The SkunkWeb STML Reference

Release 3.0

Drew Csillag

January 29, 2007

This file documents the SkunkWeb Web Application Framework.

CONTENTS

CHAPTER

ONE

Introduction to STML

The SkunkWeb Enviroment gives you two ways to do programming for your web applications: with *code modules* written in the programming language Python, and with *templates*, which are text documents with embedded programming code. In SkunkWeb, these templates' embedded programming code is written in a language called STML, or Skunk Template Markup Language. STML is like other markup languages such as HTML or XML, in that it uses tags to express its instructions.

This guide introduces the syntax and conventions of STML, and also includes a reference for all STML tags.

STML documents are text (usually HTML) documents with embedded programming tags. These programming tags are written in the special tag language we call STML (Skunk Template Markup Language).

At first glance, an STML document looks just like a regular text document. Its filename ends in `.html` (or `.txt` or whatever you want the output to be). Once you look at the document, however, you will notice the STML tags, which look like HTML tags with colons (:) inside them:

```
<:tag_name:>
```

SkunkWeb can recognize these STML tags, and follow the instructions you give in the tags.

STML tags are more complex and powerful than HTML tags. In HTML, a tag has a name (such as BR), and optionally some *attributes*, which are pairs of strings with a name and a value:

```
<A HREF="index.html">
```

In the above HTML tag, A is the tag's name, and the tag has one attribute. The attribute's name is HREF, and its value is the string `"index.html"` (quotes not included). In HTML, attribute values are always *strings*, and the quotation marks around a string are required only if there are spaces or tabs in the string.

STML tags are similar. However, STML uses Python! So the values of attributes can be things other than strings. Consider this STML tag:

```
<:happy_tag foo=hi bar="Hello there"
     baz=5 dib='myVar':>
```

The name of the STML tag is `happy_tag`. (This tag is not a real STML tag, of course.) The tag has four attributes, with keys `foo`, `bar`, `baz`, and `dib`. Looks like an HTML tag so far, correct? But look at the attribute *values*.

The value of attribute foo is the `"hi"` (quotes not included). It's a string; if you type something without any quotes, STML assumes the value is a string. The value of attribute `bar` is `"Hello there"` (quotes not included in the value). It's also a string, of course. When you quote string values in STML, you can use Python's way of quoting

strings with single or double quotes.

The value of attribute `baz` is `5`. In HTML, this would be the string `"5"`, and so it is in STML. But STML, which uses Python, is smart enough to know about Python's real data types, such as integers and floating-point numbers.

Therefore, if you want an attribute value to be something other than a string, use STML's best feature: the *backticks*. Any value which you enclose in backticks `‘like this‘` will be taken to be a Python expression. Thus when you say `bar=5`, STML sees the string `"5"` for `bar`, but when you say `bar=‘5‘`, STML sees the Python expression `5`, which it knows is an integer.

In short, the backticks tell STML to evaluate the contents of the backticked expression. Otherwise, the value will be taken as a string argument. This is analogous to quoting in regular Python: regular quotes (double or single) do not evaluate their contents but backticks do evaluate their contents.

The attribute `dib` is an example of more powerful uses of backticked Python expressions: the use of Python variables. The `dib` argument has a value `‘myVar‘`, which in Python means the Python object in the variable named `myVar`. Now you have a real programming enviroment in your STML document! You can use certain STML tags to assign strings and other Python objects to variable names, and then use those variable names in later STML tags:

```
<:set name=myVar expr=‘56.7‘:>
<:val expr=‘myVar‘:>
```

The STML `<:set:>` (see section 2.2, page 6) tag lets you assign the result of a Python expression in the attribute named `expr` to a variable name in the attributed named `name`. In the example, you assign the floating point number `56.7` to the variable `myVar`. Then the `<:val:>` (see section 2.1, page 5) tag displays the result of a Python expression in the attribute named `expr` as HTML output for the user.

You can do a lot inside the backticks. Any Python expression is legal: make calls to functions, make instances of Python classes, do arithmetic, do boolean logic with `and` and `or`. Python has a built-in function `int()`, for example, which takes an object (like a floating-point number) and turns it into an integer:

```
<:set name=myVar expr=‘56.7‘:>
<:set name=myOtherVar expr=‘int(myVar)‘:>
```

STML sees the Python expression `int(myVar)` and executes it; then it puts the result to the variable `myOtherVar`. `myOtherVar` now contains the integer `56`.

STML has one other feature that is very different from HTML: in most cases, attributes do not have to be specified by name, because the order in which the tag expects the attributes is always the same. Thus you can specify just values without names, in the order expected by the tag. (This reference guide tells you the expected order of attributes of each and every STML tag.)

For instance, the `<:set:>` tag in STML expects an attribute named `name` followed by and attribute named `expr`. Instead of typing

```
<:set name=myVar expr=‘56‘:>
```

all the time, you can just type the values for `name` and `expr` in that order:

```
<:set myVar ‘56‘:>
```

If you do use attribute names in STML tags, the order of attributes does not matter, because STML will see them by name and disregard the order of the attributes. The following example works, although you would probably not do it in practice:

```
<:set expr='56' name=myVar:>
```

A few STML tags do not allow you to drop the attribute names in certain cases. Their documentation in this manual will always tell you when you cannot.

CHAPTER

TWO

Setting and Showing Things

2.1 <:val:>

```
<:val expr [fmt="plain"]:>
```

Causes the Python expression `expr` to be evaluated, and the resulting Python object is converted to a string and displayed at that place in your document. (All Python objects can be turned into strings.)

For example, if you would like to display the result of 5 + 2:

```
<:val ‘5 + 2‘:>
```

The `<:val:>` tag is just like calling the `str()` function in Python on the resulting object, with one exception: if the object is the `None` object, `<:val:>` prints an empty string `""` instead of the string `"None"`. The following two tags show the value of the variable named `myVar` and show nothing, respectively:

```
<:val ‘myVar‘:>
<:val ‘None‘:>
```

`<:val:>` only accepts Python expressions as its attribute, not full Python statements. That means that you cannot assign objects to variable names with this tag. Use the `<:set:>` (see section 2.2, page 6) tag to assign objects to variable names. You can, however, call functions and classes, and make boolean expressions with `and`, `or`, and `not`. For example, if you want to call a function `myFunction`, which returns some HTML to include in your document:

```
<:val ‘myFunction()‘:>
```

The string printed by `<:val:>` is just a plain string by default; no special formatting takes place. Sometimes you want the string to be formatted in a special way: you want HTML special characters escaped, or you want all characters escaped so that the string can be made part of a URL. The `fmt` argument lets you specify one of many common formatting or escaping operations on the string:

- `"plain"` or `"plaintext"`: No escaping. This is the default.
- `"latin"` or `"latinquote"`: Escape only extended characters to ISO Latin1 entities: `í` become `&iacute;`, etc.
- `"html"` or `"htmlquote"`: Escape only things that would break HTML formatting: `< > & ‘‘ ’`. These

characters are escaped to the named HTML entities for them: `<`, etc.

- `"uri"`, `"url"`, `"uriquote"` or `"urlquote"`: Escape all URL-unsafe characters to `%XX` format.
- `"fulluri"` or `"fullurl"`: Escape every character into URL-safe `%XX` format.
- `"base64"`: base64 encodes the string.

Examples of `fmt`, with the output shown after each example:

```
<:set myVar "Click here >>>>":>

<:val ‘myVar‘:>
Click here >>>>

<:val ‘myVar‘ fmt="html":>
Click here &gt;&gt;&gt;&gt;

<:val ‘myVar‘ fmt="url":>
Click%20here%20%3E%3E%3E%3E

<:val ‘myVar‘ fmt="fullurl":>
%43%6C%69%63%6B%20%68%65%72%65%20%3E%3E%3E%3E
```

The `<:val:>` tag is equivalent to the following code executed in a Python component:

```
import DT.DTCommon
print DT.DTCommon.ValFmtRgy[fmt](expr)
```

2.2 <:set:>

```
<:set name value:>
```

Allows you to assign the object in `value` to the local variable named in `name`. Use this tag to do all of your assignments to variables.

You can also use the `<:call:>` (see section 2.3, page 7) tag to make assignments, by putting a regular Python assignment statement in backticks: `<:call ‘foo = 567‘:>`. STML still lets you do that. Use the `<:set:>` tag to assign objects to simple variable names. Use the `<:call:>` for assignments that are more complex; see its reference page for details.

These two tags assign the integer 567 to the variable name `myVar`, and then assign a new object returned by calling `module.function` to the same variable named `myVar`.

```
<:set myVar ‘567‘:>
<:set myVar ‘module.function()‘:>
```

2.3 <:call:>

```
<:call expr:>
```

Allows you to execute the Python expression in `expr`. Any object returned by the expression is ignored, and *no output is included in your template*. Think of this tag as being just like the `<:val:>` tag, except that `<:call:>` produces no output.

