and **path** parameters specify when the cookie will get sent; it will be restricted to certain document paths at certain domains, based on the cookie standard. If these are omitted, then path and/or domain information will not be sent in the cookie

header. Lastly, the **secure** parameter, which defaults to false if omitted, determines whether the cookie information can be sent over an HTTP connection, or only via HTTPS.

- **delete**(key):
  Send a cookie delete header to the browser to delete the **key** cookie. The same may be achieved by: del cookie[key].

The example below shows to manage browser cookies.

examples/cookie.spy

```
[[.import name=cookie]]
<html><body>
 Managing cookies is simple. Use the following forms
 to create and destroy cookies. Remember to refresh
 once, because the cookie will only be transmitted on
 the <i>following</i> request.<br>
 [[-- input forms --]]
 <hr>
 <form action="[[=request.uri('path')]]" method=post>
  <table><tr>
   <td align=right>Cookie name:</td>
   <td><input type=text name=name></td>
   <td>(required)</td>
  </tr><tr>
   <td align=right>value:</td>
   <td><input type=text name=value></td>
   <td>(required for set)</td>
  </tr><tr>
   <td align=right>expiration:</td>
   <td><input type=text name=exp> seconds.</td>
   <td>(optional)</td>
  </tr><tr>
   <td colspan=3>
    <input type=submit name=operation value=set>
    <input type=submit name=operation value=delete>
    <input type=submit name=operation value=refresh>
   </td>
  </tr></table>
 </form>
 <hr>
 [[-- show cookies --]]
 Cookies: [[=len(cookie.get().keys())]]<br>
 <table>
  <tr>
   <td><b>name</b></td>
   <td><b>value</b></td>
  </tr>
  [[for c in cookie.get().keys(): {]]
   <tr>
    <td>[[=c]]</td>
```

```
      <td>[[=cookie.get(c)]]</td>
    </tr>
  [[ } ]]
 </table>
 [[-- set cookies --]]
 [[\
   operation = request.post('operation')
   if operation:
     operation = operation[0]
     name = request.post('name')[0]
     value = request.post('value')[0]
     if operation == 'set' and name and value:
       cookie.set(name, value)
     if operation == 'delete' and name:
       cookie.delete(name)
 ]]
</body></html>
```

Run this code

3.8.6. *Session*

Sessions allow information to be efficiently passed from one user request to the next via some browser mechanism: get, post or cookie. Potentially large or sensitive information is stored at the server, and only a short identifier is sent to the client to be returned on callback. Sessions are often used to create sequences of stateful pages that represent an application or work-flow.

This module automates sessioning for a Spyce web site. It emulates a dictionary specific to each user (really each browser) accessing your web site. You simply use session as if it were a dictionary variable, and its contents automatically change depending upon the user calling the page. For example:

examples/session2.spy

```
[[.import name=session ]]
[[\
  session['visited'] = session.get('visited', 0) + 1
]]

<spy:parent title="Session example" />

You visited us [[= session['visited'] ]] times.
```

Run this code

In the example above, the 'visited' key would now be valid for all pages on your site, until the session expires.

Global session options

These options are configured only in the Spyce config file:

- **session_store**: declares the backing storage for session information. It should be either session.DbmStore(path) or session.MemoryStore(). In-memory sessions storage is faster, but volatile, and does not work in multi-process server configurations. Advanced users are welcome to create their own storage manager, by subclassing **session.SessionStore**.

Advanced users can create their own storage manager by subclassing session.SessionStore.

Per-session options

These options are set in the Spyce config file, but may be overridden at module-import time on a per-page basis:

- **session_path**: the default path to attach the session to. This refers to cookie semantics. For example, if the path is /myapp, the session will only be valid under /myapp pages (and below). The default is '/', which means the session is valid site-wide.
- **session_expire**: the number of seconds the session is good for. The default is one day.

  You should clean up expired session state periodically. The easiest way is to schedule session.clean_store every day or so in your config file:

  ```
  import session, scheduler
  scheduler.schedule_daily(0, 0, session.clean_store)
  ```

  (Note: for backwards compatibility, there is also a module called "session1." New code should simply use the module described here.) **3.8.7. *Pool***

  The pool module provides support for server-pooled variables. That is support for variables whose lifetime begins when declared, and ends when explicitly deleted or when the server dies. These variables are often useful for caching information that would be expensive to compute from scratch for each request. Another common use of pool variables is to store file- or memory-based lock objects for concurrency control. A pooled variable can hold any Python value.

  The pool module may be accessed as a regular dictionary, supporting the usual get, set, delete, has_key, keys, values and clearoperations.

  The example below shows how the module is used:

  examples/pool.spy

  ```
  [[.import names="pool"]]
  <html><body>
    The pool module supports long-lived server-pooled objects,<br>
    useful for database connections, and other variables<br>
    that are expensive to compute.<br>
    [[\
      if 'foo' in pool:
  ```

```
    print 'Pooled object foo EXISTS.'
   else:
    print 'Pooled object foo EXISTS.'
    pool['foo'] = 1
    print 'Pooled object foo CREATED.'
  ]]
  <br>
  Value: [[=pool['foo'] ]] <p>
</body></html>
```

Run this code

Pool performance suffers when not used with the Spyce webserver in threaded concurrency mode; Spyce has to un/pickle the shared pool with each request since there is no single long-lived process that can keep the data in-memory. **3.8.8.** ***Transform***

The transform module contains useful text transformation functions, commonly used during web-page generation.

- **html_encode**(string, [also]):
  Returns a HTML-encoded **string**, with special characters replaced by entity references as defined in the HTML 3.2 and 4 specifications. The optional **also** parameter can be used to encode additional characters.
- **url_encode**(string,):
  Returns an URL-encoded **string**, with special characters replaced with %XX equivalents as defined by the URI RFC document.

The transform module also be used to intercept and insert intermediate processing steps when **response.writeStatic()**, **response.writeExpr()** and **response.write()** are called to emit static html, expressions and dynamic content, respectively. It can be useful, for example, to automatically ensure that expressions never produce output that is HTML-unsafe, in other words strings that contain characters such as &, < and >. Many interesting processing functions can be defined. By default, the transform module leaves all output untouched. These processing functions, called filters, can be inserted via the following module functions:

- **static**([fn]):
  Defines the processing performed on all static HTML strings from this point forwards. The **fn** parameter is explained below.
- **expr**([fn]):
  Defines the processing performed on all the results of all expression tags from this point forwards. The **fn** parameter is explained below.
- **dynamic**([fn]):
  Defines the processing performed on all dynamic content generated, i.e. content generated using response.write in the code tags. The **fn**parameter is explained below.

Each of the functions above take a single, optional parameter, which specifies the processing to be performed. The parameter can be one of the following types:

- **None**:
  If the paramter is None, or omitted, then no processing is performed other converting the output to a string.
- **Function**:
  If a parameter of function type is specified, then that function is called to process the output. The function input can be any Python type, and the function output may be any Python type. The result is then converted into a string and emitted. The first parameter to a filter will always be the object to be processed for output. However, the function should be properly defined so as to possibly accept other parameters. The details of how to define filters are explained below.
- **String**:
  If a paramter of string type is specified, then the string should be of the following format: "file:name", where **file** is the location where the function is defined and **name** is the name of the filter. The file component is optional, and is searched for using the standard module-finding rules. If only the function name is specified, then the default location (inside the transform module itself) is used, where the standard Spyce filters reside. The standard Spyce filters are described below.
- **List / Tuple**:
  If a parameter of list or tuple type is specified, its elements should be functions, strings, lists or tuples. The compound filter is recursively defined as f=fn(...f2(f1())...), for the parameter (f1,f2,...,fn).

Having explained how to install filters, we now list the standard Spyce filters and show how they are used:

- **ignore_none**(o):
  Emits any input **o** except for None, which is converted into an empty string.
- **truncate**(o, [maxlen]):
  If **maxlen** is specified, then only the first maxlen characters of input **o** are returned, otherwise the entire original.
- **html_encode**(o, [also]):
  Converts any '&', '<' and '>' characters of input **o** into HTML entities for safe inclusion in among HTML. The optional **also** parameter can specify, additional characters that should be entity encoded.
- **url_encode**(o):
  Converts input **o** into a URL-encoded string.
- **nb_space**(o):
  Replaces all spaces in input **o** with "&nbsp;".
- **silence**(o):
  Outputs nothing.

The optional parameters to some of these filters can be passed to the various write functions as **named parameters**. They can also be specified in an expression tag, as in the following example. (One should simply imagine that the entire expression tag is replaced with a call to response.writeExpr).

```
[[.import name=transform]]
[[ transform.expr(("truncate", "html_encode")) ]]
```

[[='This is an unsafe (< > &) string... '*100, maxlen=500]]

In the example above, the unsafe string is repeated 100 times. It is then passed through a truncate filter that will accept only the first 500 characters. It is then passed through the html_encode filter that will convert the unsafe characters into their safe, equivalent HTML entities. The resulting string is emitted.

The parameters (specified by their names) are simply accepted by the appropriate write method (writeExpr() in the case above) and passed along to the installed filter. Note that in the case of compound filters, the parameters are passed to **ALL** the functions. The html_encode filter is written to ignore the maxlen parameter, and does not fail.

