Smarty3 Manual

Monte Ohrt <monte at ohrt dot com>
Uwe Tews <uwe dot tews at googlemail dot com>

Smarty - the compiling PHP template engine

by Monte Ohrt <monte at ohrt dot com> and Uwe Tews <uwe dot tews at googlemail dot com>

Publication date 2010-11-11

Table of Contents

List of Examples

Preface

The Philosophy

The Smarty design was largely driven by these goals:

- clean separation of presentation from application code
- PHP backend, Smarty template frontend
- compliment PHP, not replace it
- fast development/deployment for programmers and designers
- quick and easy to maintain
- syntax easy to understand, no PHP knowledge necessary
- flexibility for custom development
- security: insulation from PHP
- free, open source

What is Smarty?

Smarty is PHP a framework for separating presentation (HTML/CSS) from application logic (PHP). This implies that PHP code is application logic, and is separated from the presentation.

Two camps of thought

When it comes to templating in PHP, there are basically two camps of thought. The first camp exclaims that "PHP is a template engine". Simply because PHP *can* mix with presentation doesn't mean it's a preferable choice. The only virtue to this approach is speed from a pure script-execution point of view. However, the PHP syntax is quite ugly and doesn't mix well with tagged markup such as HTML.

The second camp exclaims that presentation should be void of all programming code, and instead use simple tags to indicate where application content is revealed. This approach is common with other development languages, and is also the approach that Smarty takes. The idea is to keep the templates focused squarely on presentation, not application code, and with as little overhead as possible.

Why is separating PHP from templates important?

Two major benefits:

- SYNTAX: Templates typically consist of semantic markup such as HTML. PHP syntax works well for application code, but quickly degenerates when mixed with HTML. Smarty's simple {tag} syntax is designed specifically to express presentation. Smarty focuses your templates on presentation and less on "code". This lends to quicker template deployment and easier maintenance. Smarty syntax requires no working knowledge of PHP, and is intuitive for programmers and non-programmers alike.
- INSULATION: When PHP is mixed with templates, there are no restrictions on what type of logic can be injected into a template. Smarty insulates the templates from PHP, creating a controlled separation of presentation from business logic. Smarty also has security features that can further enforce restrictions on templates.

Web designers and PHP

A common question: "Web designers have to learn a syntax anyways, why not PHP?". Of course web designers can learn PHP, but this isn't about their ability to learn it. They learn PHP, then start stuffing things into templates that don't belong there (you just handed them a swiss-army knife when they just needed a knife.) You can teach them the rules of application development, but then you may as well call them developers. Smarty gives web designers exactly the tools they need, and gives developers fine-grained control over those tools.

Implementation is Important

A common problem with Smarty development is poor implementation. If you abuse Smarty fundamentals or come across an application that does (i.e. injecting PHP in templates), you may end up with something worse than what Smarty was intended to resolve. See the Best Practices section on the Smarty website for examples of what to look out for.

How does it work?

Under the hood, Smarty "compiles" (basically copies and converts) the templates into PHP scripts. This happens once when each template is first invoked, and then the compiled versions are used from that point forward. Smarty takes care of this for you, so the template designer just edits the Smarty templates and never has to manage the compiled versions. This approach keeps the templates easy to maintain, and yet keeps execution times extremely fast since the compiled code is just PHP. And of course, all PHP scripts take advantage of PHP op-code caches such as APC.

Template Inheritance

Template inheritance is new to Smarty 3, and it's one of many great new features. Before template inheritance, we managed our templates in pieces such as header and footer templates. This organization lends itself to many problems that require some hoop-jumping, such as managing content within the header/footer on a per-page basis. With template inheritance, instead of including other templates we maintain our templates as single pages. We can then manipulate blocks of content within by inheriting them. This makes templates intuitive, efficient and easy to manage. See the Template Inheritance section of th Smarty website for more info.

Why not use XML/XSLT syntax?

There are a couple of good reasons. First, Smarty can be used for more than just XML/HTML based templates, such as generating emails, javascript, CSV, and PDF documents. Second, XML/XSLT syntax is even more verbose and fragile than PHP code! It is perfect for computers, but horrible for humans. Smarty is about being easy to read, understand and maintain.

Template Security

Although Smarty insulates you from PHP, you still have the option to use it in certain ways if you wish. Template security forces the restriction of PHP (and select Smarty functions.) This is useful if you have third parties editing templates, and you don't want to unleash the full power of PHP or Smarty to them.

Integration

Sometimes Smarty gets compared to Model-View-Controller (MVC) frameworks. Smarty is not an MVC, it is just the presentation layer, much like the View (V) part of an MVC. As a matter of fact, Smarty can easily be integrated as the view layer of an MVC. Many of the more popular ones have integration instructions for Smarty, or you may find some help here in the forums and documentation.

Other Template Engines

Smarty is not the only engine following the "Separate Programming Code from Presentation" philosophy. For instance, some Python developers decided that mixing Python with HTML/CSS wasn't such a great idea, and built some template engines around the same principles such as Django Templates and CheetahTemplate.

What Smarty is Not

Smarty is not meant to be a stand-alone tool for developing websites from scratch. Smarty is a tool to facilitate the presentation layer of your application. You may decide to use an existing framework with Smarty, or you might build your own codebase and use Smarty for the frontend. Either way, Smarty is a great way to decouple presentation from your application code.

Summary

Whether you are using Smarty for a small website or massive enterprise solution, it can accommodate your needs. There are numerous features that make Smarty a great choice:

- separation of PHP from HTML/CSS just makes sense
- readability for organization and management
- security for 3rd party template access
- feature completeness, and easily extendable to your own needs
- massive user base, Smarty is here to stay
- LGPL license for commercial use
- 100% free to use, open source project

Should I use Smarty?

Smarty's long track record has indicated that it significantly improves the speed of deployment and maintenance of templates. If maintenance is crucial to your application (or business), Smarty is certainly a good fit.

However, Smarty is not a magic bullet. It is a tool for managing presentation. Using Smarty, another template engine, or plain PHP is largely going to be determined by your own requirements and tastes. It is certainly possible to maintain presentation mixed with PHP if you are happy with that. You may not have a need for Smarty if you don't need cleaner template syntax, template inheritance, caching, plugins, security (i.e. insulation from PHP), quicker presentational deployment and easier maintenance. The best approach is to ask lots of questions, install and test Smarty for yourself and make an informed decision for your project.

Part I. Getting Started

Table of Contents

Chapter 1. What is Smarty?

Smarty is a template engine for PHP. More specifically, it facilitates a manageable way to separate application logic and content from its presentation. This is best described in a situation where the application programmer and the template designer play different roles, or in most cases are not the same person.

For example, let's say you are creating a web page that is displaying a newspaper article.

- The article `$headline`, `$tagline`, `$author` and `$body` are content elements, they contain no information about how they will be presented. They are passed into Smarty by the application.
- Then the template designer edits the templates and uses a combination of HTML tags and template tags to format the presentation of these variables with elements such as tables, div's, background colors, font sizes, style sheets, svg etc.
- One day the programmer needs to change the way the article content is retrieved, ie a change in application logic. This change does not affect the template designer, the content will still arrive in the template exactly the same.
- Likewise, if the template designer wants to completely redesign the templates, this would require no change to the application logic.
- Therefore, the programmer can make changes to the application logic without the need to restructure templates, and the template designer can make changes to templates without breaking application logic.

One design goal of Smarty is the separation of business logic and presentation logic.

- This means templates can certainly contain logic under the condition that it is for presentation only. Things such as including other templates, alternating table row colors, upper-casing a variable, looping over an array of data and displaying it are examples of presentation logic.
- This does not mean however that Smarty forces a separation of business and presentation logic. Smarty has no knowledge of which is which, so placing business logic in the template is your own doing.
- Also, if you desire *no* logic in your templates you certainly can do so by boiling the content down to text and variables only.

Some of Smarty's features:

- It is extremely fast.
- It is efficient since the PHP parser does the dirty work.
- No template parsing overhead, only compiles once.
- It is smart about recompiling only the template files that have changed.
- You can easily create your own custom functions and variable modifiers, so the template language is extremely extensible.
- Configurable template {delimiter} tag syntax, so you can use `{$foo}`, `{{$foo}}`, `<!--{$foo}-->`, etc.
- The `{if}..{elseif}..{else}..{/if}` constructs are passed to the PHP parser, so the `{if...}` expression syntax can be as simple or as complex an evaluation as you like.

- Allows unlimited nesting of `sections`, `if's` etc.
- Built-in caching support
- Arbitrary template sources
- Custom cache handling functions
- Template Inheritance for easy management of template content.
- Plugin architecture

Chapter 2. Installation

Requirements

Smarty requires a web server running PHP 4.0.6 or greater.

Basic Installation

Install the Smarty library files which are in the `/libs/` sub directory of the distribution. These are `.php` files that you SHOULD NOT edit. They are shared among all applications and only get changed when you upgrade to a new version of Smarty.

In the examples below the Smarty tarball has been unpacked to:

- `/usr/local/lib/Smarty-v.e.r/` for *nix machines
- and `c:\webroot\libs\Smarty-v.e.r\` for the windows environment.

Example 2.1. Required Smarty library files

```
Smarty-v.e.r/
   libs/
      Smarty.class.php
      debug.tpl
      sysplugins/* (everything)
      plugins/*    (everything)
```

Smarty uses a PHP constant [http://php.net/define] named `SMARTY_DIR` which is the **full system file path** to the Smarty `libs/` directory. Basically, if your application can find the `Smarty.class.php` file, you do not need to set the `SMARTY_DIR` as Smarty will figure it out on its own. Therefore, if `Smarty.class.php` is not in your include_path [http://php.net/ini.core.php#ini.include-path], or you do not supply an absolute path to it in your application, then you must define `SMARTY_DIR` manually. `SMARTY_DIR` **must include a trailing slash/**.

Here's how you create an instance of Smarty in your PHP scripts:

```
<?php
// NOTE: Smarty has a capital 'S'
require_once('Smarty.class.php');
$smarty = new Smarty();
?>
```

Try running the above script. If you get an error saying the `Smarty.class.php` file could not be found, you need to do one of the following:

Example 2.2. Set SMARTY_DIR constant manually

```
<?php
// *nix style (note capital 'S')
define('SMARTY_DIR', '/usr/local/lib/Smarty-v.e.r/libs/');

// windows style
define('SMARTY_DIR', 'c:/webroot/libs/Smarty-v.e.r/libs/');

// hack version example that works on both *nix and windows
// Smarty is assumend to be in 'includes/' dir under current script
define('SMARTY_DIR',str_replace("\\","/",getcwd()).'/includes/Smarty-v.e.r/libs/')

require_once(SMARTY_DIR . 'Smarty.class.php');
$smarty = new Smarty();
?>
```

Example 2.3. Supply absolute path to library file

```
<?php
// *nix style (note capital 'S')
require_once('/usr/local/lib/Smarty-v.e.r/libs/Smarty.class.php');

// windows style
require_once('c:/webroot/libs/Smarty-v.e.r/libs/Smarty.class.php');

$smarty = new Smarty();
?>
```

Example 2.4. Add the library path to the `php.ini` file

```
;;;;;;;;;;;;;;;;;;;;;;;;;
; Paths and Directories ;
;;;;;;;;;;;;;;;;;;;;;;;;;

; *nix: "/path1:/path2"
include_path = ".:/usr/share/php:/usr/local/lib/Smarty-v.e.r/libs/"

; Windows: "\path1;\path2"
include_path = ".;c:\php\includes;c:\webroot\libs\Smarty-v.e.r\libs\"
```

Example 2.5. Appending the include path in a php script with `ini_set()` [http://php.net/ini-set]

```
<?php
// *nix
ini_set('include_path', ini_get('include_path').PATH_SEPARATOR.'/usr/local/lib/Sma

// windows
ini_set('include_path', ini_get('include_path').PATH_SEPARATOR.'c:/webroot/lib/Sma
?>
```

Now that the library files are in place, it's time to setup the Smarty directories for your application:

- Smarty requires four directories which are by default named `templates/`, `templates_c/`, `configs/` and `cache/`
- Each of these are definable by the Smarty class properties `$template_dir`, `$compile_dir`, `$config_dir`, and `$cache_dir` respectively
- It is highly recommended that you setup a separate set of these directories for each application that will use Smarty
- You can verify if your system has the correct access rights for these directories with `testInstall()`.

For our installation example, we will be setting up the Smarty environment for a guest book application. We picked an application only for the purpose of a directory naming convention. You can use the same environment for any application, just replace `guestbook/` with the name of your application.

Example 2.6. What the file structure looks like

```
/usr/local/lib/Smarty-v.e.r/libs/
        Smarty.class.php
        debug.tpl
        sysplugins/*
        plugins/*

/web/www.example.com/
        guestbook/
        templates/
            index.tpl
        templates_c/
        configs/
        cache/
        htdocs/
            index.php
```

Be sure that you know the location of your web server's document root as a file path. In the following examples, the document root is `/web/www.example.com/guestbook/htdocs/`. The Smarty directories are only accessed by the Smarty library and never accessed directly by the web browser. Therefore to avoid any security concerns, it is recommended (but not mandatory) to place these directories *outside* of the web server's document root.

You will need as least one file under your document root, and that is the script accessed by the web browser. We will name our script `index.php`, and place it in a subdirectory under the document root `/htdocs/`.

Smarty will need **write access** (windows users please ignore) to the *`$compile_dir`* and *`$cache_dir`* directories (`templates_c/` and `cache/`), so be sure the web server user account can write to them.

Note

This is usually user “nobody” and group “nobody”. For OS X users, the default is user “www” and group “www”. If you are using Apache, you can look in your `httpd.conf` file to see what user and group are being used.

Example 2.7. Permissions and making directories writable

```
chown nobody:nobody /web/www.example.com/guestbook/templates_c/
chmod 770 /web/www.example.com/guestbook/templates_c/

chown nobody:nobody /web/www.example.com/guestbook/cache/
chmod 770 /web/www.example.com/guestbook/cache/
```

Note

`chmod 770` will be fairly tight security, it only allows user “nobody” and group “nobody” read/write access to the directories. If you would like to open up read access to anyone (mostly for your own convenience of viewing these files), you can use `775` instead.

We need to create the `index.tpl` file that Smarty will display. This needs to be located in the *`$template_dir`*.

Example 2.8. /web/www.example.com/guestbook/templates/index.tpl

```
{* Smarty *}

Hello {$name}, welcome to Smarty!
```

Technical Note

`{* Smarty *}` is a template comment. It is not required, but it is good practice to start all your template files with this comment. It makes the file easy to recognize regardless of the file extension. For example, text editors could recognize the file and turn on special syntax highlighting.

Now lets edit `index.php`. We'll create an instance of Smarty, `assign()` a template variable and `display()` the `index.tpl` file.

Example 2.9. Editing /web/www.example.com/docs/guestbook/index.php

```
<?php

require_once(SMARTY_DIR . 'Smarty.class.php');

$smarty = new Smarty();

$smarty->template_dir = '/web/www.example.com/guestbook/templates/';
$smarty->compile_dir  = '/web/www.example.com/guestbook/templates_c/';
$smarty->config_dir   = '/web/www.example.com/guestbook/configs/';
$smarty->cache_dir    = '/web/www.example.com/guestbook/cache/';

$smarty->assign('name','Ned');

//** un-comment the following line to show the debug console
//$smarty->debugging = true;

$smarty->display('index.tpl');

?>
```

Note

In our example, we are setting absolute paths to all of the Smarty directories. If `/web/www.example.com/guestbook/` is within your PHP include_path, then these settings are not necessary. However, it is more efficient and (from experience) less error-prone to set them to absolute paths. This ensures that Smarty is getting files from the directories you intended.

Now navigate to the `index.php` file with the web browser. You should see *"Hello Ned, welcome to Smarty!"*

You have completed the basic setup for Smarty!

Extended Setup

This is a continuation of the basic installation, please read that first!

A slightly more flexible way to setup Smarty is to extend the class [http://php.net/ref.classobj] and initialize your Smarty environment. So instead of repeatedly setting directory paths, assigning the same vars, etc., we can do that in one place.

Lets create a new directory `/php/includes/guestbook/` and make a new file called `setup.php`. In our example environment, `/php/includes` is in our `include_path`. Be sure you set this up too, or use absolute file paths.

Example 2.10. /php/includes/guestbook/setup.php

```
<?php

// load Smarty library
require('Smarty.class.php');

// The setup.php file is a good place to load
// required application library files, and you
// can do that right here. An example:
// require('guestbook/guestbook.lib.php');

class Smarty_GuestBook extends Smarty {

   function Smarty_GuestBook()
   {

        // Class Constructor.
        // These automatically get set with each new instance.

        parent::__construct();

        $this->template_dir = '/web/www.example.com/guestbook/templates/';
        $this->compile_dir  = '/web/www.example.com/guestbook/templates_c/';
        $this->config_dir   = '/web/www.example.com/guestbook/configs/';
        $this->cache_dir    = '/web/www.example.com/guestbook/cache/';

        $this->caching = Smarty::CACHING_LIFETIME_CURRENT;
        $this->assign('app_name', 'Guest Book');
   }

}
?>
```

Now lets alter the `index.php` file to use `setup.php`:

Example 2.11. /web/www.example.com/guestbook/htdocs/index.php

```
<?php

require('guestbook/setup.php');

$smarty = new Smarty_GuestBook();

$smarty->assign('name','Ned');

$smarty->display('index.tpl');
?>
```

Now you see it is quite simple to bring up an instance of Smarty, just use `Smarty_GuestBook()` which automatically initializes everything for our application.

Part II. Smarty For Template Designers

Table of Contents

Chapter 3. Basic Syntax

All Smarty template tags are enclosed within delimiters. By default these are `{` and `}`, but they can be changed.

For the examples in this manual, we will assume that you are using the default delimiters. In Smarty, all content outside of delimiters is displayed as static content, or unchanged. When Smarty encounters template tags, it attempts to interpret them, and displays the appropriate output in their place.

Comments

Template comments are surrounded by asterisks, and that is surrounded by the delimiter tags like so:

```
{* this is a comment *}
```

Smarty comments are NOT displayed in the final output of the template, unlike `<!-- HTML comments -->`. These are useful for making internal notes in the templates which no one will see ;-)

Example 3.1. Comments within a template

```
{* I am a Smarty comment, I don't exist in the compiled output  *}
<html>
<head>
<title>{$title}</title>
</head>
<body>

{* another single line smarty comment  *}
<!-- HTML comment that is sent to the browser -->

{* this multiline smarty
   comment is
   not sent to browser
*}

{*********************************************************
Multi line comment block with credits block
  @ author:         bg@example.com
  @ maintainer:     support@example.com
  @ para:           var that sets block style
  @ css:            the style output
**********************************************************}

{* The header file with the main logo and stuff  *}
{include file='header.tpl'}


{* Dev note:  the $includeFile var is assigned in foo.php script  *}
<!-- Displays main content block -->
{include file=$includeFile}

{* this <select> block is redundant *}
{*
<select name="company">
  {html_options options=$vals selected=$selected_id}
</select>
*}

<!-- Show header from affiliate is disabled -->
{* $affiliate|upper *}

{* you cannot nest comments *}
{*
<select name="company">
  {* <option value="0">-- none -- </option> *}
  {html_options options=$vals selected=$selected_id}
</select>
*}

</body>
</html>
```

Variables

Template variables start with the $dollar sign. They can contain numbers, letters and underscores, much like a PHP variable [http://php.net/language.variables]. You can reference arrays by index numerically or non-numerically. Also reference object properties and methods.

Config file variables are an exception to the $dollar syntax and are instead referenced with surrounding #hashmarks#, or via the *`$smarty.config`* variable.

Example 3.2. Variables

```
{* display the server variable "SERVER_NAME" ($_SERVER['SERVER_NAME'])*}
{$smarty.server.SERVER_NAME}

Math and embedding tags:

{$x+$y}                             // will output the sum of x and y.
{assign var=foo value=$x+$y}        // in attributes
{$foo[$x+3]}                        // as array index
{$foo={counter}+3}                  // tags within tags
{$foo="this is message {counter}"}  // tags within double quoted strings

Defining Arrays:

{assign var=foo value=[1,2,3]}
{assign var=foo value=['y'=>'yellow','b'=>'blue']}
{assign var=foo value=[1,[9,8],3]}   // can be nested

Short variable assignment:

{$foo=$bar+2}
{$foo = strlen($bar)}               // function in assignment
{$foo = myfunct( ($x+$y)*3 )}       // as function parameter
{$foo.bar=1}                        // assign to specific array element
{$foo.bar.baz=1}
{$foo[]=1}                          // appending to an array

Smarty "dot" syntax (note: embedded {} are used to address ambiguities):

{$foo.a.b.c}        =>  $foo['a']['b']['c']
{$foo.a.$b.c}       =>  $foo['a'][$b]['c']         // with variable index
{$foo.a.{$b+4}.c}   =>  $foo['a'][$b+4]['c']       // with expression as index
{$foo.a.{$b.c}}     =>  $foo['a'][$b['c']]         // with nested index

PHP-like syntax, alternative to "dot" syntax:

{$foo[1]}             // normal access
{$foo['bar']}
{$foo['bar'][1]}
{$foo[$x+$x]}         // index may contain any expression
{$foo[$bar[1]]}       // nested index
{$foo[section_name]}  // smarty {section} access, not array access!

Variable variables:

$foo                     // normal variable
$foo_{$bar}              // variable name containing other variable
$foo_{$x+$y}             // variable name containing expressions
$foo_{$bar}_buh_{$blar}  // variable name with multiple segments
{$foo_{$x}}              // will output the variable $foo_1 if $x has a value of 1

Object chaining:

{$object->method1($x)->method2($y)}

Direct PHP function access:

{time()}
```

Note

Although Smarty can handle some very complex expressions and syntax, it is a good rule of thumb to keep the template syntax minimal and focused on presentation. If you find your template syntax getting too complex, it may be a good idea to move the bits that do not deal explicitly with presentation to PHP by way of plugins or modifiers.

Request variables such as `$_GET`, `$_SESSION`, etc are available via the reserved *`$smarty`* variable.

See also *`$smarty`*, config variables `{assign}` and `assign()`.

Functions

Every Smarty tag either prints a variable or invokes some sort of function. These are processed and displayed by enclosing the function and its attributes within delimiters like so: `{funcname attr1="val1" attr2="val2"}`.

Example 3.3. function syntax

```
{config_load file="colors.conf"}

{include file="header.tpl"}
{insert file="banner_ads.tpl" title="My Site"}

{if $logged_in}
    Welcome, <span style="color:{#fontColor#}">{$name}!</span>
{else}
    hi, {$name}
{/if}

{include file="footer.tpl"}
```

- Both built-in functions and custom functions have the same syntax within templates.
- Built-in functions are the **inner** workings of Smarty, such as `{if}`, `{section}` and `{strip}`. There should be no need to change or modify them.
- Custom functions are **additional** functions implemented via plugins. They can be modified to your liking, or you can create new ones. `{html_options}` is an example of a custom function.

See also `registerPlugin()`

Attributes

Most of the functions take attributes that specify or modify their behavior. Attributes to Smarty functions are much like HTML attributes. Static values don't have to be enclosed in quotes, but it is required for literal strings. Variables with or without modifers may also be used, and should not be in quotes. You can even use PHP function results, plugin results and complex expressions.

Some attributes require boolean values (`TRUE` or `FALSE`). These can be specified as `true` and `false`. If an attribute has no value assigned it gets the default boolean value of true.

Example 3.4. function attribute syntax

```
{include file="header.tpl"}

{include file="header.tpl" nocache}  // is equivalent to nocache=true

{include file="header.tpl" attrib_name="attrib value"}

{include file=$includeFile}

{include file=#includeFile# title="My Title"}

{assign var=foo value={counter}}  // plugin result

{assign var=foo value=substr($bar,2,5)}  // PHP function result

{assign var=foo value=$bar|strlen}  // using modifer

{assign var=foo value=$buh+$bar|strlen}  // more complex expresiion

{html_select_date display_days=true}

{mailto address="smarty@example.com"}

<select name="company_id">
  {html_options options=$companies selected=$company_id}
</select>
```

Note

Although Smarty can handle some very complex expressions and syntax, it is a good rule of thumb to keep the template syntax minimal and focused on presentation. If you find your template syntax getting too complex, it may be a good idea to move the bits that do not deal explicitly with presentation to PHP by way of plugins or modifiers.

Embedding Vars in Double Quotes

- Smarty will recognize assigned variables embedded in "double quotes" so long as the variable name contains only numbers, letters and under_scores. See naming [http://php.net/language.variables] for more detail.

- With any other characters, for example a .period or `$object>reference`, then the variable must be surrounded by `` `backticks` ``.

- In addition Smarty3 does allow embedded Smarty tags in double quoted strings. This is usefull if you want to include variables with modifers, plugin or PHP function results.

Example 3.5. Syntax examples

```
{func var="test $foo test"}              // sees $foo
{func var="test $foo_bar test"}          // sees $foo_bar
{func var="test `$foo[0]` test"}         // sees $foo[0]
{func var="test `$foo[bar]` test"}       // sees $foo[bar]
{func var="test $foo.bar test"}          // sees $foo (not $foo.bar)
{func var="test `$foo.bar` test"}        // sees $foo.bar
{func var="test `$foo.bar` test"|escape} // modifiers outside quotes!
{func var="test {$foo|escape} test"}     // modifiers inside quotes!
{func var="test {time()} test"}          // PHP function result
{func var="test {counter} test"}         // plugin result
{func var="variable foo is {if !$foo}not {/if} defined"} // Smarty block function
```

Example 3.6. Examples

```
{* will replace $tpl_name with value *}
{include file="subdir/$tpl_name.tpl"}

{* does NOT replace $tpl_name *}
{include file='subdir/$tpl_name.tpl'} // vars require double quotes!

{* must have backticks as it contains a dot "." *}
{cycle values="one,two,`$smarty.config.myval`"}

{* must have backticks as it contains a dot "." *}
{include file="`$module.contact`.tpl"}

{* can use variable with dot syntax *}
{include file="`$module.$view`.tpl"}
```

Note

Although Smarty can handle some very complex expressions and syntax, it is a good rule of thumb to keep the template syntax minimal and focused on presentation. If you find your template syntax getting too complex, it may be a good idea to move the bits that do not deal explicitly with presentation to PHP by way of plugins or modifiers.

See also `escape`.

Math

Math can be applied directly to variable values.

Example 3.7. math examples

```
{$foo+1}

{$foo*$bar}

{* some more complicated examples *}

{$foo->bar-$bar[1]*$baz->foo->bar()-3*7}

{if ($foo+$bar.test%$baz*134232+10+$b+10)}

{$foo|truncate:"`$fooTruncCount/$barTruncFactor-1`"}

{assign var="foo" value="`$foo+$bar`"}
```

Note

Although Smarty can handle some very complex expressions and syntax, it is a good rule of thumb to keep the template syntax minimal and focused on presentation. If you find your template syntax getting too complex, it may be a good idea to move the bits that do not deal explicitly with presentation to PHP by way of plugins or modifiers.

Escaping Smarty Parsing

It is sometimes desirable or even necessary to have Smarty ignore sections it would otherwise parse. A classic example is embedding Javascript or CSS code in a template. The problem arises as those languages use the { and } characters which are also the default delimiters for Smarty.

Note

A good practice for avoiding escapement altogether is by separating your Javascript/CSS into their own files and use standard HTML methods to access them. This will also take advantage of browser script caching. When you need to embed Smarty variables/functions into your Javascript/CSS, then the following applies.

In Smarty templates, the { and } braces will be ignored so long as they are surrounded by white space. This behavior can be disabled by setting the Smarty class variable *$auto_literal* to false.

Example 3.8. Using the auto-literal feature

```
<script>
   // the following braces are ignored by Smarty
   // since they are surrounded by whitespace
   function foobar {
 alert('foobar!');
   }
   // this one will need literal escapement
   {literal}
 function bazzy {alert('foobar!');}
   {/literal}
</script>
```

`{literal}..{/literal}` blocks are used for escaping blocks of template logic. You can also escape the braces individually with `{ldelim}`,`{rdelim}` tags or `{$smarty.ldelim}`,`{$smarty.rdelim}` variables.