Examples:

```
<:call ‘function()‘:>
<:call ‘module.function(var1, var2)‘:>
```

You can also use this tag to do complex assignments that the `<:set:>` (see section 2.2, page 6) tag cannot handle, such as assignments to items in a list or dictionary, or to attributes in an object. Here are some examples, with `<:set:>` used for the simple assignments to variable names:

```
<:set myList ‘[1,2,3,4]‘:>
<:call ‘myList[0] = function()‘:>
<:set myDict ‘{}‘:>
<:call ‘myDict["foo"] = "Hey there"‘:>
<:set myObj ‘newObject()‘:>
<:call ‘myObj.someAttribute = 0‘:>
```

2.4 <:spool:>

```
<:spool name:><:/spool:>
```

Takes the content generated between the open and close `<:spool:>` tags and assigns it to the local variable named `name`. The name argument cannot be a Python expression, only a string with the variable name.

This tag is deceptively powerful: you can execute any STML you wish, collect all the HTML output into a single string, and do anything to that string after ending the `<:spool:>`. In this example, the variable `spoolVar` would end up containing the string `"\n<A HREF=index.html>Click here</A>\n"`:

```
<:set myUrl "index.html":>
<:spool spoolVar:>
<:url path=‘myUrl‘ text="Click here":>
<:/spool:>
```

Spooling is equivalent to the following Python code in a Python component:

```
import sys, cStringIO
tempname = sys.stdout
sys.stdout = cStringIO.StringIO()
# stuff between <:spool:> tags
name = sys.stdout.getvalue()
sys.stdout = tempname
```

Sometimes a debugging feature of SkunkWeb, component comments, can be a nuisance when you are spooling the contents of component calls. To get around this you can temporarily reset the value of the configuration variable `componentCommentLevel` that controls this feature:

```
<:import SkunkWeb.Configuration as=C:>
<:set oldval 'C.componentCommentLevel':>
<:call 'C.componentCommentLevel=0':>
<:spool foo:><:component foo.comp:><:/spool:>
<:call 'C.componentCommentLevel=oldval':>
```

or, in STML (as of SkunkWeb 3.4), it is equivalent to write:

```
<:spool myspool comments='0':>
  There won't be component comments here! <:component mycomp.comp:>
<:/spool:>
```

The `comments` argument can take the same values as `componentCommentLevel` itself: 0 (no comments) through 3 (verbose comments).

2.5 <:args:>

```
<:args [argname] [argname1=expr] [argname2=expr...]:>
```

This tag makes short and quick work of handling arguments submitted by the user of your website through a GET or POST request. It allows you to copy CGI arguments into local variables of the same name, convert the CGI arguments (which are always strings) into Python data types, and assign default values to those variables if the conversion fails or if the caller failed to pass them in.

Each argument of this tag specifies a name, and optionally a Python expression value. The name of each argument indicates the name of the user argument *and also* the name of the local variable to hold the value. If you just specify this name, it will copy the user argument of that name into a local variable of that name. If the user passed no argument by that name, it assigns the `None` object to the variable.

If you do specify a Python expression as a value, it will examine the expression, and take the following actions:

1. If the expression results in a function or other "callable" object, it is a "conversion" function. It will take the user argument of that name, call the function with it, and assign the returned value to the local variable of that name. If the function raises an exception, or if the user passed no argument of that name, it assigns `None` to the local variable:

```
<:args this='int':>
```

In the above STML, the user argument named `this` will be converted into an integer with the built-in Python function `int`. Then the result will be assigned to a local variable `this`. If anything goes wrong, or if the user passed no `this` argument, the local variable `this` will be `None`.

2. If the expression results in a regular, or "non-callable", object, the it considers this object the default value. If the user passed no argument of that name, the templating engine assigns this object to the local variable of that name.

3. If the expression is a list or tuple of two items, the templating engine considers the first item to be the conversion function, and the second item to be the default value:

```
<:args this='(int, 5)':>
```

In this example, SkunkWeb looks for the user argument named `this`. If it's there, SkunkWeb calls the function `int` with the value, and then assigns the rsult to the local variable `this`. If the user passed no arugment named `this`, or if the call to `int` fails in some way, SkunkWeb assigns the integer 5 to `this`.

This tag only needs to be used once, near the beginning of the STML document which serves the Web request. However, you may use it as many times as you like. Each time this tag executes, it looks in the SkunkWeb dictionary object `CONNECTION.args`, extracts values from it, and performs the necessary conversion and variable assignments. You cannot use this tag in an STML component, however; STML components do not have access to any part of the `CONNECTION` object (unless of course, the `CONNECTION` object was passed to it explicitly).

Also note that you are not limited to the built-in Python functions `int`, `float`, `long`, etc. for your conversion functions. Any function, or any callable object for that matter, can be used. Here's a large example of an `<:args:>` tag that does a lot of work:

```
<:import Date:>
<:args foo
       bar='0'
       baz='(int, 5)'
       dib='(float, 0.0)'
       gad='(Date.LocalDate, Date.LocalDate())'
:>
```

As of SkunkWeb 3.2, the `CONNECTION` object has a method, `extract_args`, which can be used for the same purpose whenever the `CONNECTION` object is present (e.g., in python top-level documents). `CONNECTION.extract_args` returns a dictionary of extracted values; to achieve the same result as above in pure Python, you would do the following:

```
import Date
d=CONNECTION.extract_args('foo',
                          bar=0,
                          baz=(int, 5),
                          dib=(float, 0.0),
                          gad=(Date.LocalDate, Date.LocalDate()))
locals().update(d)
```

2.6 <:compargs:>

```
<:compargs [argname] [argname1=expr]
     [argname2=expr...] [**kwargname]:>
```

This tag allows you to create a component signature, i.e. specify which arguments the component expects and provide some default values for them. When someone calls your component, SkunkWeb will make sure that the arguments it is called with match the signature. The arguments specified by the `<:compargs:>` tag will appear in the local namespace. The tag is optional, if you do not specify it, all of the arguments passed to the component will appear in the local namespace. You can optionally specify default values for the arguments. If an argument is not passed by the caller and it has a default value, the default value will be used for the local variable

If your component accepts variable number of arguments, you can use the ***kwargs* notation to specify a container for any additional arguments. Any arguments not explicitely listed in the `<:compargs:>` tag will be stored in the *kwargs* variable. If you do not specify such a variable, and the component is called with arguments not listed in the `<:compargs:>` tag, an exception will be raised.

Suppose your component has the following signature:

```
<:compargs x y='10' z='hello' **kwargs :>
```

Suppose someone calls the component with following call:

```
<:component comp x='hi' y='11' extra='12':>
```

In this case the following variables will appear in local namespace:

`x` The value is string 'hi'

`y` The value is integer 11

`z` The value is string 'hello', taken from default

`kwargs` The value is a dictionary: `{'extra' :  12 }`

2.7 <:default:>

```
<:default name [value]:>
```

If in the current STML document, there exists no local variable called `name` (or it's value is `None`), create it and assign it the object in `value`. The `value` argument defaults to the empty string.

This tag is often used in STML components, which are STML documents that can be embedded in other STML documents. Components are called with argument attributes, which end up as local variables in the component document. If the calling STML document does not pass a variable you need, it won't be in your component document, and when you try to use it, you will get a Python `NameError`, meaning that there is no variable under that name. This tag makes sure that the variable name exists and that it has a value.

2.8 <:date:>

```
<:date [fmt] [date]
     [lang] [to_zone] [from_zone]:>
```

Displays a string representation of a date/time object. This is an STML tag equivalent of the `DateString` function in the Skunk pylibs `Date` module.

Note that all attribute arguments to this tag are optional.

`fmt` must be a string, and specifies the format in which to display the date/time. If not specified, `fmt` defaults to the `Date` module's default format, which is the ISO format `yyyy-mm-dd hh:mi:ss`.

The `fmt` string may contain any combination of the following format elements. Anything not recognized as a format is included in the output as static text:

`yyyy` four-digit year

`yy` two-digit year

`mm` two-digit month

`dd` two-digit day

`hh24` 24-hour hour

`hh12` 12-hour hour

`hh` equivalent to `hh24`

`mi` or `min` minutes

`SS` seconds

For all of the above, leading zeroes are included as necessary: "01" instead of 1, etc. However, if the format element is in UPPERCASE, leading zeroes are not included: "MM" `->` "1", "mm" `->` "01".

`mon` abbreviated name of month

`month` full name of month

`dy` abbreviated weekday name

`day` full weekday name

`am or pm` whether time is AM or PM

For all of the above, if the element is in UPPERCASE, the result will be in ALL CAPS: "MONTH" -¿ "JANUARY". If the element is Initcap, the result will be initcapped: "Month" -¿ "January". Any other combination of case will be considered lowercase: "month" or "moNth" -¿ "january".