For those who would like to write their own filters, looking at the definition of the truncate filter will help. The other standard filters are in modules/transform.py.

def truncate(o, maxlen=None, **kwargs):

When writing a filter, any function will do, but it is strongly advised to follow the model above. The important points are:

- The input o can be of **any type**, not only a string.
- The function **result** does not have to be string either. It is automatically stringified at the end.
- The function can accept **parameters** that modify its behaviour, such as maxlen, above.
- It is recommended to provide convenient user **defaults** for all parameters.
- The last parameter should be ****kwargs** so that unneeded parameters are quietly passed along.

Lastly, one can retrieve filters. This can be useful when creating new functions that depend on existing filters, but can not be compounded using the tuple syntax above. For example, one might use one filter or another conditionally. For whatever purpose, the following module function is provided to retreive standard Spyce filters, if needed:

- **create**([fn]):
  Returns a filter. The **fn** parameter can be of type None, function, string, list or tuple and is handled as in the installation functions discussed above.

The transform module is flexible, but not complicated to use. The example below is *not* examplary of typical use. Rather it highlights some of the flexibility, so that users can think about creative uses.

examples/transform.spy

```
[[.import name=transform]]
[[\
def tag(o, tags=[], **kwargs):
  import string
  pre = string.join(map(lambda x: '<'+x+'>',tags))
```

```
 tags.reverse()
 post = string.join(map(lambda x: '</'+x+'>',tags))
 return pre+str(o)+post
def bold(o, _tag=tag, **kwargs):
 kwargs['tags'] = ['b']
 return apply(_tag, (o,), kwargs)
def bolditalic(o, _tag=tag, **kwargs):
 kwargs['tags'] = ['b','i']
 return apply(_tag, (o,), kwargs)
myfilter = transform.create(['html_encode', bolditalic])
mystring = 'bold and italic unsafe string: < > &'
def simpletable(o, **kwargs):
 s = '<table border=1>'
 for row in o:
  s=s+'<tr>'
  for cell in row:
   s=s+'<td>'+str(cell)+'</td>'
  s=s+'</tr>'
 s = s+'</table>'
 return s
]]
<html><body>
 install an expression filter:<br>
 [[transform.expr(['html_encode', tag])]]
 1.[[=mystring, tags=['b','i'] ]]
 <br>
 [[transform.expr(myfilter)]]
 2.[[=mystring]]
 [[transform.expr()]]
 <p>
 or use a filter directly:<br>
 1.[[=transform.create(['html_encode',tag])(mystring,tags=['b','i'])]]
 <br>
 2.[[=myfilter(mystring)]]
 <p>
 Formatting data in a table...<br>
 [[=simpletable([ [1,2,3], [4,5,6] ])]]
 <p>
 Though the transform module is flexible, <br>
 most users will probably only install the <br>
 <b>html_encode</b> filter.
</body></html>
```

Run this code

3.8.9. *Compress*

The compress module supports dynamic compression of Spyce output, and can save bandwidth in addition to static compaction. The different forms of compression supported are described below.

- **spaces**([boolean]):
  Controls dynamic space compression. Dynamic space compression will eliminate consecutive whitespaces (spaces, newlines and tabs) in the output stream, each time it is flushed. The optional **boolean** parameter defaults to true.
- **gzip**([level]):
  Applies gzip compression to the Spyce output stream, but only if the browser can support gzip content encoding. Note that this function will fail if the output stream has already been flushed, and should generally only be used with buffered output streams. The optional **level**parameter specifies the compression level, between 1 and 9 inclusive. A value of zero disables compression. If level is omitted, the default gzip compression level is used. This function will automatically check the request's *Accept-Encoding* header, and set the response's *Content-Encoding* header.

The example below shows the compression module in use.

examples/compress.spy

```
[[.import name=compress args="gzip=1, spaces=1"]]
[[\
  response.write('<html><body>')
  response.write('  Space compression will remove these     spaces.<br>')
  response.write('  gzip compression will highly compress this:<br>')
  for i in range(1000):
    response.write('  hello')
  response.write('</body></html>')
]]
```

Run this code

Note that the compression functions need not be called at the beginning of the input, but before the output stream is flushed. Also, to really see what is going on, you should telnet to your web server, and provide something like the following request.

```
GET /spyce/examples/compress.spy HTTP/1.1
Accept-Encoding: gzip
```

3.8.10. *Include*

Many websites carry a theme across their various pages, which is often achieved by including a common header or footer. This is best done with a parent template from the spy:parent tag, but you can also do this with the include module for backwards compatibility with Spyce 1.x.

Another option to consider for repeating a common task is a custom active tag.

The include module can also pretty print Spyce code or include the contents of anything in your filesystem.

- **spyce**(file, [context]):
  Dynamically includes the specified **file** (corresponding to the Spyce document root, not filesystem), and processes it as Spyce code. The return value is that of the included Spyce file. One can optionally provide a **context** value to the included file. If omitted, the value defaults to None. All currently imported modules are passed along into the included file without re-initialization. However, for each explicit [[.import]] tag in the included file, a new module is initialized and also finalized up at the end of processing. The include module provides three fields for use inside included files:
  - include.**context**: This field stores the value passed in at the point of inclusion. Note that if the value is one that is passed by reference (as is the case with object, list, and dictionary types), then the context may be used to pass information back to the including file, in addition to the return value.
  - include.**vars**: If the include context is of type dictionary, then the vars field is initialized, otherwise it is None. The vars field provides attribute-based access to the context dictionary, merely for convenience. In other words, include.vars.x is equivalent to include.context['x'].

  Note that either the locals() or globals() dictionaries may be passed in as include contexts. However, be advised that due to Python optimizations of local variable access, any updates to the locals() dictionary may not be reflected in the local namespace under all circumstances and all versions of Python. In fact, this is the reason why the context has been made explicit, and does not simply grab the locals() dictionary. It may, however, safely be used for read access. With respect to the globals() dictionary, it is not advised to pollute this namespace.

- **spyceStr**(file, [context]):
  Same as **spyce**(), but performs no output and instead returns the processed included Spyce file as a string.
- **dump**(file, [binary]):
  Contents of the **file** (from the filesystem -- use spyceUtil.url2file(url, request.filename) if you need to turn a url into a filesystem path) are returned. If the **binary** parameter is true, the file is opened in binary mode. By default, text mode is used.

  Be careful not to blindly trust the user to specify which file to dump, since anything your Spyce process has access to in the filesystem is fair game.

- **spycecode**(file):
  Contents of the **file** (relative to the Spyce document root) are returned as HTML formatted Spyce code.

The example below (taken from this documentation file), uses a common header template only requiring two context variables to change the title and the highlighted link:

```
[[.import name=include]]
[[include.spyce('inc/head.spi',
   {'pagename': 'Documentation',
    'page': 'manual.html'}) ]]
```

In **head.spi**, we use this information to set the title:

```
[[.import name=include]]
<title>[[=include.context['pagename'] ]]</title>
```

By convention, included files are given the extension **.spi**.

Below we contrast the difference between static and dynamic includes. A dynamic include is included on each request; a static include is inserted at compile time. A static include runs in the same context, while a dynamic include has a separate context.

examples/include.spy

```
[[.import name=include]]
<html><body>
  main file<br>
  <hr>
  [[
    context = {'foo': 'old value'}
    result=include.spyce('include.spi', context)
  ]]
  <hr>
  main file again<br>
  context: [[=context]]<br>
  return value: [[=result]]<br>
</body></html>
```

Run this code

examples/include.spi

```
begin include<br>
context: [[=include.context ]]<br>
from: [[=request.stack()[-2] ]]<br>
foo was [[=include.vars.foo]]<br>
setting foo to 'new value' [[include.vars.foo = 'new value']]<br>
returing 'retval'<br>
end include<br>
[[ return 'retval' ]]
```

examples/includestatic.spy

```
<html><body>
  [[x=1]]
  main file<br>
  x=[[=x]]<br>
  <hr>
  [[.include file="includestatic.spi"]]
  <hr>
  main file again<br>
  x=[[=x]]
</body></html>
```

Run this code

examples/includestatic.spi

```
begin included file<br>
changing value of x<br>
[[x=2]]
end included file<br>
```

3.8.11. *Internal modules*

These modules are used internally by Spce. Documentation is included for those curious about Spyce internals; ordinarily you will never use these modules directly. The error module is implicitly loaded and provides error-handling functionality. An error is any unhandled runtime exception that occurs **during Spyce processing**. This mechanism does not include exceptions that are not related to Spyce processing (i.e. server-related exceptions), that can be caused before or after Spyce processing by invalid syntax, missing files and file access restrictions. To install a server-level error handler use a configuration file. The default page-level error handler can also be modified in the configuration file. This module allows the user to install page-level error handling code, overriding the default page-level handler, by using one of the following functions:

- **setStringHandler**(string):
  Installs a function that will processes the given **string**, as Spyce code, for error handling.
- **setFileHandler**(uri):
  Installs a function that will processes the given **uri** for error handling.
- **setHandler**(fn):
  Installs the **fn** function for error handling. The function is passed one parameter, a reference to the error module. From this, all the error information as well as references to other modules and Spyce objects can be accessed.

The error module provides the following information about an error:

- **isError**():
  Returns whether an error is being handled.
- **getMessage**():
  Return the error message; the string of the object that was raised, or None if there is no current error.
- **getType**():
  Return the error type; the type of the object that was raised, or None if there is no current error.
- **getFile**():
  Return the file where the error was raised, or None if there is no current error.
- **getTraceback**():
  Return the stack trace as an array of tuples, or None if there is no current error. Each tuple entry is of the form: (file, line numbers, function name, code context).
- **getString**():
  Return the string of the entire error (the string representation of the message, type, location and stack trace), or None if there is no current error.