Smarty's default delimiters { and } cleanly represent presentational content. However if another set of delimiters suit your needs better, you can change them with Smarty's *`$left_delimiter`* and *`$right_delimiter`* values.

Note

Changing delimiters affects ALL template syntax and escapement. Be sure to clear out cache and compiled files if you decide to change them.

Example 3.9. changing delimiters example

```
<?php

$smarty->left_delimiter = '<!--{';
$smarty->right_delimiter = '}-->';

$smarty->assign('foo', 'bar');
$smarty->assign('name', 'Albert');
$smarty->display('example.tpl');

?>
```

Where the template is:

```
Welcome <!--{$name}--> to Smarty
<script language="javascript">
  var foo = <!--{$foo}-->;
  function dosomething() {
    alert("foo is " + foo);
  }
  dosomething();
</script>
```

Chapter 4. Variables

Smarty has several different types of variables. The type of the variable depends on what symbol it is prefixed or enclosed within.

Variables in Smarty can be either displayed directly or used as arguments for functions, attributes and modifiers, inside conditional expressions, etc. To print a variable, simply enclose it in the delimiters so that it is the only thing contained between them.

Example 4.1. Example variables

```
{$Name}

{$product.part_no} <b>{$product.description}</b>

{$Contacts[row].Phone}

<body bgcolor="{#bgcolor#}">
```

Note

An easy way to examine assigned Smarty variables is with the debugging console.

Variables assigned from PHP

Assigned variables that are referenced by preceding them with a dollar ($) sign.

Example 4.2. Assigned variables

PHP code

```
<?php

$smarty = new Smarty();

$smarty->assign('firstname', 'Doug');
$smarty->assign('lastname', 'Evans');
$smarty->assign('meetingPlace', 'New York');

$smarty->display('index.tpl');

?>
```

`index.tpl` source:

```
Hello {$firstname} {$lastname}, glad to see you can make it.
<br />
{* this will not work as $variables are case sensitive *}
This weeks meeting is in {$meetingplace}.
{* this will work *}
This weeks meeting is in {$meetingPlace}.
```

This above would output:

```
Hello Doug Evans, glad to see you can make it.
<br />
This weeks meeting is in .
This weeks meeting is in New York.
```

Associative arrays

You can also reference associative array variables by specifying the key after a dot "." symbol.

Example 4.3. Accessing associative array variables

```
<?php
$smarty->assign('Contacts',
    array('fax' => '555-222-9876',
          'email' => 'zaphod@slartibartfast.example.com',
          'phone' => array('home' => '555-444-3333',
                           'cell' => '555-111-1234')
                           )
         );
$smarty->display('index.tpl');
?>
```

`index.tpl` source:

```
{$Contacts.fax}<br />
{$Contacts.email}<br />
{* you can print arrays of arrays as well *}
{$Contacts.phone.home}<br />
{$Contacts.phone.cell}<br />
```

this will output:

```
555-222-9876<br />
zaphod@slartibartfast.example.com<br />
555-444-3333<br />
555-111-1234<br />
```

Array indexes

You can reference arrays by their index, much like native PHP syntax.

Example 4.4. Accessing arrays by index

```
<?php
$smarty->assign('Contacts', array(
                           '555-222-9876',
                           'zaphod@slartibartfast.example.com',
                            array('555-444-3333',
                                  '555-111-1234')
                            ));
$smarty->display('index.tpl');
?>
```

`index.tpl` source:

```
{$Contacts[0]}<br />
{$Contacts[1]}<br />
{* you can print arrays of arrays as well *}
{$Contacts[2][0]}<br />
{$Contacts[2][1]}<br />
```

This will output:

```
555-222-9876<br />
zaphod@slartibartfast.example.com<br />
555-444-3333<br />
555-111-1234<br />
```

Objects

Properties of objects assigned from PHP can be referenced by specifying the property name after the `->` symbol.

Example 4.5. Accessing object properties

```
name:  {$person->name}<br />
email: {$person->email}<br />
```

this will output:

```
name:  Zaphod Beeblebrox<br />
email: zaphod@slartibartfast.example.com<br />
```

Variable scopes

You have the choice to assign variables to the scope of the main Smarty object, data objects created with `createData()`, and template objects created with `createTemplate()`. These objects can be chained. A template sees all the variables of its own object and all variables assigned to the objects in its chain of parent objects.

By default templates which are rendered by `$smarty->display(...)` or `$smarty->fetch(...)` calls are automatically linked to the Smarty object variable scope.

By assigning variables to individual data or template objects you have full control which variables can be seen by a template.

Example 4.6. Variable scope examples

```
// assign variable to Smarty object scope
$smarty->assign('foo','smarty');

// assign variables to data object scope
$data = $smarty->createData();
$data->assign('foo','data');
$data->assign('bar','bar-data');

// assign variables to other data object scope
$data2 = $smarty->createData($data);
$data2->assign('bar','bar-data2');

// assign variable to template object scope
$tpl = $smarty->createTemplate('index.tpl');
$tpl->assign('bar','bar-template');

// assign variable to template object scope with link to Smarty object
$tpl2 = $smarty->createTemplate('index.tpl',$smarty);
$tpl2->assign('bar','bar-template2');

// This display() does see $foo='smarty' from the $smarty object
$smarty->display('index.tpl');

// This display() does see $foo='data' and $bar='bar-data' from the data object $d
$smarty->display('index.tpl',$data);

// This display() does see $foo='data' from the data object $data
// and $bar='bar-data2' from the data object $data2
$smarty->display('index.tpl',$data2);

// This display() does see $bar='bar-template' from the template object $tpl
$tpl->display();  // or $smarty->display($tpl);

// This display() does see $bar='bar-template2' from the template object $tpl2
// and $foo='smarty' form the Smarty object $foo
$tpl2->display();  // or $smarty->display($tpl2);
```

See also `assign()`, `createData()` and `createTemplate()`.

Variables loaded from config files

Variables that are loaded from the config files are referenced by enclosing them within `#hash_marks#`, or with the smarty variable *`$smarty.config`*. The later syntax is useful for embedding into quoted attribute values, or accessing variable values such as $smarty.config.$foo.

Example 4.7. config variables

Example config file - `foo.conf`:

```
pageTitle = "This is mine"
bodyBgColor = '#eeeeee'
tableBorderSize = 3
tableBgColor = "#bbbbbb"
rowBgColor = "#cccccc"
```

A template demonstrating the *#hash#* method:

```
{config_load file='foo.conf'}
<html>
<title>{#pageTitle#}</title>
<body bgcolor="{#bodyBgColor#}">
<table border="{#tableBorderSize#}" bgcolor="{#tableBgColor#}">
<tr bgcolor="{#rowBgColor#}">
    <td>First</td>
    <td>Last</td>
    <td>Address</td>
</tr>
</table>
</body>
</html>
```

A template demonstrating the *$smarty.config* method:

```
{config_load file='foo.conf'}
<html>
<title>{$smarty.config.pageTitle}</title>
<body bgcolor="{$smarty.config.bodyBgColor}">
<table border="{$smarty.config.tableBorderSize}" bgcolor="{$smarty.config.tableBgC
<tr bgcolor="{$smarty.config.rowBgColor}">
    <td>First</td>
    <td>Last</td>
    <td>Address</td>
</tr>
</table>
</body>
</html>
```

Both examples would output:

```
<html>
<title>This is mine</title>
<body bgcolor="#eeeeee">
<table border="3" bgcolor="#bbbbbb">
<tr bgcolor="#cccccc">
 <td>First</td>
 <td>Last</td>
 <td>Address</td>
</tr>
</table>
</body>
</html>
```

Config file variables cannot be used until after they are loaded in from a config file. This procedure is explained later in this document under `{config_load}`.

See also variables and $smarty reserved variables

{$smarty} reserved variable

The PHP reserved *`{$smarty}`* variable can be used to access several environment and request variables. The full list of them follows.

Request variables

The request variables [http://php.net/reserved.variables] such as `$_GET`, `$_POST`, `$_COOKIE`, `$_SERVER`, `$_ENV` and `$_SESSION` (see `$request_vars_order` and `$request_use_auto_globals`) can be accessed as demonstrated in the examples below:

Example 4.8. Displaying request variables

```
{* display value of page from URL ($_GET) http://www.example.com/index.php?page=fo
{$smarty.get.page}

{* display the variable "page" from a form ($_POST['page']) *}
{$smarty.post.page}

{* display the value of the cookie "username" ($_COOKIE['username']) *}
{$smarty.cookies.username}

{* display the server variable "SERVER_NAME" ($_SERVER['SERVER_NAME'])*}
{$smarty.server.SERVER_NAME}

{* display the system environment variable "PATH" *}
{$smarty.env.PATH}

{* display the php session variable "id" ($_SESSION['id']) *}
{$smarty.session.id}

{* display the variable "username" from merged get/post/cookies/server/env *}
{$smarty.request.username}
```

Note

For historical reasons *`{$SCRIPT_NAME}`* is short-hand for *`{$smarty.server.SCRIPT_NAME}`*.

```
<a href="{$SCRIPT_NAME}?page=smarty">click me</a>
<a href="{$smarty.server.SCRIPT_NAME}?page=smarty">click me</a>
```

Note

Although Smarty provides direct access to PHP super globals for convenience, it should be used with caution. Specifically, GET and POST and REQUEST are commonly used for presentation purposes, but directly accessing SERVER, ENV, COOKIE and SESSION vars is typically avoided, as this is mixing underlying application code structure into the templates. A good practice is to assign specific needed values to template vars.

{$smarty.now}

The current timestamp [http://php.net/function.time] can be accessed with *`{$smarty.now}`*. The value reflects the number of seconds passed since the so-called Epoch on January 1, 1970, and can be passed directly to the `date_format` modifier for display. Note that `time()` [http://php.net/function.time] is called on each invocation; eg a script that takes three seconds to execute with a call to *`$smarty.now`* at start and end will show the three second difference.

```
{* use the date_format modifier to show current date and time *}
{$smarty.now|date_format:'%Y-%m-%d %H:%M:%S'}
```

{$smarty.const}

You can access PHP constant values directly. See also smarty constants.

```
<?php
// the constant defined in php
define('MY_CONST_VAL','CHERRIES');
?>
```

Output the constant in a template with

```
{$smarty.const.MY_CONST_VAL}
```

Note

Although Smarty provides direct access to PHP constants for convenience, it is typically avoided as this is mixing underlying application code structure into the templates. A good practice is to assign specific needed values to template vars.

{$smarty.capture}

Template output captured via the built-in `{capture}..{/capture}` function can be accessed using the *`{$smarty.capture}`* variable. See the `{capture}` page for more information.

{$smarty.config}

{$smarty.config}* variable can be used to refer to loaded config variables. *{$smarty.config.foo} is a synonym for *{#foo#}*. See the {config_load} page for more info.

{$smarty.section}

The *{$smarty.section}* variables can be used to refer to `{section}` loop properties. These have some very useful values such as `.first`, `.index`, etc.

Note

The `{$smarty.foreach}` variable is no longer used with the new `{foreach}` syntax, but is still supported with Smarty 2.x style foreach syntax.

{$smarty.template}

Returns the name of the current template being processed (without the directory.)

{$smarty.current_dir}

Returns the name of the directory for the current template being processed.

{$smarty.version}

Returns the version of Smarty the template was compiled with.

```
<div id="footer">Powered by Smarty {$smarty.version}</div>
```

{$smarty.block.child}

Returns block text from child template. See Template interitance.

{$smarty.block.parent}

Returns block text from parent template. See Template interitance

{$smarty.ldelim}, {$smarty.rdelim}

These variables are used for printing the left-delimiter and right-delimiter value literally, the same as `{ldelim},{rdelim}`.

See also assigned variables and config variables

Chapter 5. Variable Modifiers

Variable modifiers can be applied to variables, custom functions or strings. To apply a modifier, specify the value followed by a `|` (pipe) and the modifier name. A modifier may accept additional parameters that affect its behavior. These parameters follow the modifer name and are separated by a `:` (colon). Also, *all php-functions can be used as modifiers implicitly* (more below) and modifiers can be combined.

Example 5.1. Modifier examples

```
{* apply modifier to a variable *}
{$title|upper}

{* modifier with parameters *}
{$title|truncate:40:"..."}

{* apply modifier to a function parameter *}
{html_table loop=$myvar|upper}

{* with parameters *}
{html_table loop=$myvar|truncate:40:"..."}

{* apply modifier to literal string *}
{"foobar"|upper}

{* using date_format to format the current date *}
{$smarty.now|date_format:"%Y/%m/%d"}

{* apply modifier to a custom function *}
{mailto|upper address="smarty@example.com"}

{* using  php's str_repeat *}
{"="|str_repeat:80}

{* php's count *}
{$myArray|@count}

{* this will uppercase and truncate the whole array *}
<select name="name_id">
{html_options output=$my_array|upper|truncate:20}
</select>
```

- Modifiers can be applied to any type of variables, including arrays and objects.

Note

The default behavior was changed with Smarty 3. In Smarty 2.x, you had to use an "`@`" symbol to apply a modifier to an array, such as `{$articleTitle|@count}`. With Smarty 3, the "`@`" is no longer necessary, and is ignored.

If you want a modifier to apply to each individual item of an array, you will either need to loop the array in the template, or provide for this functionality inside your modifier function.

- Modifiers are autoloaded from the *`$plugins_dir`* or can be registered explicitly with the `registerPlugin()` function. The later is useful for sharing a function between php scripts and smarty templates.

- All php-functions can be used as modifiers implicitly, as demonstrated in the example above. However, using php-functions as modifiers has two little pitfalls:

  - First - sometimes the order of the function-parameters is not the desirable one. Formatting `$foo` with `{"%2.f"|sprintf:$foo}` actually works, but asks for the more intuitive, like `{$foo|string_format:"%2.f"}` that is provided by the Smarty distribution.

  - Secondly - if security is enabled, all php-functions that are to be used as modifiers have to be declared trusted in the *`$modifiers`* property of the securty policy. See the Security section for details.

See also `registerPlugin()`, combining modifiers. and extending smarty with plugins

capitalize

This is used to capitalize the first letter of all words in a variable. This is similar to the PHP `ucwords()` [http://php.net/ucwords] function.

Parameter Position	Type	Required	Default	Description
1	boolean	No	`FALSE`	This determines whether or not words with digits will be uppercased

Example 5.2. capitalize

```
<?php

$smarty->assign('articleTitle', 'next x-men film, x3, delayed.');

?>
```

Where the template is:

```
{$articleTitle}
{$articleTitle|capitalize}
{$articleTitle|capitalize:true}
```

Will output:

```
next x-men film, x3, delayed.
Next X-Men Film, x3, Delayed.
Next X-Men Film, X3, Delayed.
```

See also `lower` and `upper`

cat

This value is concatenated to the given variable.

Parameter Position	Type	Required	Default	Description
1	string	No	*empty*	This value to catenate to the given variable.

Example 5.3. cat

```
<?php

$smarty->assign('articleTitle', "Psychics predict world didn't end");

?>
```

Where template is:

```
{$articleTitle|cat:' yesterday.'}
```

Will output:

```
Psychics predict world didn't end yesterday.
```

count_characters

This is used to count the number of characters in a variable.

Parameter Position	Type	Required	Default	Description
1	boolean	No	FALSE	This determines whether or not to include whitespace characters in the count.

Example 5.4. count_characters

```
<?php

$smarty->assign('articleTitle', 'Cold Wave Linked to Temperatures.');

?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|count_characters}
{$articleTitle|count_characters:true}
```

Will output:

```
Cold Wave Linked to Temperatures.
29
33
```

See also `count_words`, `count_sentences` and `count_paragraphs`.

count_paragraphs

This is used to count the number of paragraphs in a variable.

Example 5.5. count_paragraphs

```
<?php

$smarty->assign('articleTitle',
                 "War Dims Hope for Peace. Child's Death Ruins Couple's Holiday.\n
                 Man is Fatally Slain. Death Causes Loneliness, Feeling of Isolati
                );

?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|count_paragraphs}
```

Will output:

```
War Dims Hope for Peace. Child's Death Ruins Couple's Holiday.

Man is Fatally Slain. Death Causes Loneliness, Feeling of Isolation.
2
```

See also `count_characters`, `count_sentences` and `count_words`.

count_sentences

This is used to count the number of sentences in a variable.

Example 5.6. count_sentences

```
<?php

$smarty->assign('articleTitle',
                 'Two Soviet Ships Collide - One Dies.
                 Enraged Cow Injures Farmer with Axe.'
                 );

?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|count_sentences}
```

Will output:

```
Two Soviet Ships Collide - One Dies. Enraged Cow Injures Farmer with Axe.
2
```

See also `count_characters`, `count_paragraphs` and `count_words`.

count_words

This is used to count the number of words in a variable.

Example 5.7. count_words

```
<?php

$smarty->assign('articleTitle', 'Dealers Will Hear Car Talk at Noon.');

?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|count_words}
```

This will output:

```
Dealers Will Hear Car Talk at Noon.
7
```

See also `count_characters`, `count_paragraphs` and `count_sentences`.

date_format

This formats a date and time into the given `strftime()` [http://php.net/strftime] format. Dates can be passed to Smarty as unix timestamps [http://php.net/function.time], mysql timestamps or any string made up of month day year, parsable by php's `strtotime()` [http://php.net/strtotime]. Designers can then use `date_format` to have complete control of the formatting of the date. If the date passed to `date_format` is empty and a second parameter is passed, that will be used as the date to format.

Parameter Position	Type	Required	Default	Description
1	string	No	%b %e, %Y	This is the format for the outputted date.
2	string	No	n/a	This is the default date if the input is empty.

Note

Since Smarty-2.6.10 numeric values passed to `date_format` are *always* (except for mysql timestamps, see below) interpreted as a unix timestamp.

Before Smarty-2.6.10 numeric strings that where also parsable by `strtotime()` in php (like `YYYYMMDD`) where sometimes (depending on the underlying implementation of `strtotime()`) interpreted as date strings and NOT as timestamps.

The only exception are mysql timestamps: They are also numeric only and 14 characters long (`YYYYMMDDHHMMSS`), mysql timestamps have precedence over unix timestamps.

Programmers note

`date_format` is essentially a wrapper to PHP's `strftime()` [http://php.net/strftime] function. You may have more or less conversion specifiers available depending on your system's `strftime()` [http://php.net/strftime] function where PHP was compiled. Check your system's manpage for a full list of valid specifiers. However, a few of the specifiers are emulated on Windows. These are: %D, %e, %h, %l, %n, %r, %R, %t, %T.

Example 5.8. date_format

```
<?php

$config['date'] = '%I:%M %p';
$config['time'] = '%H:%M:%S';
$smarty->assign('config', $config);
$smarty->assign('yesterday', strtotime('-1 day'));

?>
```

This template uses *`$smarty.now`* to get the current time:

```
{$smarty.now|date_format}
{$smarty.now|date_format:"%D"}
{$smarty.now|date_format:$config.date}
{$yesterday|date_format}
{$yesterday|date_format:"%A, %B %e, %Y"}
{$yesterday|date_format:$config.time}
```

This above will output:

```
Jan 1, 2022
01/01/22
02:33 pm
Dec 31, 2021
Monday, December 1, 2021
14:33:00
```

date_format conversion specifiers:

- %a - abbreviated weekday name according to the current locale
- %A - full weekday name according to the current locale
- %b - abbreviated month name according to the current locale
- %B - full month name according to the current locale
- %c - preferred date and time representation for the current locale
- %C - century number (the year divided by 100 and truncated to an integer, range 00 to 99)
- %d - day of the month as a decimal number (range 01 to 31)
- %D - same as %m/%d/%y
- %e - day of the month as a decimal number, a single digit is preceded by a space (range 1 to 31)
- %g - Week-based year within century [00,99]
- %G - Week-based year, including the century [0000,9999]
- %h - same as %b
- %H - hour as a decimal number using a 24-hour clock (range 00 to 23)
- %I - hour as a decimal number using a 12-hour clock (range 01 to 12)
- %j - day of the year as a decimal number (range 001 to 366)
- %k - Hour (24-hour clock) single digits are preceded by a blank. (range 0 to 23)
- %l - hour as a decimal number using a 12-hour clock, single digits preceeded by a space (range 1 to 12)
- %m - month as a decimal number (range 01 to 12)
- %M - minute as a decimal number
- %n - newline character
- %p - either `am' or `pm' according to the given time value, or the corresponding strings for the current locale
- %r - time in a.m. and p.m. notation
- %R - time in 24 hour notation
- %S - second as a decimal number
- %t - tab character
- %T - current time, equal to %H:%M:%S

- %u - weekday as a decimal number [1,7], with 1 representing Monday
- %U - week number of the current year as a decimal number, starting with the first Sunday as the first day of the first week
- %V - The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week.
- %w - day of the week as a decimal, Sunday being 0
- %W - week number of the current year as a decimal number, starting with the first Monday as the first day of the first week
- %x - preferred date representation for the current locale without the time
- %X - preferred time representation for the current locale without the date
- %y - year as a decimal number without a century (range 00 to 99)
- %Y - year as a decimal number including the century
- %Z - time zone or name or abbreviation
- %% - a literal `%' character

See also *`$smarty.now`*, `strftime()` [http://php.net/strftime], `{html_select_date}` and the date tips page.

default

This is used to set a default value for a variable. If the variable is unset or an empty string, the given default value is printed instead. Default takes the one argument.

Parameter Position	Type	Required	Default	Description
1	string	No	*empty*	This is the default value to output if the variable is empty.

Example 5.9. default

```
<?php

$smarty->assign('articleTitle', 'Dealers Will Hear Car Talk at Noon.');
$smarty->assign('email', '');

?>
```

Where template is:

```
{$articleTitle|default:'no title'}
{$myTitle|default:'no title'}
{$email|default:'No email address available'}
```

Will output:

```
Dealers Will Hear Car Talk at Noon.
no title
No email address available
```

See also the default variable handling and the blank variable handling pages.

escape

`escape` is used to encode or escape a variable to `html`, `url`, `single quotes`, `hex`, `hexentity`, `javascript` and `mail`. By default its `html`.

Parameter Position	Type	Required	Possible Values	Default	Description
1	string	No	`html`, `htmlall`, `url`, `urlpathinfo`, `quotes`, `hex`, `hexentity`, `javascript`, `mail`	`html`	This is the escape format to use.
2	string	No	`ISO-8859-1`, `UTF-8`, and any character set	`UTF-8`	The character set encoding passed to

Parameter Position	Type	Required	Possible Values	Default	Description
			supported by htmlentities() [http://php.net/htmlentities]		htmlentities() et. al.

Example 5.10. escape

```
<?php

$smarty->assign('articleTitle',
                "'Stiff Opposition Expected to Casketless Funeral Plan'"
                );
$smarty->assign('EmailAddress','smarty@example.com');

?>
```

These are example `escape` template lines followed by the output

```
{$articleTitle}
'Stiff Opposition Expected to Casketless Funeral Plan'

{$articleTitle|escape}
&#039;Stiff Opposition Expected to Casketless Funeral Plan&#039;

{$articleTitle|escape:'html'}    {* escapes  & " ' < > *}
&#039;Stiff Opposition Expected to Casketless Funeral Plan&#039;

{$articleTitle|escape:'htmlall'} {* escapes ALL html entities *}
&#039;Stiff Opposition Expected to Casketless Funeral Plan&#039;

<a href="?title={$articleTitle|escape:'url'}">click here</a>
<a
href="?title=%27Stiff%20Opposition%20Expected%20to%20Casketless%20Funeral%20Plan%2

{$articleTitle|escape:'quotes'}
\'Stiff Opposition Expected to Casketless Funeral Plan\'

<a href="mailto:{$EmailAddress|escape:"hex"}">{$EmailAddress|escape:"hexentity"}</
{$EmailAddress|escape:'mail'}    {* this converts to email to text *}
<a href="mailto:%62%6f%..snip..%65%74">&#x62;&#x6f;&#x62..snip..&#x65;&#x74;</a>

{'mail@example.com'|escape:'mail'}
smarty [AT] example [DOT] com
```

Example 5.11. Other examples

```
{* the "rewind" paramater registers the current location *}
<a href="$my_path?page=foo&rewind=$my_uri|urlencode}">click here</a>
```

This snippet is useful for emails, but see also `{mailto}`

```
{* email address mangled *}
<a href="mailto:{$EmailAddress|escape:'hex'}">{$EmailAddress|escape:'mail'}</a>
```

See also escaping smarty parsing, `{mailto}` and the obfuscating email addresses page.

indent

This indents a string on each line, default is 4. As an optional parameter, you can specify the number of characters to indent. As an optional second parameter, you can specify the character to use to indent with eg use `"\t"` for a tab.

Parameter Position	Type	Required	Default	Description
1	integer	No	4	This determines how many characters to indent to.
2	string	No	(one space)	This is the character used to indent with.

Example 5.12. indent

```
<?php

$smarty->assign('articleTitle',
                'NJ judge to rule on nude beach.
Sun or rain expected today, dark tonight.
Statistics show that teen pregnancy drops off significantly after 25.'
                );
?>
```

Where template is:

```
{$articleTitle}

{$articleTitle|indent}

{$articleTitle|indent:10}

{$articleTitle|indent:1:"\t"}
```

Will output:

```
NJ judge to rule on nude beach.
Sun or rain expected today, dark tonight.
Statistics show that teen pregnancy drops off significantly after 25.

    NJ judge to rule on nude beach.
    Sun or rain expected today, dark tonight.
    Statistics show that teen pregnancy drops off significantly after 25.

          NJ judge to rule on nude beach.
          Sun or rain expected today, dark tonight.
          Statistics show that teen pregnancy drops off significantly after 25.

        NJ judge to rule on nude beach.
        Sun or rain expected today, dark tonight.
        Statistics show that teen pregnancy drops off significantly after 25.
```

See also `strip`, `wordwrap` and `spacify`.

lower

This is used to lowercase a variable. This is equivalent to the PHP `strtolower()` [http://php.net/strtolower] function.

Example 5.13. lower

```
<?php

$smarty->assign('articleTitle', 'Two Convicts Evade Noose, Jury Hung.');

?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|lower}
```

This will output:

```
Two Convicts Evade Noose, Jury Hung.
two convicts evade noose, jury hung.
```

See also `upper` and `capitalize`.

nl2br

All `"\n"` line breaks will be converted to html `<br />` tags in the given variable. This is equivalent to the PHP's `nl2br()` [http://php.net/nl2br] function.

Example 5.14. nl2br

```
<?php

$smarty->assign('articleTitle',
                "Sun or rain expected\ntoday, dark tonight"
                );

?>
```

Where the template is:

```
{$articleTitle|nl2br}
```

Will output:

```
Sun or rain expected<br />today, dark tonight
```

See also `word_wrap`, `count_paragraphs` and `count_sentences`.

regex_replace

A regular expression search and replace on a variable. Use the `preg_replace()` [http://php.net/preg_replace] syntax from the PHP manual.

Note

Although Smarty supplies this regex convenience modifier, it is usually better to apply regular expressions in PHP, either via custom functions or modifiers. Regular expressions are considered application code and are not part of presentation logic.

Parameters

Parameter Position	Type	Required	Default	Description
1	string	Yes	*n/a*	This is the regular expression to be replaced.
2	string	Yes	*n/a*	This is the string of text to replace with.

Example 5.15. regex_replace

```
<?php

$smarty->assign('articleTitle', "Infertility unlikely to\nbe passed on, experts sa

?>
```

Where template is:

```
{* replace each carriage return, tab and new line with a space *}

{$articleTitle}
{$articleTitle|regex_replace:"/[\r\t\n]/":" "}
```

Will output:

```
Infertility unlikely to
be passed on, experts say.
Infertility unlikely to be passed on, experts say.
```

See also `replace` and `escape`.

replace

A simple search and replace on a variable. This is equivalent to the PHP's `str_replace()` [http://php.net/str_replace] function.

Parameter Position	Type	Required	Default	Description
1	string	Yes	*n/a*	This is the string of text to be replaced.
2	string	Yes	*n/a*	This is the string of text to replace with.