`date` is a Python expression resulting in a date/time (strings are *not* allowed). If not specified, it defaults to the current local date/time of the machine.

`lang` must be a string specifying the language in which to format the date/time, using the Skunk convention of `'eng' 'esp' 'por'` for English, Spanish, and Portuguese, respectively. If not given, it defaults to `'eng'`.

`to_zone` is a timezone argument, meaning, "Please convert the `date` argument to this timezone before displaying." It defaults to the SkunkWeb server machine's local timezone. See the `Date` module for a full discussion of timezone support. (**Note:** This argument formerly had the name `timezone`. STML still understands the old argument name. If you specify both `to_zone` and a `timezone` argument, STML will use `to_zone` and ignore `timezone`.)

`from_zone` is a timezone argument, meaning, "Please convert the `date` argument from this timezone into the SkunkWeb server machine's local timezone before doing anything else." See the `Date` module for a full explanation. **Note:** This argument used to be named `srctimezone`. STML still understands the old argument name. If you specify both `from_zone` and a `srctimezone` argument, `srctimezone` will be ignored.

Here are some examples, if the current date is 2000-08-01 20:34:56 (a Tuesday), and your server is running on Eastern Daylight Time, which is four hours behind UTC (Greenwich Time). Each tag is followed by the output which would be displayed:

```
<:date:>
2000-08-01 20:34:56

<:date "dd mon yy":>
01 aug 00

<:date "hh12:mi am":>
08:34 pm

<:date "HH12:mi am":>
8:34 pm

<:date "hh12:mi am" to_zone="UTC":>
12:34 am

<:date "Month DD, yyyy, hh12:mi am" to_zone="UTC":>
August 2, 2000, 12:34 am

<:date "DD month yyyy, hhhmi" lang="esp" to_zone="UTC":>
2 agosto 2000 00h34
```

To do the equivalent in Python components, if unspecified arguments are `None`, is:

```
import Date
temp_date = Date.Convert ( date, to_zone or 'LOCAL', from_zone or 'LOCAL' )
print Date.DateString ( temp_date, fmt, lang )
```

2.9 <:del:>

```
<:del name:>
```

This tag deletes the local variable named by `name` from your STML document. It's identical to the Python statement `del` *`name`*. You will use this tag very rarely, but it's there if you need it.

In this example, the variable named `myVar` is set to the integer `5`, then deleted from the variable namespace. The third tag, which tries to access the variable name, results in a `NameError`, meaning that nothing exists under that name:

```
<:set myVar ‘5‘:>
<:del myVar:>
<:val ‘myVar‘:>
```

CHAPTER

THREE

Flow Control and Error Handling

3.1 <:if:>, <:elif:>, <:else:>

```
<:if expr:>
   [ <:elif expr:> ]
   [ <:else:> ]
<:/if:>
```

If the Python expression `expr` evaluates to true, render until an `<:/if:>`, `<:else:>` or `<:elif:>` tag is encountered. The `expr` arguments in `<:if:>` and `<:elif:>` tags *must* be enclosed in backticks, of course.

If it finds an `<:elif:>` tag and nothing has yet been rendered within this `<:if:>` block and `expr` evaluates to true, renders until it runs into another `<:elif:>`, `<:else:>` or `<:/if:>` tag. If an `<:else:>` is encountered and nothing has yet been rendered in this `<:if:>` block, renders until it hits the `<:/if:>`, `<:elif:>` or another `<:else:>` (if you put an `<:else:>` before an `<:elif:>` or another `<:else:>`, which while syntactically valid, is erroneous since the contents following the `<:elif:>` or `<:else:>` will never be rendered).

This is just a fancy way of saying that these tags work just like `if`, `elif`, and `else` in Python.

Using `<:elif:>` outside of an `<:if:>` block will raise an error. Using `<:else:>` outside of an `<:if:>`, `<:try:>` or `<:for:>` block will also raise an error.

In the following example, the output will be `Shaft is the Man.`

```
<:set myVar ‘1‘:>
<:if ‘myVar > 0‘:>
    Shaft is
    <:if ‘myVar == 6‘:>
    the black private dick.
    <:elif ‘myVar == 4‘:>
    a sex machine with all the chicks.
    <:else:>
    the Man.
    <:/if:>
<:else:>
    Why is myVar not greater than 0?
<:/if:>
```

3.2 <:for:>

```
<:for expr [name=sequence_item]>
    [ <:break:> ]
    [ <:continue:> ]
    [ <:else:> ]
<:/for:>
```

The <:for:><:/for:> tag implements a loop: it expects the expression in expr to return a sequence of items (a tuple or list, for example). For each item in the sequence, <:for:> assigns the item to the variable name given in the name argument, and does whatever is inside the <:for:> tag. The name parameter is optional; if not specified, <:for:> will assign each item to a local variable called sequence_item.

If the sequence in expr is empty or false, <:for:> will do nothing, unless you put an <:else:> tag inside the <:for:> block. The <:for:> tag will execute everything after an <:else:> tag if the sequence in expr is empty.

If you know that the items of the sequence will be assignable to a tuple, you can put a string representation of a tuple in the name parameter, just as in Python.

Examples:

```
<:for ‘(1,2,3)‘:>
current loop value: <:val ‘sequence_item‘:><BR>
<:/for:>

<:for ‘[1,2,3]‘ foo:>
current loop value: <:val ‘foo‘:><BR>
<:/for:>

<:for ‘[]‘ foo:>
This wouldn’t show
<:else:>
Loop is empty
<:/for:>

<:for ‘d.items()‘ "(k, v)":>
 <:val ‘"%0.2f : %0.2f" % (k/2, v/3)‘:>
<:/for:>
```

There are two special tags which allow you to force either an exit from the loop or an immediate return to the top of the loop: <:break:> and <:continue:>. The <:break:> tag will cause an immediate exit from the <:for:> tag. The <:continue:> tag will cause an immediate return to the top of the <:for:> tag, where it will assign the next item in the sequence to the variable, or if there are no items left in the sequence, it will exit the <:for:> tag. For example, this STML block will print the numbers 1 2 3:

```
<:set myList ‘[1,2,3,4,3,2,1]‘:>
<:for ‘myList‘ item:>
   <:if ‘item > 3‘:>
       <:break:>
   <:/if:>
   <:val ‘item‘:>
<:/for:>
```

And this STML block will print the numbers `1 2 3 3 2 1`:

```
<:set myList '[1,2,3,4,3,2,1]':>
<:for 'myList' item:>
   <:if 'item > 3':>
       <:continue:>
   <:/if:>
   <:val 'item':>
<:/for:>
```

3.3 <:while:>

```
<:while expr:>
   [ <:break:> ]
   [ <:continue:> ]
<:/while:>
```

This tag evaluates the expression `expr`. If the expression is true, it executes all of the things inside the tag. Then it repeats this process until the expression is false. It is functionally identical to the Python `while` statement.

This tag is vulnerable to infinite loops, unless you do something inside the tag to change the value of the expression, the tag will execute indefinitely, until the SkunkWeb server calls a timeout and raises an error. For example, this STML will cause an infinite loop:

```
<:set myVar '1':>
<:while 'myVar':>
   hello, i'm looping
<:/while:>
```

There are two special tags which work the same way they do inside a `<:for:>` tag: `<:break:>` and `<:continue:>`. The `<:break:>` tag will cause an immediate exit from the `<:while:>` tag. The `<:continue:>` tag will cause an immediate return to the top of the `<:while:>` tag, where it will evaluate the expression again and continue executing if the expression is true. For example, either of these STML blocks will print "looping" 100 times:

```
<:set myVar '1':>
<:while 'myVar <= 100':>
   looping
   <:set myVar 'myVar + 1':>
<:/while:>
```

```
<:set myVar ‘1‘:>
<:while ‘1‘:>
   looping
   <:if ‘myVar > 100‘:>
       <:break:>
   <:set myVar ‘myVar + 1‘:>
   <:/if:>
<:/while:>
```

3.4 <:break:>

```
<:break:>
```

When inside a looping tag, tells the templating engine to exit the loop immediately. Not allowed outside of a looping tag. See the <:for:> (see section 3.2, page 16) tag and the <:while:> (see section 3.3, page 17) tag for examples.

3.5 <:continue:>

```
<:continue:>
```

When inside a looping tag, tells the templating engine to return immediately to the top of the loop and continue executing. Not allowed outside of a looping tag. See the <:for:> (see section 3.2, page 16) tag and the <:while:> (see section 3.3, page 17) tag for examples.