The default error handling function uses the following string handler:

```
[[.module name=transform]]
[[transform.expr('html_encode')]]
<html>
<title>Spyce exception: [[=error.getMessage()]]</title>
<body>
<table cellspacing=10 border=0>
  <tr><td colspan=2><h1>Spyce exception</h1></td></tr>
  <tr><td valign=top align=right><b>File:</b></td><td>[[=error.getFile()]]</tr>
  <tr><td valign=top align=right><b>Message:</b></td>
    <td><pre>[[=error.getMessage()]]</pre></tr>
  <tr><td valign=top align=right><b>Stack:</b></td><td>
    [[\
      L = list(error.getTraceback())
      L.reverse()
    ]]
    [[ for frame in L: { ]]
      [[=frame[0] ]]:[[=frame[1] ]], in [[=frame[2] ]]:<br>
      <table border=0><tr><td width=10></td><td>
        <pre>[[=frame[3] ]]</pre>
      </td></tr></table>
    [[ } ]]
    </td></tr>
</table>
</body></html>
```

The example below shows the error module in use. Error handling can often be used to send emails notifying webmasters of problems, as this example shows.

examples/error.spy

```
[[error.setFileHandler('error.spi') ]]
This is a page with an error...
[[ raise 'an error' ]]
```

Run this code

examples/error.spi

```
<h1>Oops</h1>
An error occurred while processing your request.
We have logged this for our webmasters, and they
will fix it shortly. We apologize for the inconvenience.
In the meantime, please use the parts of our site that
actually do work... <a href="somewhere">somewhere</a>.
[[\
  # could redirect the user immediately
  #response.getModule('redirect').external('somewhere.spy')

  # could send an email
  import time
  msg = '''
time: %s
error: %s
env: %s
other info...
''' % (
    time.asctime(time.localtime(time.time())),
    error.getString(),
    request.env()
  )
  #function_to_send_email('webmaster@foo.com', msg)

  #or perform other generic error handling...
]]
```

This mechanism is not a subsititute for proper exception handling within the code itself, and should not be abused. It does, however, serve as a useful catch-all for bugs that slip through the cracks.

The stdout module is loaded implicitly and redirects Python's sys.stdout (in a thread-safe manner) to the appropriate response object for the duration of Spyce processing. This allows one to use print, without having to write print >> response, ... The stdout module provides a variable stdout.stdout, which refers to the original stream, but is unlikely to be needed. It may also be useful to know that sys.stderr is, under many configurations, connected to the webserver error log.

In addition, the stdout module provides the following functions for capturing or redirecting output:

- **push**([filename]):
  Begin capturing output. Namely, the current output stream is pushed onto the stack and replaced with a memory buffer. An optional **filename**may be associated with this operation (see pop() method below).
- **pop**():
  Close current output buffer, and return the captured output as a string. If a filename was associated with the push(), then the string will also be written to that file.
- **capture**(f, [*args], [**kwargs]):
  Push the current stream, call the given function **f** with any supplied arguments ***args** and keyword arguments ****kwargs**, and then pop it back. Capture returns a tuple (r,s), where r is the result returned by f and s is a string of its output.

The example below show how the module is used:

examples/stdout.spy

```
<html><body>
  [[ print '''Using the stdout module redirects
    stdout to the response object, so you can use
    <b>print</b>!''']]<br>
  redirecting stdout can be used to...
  [[stdout.push()]]
  [[print 'capture']] out[[='put']]
  [[cached = stdout.pop()]]
  ... for later: <br>
  [[=cached]]
</body></html>
```

Run this code

This module is used internally by Spyce, not usually by users. Documentation is included for those curious about Spyce internals. The spylambda module is loaded implicitly and allows the definition of functions based on Spyce scripts; see Spyce Lambdas. The spylambda module provides the following methods:

- **define**(args, code, [memoize]):
  Returns a function that accepts the given **args** and executes the Spyce script defined by the **code** parameter. Note that the code is compiled immediately and that spyce.spyceSyntaxError or spyce.spycePythonError exceptions can be thrown for invalid code arguments. The optional **memoize** parameter sets whether the spyce can or can not be memoized, with the default being false. Memoizing a function means capturing the result and output and caching them, keyed on the function parameters. Later, if a function is called again with the same parameters, the cached information is returned, if it exists, and the function may not actually be called. Thus, you should only memoize functions that are truly functional, i.e. they do not have side-effects: they only return a value and output data to the response object, and their behaviour depends

exclusively on their parameters. If you memoize code that does have side-effects, those side-effects may not occur on every invocation.
- **__call__**(args, code, _spyceCache):
  This is an alias to the define function. Because of the special method name, the spylambda module object can be called as if it were a function.

The taglib module is loaded implicitly and supports Active Tags functionality. The taglib module provides the following methods:

- **load**(libname, [libfrom], [libas]):
  Loads a tag library class named **libname** from a file called **libfrom** in the search path, and installed it under the tag prefix **libas**. The default for libfrom is *libname*.py. The default for libas is *libname*. Once installed, a library name is its unique tag prefix.
- **unload**(libname):
  Unload a tag library that is installed under the **libname** prefix. This is usually performed only at the end of a request.
- **tagPush**(libname, tagname, pair):
  Push a new tag object for a **libname**:**tagname** tag onto the tag stack.
  The **pair** parameter is a flag indicating whether this is a singleton or a paired tag.
- **tagPop**():
  Pop the current tag from the tag stack.
- **getTag**():
  Return the current tag object.
- **outPush**():
  Begin capturing the current output stream. This is usually called by the tagBegin method.
- **outPopCond**():
  End capturing the current output stream, and return the captured contents. It will only "pop" once, even if called multiple times for the same tag. This method is usually called by either the tagEnd(), tagCatch, or tagPop() methods.
- **tagBegin**(attrs):
  This method sets the tag output and variable environment, and then calls the tag's **begin()** method with the given **attrs** tag attribute dictionary. This method returns a flag, and the tag body must be processed if and only if this flag is true.
- **tagBody**():
  This method sets the tag output and variable environment, and then calls the tag's **body()** method with the captured output of the body processing. If this method returns true, then the processing of the body must be repeated.
- **tagEnd**():
  This method sets the tag output and variable environment, and then calls the tag's **end()** method. This method must be called if the tagBegin() method completes successfully in order to preserve tag semantics.
- **tagCatch**():
  This method should be called if any of the tagBegin, tagBody or tagEnd methods raise an exception. It calls the tag's **catch()** method with the current exception.

3.8.12. *Writing Modules*

Writing your own Spyce modules is simple.

A Spyce modules is simply a Python class that exposes specific methods to the Spyce server. The most important are **start**, **finish**, and **init**. With these, a Spyce module may access the internal request and response structures or alter the behaviour of the runtime engine in some way.

Let us begin with a basic example called myModule. It is a module that implements one function named foo().

examples/myModule.py

```
from spyceModule import spyceModule

class myModule(spyceModule):
  def foo(self):
    print 'foo called'
```

Saving this code in myModule.py in the same directory as the Spyce script, or somewhere on the module path, we could use it as expected:

```
[[.import name=myModule]]
[[ myModule.foo() ]]
```

A Spyce module can be any Python class that derives from **spyceModule.spyceModule**. When it is loaded, Spyce assigns it a __file__ attribute indicating its source location. Do not override the **__init__(...)** method because it is inherited from spyceModule and has an fixed signature that is expected by the Spyce engine's module loader. The inherited method accepts a Spyce API object, a Bastion of **spyce.spyceWrapper**, an internal engine object, and stores it in **self._api**. This is the building block for all the functionality that any module provides. The available API methods of the wrapper are (listed in spyceModule.spyceModuleAPI):

- **getFilename**: Return filename of current Spyce
- **getCode**: Return processed Spyce (i.e. Python) code
- **getCodeRefs**: Return python-to-Spyce code line references
- **getModRefs**: Return list of import references in Spyce code
- **getServerObject**: Return unique (per engine instance) server object
- **getServerGlobals**: Return server configuration globals
- **getServerID**: Return unique server identifier
- **getModules**: Return references to currently loaded modules

- **getModule**: Get module reference. The module is dynamically loaded and initialised if it does not exist (ie. if it was not explicitly imported, but requested by another module during processing)
- **setModule**: Add existing module (by reference) to Spyce namespace (used for includes)
- **getGlobals**: Return the Spyce global namespace dictionary
- **registerModuleCallback**: Register a callback for modules change
- **unregisterModuleCallback**: Unregister a callback for modules change
- **getRequest**: Return internal request object
- **getResponse**: Return internal response object
- **setResponse**: Set internal response object
- **registerResponseCallback**: Register a callback for when internal response changes
- **unregisterResponseCallback**: Unregister a callback for when internal response changes
- **spyceString**: Return a spyceCode object of a string
- **spyceFile**: Return a spyceCode object of a file
- **spyceModule**: Return Spyce module class
- **spyceTaglib**: Return Spyce taglib class
- **setStdout**: Set the stdout stream (thread-safe)
- **getStdout**: Get the stdout stream (thread-safe)

For convenience, one can sub-class the **spyceModulePlus** class instead of the regular **spyceModule**. The spyceModulePlus defines a**self.modules** field, which can be used to acquire references to other modules loaded into the Spyce environment. The *response* module, for instance, would be referenced as *self.modules.response*. Modules are loaded on demand, if necessary. The spyceModulePlus also contains a**self.globals** field, which is a reference to the Spyce global namespace dictionary, though this should rarely be needed.