Example 5.16. replace

```
<?php

$smarty->assign('articleTitle', "Child's Stool Great for Use in Garden.");

?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|replace:'Garden':'Vineyard'}
{$articleTitle|replace:' ':'   '}
```

Will output:

```
Child's Stool Great for Use in Garden.
Child's Stool Great for Use in Vineyard.
Child's   Stool   Great   for   Use   in   Garden.
```

See also `regex_replace` and `escape`.

spacify

`spacify` is a way to insert a space between every character of a variable. You can optionally pass a different character or string to insert.

Parameter Position	Type	Required	Default	Description
1	string	No	*one space*	This what gets inserted between each character of the variable.

Example 5.17. spacify

```
<?php

$smarty->assign('articleTitle', 'Something Went Wrong in Jet Crash, Experts Say.')

?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|spacify}
{$articleTitle|spacify:"^^"}
```

Will output:

```
Something Went Wrong in Jet Crash, Experts Say.
S o m e t h i n g   W .... snip ....  s h ,   E x p e r t s   S a y .
S^^o^^m^^e^^t^^h^^i^^n^^g^^ .... snip .... ^^e^^r^^t^^s^^ ^^S^^a^^y^^.
```

See also wordwrap and nl2br.

string_format

This is a way to format strings, such as decimal numbers and such. Use the syntax for sprintf() [http://php.net/sprintf] for the formatting.

Parameter Position	Type	Required	Default	Description
1	string	Yes	*n/a*	This is what format to use. (sprintf)

Example 5.18. string_format

```
<?php

$smarty->assign('number', 23.5787446);

?>
```

Where template is:

```
{$number}
{$number|string_format:"%.2f"}
{$number|string_format:"%d"}
```

Will output:

```
23.5787446
23.58
24
```

See also `date_format`.

strip

This replaces all repeated spaces, newlines and tabs with a single space, or with the supplied string.

Note

If you want to strip blocks of template text, use the built-in `{strip}` function.

Example 5.19. strip

```
<?php
$smarty->assign('articleTitle', "Grandmother of\neight makes\t    hole in one.");
$smarty->display('index.tpl');
?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|strip}
{$articleTitle|strip:'&nbsp;'}
```

Will output:

```
Grandmother of
eight makes        hole in one.
Grandmother of eight makes hole in one.
Grandmother&nbsp;of&nbsp;eight&nbsp;makes&nbsp;hole&nbsp;in&nbsp;one.
```

See also `{strip}` and `truncate`.

strip_tags

This strips out markup tags, basically anything between < and >.

Parameter Position	Type	Required	Default	Description
1	bool	No	`TRUE`	This determines whether the tags are replaced by ' ' or ''

Example 5.20. strip_tags

```
<?php

$smarty->assign('articleTitle',
                "Blind Woman Gets <font face=\"helvetica\">New
Kidney</font> from Dad she Hasn't Seen in <b>years</b>."
               );

?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|strip_tags} {* same as {$articleTitle|strip_tags:true} *}
{$articleTitle|strip_tags:false}
```

Will output:

```
Blind Woman Gets <font face="helvetica">New Kidney</font> from Dad she Hasn't Seen
Blind Woman Gets  New Kidney  from Dad she Hasn't Seen in  years .
Blind Woman Gets New Kidney from Dad she Hasn't Seen in years.
```

See also `replace` and `regex_replace`.

truncate

This truncates a variable to a character length, the default is 80. As an optional second parameter, you can specify a string of text to display at the end if the variable was truncated. The characters in the string are included with the original truncation length. By default, `truncate` will attempt to cut off at a word boundary. If you want to cut off at the exact character length, pass the optional third parameter of `TRUE`.

Parameter Position	Type	Required	Default	Description
1	integer	No	80	This determines how many characters to truncate to.
2	string	No	...	This is a text string that replaces the truncated text. Its length is included

Parameter Position	Type	Required	Default	Description
				in the truncation length setting.
3	boolean	No	FALSE	This determines whether or not to truncate at a word boundary with FALSE, or at the exact character with TRUE.
4	boolean	No	FALSE	This determines whether the truncation happens at the end of the string with FALSE, or in the middle of the string with TRUE. Note that if this setting is TRUE, then word boundaries are ignored.

Example 5.21. truncate

```
<?php
$smarty->assign('articleTitle', 'Two Sisters Reunite after Eighteen Years at Check
?>
```

where template is:

```
{$articleTitle}
{$articleTitle|truncate}
{$articleTitle|truncate:30}
{$articleTitle|truncate:30:""}
{$articleTitle|truncate:30:"---"}
{$articleTitle|truncate:30:"":true}
{$articleTitle|truncate:30:"...":true}
{$articleTitle|truncate:30:'..':true:true}
```

This will output:

```
Two Sisters Reunite after Eighteen Years at Checkout Counter.
Two Sisters Reunite after Eighteen Years at Checkout Counter.
Two Sisters Reunite after...
Two Sisters Reunite after
Two Sisters Reunite after---
Two Sisters Reunite after Eigh
Two Sisters Reunite after E...
Two Sisters Re..ckout Counter.
```

upper

This is used to uppercase a variable. This is equivalent to the PHP `strtoupper()` [http://php.net/strtoupper] function.

Example 5.22. upper

```
<?php
$smarty->assign('articleTitle', "If Strike isn't Settled Quickly it may Last a Whi
?>
```

Where template is:

```
{$articleTitle}
{$articleTitle|upper}
```

Will output:

```
If Strike isn't Settled Quickly it may Last a While.
IF STRIKE ISN'T SETTLED QUICKLY IT MAY LAST A WHILE.
```

See also `lower` and `capitalize`.

wordwrap

Wraps a string to a column width, the default is 80. As an optional second parameter, you can specify a string of text to wrap the text to the next line, the default is a carriage return `"\n"`. By default, `wordwrap` will attempt to wrap at a word boundary. If you want to cut off at the exact character length, pass the optional third parameter as `TRUE`. This is equivalent to the PHP `wordwrap()` [http://php.net/wordwrap] function.

Parameter Position	Type	Required	Default	Description
1	integer	No	80	This determines how many columns to wrap to.
2	string	No	\n	This is the string used to wrap words with.
3	boolean	No	`FALSE`	This determines whether or not to wrap at a word boundary (`FALSE`), or at the exact character (`TRUE`).

Example 5.23. wordwrap

```
<?php

$smarty->assign('articleTitle',
                "Blind woman gets new kidney from dad she hasn't seen in years."
               );

?>
```

Where template is

```
{$articleTitle}

{$articleTitle|wordwrap:30}

{$articleTitle|wordwrap:20}

{$articleTitle|wordwrap:30:"<br />\n"}

{$articleTitle|wordwrap:26:"\n":true}
```

Will output:

```
Blind woman gets new kidney from dad she hasn't seen in years.

Blind woman gets new kidney
from dad she hasn't seen in
years.

Blind woman gets new
kidney from dad she
hasn't seen in
years.

Blind woman gets new kidney<br />
from dad she hasn't seen in<br />
years.

Blind woman gets new kidn
ey from dad she hasn't se
en in years.
```

See also `nl2br` and `{textformat}`.

Chapter 6. Combining Modifiers

You can apply any number of modifiers to a variable. They will be applied in the order they are combined, from left to right. They must be separated with a `|` (pipe) character.

Example 6.1. combining modifiers

```
<?php

$smarty->assign('articleTitle', 'Smokers are Productive, but Death Cuts Efficiency

?>
```

where template is:

```
{$articleTitle}
{$articleTitle|upper|spacify}
{$articleTitle|lower|spacify|truncate}
{$articleTitle|lower|truncate:30|spacify}
{$articleTitle|lower|spacify|truncate:30:". . ."}
```

The above example will output:

```
Smokers are Productive, but Death Cuts Efficiency.
S M O K E R S   A R ....snip....  H   C U T S   E F F I C I E N C Y .
s m o k e r s   a r ....snip....  b u t   d e a t h   c u t s...
s m o k e r s   a r e   p r o d u c t i v e ,   b u t . . .
s m o k e r s   a r e   p. . .
```

Chapter 7. Built-in Functions

Smarty comes with several built-in functions. These built-in functions are the integral part of the smarty template engine. They are compiled into corresponding inline PHP code for maximum performance.

You cannot create your own custom functions with the same name; and you should not need to modify the built-in functions.

A few of these functions have an *`assign`* attribute which collects the result the function to a named template variable instead of being output; much like the `{assign}` function.

{$var=...}

This is a short-hand version of the {assign} function. You can assign values directly to the template, or assign values to array elements too.

Note

Assignment of variables in-template is essentially placing application logic into the presentation that may be better handled in PHP. Use at your own discretion.

The following attributes can be added to the tag:

Attributes:

Attribute Name	Shorthand	Type	Required	Default	Description
scope	n/a	string	No	*n/a*	The scope of the assigned variable: 'parent','root' or 'global'

Option Flags:

Name	Description
nocache	Assingns the variable with the 'nocache' attribute

Example 7.1. Simple assignment

```
{$name='Bob'}

The value of $name is {$name}.
```

The above example will output:

```
The value of $name is Bob.
```

Example 7.2. Assignment with math

```
{$running_total=$running_total+$some_array[row].some_value}
```

Example 7.3. Assignment of an array element

```
{$user.name="Bob"}
```

Example 7.4. Assignment of an multidimensional array element

```
{$user.name.first="Bob"}
```

Example 7.5. Appending an array

```
{$users[]="Bob"}
```

Example 7.6. Assigment in the scope of calling template

Variables assigned in the included template will be seen in the including template.

```
{include file="sub_template.tpl"}
...
{* display variable assigned in sub_template *}
{$foo}<br>
...
```

The template above includes the example `sub_template.tpl` below

```
...
{* foo will be known also in the including template *}
{$foo="something" scope=parent}
{* bar is assigned only local in the including template *}
{$bar="value"}
...
```

See also `{assign}` and `{append}`

{append}

`{append}` is used for creating or appending template variable arrays **during the execution of a template**.

Note

Assignment of variables in-template is essentially placing application logic into the presentation that may be better handled in PHP. Use at your own discretion.

Attributes:

Attribute Name	Type	Required	Default	Description
var	string	Yes	*n/a*	The name of the variable being assigned
value	string	Yes	*n/a*	The value being assigned
index	string	No	*n/a*	The index for the new array element. If not specified the value is append to the end of the array.
scope	string	No	*n/a*	The scope of the assigned variable: 'parent','root' or 'global'

Option Flags:

Name	Description
nocache	Assingns the variable with the 'nocache' attribute

Example 7.7. {append}

```
{append var='name' value='Bob' index='first'}
{append var='name' value='Meyer' index='last'}
// or
{append 'name' 'Bob' index='first'} {* short-hand *}
{append 'name' 'Meyer' index='last'} {* short-hand *}

The first name is {$name.first}.<br>
The last name is {$name.last}.
```

The above example will output:

```
The first name is Bob.
The last name is Meyer.
```

See also `append()` and `getTemplateVars()`.

{assign}

`{assign}` is used for assigning template variables **during the execution of a template**.

Note

Assignment of variables in-template is essentially placing application logic into the presentation that may be better handled in PHP. Use at your own discretion.

Note

See also the `short-form` method of assigning template vars.

Attributes:

Attribute Name	Type	Required	Default	Description
var	string	Yes	*n/a*	The name of the variable being assigned
value	string	Yes	*n/a*	The value being assigned
scope	string	No	*n/a*	The scope of the assigned variable:

Attribute Name	Type	Required	Default	Description
				'parent','root' or 'global'

Option Flags:

Name	Description
nocache	Assingns the variable with the 'nocache' attribute

Example 7.8. {assign}

```
{assign var="name" value="Bob"}
{assign "name" "Bob"} {* short-hand *}

The value of $name is {$name}.
```

The above example will output:

```
The value of $name is Bob.
```

Example 7.9. {assign} as a nocache variable

```
{assign var="name" value="Bob" nocache}
{assign "name" "Bob" nocache} {* short-hand *}

The value of $name is {$name}.
```

The above example will output:

```
The value of $name is Bob.
```

Example 7.10. {assign} with some maths

```
{assign var=running_total value=$running_total+$some_array[$row].some_value}
```

Example 7.11. {assign} in the scope of calling template

Variables assigned in the included template will be seen in the including template.

```
{include file="sub_template.tpl"}
...
{* display variable assigned in sub_template *}
{$foo}<br>
...
```

The template above includes the example `sub_template.tpl` below

```
...
{* foo will be known also in the including template *}
{assign var="foo" value="something" scope=parent}
{* bar is assigned only local in the including template *}
{assign var="bar" value="value"}
...
```

Example 7.12. {assign} a variable to current scope tree

You can assign a variable to root of the current root tree. The variable is seen by all templates using the same root tree.

```
{assign var=foo value="bar" scope="root"}
```

Example 7.13. {assign} a global variable

A glkobal variable is seen by all templates.

```
{assign var=foo value="bar" scope="global"}
{assign "foo" "bar" scope="global"} {* short-hand *}
```

Example 7.14. Accessing {assign} variables from a PHP script

To access `{assign}` variables from a php script use `getTemplateVars()`. Here's the template that creates the variable *`$foo`*.

```
{assign var="foo" value="Smarty"}
```

The template variables are only available after/during template execution as in the following script.

```
<?php

// this will output nothing as the template has not been executed
echo $smarty->getTemplateVars('foo');

// fetch the template to a variable
$whole_page = $smarty->fetch('index.tpl');

// this will output 'smarty' as the template has been executed
echo $smarty->getTemplateVars('foo');

$smarty->assign('foo','Even smarter');

// this will output 'Even smarter'
echo $smarty->getTemplateVars('foo');

?>
```

The following functions can also *optionally* assign template variables.

`{capture}`, `{include}`, `{include_php}`, `{insert}`, `{counter}`, `{cycle}`, `{eval}`, `{fetch}`, `{math}`, `{textformat}`

See also `{$var=...}`, `assign()` and `getTemplateVars()`.

{block}

`{block}` is used to define a named area of template source for template inheritance. For details see section of Template Interitance.

The `{block}` template source area of a child template will replace the correponding areas in the parent template(s).

Optionally `{block}` areas of child and parent templates can be merged into each other. You can append or prepend the parent `{block}` content by using the `append` or `prepend` option flag with the childs `{block}` definition. With the {$smarty.block.parent} the `{block}` content of the parent template can be inserted at any location of the child `{block}` content. {$smarty.block.child} inserts the `{block}` content of the child template at any location of the parent `{block}`.

`{blocks}'s` can be nested.

Attributes:

Attribute Name	Type	Required	Default	Description
name	string	Yes	*n/a*	The name of the template source block

Option Flags (in child templates only):

Name	Description
append	The `{block}` content will be be appended to the content of the parent template `{block}`
prepend	The `{block}` content will be prepended to the content of the parent template `{block}`
nocache	Disables caching of the `{block}` content

Example 7.15. Simple `{block}` example

parent.tpl

```
<html>
  <head>
    <title>{block name="title"}Default Title{/block}</title>
    <title>{block "title"}Default Title{/block}</title>  {* short-hand  *}
  </head>
</html>
```

child.tpl

```
{extends file="parent.tpl"}
{block name="title"}
Page Title
{/block}
```

The result would look like

```
<html>
  <head>
    <title>Page Title</title>
  </head>
</html>
```

Example 7.16. Prepend `{block}` example

parent.tpl

```
<html>
  <head>
    <title>{block name="title"}Title - {/block}</title>
  </head>
</html>
```

child.tpl

```
{extends file="parent.tpl"}
{block name="title" prepend}
Page Title
{/block}
```

The result would look like

```
<html>
  <head>
    <title>Title - Page Title</title>
  </head>
</html>
```

Example 7.17. Append `{block}` example

parent.tpl

```
<html>
  <head>
    <title>{block name="title"} is my titel{/block}</title>
  </head>
</html>
```

child.tpl

```
{extends file="parent.tpl"}
{block name="title" append}
Page Title
{/block}
```

The result would look like

```
<html>
  <head>
    <title>Page title is my titel</title>
  </head>
</html>
```

Example 7.18. `{$smarty.block.child}` example

parent.tpl

```
<html>
  <head>
    <title>{block name="title"}The {$smarty.block.child} was inserted here{/block}
  </head>
</html>
```

child.tpl

```
{extends file="parent.tpl"}
{block name="title" append}
Child Title
{/block}
```

The result would look like

```
<html>
  <head>
    <title>The - Child Title - was inserted here</title>
  </head>
</html>
```

Example 7.19. `{$smarty.block.parent}` example

parent.tpl

```
<html>
  <head>
    <title>{block name="title"}Parent Title{/block}</title>
  </head>
</html>
```

child.tpl

```
{extends file="parent.tpl"}
{block name="title" append}
You will see now - {$smarty.block.parent} - here
{/block}
```

The result would look like

```
<html>
  <head>
    <title>You will see now - Parent Title - here</title>
  </head>
</html>
```

See also Template interitance, *`$smarty.block.parent`*, *`$smarty.block.child`*, and `{extends}`

{call}

`{call}` is used to call a template function defined by the `{function}` tag just like a plugin function.

> ### Note
>
> Template functions are defined global. Since the Smarty compiler is a single-pass compiler, The `{call}` tag must be used to call a template function defined externally from the given template. Otherwise you can directly use the function as `{funcname ...}` in the template.

- The `{call}` tag must have the *`name`* attribute which contains the the name of the template function.

- Values for variables can be passed to the template function as attributes.

Attributes:

Attribute Name	Type	Required	Default	Description
name	string	Yes	*n/a*	The name of the template function
assign	string	No	*n/a*	The name of the variable that the output of called template function will be assigned to
[var ...]	[var type]	No	*n/a*	variable to pass local to template function

Option Flags:

Name	Description
nocache	Cakk the template function in nocache mode

Example 7.20. Calling a recursive menu example

```
{* define the function *}
{function name=menu level=0}
  <ul class="level{$level}">
  {foreach $data as $entry}
    {if is_array($entry)}
      <li>{$entry@key}</li>
      {call name=menu data=$entry level=$level+1}
    {else}
      <li>{$entry}</li>
    {/if}
  {/foreach}
  </ul>
{/function}

{* create an array to demonstrate *}
{$menu = ['item1','item2','item3' => ['item3-1','item3-2','item3-3' =>
['item3-3-1','item3-3-2']],'item4']}

{* run the array through the function *}
{call name=menu data=$menu}
{call menu data=$menu} {* short-hand *}
```

Will generate the following output

```
* item1
* item2
* item3
      o item3-1
      o item3-2
      o item3-3
            + item3-3-1
            + item3-3-2
* item4
```

See also `{function}`

{capture}

`{capture}` is used to collect the output of the template between the tags into a variable instead of displaying it. Any content between `{capture name='foo'}` and `{/capture}` is collected into the variable specified in the *`name`* attribute.

The captured content can be used in the template from the variable *`$smarty.capture.foo`* where “foo” is the value passed in the *`name`* attribute. If you do not supply the *`name`* attribute, then “default” will be used as the name ie *`$smarty.capture.default`*.

`{capture}'s` can be nested.

Attributes:

Attribute Name	Type	Required	Default	Description
name	string	Yes	*n/a*	The name of the captured block
assign	string	No	*n/a*	The variable name where to assign the captured output to
append	string	No	*n/a*	The name of an array variable where to append the captured output to

Option Flags:

Name	Description
nocache	Disables caching of this captured block

Caution

Be careful when capturing `{insert}` output. If you have *`$caching`* enabled and you have `{insert}` commands that you expect to run within cached content, do not capture this content.

Example 7.21. {capture} with the name attribute

```
{* we don't want to print a div tag unless content is displayed *}
{capture name="banner"}
{capture "banner"} {* short-hand *}
  {include file="get_banner.tpl"}
{/capture}

{if $smarty.capture.banner ne ""}
<div id="banner">{$smarty.capture.banner}</div>
{/if}
```

Example 7.22. {capture} into a template variable

This example demonstrates the capture function.

```
{capture name=some_content assign=popText}
{capture some_content assign=popText} {* short-hand *}
The server is {$my_server_name|upper} at {$my_server_addr}<br>
Your ip is {$my_ip}.
{/capture}
<a href="#">{$popText}</a>
```

Example 7.23. {capture} into a template array variable

This example also demonstrates how multiple calls of capture can be used to create an array with captured content.

```
{capture append="foo"}hello{/capture}I say just {capture append="foo"}world{/captu
{foreach $foo as $text}{$text} {/foreach}
```

The above example will output:

```
I say just hello world
```

See also *`$smarty.capture`*, `{eval}`, `{fetch}`, `fetch()` and `{assign}`.

{config_load}

`{config_load}` is used for loading config *`#variables#`* from a configuration file into the template.

Attributes:

Attribute Name	Type	Required	Default	Description
file	string	Yes	*n/a*	The name of the config file to include
section	string	No	*n/a*	The name of the section to load
scope	string	no	*local*	How the scope of the loaded variables are treated, which must be one of local, parent or global. local

Attribute Name	Type	Required	Default	Description
				means variables are loaded into the local template context. parent means variables are loaded into both the local context and the parent template that called it. global means variables are available to all templates.

Example 7.24. {config_load}

The `example.conf` file.

```
#this is config file comment

# global variables
pageTitle = "Main Menu"
bodyBgColor = #000000
tableBgColor = #000000
rowBgColor = #00ff00

#customer variables section
[Customer]
pageTitle = "Customer Info"
```

and the template

```
{config_load file="example.conf"}
{config_load "example.conf"}  {* short-hand *}

<html>
<title>{#pageTitle#|default:"No title"}</title>
<body bgcolor="{#bodyBgColor#}">
<table border="{#tableBorderSize#}" bgcolor="{#tableBgColor#}">
   <tr bgcolor="{#rowBgColor#}">
      <td>First</td>
      <td>Last</td>
      <td>Address</td>
   </tr>
</table>
</body>
</html>
```

Config Files may also contain sections. You can load variables from within a section with the added attribute *`section`*. Note that global config variables are always loaded along with section variables, and same-named section variables overwrite the globals.

Note

Config file *sections* and the built-in template function called `{section}` have nothing to do with each other, they just happen to share a common naming convention.

Example 7.25. function {config_load} with section

```
{config_load file='example.conf' section='Customer'}
{config_load 'example.conf' 'Customer'} {* short-hand *}

<html>
<title>{#pageTitle#}</title>
<body bgcolor="{#bodyBgColor#}">
<table border="{#tableBorderSize#}" bgcolor="{#tableBgColor#}">
   <tr bgcolor="{#rowBgColor#}">
      <td>First</td>
      <td>Last</td>
      <td>Address</td>
   </tr>
</table>
</body>
</html>
```

See *$config_overwrite* to create arrays of config file variables.

See also the config files page, config variables page, *$config_dir*, `get_config_vars()` and `config_load()`.

{debug}

`{debug}` dumps the debug console to the page. This works regardless of the debug settings in the php script. Since this gets executed at runtime, this is only able to show the assigned variables; not the templates that are in use. However, you can see all the currently available variables within the scope of a template.

If caching is enabled and a page is loaded from cache `{debug}` does show only the variables which assigned for the cached page.

See also the debugging console page.

{extends}

`{extends}` tags are used in child templates in template inheritance for extending parent templates. For details see section of Template Interitance.

- The `{extends}` tag must be on the first line of the template.

- If a child template extends a parent template with the `{extends}` tag it may contain only `{block}` tags. Any other template content is ignored.

- Use the syntax for template resources to extend files outside of the *$template_dir* directory.

Attributes:

Attribute Name	Type	Required	Default	Description
file	string	Yes	*n/a*	The name of the template file which is extended

Example 7.26. Simple {extends} example

```
{extends file='parent.tpl'}
{extends 'parent.tpl'}  {* short-hand *}
```

See also Template Interitance and `{block}`.

{for}

The `{for}{forelse}` tag is used to create simple loops. The following different formarts are supported:

- `{for $var=$start to $end}` simple loop with step size of 1.
- `{for $var=$start to $end step $step}` loop with individual step size.

`{forelse}` is executed when the loop is not iterated.

Attributes:

Attribute Name	Shorthand	Type	Required	Default	Description
max	n/a	integer	No	*n/a*	Limit the number of iterations

Option Flags:

Name	Description
nocache	Disables caching of the `{for}` loop

Example 7.27. A simple `{for}` loop

```
<ul>
{for $foo=1 to 3}
    <li>{$foo}</li>
{/for}
</ul>
```

The above example will output:

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

Example 7.28. Using the `max` attribute

```
$smarty->assign('to',10);
```

```
<ul>
{for $foo=3 to $to max=3}
    <li>{$foo}</li>
{/for}
</ul>
```

The above example will output:

```
<ul>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
```

Example 7.29. Excution of `{forelse}`

```
$smarty->assign('start',10);
$smarty->assign('to',5);
```

```
<ul>
{for $foo=$start to $to}
    <li>{$foo}</li>
{forelse}
  no iteration
{/for}
</ul>
```

The above example will output:

```
  no iteration
```

See also `{foreach}` and `{section}`

{foreach},{foreachelse}

`{foreach}` is used for looping over arrays of data. `{foreach}` has a simpler and cleaner syntax than the `{section}` loop, and can also loop over associative arrays.

`{foreach $arrayvar as $itemvar}`

`{foreach $arrayvar as $keyvar=>$itemvar}`

Note

This foreach syntax does not accept any named attributes. This syntax is new to Smarty 3, however the Smarty 2.x syntax `{foreach from=$myarray key="mykey" item="myitem"}` is still supported.

- `{foreach}` loops can be nested.
- The *`array`* variable, usually an array of values, determines the number of times `{foreach}` will loop. You can also pass an integer for arbitrary loops.
- `{foreachelse}` is executed when there are no values in the *`array`* variable.
- `{foreach}` properties are *`@index`*, *`@iteration`*, *`@first`*, *`@last`*, *`@show`*, *`@total`*.

- Instead of specifying the *key* variable you can access the current key of the loop item by *{$item@key}* (see examples below).

Note

The `$var@property` syntax is new to Smarty 3, however when using the Smarty 2 `{foreach from=$myarray key="mykey" item="myitem"}` style syntax, the `$smarty.foreach.name.property` syntax is still supported.

Note

Although you can retrieve the array key with the syntax `{foreach $myArray as $myKey => $myValue}`, the key is always available as `$myValue@key` within the foreach loop.

Option Flags:

Name	Description
nocache	Disables caching of the `{foreach}` loop

Example 7.30. A simple {foreach} loop

```
<?php
$arr = array('red', 'green', 'blue');
$smarty->assign('myColors', $arr);
?>
```

Template to output *$myColors* in an un-ordered list

```
<ul>
{foreach $myColors as $color}
    <li>{$color}</li>
{/foreach}
</ul>
```

The above example will output:

```
<ul>
    <li>red</li>
    <li>green</li>
    <li>blue</li>
</ul>
```

Example 7.31. Demonstrates the an additional *key* variable

```
<?php
$people = array('fname' => 'John', 'lname' => 'Doe', 'email' => 'j.doe@example.com
$smarty->assign('myPeople', $people);
?>
```

Template to output *$myArray* as key/value pairs.

```
<ul>
{foreach $myPeople as $value}
   <li>{$value@key}: {$value}</li>
{/foreach}
</ul>
```

The above example will output:

```
<ul>
    <li>fname: John</li>
    <li>lname: Doe</li>
    <li>email: j.doe@example.com</li>
</ul>
```

Example 7.32. {foreach} with nested *item* and *key*

Assign an array to Smarty, the key contains the key for each looped value.

```
<?php
 $smarty->assign('contacts', array(
                             array('phone' => '555-555-1234',
                                   'fax' => '555-555-5678',
                                   'cell' => '555-555-0357'),
                             array('phone' => '800-555-4444',
                                   'fax' => '800-555-3333',
                                   'cell' => '800-555-2222')
                             ));
?>
```

The template to output *$contact*.

```
{* key always available as a property *}
{foreach $contacts as $contact}
  {foreach $contact as $value}
    {$value@key}: {$value}
  {/foreach}
{/foreach}

{* accessing key the PHP syntax alternate *}
{foreach $contacts as $contact}
  {foreach $contact as $key => $value}
    {$key}: {$value}
  {/foreach}
{/foreach}
```

Either of the above examples will output:

```
  phone: 555-555-1234
  fax: 555-555-5678
  cell: 555-555-0357
  phone: 800-555-4444
  fax: 800-555-3333
  cell: 800-555-2222
```

Example 7.33. Database example with {foreachelse}

A database (PDO) example of looping over search results. This example is looping over a PHP iterator instead of an array().

```
<?php
  include('Smarty.class.php');

  $smarty = new Smarty;

  $dsn = 'mysql:host=localhost;dbname=test';
  $login = 'test';
  $passwd = 'test';

  // setting PDO to use buffered queries in mysql is
  // important if you plan on using multiple result cursors
  // in the template.

  $db = new PDO($dsn, $login, $passwd, array(
     PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true));

  $res = $db->prepare("select * from users");
  $res->execute();
  $res->setFetchMode(PDO::FETCH_LAZY);

  // assign to smarty
  $smarty->assign('res',$res);

  $smarty->display('index.tpl');?>
?>
```

```
{foreach $res as $r}
  {$r.id}
  {$r.name}
{foreachelse}
  .. no results ..
{/foreach}
```

The above is assuming the results contain the columns named `id` and `name`.