3.6 <:try:>, <:except:>, <:else:>, <:finally:>

```
<:try:>
    <:except:>
    [ <:else:> ]
<:/try:>
```

or

```
<:try:>
    <:except [exc]:>
    [ <:except [exc]:> ]
    [ <:except:> ]
    [ <:else:> ]
<:/try:>
```

or

```
<:try:>
    <:finally:>
<:/try:>
```

This tag executes everything inside of it up until it first sees an `<:except:>` or `<:finally:>` tag. If an exception (error) occurs during that execution, this tag will then take one of the following actions:

1. If there is an `<:except:>` tag which specifies the exception in its `exc` argument, it will execute the stuff occurring after that `<:except:>` tag until it meets another `<:except:>` tag, an `<:else:>` tag, or the end of the `<:try:>` tag.
2. Otherwise, if there is an `<:except:>` tag with nothing specified in `exc`, it executes the stuff after that `<:except:>` tag.
3. Otherwise, there is nothing to catch the exception, so it "propagates" upward to another containing `<:try:>` block. If there is no containing `<:try:>` tag, SkunkWeb generates a page error.

If you have an `<:else:>` tag after all of your `<:except:>` tags, the stuff after `<:else:>` will be executed if no exception has occurred.

If you wish to specify particular exceptions in an `<:except:>` tag, use a backticked expression with the exception object. If you want to specify more than one exception for an `<:except:>` tag, specify them as a tuple expression.

Example:

```
<:try:>
    <:call ‘something_which_will_blow_up()‘:>
<:except ‘KeyError‘:>
    We got a KeyError.
<:except ‘(TypeError, ValueError)‘:>
    We got a TypeError or ValueError.
<:except:>
    We got some other error.
    But I know it wasn’t a KeyError, TypeError, or ValueError!
<:else:>
    Hey, no error!!!!
<:/try:>
```

All of this behavior is identical to the way that `try:`, `except:` and `else:` work in regular Python. Read the Python documentation about exceptions for more info.

Note the last way to use the `<:try:>` tag: with a `<:finally:>` tag. The tags behave the same way as Python's `try:` and `finally:` statements.

CAVEAT PROGRAMMER: beware of the `<:except:>` tag with no arguments, as you may catch the document timeout exception (`requestHandler.requestHandler.DocumentTimeout`). If you catch it and ignore it, your document will *never* time out. Moral of the story, either:

1. Don't use the `<:except:>` tag without arguments
2. Make sure you don't do anything in your `<:try:>` block that may block.

FLASH! As of SkunkWeb 3.2.3, the document timeout exception will be reraised every second after a timeout occurs, so while you still need to be careful (say you have an operation that times out, wrapped in a `try:` block with a catchall `except:` in a while loop), you don't have to be quite as careful.

3.7 <:raise:>

```
<:raise [exc]:>
```

Raises the exception `exc`, or if `exc` isn't specified, the current exception if one was previously caught. If you specify `exc` without backticks, the tag will raise the exception as the string you indicated:

```
<:raise BadBadError:>
```

will raise the *string* `"BadBadError"` as the exception. If you want to raise a real exception, such as one of Python's built-in exceptions or an exception class from a module, you must put it in backticks. For example, if you want to reject the value of some variable passed to you with the Python built-in exception `ValueError`:

```
<:raise `ValueError`:>
```

Please note that if you do not specify `exc`, and an exception has not previously occurred, SkunkWeb will raise a `TypeError` exception with the message "exceptions must be strings, classes, or instances".

This tag behaves identically to the `raise` statement in Python.

3.8 <:halt:>

```
<:halt:>
```

Causes execution of the current STML document or component to stop entirely. If the templating engine is executing a component or include, it stops executing that component or include and returns execution to the template which did the include or component call. If there is no calling template (you're at the top level template), then the templating engine stops executing any STML, and then returns the response to the user as it currently stands.

CHAPTER

FOUR

Including or Calling Other Documents

The biggest benefit of STML documents is that you can reuse them inside other STML documents. We call these embeddable STML documents components. They have a different file extension: while your regular STML documents end with the extension of the kind of output they generate (like `.html`), all STML component documents end with the file extension `.inc` for include components, `.comp` for regular components, (or `.dcmp` for data components; more about those later).

If you have a block of HTML that must be included on every page of your website, make it an STML component, and then your other STML templates can embed the template in their own output. This lets you use an efficient, modular design when building your website.

There are two very different ways to embed an STML document in another one: by including the embedded template's STML code, or by calling it as an STML component. Let's examine the differences between include and component in the next sections.

4.1 `<:include:>`

```
<:include name:>
```

The STML tag `<:include:>` executes an STML component document named `name` in the namespace of the currently executing component.

The format of the string for `name` is a path, such as `/foo/bar.inc` or `foo.inc`. If the path is *absolute*, starting with a slash, the templating engine will look for the component at that path underneath your `documentRoot` of STML documents. If the path is *relative*, not starting with a slash, the templating engine looks for the component inside the current document's directory. You can also use '`..`' to move up a directory, just as you do on a Unix or DOS command line (but you cannot escape the `documentRoot`).

For example, you have these three files in your documentRoot directory:

```
/header.inc
/myDir/index.html
/myDir/foo.inc
```

Inside `/myDir/index.html`, you want to include both component files. You can use these tags, the first using an absolute path and the second using a relative path:

```
<:include /header.inc:>
<:include foo.inc:>
```

Unlike the <:component:> (see section 4.2, page 22) tag, there is no caching, nor can you pass any attributes in the tag. Any variables that the included template creates, deletes, or changes take effect in the including template! For example, you have a template /index.html with the following code:

```
<:set myVar ‘5‘:>
<:include foo.inc:>
<:val ‘myVar‘:><BR>
<:val ‘myOtherVar‘:>
```

and the STML component document foo.inc contains this STML code:

```
<:set myVar ‘66‘:>
<:set myOtherVar "hey man":>
Hey I’m done!
```

then the output of /index.html is:

```
Hey I’m done!
66
hey man
```

A way to do includes from Python code is as follows:

```
import AE.Component
print AE.Component.callComponent(name, argdict,
        compType=AE.Component.DT_INCLUDE)
```

Re file extensions: The .inc and .pyinc file extensions were added in SkunkWeb 3.2; previous versions used the .comp and .pycomp file extensions for includes, and these will still work. However, since it is rather important to distinguish the two types of components, using the new include extensions is recommended.

NOTE: The main thing to keep in mind here is that if you do the include from a Python component, things work as you would expect, but if you call it from Python code that isn’t a component (or an include), the included component runs in the namespace of the currently running component, not the namespace of the Python module you call it from.

4.1.1 Python component files (.pyinc and .pycomp)

SkunkWeb also allows you to write includable components in pure Python code. This is sometimes more convenient when you wish to do a lot of programming and generate no (or little) HTML output for the user. These files are simple Python scripts and end in the file extensions .pyinc or .pycomp. Use them in the <:include:> or <:component:> (see section 4.2, page 22) tags, respectively, just as you would include .inc or regular .comp component files. See Python Components, section 9, page 43)

4.2 <:component:>

<:component name [*named_args*] [cache=no|yes|defer|force|old] [__args__]:>

This tag calls an STML component much as you would call a function in Python, and displays any resulting output of the component. It is *not* like including the STML component in your document. A component called with this tag does not get access to your document’s local variables; you must pass any variables explicitly as named attributes in the <:component:> tag.

Since the component being called does not receive access to the calling document's local variables, it also does not have any access to the `CONNECTION` object, as included STML documents do. The only variables a called STML component has are the ones passed as attributes of the `<:component:>` tag, and any variables which the component creates for itself.

```
<:component foo.comp this='56.7' that='[5,6,7]':>
```

executes the component `foo.comp` in the current directory, passing it the variables `this` and `that`. When `foo.comp` executes, it sees `this` and `that` as local variables, with the integer 56.7 and the list `[5,6,7]` assigned to them.

The `__args__` argument is a way to be able to pass a dictionary of arguments to a component programatically. For example,

```
<:component foo.comp __args__='{'this':56.7, 'that': [5,6,7]}':>
```

is exactly equivalent to the previous example.

And as usual for STML tags, the `__args__` could be any expression which results in either a dictionary or `None`.

Arguments specified via `__args__` are overridden by values explicitly specified in the tag. For example,

```
<:component foo.comp a="3" __args__='{'a':"5", 'b': 5}':>
```

The `foo.comp` would get `"3"` as the value of the variable `a`.

One thing of note, specifying `'cache'` as keys in the dictionary argument to `__args__` does **NOT** affect either the deferral or caching semantics of the component, they get passed through as arguments to the component.