Note: It is not expected that many module writers will need the entire API functionality. In fact, the vast majority of modules will use a small portion of the API, if at all. Many of these functions are included for just one of the standard Spyce modules that needs to perform some esoteric function.

Three Spyce module methods, **start()**, **init([args])** and **finish(error)** are special in that they are automatically called by the runtime during Spyce request startup, processing and cleanup, respectively. The modules are started in the order in which module directives appear in the file, before processing begins. The implicitly loaded modules are always loaded first. The init method is called during Spyce processing at the location of the module directive in the file, with the optional args attribute is passed as the arguments of this call. Finally, after Spyce processing is complete, the modules are finalized in reverse order. If there is an unhandled exception, it will be wrapped in a spyce.spyceException object and passed as the first parameter to finish(). During successful completion of Spyce processing (i.e. without exception), the error parameter is None. The default inherited start, init and finish methods from spyceModule are noops.

Note 2: When writing a Spyce module, consider carefully why you are selecting a Spyce module over a regular Python module. If it is just code, that does not interact with the Spyce engine, then a regular Python import instead of an Spyce [[.import]] can just as easily bring in the necessary code, and is preferred. In other words, choose a Spyce module only when there is a need for per-request initialization or for one of the engine APIs.

Module writers are encouraged to look at the existing standard modules as examples and the definitions of the core Spyce objects in spyce.py as well. If you write or use a novel Spyce module that you think is of general use, please email your contribution, or a link to it. Also, please keep in mind that the standard modules are designed with the goal of being minimalist. Much functionality is readily available using the Python language libraries. If you think that they should be expanded, also please send a note.

3.9. Tags

The previous chapter discussed the Spyce module facility, the standard Spyce modules and how users can create their own modules to extend Spyce. Spyce functionality can also be extended via active tags, which are defined in tag libraries. This chapter describes what Spyce active tags are, and how they are used. We then describe each of the standard active tag libraries and, finally, how to define new tags libraries.

It is important, from the outset, to define what an active tag actually does. A few illustrative examples may help. The examples below all use tags that are defined in the core tag library, that has been installed under the **spy** prefix, as is the default.

- <spy:parent src="parent.spi"/>
  Wraps the current page in the parent template found at parent.spi in the same directory.

-
  ```
  <spy:for items="=range(5)">
    <spy:print value="=foo"/>
  </spy:for>
  ```
  As expected, these tags will print the value of **foo**, set to bar above, 5 times.

Common mistake: Don't use [[=]] to send values to active tag attributes: [[=]] sends its result directly to the output stream. And since those tokens are parsed with higher precedence than the tag markup, Spyce won't recognize your tag at all and will print it verbatim to the client. Instead, prefix an expression with =, as in "=range(5)" above, and Spyce will *eval* it before sending it to the tag.

Note that the same output could have been achieved in many different ways, and entirely without active tags. The manner in which you choose to organize your script or application, and when you choose active tags over other alternatives, is a matter of personal preference. Notice also that active tags entirely control their output and what they do with their attributes and the result of processing their bodies (in fact, whether the body of the tag is even processed). Tags can even supply additional syntax constraints on their attributes that will be enforced at compile-time. Most commonly a tag could require that certain attributes exist, and possibly that it be used only as a single or only as a paired (open and close) tag. Unlike early versions of HTML, active tags must be strictly balanced, and this will be enforced by the Spyce compiler.

Below, each individual standard Spyce tag library is documented, followed by a description of how one would write a new active tag library. The following general information will be useful for reading that material.

- Active tags are installed using the [[.taglib]] directive, under some prefix. Tag libraries may also be be loaded globally in the config module; by default the core and form libraries are preloaded. Active tags are of the format <pre:name ... >, where **pre** is the prefix under which the tag library was installed, and **name** is defined by the tag library. In the following tag library documentation, the prefix is omitted from the syntax.
- The following notation is used in the documentation of the tag libraries below:
  - <name .../> : The tag should be used as a singleton.
  - <name ... > ... </name> : The tag should be used as an open-close pair.
  - [x (default)] : The attribute is optional. Attributes not enclosed in brackets are required.
  - foo|bar : indicates that an attribute may be one of two constant strings. The underlined value is the default.
  - *string* : an arbitrary string constant, never evaluated as Python
  - *exprstring* : may be a string constant, and may be of the form '=expr', where expr is a Python expression that will be evaluated in the tag context.
  - *expr*: a Python expression. (Currently only the "data" parameters of some form and core tags use this rather than exprstring.)
  - *exports foo, *bar* Exports to the parent context the variable foo and the variable with the name given by the expression bar. Normally, implementation details of tags will not affect the parent context, so you do not have to worry about your variables being clobbered. Tags may, however, export specific parts of their own context to the parent. See, for example, the let and for tags in the core taglib. **Note:**exporting of variables whose name cannot be determined at compile time is deprecated, and will be removed in Spyce 2.2.

3.9.1. *Core*

The core tag library is aliased as "spy" by default.

This library contains various frequently-used tags: the parent tag, login tags, and some tags for generating lists and tables from Python iterators.

Parent Tag

- <**parent** [src=*url*] [other parameters] />
  Specifies a parent template to apply to the current page, which is passed to the parent as child._body. Any extra parameters are also passed in the child dictionary. If src is not given, 'parent.spi' used if it exists in the current directory; otherwise, the default parent is used as specified in the config module.

examples/hello-templated.spy

```
<spy:parent title="Hello, world!" />

[[-- Spyce can embed chunks of Python code, like this. --]]
[[\
  import time
  t = time.ctime()
]]

[[--
  pull the count from the GET variable.
  the int cast is necessary since request stores everything as a string
--]]
[[ for i in range(int(request.get1('count', 2))):{ ]]
<div>Hello, world!</div>
[[ } ]]

The time is now [[= t ]].
```

Run this code

Login Tags

- <**login** [validator=*function name*] />
  Generates a login form according to the template specified in your config file. If **validator** is not specified, the default validator from your config file is used. (**validator** may be the name of a function in a different Python module; just prefix it with the module name and Spyce will automaticall import it when necessary.)

examples/login-optional.spy

```
<html><body>

<f:form>
  [[ if request.login_id():{ ]]
    You are logged in with user id [[= request.login_id() ]]
    <spy:logout />
  [[ } else: { ]]
    <spy:login />
    You are not logged in.  (You may login with username/password: spyce/spyce.)
  [[ } ]]

  <p>
  (Here is some content that is the same before and after login.)
</f:form>

</body></html>
```

Run this code

- <**logout** />
  Generates a logout button that will clear the cookie generated by **login** and **login_required**.
- <**login_required** [validator=*function name*] />
  If a valid login cookie is not present, generates a login form according to the template specified in your config file, then halts execution of the current page.

  (You may log in to this example as user **spyce**, password **spyce**.)

examples/login-required.spy

```
<spy:parent title="Login example" />

<f:form>
  <spy:login_required />

  You are logged in.
  <spy:logout />
</f:form>
```

Run this code

Convenience Tags

These tags are shortcuts for creatings lists and tables. As with the form tag library, any python iterator may be given as the data parameter. Also as with the form tags, any unrecognized parameters will be passed through to the generated HTML.

- <**ul** data=*expr*] />
  Convenience tag for the common use of ul; equivalent to
  - <ul>
  - [[for item in data:{]]
  -   <li>[[= item]]</li>
  - [[}]]
  - </ul>

- <**ol** data=*expr*] />
  Like ul, but for ordered lists.

- <**dl** data=*expr*] />
  Convenience tag for the common use of dl; equivalent to
  - <dl>
  - [[for term, desc in data:{]]
  -   <dt>[[= term]]</dt>
  -   <dd>[[= desc]]</dd>
  - [[}]]
  - </dl>

- <**table** data=*expr*] />
  Convenience tag for the common use of table; equivalent to
  - <table>
  - [[for row in data:{]]
  -   <tr>
  -   [[for cell in row:{]]
  -     <td>[[= cell]]</td>
  -   [[}]]
  -   </tr>
  - [[}]]
  - </table>

3.9.2. *Form*

The form tag library is aliased as "spy" by default.

This library simplifies the generation and handling of forms by automating away repetitive tasks. Let's take a look at a simple example:

examples/formintro.spy

```
<spy:parent title="Form tag intro" />

<f:form>

<div class="simpleform">
  <f:text name="text1" label="Text input" default="change me" size=10
maxlength=30 />
  <f:text name="text2" label="Text input 2" value="change me" size=10
maxlength=30 />
</div>

<fieldset style="clear: both">
  <legend>One or two?  Or both?</legend>
  <f:checkboxlist class="radio" name="checkboxlist" data="[(1, 'one'), (2, 'two')]" />
</fieldset>

<div style="clear: both"><f:submit /></div>

</f:form>
```

Run this code

This demonstrates several properties of Spyce form tags:

- Most tags take an optional label parameter; this is turned into an HTML label tag associated with the form element itself.
- If you View Source in your browser while running this sample, you can see that Spyce generates an id with the same value as the name parameter. (You can override this by explicitly specifying a different id parameter, if you need.)

- You can pass arbitrary parameters (such as the class parameter for <f:form>) to a Spyce form tag; parameters that do not have special meaning to Spyce will be passed through to the HTML output.
- Try changing the form values and submitting. By default, Spyce automatically remembers the user input for you, unless you give a tag a *value*parameter (or *selected* for collection elements), which has highest precedence. Note the different behavior of text1 and text2 in this example.
- Spyce provides some higher-level tags such as checkboxlist that result in multiple elements at the HTML level. For these tags, a "data" parameter is expected, which is always interpreted as a Python expression. Any iterable may be used as data, including generators and generator expressions for those with recent Python versions. Typically data would come from the database; here we're just using a literal list.