What is the advantage of an iterator vs. looping over a plain old array? With an array, all the results are accumulated into memory before being looped. With an iterator, each result is loaded/released within the loop. This saves processing time and memory, especially for very large result sets.

@index

`index` contains the current array index, starting with zero.

Example 7.34. *`index`* example

```
{* output empty row on the 4th iteration (when index is 3) *}
<table>
{foreach $items as $i}
  {if $i@index eq 3}
     {* put empty table row *}
     <tr><td>nbsp;</td></tr>
  {/if}
  <tr><td>{$i.label}</td></tr>
{/foreach}
</table>
```

@iteration

`iteration` contains the current loop iteration and always starts at one, unlike *`index`*. It is incremented by one on each iteration.

Example 7.35. *`iteration`* example: is div by

The *"is div by"* operator can be used to detect a specific iteration. Here we bold-face the name every 4th iteration.

```
{foreach $myNames as $name}
  {if $name@iteration is div by 4}
    <b>{$name}</b>
  {/if}
  {$name}
{/foreach}
```

Example 7.36. *iteration* example: is even/odd by

The *"is even by"* and *"is odd by"* operators can be used to alternate something every so many iterations. Choosing between even or odd rotates which one starts. Here we switch the font color every 3rd iteration.

```
{foreach $myNames as $name}
  {if $name@iteration is even by 3}
    <span style="color: #000">{$name}</span>
  {else}
    <span style="color: #eee">{$name}</span>
  {/if}
{/foreach}
```

This will output something similar to this:

```
    <span style="color: #000">...</span>
    <span style="color: #000">...</span>
    <span style="color: #000">...</span>
    <span style="color: #eee">...</span>
    <span style="color: #eee">...</span>
    <span style="color: #eee">...</span>
    <span style="color: #000">...</span>
    <span style="color: #000">...</span>
    <span style="color: #000">...</span>
    <span style="color: #eee">...</span>
    <span style="color: #eee">...</span>
    <span style="color: #eee">...</span>
    ...
```

@first

first is TRUE if the current {foreach} iteration is the initial one. Here we display a table header row on the first iteration.

Example 7.37. *`first`* property example

```
{* show table header at first iteration *}
<table>
{foreach $items as $i}
  {if $i@first}
    <tr>
      <th>key</td>
      <th>name</td>
    </tr>
  {/if}
  <tr>
    <td>{$i@key}</td>
    <td>{$i.name}</td>
  </tr>
{/foreach}
</table>
```

@last

`last` is set to `TRUE` if the current `{foreach}` iteration is the final one. Here we display a horizontal rule on the last iteration.

Example 7.38. *`last`* property example

```
{* Add horizontal rule at end of list *}
{foreach $items as $item}
  <a href="#{$item.id}">{$item.name}</a>{if $prod@last}<hr>{else},{/if}
{foreachelse}
  ... no items to loop ...
{/foreach}
```

@show

The show *`show`* property can be used after the execution of a `{foreach}` loop to detect if data has been displayed or not. *`show`* is a boolean value.

Example 7.39. *`show`* property example

```
<ul>
{foreach $myArray as $name}
    <li>{$name}</li>
{/foreach}
</ul>
{if $name@show} do something here if the array contained data {/if}
```

@total

`total` contains the number of iterations that this `{foreach}` will loop. This can be used inside or after the `{foreach}`.

Example 7.40. *`total`* property example

```
{* show number of rows at end *}
{foreach $items as $item}
  {$item.name}<hr/>
  {if $prod@last}
    <div id="total">{$prod@total} items</div>
  {/if}
{foreachelse}
 ... no items to loop ...
{/foreach}
```

See also `{section}` and `{for}`.

{function}

`{function}` is used to create functions within a template and call them just like a plugin function. Instead of writing a plugin that generates presentational content, keeping it in the template is often a more manageable choice. It also simplifies data traversal, such as deeply nested menus.

> ### Note
>
> Template functions are defined global. Since the Smarty compiler is a single-pass compiler, The `{call}` tag must be used to call a template function defined externally from the given template. Otherwise you can directly use the function as `{funcname ...}` in the template.

- The `{function}` tag must have the *`name`* attribute which contains the the name of the template function. A tag with this name can be used to call the template function.
- Default values for variables can be passed to the template function as attributes. The default values can be overwritten when the template function is being called..
- You can use all variables from the calling template inside the template function. Changes to variables or new created variables inside the template function have local scope and are not visible inside the calling template after the template function is executed.

Attributes:

Attribute Name	Type	Required	Default	Description
name	string	Yes	*n/a*	The name of the template function
[var ...]	[var type]	No	*n/a*	default variable value to pass local to the template function

Example 7.41. Recursive menu {function} example

```
{* define the function *}
{function name=menu level=0}
{function menu level=0}          {* short-hand *}
  <ul class="level{$level}">
  {foreach $data as $entry}
    {if is_array($entry)}
      <li>{$entry@key}</li>
      {menu data=$entry level=$level+1}
    {else}
      <li>{$entry}</li>
    {/if}
  {/foreach}
  </ul>
{/function}

{* create an array to demonstrate *}
{$menu = ['item1','item2','item3' => ['item3-1','item3-2','item3-3' =>
['item3-3-1','item3-3-2']],'item4']}

{* run the array through the function *}
{menu data=$menu}
```

Will generate the following output

```
* item1
* item2
* item3
      o item3-1
      o item3-2
      o item3-3
            + item3-3-1
            + item3-3-2
* item4
```

See also `{call}`

{if},{elseif},{else}

`{if}` statements in Smarty have much the same flexibility as PHP if [http://php.net/if] statements, with a few added features for the template engine. Every `{if}` must be paired with a matching `{/if}`. `{else}` and `{elseif}` are also permitted. All PHP conditionals and functions are recognized, such as ||, *or*, *&&*, *and*, *is_array()*, etc.

If secutry is enabled, only PHP functions from *`$php_functions`* property of the secutry policy are allowed. See the Security section for details.

The following is a list of recognized qualifiers, which must be separated from surrounding elements by spaces. Note that items listed in [brackets] are optional. PHP equivalents are shown where applicable.

Qualifier	Alternates	Syntax Example	Meaning	PHP Equivalent
==	eq	$a eq $b	equals	==
!=	ne, neq	$a neq $b	not equals	!=
>	gt	$a gt $b	greater than	>
<	lt	$a lt $b	less than	<
>=	gte, ge	$a ge $b	greater than or equal	>=
<=	lte, le	$a le $b	less than or equal	<=
===		$a === 0	check for identity	===
!	not	not $a	negation (unary)	!
%	mod	$a mod $b	modulous	%
is [not] div by		$a is not div by 4	divisible by	$a % $b == 0
is [not] even		$a is not even	[not] an even number (unary)	$a % 2 == 0
is [not] even by		$a is not even by $b	grouping level [not] even	($a / $b) % 2 == 0
is [not] odd		$a is not odd	[not] an odd number (unary)	$a % 2 != 0
is [not] odd by		$a is not odd by $b	[not] an odd grouping	($a / $b) % 2 != 0

```
{* same as above *}
{if $name == 'Fred' || $name == 'Wilma'}
   ...
{/if}
```

Example 7.42. {if} statements

```
{* parenthesis are allowed *}
{if ( $amount < 0 or $amount > 1000 ) and $volume >= #minVolAmt#}
   ...
{/if}


{* you can also embed php function calls *}
{if count($var) gt 0}
   ...
{/if}

{* check for array. *}
{if is_array($foo) }
   .....
{/if}

{* check for not null. *}
{if isset($foo) }
   .....
{/if}


{* test if values are even or odd *}
{if $var is even}
   ...
{/if}
{if $var is odd}
   ...
{/if}
{if $var is not odd}
   ...
{/if}


{* test if var is divisible by 4 *}
{if $var is div by 4}
   ...
{/if}


{*
  test if var is even, grouped by two. i.e.,
  0=even, 1=even, 2=odd, 3=odd, 4=even, 5=even, etc.
*}
{if $var is even by 2}
   ...
{/if}

{* 0=even, 1=even, 2=even, 3=odd, 4=odd, 5=odd, etc. *}
{if $var is even by 3}
   ...
{/if}
```

Example 7.43. {if} with more examples

```
{if isset($name) && $name == 'Blog'}
     {* do something *}
{elseif $name == $foo}
    {* do something *}
{/if}

{if is_array($foo) && count($foo) > 0}
    {* do a foreach loop *}
{/if}
```

{include}

`{include}` tags are used for including other templates in the current template. Any variables available in the current template are also available within the included template.

- The `{include}` tag must have the *`file`* attribute which contains the template resource path.
- Setting the optional *`assign`* attribute specifies the template variable that the output of `{include}` is assigned to, instead of being displayed. Similar to `{assign}`.
- Variables can be passed to included templates as attributes. Any variables explicitly passed to an included template are only available within the scope of the included file. Attribute variables override current template variables, in the case when they are named the same.
- You can use all variables from the including template inside the included template. But changes to variables or new created variables inside the included template have local scope and are not visible inside the including template after the `{include}` statement. This default behaviour can be changed for all variables assigned in the included template by using the scope attribute at the `{include}` statement or for individual variables by using the scope attribute at the `{assign}` statement. The later is usefull to return values from the inculded template to the includung template.
- Use the syntax for template resources to `{include}` files outside of the *`$template_dir`* directory.

Attributes:

Attribute Name	Type	Required	Default	Description
file	string	Yes	*n/a*	The name of the template file to include
assign	string	No	*n/a*	The name of the variable that the output of include will be assigned to
cache_lifetime	integer	No	*n/a*	Enable caching of this subtemplate with an individual cache lifetime

Attribute Name	Type	Required	Default	Description
compile_id	string/integer	No	*n/a*	Compile this subtemplate with an individual compile_id
cache_id	string/integer	No	*n/a*	Enable caching of this subtemplate with an individual cache_id
scope	string	No	*n/a*	Define the scope of all in the subtemplate assigned variables: 'parent','root' or 'global'
[var ...]	[var type]	No	*n/a*	variable to pass local to template

Option Flags:

Name	Description
nocache	Disables caching of this subtemplate
caching	Enable caching of this subtemplate
inline	If set merge the compile code of the subtemplate into the compiled calling template

Example 7.44. Simple {include} example

```
<html>
<head>
  <title>{$title}</title>
</head>
<body>
{include file='page_header.tpl'}

{* body of template goes here, the $tpl_name variable
   is replaced with a value eg 'contact.tpl'
*}
{include file="$tpl_name.tpl"}

{* using shortform file attribute *}
{include 'page_footer.tpl'}
</body>
</html>
```

Example 7.45. {include} passing variables

```
{include 'links.tpl' title='Newest links' links=$link_array}
{* body of template goes here *}
{include 'footer.tpl' foo='bar'}
```

The template above includes the example `links.tpl` below

```
<div id="box">
<h3>{$title}{/h3>
<ul>
{foreach from=$links item=l}
.. do stuff  ...
</foreach}
</ul>
</div>
```

Example 7.46. {include} using parent scope

Variables assigned in the included template will be seen in the including template.

```
{include 'sub_template.tpl' scope=parent}
...
{* display variables assigned in sub_template *}
{$foo}<br>
{$bar}<br>
...
```

The template above includes the example `sub_template.tpl` below

```
...
{assign var=foo value='something'}
{assign var=bar value='value'}
...
```

Example 7.47. {include} with disabled caching

The included template will not be cached.

```
{include 'sub_template.tpl' nocache}
...
```

Example 7.48. {include} with individual cache lifetime

In this example included template will be cached with an individual cache lifetime of 500 seconds.

```
{include 'sub_template.tpl' cache_lifteime=500}
...
```

Example 7.49. {include} with forced caching

In this example included template will be cached independent of the global cahing setting.

```
{include 'sub_template.tpl' caching}
...
```

Example 7.50. {include} and assign to variable

This example assigns the contents of `nav.tpl` to the `$navbar` variable, which is then output at both the top and bottom of the page.

```
<body>
  {include 'nav.tpl' assign=navbar}
  {include 'header.tpl' title='Smarty is cool'}
    {$navbar}
    {* body of template goes here *}
    {$navbar}
  {include 'footer.tpl'}
</body>
```

Example 7.51. Various {include} resource examples

```
{* absolute filepath *}
{include file='/usr/local/include/templates/header.tpl'}

{* absolute filepath (same thing) *}
{include file='file:/usr/local/include/templates/header.tpl'}

{* windows absolute filepath (MUST use "file:" prefix) *}
{include file='file:C:/www/pub/templates/header.tpl'}

{* include from template resource named "db" *}
{include file='db:header.tpl'}

{* include a $variable template - eg $module = 'contacts' *}
{include file="$module.tpl"}

{* wont work as its single quotes ie no variable substitution *}
{include file='$module.tpl'}

{* include a multi $variable template - eg amber/links.view.tpl *}
{include file="$style_dir/$module.$view.tpl"}
```

See also `{include_php}`, `{insert}`, `{php}`, template resources and componentized templates.

{include_php}

IMPORTANT NOTICE

`{include_php}` is deprecated from Smarty, use registered plugins to properly insulate presentation from the application code.

Attribute Name	Type	Required	Default	Description
file	string	Yes	*n/a*	The name of the php file to include as absolute path
once	boolean	No	*`TRUE`*	whether or not to include the php file more than once if included multiple times
assign	string	No	*n/a*	The name of the variable that the output of include_php will be assigned to

Option Flags:

Name	Description
nocache	Disables caching of inluded PHP script

`{include_php}` tags are used to include a php script in your template. The path of the attribute *`file`* can be either absolute, or relative to *`$trusted_dir`*. If security is enabled, then the script must be located in the *`$trusted_dir`* path of the securty policy. See the Security section for details.

By default, php files are only included once even if called multiple times in the template. You can specify that it should be included every time with the *`once`* attribute. Setting once to `FALSE` will include the php script each time it is included in the template.

You can optionally pass the *`assign`* attribute, which will specify a template variable name that the output of `{include_php}` will be assigned to instead of displayed.

The smarty object is available as *`$_smarty_tpl->smarty`* within the PHP script that you include.

Example 7.52. function {include_php}

The `load_nav.php` file:

```
<?php

// load in variables from a mysql db and assign them to the template
require_once('database.class.php');
$db = new Db();
$db->query('select url, name from navigation order by name');
$this->assign('navigation', $db->getRows());

?>
```

where the template is:

```
{* absolute path, or relative to $trusted_dir *}
{include_php file='/path/to/load_nav.php'}
{include_php '/path/to/load_nav.php'}             {* short-hand *}

{foreach item='nav' from=$navigation}
  <a href="{$nav.url}">{$nav.name}</a><br />
{/foreach}
```

See also `{include}`, *`$trusted_dir`*, `{php}`, `{capture}`, template resources and componentized templates

{insert}

`{insert}` tags work much like `{include}` tags, except that `{insert}` tags are NOT cached when template caching is enabled. They will be executed on every invocation of the template.

Attribute Name	Type	Required	Default	Description
name	string	Yes	*n/a*	The name of the insert function (insert_*name*) or insert plugin
assign	string	No	*n/a*	The name of the template variable the output will be assigned to
script	string	No	*n/a*	The name of the php script that is included before the insert function is called
[var ...]	[var type]	No	*n/a*	variable to pass to insert function

Let's say you have a template with a banner slot at the top of the page. The banner can contain any mixture of HTML, images, flash, etc. so we can't just use a static link here, and we don't want this contents cached with the page. In comes the {insert} tag: the template knows #banner_location_id# and #site_id# values (gathered from a config file), and needs to call a function to get the banner contents.

Example 7.53. function {insert}

```
{* example of fetching a banner *}
{insert name="getBanner" lid=#banner_location_id# sid=#site_id#}
{insert "getBanner" lid=#banner_location_id# sid=#site_id#} {* short-hand *}
```

In this example, we are using the name “getBanner” and passing the parameters #banner_location_id# and #site_id#. Smarty will look for a function named insert_getBanner() in your PHP application, passing the values of #banner_location_id# and #site_id# as the first argument in an associative array. All {insert} function names in your application must be prepended with "insert_" to remedy possible function namespace conflicts. Your insert_getBanner() function should do something with the passed values and return the results. These results are then displayed in the template in place of the {insert} tag. In this example, Smarty would call this function: insert_getBanner(array("lid" => "12345","sid" => "67890")); and display the returned results in place of the {insert} tag.

- If you supply the *`assign`* attribute, the output of the `{insert}` tag will be assigned to this template variable instead of being output to the template.

  ## Note

  Assigning the output to a template variable isn't too useful with caching enabled.

- If you supply the *`script`* attribute, this php script will be included (only once) before the `{insert}` function is executed. This is the case where the insert function may not exist yet, and a php script must be included first to make it work.

  The path can be either absolute, or relative to *`$trusted_dir`*. If security is enabled, then the script must be located in the *`$trusted_dir`* path of the securty policy. See the Security section for details.

The Smarty object is passed as the second argument. This way you can reference and modify information in the Smarty object from within the `{insert}` function.

If no PHP script can be found Smarty is looking for a corresponding insert plugin.

Technical Note

It is possible to have portions of the template not cached. If you have caching turned on, `{insert}` tags will not be cached. They will run dynamically every time the page is created, even within cached pages. This works good for things like banners, polls, live weather, search results, user feedback areas, etc.

See also `{include}`

{ldelim},{rdelim}

`{ldelim}` and `{rdelim}` are used for escaping template delimiters, by default **{** and **}**. You can also use `{literal}{/literal}` to escape blocks of text eg Javascript or CSS. See also the complimentary *`{$smarty.ldelim}`*.

Example 7.54. {ldelim}, {rdelim}

```
{* this will print literal delimiters out of the template *}

{ldelim}funcname{rdelim} is how functions look in Smarty!
```

The above example will output:

```
{funcname} is how functions look in Smarty!
```

Another example with some Javascript

```
<script language="JavaScript">
function foo() {ldelim}
    ... code ...
{rdelim}
</script>
```

will output

```
<script language="JavaScript">
function foo() {
    .... code ...
}
</script>
```

Example 7.55. Another Javascript example

```
<script language="JavaScript" type="text/javascript">
    function myJsFunction(){ldelim}
        alert("The server name\n{$smarty.server.SERVER_NAME}\n{$smarty.server.SERV
    {rdelim}
</script>
<a href="javascript:myJsFunction()">Click here for Server Info</a>
```

See also `{literal}` and escaping Smarty parsing.

{literal}

`{literal}` tags allow a block of data to be taken literally. This is typically used around Javascript or stylesheet blocks where {curly braces} would interfere with the template delimiter syntax. Anything within `{literal}{/literal}` tags is not interpreted, but displayed as-is. If you need template tags embedded in a `{literal}` block, consider using `{ldelim}{rdelim}` to escape the individual delimiters instead.

Note

`{literal}{/literal}` tags are normally not necessary, as Smarty ignores delimiters that are surrounded by whitespace. Be sure your javascript and CSS curly braces are surrounded by whitespace. This is new behavior to Smarty 3.

Example 7.56. {literal} tags

```
<script>
   // the following braces are ignored by Smarty
   // since they are surrounded by whitespace
   function myFoo {
     alert('Foo!');
   }
   // this one will need literal escapement
   {literal}
     function myBar {alert('Bar!');}
   {/literal}
</script>
```

See also `{ldelim} {rdelim}` and the escaping Smarty parsing page.

{nocache}

`{nocache}` is used to disable caching of a template section. Every `{nocache}` must be paired with a matching `{/nocache}`.

Note

Be sure any variables used within a non-cached section are also assigned from PHP when the page is loaded from the cache.

Example 7.57. Preventing a template section from being cached

```
Today's date is
{nocache}
{$smarty.now|date_format}
{/nocache}
```

The above code will output the current date on a cached page.

See also the caching section.

{php}

IMPORTANT NOTICE

`{php}` tags are deprecated from Smarty, and should not be used. Put your PHP logic in PHP scripts or plugin functions instead.

The `{php}` tags allow PHP code to be embedded directly into the template. They will not be escaped, regardless of the *`$php_handling`* setting.

Example 7.58. php code within {php} tags

```
{php}
   // including a php script directly from the template.
   include('/path/to/display_weather.php');
{/php}
```

Example 7.59. {php} tags with global and assigning a variable

```
{* this template includes a {php} block that assign's the variable $varX *}
{php}
   global $foo, $bar;
   if($foo == $bar){
     echo 'This will be sent to browser';
   }
  // assign a variable to Smarty
  $this->assign('varX','Toffee');
{/php}
{* output the variable *}
<strong>{$varX}</strong> is my fav ice cream :-)
```

See also *`$php_handling`*, `{include_php}`, `{include}`, `{insert}` and componentized templates.

{section},{sectionelse}

A `{section}` is for looping over **sequentially indexed arrays of data**, unlike `{foreach}` which is used to loop over a **single associative array**. Every `{section}` tag must be paired with a closing `{/section}` tag.

Note

The `{foreach}` loop can do everything a {section} loop can do, and has a simpler and easier syntax. It is usually preferred over the {section} loop.

Note

{section} loops cannot loop over associative arrays, they must be numerically indexed, and sequential (0,1,2,...). For associative arrays, use the `{foreach}` loop.

Attribute Name	Type	Required	Default	Description
name	string	Yes	*n/a*	The name of the section
loop	mixed	Yes	*n/a*	Value to determine the number of loop iterations
start	integer	No	*0*	The index position that the section will begin looping. If the value is negative, the start position is calculated from the end of the array. For example, if there are seven values in the loop array and start is -2, the start index is 5. Invalid values (values outside of the length of the loop array) are automatically truncated to the closest valid value.
step	integer	No	*1*	The step value that will be used to traverse the loop array. For example, step=2 will loop on index 0,2,4,

Attribute Name	Type	Required	Default	Description
				etc. If step is negative, it will step through the array backwards.
max	integer	No	*n/a*	Sets the maximum number of times the section will loop.
show	boolean	No	*`TRUE`*	Determines whether or not to show this section

Option Flags:

Name	Description
nocache	Disables caching of the `{section}` loop

- Required attributes are *`name`* and *`loop`*.

- The *`name`* of the `{section}` can be anything you like, made up of letters, numbers and underscores, like PHP variables [http://php.net/language.variables].

- {section}'s can be nested, and the nested `{section}` names must be unique from each other.

- The *`loop`* attribute, usually an array of values, determines the number of times the `{section}` will loop. You can also pass an integer as the loop value.

- When printing a variable within a `{section}`, the `{section}` *`name`* must be given next to variable name within [brackets].

- `{sectionelse}` is executed when there are no values in the loop variable.

- A `{section}` also has its own variables that handle `{section}` properties. These properties are accessible as: *`{$smarty.section.name.property}`* where “name” is the attribute *`name`*.

- `{section}` properties are *`index`*, *`index_prev`*, *`index_next`*, *`iteration`*, *`first`*, *`last`*, *`rownum`*, *`loop`*, *`show`*, *`total`*.

Example 7.60. Looping a simple array with {section}

`assign()` an array to Smarty

```
<?php
$data = array(1000,1001,1002);
$smarty->assign('custid',$data);
?>
```

The template that outputs the array

```
{* this example will print out all the values of the $custid array *}
{section name=customer loop=$custid}
{section customer $custid} {* short-hand *}
  id: {$custid[customer]}<br />
{/section}
<hr />
{*  print out all the values of the $custid array reversed *}
{section name=foo loop=$custid step=-1}
{section foo $custid step=-1} {* short-hand *}
  {$custid[foo]}<br />
{/section}
```

The above example will output:

```
id: 1000<br />
id: 1001<br />
id: 1002<br />
<hr />
id: 1002<br />
id: 1001<br />
id: 1000<br />
```

Example 7.61. {section} without an assigned array

```
{section name=foo start=10 loop=20 step=2}
  {$smarty.section.foo.index}
{/section}
<hr />
{section name=bar loop=21 max=6 step=-2}
  {$smarty.section.bar.index}
{/section}
```

The above example will output:

```
10 12 14 16 18
<hr />
20 18 16 14 12 10
```

Example 7.62. Naming a {section}

The *`name`* of the `{section}` can be anything you like, see PHP variables [http://php.net/language.variables]. It is used to reference the data within the `{section}`.

```
{section name=anything loop=$myArray}
  {$myArray[anything].foo}
  {$name[anything]}
  {$address[anything].bar}
{/section}
```

Example 7.63. Looping an associative array with {section}

This is an example of printing an associative array of data with a `{section}`. Following is the php script to assign the *`$contacts`* array to Smarty.

```
<?php
$data = array(
          array('name' => 'John Smith', 'home' => '555-555-5555',
                'cell' => '666-555-5555', 'email' => 'john@myexample.com'),
          array('name' => 'Jack Jones', 'home' => '777-555-5555',
                'cell' => '888-555-5555', 'email' => 'jack@myexample.com'),
          array('name' => 'Jane Munson', 'home' => '000-555-5555',
                'cell' => '123456', 'email' => 'jane@myexample.com')
        );
$smarty->assign('contacts',$data);
?>
```

The template to output *`$contacts`*

```
{section name=customer loop=$contacts}
<p>
  name: {$contacts[customer].name}<br />
  home: {$contacts[customer].home}<br />
  cell: {$contacts[customer].cell}<br />
  e-mail: {$contacts[customer].email}
</p>
{/section}
```

The above example will output:

```
<p>
  name: John Smith<br />
  home: 555-555-5555<br />
  cell: 666-555-5555<br />
  e-mail: john@myexample.com
</p>
<p>
  name: Jack Jones<br />
  home phone: 777-555-5555<br />
  cell phone: 888-555-5555<br />
  e-mail: jack@myexample.com
</p>
<p>
  name: Jane Munson<br />
  home phone: 000-555-5555<br />
  cell phone: 123456<br />
  e-mail: jane@myexample.com
</p>
```

Example 7.64. {section} demonstrating the `loop` variable

This example assumes that *`$custid`*, *`$name`* and *`$address`* are all arrays containing the same number of values. First the php script that assign's the arrays to Smarty.

```
<?php

$id = array(1001,1002,1003);
$smarty->assign('custid',$id);

$fullnames = array('John Smith','Jack Jones','Jane Munson');
$smarty->assign('name',$fullnames);

$addr = array('253 Abbey road', '417 Mulberry ln', '5605 apple st');
$smarty->assign('address',$addr);

?>
```

The *`loop`* variable only determines the number of times to loop. You can access ANY variable from the template within the `{section}`. This is useful for looping multiple arrays. You can pass an array which will determine the loop count by the array size, or you can pass an integer to specify the number of loops.

```
{section name=customer loop=$custid}
<p>
  id: {$custid[customer]}<br />
  name: {$name[customer]}<br />
  address: {$address[customer]}
</p>
{/section}
```

The above example will output:

```
<p>
  id: 1000<br />
  name: John Smith<br />
  address: 253 Abbey road
</p>
<p>
  id: 1001<br />
  name: Jack Jones<br />
  address: 417 Mulberry ln
</p>
<p>
  id: 1002<br />
  name: Jane Munson<br />
  address: 5605 apple st
</p>
```

{section}'s can be nested as deep as you like. With nested {section}'s, you can access complex data structures, such as multi-dimensional arrays. This is an example `.php` script thats assign's the arrays.