A way to call components from Python code is (see 9, page 43 for details):

```
import AE.Component
print AE.Component.callComponent(name, argdict,
cache = whatever)
```

The `whatever` values are from the `AE.Component` module constants of the following:

`NO` do not use the cache

`YES` analogous to `cache=yes` in the `<:component:>` tag

`DEFER` analogous to `cache=defer` in the `<:component:>` tag

`FORCE` analogous to `cache=force` in the `<:component:>` tag

`OLD` analogous to `cache=old` in the `<:component:>` tag

4.2.1 Python component files (`.pycomp`)

SkunkWeb also allows you to write components in pure Python code. This is sometimes more convenient when you wish to do a lot of programming and generate no HTML output for the user. These files are simple Python scripts and end in the file extension `.pycomp`. Use them in the `<:component:>` or `<:include:>` (see section 4.1, page 21) tags just as you would regular `.comp` component files.

4.2.2 Making components cacheable

There is a special attribute for the `<:component:>` tag, called `cache`. If you set cache to the string “yes” or “true”, the called component will have its output cached to the SkunkWeb server’s disk. Setting it to the string “no” or “false”, or not setting it at all, turns caching off.

The next time you call the same component with the same attribute names and values (that is, the same arguments), the SkunkWeb server won’t bother to execute the component. Instead, it will just read the cached output from disk. It’s a great speed increase for your application.

If you call the component with different attributes names and values, the SkunkWeb server creates a *different* cached output file for those new attributes. If you say:

```
<:component foo.comp cache=yes arg="hi":>
```

and then later

```
<:component foo.comp cache=yes arg="blah" arg2='5':>
```

then SkunkWeb will keep two different cached output files for `foo.comp`: one file for when it is called with `arg="hi"`, and one file for when it’s called with `arg="blah"` and `arg2='5'`. Get it?

As to setting when a cached component will expire, (see section 4.5, page 27).

4.2.3 “Deferred” component rendering

The `<:component:>` and `<:datacomp:>` also accept as an argument value to the `cache` argument the value `"defer"`. If this has been specified, then SkunkWeb will handle the caching of the component in a graceful, and often more efficient, way.

When deferral is activated, SkunkWeb will look in the cache to see if there is output cached for the component under the set of arguments you passed. If it finds cached output, SkunkWeb uses it and does not execute the component, just as if it weren’t specified. If SkunkWeb does not find any cached output, it executes the component and caches the result, just as if deferral was not requested. However, if SkunkWeb finds cached output, but it is *expired*, SkunkWeb will use the cached output anyway, and “defer” the execution of the component until *after* all of the output has been sent back to the client in the response. Then, after sending all of the output back to the user, SkunkWeb executes the component and caches its output.

This gives you a great performance gain: if any component’s cached output expires, no individual user will have to wait for the component to be executed before receiving her response. The component’s cache will be updated offline, behind the scenes.

You can make all of your component calls automatically “deferred” by setting the configuration variable `DeferByDefault` to 1.

4.2.4 “Forced” component rendering

Like deferred component rendering, there are times when you want to render (ignore any available cached version) and cache a component regardless of whether or not it has expired. In which case, specify the value `"force"` to the `cache` attribute in the tag.

4.2.5 "Old" Component Rendering

Almost the complete opposite of forced component rendering. In this case, you're effectively saying, "gimme whatever is in the cache, I don't care how old it is." The only time when component rendering will actually occur when `old` is specified is if there is **NO** cached version at all.

4.2.6 Restrictions on arguments to cacheable components

If you want to call a component with `cache=yes` (or `cache=defer` or `cache=force`), there is one restriction you must obey:

- *The attributes passed to the cached component must be, as Python says, hashable.*

The reason for this restriction is that the SkunkWeb server needs to take all of the attributes you pass and flatten them into a short, unique value. The SkunkWeb server uses this value to look up the cached output it has stored in its disk cache.

So, what does hashable mean? Well, just consider it to mean "able to be flattened" until you learn more Python. Here's a rule of thumb:

- Numbers, strings, dates, and the `None` object are hashable
- Tuples (read-only sequences) are hashable if all of its items are hashable
- Lists (changeable sequences) are not hashable, but SkunkWeb makes an exception and lets you use them as arguments to cacheable components, as long as the list's items are either hashable or are lists or dictionaries. However, *you must be very careful* that the called component does not try to change the list in any way; if it does, you will find that the list changes only the first time the component is called, becuase on subequent calls, the output is cached and the component will not be executed.
- Dictionaries are not hashable, but SkunkWeb makes an exception for them just as it does for lists, with the same cautions and restrictions.
- Any combination of acceptable objects is acceptable: a tuple of lists of dates, etc.
- `DateTime` objects are not natively hashable, but SkunkWeb has a provision in the cache key generation code to make it as if they were hashable.
- Instances of Python classes which you write yourself are not acceptable, unless you implement a `__hash__` or `__cachekey__` method in the class. Consult the Skunk mailing list if you want to make instances of your own classes acceptable arguments to cacheable components.

4.3 `<:datacomp:>` and `<:return:>`

`<:datacomp var name [`*named_args*`] [cache=no] [defer=no] [__args__]:>`

Data components are like regular STML components, with these differences:

- Data component file end with the file extension `.dcmp` (or `.pydcmp`) instead of `.comp` (or `.pycomp`).
- Any output of a data component is ignored. Instead, the data component must return a Python object with the `<:return:>` tag.

Think of a data component as a Python function. You call it with attributes instead of arguments, you write it in STML instead of straight Python, but it returns a Python object just like a Python function.

The `<:datacomp:>` tag is used like the `<:component:>` (see section 4.2, page 22) tag, except for the first argument, `var`. `var` must be a string which names the local variable that will hold the return value of the data component:

```
<:datacomp returnObj foo.dcmp arg1="foo":>
```

In the above tag, the variable named `returnObj` will hold the object returned by data component `foo.dcmp`.

The `<:return:>` tag inside the data component tells the component to stop executing and return immediately the result of the Python expression in the tag. Examples:

```
<:return 'someVar':>
<:return '5':>
<:return '{'hey': 'dude', 'i': 'made', 'a': 'dictionary'}':>
```

If you forget to put a return tag in your data component, the object returned is the `None` object, just like a Python function that forgets to return something.

The `__args__` argument works exactly like its `<:component:>` counterpart.

Like calling regular components from Python code, a way to get the value of a datacomponent is:

```
import AE.Component
val = AE.Component.callComponent(name, argdict,
        cache = whatever, compType=AE.Component.DT_DATA)
```

See section 4.2, page 23 for valid values of `whatever`.

If you are in a Python data component and want to return the value of a variable `foo`, you'd do this:

```
raise ReturnValue, foo
```

4.3.1 Cacheable data components

As with regular components, you may call data components with `cache=yes` (or `cache=defer` or `cache=force` or `cache=old`). Data components use the `<:cache:>` (see section 4.5, page 27) tag to specify how long to cache, just like components do.

Arguments to cached data components must obey the same restrictions as arguments to cached regular components (see section 4.2.6, page 25).

Cached data components must also obey another restriction: the object returned by a data component must be *pickleable*. That's Python's way of saying serializable. If you don't know much about Python's `pickle` module, don't worry. The rule of thumb is: practically any kind of object you work with is pickleable. All Python basic data types can be pickled, as can instances of any classes you design. Therefore there is truly very little restriction on the kinds of objects you can return from a data component.

4.4 `<:type:>`

`<:type` *named_args*`:>`

The <:type:> tag allows you to to run-time type checking on variables. Only keyword arguments are allowed. The <:type:> tag may appear anywhere in an STML document. The argument name is the name of the variable that you wish to check. The value is a Python expression that is either a type, a class, or a tuple of types and/or classes.

The named variables are then checked (in no particular order) to see if they pass an `isinstance` check with the value, or in the case where the value is a tuple, check to see if `isinstance` succeeds on any of the types/classes in the tuple. If there is no match, a `TypeError` will be raised.

Passing values other than classes or types is erroneous and will raise a `TypeError` at run-time. If a named variable doesn't exist, a `NameError` will be raised.

Some examples:

```
<:type x='IntType' y='StringType':>
```

Check that `x` is an integer and that `y` is a string.

```
<:type foo='(IntType, MyClass)':>
```

Checks that `foo` is either an integer, or an instance of `MyClass`.

```
<:type foo='y.__class__':>
```

Checks to see that `foo` is an instance of the same class as `y`.

4.5 <:cache:>

```
<:cache [ until | duration ]:>
```

This tag, located in STML components and data components (it's not allowed in non-components), tells SkunkWeb how long, or until when, to cache the component's output (or data component's return object) if it is called with `cache=yes`.

This tag accepts one argument, either `until` or `duration`. You must pass this argument by name: `<:cache until=foo:>` or `<:cache duration=foo:>`.

If you do not use either argument, the component will cache for around 30 seconds by default. The default cache length is configurable in your SkunkWeb configuration file using the `defaultExpiryDuration` configuration variable.