Handlers

Active Handlers allow you to "attach" python functions to form submissions. They are described in the Active Handlers manual page.

Reference

First, some general rules:

The text displayed by a text-centric tag can come from one of three places. In order of decreasing priority, these are

- the **value** parameter
- the value submitted by the user is used
- the **default** parameter

If none of these are found, the input will be empty.

For determining whether option, radio, and checkbox tags are checked or selected, a similar process is followed, with **selected** and **checked**parameters as the highest-priority source. The same parameters are used for select, radiolist, and checkboxlist tags; the only difference is for the collection tags, you can also specify multiple values in a Python list (or other iterable) in either
the **selected/checked** or **default** parameters.

All tags except form and submit can be given names that tell Spyce how to treat their submitted values when passing to an Active Handler function. Adding ":int", ":float", ":bool", or ":list" is allowed. The first three tell Spyce what kind of Python value to convert the submission to; ":list" may be combined with these, or used separately, and tells Spyce to pass a list of all values submitted instead of a single one. (An example is given in the Active Handlers page.) Finally, here is the list of tags:

- <**form** [method=*exprstring*] [action=*exprstring*] ...> </**form**>
  Begin a new form. The **method** parameter defaults to 'POST'.
  The **action** parameter defaults to the current page.

- <**submit** [handler=*exprstring*] [value=*exprstring*] ... />
  Create a submit button. The **value** parameter is emitted as the button text. If **handler** is given, Spyce will call the function(s) it represents at the beginning of the page load after this button is clicked. (Multiple function names may be separated with commas.)

  If the handler is in a different [python] module, Spyce will automatically import it before use.

  A handler may take zero or more arguments. For the first non-self argument (if present), Spyce always passes a moduleFinder corresponding to the current spyceWrapper object; it is customary to call this argument "api." moduleFinder provides __getitem__ access to loaded modules; thus, "api.request" would be the current request module. If a requested module is not found, it is loaded.

  (You can also directly access the wrapper with api._wrapper, providing access to anything module authors have, but you will rarely if ever need to do this.)

  For other handler function parameters, Spyce will pass the values for the corresponding form input, or None if nothing was found in the GET or POST variables.

  See also the Active Handlers language section for a higher-level overview.

  Limitation: currently, Active Handlers require resubmitting to the same spyce page; of course, the handler method may perform an internal or external redirect.

- <**hidden** name=*exprstring* [value=*exprstring*] [default=*exprstring*] .../>
  Create a hidden form field. The **name** parameter is evaluated and emitted.
- <**text** name=*exprstring* [value=*exprstring*] [default=*exprstring*] .../>
  Create a form text field. The **name** parameter is evaluated and emitted.
- <**date**name=*exprstring* [value=*exprstring*] [default=*exprstring*] [size=*exprstring*] [format=*exprstring*] .../>
  Create a form text field with a javascript date picker. Format defaults to MM/DD/YYYY. Maxlength is always len(format); this is also the default size, but size may be overridden for aesthetics.
- <**password**name=*exprstring* [value=*exprstring*] [default=*exprstring*] [size=*exprstring*] [maxlength=*exprstring*] .../>
  Create a form password field. Parameters are the same as for **text** fields, explained above.
- <**textarea** name=*exprstring* [value=*exprstring*] [rows=*exprstring*] [cols=*exprstring*] ...>default</**textarea**>
  Create a form textarea field. The **name** parameter is evaluated and emitted. The **value** optional parameter is evaluated. A **default** may be provided in the body of the tag. The value emitted is, in order of decreasing priority: local tag value, value in submitted request dictionary, local tag default. We search this list for the first non-None value. The **rows** and **cols** optional parameters are evaluated and emitted.

- <**radio** name=*exprstring* value=*exprstring* [checked] [default] .../>
  Create a form radio-box. The **name** and **value** parameters are evaluated and emitted. A **checked** and **default** flags affect whether this box is checked. The box is checked based on the following values, in decreasing order of priority: tag value, value in submitted request dictionary, tag default. We search this list for the first non-None value.
- <**checkbox** name=*exprstring* value=*exprstring* [checked] [default] .../>
  Create a form check-box. Parameters are the same as for **radio** fields, explained above.
- <**select** name=*exprstring* [selected=*exprstring*] [default=*exprstring*] [data=*expr*] ...>...</**select**>
  Create a form select block. The **name** parameter is evaluated and emitted. The optional **data** should be an iterable of (description, value) pairs.
- <**option** [text=*exprstring*] [value=*exprstring*] [selected] [default] .../>
  <**option** [value=*exprstring*] [selected] [default] ...>text</**option**>
  Create a form selection option. This tag must be nested within a **select** tag. The **text** optional parameter is evaluated and emitted in the body of the tag. It can also be provided in the body of the tag, as you might be used to seeing in HTML.
- <**radiolist** name=*exprstring* data=*expr* [checked=*exprstring*] [default=*exprstring*] ...>...</**select**>
  Create multiple radio buttons from **data**, which should be an iterable of (description, value) pairs.
- <**checkboxlist** name=*exprstring* data=*expr* [checked=*exprstring*] [default=*exprstring*] ...>...</**select**>
  Create multiple checkboxes from **data**, which should be an iterable of (description, value) pairs.

Here is an example of all of these tags in use:

examples/formtag.spy

```
<spy:parent title="Form tag example" />

<f:form>

<h2>Primitive controls</h2>

<div class="simpleform">
  <f:text name="mytext" label="Text" default="some text" size=10 maxlength=30 />

  <f:password name="mypass" label="Password" default="secret" />

  <f:textarea name="mytextarea" label="Textarea" default="rimon" rows=2
cols=50></f:textarea>

  <label for="mycheck">Checkbox</label><f:checkbox name=mycheck value=check1
/>
```

```
  <label for="myradio1">Radio option 1</label><f:radio name=myradio
value=option1 />
  <label for="myradio2">Radio option 2</label><f:radio name=myradio
value=option2 />
</div>

<div style="clear: both">
  <h2 style="padding-top: 1em;">Compound controls</h2>
  [[-- a simple data source for the compound controls -- in practice
                this would probably come from the database --]]
  [[ L = [('option %d' %i, str(i)) for i in range(5)] ]]

  <fieldset>
    <legend>Radiolist</legend>
    <f:radiolist class=radio name=radiolist data="L" default="3" />
  </fieldset>

  <fieldset>
    <legend>Checkboxlist</legend>
    <f:checkboxlist class=radio name=checkboxlist data="L" default="=['0', '1']" />
  </fieldset>

  <fieldset>
    <legend>Select</legend>
    <f:select name=myselect multiple size=5 data="L" default="2" />
  </fieldset>

  <fieldset>
    <legend>Date</legend>
    <f:date name=mydate />
  </fieldset>

  <h2 style="clear:both; padding-top: 1em;">Test it!</h2>
  <input type="submit" name="foo" value="Submit!">

</f:form>
```

Run this code

3.9.3. *Active Handlers*

Active Handlers allow you to "attach" python functions to Spyce form submissions. Instead of old-fashioned inspection of the request environment, Spyce will pull the parameters required by your function's signature out for you. Let's have a look at an example. Here, we define a *calculate*function and assign it to be the handler for our submit input:

examples/handlerintro.spy

```
[[!
def calculate(self, api, x, y):
   self.result = x * y
]]

<spy:parent title="Active Handler example" />
<f:form>
   <f:text name="x:float" default="2" label="first value" />
   <f:text name="y:float" default="3" label="second value" />

   <f:submit handler="self.calculate" value="Multiply" />
</f:form>

<p>
Result: [[= hasattr(self, 'result') and self.result or '(no result yet)' ]]
</p>
```

Run this code

- Handlers may be inline, as the *calculate* handler is here, or in a separate Python module. When using a handler from a Python module, Spyce will automatically import the module when needed. (Handler parameters are strings, not Python references.) The todo demo demonstrates using handlers this way.
- You can give your form inputs a data type; Spyce will perform the conversion automatically before passing parameters to your handler function.

Active Handlers also make it easy to incorporate user-friendly error messages into your forms simply by raising a HandlerError:

examples/handlervalidate.spy

```
[[!
def calculate(self, api, x):
   from spyceException import HandlerError
   try:
      x = float(x)
   except ValueError:
      raise HandlerError('Value', 'Please input a number')
   if x < 0:
      raise HandlerError('Value', 'Cannot take the square root of negative numbers!')
   self.result = x ** 0.5
]]

<spy:parent title="Active Handler Validation" />
<f:form>
   <f:text name="x" default="-1" label="Value:" />
   <f:submit handler="self.calculate" value="Square root" />
</f:form>
```

```
<p>
Result: [[= hasattr(self, 'result') and self.result or '(no result yet)' ]]
</p>
```

Run this code

You can show multiple errors at once with a CompoundHandlerError:

examples/handlervalidate2.spy

```
[[!
def errors(self, api):
    from spyceException import HandlerError, CompoundHandlerError
    cve = CompoundHandlerError()
    cve.add(HandlerError('One', 'First error'))
    cve.add(HandlerError('Two', 'Second error'))
    if cve:
        raise cve
]]

<spy:parent title="Active Handler Validation 2" />
<f:form>
    <f:submit handler="self.errors" value="Show me some errors" />
</f:form>
```

Run this code

All Spyce modules are available via the **api** handler parameter (which should always be the first parameter (after **self** in a class). Here is an example that uses the db module:

examples/db.spy

```
<spy:parent title="To-do demo" />

[[!
def list_new(self, api, name):
    if api.db.todo_lists.selectfirst_by(name=name):
        raise HandlerError('New list', 'a list with that description already exists')
    api.db.todo_lists.insert(name=name)
    api.db.flush()
]]

(This is an self-contained example using the same database as the
<a href=/demos/to-do/index.spy>to-do demo</a>.)

<h2>To-do lists</h2>

[[ lists = db.todo_lists.select(order_by=db.todo_lists.c.name) ]]
```

```
<spy:ul data="[L.name for L in lists]" />

<h2>New list</h2>
<f:form>
<f:submit value="New list" handler=self.list_new />:
<f:text name=name value="" />
</f:form>
```

Run this code

Handlers in Active Tags allow you to create reusable components, as in the the chat demo.