Example 7.65. Nested {section}'s

```
$id = array(1001,1002,1003);
$smarty->assign('custid',$id);

$fullnames = array('John Smith','Jack Jones','Jane Munson');
$smarty->assign('name',$fullnames);

$addr = array('253 N 45th', '417 Mulberry ln', '5605 apple st');
$smarty->assign('address',$addr);

$types = array(
           array( 'home phone', 'cell phone', 'e-mail'),
           array( 'home phone', 'web'),
           array( 'cell phone')
         );
$smarty->assign('contact_type', $types);

$info = array(
           array('555-555-5555', '666-555-5555', 'john@myexample.com'),
           array( '123-456-4', 'www.example.com'),
           array( '0457878')
        );
$smarty->assign('contact_info', $info);

?>
```

In this template, *$contact_type[customer]* is an array of contact types for the current customer.

```
{section name=customer loop=$custid}
<hr>
  id: {$custid[customer]}<br />
  name: {$name[customer]}<br />
  address: {$address[customer]}<br />
  {section name=contact loop=$contact_type[customer]}
    {$contact_type[customer][contact]}: {$contact_info[customer][contact]}<br />
  {/section}
{/section}
```

The above example will output:

```
<hr>
  id: 1000<br />
  name: John Smith<br />
  address: 253 N 45th<br />
    home phone: 555-555-5555<br />
    cell phone: 666-555-5555<br />
    e-mail: john@myexample.com<br />
<hr>
  id: 1001<br />
  name: Jack Jones<br />
  address: 417 Mulberry ln<br />
    home phone: 123-456-4<br />
    web: www.example.com<br />
<hr>
  id: 1002<br />
  name: Jane Munson<br />
  address: 5605 apple st<br />
    cell phone: 0457878<br />
```

Example 7.66. Database example with a {sectionelse}

Results of a database search (eg ADODB or PEAR) are assigned to Smarty

```
<?php
$sql = 'select id, name, home, cell, email from contacts '
      ."where name like '$foo%' ";
$smarty->assign('contacts', $db->getAll($sql));
?>
```

The template to output the database result in a HTML table

```
<table>
<tr><th>&nbsp;</th><th>Name></th><th>Home</th><th>Cell</th><th>Email</th></tr>
{section name=co loop=$contacts}
  <tr>
    <td><a href="view.php?id={$contacts[co].id}">view<a></td>
    <td>{$contacts[co].name}</td>
    <td>{$contacts[co].home}</td>
    <td>{$contacts[co].cell}</td>
    <td>{$contacts[co].email}</td>
  <tr>
{sectionelse}
  <tr><td colspan="5">No items found</td></tr>
{/section}
</table>
```

.index

`index` contains the current array index, starting with zero or the `start` attribute if given. It increments by one or by the `step` attribute if given.

Note

If the `step` and `start` properties are not modified, then this works the same as the `iteration` property, except it starts at zero instead of one.

Example 7.67. {section} `index` property

Note

`$custid[customer.index]` and `$custid[customer]` are identical.

```
{section name=customer loop=$custid}
  {$smarty.section.customer.index} id: {$custid[customer]}<br />
{/section}
```

The above example will output:

```
0 id: 1000<br />
1 id: 1001<br />
2 id: 1002<br />
```

.index_prev

`index_prev` is the previous loop index. On the first loop, this is set to -1.

.index_next

`index_next` is the next loop index. On the last loop, this is still one more than the current index, respecting the setting of the *`step`* attribute, if given.

Example 7.68. `index`, `index_next` and `index_prev` properties

```
<?php
$data = array(1001,1002,1003,1004,1005);
$smarty->assign('rows',$data);
?>
```

Template to output the above array in a table

```
{* $rows[row.index] and $rows[row] are identical in meaning *}
<table>
  <tr>
    <th>index</th><th>id</th>
    <th>index_prev</th><th>prev_id</th>
    <th>index_next</th><th>next_id</th>
  </tr>
{section name=row loop=$rows}
  <tr>
    <td>{$smarty.section.row.index}</td><td>{$rows[row]}</td>
    <td>{$smarty.section.row.index_prev}</td><td>{$rows[row.index_prev]}</td>
    <td>{$smarty.section.row.index_next}</td><td>{$rows[row.index_next]}</td>
  </tr>
{/section}
</table>
```

The above example will output a table containing the following:

```
index  id    index_prev prev_id index_next next_id
0      1001  -1                 1          1002
1      1002  0          1001    2          1003
2      1003  1          1002    3          1004
3      1004  2          1003    4          1005
4      1005  3          1004    5
```

.iteration

`iteration` contains the current loop iteration and starts at one.

Note

This is not affected by the `{section}` properties *`start`*, *`step`* and *`max`*, unlike the *`index`* property. *`iteration`* also starts with one instead of zero unlike *`index`*. *`rownum`* is an alias to *`iteration`*, they are identical.

Example 7.69. A section's `iteration` property

```
<?php
// array of 3000 to 3015
$id = range(3000,3015);
$smarty->assign('arr',$id);
?>
```

Template to output every other element of the `$arr` array as `step=2`

```
{section name=cu loop=$arr start=5 step=2}
  iteration={$smarty.section.cu.iteration}
  index={$smarty.section.cu.index}
  id={$custid[cu]}<br />
{/section}
```

The above example will output:

```
iteration=1 index=5 id=3005<br />
iteration=2 index=7 id=3007<br />
iteration=3 index=9 id=3009<br />
iteration=4 index=11 id=3011<br />
iteration=5 index=13 id=3013<br />
iteration=6 index=15 id=3015<br />
```

Another example that uses the *`iteration`* property to output a table header block every five rows.

```
<table>
{section name=co loop=$contacts}
  {if $smarty.section.co.iteration is div by 5}
    <tr><th>&nbsp;</th><th>Name></th><th>Home</th><th>Cell</th><th>Email</th></tr>
  {/if}
  <tr>
    <td><a href="view.php?id={$contacts[co].id}">view<a></td>
    <td>{$contacts[co].name}</td>
    <td>{$contacts[co].home}</td>
    <td>{$contacts[co].cell}</td>
    <td>{$contacts[co].email}</td>
  <tr>
{/section}
</table>
```

An that uses the *`iteration`* property to alternate a text color every third row.

```
  <table>
  {section name=co loop=$contacts}
    {if $smarty.section.co.iteration is even by 3}
      <span style="color: #ffffff">{$contacts[co].name}</span>
    {else}
      <span style="color: #dddddd">{$contacts[co].name}</span>
    {/if}
  {/section}
  </table>
```

Note

The *"is div by"* syntax is a simpler alternative to the PHP mod operator syntax. The mod operator is allowed: `{if $smarty.section.co.iteration % 5 == 1}` will work just the same.

Note

You can also use *"is odd by"* to reverse the alternating.

.first

`first` is set to `TRUE` if the current `{section}` iteration is the initial one.

.last

`last` is set to `TRUE` if the current section iteration is the final one.

Example 7.70. {section} property `first` and `last`

This example loops the `$customers` array, outputs a header block on the first iteration and on the last outputs the footer block. Also uses the *`total`* property.

```
{section name=customer loop=$customers}
  {if $smarty.section.customer.first}
    <table>
    <tr><th>id</th><th>customer</th></tr>
  {/if}

  <tr>
    <td>{$customers[customer].id}}</td>
    <td>{$customers[customer].name}</td>
  </tr>

  {if $smarty.section.customer.last}
    <tr><td></td><td>{$smarty.section.customer.total} customers</td></tr>
    </table>
  {/if}
{/section}
```

.rownum

`rownum` contains the current loop iteration, starting with one. It is an alias to *`iteration`*, they work identically.

.loop

`loop` contains the last index number that this {section} looped. This can be used inside or after the `{section}`.

Example 7.71. {section} property `loop`

```
{section name=customer loop=$custid}
  {$smarty.section.customer.index} id: {$custid[customer]}<br />
{/section}
There are {$smarty.section.customer.loop} customers shown above.
```

The above example will output:

```
0 id: 1000<br />
1 id: 1001<br />
2 id: 1002<br />
There are 3 customers shown above.
```

.show

`show` is used as a parameter to section and is a boolean value. If `FALSE`, the section will not be displayed. If there is a `{sectionelse}` present, that will be alternately displayed.

Example 7.72. `show` property

Boolean `$show_customer_info` has been passed from the PHP application, to regulate whether or not this section shows.

```
{section name=customer loop=$customers show=$show_customer_info}
  {$smarty.section.customer.rownum} id: {$customers[customer]}<br />
{/section}

{if $smarty.section.customer.show}
  the section was shown.
{else}
  the section was not shown.
{/if}
```

The above example will output:

```
1 id: 1000<br />
2 id: 1001<br />
3 id: 1002<br />

the section was shown.
```

.total

`total` contains the number of iterations that this `{section}` will loop. This can be used inside or after a `{section}`.

Example 7.73. `total` property example

```
{section name=customer loop=$custid step=2}
  {$smarty.section.customer.index} id: {$custid[customer]}<br />
{/section}
   There are {$smarty.section.customer.total} customers shown above.
```

See also `{foreach}`, `{for}` and *`$smarty.section`*.

{strip}

Many times web designers run into the issue where white space and carriage returns affect the output of the rendered HTML (browser "features"), so you must run all your tags together in the template to get the desired results. This usually ends up in unreadable or unmanageable templates.

Anything within `{strip}{/strip}` tags are stripped of the extra spaces or carriage returns at the beginnings and ends of the lines before they are displayed. This way you can keep your templates readable, and not worry about extra white space causing problems.

Note

`{strip}{/strip}` does not affect the contents of template variables, see the strip modifier instead.

Example 7.74. {strip} tags

```
{* the following will be all run into one line upon output *}
{strip}
<table border='0'>
 <tr>
  <td>
   <a href="{$url}">
    <font color="red">This is a test</font>
   </a>
  </td>
 </tr>
</table>
{/strip}
```

The above example will output:

```
<table border='0'><tr><td><a href="http://. snipped...</a></td></tr></table>
```

Notice that in the above example, all the lines begin and end with HTML tags. Be aware that all the lines are run together. If you have plain text at the beginning or end of any line, they will be run together, and may not be desired results.

See also the `strip` modifier.

{while}

`{while}` loopss in Smarty have much the same flexibility as PHP while [http://php.net/while] statements, with a few added features for the template engine. Every `{while}` must be paired with a matching `{/while}`. All PHP conditionals and functions are recognized, such as ||, *or*, *&&*, *and*, *is_array()*, etc.

The following is a list of recognized qualifiers, which must be separated from surrounding elements by spaces. Note that items listed in [brackets] are optional. PHP equivalents are shown where applicable.

Qualifier	Alternates	Syntax Example	Meaning	PHP Equivalent
==	eq	$a eq $b	equals	==

Qualifier	Alternates	Syntax Example	Meaning	PHP Equivalent
!=	ne, neq	$a neq $b	not equals	!=
>	gt	$a gt $b	greater than	>
<	lt	$a lt $b	less than	<
>=	gte, ge	$a ge $b	greater than or equal	>=
<=	lte, le	$a le $b	less than or equal	<=
===		$a === 0	check for identity	===
!	not	not $a	negation (unary)	!
%	mod	$a mod $b	modulous	%
is [not] div by		$a is not div by 4	divisible by	$a % $b == 0
is [not] even		$a is not even	[not] an even number (unary)	$a % 2 == 0
is [not] even by		$a is not even by $b	grouping level [not] even	($a / $b) % 2 == 0
is [not] odd		$a is not odd	[not] an odd number (unary)	$a % 2 != 0
is [not] odd by		$a is not odd by $b	[not] an odd grouping	($a / $b) % 2 != 0

Example 7.75. {while} loop

```
{while $foo > 0}
  {$foo--}
{/while}
```

The above example will count down the value of $foo until 1 is reached.

See also `{foreach}` and `{for}`.

Chapter 8. Custom Functions

Smarty comes with several custom plugin functions that you can use in the templates.

{counter}

`{counter}` is used to print out a count. `{counter}` will remember the count on each iteration. You can adjust the number, the interval and the direction of the count, as well as determine whether or not to print the value. You can run multiple counters concurrently by supplying a unique name for each one. If you do not supply a name, the name “default” will be used.

If you supply the *`assign`* attribute, the output of the `{counter}` function will be assigned to this template variable instead of being output to the template.

Attribute Name	Type	Required	Default	Description
name	string	No	*default*	The name of the counter
start	number	No	*1*	The initial number to start counting from
skip	number	No	*1*	The interval to count by
direction	string	No	*up*	The direction to count (up/down)
print	boolean	No	*`TRUE`*	Whether or not to print the value
assign	string	No	*n/a*	the template variable the output will be assigned to

Example 8.1. {counter}

```
{* initialize the count *}
{counter start=0 skip=2}<br />
{counter}<br />
{counter}<br />
{counter}<br />
```

this will output:

```
0<br />
2<br />
4<br />
6<br />
```

{cycle}

`{cycle}` is used to alternate a set of values. This makes it easy to for example, alternate between two or more colors in a table, or cycle through an array of values.

Attribute Name	Type	Required	Default	Description
name	string	No	*default*	The name of the cycle
values	mixed	Yes	*N/A*	The values to cycle through, either a comma delimited list (see delimiter attribute), or an array of values
print	boolean	No	*`TRUE`*	Whether to print the value or not
advance	boolean	No	*`TRUE`*	Whether or not to advance to the next value
delimiter	string	No	,	The delimiter to use in the values attribute
assign	string	No	*n/a*	The template variable the output will be assigned to
reset	boolean	No	*`FALSE`*	The cycle will be set to the first value and not advanced

- You can `{cycle}` through more than one set of values in a template by supplying a *`name`* attribute. Give each `{cycle}` an unique *`name`*.
- You can force the current value not to print with the *`print`* attribute set to `FALSE`. This would be useful for silently skipping a value.
- The *`advance`* attribute is used to repeat a value. When set to `FALSE`, the next call to `{cycle}` will print the same value.
- If you supply the *`assign`* attribute, the output of the `{cycle}` function will be assigned to a template variable instead of being output to the template.

Example 8.2. {cycle}

```
{section name=rows loop=$data}
<tr class="{cycle values="odd,even"}">
   <td>{$data[rows]}</td>
</tr>
{/section}
```

The above template would output:

```
<tr class="odd">
   <td>1</td>
</tr>
<tr class="even">
   <td>2</td>
</tr>
<tr class="odd">
   <td>3</td>
</tr>
```

{eval}

`{eval}` is used to evaluate a variable as a template. This can be used for things like embedding template tags/variables into variables or tags/variables into config file variables.

If you supply the *`assign`* attribute, the output of the `{eval}` function will be assigned to this template variable instead of being output to the template.

Attribute Name	Type	Required	Default	Description
var	mixed	Yes	*n/a*	Variable (or string) to evaluate
assign	string	No	*n/a*	The template variable the output will be assigned to

Technical Note

- Evaluated variables are treated the same as templates. They follow the same escapement and security features just as if they were templates.
- Evaluated variables are compiled on every invocation, the compiled versions are not saved! However if you have caching enabled, the output will be cached with the rest of the template.

Example 8.3. {eval}

The contents of the config file, `setup.conf`.

```
emphstart = <strong>
emphend = </strong>
title = Welcome to {$company}'s home page!
ErrorCity = You must supply a {#emphstart#}city{#emphend#}.
ErrorState = You must supply a {#emphstart#}state{#emphend#}.
```

Where the template is:

```
{config_load file='setup.conf'}

{eval var=$foo}
{eval var=#title#}
{eval var=#ErrorCity#}
{eval var=#ErrorState# assign='state_error'}
{$state_error}
```

The above template will output:

```
This is the contents of foo.
Welcome to Foobar Pub & Grill's home page!
You must supply a <strong>city</strong>.
You must supply a <strong>state</strong>.
```

Example 8.4. Another {eval} example

This outputs the server name (in uppercase) and IP. The assigned variable *`$str`* could be from a database query.

```
<?php
$str = 'The server name is {$smarty.server.SERVER_NAME|upper} '
       .'at {$smarty.server.SERVER_ADDR}';
$smarty->assign('foo',$str);
?>
```

Where the template is:

```
    {eval var=$foo}
```

{fetch}

`{fetch}` is used to retrieve files from the local file system, http, or ftp and display the contents.

- If the file name begins with *`http://`*, the web site page will be fetched and displayed.

  ### Note

  This will not support http redirects, be sure to include a trailing slash on your web page fetches where necessary.

- If the file name begins with *`ftp://`*, the file will be downloaded from the ftp server and displayed.

- For local files, either a full system file path must be given, or a path relative to the executed php script.

  ### Note

  If security is enabled and you are fetching a file from the local file system, `{fetch}` will only allow files from within the *`$secure_dir`* path of the securty policy. See the Security section for details.

- If the *`assign`* attribute is set, the output of the `{fetch}` function will be assigned to this template variable instead of being output to the template.

Attribute Name	Type	Required	Default	Description
file	string	Yes	*n/a*	The file, http or ftp site to fetch
assign	string	No	*n/a*	The template variable the output will be assigned to

Example 8.5. {fetch} examples

```
{* include some javascript in your template *}
{fetch file='/export/httpd/www.example.com/docs/navbar.js'}

{* embed some weather text in your template from another web site *}
{fetch file='http://www.myweather.com/68502/'}

{* fetch a news headline file via ftp *}
{fetch file='ftp://user:password@ftp.example.com/path/to/currentheadlines.txt'}
{* as above but with variables *}
{fetch file="ftp://`$user`:`$password`@`$server`/`$path`"}

{* assign the fetched contents to a template variable *}
{fetch file='http://www.myweather.com/68502/' assign='weather'}
{if $weather ne ''}
  <div id="weather">{$weather}</div>
{/if}
```

See also `{capture}`, `{eval}`, `{assign}` and `fetch()`.

{html_checkboxes}

`{html_checkboxes}` is a custom function that creates an html checkbox group with provided data. It takes care of which item(s) are selected by default as well.

Attribute Name	Type	Required	Default	Description
name	string	No	*checkbox*	Name of checkbox list
values	array	Yes, unless using options attribute	*n/a*	An array of values for checkbox buttons
output	array	Yes, unless using options attribute	*n/a*	An array of output for checkbox buttons
selected	string/array	No	*empty*	The selected checkbox element(s)
options	associative array	Yes, unless using values and output	*n/a*	An associative array of values and output
separator	string	No	*empty*	String of text to separate each checkbox item
assign	string	No	*empty*	Assign checkbox tags to an array instead of output

Attribute Name	Type	Required	Default	Description
labels	boolean	No	*TRUE*	Add <label>-tags to the output

- Required attributes are *values* and *output*, unless you use *options* instead.
- All output is XHTML compliant.
- All parameters that are not in the list above are printed as name/value-pairs inside each of the created <input>-tags.

Example 8.6. {html_checkboxes}

```
<?php

$smarty->assign('cust_ids', array(1000,1001,1002,1003));
$smarty->assign('cust_names', array(
                                'Joe Schmoe',
                                'Jack Smith',
                                'Jane Johnson',
                                'Charlie Brown')
                              );
$smarty->assign('customer_id', 1001);

?>
```

where template is

```
{html_checkboxes name='id' values=$cust_ids output=$cust_names
   selected=$customer_id  separator='<br />'}
```

or where PHP code is:

```
<?php

$smarty->assign('cust_checkboxes', array(
                                     1000 => 'Joe Schmoe',
                                     1001 => 'Jack Smith',
                                     1002 => 'Jane Johnson',
                                     1003 => 'Charlie Brown')
                                   );
$smarty->assign('customer_id', 1001);

?>
```

and the template is

```
{html_checkboxes name='id' options=$cust_checkboxes
   selected=$customer_id separator='<br />'}
```

both examples will output:

```
<label><input type="checkbox" name="id[]" value="1000" />Joe Schmoe</label><br />
<label><input type="checkbox" name="id[]" value="1001" checked="checked" />Jack Sm
<br />
<label><input type="checkbox" name="id[]" value="1002" />Jane Johnson</label><br /
<label><input type="checkbox" name="id[]" value="1003" />Charlie Brown</label><br
```

Example 8.7. Database example (eg PEAR or ADODB):

```
<?php

$sql = 'select type_id, types from contact_types order by type';
$smarty->assign('contact_types',$db->getAssoc($sql));

$sql = 'select contact_id, contact_type_id, contact '
       .'from contacts where contact_id=12';
$smarty->assign('contact',$db->getRow($sql));

?>
```

The results of the database queries above would be output with.

```
{html_checkboxes name='contact_type_id' options=$contact_types
        selected=$contact.contact_type_id separator='<br />'}
```

See also `{html_radios}` and `{html_options}`

{html_image}

`{html_image}` is a custom function that generates an HTML `<img>` tag. The *`height`* and *`width`* are automatically calculated from the image file if they are not supplied.

Attribute Name	Type	Required	Default	Description
file	string	Yes	*n/a*	name/path to image
height	string	No	*actual image height*	Height to display image
width	string	No	*actual image width*	Width to display image
basedir	string	no	*web server doc root*	Directory to base relative paths from
alt	string	no	*“”*	Alternative description of the image
href	string	no	*n/a*	href value to link the image to
path_prefix	string	no	*n/a*	Prefix for output path

- *`basedir`* is the base directory that relative image paths are based from. If not given, the web server's document root `$_ENV['DOCUMENT_ROOT']` is used as the base. If security is enabled, then the image must be located in the *`$secure_dir`* path of the securty policy. See the Security section for details.

- *`href`* is the href value to link the image to. If link is supplied, an `<a href="LINKVALUE"><a>` tag is placed around the image tag.
- *`path_prefix`* is an optional prefix string you can give the output path. This is useful if you want to supply a different server name for the image.
- All parameters that are not in the list above are printed as name/value-pairs inside the created `<img>` tag.

Technical Note

`{html_image}` requires a hit to the disk to read the image and calculate the height and width. If you don't use template caching, it is generally better to avoid `{html_image}` and leave image tags static for optimal performance.

Example 8.8. {html_image} example

```
{html_image file='pumpkin.jpg'}
{html_image file='/path/from/docroot/pumpkin.jpg'}
{html_image file='../path/relative/to/currdir/pumpkin.jpg'}
```

Example output of the above template would be:

```
<img src="pumpkin.jpg" alt="" width="44" height="68" />
<img src="/path/from/docroot/pumpkin.jpg" alt="" width="44" height="68" />
<img src="../path/relative/to/currdir/pumpkin.jpg" alt="" width="44" height="68" /
```

{html_options}

`{html_options}` is a custom function that creates the html `<select><option>` group with the assigned data. It takes care of which item(s) are selected by default as well.

Attribute Name	Type	Required	Default	Description
values	array	Yes, unless using options attribute	*n/a*	An array of values for dropdown
output	array	Yes, unless using options attribute	*n/a*	An array of output for dropdown
selected	string/array	No	*empty*	The selected option element(s)
options	associative array	Yes, unless using values and output	*n/a*	An associative array of values and output
name	string	No	*empty*	Name of select group

- Required attributes are *`values`* and *`output`*, unless you use the combined *`options`* instead.
- If the optional *`name`* attribute is given, the `<select></select>` tags are created, otherwise ONLY the `<option>` list is generated.
- If a given value is an array, it will treat it as an html `<optgroup>`, and display the groups. Recursion is supported with `<optgroup>`.
- All parameters that are not in the list above are printed as name/value-pairs inside the `<select>` tag. They are ignored if the optional *`name`* is not given.
- All output is XHTML compliant.

Example 8.9. Associative array with the `options` attribute

```
<?php
$smarty->assign('myOptions', array(
                                1800 => 'Joe Schmoe',
                                9904 => 'Jack Smith',
                                2003 => 'Charlie Brown')
                                );
$smarty->assign('mySelect', 9904);
?>
```

The following template will generate a drop-down list. Note the presence of the *`name`* attribute which creates the `<select>` tags.

```
{html_options name=foo options=$myOptions selected=$mySelect}
```

Output of the above example would be:

```
<select name="foo">
<option label="Joe Schmoe" value="1800">Joe Schmoe</option>
<option label="Jack Smith" value="9904" selected="selected">Jack Smith</option>
<option label="Charlie Brown" value="2003">Charlie Brown</option>
</select>
```

Example 8.10. Dropdown with seperate arrays for `values` and `ouptut`

```
<?php
$smarty->assign('cust_ids', array(56,92,13));
$smarty->assign('cust_names', array(
                              'Joe Schmoe',
                              'Jane Johnson',
                              'Charlie Brown'));
$smarty->assign('customer_id', 92);
?>
```

The above arrays would be output with the following template (note the use of the php `count()` [http://php.net/function.count] function as a modifier to set the select size).

```
<select name="customer_id" size="{$cust_names|@count}">
   {html_options values=$cust_ids output=$cust_names selected=$customer_id}
</select>
```

The above example would output:

```
<select name="customer_id" size="3">
    <option label="Joe Schmoe" value="56">Joe Schmoe</option>
    <option label="Jack Smith" value="92" selected="selected">Jane Johnson</option
    <option label="Charlie Brown" value="13">Charlie Brown</option>
</select>
```

Example 8.11. Database example (eg ADODB or PEAR)

```
<?php

$sql = 'select type_id, types from contact_types order by type';
$smarty->assign('contact_types',$db->getAssoc($sql));

$sql = 'select contact_id, name, email, contact_type_id
        from contacts where contact_id='.$contact_id;
$smarty->assign('contact',$db->getRow($sql));

?>
```

Where a template could be as follows. Note the use of the `truncate` modifier.

```
<select name="type_id">
    <option value='null'>-- none --</option>
    {html_options options=$contact_types|truncate:20 selected=$contact.type_id}
</select>
```

Example 8.12. Dropdown's with <optgroup>

```
<?php
$arr['Sport'] = array(6 => 'Golf', 9 => 'Cricket',7 => 'Swim');
$arr['Rest']  = array(3 => 'Sauna',1 => 'Massage');
$smarty->assign('lookups', $arr);
$smarty->assign('fav', 7);
?>
```

The script above and the following template

```
{html_options name=foo options=$lookups selected=$fav}
```

would output:

```
<select name="foo">
<optgroup label="Sport">
<option label="Golf" value="6">Golf</option>
<option label="Cricket" value="9">Cricket</option>
<option label="Swim" value="7" selected="selected">Swim</option>
</optgroup>
<optgroup label="Rest">
<option label="Sauna" value="3">Sauna</option>
<option label="Massage" value="1">Massage</option>
</optgroup>
</select>
```

See also `{html_checkboxes}` and `{html_radios}`

{html_radios}

`{html_radios}` is a custom function that creates a HTML radio button group. It also takes care of which item is selected by default as well.

Attribute Name	Type	Required	Default	Description
name	string	No	*radio*	Name of radio list
values	array	Yes, unless using options attribute	*n/a*	An array of values for radio buttons
output	array	Yes, unless using options attribute	*n/a*	An array of output for radio buttons

Attribute Name	Type	Required	Default	Description
selected	string	No	*empty*	The selected radio element
options	associative array	Yes, unless using values and output	*n/a*	An associative array of values and output
separator	string	No	*empty*	String of text to separate each radio item
assign	string	No	*empty*	Assign radio tags to an array instead of output

- Required attributes are `values` and `output`, unless you use `options` instead.
- All output is XHTML compliant.
- All parameters that are not in the list above are output as name/value-pairs inside each of the created `<input>`-tags.