4.5.1 Using the `duration` argument

If you specify `duration`, the value should be a string specifying the length of time to cache. It's a string with no whitespace, and a sequence of integer-letter pairs specifying untis of time:

`integer` `d`	Cache for *integer* number of days.
`integer` `h`	Cache for *integer* number of hours.
`integer` `m`	Cache for *integer* number of minutes.
`integer` `s`	Cache for *integer* number of seconds.

You may use any combination of these integer-letter pairs, in any order; SkunkWeb will add them all up and cache for the sum period of time:

```
# cache for 10 minutes
<:cache duration=10m:>
# cache for 10 hours, 10 minutes
<:cache duration=10h10m:>
# cache for 2 days, 40 seconds
<:cache duration=40s2d:>
# cache for 60 minutes (silly)
<:cache duration=40m20m:>
```

The Python equivalent of the `<:cache duration=`*`foo`*`:>` tag is:

```
from Date import TimeUtil
__expiration = TimeUtil.convertDuration(foo)
```

4.5.2 Using the `until` argument

If you specify `until`, the value can be one of many things, all intended to indicate a point in time until which to cache. The `until` argument accepts any one of the following:

- A string of the form `"hh:mi"` or `"hh:mi:ss"`, specifying a time in hours, minutes, and optionally seconds, on a 24-hour clock. This tells SkunkWeb to cache until the next occurrence of that time. If you say `until=06:00`, and SkunkWeb executes the component at 05:30, it will cache for 30 minutes; but if SkunkWeb executes the component at 07:00, it will cache for 23 hours until the next day at 06:00.
- A string of the form `":mi"`, meaning “minutes after the hour”. This tells SkunkWeb to cache until that many minutes after the current hour, or after the next hour if those minutes have already passed.
- A `DateTime` object, as produced by functions in the `Date` or `DateTime` modules, or retrieved from a database using skunk.org’s `SQL` module or SQL personality of SkunkWeb.
- A string of the form `"yyyy-mm-dd"`, `"yyyy-mm-dd hh:mi"`, or `"yyyy-mm-dd hh:mi:ss"`. These strings specify dates according to the ISO format supported by the `Date` module.
- A string of the form `"mm/dd/yyyy"`, `"mm/dd/yyyy hh:mi"`, or `"mm/dd/yyyy hh:mi:ss"`. These strings specify dates according to a common U.S. date format. These string forms are only supported for backward compatibility; if you must use a string specifying an explicit date, use the ISO format above.
- A `RelativeDateTime` object from the `DateTime` module. these objects allow you to specify very complex relative dates, such as “the first Sunday of next month at noon”. Be careful with these objects: unless you read the documentation fully, you can end up constructing a `RelativeDateTime` that moves you *backwards* in time. If you do that, the `<:cache:>` will raise an exception telling you of your mistake.
- A tuple or list containing ANY combination of the above items. In this case, SkunkWeb will evaluate all items, and choose the result that is nearest the current date/time for the cache expiration. This means you can do neat complex cache control:

```
# expires the cache at 0, 15, 30, and 45 minutes after the hour
<:cache until='(":00", ":15", ":30", ":45")':>

# expires the cache at 15 minutes after the hour,
# and also at 06:00 every day, and also
# at 5pm (17:00) on Sundays.
<:import DateTime "RelativeDateTime":>
<:cache until='(":15", "06:00",
   RelativeDateTime(
        weekday=0, hour=17,
        minute=0, second=0))':>
```

The Python equivalent of the `<:cache until=`*`foo`*`:>` tag is:

```
import TimeUtil
__expiration = TimeUtil.convertUntil(foo)
```

4.6 <:import:>

```
<:import name [items] [as]:>
```

Imports a Python code module into the STML template. This is practically identical to the `import` statement in Python.

If the `items` parameter is specified, it should be either `'*'` or, a space delimited list of names in the module to copy into the STML template's namespace. If items is '*', all names in the module (that don't begin with '_') are copied into the STML template's namespace.

If the `as` parameter is specified, it acts just like the `as` clause to the Python `import` statement. Some examples, with their equivalents in straight Python:

```
<:import string:>  import string

<:import string find:>  from string import find

<:import string "find replace join":>  from string import find, replace, join

<:import string *:>  from string import *

<:import string find string_find:>  from string import find as string_find
```

Note on where modules come from: When you do an `<:import foo:>` in STML, the SkunkWeb server first looks for the module `foo` in Python's standard module directories. If it does not find the module in any of those directories, SkunkWeb then looks in your custom SkunkWeb module library, which is usually `/usr/local/skunk/site-libs` (but could be changed by modifying `sys.path` in `sw.conf`) for a file `foo.py`. If SkunkWeb still cannot find the module `foo`, it raises an `ImportError`.

CHAPTER

FIVE

HTML Helper Tags

The STML tags in this section help you construct URLs and common HTML tags which use URLs for links, images, forms, and redirects. These tags are only enabled if using the AE library under the SkunkWeb server or under `swpython`. Chances are exceptionally fair that this is the case.

For the most part, unless you have a dynamic url scheme (e.g. your user-visible URLs are in Spanish, but English in the code, or perhaps you use Akamai for images), you probably don't need these at all, but if you do, they can be *very* useful. The behavior of these can be modified by writing a service that overrides the relavent functions in the `templating.UrlBuilder` module (under `SkunkWeb/Services`).

The default behavior (i.e. you've not overriden bits of `templating.UrlBuilder`) of these tags is very simple. Your SkunkWeb administrator or lead developer may customize the behavior of these tags to fit the needs of your website. If these STML tags behave differently or strangely in your web application, it's probably because they've been customized; ask your local SkunkWeb guru.

5.1 <:url:>

```
<:url path [queryargs] [text] [noescape] [abs] [extra_args]:>
```

Produces a URL or an HTML <A> tag. All arguments except `path` are optional.

- `queryargs` can be a dictionary of arguments which you would like added to the URL as a query string, for example, the dictionary {`'foo':5, 'bar':'yes'`} gets translated to appending `?foo=5&bar=yes` to the url. Defaults to an empty dictionary.
- `text` is an optional string. If it is present, the tag will not just produce a URL, but a full HTML <A> tag with an HREF attribute holding the URL and the text in between.
- The `noescape` argument, if present, tells SkunkWeb *not* to escape URL-unsafe characters in the `path` argument, such as spaces or extended characters. *By default, SkunkWeb will escape all non-safe characters except for the / character, which it leaves intact. It will escape the `:` character in your `path` argument, however.*
- The `abs` argument, if present and true, will cause the url to be made absolute. At the moment, it is only legal to use this argument if the path argument begins with `/`. For instance:

  ```
  <:url /index.html abs=1:>
  # produces, when served on localhost:8080:
  http://localhost:8080/index.html
  ```

  `abs` is new in SkunkWeb 3.3.1.

- `extra_args` is not an argument name: it just means that any other arguments you pass by name will be put in the `<A>` tag if `text` is not empty. Example:

```
<:url /index.html text="Home Page" target="_blank" :>
# produces
<a href="/index.html" target="_blank">Home Page<a>
```

The equivalent Python component code would be:

```
import templating.UrlBuilder
print templating.UrlBuilder.url(path, queryargs, text,
     extra_args_as_dict, noescape)
```

5.2 <:img:>

```
<:img path [queryargs] [noescape] [extra_args]:>
```

Creates an appropriate HTML `<IMG>` tag. Arguments similar to `<:url:>` (see section 5.1, page 31).

The `queryargs` argument, if present, allows you to pass a dictionary of arguments as a querystring to the image document, as is useful for dynamically generated images (thumbnails, graphs, etc.). Defaults to an empty dictionary.

The `noescape` argument, if present, tells SkunkWeb *not* to escape URL-unsafe characters in the `path` argument, such as spaces or extended characters. *By default, SkunkWeb will escape all non-safe characters except for the `/` character, which it leaves intact. It will escape the `:` character in your `path` argument, however.*

The `extra_args` parameter is not actually a real argument name, but is a pass through to let you pass through arguments to the `<img>` tag. Keyword arguments only.

Examples:

```
<:img path=this.gif border=1:>
# produces <IMG SRC="this.gif" border="1">
```

The equivalent Python component code would be:

```
import templating.UrlBuilder
print templating.UrlBuilder.image(path, queryargs,
     extra_args_as_dict, noescape)
```

5.3 <:form:>

`<:form path [noescape] [`*`extra_args`*`]:>`

Creates an appropriate HTML `<FORM>` start tag. Arguments are similar to the `<:url:>` (see section 5.1, page 31). The `extra_args` parameter is not actually a real argument name, but is a pass through to let you pass through arguments to the `<FORM>` tag such as `METHOD`. These arguments must be passed by name only.

IMPORTANT NOTE! This tag only generates the start HTML form tag, but *not* the closing `</FORM>` tag. Always remember to close your HTML forms with the raw HTML tag `</form>`.