(Since Spyce captures stdout, you can use **print** to debug handlers.) **3.9.4.** ***Writing Tag Libraries***

Creating your own active tags is quite easy and this section explains how. You may want to create your own active tags for a number of reasons. More advanced uses of tags include database querying, separation of business logic, or component rendering. On the other hand, you might consider creating simpler task-specific tag libraries. For example, if you do not wish to rely on style-sheets you could easily define your own custom tags to perform the formatting in a consistent manner at the server. Another convenient use for tags is to automatically fill forms with session data. These are only a few of the uses for tags. As you will see, writing a Spyce active tag is far simpler than writing a JSP tag.

(The chatbox demo gives an example of an active tag.)

Tag libraries must be placed in a separate file from request-handling Spyce pages. The following directives apply specifically to tag library definition:

- [[.**tagcollection**]] :
  Indicates that the current file will be an Active Tag library. Must be at the start of the file.
- [[.**begin** name=*string* [buffers=*True*|***False***] [singleton=*True*|***False***] [kwattrs=*string*]]] :
  Begin defining a tag named *name*. Optional attributes:
  - buffers: if true, Spyce will evaluate the code between the begin and end tags and pass it to the tag as the variable _content. For instance, the following simplistic tag makes its contents bold:

examples/tagbold.spi

```
[[.begin name=bold buffers=True]]

<b>[[=_contents]]</b>

[[.end]]
```

        - singleton: if true, Spyce will not allow paired use of the tag (<tag></tag>) and only allow singleton use (<tag />). If false, the reverse is true.
        - kwattrs: the name of the dict in which to place attributes not specified with [[.attr]] directives. If not given, Spyce will raise an error if unexpected attributes are seen.
    - [[.**attr** name=*string* [default=*string*]]] :
      Specify that the current tag being defined expects an attribute named *name*. If a default string is given, the attribute is optional. (Dynamic attributes may be accepted with the kwattrs option in the begin directive.) (If the default string is prefixed with '=', it will be evaluated as python code at runtime.)
    - [[.**export** var=*string* [as=*string*]]] :
      Specifies that the variable from the tag context named *var* will be exported back to the calling page. The optional *as* attribute may be used to give the variable a different name in the calling context.
    - [[.**end**]] :
      Ends definition of the current tag.

Active tags may specify handlers as in normal Spyce code; this may be done inline with class chunks, or as a reference to a separate .py module. This allows building reusable components easily! Again, the chatbox demo demonstrates this.

(Be careful if you take the class chunk approach with handlers, since all class chunks that get used in a given page are pulled into the same namespace. By convention, tag handlers defined in reusable tags are prefixed with the tag name, e.g., chatbox_addline.)

Active tags should *not* contain f:form active tags; this needs to be done by the .spy page for the Spyce compiler to link up Active Handlers correctly.

One limitation of using the Active Tag directives described here is that tags within a single collection may not call each other. Usually, you can work around this by defining common code inside a .py module and importing that. If this is not an option, you can create Active Tags manually. This is described in the next section.

3.9.5. *Writing Tag Libraries the hard way*

You can still write tag libraries manually if the tag declaration language doesn't give you the tools you need. (About the only functionality it doesn't currently expose is the looping ability used in spy:for.) You may wish to approximate what you want with a new-style tag library and examine the compiler's output (spyceCmd.py -c).

We begin with a basic example:

examples/myTaglib.py

```
from spyceTag import spyceTagLibrary, spyceTagPlus

class tag_foo(spyceTagPlus):
  name = 'foo'
  mustend = 1
  def syntax(self):
    self.syntaxPairOnly()
    self.syntaxNonEmpty('val')
  def begin(self, val):
    self.getOut().write('<font size="%s"><b>' % str(val))
  def end(self):
    self.getOut().write('</b></font><br>')

class myTaglib(spyceTagLibrary):
  tags = [
    tag_foo,
  ]
```

Saving this code in myTaglib.py, in the same directory as your script or anywhere else in the search path, one could then use the **foo** active tag (defined above), as follows:

examples/tag.spy

```
[[.taglib name=myTaglib as=me]]
<html><body>
[[ for x in range(2, 6):{ ]]
  <me:foo val="=x">size [[= x ]]</me:foo>
[[ } ]]
</body></html>
```

Run this code

An active tag library can be any Python class that derives from **spyceTag.spyceTagLibrary**. The interesting aspects of this class definition to implementors are:

- **tags**:
  This field is usually all that requires redefinition. It should be a list of the *classes* (as opposed to instances) of the active tags.
- **start**():
  This methd is invoked by the engine upon loading the library. The inherited method is a noop.
- **finish**():
  This method is invoked by the engine upon unloading the library after a request. The inherited method is a noop.

Each active tag can be any Python class that derives from **spyceTag.spyceTag**. The interesting aspects of the class definition for tag implementors are:

- **name**:
  This field MUST be overidden to indicate the name of the tag that this class defines.
- **buffer**:
  This flag indicates whether the processing of the body of the tag should be performed with the current output stream (unbuffered) or with a new, buffered output stream. Buffering is useful for tags that may want to transform, or otherwise use, the output of processing their own bodies, before it is sent to the client. The inherited default is false.
- **conditional**:
  This flag indicates whether this tag may conditionally control the execution of its body. If true, then the begin() method of the tag must return true to process the tag body, or false to skip it. If the flag is set to false, then return value of the begin() method is ignored, and the body executed (unless an exception is triggered). Some tags, such as the core:if tag, require this functionality, and will set the flag true. Many other kinds of tags do not, thus saving a level of indentation (which is unfortunately limited in Python -- hence the need for this switch). The inherited default is false.
- **loops**:
  This flag indicates whether this tag may want to loop over the execution of its body. If true, then the body() method of the tag must return true to repeat the body processing, or false to move on to the end() of the tag. If the flag is set to false, then the return value of the body() method is ignored, and the body is not looped. Some tags, such as the core:for tag, require this functionality, and will set the flag true. Many other kinds of tags do not, thus saving a level of indentation. The inherited default is false.
- **catches**:
  This flag indicates whether this tag may want to catch any exceptions that occur during the execution of its body. If true, then the catch() method of the tag will be invoked on exception. If the flag is false, the exception will continue to propagate beyond this point. Some tags, such as the core:catch, require this functionality, and will set the flag true. Many other kinds of tags do not, thus saving a level of indentation. The inherited default is false.
- **mustend**:
  This flag indicates whether this tag wants the end() method to get called, if the begin() completes successfully, *no matter what*. In other words, the call to end() is placed in the finally clause of the try-finally block which begins just after the begin(). This is useful for tag cleanup. However, many tags do not perform anything in the end() of their tag, or perhaps perform operations that are not important in the case of an exception. Such tags do not require this functionaliy, thus saving a level of indentation. The inherited default is false.
- **syntax**():
  This method is invoked at compile time to perform any additional tag-specific syntax checks. The inherited method returns None, which means that there are no syntax errors. If a syntax error is detected, this function should return a string with a helpful message about the problem. Alternatively, one could raise an **spyceTagSyntaxException**.
- **begin**(...):
  This method is invoked when the corresponding start tag is encountered in the document. All the attributes of the tag are passed in by name. This method may

return a boolean flag. If **conditional** is set to true (see above), then this flag indicates whether the body of the tag should be processed (true), or skipped (false). The inherited method performs no operation, except to return true.

- **body**(contents):
  This method is invoked when the body of the tag has *completed* processing. It will be called also for a singleton, which we assume simply has an empty body. However, it will not be called if the begin() method has chosen to skip body processing entirely. If the tag sets **buffer** to true for capturing the body processing output (see above), then the string output of the body processing has been captured and stored in **contents**. It is the responsibility of this method to emit something, if necessary. If the tag does not buffer then **contents** will be None, and any output has already been written to the enclosing scope. If the **loops** flag is set to true, then this method is expected to return a boolean flag. If the flag is true, then the body will be processed again, followed by another invocation of this method. And again, and again, until false is received. The inherited tag method performs nothing and returns false.
- **end**():
  This method is invoked when the corresponding end tag is encountered. For a singleton tag, we assume that the end tag immediately follows the begin, and still invoke this method. If the **mustend** flag has been set to true, then the runtime engine semantics ensure that if the begin method terminates successfully, this method *will* get called, even in the case of an exception during body processing. The inherited method is a noop.
- **catch**(ex):
  If the **catches** flag has been set to true, then if any exception occurs in the begin(), body() or end() methods or from within the body processing, this method will be invoked. The parameter **ex** holds the value that was thrown. The inherited method simply re-raises the exception.
- **getPrefix**():
  Return the prefix under which this tag library was installed.
- **getAttributes**():
  Return a dictionary of tag attributes.
- **getPaired**():
  Return true if this is a paired (open and close) tag, or false if it is a singleton.
- **getParent**([name]):
  Return the object of the direct parent tag, or None if this is the root active tag. Plain (inactive) tags do not have objects associated with them in this hierarchy. If the optional **name** parameter is provided, then we search up the tree for an active tag of the same library and with the given name. If such a tag can not be found, then return None.
- **getOut**():
  Return the (possibly buffered) output stream that this tag should write to.
- **getBuffered**():
  Returns true if the tag output stream is a local buffer, or false if the output is connected to the enlosing scope.