Example 8.13. {html_radios} first example

```
<?php

$smarty->assign('cust_ids', array(1000,1001,1002,1003));
$smarty->assign('cust_names', array(
                              'Joe Schmoe',
                              'Jack Smith',
                              'Jane Johnson',
                              'Charlie Brown')
                              );
$smarty->assign('customer_id', 1001);

?>
```

Where template is:

```
{html_radios name='id' values=$cust_ids output=$cust_names
       selected=$customer_id separator='<br />'}
```

Example 8.14. {html_radios} second example

```
<?php

$smarty->assign('cust_radios', array(
                               1000 => 'Joe Schmoe',
                               1001 => 'Jack Smith',
                               1002 => 'Jane Johnson',
                               1003 => 'Charlie Brown'));
$smarty->assign('customer_id', 1001);

?>
```

Where template is:

```
{html_radios name='id' options=$cust_radios
     selected=$customer_id separator='<br />'}
```

Both examples will output:

```
<label for="id_1000">
<input type="radio" name="id" value="1000" id="id_1000" />Joe Schmoe</label><br />
<label for="id_1001"><input type="radio" name="id" value="1001" id="id_1001" check
<label for="id_1002"><input type="radio" name="id" value="1002" id="id_1002" />Jan
<label for="id_1003"><input type="radio" name="id" value="1003" id="id_1003" />Cha
```

Example 8.15. {html_radios} - Database example (eg PEAR or ADODB):

```
<?php

$sql = 'select type_id, types from contact_types order by type';
$smarty->assign('contact_types',$db->getAssoc($sql));

$sql = 'select contact_id, name, email, contact_type_id '
        .'from contacts where contact_id='.$contact_id;
$smarty->assign('contact',$db->getRow($sql));

?>
```

The variable assigned from the database above would be output with the template:

```
{html_radios name='contact_type_id' options=$contact_types
     selected=$contact.contact_type_id separator='<br />'}
```

See also `{html_checkboxes}` and `{html_options}`

{html_select_date}

`{html_select_date}` is a custom function that creates date dropdowns. It can display any or all of year, month, and day. All parameters that are not in the list below are printed as name/value-pairs inside the `<select>` tags of day, month and year.

Attribute Name	Type	Required	Default	Description
prefix	string	No	Date_	What to prefix the var name with
time	timestamp/ YYYY-MM-DD	No	current time in unix timestamp or YYYY-MM-DD format	What date/time to use
start_year	string	No	current year	The first year in the dropdown, either year number, or relative to current year (+/- N)
end_year	string	No	same as start_year	The last year in the dropdown, either year number, or relative to current year (+/- N)
display_days	boolean	No	`TRUE`	Whether to display days or not

Attribute Name	Type	Required	Default	Description
display_months	boolean	No	TRUE	Whether to display months or not
display_years	boolean	No	TRUE	Whether to display years or not
month_format	string	No	%B	What format the month should be in (strftime)
day_format	string	No	%02d	What format the day output should be in (sprintf)
day_value_format	string	No	%d	What format the day value should be in (sprintf)
year_as_text	boolean	No	FALSE	Whether or not to display the year as text
reverse_years	boolean	No	FALSE	Display years in reverse order
field_array	string	No	null	If a name is given, the select boxes will be drawn such that the results will be returned to PHP in the form of name[Day], name[Year], name[Month].
day_size	string	No	null	Adds size attribute to select tag if given
month_size	string	No	null	Adds size attribute to select tag if given
year_size	string	No	null	Adds size attribute to select tag if given
all_extra	string	No	null	Adds extra attributes to all select/input tags if given
day_extra	string	No	null	Adds extra attributes to select/input tags if given
month_extra	string	No	null	Adds extra attributes to select/input tags if given
year_extra	string	No	null	Adds extra attributes to select/input tags if given

Attribute Name	Type	Required	Default	Description
field_order	string	No	MDY	The order in which to display the fields
field_separator	string	No	\n	String printed between different fields
month_value_format	string	No	%m	strftime() format of the month values, default is %m for month numbers.
year_empty	string	No	null	If supplied then the first element of the year's select-box has this value as it's label and “” as it's value. This is useful to make the select-box read “Please select a year” for example. Note that you can use values like “-MM-DD” as time-attribute to indicate an unselected year.
month_empty	string	No	null	If supplied then the first element of the month's select-box has this value as it's label and “” as it's value. . Note that you can use values like “YYYY--DD” as time-attribute to indicate an unselected month.
day_empty	string	No	null	If supplied then the first element of the day's select-box has this value as it's label and “” as it's value. Note that you can use values like “YYYY-MM-” as time-attribute to indicate an unselected day.

Note

There is an useful php function on the date tips page for converting `{html_select_date}` form values to a timestamp.

Example 8.16. {html_select_date}

Template code

```
{html_select_date}
```

This will output:

```
<select name="Date_Month">
<option value="1">January</option>
<option value="2">February</option>
<option value="3">March</option>
  ..... snipped .....
<option value="10">October</option>
<option value="11">November</option>
<option value="12" selected="selected">December</option>
</select>
<select name="Date_Day">
<option value="1">01</option>
<option value="2">02</option>
<option value="3">03</option>
  ..... snipped .....
<option value="11">11</option>
<option value="12">12</option>
<option value="13" selected="selected">13</option>
<option value="14">14</option>
<option value="15">15</option>
  ..... snipped .....
<option value="29">29</option>
<option value="30">30</option>
<option value="31">31</option>
</select>
<select name="Date_Year">
<option value="2006" selected="selected">2006</option>
</select>
```

Example 8.17. {html_select_date} second example

```
{* start and end year can be relative to current year *}
{html_select_date prefix='StartDate' time=$time start_year='-5'
   end_year='+1' display_days=false}
```

With 2000 as the current year the output:

```
<select name="StartDateMonth">
<option value="1">January</option>
<option value="2">February</option>
.... snipped ....
<option value="11">November</option>
<option value="12" selected="selected">December</option>
</select>
<select name="StartDateYear">
<option value="1995">1995</option>
.... snipped ....
<option value="1999">1999</option>
<option value="2000" selected="selected">2000</option>
<option value="2001">2001</option>
</select>
```

See also `{html_select_time}`, `date_format`, *`$smarty.now`* and the date tips page.

{html_select_time}

`{html_select_time}` is a custom function that creates time dropdowns for you. It can display any or all of hour, minute, second and meridian.

The *`time`* attribute can have different formats. It can be a unique timestamp, a string of the format `YYYYMMDDHHMMSS` or a string that is parseable by PHP's `strtotime()` [http://php.net/strtotime].

Attribute Name	Type	Required	Default	Description
prefix	string	No	Time_	What to prefix the var name with
time	timestamp	No	current time	What date/time to use
display_hours	boolean	No	`TRUE`	Whether or not to display hours
display_minutes	boolean	No	`TRUE`	Whether or not to display minutes
display_seconds	boolean	No	`TRUE`	Whether or not to display seconds

Attribute Name	Type	Required	Default	Description
display_meridian	boolean	No	TRUE	Whether or not to display meridian (am/pm)
use_24_hours	boolean	No	TRUE	Whether or not to use 24 hour clock
minute_interval	integer	No	1	Number interval in minute dropdown
second_interval	integer	No	1	Number interval in second dropdown
field_array	string	No	n/a	Outputs values to array of this name
all_extra	string	No	null	Adds extra attributes to select/ input tags if given
hour_extra	string	No	null	Adds extra attributes to select/ input tags if given
minute_extra	string	No	null	Adds extra attributes to select/ input tags if given
second_extra	string	No	null	Adds extra attributes to select/ input tags if given
meridian_extra	string	No	null	Adds extra attributes to select/ input tags if given

Example 8.18. {html_select_time}

```
{html_select_time use_24_hours=true}
```

At 9:20 and 23 seconds in the morning the template above would output:

```
<select name="Time_Hour">
<option value="00">00</option>
<option value="01">01</option>
... snipped ....
<option value="08">08</option>
<option value="09" selected>09</option>
<option value="10">10</option>
... snipped ....
<option value="22">22</option>
<option value="23">23</option>
</select>
<select name="Time_Minute">
<option value="00">00</option>
<option value="01">01</option>
... snipped ....
<option value="19">19</option>
<option value="20" selected>20</option>
<option value="21">21</option>
... snipped ....
<option value="58">58</option>
<option value="59">59</option>
</select>
<select name="Time_Second">
<option value="00">00</option>
<option value="01">01</option>
... snipped ....
<option value="22">22</option>
<option value="23" selected>23</option>
<option value="24">24</option>
... snipped ....
<option value="58">58</option>
<option value="59">59</option>
</select>
<select name="Time_Meridian">
<option value="am" selected>AM</option>
<option value="pm">PM</option>
</select>
```

See also *`$smarty.now`*, `{html_select_date}` and the date tips page.

{html_table}

`{html_table}` is a custom function that dumps an array of data into an HTML `<table>`.

Attribute Name	Type	Required	Default	Description
loop	array	Yes	*n/a*	Array of data to loop through
cols	mixed	No	*3*	Number of columns in the table or a comma-separated list of column heading names or an array of column heading names.if the cols-attribute is empty, but rows are given, then the number of cols is computed by the number of rows and the number of elements to display to be just enough cols to display all elements. If both, rows and cols, are omitted cols defaults to 3. if given as a list or array, the number of columns is computed from the number of elements in the list or array.
rows	integer	No	*empty*	Number of rows in the table. if the rows-attribute is empty, but cols are given, then the number of rows is computed by the number of cols and the number of elements to display to be just enough rows to display all elements.
inner	string	No	*cols*	Direction of consecutive elements in the loop-array to be

Attribute Name	Type	Required	Default	Description
				rendered. *cols* means elements are displayed col-by-col. *rows* means elements are displayed row-by-row.
caption	string	No	*empty*	Text to be used for the `<caption>` element of the table
table_attr	string	No	*border="1"*	Attributes for `<table>` tag
th_attr	string	No	*empty*	Attributes for `<th>` tag (arrays are cycled)
tr_attr	string	No	*empty*	attributes for `<tr>` tag (arrays are cycled)
td_attr	string	No	*empty*	Attributes for `<td>` tag (arrays are cycled)
trailpad	string	No	*&nbsp;*	Value to pad the trailing cells on last row with (if any)
hdir	string	No	*right*	Direction of each row to be rendered. possible values: *right* (left-to-right), and *left* (right-to-left)
vdir	string	No	*down*	Direction of each column to be rendered. possible values: *down* (top-to-bottom), *up* (bottom-to-top)

- The *`cols`* attribute determines how many columns will be in the table.
- The *`table_attr`*, *`tr_attr`* and *`td_attr`* values determine the attributes given to the `<table>`, `<tr>` and `<td>` tags.
- If *`tr_attr`* or *`td_attr`* are arrays, they will be cycled through.
- *`trailpad`* is the value put into the trailing cells on the last table row if there are any present.

Example 8.19. {html_table}

```
<?php
$smarty->assign( 'data', array(1,2,3,4,5,6,7,8,9) );
$smarty->assign( 'tr', array('bgcolor="#eeeeee"','bgcolor="#dddddd"') );
$smarty->display('index.tpl');
?>
```

The variables assigned from php could be displayed as these three examples demonstrate. Each example shows the template followed by output.

```
{**** Example One ****}
{html_table loop=$data}

<table border="1">
<tbody>
<tr><td>1</td><td>2</td><td>3</td></tr>
<tr><td>4</td><td>5</td><td>6</td></tr>
<tr><td>7</td><td>8</td><td>9</td></tr>
</tbody>
</table>


{**** Example Two ****}
{html_table loop=$data cols=4 table_attr='border="0"'}

<table border="0">
<tbody>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>9</td><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;</td></tr>
</tbody>
</table>


{**** Example Three ****}
{html_table loop=$data cols="first,second,third,fourth" tr_attr=$tr}

<table border="1">
<thead>
<tr>
<th>first</th><th>second</th><th>third</th><th>fourth</th>
</tr>
</thead>
<tbody>
<tr bgcolor="#eeeeee"><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr bgcolor="#dddddd"><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr bgcolor="#eeeeee"><td>9</td><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;</td></tr>
</tbody>
</table>
```

{mailto}

`{mailto}` automates the creation of a `mailto:` anchor links and optionally encodes them. Encoding emails makes it more difficult for web spiders to lift email addresses off of a site.

Technical Note

Javascript is probably the most thorough form of encoding, although you can use hex encoding too.

Attribute Name	**Type**	**Required**	**Default**	**Description**
address	string	Yes	*n/a*	The e-mail address
text	string	No	*n/a*	The text to display, default is the e-mail address
encode	string	No	*none*	How to encode the e-mail. Can be one of `none`, `hex`, `javascript` or `javascript_charcode`.
cc	string	No	*n/a*	Email addresses to carbon copy, separate entries by a comma.
bcc	string	No	*n/a*	Email addresses to blind carbon copy, separate entries by a comma
subject	string	No	*n/a*	Email subject
newsgroups	string	No	*n/a*	Newsgroups to post to, separate entries by a comma.
followupto	string	No	*n/a*	Addresses to follow up to, separate entries by a comma.
extra	string	No	*n/a*	Any extra information you want passed to the link, such as style sheet classes

Example 8.20. {mailto} example lines followed by the result

```
{mailto address="me@example.com"}
<a href="mailto:me@example.com" >me@example.com</a>

{mailto address="me@example.com" text="send me some mail"}
<a href="mailto:me@example.com" >send me some mail</a>

{mailto address="me@example.com" encode="javascript"}
<script type="text/javascript" language="javascript">
   eval(unescape('%64%6f% ... snipped ...%61%3e%27%29%3b'))
</script>

{mailto address="me@example.com" encode="hex"}
<a href="mailto:%6d%65.. snipped..3%6f%6d">&#x6d;&..snipped...#x6f;&#x6d;</a>

{mailto address="me@example.com" subject="Hello to you!"}
<a href="mailto:me@example.com?subject=Hello%20to%20you%21" >me@example.com</a>

{mailto address="me@example.com" cc="you@example.com,they@example.com"}
<a href="mailto:me@example.com?cc=you@example.com%2Cthey@example.com" >me@example.

{mailto address="me@example.com" extra='class="email"'}
<a href="mailto:me@example.com" class="email">me@example.com</a>

{mailto address="me@example.com" encode="javascript_charcode"}
<script type="text/javascript" language="javascript">
    <!--
    {document.write(String.fromCharCode(60,97, ... snipped ....60,47,97,62))}
    //-->
</script>
```

See also `escape`, `{textformat}` and obfuscating email addresses.

{math}

`{math}` allows the template designer to do math equations in the template.

- Any numeric template variables may be used in the equations, and the result is printed in place of the tag.

- The variables used in the equation are passed as parameters, which can be template variables or static values.

- +, -, /, *, abs, ceil, cos, exp, floor, log, log10, max, min, pi, pow, rand, round, sin, sqrt, srans and tan are all valid operators. Check the PHP documentation for further information on these math [http://php.net/eval] functions.

- If you supply the *`assign`* attribute, the output of the `{math}` function will be assigned to this template variable instead of being output to the template.

Technical Note

`{math}` is an expensive function in performance due to its use of the php `eval()` [http://php.net/eval] function. Doing the math in PHP is much more efficient, so whenever possible do the math calculations in the script and `assign()` the results to the template. Definitely avoid repetitive `{math}` function calls, eg within `{section}` loops.

Attribute Name	Type	Required	Default	Description
equation	string	Yes	*n/a*	The equation to execute
format	string	No	*n/a*	The format of the result (sprintf)
var	numeric	Yes	*n/a*	Equation variable value
assign	string	No	*n/a*	Template variable the output will be assigned to
[var ...]	numeric	Yes	*n/a*	Equation variable value

Example 8.21. {math}

Example a:

```
{* $height=4, $width=5 *}

{math equation="x + y" x=$height y=$width}
```

The above example will output:

```
9
```

Example b:

```
{* $row_height = 10, $row_width = 20, #col_div# = 2, assigned in template *}

{math equation="height * width / division"
height=$row_height
width=$row_width
division=#col_div#}
```

The above example will output:

```
100
```

Example c:

```
{* you can use parenthesis *}

{math equation="(( x + y ) / z )" x=2 y=10 z=2}
```

The above example will output:

```
6
```

Example d:

```
{* you can supply a format parameter in sprintf format *}

{math equation="x + y" x=4.4444 y=5.0000 format="%.2f"}
```

The above example will output:

```
9.44
```

{textformat}

`{textformat}` is a block function used to format text. It basically cleans up spaces and special characters, and formats paragraphs by wrapping at a boundary and indenting lines.

You can set the parameters explicitly, or use a preset style. Currently “email” is the only available style.

Attribute Name	Type	Required	Default	Description
style	string	No	*n/a*	Preset style
indent	number	No	*0*	The number of chars to indent every line
indent_first	number	No	*0*	The number of chars to indent the first line
indent_char	string	No	*(single space)*	The character (or string of chars) to indent with
wrap	number	No	*80*	How many characters to wrap each line to
wrap_char	string	No	*\n*	The character (or string of chars) to break each line with
wrap_cut	boolean	No	*`FALSE`*	If `TRUE`, wrap will break the line at the exact character instead of at a word boundary
assign	string	No	*n/a*	The template variable the output will be assigned to

Example 8.22. {textformat}

```
{textformat wrap=40}

This is foo.
This is foo.
This is foo.
This is foo.
This is foo.
This is foo.

This is bar.

bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.

{/textformat}
```

The above example will output:

```
This is foo. This is foo. This is foo.
This is foo. This is foo. This is foo.

This is bar.

bar foo bar foo foo. bar foo bar foo
foo. bar foo bar foo foo. bar foo bar
foo foo. bar foo bar foo foo. bar foo
bar foo foo. bar foo bar foo foo.
```

```
{textformat wrap=40 indent=4}

This is foo.
This is foo.
This is foo.
This is foo.
This is foo.
This is foo.

This is bar.

bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.

{/textformat}
```

The above example will output:

```
This is foo. This is foo. This is
foo. This is foo. This is foo. This
is foo.

This is bar.

bar foo bar foo foo. bar foo bar foo
foo. bar foo bar foo foo. bar foo
bar foo foo. bar foo bar foo foo.
bar foo bar foo foo. bar foo bar
foo foo.
```

```
{textformat wrap=40 indent=4 indent_first=4}

This is foo.
This is foo.
This is foo.
This is foo.
This is foo.
This is foo.

This is bar.

bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.

{/textformat}
```

The above example will output:

```
This is foo. This is foo. This
is foo. This is foo. This is foo.
This is foo.

This is bar.

bar foo bar foo foo. bar foo bar
foo foo. bar foo bar foo foo. bar
foo bar foo foo. bar foo bar foo
foo. bar foo bar foo foo. bar foo
bar foo foo.
```

```
{textformat style="email"}

This is foo.
This is foo.
This is foo.
This is foo.
This is foo.
This is foo.

This is bar.

bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.
bar foo bar foo     foo.

{/textformat}
```

The above example will output:

```
This is foo. This is foo. This is foo. This is foo. This is foo. This is
foo.

This is bar.

bar foo bar foo foo. bar foo bar foo foo. bar foo bar foo foo. bar foo
bar foo foo. bar foo bar foo foo. bar foo bar foo foo. bar foo bar foo
foo.
```

See also `{strip}` and `wordwrap`.

Chapter 9. Config Files

Config files are handy for designers to manage global template variables from one file. One example is template colors. Normally if you wanted to change the color scheme of an application, you would have to go through each and every template file and change the colors. With a config file, the colors can be kept in one place, and only one file needs to be updated.

Example 9.1. Example of config file syntax

```
# global variables
pageTitle = "Main Menu"
bodyBgColor = #000000
tableBgColor = #000000
rowBgColor = #00ff00

[Customer]
pageTitle = "Customer Info"

[Login]
pageTitle = "Login"
focus = "username"
Intro = """This is a value that spans more
           than one line. you must enclose
           it in triple quotes."""

# hidden section
[.Database]
host=my.example.com
db=ADDRESSBOOK
user=php-user
pass=foobar
```

Values of config file variables can be in quotes, but not necessary. You can use either single or double quotes. If you have a value that spans more than one line, enclose the entire value with triple quotes ("""). You can put comments into config files by any syntax that is not a valid config file syntax. We recommend using a `#` (hash) at the beginning of the line.

The example config file above has two sections. Section names are enclosed in [brackets]. Section names can be arbitrary strings not containing `[` or `]` symbols. The four variables at the top are global variables, or variables not within a section. These variables are always loaded from the config file. If a particular section is loaded, then the global variables and the variables from that section are also loaded. If a variable exists both as a global and in a section, the section variable is used. If you name two variables the same within a section, the last one will be used unless *`$config_overwrite`* is disabled.

Config files are loaded into templates with the built-in template function `{config_load}` or the API `config_load()` function.

You can hide variables or entire sections by prepending the variable name or section name with a period eg `[.hidden]`. This is useful if your application reads the config files and gets sensitive data from them that the template engine does not need. If you have third parties doing template editing, you can be certain that they cannot read sensitive data from the config file by loading it into the template.

See also `{config_load}`, *`$config_overwrite`*, `get_config_vars()`, `clear_config()` and `config_load()`

Chapter 10. Debugging Console

There is a debugging console included with Smarty. The console informs you of all the included templates, assigned variables and config file variables for the current invocation of the template. A template file named `debug.tpl` is included with the distribution of Smarty which controls the formatting of the console.

Set *`$debugging`* to `TRUE` in Smarty, and if needed set *`$debug_tpl`* to the template resource path to `debug.tpl` (this is in `SMARTY_DIR` by default). When you load the page, a Javascript console window will pop up and give you the names of all the included templates and assigned variables for the current page.

To see the available variables for a particular template, see the `{debug}` template function. To disable the debugging console, set *`$debugging`* to `FALSE`. You can also temporarily turn on the debugging console by putting `SMARTY_DEBUG` in the URL if you enable this option with *`$debugging_ctrl`* .

Technical Note

The debugging console does not work when you use the `fetch()` API, only when using `display()`. It is a set of javascript statements added to the very bottom of the generated template. If you do not like javascript, you can edit the `debug.tpl` template to format the output however you like. Debug data is not cached and `debug.tpl` info is not included in the output of the debug console.

Note

The load times of each template and config file are in seconds, or fractions thereof.

See also troubleshooting.

Part III. Smarty For Programmers

Table of Contents

Chapter 11. Constants

SMARTY_DIR

This is the **full system path** to the location of the Smarty class files. If this is not defined in your script, then Smarty will attempt to determine the appropriate value automatically. If defined, the path **must end with a trailing slash/**.

Example 11.1. SMARTY_DIR

```
<?php
// set path to Smarty directory *nix style
define('SMARTY_DIR', '/usr/local/lib/php/Smarty-v.e.r/libs/');

// path to Smarty windows style
define('SMARTY_DIR', 'c:/webroot/libs/Smarty-v.e.r/libs/');

// include the smarty class, note 'S' is upper case
require_once(SMARTY_DIR . 'Smarty.class.php');
?>
```

See also *`$smarty.const`* and *`$php_handling constants`*

Chapter 12. Smarty Class Variables

These are all of the available Smarty class variables. You can access them directly, or use the corresponding setter/getter methods.

Note

All class variables have magic setter/getter methods available. setter/getter methods are camelCaseFormat, unlike the variable itself. So for example, you can set and get the $smarty->template_dir variable with $smarty->setTemplateDir($dir) and $dir = $smarty->getTemplateDir() respectively.

Note

See `Changing settings by template` section for how to change Smarty class variables for individual templates.

$template_dir

This is the name of the default template directory. If you do not supply a resource type when including files, they will be found here. By default this is `./templates`, meaning that Smarty will look for the `templates/` directory in the same directory as the executing php script.

Technical Note

It is not recommended to put this directory under the web server document root.

$compile_dir

This is the name of the directory where compiled templates are located. By default this is `./templates_c`, meaning that Smarty will look for the `templates_c/` directory in the same directory as the executing php script. **This directory must be writeable by the web server**, see install for more info.

Technical Note

This setting must be either a relative or absolute path. include_path is not used for writing files.

Technical Note

It is not recommended to put this directory under the web server document root.

See also *`$compile_id`* and *`$use_sub_dirs`*.

$config_dir

This is the directory used to store config files used in the templates. Default is `./configs`, meaning that Smarty will look for the `configs/` directory in the same directory as the executing php script.

Technical Note

It is not recommended to put this directory under the web server document root.

$plugins_dir

This is the directory or directories where Smarty will look for the plugins that it needs. Default is `plugins/` under the `SMARTY_DIR`. If you supply a relative path, Smarty will first look under the `SMARTY_DIR`, then relative to the current working directory, then relative to the PHP include_path. If *`$plugins_dir`* is an array of directories, Smarty will search for your plugin in each plugin directory **in the order they are given**.

Technical Note

For best performance, do not setup your *`$plugins_dir`* to have to use the PHP include path. Use an absolute pathname, or a path relative to `SMARTY_DIR` or the current working directory.

Example 12.1. Appending a local plugin dir

```
<?php

$smarty->plugins_dir[] = 'includes/my_smarty_plugins';

?>
```

Example 12.2. Multiple $plugins_dir

```
<?php

$smarty->plugins_dir = array(
                       'plugins', // the default under SMARTY_DIR
                       '/path/to/shared/plugins',
                       '../../includes/my/plugins'
                       );

?>
```

$debugging

This enables the debugging console. The console is a javascript popup window that informs you of the included templates, variables assigned from php and config file variables for the current script. It does not show variables assigned within a template with the `{assign}` function.

The console can also be enabled from the url with *`$debugging_ctrl`*.

See also `{debug}`, *`$debug_tpl`*, and *`$debugging_ctrl`*.

$debug_tpl

This is the name of the template file used for the debugging console. By default, it is named `debug.tpl` and is located in the `SMARTY_DIR`.

See also *`$debugging`* and the debugging console section.

$debugging_ctrl

This allows alternate ways to enable debugging. `NONE` means no alternate methods are allowed. `URL` means when the keyword `SMARTY_DEBUG` is found in the `QUERY_STRING`, debugging is enabled for that invocation of the script. If *`$debugging`* is `TRUE`, this value is ignored.

Example 12.3. $debugging_ctrl on localhost

```
<?php
// shows debug console only on localhost ie
// http://localhost/script.php?foo=bar&SMARTY_DEBUG
$smarty->debugging = false; // the default
$smarty->debugging_ctrl = ($_SERVER['SERVER_NAME'] == 'localhost') ? 'URL' : 'NONE
?>
```

See also debugging console section and *`$debugging`*.

$auto_literal

The Smarty delimiter tags { and } will be ignored so long as they are surrounded by white space. This behavior can be disabled by setting auto_literal to false.

```
<?php
$smarty->auto_literal = false;
?>
```

See also Escaping Smarty Parsing,

$autoload_filters

If there are some filters that you wish to load on every template invocation, you can specify them using this variable and Smarty will automatically load them for you. The variable is an associative array where keys are filter types and values are arrays of the filter names. For example:

```
<?php
$smarty->autoload_filters = array('pre' => array('trim', 'stamp'),
                                        'output' => array('convert'));
?>
```

See also `registerFilter()` and `loadFilter()`

$compile_check

Upon each invocation of the PHP application, Smarty tests to see if the current template has changed (different time stamp) since the last time it was compiled. If it has changed, it recompiles that template. If the template has not been compiled, it will compile regardless of this setting. By default this variable is set to `TRUE`.

Once an application is put into production (ie the templates won't be changing), the compile check step is no longer needed. Be sure to set *`$compile_check`* to `FALSE` for maximal performance. Note that if you change this to `FALSE` and a template file is changed, you will *not* see the change since the template will not get recompiled. If *`$caching`* is enabled and *`$compile_check`* is enabled, then the cache files will get regenerated if an involved template file or config file was updated. See *`$force_compile`* and `clear_compiled_tpl()`.

$force_compile

This forces Smarty to (re)compile templates on every invocation. This setting overrides *`$compile_check`*. By default this is `FALSE`. This is handy for development and debugging. It should never be used in a production environment. If *`$caching`* is enabled, the cache file(s) will be regenerated every time.

$caching

This tells Smarty whether or not to cache the output of the templates to the *`$cache_dir`*. By default this is set to the constant Smarty::CACHING_OFF. If your templates consistently generate the same content, it is advisable to turn on *`$caching`*, as this may result in significant performance gains.

You can also have multiple caches for the same template.

- A constant value of Smarty::CACHING_LIFETIME_CURRENT or Smarty::CACHING_LIFETIME_SAVED enables caching.

- A value of Smarty::CACHING_LIFETIME_CURRENT tells Smarty to use the current *`$cache_lifetime`* variable to determine if the cache has expired.