The equivalent Python component code would be:

```
import templating.UrlBuilder
print templating.UrlBuilder.form(path,
     extra_args_as_dict, noescape)
```

5.4 <:hidden:>

`<:hidden` *named_args*`:>`

Creates a series of HTML hidden input fields for each named attribute in the tag, escaping html entities as appropriate.

Example:

```
<:set that "Hello//world":>
<:hidden this=‘that‘ count=‘5‘:>
```

produces

```
<INPUT TYPE=HIDDEN NAME="this" VALUE="Hello%2F%2Fworld">
<INPUT TYPE=HIDDEN NAME="count" VALUE="5">
```

The Python component code to mimic this tag is:

```
import templating.UrlBuilder
print templating.UrlBuilder( named_args_as_dict )
```

5.5 <:redirect:>

```
<:redirect url:>}
<:redirect path [queryargs] [noescape]:>
```

If called with the named argument `url`, the argument should contain an absolute URL. Example:

```
<:redirect url="http://www.starmedia.com/":>
```

Alternately, you may call this tag with arguments similar to those for `<:url:>` (see section 5.1, page 31). `path` is the path, `queryargs` an optional dictionary of arguments to put in the query string,

The `noescape` argument, if present, tells SkunkWeb *not* to escape URL-unsafe characters in the `path` argument, such as spaces or extended characters. *By default, SkunkWeb will escape all non-safe characters except for the / character, which it leaves intact. It will escape the `:` character in your `path` argument, however.*

Please note that either `url` or `path` must be passed by name, like `url=foo` or `path=foo`.

The equivalent Python component code is:

```
import templating.UrlBuilder
templating.UrlBuilder.redirect(path, url, queryargs, noescape)
```

CHAPTER

SIX

Internet Services

Originally, STML planned to have many tags which performed operations on common Internet services. The skunk.org team soon realized, however, that it was a better approach to use Python's standard modules for Internet services. If you need to perform some operation, such as fetching a web page from a server, or talking to a UseNet newsgroup, or fetching mail from a POP mail server, write your own wrapper module for use in SkunkWeb, and make use of Python's fantastic standard modules, such as `httplib` and `urllib`.

STML gives you only one tag for Internet services: a tag for sending electronic mail, since that is a common task performed by web applications.

6.1 <:sendmail:>

`<:sendmail to_addrs subject msg [from_addr] [extended]:>`

The `to_addrs` parameter is a list of strings (or a single string) containing the recipients email address(es). The `subject` argument is a string containing the subject line. The `msg` argument contains the entire message as a single string. It's possible to have a message with or without headers in the msg parameter. The `from_addr` argument is an optional string with the email address of the sender, and it defaults to the address in the `FromAddress` configuration variable. Beware that this address must be valid and reachable from the Internet (or within your organization in case of an intranet application). The address is also used as envelope sender except if one is given in `extended`. `extended` is one or many additional keyword arguments that are described later.

To do the equivalent in Python components:

```
import templating.MailServices
templating.MailServices.sendmail (
    to_addrs, subject, msg, from_addr [, extended] )
```

The `sendmail` function (and the equivalent tag) will raise a `templating.MailServices.MailError` on any mail failure.

The From: and To: headers are always built from supplied parameters. If they are present in `msg` they are overwritten. Cc: and Bcc: headers in the supplied mail text are not touched and not used.

The sendmail tag adds a correct Date: header, ensures that the From: header is set and generates a Message-Id: header if necessary. If the raw parameter (in `extended`) is true, these headers are not touched.

The additional keyword arguments (`extended`) may be the following:

`to_name` a string that is used in the To: header directly. It is merged with to_addrs if the there is only one value for to_addrs. If there are multiple to_addrs, it is used literally and may then contain an address too. Because headers are only allowed to contain 7bit data a given character set is applied to this parameter. At the moment

only ASCII and iso-8859-1 are supported.

`from_name` A string that is merged with from_addr in the From: header. This header also has to contain only 7bit data, so a given character set is used for it. The same restriction as with to_name applies.

`envelope_sender` The sender address that is used in the SMTP dialogue. This address gets possible bounces (returned mails that couldn't be delivered). The parameter is a string.

`charset` A string that describes the character set of the message itself and some headers (Subject:, From:, To:). At the moment only `iso-8859-1` and `us-ascii` are supported because of limitations in the rfc822 module. The default is `us-ascii`.

`encoding` A string that defines the encoding method for the message body. The encoding is applied if given, so don't specify one if your message is already encoded. Possible values are `base64`, `7bit`, `8bit` and `quoted-printable`.
When 7bit is used, the MIME headers (Mime-Version, Content-Encoding and Content-Type are not set (and deleted if present in `msg`). The default value is `7bit`.

`raw` This parameter is interpreted as True or False. If this parameter is set to "True" the content of the message is not altered in any way. In this case the `envelope_sender` has to be set, the receivers are taken from `to_addrs` as usual.

CHAPTER

SEVEN

Text Messages with Message Catalogs

In many web applications, you will find that you need to display text messages to the web user, such as error messages, alert messages, success messages, and so forth. Yet you do not wish to type these messages directly into your STML templates; what if you use them in many places on your Web site, and then you have to change the text of a message? You do not want to have to hand-edit each STML document to make the change.

STML provides a nice way to manage text messages and avoid hard-coding them into your STML documents: **message catalogs**. An STML message catalog acts somewhat like a Python dictionary: you give it a *key*, which is a string, and the mesage catalog looks up the message for that key. Unlike a dictionary, an STML message catalog does not raise an error if it cannot find a message under the key you request; instead, the message catalog returns an empty string `""`.

Unlike a dictionary, a message catalog is not an object which you work with directly. Instead, you use the `<:msg:>` tag to access messages in the catalog.

There are two kinds of message catalogs in STML:

Regular, or "simple", catalogs Dictionary-like message catalogs. Each message "name" is like a dictionary key, with a single message stored for each key.

Multi-lingual, or "complex", catalogs: Like a simple catalog, except that each message key stores multiple messages, one message for each language in the catalog. (Currently supported languages are `'esp'` (Spanish), `'por'` (Portuguese), and `'eng'` (English).

The following sections explain the three tags which provide message catalog features.

7.1 `<:catalog:>` for message catalogs

```
<:catalog catname name:>
```

Loads a message catalog named `catname` into the local variable `name`. Once the catalog is loaded, you may use the `<:msg:>` (see section 7.2, page 38) tag to access messages in the catalog.

The catalog name in `catname` is of the path to the message catalog (simple message catalogs have the extension `.msg`, complex, or multi-lingual ones have an `.cat` extension) , just like names for includes, components, and data components.

The equivalent Python code is:

```
import AE.Cache
name = AE.Cache.getMessageCatalog ( catname )
```

7.2 <:msg:> for retrieving messages

<:msg *catname msgname* [lang] [fmt] [substitution_args]:>

This tag accesses a message in the catalog object previously loaded into the local variable catname with a <:catalog:> tag. If the message does not exist, the message is an empty string. The returned message is then displayed in the template output, as in a <:val:> (see section 2.1, page 5) tag.

For example, you load a simple message catalog file foo/bar/mycatalog.msg with the following tag:

```
<:catalog foo/bar/mycatalog.msg thisCat:>
```

The catalog object is now loaded into the local variable thisCat. Then, to access the message under the name welcome_user, you use the tag:

```
<:msg thisCat.welcome_user:>
```

If the catalog object is a multi-lingual message catalog, you may pass the optional argument lang. If you do not specify lang, the SkunkWeb server will look for a global variable in your template called lang, and use that. You can configure your SkunkWeb server to set this global variable automatically. If SkunkWeb cannot find this global variable, it will raise an error.

For example, you load a multi-lingual message catalog file foo/bar/mycatalog.cat with the following tag:

```
<:catalog foo/bar/mycatalog.cat thisCat:>
```

The catalog object is now loaded into the local variable thisCat. Then, to access the message under the name welcome_user, for Portuguese, you use the tag:

```
<:msg thisCat.welcome_user lang=por:>
```

Alternately, if you know that the global variable lang is set properly, you can leave out the lang argument.

fmt is an optional argument, and works just like the fmt argument in the <:val:> (see section 2.1, page 5) tag.

The substitution_args marker is not an argument, but instead means that any named arguments you specify will be used to do "variable substitution" if the message contains placeholders for variable substitution. See the variable subtitution (see section 7.3, page 38) section for details.

The equivalent Python code is:

```
import AE.MsgCatalog
import DT.DTCommon
print AE.MsgCatalog.getMessage (
    catalog, msgname, lang, DT.DTCommon.ValFmtRgy[ fmt ],
    substitution_args_as_dict )
```

7.3 Variable Subsitution in Messages

You can, if you wish, put "placeholders" in messages and then, in the <:msg:> (see section 7.2, page 38) tag, subtitute the values of variables for those placeholders.