Note that Spyce goes to a lot of effort to avoid unnecessary indentation in the code it generates for Active Tags. A limitation of the Python runtime is that the level of indentation within any method is limited by a compile-time constant. You can change it, of course, but in most Python distributions as of this writing (Feb 2005), this is currently set to 100. See for instance this thread from python-bugs.

For convenience, tag implementors may wish to derive their implementations from **spyceTagPlus**, which provides some useful additional methods:

- **getModule**(name):
  Return a reference to a module from the page context. The module is loaded, if necessary.
- ~~**syntaxExist**([must]*)~~:
  Removed in 2.0; Spyce now checks the signature of the begin method; arguments with no default are enforced as required automatically.
- **syntaxNonEmpty**([names]*):
  Ensure that if the attributes listed in **names** exist, then each of them does not contain an empty string value. Otherwise, a spyceTagSyntaxException is thrown. Note that the actual existence of a tag is checked by syntaxExists(), and that this method only checks that a tag is non-empty. Specifically, there is no exception raised from this method, if the attribute does not exist.
- **syntaxValidSet**(name, validSet):
  Ensure that the value of the attribute **name**, if it exists, is one of the values in the set **validSet**. Otherwise, a spyceTagSyntaxException is raised.
- **syntaxPairOnly**():
  Ensure that this tag is a paired tag. Otherwise, a spyceTagSyntaxException is thrown.
- **syntaxSingleOnly**():
  Ensure that this tag is a singleton tag. Otherwise, a spyceTagSyntaxException is thrown.

Despite the length of this description, most tags are trivial to write, as shown in the initial example. The easiest way to start is by having at a look at various implemented tag libraries, such as tags/core.py. The more curious reader is welcome to look at the tag library internals in spyceTag.py and modules/taglib.py. The tag semantics are ensured by the Spyce compiler (see spyceCompile.py), though it is likely easier simply to look at the generated Python code using the "spyce -c" command-line facility.

3.10. Installation

Spyce can be installed and used in many configurations. Hopefully, one of the ones below will suit your needs. If not, feel free to email the lists asking for assistance in using a different setup. If you have successfully set up Spyce by some other method, please also email us the details. Finally, if you had troubles following these instructions, please send suggestions on how to improve them.

3.10.1. *Overview*

Spyce (the core engine and all the standard modules) currently requires **Python version 2.3 or greater**. Spyce uses no version-specific Apache features.

Spyce supports operation through its own webserver, FastCGI, mod_python, Apache proxy, CGI, and the command-line. Some of these require configuration-specific tweaks. These are kept to an absolute minimum, however; where possible, the configuration of the Spyce engine is performed through the Spyce configuration module. The supported adapters are:

- **Web server:** The preferred alternative is to serve Spyce files via its built-in webserver. For production use, it is recommended to do this as a proxy behind another server such as Apache that handles static content, url rewriting, ssl, etc. This is the best option if it is available to you, since you only have a single process (with concurrency provided by multithreading) which makes resource pooling (data, database handles, etc.) an easy way to help your site scale. It also avoids the waste of loading your code into multiple Python interpreters.
- **FastCGI:** This is a CGI-like interface that is relatively fast, because it does not incur the large process startup overhead on each request.
- **mod_python:** This is another relatively performant way to serve up Spyce. If this is that is your chosen Apache integration route, make sure you can first get mod_python running on your system, before adding Spyce to the mix.
- **CGI:** Failing these alternatives, you can always process requests via regular CGI, but this alternative is the slowest option and is intended primarily for those who do not have much control over their web environments.
- **Command line:** Spyce is useful as a command-line tool for pre-processing Spyce pages and creating static HTML files. This documentation, for instance, is produced this way.
- **Others**: Spyce abstracts its operating environment using a thin abstraction layer. Spyce users have written small Spyce adapters for the Xitami webserver and also to integrate with the Coil framework. Writing your own adapter, should the need arise, is therefore a realistic possibility.

The following sections assume that you have already downloaded and uncompressed Spyce. **3.10.2.** ***Web Server***

The preferred method of running Spyce it to run it as a web-server.

The Spyce web server configuration is defined in the "webserver options" section of the Spyce configuration module.

- Start the Spyce web server. The command-line syntax for starting the server is:
  spyceCmd.py -l
  Spyce will now be running on port 8000. You can change the port in your spyceconf module.
- If you are going to proxy Spyce behind Apache (this is done automatically if you installed via RPM):
  1. Copy the proxy section ("Spyce via proxy") from spyceApache.conf into httpd.conf.
  2. Make sure mod_proxy is installed and enabled. Check httpd.conf for a mod_proxy section; look for instructions like "Uncomment the following lines to enable the proxy server."
  3. Restart Apache

3.10.3. *CGI/FastCGI installation*

- Ensure that you have Apache and FastCGI installed and functioning.
- Create a symlink to the command-line executable:
  ln -sf /usr/share/spyce/run_spyceCmd.py /usr/bin/spyce
  or wherever you have chosen to install it.
- Copy the "Spyce via cgi or fcgi" lines from spyceApache.conf to your **/etc/httpd/conf/httpd.conf** file, and replace the **XXX** with the appropriate path to your spyce installation.
- Restart Apache

Alternative CGI configuration: The alternative CGI configuration directs the webserver to execute the .spy file itself, not the Spyce engine. This is appropriate if you do not control Apache on the server you are installing to, e.g., low-end hosting plans. Each .spy file should have execute permissions for the web server, and the first line should be:

```
#!/usr/bin/python /home/username/spyce/run_spyceCGI.py
```

(Adjust to the correct Spyce installation path as necessary.) Then add the following line to the httpd.conf, or to the .htaccess file in the same directory.

```
AddHandler cgi-script spy
```

Finally, ensure that the directory itself has the **ExecCGI** option enabled, and set cgi_allow_only_redirect to False in your Spyce configuration module. For more details, please refer to the Apache documentation, specifically ExecCGI option, Directory, Location, AllowOverrideand Apache CGI documentation, for more information on how to get a standard CGI setup working.

3.10.4. *Mod_Python*

mod_python is primarily recommended if you cannot get FastCGI to work. (Some users have reported problems with FastCGI on Windows.)

Before you try to install Spyce, first get mod_python to work! You may have to compile from sources, and possibly do the same for Python. (See the documentation at: mod_python.)

To use Spyce via mod_python:

- Copy the "Spyce via mod_python" lines from spyceApache.conf to your **/etc/httpd/conf/httpd.conf** file, and replace the **XXX** with the appropriate path to your spyce installation.
- Restart Apache

3.10.5. *Notes for Windows*

Besides your preferred installation option above, remember to add the following line to Apache's httpd.conf:

ScriptInterpreterSource registry

This assumes that Python has registered itself with the Windows registry to run .py files. Otherwise, you can also omit this line, but make sure that the first line of the run_spyceCGI.py file points to a valid Python executable, as in:

```
#! c:/python24/python.exe
```

If you are running using IIS on Windows, you can take a look at how to configure IIS or PWS for Python/CGI scripts.

The basics for getting IIS to work with Spyce are:

- Start the IIS administration console. You get to it from the **Control Panel**. Click on **Administrative Tools**, and then **Internet Services Manager**.
- Drill down to your **Default Web Site**. Right click and select **Properties**.
- Select the **Home Directory** tab, and click on the **Configuration...** button near the bottom right.
- Add an application mapping. On the **executable** line you should type the equivalent of:
  "c:\program files\python22\python.exe" "c:\program files\spyce\spyceCGI.py".
  Set the **extension** to **.spy**, or whatever you like.
  Limit the **Verbs** to **GET,POST**.
  Select the **Script engine** and **Check that file exists** check-boxes.
- Click **OK** twice. Make sure to propagate these properties to all sub-nodes. That is, click **Select All** and then **OK** once more.
- You should now be able to browse .spy files within your website. Note, that this is a very slow mechanism, since it utilizes CGI and restarts the Spyce engine on each request.
- Using the Spyce proxy web server or installing FastCGI are much advised for the vast majority of environments.

3.10.6. *Notes on RPM installation*

- The RPM installation installs Spyce to /usr/share/spyce and creates a /usr/bin/spyce as a symlink to spyceCmd.py. It also configures Apache to forward .spy requests to the Spyce server on port 8000. You're still responsible for starting the spyce server as described in the Web Serverconfiguration section.
- If you upgrading Spyce, it is recommended to uninstall the previous version using **rpm -e spyce**, and then install a fresh copy, as opposed to using the **rpm -U** option.
- Historical note to Redhat users: RH releases prior to 8.0 still used Python version 1.5, as many standard scripts depended on it. If you want to run Spyce with some of the newer Python2 rpms, you will need to change top line of the run_spyceCmd.py, run_spyceCGI.py, run_spyceModpy.py and verchk.py scripts, or reconfigure your path so that the default python is the version that you want.