- A value of Smarty::CACHING_LIFETIME_SAVED tells Smarty to use the *`$cache_lifetime`* value at the time the cache was generated. This way you can set the *`$cache_lifetime`* just before fetching the template to have granular control over when that particular cache expires. See also `isCached()`.

- If *`$compile_check`* is enabled, the cached content will be regenerated if any of the templates or config files that are part of this cache are changed.

- If *`$force_compile`* is enabled, the cached content will always be regenerated.

See also *`$cache_dir`*, *`$cache_lifetime`*, *`$cache_handler_func`*, *`$cache_modified_check`*, `is_cached()` and the caching section.

$cache_id

Persistent cache_id identifier. As an alternative to passing the same *`$cache_id`* to each and every function call, you can set this *`$cache_id`* and it will be used implicitly thereafter.

With a *`$cache_id`* you can have multiple cache files for a single call to `display()` or `fetch()` depending for example from different content of the same template. See the caching section for more information.

$cache_dir

This is the name of the directory where template caches are stored. By default this is `./cache`, meaning that Smarty will look for the `cache/` directory in the same directory as the executing php script. **This directory must be writeable by the web server**, see install for more info.

You can also use your own custom cache handler function to control cache files, which will ignore this setting. See also *`$use_sub_dirs`*.

Technical Note

This setting must be either a relative or absolute path. include_path is not used for writing files.

Technical Note

It is not recommended to put this directory under the web server document root.

See also *`$caching`*, *`$use_sub_dirs`*, *`$cache_lifetime`*, *`$cache_handler_func`*, *`$cache_modified_check`* and the caching section.

$cache_lifetime

This is the length of time in seconds that a template cache is valid. Once this time has expired, the cache will be regenerated.

- *`$caching`* must be turned on (either SMARTY_CACHING_LIFETIME_CURRENT or SMARTY_CACHING_LIFETIME_SAVED) for *`$cache_lifetime`* to have any purpose.
- A *`$cache_lifetime`* value of -1 will force the cache to never expire.
- A value of 0 will cause the cache to always regenerate (good for testing only, to disable caching a more efficient method is to set *`$caching`* = SMARTY_CACHING_OFF).
- If you want to give certain templates their own cache lifetime, you could do this by setting *`$caching`* = SMARTY_CACHING_LIFETIME_SAVED, then set *`$cache_lifetime`* to a unique value just before calling `display()` or `fetch()`.

If *`$force_compile`* is enabled, the cache files will be regenerated every time, effectively disabling caching. You can clear all the cache files with the `clear_all_cache()` function, or individual cache files (or groups) with the `clear_cache()` function.

$cache_handler_func

You can supply a custom function to handle cache files instead of using the built-in method using the *`$cache_dir`*. See the custom cache handler function section for more details.

$cache_modified_check

If set to TRUE, Smarty will respect the If-Modified-Since header sent from the client. If the cached file timestamp has not changed since the last visit, then a '304: Not Modified' header will be sent instead of the content. This works only on cached content without {insert} tags.

See also *$caching*, *$cache_lifetime*, *$cache_handler_func*, and the caching section.

$config_overwrite

If set to TRUE, the default then variables read in from config files will overwrite each other. Otherwise, the variables will be pushed onto an array. This is helpful if you want to store arrays of data in config files, just list each element multiple times.

Example 12.4. Array of config #variables#

This examples uses {cycle} to output a table with alternating red/green/blue row colors with *$config_overwrite* = FALSE.

The config file.

```
# row colors
rowColors = #FF0000
rowColors = #00FF00
rowColors = #0000FF
```

The template with a {section} loop.

```
<table>
  {section name=r loop=$rows}
  <tr bgcolor="{cycle values=#rowColors#}">
    <td> ....etc.... </td>
  </tr>
  {/section}
</table>
```

See also {config_load}, get_config_vars(), clear_config(), config_load() and the config files section.

$config_booleanize

If set to TRUE, config files values of on/true/yes and off/false/no get converted to boolean values automatically. This way you can use the values in the template like so: {if #foobar#}...{/if}. If foobar was on, true or yes, the {if} statement will execute. Defaults to TRUE.

$config_read_hidden

If set to `TRUE`, hidden sections ie section names beginning with a .period in config files can be read from templates. Typically you would leave this `FALSE`, that way you can store sensitive data in the config files such as database parameters and not worry about the template loading them. `FALSE` by default.

$config_fix_newlines

If set to `TRUE`, mac and dos newlines ie `'\r'` and `'\r\n'` in config files are converted to `'\n'` when they are parsed. Default is `TRUE`.

$default_template_handler_func

This function is called when a template cannot be obtained from its resource.

$php_handling

This tells Smarty how to handle PHP code embedded in the templates. There are four possible settings, the default being `Smarty::PHP_PASSTHRU`. Note that this does NOT affect php code within `{php}{/php}` tags in the template.

- `Smarty::PHP_PASSTHRU` - Smarty echos tags as-is.
- `Smarty::PHP_QUOTE` - Smarty quotes the tags as html entities.
- `Smarty::PHP_REMOVE` - Smarty removes the tags from the templates.
- `Smarty::PHP_ALLOW` - Smarty will execute the tags as PHP code.

Note

Embedding PHP code into templates is highly discouraged. Use custom functions or modifiers instead.

$trusted_dir

`$trusted_dir` is only for use when security is enabled. This is an array of all directories that are considered trusted. Trusted directories are where you keep php scripts that are executed directly from the templates with `{include_php}`.

$left_delimiter

This is the left delimiter used by the template language. Default is `{`.

See also *`$right_delimiter`* and escaping smarty parsing .

$right_delimiter

This is the right delimiter used by the template language. Default is `}`.

See also *$left_delimiter* and escaping smarty parsing.

$compiler_class

Specifies the name of the compiler class that Smarty will use to compile the templates. The default is 'Smarty_Compiler'. For advanced users only.

$request_vars_order

The order in which request variables are registered, similar to variables_order in php.ini

See also *$smarty.request* and *$request_use_auto_globals*.

$request_use_auto_globals

Specifies if Smarty should use PHP's `$HTTP_*_VARS[]` when `FALSE` or `$_*[]` when `TRUE` which is the default value. This affects templates that make use of `{$smarty.request.*}`, `{$smarty.get.*}` etc.

Caution

If you set `$request_use_auto_globals to true`, *$request_vars_order* has no effect but PHP's configuration value `gpc_order` is used.

$compile_id

Persistant compile identifier. As an alternative to passing the same *$compile_id* to each and every function call, you can set this *$compile_id* and it will be used implicitly thereafter.

With a *$compile_id* you can work around the limitation that you cannot use the same *$compile_dir* for different *$template_dirs*. If you set a distinct *$compile_id* for each *$template_dir* then Smarty can tell the compiled templates apart by their *$compile_id*.

If you have for example a prefilter that localizes your templates (that is: translates language dependend parts) at compile time, then you could use the current language as *$compile_id* and you will get a set of compiled templates for each language you use.

Another application would be to use the same compile directory across multiple domains / multiple virtual hosts.

Example 12.5. $compile_id in a virtual host environment

```
<?php

$smarty->compile_id = $_SERVER['SERVER_NAME'];
$smarty->compile_dir = '/path/to/shared_compile_dir';

?>
```

$use_sub_dirs

Smarty will create subdirectories under the compiled templates and cache directories if *`$use_sub_dirs`* is set to `TRUE`, default is `FALSE`. In an environment where there are potentially tens of thousands of files created, this may help the filesystem speed. On the other hand, some environments do not allow PHP processes to create directories, so this must be disabled which is the default.

Sub directories are more efficient, so use them if you can. Theoretically you get much better perfomance on a filesystem with 10 directories each having 100 files, than with 1 directory having 1000 files. This was certainly the case with Solaris 7 (UFS)... with newer filesystems such as ext3 and especially reiserfs, the difference is almost nothing.

Technical Note

- `$use_sub_dirs=true` doesn't work with safe_mode=On [http://php.net/features.safe-mode], that's why it's switchable and why it's off by default.
- `$use_sub_dirs=true` on Windows can cause problems.
- Safe_mode is being deprecated in PHP6.

See also *`$compile_id`*, *`$cache_dir`*, and *`$compile_dir`*.

$default_modifiers

This is an array of modifiers to implicitly apply to every variable in a template. For example, to HTML-escape every variable by default, use `array('escape:"htmlall"')`. To make a variable exempt from default modifiers, pass the special `smarty` modifier with a parameter value of `nodefaults` modifier to it, such as `{$var|smarty:nodefaults}`.

$default_resource_type

This tells smarty what resource type to use implicitly. The default value is `file`, meaning that `$smarty->display('index.tpl')` and `$smarty->display('file:index.tpl')` are identical in meaning. See the resource chapter for more details.

Chapter 13. Smarty Class Methods()

Note

See `Changing settings by template` section for how to use the functions for individual templates.

Name

append() — append an element to an assigned array

Description

```
void append(mixed var);
```

```
void append(string varname,
            mixed var,
            bool merge);
```

If you append to a string value, it is converted to an array value and then appended to. You can explicitly pass name/value pairs, or associative arrays containing the name/value pairs. If you pass the optional third parameter of `TRUE`, the value will be merged with the current array instead of appended.

Technical Note

The *`merge`* parameter respects array keys, so if you merge two numerically indexed arrays, they may overwrite each other or result in non-sequential keys. This is unlike the PHP `array_merge()` [http://php.net/array_merge] function which wipes out numerical keys and renumbers them.

Example 13.1. append

```
<?php
// This is effectively the same as assign()
$smarty->append('foo', 'Fred');
// After this line, foo will now be seen as an array in the template
$smarty->append('foo', 'Albert');

$array = array(1 => 'one', 2 => 'two');
$smarty->append('X', $array);
$array2 = array(3 => 'three', 4 => 'four');
// The following line will add a second element to the X array
$smarty->append('X', $array2);

// passing an associative array
$smarty->append(array('city' => 'Lincoln', 'state' => 'Nebraska'));
?>
```

See also `appendByRef()`, `assign()` and `getTemplateVars()`

Name

appendByRef() — append values by reference

Description

```
void appendByRef(string varname,
                 mixed var,
                 bool merge);
```

This is used to `append()` values to the templates by reference.

Technical Note

With the introduction of PHP5, `appendByRef()` is not necessary for most intents and purposes. `appendByRef()` is useful if you want a PHP array index value to be affected by its reassignment from a template. Assigned object properties behave this way by default.

Technical Note

The *`merge`* parameter respects array keys, so if you merge two numerically indexed arrays, they may overwrite each other or result in non-sequential keys. This is unlike the PHP `array_merge()` [http://php.net/array_merge] function which wipes out numerical keys and renumbers them.

Example 13.2. appendByRef

```
<?php
// appending name/value pairs
$smarty->appendByRef('Name', $myname);
$smarty->appendByRef('Address', $address);
?>
```

See also `append()`, `assign()` and `getTemplateVars()`.

Name

assign() — assign variables/objects to the templates

Description

```
void assign(mixed var);


void assign(string varname,
            mixed var);
```

You can explicitly pass name/value pairs, or associative arrays containing the name/value pairs.

Note

When you assign/register objects to templates, be sure that all properties and methods accessed from the template are for presentation purposes only. It is very easy to inject application logic through objects, and this leads to poor designs that are difficult to manage. See the Best Practices section of the Smarty website.

Example 13.3. assign()

```
<?php
// passing name/value pairs
$smarty->assign('Name', 'Fred');
$smarty->assign('Address', $address);

// passing an associative array
$smarty->assign(array('city' => 'Lincoln', 'state' => 'Nebraska'));

// passing an array
$myArray = array('no' => 10, 'label' => 'Peanuts');
$smarty->assign('foo',$myArray);

// passing a row from a database (eg adodb)
$sql = 'select id, name, email from contacts where contact ='.$id;
$smarty->assign('contact', $db->getRow($sql));
?>
```

These are accessed in the template with

```
{* note the vars are case sensitive like php *}
{$Name}
{$Address}
{$city}
{$state}

{$foo.no}, {$foo.label}
{$contact.id}, {$contact.name},{$contact.email}
```

To access more complex array assignments see `{foreach}` and `{section}`

See also `assignByRef()`, `getTemplateVars()`, `clearAssign()`, `append()` and `{assign}`

Name

assignByRef() — assign values by reference

Description

```
void assignByRef(string varname,
                 mixed var);
```

This is used to `assign()` values to the templates by reference.

Technical Note

With the introduction of PHP5, `assignByRef()` is not necessary for most intents and purposes. `assignByRef()` is useful if you want a PHP array index value to be affected by its reassignment from a template. Assigned object properties behave this way by default.

Example 13.4. assignByRef()

```
<?php
// passing name/value pairs
$smarty->assignByRef('Name', $myname);
$smarty->assignByRef('Address', $address);
?>
```

See also `assign()`, `clearAllAssign()`, `append()`, `{assign}` and `getTemplateVars()`.

Name

clearAllAssign() — clears the values of all assigned variables

Description

```
void clearAllAssign();
```

Example 13.5. clearAllAssign()

```
<?php
// passing name/value pairs
$smarty->assign('Name', 'Fred');
$smarty->assign('Address', $address);

// will output above
print_r( $smarty->getTemplateVars() );

// clear all assigned variables
$smarty->clearAllAssign();

// will output nothing
print_r( $smarty->getTemplateVars() );

?>
```

See also `clearAssign()`, `clearConfig()`, `getTemplateVars()`, `assign()` and `append()`

Name

clearAllCache() — clears the entire template cache

Description

```
void clearAllCache(int expire_time);
```

As an optional parameter, you can supply a minimum age in seconds the cache files must be before they will get cleared.

Example 13.6. clearAllCache

```
<?php
// clear the entire cache
$smarty->cache->clearAll();

// clears all files over one hour old
$smarty->cache->clearAll(3600);
?>
```

See also `clearCache()`, `isCached()` and the caching page.

Name

clearAssign() — clears the value of an assigned variable

Description

```
void clearAssign(mixed var);
```

This can be a single value, or an array of values.

Example 13.7. clearAssign()

```
<?php
// clear a single variable
$smarty->clearAssign('Name');

// clears multiple variables
$smarty->clearAssign(array('Name', 'Address', 'Zip'));
?>
```

See also `clearAllAssign()`, `clearConfig()`, `getTemplateVars()`, `assign()` and `append()`

Name

clearCache() — clears the cache for a specific template

Description

```
void clearCache(string template,
                string cache_id,
                string compile_id,
                int expire_time);
```

- If you have multiple caches for a template, you can clear a specific cache by supplying the *cache_id* as the second parameter.
- You can also pass a *$compile_id* as a third parameter. You can group templates together so they can be removed as a group, see the caching section for more information.
- As an optional fourth parameter, you can supply a minimum age in seconds the cache file must be before it will get cleared.

Example 13.8. clearCache()

```
<?php
// clear the cache for a template
$smarty->cache->clear('index.tpl');

// clear the cache for a particular cache id in an multiple-cache template
$smarty->cache->clear('index.tpl', 'MY_CACHE_ID');
?>
```

See also `clearAllCache()` and `caching` section.

Name

clearCompiledTemplate() — clears the compiled version of the specified template resource

Description

```
void clearCompiledTemplate(string tpl_file,
                           string compile_id,
                           int exp_time);
```

This clears the compiled version of the specified template resource, or all compiled template files if one is not specified. If you pass a *`$compile_id`* only the compiled template for this specific *`$compile_id`* is cleared. If you pass an exp_time, then only compiled templates older than *`exp_time`* seconds are cleared, by default all compiled templates are cleared regardless of their age. This function is for advanced use only, not normally needed.

Example 13.9. clearCompiledTemplate()

```
<?php
// clear a specific template resource
$smarty->clearCompiledTemplate('index.tpl');

// clear entire compile directory
$smarty->clearCompiledTemplate();
?>
```

See also `clearCache()`.

Name

clearConfig() — clears assigned config variables

Description

```
void clearConfig(string var);
```

This clears all assigned config variables. If a variable name is supplied, only that variable is cleared.

Example 13.10. clearConfig()

```
<?php
// clear all assigned config variables.
$smarty->clearConfig();

// clear one variable
$smarty->clearConfig('foobar');
?>
```

See also `getConfigVars()`, `config variables`, `config files`, `{config_load}`, `configLoad()` and `clearAssign()`.

Name

compileAllConfig() — compiles all known config files

Description

```
string compileAllConfig(string extension,
                        boolean force,
                        integer timelimit,
                        integer maxerror);
```

This function compiles config files found in the `$config_dir` folder. It uses the following parameters:

- *`extension`* is an optional string which defines the file extention for the config files. The default is ".conf".
- *`force`* is an optional boolean which controls if only modified (false) or all (true) config files shall be compiled. The default is "false".
- *`timelimit`* is an optional integer to set a runtime limit in seconds for the compilation process. The default is no limit.
- *`maxerror`* is an optional integer to set an error limit. If more config files failed to compile the function will be aborted. The default is no limit.

Note

This function may not create desired results in all configurations. Use is on own risk.

Example 13.11. compileAllTemplates()

```


<?php
include('Smarty.class.php');
$smarty = new Smarty;

// force compilation of all config files
$smarty->compileAllConfig('.config',true);

?>
```

Name

compileAllTemplates() — compiles all known templates

Description

```
string compileAllTemplates(string extension,
                           boolean force,
                           integer timelimit,
                           integer maxerror);
```

This function compiles template files found in the `$template_dir` folder. It uses the following parameters:

- *`extension`* is an optional string which defines the file extention for the template files. The default is ".tpl".

- *`force`* is an optional boolean which controls if only modified (false) or all (true) templates shall be compiled. The default is "false".

- *`timelimit`* is an optional integer to set a runtime limit in seconds for the compilation process. The default is no limit.

- *`maxerror`* is an optional integer to set an error limit. If more templates failed to compile the function will be aborted. The default is no limit.

Note

This function may not create desired results in all configurations. Use is on own risk.

Note

If any template requires registered plugins, filters or objects you must register all of them before running this function.

Note

If you are using template inheritance this function will create compiled files of parent templates which will never be used.

Example 13.12. compileAllTemplates()

```


<?php
include('Smarty.class.php');
$smarty = new Smarty;

// force compilation of all template files
$smarty->compileAllTemplates('.tpl',true);

?>
```

Name

configLoad() — loads config file data and assigns it to the template

Description

```
void configLoad(string file,
                string section);
```

This loads config file data and assigns it to the template. This works identically to the template `{config_load}` function.

Technical Note

As of Smarty 2.4.0, assigned template variables are kept across invocations of `fetch()` and `display()`. Config vars loaded from `configLoad()` are always global in scope. Config files are also compiled for faster execution, and respect the *`$force_compile`* and *`$compile_check`* settings.

Example 13.13. configLoad()

```
<?php
// load config variables and assign them
$smarty->configLoad('my.conf');

// load a section
$smarty->configLoad('my.conf', 'foobar');
?>
```

See also `{config_load}`, `getConfigVars()`, `clearConfig()`, and `config variables`

Name

createData() — creates a data object

Description

```
string createData(object parent);
```

```
string createData();
```

This creates a data object which will hold assigned variables. It uses the following parameters:

- *parent* is an optional parameter. It is an uplink to the main Smarty object, a another user-created data object or to user-created template object. These objects can be chained. Templates can access variables assigned to any of the objects in it's parent chain.

Data objects are used to create scopes for assigned variables. They can be used to have controll which variables are seen by which templates.

Example 13.14. createData()

```
<?php
include('Smarty.class.php');
$smarty = new Smarty;

// create data object with its private variable scope
$data = $smarty->createData();

// assign variable to data scope
$data->assign('foo','bar');

// create template object which will use variables from data object
$tpl = $smarty->createTemplate('index.tpl',$data);

// display the template
$tpl->display();
?>
```

See also `display()`, and `createTemplate()`,

Name

createTemplate() — returns a template object

Description

```
string createTemplate(string template,
                      object parent);
```

```
string createTemplate(string template,
                      array data);
```

```
string createTemplate(string template,
                      string cache_id,
                      string compile_id,
                      object parent);
```

```
string createTemplate(string template,
                      string cache_id,
                      string compile_id,
                      array data);
```

This creates a template object which later can be rendered by the display or fetch method. It uses the following parameters:

- *template* must be a valid template resource type and path.
- *cache_id* is an optional parameter. You can also set the *$cache_id* variable once instead of passing this to each call to this function. It is used in the event that you want to cache different content of the same template, such as pages for displaying different products. See also the caching section for more information.
- *compile_id* is an optional parameter. You can also set the *$compile_id* variable once instead of passing this to each call to this function. It is used in the event that you want to compile different versions of the same template, such as having separate templates compiled for different languages.
- *parent* is an optional parameter. It is an uplink to the main Smarty object, a user-created data object or to another user-created template object. These objects can be chained. The template can access only variables assigned to any of the objects in the parent chain.
- *data* is an optional parameter. It is an associative array containing the name/value pairs of variables which get assigned to the object.

Example 13.15. createTemplate()

```
<?php
include('Smarty.class.php');
$smarty = new Smarty;

// create template object with its private variable scope
$tpl = $smarty->createTemplate('index.tpl');

// assign variable to template scope
$tpl->assign('foo','bar');

// display the template
$tpl->display();
?>
```

See also `display()`, and `templateExists()`.

Name

disableSecurity() — disables template security

Description

```
string disableSecurity();
```

This disables securty checking on templates.

See also `enableSecurity()`, and Security.

Name

display() — displays the template

Description

```
void display(string template,
             string cache_id,
             string compile_id);
```

This displays the template unlike `fetch()`. Supply a valid template resource type and path. As an optional second parameter, you can pass a *$cache id*, see the caching section for more information.

As an optional third parameter, you can pass a *$compile_id*. This is in the event that you want to compile different versions of the same template, such as having separate templates compiled for different languages. Another use for *$compile_id* is when you use more than one *$template_dir* but only one *$compile_dir*. Set a separate *$compile_id* for each *$template_dir*, otherwise templates of the same name will overwrite each other. You can also set the *$compile_id* variable once instead of passing this to each call to this function.

Example 13.16. display()

```
<?php
include(SMARTY_DIR.'Smarty.class.php');
$smarty = new Smarty();
$smarty->setCaching(true);

// only do db calls if cache doesn't exist
if(!$smarty->isCached('index.tpl')) {

  // dummy up some data
  $address = '245 N 50th';
  $db_data = array(
               'City' => 'Lincoln',
               'State' => 'Nebraska',
               'Zip' => '68502'
             );

  $smarty->assign('Name', 'Fred');
  $smarty->assign('Address', $address);
  $smarty->assign('data', $db_data);

}

// display the output
$smarty->display('index.tpl');
?>
```

Example 13.17. Other display() template resource examples

Use the syntax for template resources to display files outside of the *`$template_dir`* directory.

```
<?php
// absolute filepath
$smarty->display('/usr/local/include/templates/header.tpl');

// absolute filepath (same thing)
$smarty->display('file:/usr/local/include/templates/header.tpl');

// windows absolute filepath (MUST use "file:" prefix)
$smarty->display('file:C:/www/pub/templates/header.tpl');

// include from template resource named "db"
$smarty->display('db:header.tpl');
?>
```

See also `fetch()` and `templateExists()`.

Name

enableSecurity() — enables template security

Description

```
string enableSecurity(string securityclass);
```

```
string enableSecurity(object securityobject);
```

```
string enableSecurity();
```

This enables securty checking on templates. It uses the following parameters:

- *`securityclass`* is an optional parameter. It's the name of the class with defines the security policy parameters.

- *`securityobject`* is an optional parameter. It's the object with defines the security policy parameters.

For the details how to setup a security policy see the Security section.

See also `disableSecurity()`, and Security.

Name

fetch() — returns the template output

Description

```
string fetch(string template,
             string cache_id,
             string $compile_id);
```

This returns the template output instead of displaying it. Supply a valid template resource type and path. As an optional second parameter, you can pass a *$cache id*, see the caching section for more information.

As an optional third parameter, you can pass a *$compile_id*. This is in the event that you want to compile different versions of the same template, such as having separate templates compiled for different languages. Another use for *$compile_id* is when you use more than one *$template_dir* but only one *$compile_dir*. Set a separate *$compile_id* for each *$template_dir*, otherwise templates of the same name will overwrite each other. You can also set the *$compile_id* variable once instead of passing this to each call to this function.

Example 13.18. fetch()

```
<?php
include('Smarty.class.php');
$smarty = new Smarty;

$smarty->setCaching(true);

// only do db calls if cache doesn't exist
if(!$smarty->isCached('index.tpl')) {

  // dummy up some data
  $address = '245 N 50th';
  $db_data = array(
               'City' => 'Lincoln',
               'State' => 'Nebraska',
               'Zip' => '68502'
             );

  $smarty->assign('Name','Fred');
  $smarty->assign('Address',$address);
  $smarty->assign($db_data);

}

// capture the output
$output = $smarty->fetch('index.tpl');

// do something with $output here
echo $output;
?>
```

Example 13.19. Using fetch() to send an email

The `email_body.tpl` template

```
Dear {$contact.name},

Welcome and thankyou for signing up as a member of our user group,

Click on the link below to login with your user name of '{$contact.login_id}'
so you can post in our forums.

http://{$smarty.server.SERVER_NAME}/login/

List master
Some user group

{include file='email_disclaimer.tpl'}
```

The `email_disclaimer.tpl` template which uses the `{textformat}` modifier.

```
{textformat wrap=40}
Unless you are named "{$contact.name}", you may read only the "odd numbered
words" (every other word beginning with the first) of the message above. If you ha
violated that, then you hereby owe the sender 10 GBP for each even
numbered word you have read
{/textformat}
```

The php script using the PHP `mail()` [http://php.net/function.mail] function

```
<?php

// get contact from database eg using pear or adodb
$query  = 'select name, email, login_id from contacts where contact_id='.$contact_
$contact = $db->getRow($sql);
$smarty->assign('contact', $contact);

mail($contact['email'], 'Subject', $smarty->fetch('email_body.tpl'));

?>
```

See also `{fetch}` `display()`, `{eval}`, and `templateExists()`.

Name

getConfigVars() — returns the given loaded config variable value

Description

```
array getConfigVars(string varname);
```

If no parameter is given, an array of all loaded config variables is returned.

Example 13.20. getConfigVars()

```
<?php

// get loaded config template var #foo#
$myVar = $smarty->getConfigVars('foo');

// get all loaded config template vars
$all_config_vars = $smarty->getConfigVars();

// take a look at them
print_r($all_config_vars);
?>
```

See also `clearConfig()`, `{config_load}`, `configLoad()` and `getTemplateVars()`.

Name

getRegisteredObject() — returns a reference to a registered object

Description

```
array getRegisteredObject(string object_name);
```

This is useful from within a custom function when you need direct access to a registered object. See the objects page for more info.

Example 13.21. getRegisteredObject()

```
<?php
function smarty_block_foo($params, $smarty)
{
  if (isset($params['object'])) {
    // get reference to registered object
    $obj_ref = $smarty->getRegisteredObject($params['object']);
    // use $obj_ref is now a reference to the object
  }
}
?>
```

See also `registerObject()`, `unregisterObject()` and objects page

Name

getTags() — return tags used by template

Description

```
string getTags(object template);
```

This function returns an array of tagname/attribute pairs for all tags used by the template. It uses the following parameters:

- *`template`* is the template object.

Note

This function is experimental.

Example 13.22. getTags()

```
<?php
include('Smarty.class.php');
$smarty = new Smarty;

// create template object
$tpl = $smarty->createTemplate('index.tpl');

// get tags
$tags = $smarty->getTags($tpl);

print_r($tags);

?>
```

Name

getTemplateVars() — returns assigned variable value(s)

Description

```
array getTemplateVars(string varname);
```

If no parameter is given, an array of all assigned variables are returned.