To make a placeholder, choose the name of the variable, and then insert that variable between double brackets in the message, for example:

```
"This is a message with a variable substitution [[thisVar]]."
```

Then, when using this message with the `<:msg:>` (see section 7.2, page 38) tag, pass a named argument as a tag attribute:

```
<:msg thisCatalog.thisMessage thisVar="here":>
```

which will produce the output:

```
This is a message with a variable substitution here.
```

7.4 Editing Message Catalogs

Creating message catalog files is a snap. You can use skunk.org's SkunkDAV software, which has a built-in message catalog editor. Or you can create an edit these files directly using your favorite text editor.

Message catalogs are just like Python dictionaries. And in message catalog files, the message names and values are expressed exactly as a big Python dictionary. For example, the contents of the simple message catalog `my_messages.msg`:

```
{
    'welcome': 'Welcome to Skunk.',
    'unauthorized': 'Go away!'
}
```

As you can see, the file's contents are a Python dictionary. The keys of the dictionary are the message names (always strings), and the values are the messages themselves (always strings).

Multi-lingual message catalogs are two-level Python dictionaries. The language is the top level, and each language key has a full dictionary of message names and values. Here's the above simple message catalog as a multilingual catalog in the file `my_messages.cat`:

```
{
    'eng': {
        'welcome': 'Welcome to Skunk.',
        'unauthorized': 'Go away!'
    },
    'esp': {
        'welcome': 'Bienvenido a Skunk.',
        'unauthorized': 'Que se vaya...'
    }
}
```

When you edit message catalog files directly, you may type anything that is legal in a Python dictionary literal: any whitespace or indenting you wish, even comments. Please note, however, that editing message catalogs in the

SkunkDAV editor results in all of your comments being lost; SkunkDAV reformats the dictionary whenever it saves the message catalog file.

CHAPTER

EIGHT

Documenting Your STML Code

8.1 <:doc:>

```
<:doc:>
    [ template, component, or datacomp documentation ]
<:/doc:>
```

This tag is a great tool for your development projects! At the beginning of any STML document, place a single <:doc:> tag. Inside the tag, write text documentation about the document (any STML tags will be ignored). Describe the role of the document in your application. Also describe the user arguments it looks for (if it's a template) or the arguments it expects (if it's a component or data component). If your STML document raises exceptions with the <:raise:> tag, describe which exceptions it can raise. For data components, also describe what kinds of objects are returned.

The **skunkdoc** tool, which comes with the SkunkWeb software, can then extract all of your STML documents' <:doc:> contents, and construct a surfable, cross-referenced HTML documentation set for your website. (It also will include all of your custom SkunkWeb Python modules and their documentation.)

Use the <:doc:> tag; you will be forever glad you did.

Note that this tag considers all of its contents to be text; STML tags are not parsed. The resulting documentation will be in plain text, unless you use the magic **skunkdoc** XML format: make the first two non-whitespace characters in the tag ** and the skunkdoc tool will parse them as XML. See the non-existant manual "Documentation Tools" on the skunk.org website to learn all about the skunkdoc XML format.

8.2 <:comment:> or <:#:>

```
<:comment:>
    [ stuff you want commented out ]
<:/comment:>
```

Similar to the HTML <!-- --> comment. Everything until the closing </comment> tag is ignored by the SkunkWeb server, and will neither be executed nor show up in the output. Comments do not nest, and whatever occurs inside a <:comment:> tag must be legal STML.

As a shortcut, this tag can also be written <:#:><:/#:>.

8.3 <:* ..... *:>

```
<:*
   [ stuff you want commented out ]
*:>
```

This form of comment is different than the previous kind. The previous kind is good when you want to either disable the contained code/text, but stinks when the contained tag structure isn't syntactically correct. The contents of this form of comment are *totally* ignored and contain anything, except a `*:>`.

CHAPTER

NINE

Python Components

Python components (we will use the term “component” here to mean regular components, data components or top-level documents) are a way to write components and/or data components using straight Python. They are useful for when the operation of the component is code-heavy and/or not terribly markup-intensive.

To call other components from within Python components, there is a function in the `AE.Component` module called, aptly enough, `callComponent` whose definition looks like this:

```
def callComponent (name, argDict, cache = 0,
                   defer = None, compType = DT_REGULAR)
```

The arguments are:

`name` the path to the component (can be relative)

`argDict` dictionary of arguments to the component

`cache` equivalent to the `cache` parameter in the component tags, but see section 4.2, page 23 for actual values to use.

`compType` either the default (a regular component), `AE.Component.DT_REGULAR`, `AE.Component.DT_DATA` for a data component or `AE.Component.DT_INCLUDE`

The return value is either the output of the component, in the case of a regular component, or the return value of the component, in the case of a data component.

9.1 Top-Level Python Documents

Top-level Python documents *are* url accessible with the file extension of `.py`. They recieve the `CONNECTION` object in their global namespace and standard output goes to the browser (for the body text, if you wish to set response headers, they must be set using the `CONNECTION` object). For example, the following is a valid top-level python script:

```
print "The current URI is: <B>%s</B>" % CONNECTION.uri
```

9.2 Python Include Components

The Python include component extension is `.pyinc`. Python include components have similar semantics as their STML counterparts. All of the variable accessible in the calling documents namespace will appear in the global namespace of the Python component being called, and all changes (additions, changes, deletions) to those variables will appear in the calling document.

Producing output works the same as in top-level python components.

9.3 Regular Python Components

Regular Python components get the default global namespace (as would an STML component) and the component arguments in the global namespace. The catch is that there is no `<:cache:>` tag in Python, unfortunately, but this can be done by setting the global variable named `__expiration` to the number of seconds past the Epoch (12:00am 1/1/1970 UTC) at which this component will expire. This may sound nasty, but of course there is a simple way to compute this value! In the `Date.TimeUtil` module) there are two functions which mimic STML's `<:cache:>` tag, `convertDuration` and `convertUntil`.

The `convertDuration` function takes the same argument as the `duration` argument to the `<:cache:>` tag as a string. So to say your component should cache for five minutes, you'd say:

```
import Date.TimeUtil
__expiration = Date.TimeUtil.convertDuration("5m")
```

The `convertUntil` function takes the same argument as the `until` argument to the `<:cache:>` tag.

Producing output works the same as in top-level python components.

9.4 Python Data Components

Python data components extension is `.pydcmp`. Python data components are similar to regular Python components as STML data components are similar to regular STML components. Unfortunately, there is no `<:return:>` tag in Python, nor can the Python `return` statement be used (since it would be at the top level, where it is a syntax error). So, in order to return a value from the data component, you raise an exception of the proper type, `ReturnValue`, with the exception value being the return value. The ReturnValue exception is preloaded into the global namespace in which the component is executed, so it doesn't need to be imported.

For example, to return the value of the variable foobar from a data component:

```
raise ReturnValue, foobar
```

The neat thing about this method of returning the value is that you can raise the exception from anywhere in the data component, i.e. from within functions, classes, wherever.

Raising other exceptions will, of course, invoke the normal error handling routines. Raising no exception returns the value `None`

Setting the caching parameters of a data component is identical to that of regular Python components.

As with normal STML data components, any generated output is ignored.

CHAPTER

TEN

Namespaces

In the global namespace of the top-level component there are, by default, only five things, four of which you can safely ignore, leaving the `CONNECTION` object. For non top-level components, you either get what was passed or inherited (in the case of include), and that's it.

10.1 The `CONNECTION` Object

The `CONNECTION` object contains (duh) information that pertains to the current web request/response (i.e. connection).

`env` The CGI-like environment dictionary.

`browser` A `Browser` object that contains...

`method` The request method (same as `env['REQUEST_METHOD']`).

`host` The hostname sent in the request headers.

`args` The query arguments (either for GET or POST).

`requestHeaders` A dictionary of request headers.

`responseHeaders` A dictionary of headers to send in the response.

`requestCookie` A `Cookie`[1] object corresponding to the cookie sent by the client.

`responseCookie` A `Cookie` object representing the cookie to be sent to the client.

`uri` The URI of the request. This URI may have been subject to rewriting[2] and may *not* be the URI as the client sent it.

`realUri` The URI of the request. This URI has not been subject to the same rewriting as `uri`. This is the URI as sent by the client.

`extract_args` See (see section 2.5, page 8).

`setContentType` A method taking a single string argument that is a shortcut to set the `Content-Type` header.

`setStatus` A method taking a single numeric status value for the HTTP response.

`redirect` Takes a single URL argument which is the URL to redirect to.

[1] see the documentation for the `Cookie` module in the Python Library Reference

[2] Either via the rewrite service or other means