3.10.7. *Notes on Apache integration*

- If you installed via RPM, Apache is configured to proxy .spy requests to the Spyce webserver. If you are running without the spyce standalone server, comment out those lines in httpd.conf (starting with "Spyce via proxy").
- To avoid confusion, you may wish to change the "root" option in your spyceconf module to be the same as Apache's DocumentRoot.
- You may wish to copy the "Documentation alias" section from spyceApache.conf. This will allow you to access the spyce documentation and examples from http://localhost/spyce/, so you can test your install by browsing to http://localhost/spyce/examples/hello.spy.

3.10.8. *Starting your first project*

Spyce provides a simple method to create a new project. This creates directories for your Spyce project, creates an initial Spyce config file, and sets up other resources.

For example, running

```
python spyceProject.py /var/firstproject
```

results in the following directory structure

```
$ tree /var/firstproject/
  /var/firstproject/
  |-- config.py
  |-- lib
  |-- login_tokens
  `-- www
      |-- _util
      |   |-- form_calendar.gif
      |   `-- form_calendar.js
      |-- index.spy
      |-- parent.spi
      `-- style.css
```

Now you can simply run spyceCmd.py -l --conf /var/firstproject/config.py and browse to http://localhost:8000/index.spy to verify that it worked.

Notes:

- The login_tokens directory is used by the spy:login tags; do not remove it.
- The lib/ directory (also initially empty) is placed on the Python sys.path, meaning you can import any Python module in this directory from any .spy file. It is good practice to put re-usable code into .py files in this directory.

3.11. Programmatic Interface

It is also possible to embed Spyce into another program. All you need is to run or embed a Python interpreter. Although other entry points into the engine code are possible, the most convenient entry points are in **spyce.py**:

- **spyceFileHandler**(request, response, filename, [sig], [args], [kwargs], [config])
- **spyceStringHandler**(request, response, code, [sig], [args], [kwargs], [config])

Either of these functions will execute some Spyce code within the context of some request object and send the output to the corresponding response object. **spyceFileHandler** gets the Spyce code to be executed from a file, while **spyceStringHandler** is passed Spyce code in a string. **3.11.1.** ***Basic usage***

For basic usage, the following arguments need to be understood:

- **request**
  is an object derived from spyce.spyceRequest, denoting the current HTTP request.
- **response**
  is an object derived from spyce.spyceResponse, denoting the HTTP response resulting from the invocation.
- **filename**
  is the name of a file which contains spyce source code (it will be read, compiled, cached and executed when spyceFileHandler is called).
- **code**
  is a string which contains spyce source code (it will be executed when spyceStringHandler is called).
  TODO: how is caching handled in this case?
- **config**
  is an object which contains the configuration to be used. The normal usage is to have a configuration file which can be imported as a normal python module. The imported module then passed as **conf**.

3.11.1. *Passing parameters*

One may wish to also pass a parameter from the calling code into the name space used inside the Spyce code so that it may be accessed from Python Statements, Chunks, Expressions, etcetera. Something analogous to Python's own execfile built-in function where you can pass a locals and globals dictionary.

Spyce code to be invoked is dealt with similar to a function body, and hence one can send parameters to that "function" upon invocation. The arguments **sig**, **args** and **kwargs** are used to control such parameter passing. With **sig**, a signature for this external code is specified,

with **args** and **kwargs** values for the actual arguments are passed.
Thus, **sig** controls what can go into **args** and **kwargs**:

- **sig**
  is a string containing a function signature in usual python syntax, without surrounding brackets.
  For example: 'x, y, a=None, b={}'.
- **args**
  is a list of values, each value corresponding to a positional argument specified in **sig**, in order.
- **kwargs**
  is a dictionary of name to value mappings, its keys should be in the list of keyword argument names specified in **sig**, its values provide actual values to be passed.

3.11.1. *Customized Request/Response classes*

explanation forthcoming; read the code for now, or send an email **3.11.1.** ***Example***

Here's an example that demonstrates such programmatic usage:

examples/programmaticUsage.py

```
import os
import spyce

#-------------------------------------------------[ a hardcoded fake request ]
class TestRequest(spyce.spyceRequest):
   environment = {'QUERY_STRING': '',
         'REQUEST_METHOD': 'get'}
   headers = {}

   def env(self, name=None):
      if not name:
         return self.environment
      return self.environment[name]

   def getHeader(self, type=None):
      return self.headers[type]

   def getServerID(self):
      os.getpid()


#-------------------------------------------------[ a hardcoded fake response ]
class TestResponse(spyce.spyceResponse):
   returncode = spyce.spyceResponse.RETURN_OK
   out = ''
   err = ''
```

```
    headers = {}
    def write(self, s):
        self.out = self.out+s
    def writeErr(self, s):
        self.err = self.err+s
    def close(self):
        pass
    def clear(self):
        self.out = ''
    def sendHeaders(self):
        pass
    def clearHeaders(self):
        self.headers = {}
    def setContentType(self, content_type):
        self.headers['content-type'] = content_type
    def setReturnCode(self, code):
        self.returncode = code
    def addHeader(self, type, data, replace=0):
        self.headers[type] = data
    def flush(self, *args):
        pass
    def unbuffer(self):
        pass


#-------------------------------------------------[ invoking Spyce code ]
import spyceConfig

req = TestRequest()
resp = TestResponse()

spyceCode  = 'Hello, the following names are defined: "[[print dir(),]]", and '
spyceCode += 'these were the parameters passed: [[print (a, b, c, d)]]\n',

spyce.spyceStringHandler(
            req,
            resp,
            spyceCode,
            sig='a, b, c="asd", d=None',
            args=(1, 'two'),
            kwargs={'c': 'aa'},
            config=spyceConfig)
print resp.err
print resp.out
```

4. ADDENDA

List of appendices:

4.1. Performance

Although flexibility usually outweighs raw performance in the choice of technology, it is nice to know that the technology that you have chosen is not a resource hog, and can scale to large production sites. The current Spyce implementation is comparable to its cousin technologies: JSP, PHP and ASP. We ran a micro-benchmark using **hello.spy** and equivalents. All benchmark files are available in the misc/benchmarkdirectory.

examples/hello.spy

```
<spy:parent title="Hello" />
[[ import spyce ]]

Hello from Spyce version
[[= spyce.__version__ ]]!
```

Run this code

Spyce was measured under CGI, FCGI, mod_python and proxy configurations. For calibration the static HTML, CGI-C, CGI-Python and FCGI-Python tests were performed. In the case of CGI-C, the request is handled by a compiled C program with the appropriate printf statements. In the case of CGI-Python, we have an executable Python script with the appropriate print statements. FCGI-Python is a similar script that is FCGI enabled. ASP was measured on a different machine, only to satisfy curiosity; those results are omitted.

Configuration	Hello world!
Spyce-modpython	**250**
modpython publisher	300
Spyce-proxy	**200**
JSP	100
PHP	450
Spyce-FCGI	**100**
Python-FCGI	140
Spyce-CGI	**8**
Python-CGI	25
C-CGI	180
Static HTML	1500

The throughput results (shown above in requests per second) were measured on a Intel PIII 700MHz, with 128 MB of RAM and a 512 KB cache running RedHat Linux 7.2 (2.4.7-10 kernel), Apache 1.3.22 and Python 2.2 using loopback (http://localhost/...) requests. Since each of the script languages requires an initial compilation phase (of which JSP seems the longest), the server was warmed up with 100 requests before executing 1000 measured requests with a concurrency level of 3, using the ab (Apache benchmark) tool. Figures are rounded to the nearest 25 requests/second.

Conclusion: All spyce configuration options except CGI can handle large websites, as the Spyce engine and cache persist between requests. The CGI version takes a hit in recompiling Spyce files on every request. (This may be alleviated using a disk-based Spyce cache (as opposed to the current memory-based implementation).)

4.2. History

The initial idea for a Python-based HTML scripting language came in May 1999, a few months after I (Rimon) had first learned of Python, while working with JSP on some website. The idea was pretty basic and I felt that someone was bound to implement it sooner or later, so I waited. But, nobody stepped up to the task, and the idea remained little more than a design in my head for two and a half years. In early 2002, after having successfully used Python extensively for various other tasks and gaining experience with the language, I began to revisit my thoughts on a Python-based HTML scripting language, and by late May 2002 the beta of version 1.0 was released.

Version 1.0 had support for standard features like get and post, cookies, session management, etc. Development was still on-going, but Spyce was mature and being used on live systems. Support of various features was enhanced for about a week or two, and then a new design idea popped into my head. **Version 1.1** was the first modular release of Spyce. Lots of prior functionality was shipped out of the core engine and into standard modules. Many, many new modules and features were added. Spyce popularity rose to the top percentile of SourceForge projects and the user base grew. **Version 1.2** represented a greatly matured release of Spyce. Spyce got a totally revamped website and documentation, and development continued. **Version 1.3** introduced active tags. More performance work, more modules, etc.

In February 2005 Jonathan Ellis was hired by SilentWhistle to work on their web application, written in Spyce. He began by overhauling the Active Tag code and was soon deep in the guts of Spyce. Jonathan added Active Handlers, parent templates, and the tag compiler on the way to **Version 2.0**.

In July 2006 **Version 2.1** introduced the Spyce login tags and database integration via SQLAlchemy, as well as improvments to parameter marshalling for Active Handlers.

For more detail, please refer to the change log. As always, user feedback is welcome and appreciated.

Ressources : http://spyce.sourceforge.net/