Example 13.23. getTemplateVars

```
<?php
// get assigned template var 'foo'
$myVar = $smarty->getTemplateVars('foo');

// get all assigned template vars
$all_tpl_vars = $smarty->getTemplateVars();

// take a look at them
print_r($all_tpl_vars);
?>
```

See also `assign()`, `{assign}`, `append()`, `clearAssign()`, `clearAllAssign()` and `getConfigVars()`

Name

isCached() — returns true if there is a valid cache for this template

Description

```
bool isCached(string template,
              string cache_id,
              string compile_id);
```

- This only works if *$caching* is set to TRUE, see the caching section for more info.
- You can also pass a *$cache_id* as an optional second parameter in case you want multiple caches for the given template.
- You can supply a *$compile id* as an optional third parameter. If you omit that parameter the persistent *$compile_id* is used if its set.
- If you do not want to pass a *$cache_id* but want to pass a *$compile_id* you have to pass NULL as a *$cache_id*.

Technical Note

If isCached() returns TRUE it actually loads the cached output and stores it internally. Any subsequent call to display() or fetch() will return this internally stored output and does not try to reload the cache file. This prevents a race condition that may occur when a second process clears the cache between the calls to isCached() and to display() in the example above. This also means calls to clearCache() and other changes of the cache-settings may have no effect after isCached() returned TRUE.

Example 13.24. isCached()

```
<?php
$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

if(!$smarty->is_cached('index.tpl')) {
// do database calls, assign vars here
}

$smarty->display('index.tpl');
?>
```

Example 13.25. isCached() with multiple-cache template

```
<?php
$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

if(!$smarty->is_cached('index.tpl', 'FrontPage')) {
  // do database calls, assign vars here
}

$smarty->display('index.tpl', 'FrontPage');
?>
```

See also `clearCache()`, `clearAllCache()`, and caching section.

Name

loadFilter() — load a filter plugin

Description

```
void loadFilter(string type,
                string name);
```

The first argument specifies the type of the filter to load and can be one of the following: `pre`, `post` or `output`. The second argument specifies the *`name`* of the filter plugin.

Example 13.26. Loading filter plugins

```
<?php

// load prefilter named 'trim'
$smarty->loadFilter('pre', 'trim');

// load another prefilter named 'datefooter'
$smarty->loadFilter('pre', 'datefooter');

// load output filter named 'compress'
$smarty->loadFilter('output', 'compress');

?>
```

See also `registerFilter()`, *`$autoload_filters`* and advanced features.

Name

registerFilter() — dynamically register filters

Description

```
void registerFilter(string type,
                    mixed callback);
```

Use this to dynamically register filters to operate on a templates. It uses the following parameters:

- *`type`* defines the type of the filter. Valid values are "pre", "post", "output" and "variable".
- *`callback`* defines the PHP callback. it can be either:
  - A string containing the function *`name`*
  - An array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name
  - An array of the form `array($class, $method)` with `$class` being the class name and `$method` being a method of the class.

Technical Note

If the chosen *`function`* callback is of the form `array(&$object, $method)`, only one instance of the same class and with the same `$method` can be registered. The latest registered *`function`* callback will be used in such a scenario.

A prefilter runs through the template source before it gets compiled. See template prefilters for more information on how to setup a prefiltering function.

A postfilter runs through the template code after it was compiled to PHP. See template postfilters for more information on how to setup a postfiltering function.

A outputfilter operates on a template's output before it is displayed. See template output filters for more information on how to set up an output filter function.

See also `unregisterFilter()`, `loadFilter()`, *`$autoload_filters`*, template pre filters template post filters template output filters section.

Name

registerPlugin() — dynamically register plugins

Description

```
void registerPlugin(string type,
                    string name,
                    mixed callback,
                    bool cacheable,
                    mixed cache_attrs);
```

This method registers functions or methods defined in your script as plugin. It uses the following parameters:

- *`type`* defines the type of the plugin. Valid values are "function", "block", "compiler" and "modifier".
- *`name`* defines the name of the plugin.
- *`callback`* defines the PHP callback. it can be either:
  - A string containing the function *`name`*
  - An array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name
  - An array of the form `array($class, $method)` with `$class` being the class name and `$method` being a method of the class.
- *`cacheable`* and *`cache_attrs`* can be omitted in most cases. See controlling cacheability of plugins output on how to use them properly.

Example 13.27. register a function plugin

```
<?php
$smarty->registerPlugin("function","date_now", "print_current_date");

function print_current_date($params, $smarty)
{
  if(empty($params["format"])) {
    $format = "%b %e, %Y";
  } else {
    $format = $params["format"];
  }
  return strftime($format,time());
}
?>
```

And in the template

```
{date_now}

{* or to format differently *}
{date_now format="%Y/%m/%d"}
```

Example 13.28. register block function plugin

```
<?php
// function declaration
function do_translation ($params, $content, $smarty, &$repeat, $template)
{
  if (isset($content)) {
    $lang = $params["lang"];
    // do some translation with $content
    return $translation;
  }
}

// register with smarty
$smarty->registerPlugin("block","translate", "do_translation");
?>
```

Where the template is:

```
{translate lang="br"}Hello, world!{/translate}
```

Example 13.29. register modifier plugin

```
<?php

// let's map PHP's stripslashes function to a Smarty modifier.
$smarty->registerPlugin("modifier","ss", "stripslashes");

?>
```

In the template, use `ss` to strip slashes.

```
<?php
{$var|ss}
?>
```

See also `unregisterPlugin()`, plugin functions, plugin block functions, plugin compiler functions, and the creating plugin modifiers section.

Name

registerObject() — register an object for use in the templates

Description

```
void registerObject(string object_name,
                    object object,
                    array allowed_methods_properties,
                    boolean format,
                    array block_methods);
```

Note

When you register/assign objects to templates, be sure that all properties and methods accessed from the template are for presentation purposes only. It is very easy to inject application logic through objects, and this leads to poor designs that are difficult to manage. See the Best Practices section of the Smarty website.

See the objects section for more information.

See also `getRegisteredObject()`, and `unregisterObject()`.

Name

registerResource() — dynamically register resources

Description

```
void registerResource(string name,
                      array resource_funcs);
```

Use this to dynamically register a resource plugin with Smarty. Pass in the *`name`* of the resource and the array of PHP functions implementing it. See template resources for more information on how to setup a function for fetching templates.

Technical Note

A resource name must be at least two characters in length. One character resource names will be ignored and used as part of the file path, such as `$smarty->display('c:/path/to/index.tpl');`

- The php-function-array *`resource_funcs`* must have 4 or 5 elements.
- With 4 elements the elements are the functions-callbacks for the respective `source`, `timestamp`, `secure` and `trusted` functions of the resource.
- With 5 elements the first element has to be an object reference or a class name of the object or class implementing the resource and the 4 following elements have to be the method names implementing `source`, `timestamp`, `secure` and `trusted`.

Example 13.30. registerResource()

```
<?php
$smarty->register->resource('db', array(
                                'db_get_template',
                                'db_get_timestamp',
                                'db_get_secure',
                                'db_get_trusted')
                                );
?>
```

See also `unregisterResource()` and the template resources section.

Name

templateExists() — checks whether the specified template exists

Description

```
bool templateExists(string template);
```

It can accept either a path to the template on the filesystem or a resource string specifying the template.

Example 13.31. templateExists()

This example uses `$_GET['page']` to `{include}` a content template. If the template does not exist then an error page is displayed instead. First the `page_container.tpl`

```
<html>
<head><title>{$title}</title></head>
<body>
{include file='page_top.tpl'}

{* include middle content page *}
{include file=$content_template}

{include file='page_footer.tpl'}
</body>
```

And the php script

```
<?php

// set the filename eg index.inc.tpl
$mid_template = $_GET['page'].'.inc.tpl';

if( !$smarty->templateExists($mid_template) ){
    $mid_template = 'page_not_found.tpl';
}
$smarty->assign('content_template', $mid_template);

$smarty->display('page_container.tpl');

?>
```

See also `display()`, `fetch()`, `{include}` and `{insert}`

Name

unregisterFilter() — dynamically unregister a filter

Description

```
void unregisterFilter(string type,
                      mixed callback);
```

Use this to dynamically unregister filters. It uses the following parameters:

- *`type`* defines the type of the filter. Valid values are "pre", "post", "output" and "variable".
- *`callback`* defines the PHP callback. it can be either:
  - A string containing the function *`name`*
  - An array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name
  - An array of the form `array($class, $method)` with `$class` being the class name and `$method` being a method of the class.

See also `registerFilter()`.

Name

unregisterPlugin — dynamically unregister plugins

Description

```
void unregisterPlugin(string type,
                      string name);
```

This method unregisters plugins which previously have been registered by registerPlugin(), It uses the following parameters:

- `type` defines the type of the plugin. Valid values are "function", "block", "compiler" and "modifier".
- `name` defines the name of the plugin.

Example 13.32. unregister function plugin

```
<?php

// we don't want template designers to have access to function plugin "date_now"
$smarty->unregisterPlugin("function","date_now");

?>
```

See also `registerPlugin()`.

Name

unregisterObject() — dynamically unregister an object

Description

```
void unregisterObject(string object_name);
```

See also `registerObject()` and objects section

Name

unregisterResource() — dynamically unregister a resource plugin

Description

```
  void unregisterResource(string name);
```

Pass in the *`name`* of the resource.

Example 13.33. unregisterResource()

```
<?php

$smarty->unregister_resource('db');

?>
```

See also `registerResource()` and template resources

Name

testInstall() — checks Smarty installation

Description

```
void testInstall();
```

This function verifies that all required working folders of the Smarty installtion can be accessed. It does output a corresponding protocoll.

Example 13.34. testInstall()

```
<?php
require_once('Smarty.class.php');
$smarty->testInstall();
?>
```

Chapter 14. Caching

Caching is used to speed up a call to `display()` or `fetch()` by saving its output to a file. If a cached version of the call is available, that is displayed instead of regenerating the output. Caching can speed things up tremendously, especially templates with longer computation times. Since the output of `display()` or `fetch()` is cached, one cache file could conceivably be made up of several template files, config files, etc.

Since templates are dynamic, it is important to be careful what you are caching and for how long. For instance, if you are displaying the front page of your website that does not change its content very often, it might work well to cache this page for an hour or more. On the other hand, if you are displaying a page with a timetable containing new information by the minute, it would not make sense to cache this page.

Setting Up Caching

The first thing to do is enable caching by setting *`$caching`* `= 1 (or 2)`.

Example 14.1. Enabling caching

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

$smarty->display('index.tpl');
?>
```

With caching enabled, the function call to `display('index.tpl')` will render the template as usual, but also saves a copy of its output to a file (a cached copy) in the *`$cache_dir`*. On the next call to `display('index.tpl')`, the cached copy will be used instead of rendering the template again.

Technical Note

The files in the *`$cache_dir`* are named similar to the template name. Although they end in the `.php` extention, they are not intended to be directly executable. Do not edit these files!

Each cached page has a limited lifetime determined by *`$cache_lifetime`*. The default value is 3600 seconds ie an hour. After that time expires, the cache is regenerated. It is possible to give individual caches their own expiration time by setting *`$caching`*`=2`. See *`$cache_lifetime`* for more details.

Example 14.2. Setting $cache_lifetime per cache

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_SAVED; // lifetime is per cache

// set the cache_lifetime for index.tpl to 5 minutes
$smarty->cache_lifetime = 300;
$smarty->display('index.tpl');

// set the cache_lifetime for home.tpl to 1 hour
$smarty->cache_lifetime = 3600;
$smarty->display('home.tpl');

// NOTE: the following $cache_lifetime setting will not work when $caching = 2.
// The cache lifetime for home.tpl has already been set
// to 1 hour, and will no longer respect the value of $cache_lifetime.
// The home.tpl cache will still expire after 1 hour.
$smarty->cache_lifetime = 30; // 30 seconds
$smarty->display('home.tpl');
?>
```

If `$compile_check` is enabled, every template file and config file that is involved with the cache file is checked for modification. If any of the files have been modified since the cache was generated, the cache is immediately regenerated. This is a slight overhead so for optimum performance, set `$compile_check` to `FALSE`.

Example 14.3. Enabling $compile_check

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;
$smarty->compile_check = true;

$smarty->display('index.tpl');
?>
```

If `$force_compile` is enabled, the cache files will always be regenerated. This effectively turns off caching. `$force_compile` is usually for debugging purposes only, a more efficient way of disabling caching is to set `$caching = 0`.

The `is_cached()` function can be used to test if a template has a valid cache or not. If you have a cached template that requires something like a database fetch, you can use this to skip that process.

Example 14.4. Using is_cached()

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

if(!$smarty->is_cached('index.tpl')) {
    // No cache available, do variable assignments here.
    $contents = get_database_contents();
    $smarty->assign($contents);
}

$smarty->display('index.tpl');
?>
```

You can keep parts of a page dynamic with the `{insert}` template function. Let's say the whole page can be cached except for a banner that is displayed down the side of the page. By using the `{insert}` function for the banner, you can keep this element dynamic within the cached content. See the documentation on `{insert}` for more details and examples.

You can clear all the cache files with the `clear_all_cache()` function, or individual cache files and groups with the `clear_cache()` function.

Example 14.5. Clearing the cache

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

// clear only cache for index.tpl
$smarty->clear_cache('index.tpl');

// clear out all cache files
$smarty->clear_all_cache();

$smarty->display('index.tpl');
?>
```

Multiple Caches Per Page

You can have multiple cache files for a single call to `display()` or `fetch()`. Let's say that a call to `display('index.tpl')` may have several different output contents depending on some condition, and you want separate caches for each one. You can do this by passing a *$cache_id* as the second parameter to the function call.

Example 14.6. Passing a $cache_id to display()

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

$my_cache_id = $_GET['article_id'];

$smarty->display('index.tpl', $my_cache_id);
?>
```

Above, we are passing the variable `$my_cache_id` to `display()` as the *$cache_id*. For each unique value of `$my_cache_id`, a separate cache will be generated for `index.tpl`. In this example, `article_id` was passed in the URL and is used as the *$cache_id*.

Technical Note

Be very cautious when passing values from a client (web browser) into Smarty or any PHP application. Although the above example of using the article_id from the URL looks handy, it could have bad consequences. The *$cache_id* is used to create a directory on the file system, so if the user decided to pass an extremely large value for article_id, or write a script that sends random article_id's at a rapid pace, this could possibly cause problems at the server level. Be sure to sanitize any data passed in before using it. In this instance, maybe you know the article_id has a length of ten characters and is made up of alpha-numerics only, and must be a valid article_id in the database. Check for this!

Be sure to pass the same *$cache_id* as the second parameter to `is_cached()` and `clear_cache()`.

Example 14.7. Passing a cache_id to is_cached()

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

$my_cache_id = $_GET['article_id'];

if(!$smarty->is_cached('index.tpl',$my_cache_id)) {
    // No cache available, do variable assignments here.
    $contents = get_database_contents();
    $smarty->assign($contents);
}

$smarty->display('index.tpl',$my_cache_id);
?>
```

You can clear all caches for a particular *`$cache_id`* by passing `NULL` as the first parameter to `clear_cache()`.

Example 14.8. Clearing all caches for a particular $cache_id

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

// clear all caches with "sports" as the $cache_id
$smarty->clear_cache(null,'sports');

$smarty->display('index.tpl','sports');
?>
```

In this manner, you can “group” your caches together by giving them the same *`$cache_id`*.

Cache Groups

You can do more elaborate grouping by setting up *`$cache_id`* groups. This is accomplished by separating each sub-group with a vertical bar **|** in the *`$cache_id`* value. You can have as many sub-groups as you like.

- You can think of cache groups like a directory hierarchy. For instance, a cache group of `'a|b|c'` could be thought of as the directory structure `'/a/b/c/'`.

- `clear_cache(null,'a|b|c')` would be like removing the files `'/a/b/c/*'`. `clear_cache(null,'a|b')` would be like removing the files `'/a/b/*'`.
- If you specify a *`$compile_id`* such as `clear_cache(null,'a|b','foo')` it is treated as an appended cache group `'/a/b/c/foo/'`.
- If you specify a template name such as `clear_cache('foo.tpl','a|b|c')` then Smarty will attempt to remove `'/a/b/c/foo.tpl'`.
- You CANNOT remove a specified template name under multiple cache groups such as `'/a/b/*/foo.tpl'`, the cache grouping works left-to-right ONLY. You will need to group your templates under a single cache group heirarchy to be able to clear them as a group.

Cache grouping should not be confused with your template directory heirarchy, the cache grouping has no knowledge of how your templates are structured. So for example, if you have a template structure like `themes/blue/index.tpl` and you want to be able to clear all the cache files for the “blue” theme, you will need to create a cache group structure that mimics your template file structure, such as `display('themes/blue/index.tpl','themes|blue')`, then clear them with `clear_cache(null,'themes|blue')`.

Example 14.9. $cache_id groups

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

// clear all caches with 'sports|basketball' as the first two cache_id groups
$smarty->clear_cache(null,'sports|basketball');

// clear all caches with "sports" as the first cache_id group. This would
// include "sports|basketball", or "sports|(anything)|(anything)|(anything)|..."
$smarty->clear_cache(null,'sports');

// clear the foo.tpl cache file with "sports|basketball" as the cache_id
$smarty->clear_cache('foo.tpl','sports|basketball');


$smarty->display('index.tpl','sports|basketball');
?>
```

Controlling Cacheability of Plugins' Output

The cacheability of plugins can be declared when registering them. The third parameter to `registerPlugin()` is called *`$cacheable`* and defaults to `TRUE`.

When registering a plugin with `$cacheable=false` the plugin is called everytime the page is displayed, even if the page comes from the cache. The plugin function behaves a little like an `{insert}` function.

In contrast to `{insert}` the attributes to the plugins are not cached by default. They can be declared to be cached with the fourth parameter *`$cache_attrs`*. *`$cache_attrs`* is an array of attribute-names that should be cached, so the plugin-function get value as it was the time the page was written to cache everytime it is fetched from the cache.

Example 14.10. Preventing a plugin's output from being cached

```
<?php
$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

function remaining_seconds($params, $smarty) {
    $remain = $params['endtime'] - time();
    if($remain >= 0){
        return $remain . ' second(s)';
    }else{
        return 'done';
    }
}

$smarty->registerPlugin('function','remaining', 'remaining_seconds', false, array(

if (!$smarty->isCached('index.tpl')) {
    // fetch $obj from db and assign...
    $smarty->assignByRef('obj', $obj);
}

$smarty->display('index.tpl');
?>
```

where `index.tpl` is:

```
Time Remaining: {remaining endtime=$obj->endtime}
```

The number of seconds till the endtime of `$obj` is reached changes on each display of the page, even if the page is cached. Since the endtime attribute is cached the object only has to be pulled from the database when page is written to the cache but not on subsequent requests of the page.

Example 14.11. Preventing a whole passage of a template from being cached

```
index.php:

<?php
$smarty->caching = Smarty::CACHING_LIFETIME_CURRENT;

function smarty_block_dynamic($param, $content, $smarty) {
    return $content;
}
$smarty->registerPlugin('block','dynamic', 'smarty_block_dynamic', false);

$smarty->display('index.tpl');
?>
```

where `index.tpl` is:

```
Page created: {'0'|date_format:'%D %H:%M:%S'}

{dynamic}

Now is: {'0'|date_format:'%D %H:%M:%S'}

... do other stuff ...

{/dynamic}
```

When reloading the page you will notice that both dates differ. One is “dynamic” one is “static”. You can do everything between `{dynamic}...{/dynamic}` and be sure it will not be cached like the rest of the page.

Chapter 15. Advanced Features

Security

Security is good for situations when you have untrusted parties editing the templates eg via ftp, and you want to reduce the risk of system security compromises through the template language.

The settings of the security policy are defined by properties of an instance of the Smarty_Security class. These are the possible settings:

- *`$php_handling`* determines how Smarty to handle PHP code embedded in templates. Possible values are:

  - Smarty::PHP_PASSTHRU -> echo PHP tags as they are

  - Smarty::PHP_QUOTE -> escape tags as entities

  - Smarty::PHP_REMOVE -> remove php tags

  - Smarty::PHP_ALLOW -> execute php tags

  The default value is Smarty::PHP_PASSTHRU.

  If security is enabled the *`$php_handling`* setting of the Smarty object is not checked for security.

- *`$secure_dir`* is an array of template directories that are considered secure. *`$template_dir`* concidered secure implicitly. The default is an empty array.

- *`$trusted_dir`* is an array of all directories that are considered trusted. Trusted directories are where you keep php scripts that are executed directly from the templates with `{include_php}`. The default is an empty array.

- *`$static_classes`* is an array of classes that are considered trusted. The default is an empty array which allows access to all static classes. To disable access to all static classes set $static_classes = null.

- *`$php_functions`* is an array of PHP functions that are considered trusted and can be used from within template. To disable access to all PHP functions set $php_functions = null. An empty array ($php_functions = array()) will allow all PHP functions. The default is array('isset', 'empty', 'count', 'sizeof', 'in_array', 'is_array','time','nl2br').

- *`$php_modifiers`* is an array of PHP functions that are considered trusted and can be used from within template as modifier. To disable access to all PHP modifier set $php_modifier = null. An empty array ($php_modifier = array()) will allow all PHP functions. The default is array('escape','count').

- *`$streams`* is an array of streams that are considered trusted and can be used from within template. To disable access to all streams set $streams = null. An empty array ($streams = array()) will allow all streams. The default is array('file').

- *`$allow_constants`* is a boolean flag which controls if constants can be accessed by the template. The default is "true".

- *`$allow_super_globals`* is a boolean flag which controls if the PHP super globals can be accessed by the template. The default is "true".

- *`$allow_php_tag`* is a boolean flag which controls if {php} and {include_php} tags can be used by the template. The default is "false".

If security is enabled, no private methods, functions or properties of static classes or assigned objects can be accessed (beginningwith '_') by the template.

To customize the security policy settings you can extend the Smarty_Security class or create an instance of it.

Example 15.1. Setting security policy by extending the Smarty_Security class

```
<?php
require 'Smarty.class.php';

class My_Security_Policy extends Smarty_Security {
  // disable all PHP functions
  public $php_functions = null;
  // remove PHP tags
  public $php_handling = Smarty::PHP_REMOVE;
  // allow everthing as modifier
  public $modifiers = array();
}
$smarty = new Smarty;
// enable security
$smarty->enableSecurity('My_Security_Policy');
?>
```

Example 15.2. Setting security policy by instance of the Smarty_Security class

```
<?php
require 'Smarty.class.php';
$smarty = new Smarty;
$my_security_policy = new Smarty_Security;
// disable all PHP functions
$my_security_policy->php_functions = null;
// remove PHP tags
$my_security_policy->php_handling = Smarty::PHP_REMOVE;
// allow everthing as modifier
$my_security_policy->$modifiers = array();
// enable security
$smarty->enableSecurity($my_security_policy);
?>
```

Example 15.3. Enable security with the default settings

```
<?php
require 'Smarty.class.php';
$smarty = new Smarty;
// enable default security
$smarty->enableSecurity();
?>
```

Note

Must security policy settings are only checked when the template gets compiled. For that reasion you should delete all cached and compiled template files when you change your security settings.

Changing settings by template

Normally you configure the Smarty settings by modifying the `Smarty class variables`. Furthermore you can register plugins, filters etc. with `Smarty functions`. Modifications done to the Smarty object will be global for all templates.

However the Smarty class variables and functions can be accessed or called by induvidual template objects. Modification done to a template object will apply only for that template and its included subtemplates.

Example 15.4. changing Smarty settings by template

```
<?php
$tpl = $smarty->createTemplate('index.tpl);
$tpl->cache_lifetime = 600;
//or
$tpl->setCacheLifetime(600);
$smarty->display($tpl);
?>
```

Example 15.5. register plugins by template

```
<?php
$tpl = $smarty->createTemplate('index.tpl);
$tpl->registerPlugin('modifier','mymodifier');
$smarty->display($tpl);
?>
```

Template Inheritance

Inheritance brings the concept of Object Oriented Programming to templates, allowing you to define one (or more) base templates that can be extended by child templates. Extending means that the child template can override all or some of the parent named block areas.

- The inheritance tree can be as big as you want (meaning you can extend a file that extends another one that extends another one and so on..), but be aware that all files have to be checked for modifications at runtime so the more inheritance the more overhead you add.
- The child templates can not define any content besides what's inside `{block}` tags they override, anything outside of `{block}` tags will be removed.
- The content of `{block}` tags from child and parent templates can be merged by the `append` or `prepend` `{block}` tag option flags and `{$smarty.block.parent}` or `{$smarty.block.child}` placeholders.
- Template inheritance is a compile time process which does create a single compiled template file. Compared to corresponding solutions based on subtemplates included with the `{include}` tag it does have much better performance when redering.
- The child template does extend its parent defined with the `{extends}` tag, which must be the first line in the child template. Instead of using the `{extends}` tags in the template files you can define the whole template inheritance tree in the PHP script when you are calling `fetch()` or `display()` with the `extends:` template resource type. The later provides even more flexibillity.

Note

If you have a subtemplate which is included with `{include}` and it does contain `{block}` areas it does work only if the `{include}` itself is called from within a surrounding `{block}`. In the final parent template you may need a dummy `{block}` for it.

Example 15.6. Template inheritance example

layout.tpl (parent)

```
<html>
<head>
  <title>{block name=title}Default Page Title{/block}</title>
  <span style="color: blue">{block name=head}{/block}</span>
</head>
<body>
{block name=body}{/block}
</body>
</html>
```

myproject.tpl (child)

```
{extends file=layout.tpl}
{block name=head}
  <link href="/css/mypage.css" rel="stylesheet" type="text/css"/>
  <script src="/js/mypage.js"></script>
{/block}
```

myproject.tpl (grandchild)

```
{extends file=project.tpl}
{block name=title}My Page Title{/block}
{block name=head}
  <link href="/css/mypage.css" rel="stylesheet" type="text/css"/>
  <script src="/js/mypage.js"></script>
{/block}
{block name=body}My HTML Page Body goes here{/block}
```

To render the above use

```
 $smarty->display('mypage.tpl');
```

The resulting output is

```
<html>
<head>
  <title>My Page Title</title>
  <link href="/css/mypage.css" rel="stylesheet" type="text/css"/>
  <script src="/js/mypage.js"></script>
</head>
<body>
My HTML Page Body goes here
</body>
</html>
```

Example 15.7. Template inheritance by template resource `extends:`

Instead of using `{extends}` tags in the template files you can define the inheritance tree in your PHP script by using the `extends:` resource type.

The code below will return same result as the example above.

```
<?php
$smarty->display('extends:layout.tpl|myproject.tpl|mypage.tpl');
?>
```

See also `{block}`, `{extends}` and `extends:` resource

Streams

Streams allow you to use PHP streams as a template resource or a variable resource. The syntax is much the same a traditional template resource names.

Smarty will first look for a registered template resource. If nothing is found, it will check if a PHP stream is available. If a stream is available, Smarty will use it to fetch the template.

Note

You can further define allowed streams with security enabled.

Example 15.8. Stream from PHP

Using a PHP stream for a template resource from the display() function.

```
$smarty->display('foo:bar.tpl');
```

Example 15.9. Stream from Template

Using a PHP stream for a template resource from within a template.

```
{include file="foo:bar.tpl"}
```

You can also use streams to call variables. *{$foo:bar}* will use the *foo://bar* stream to get the template variable.

Example 15.10. Stream Variable

Using a PHP stream for a template variable resource from within a template.

```
{$foo:bar}
```

See also `Template Resources`

Objects

Smarty allows access to PHP objects [http://php.net/object] through the templates.

> ### Note
>
> When you assign/register objects to templates, be sure that all properties and methods accessed from the template are for presentation purposes only. It is very easy to inject application logic through objects, and this leads to poor designs that are difficult to manage. See the Best Practices section of the Smarty website.

There are two ways to access them.

- One way is to register objects to the template, then use access them via syntax similar to custom functions.
- The other way is to `assign()` objects to the templates and access them much like any other assigned variable.

The first method has a much nicer template syntax. It is also more secure, as a registered object can be restricted to certain methods or properties. However, **a registered object cannot be looped over or assigned in arrays of objects**, etc. The method you choose will be determined by your needs, but use the first method whenever possible to keep template syntax to a minimum.

If security is enabled, no private methods or functions can be accessed (beginningwith '_'). If a method and property of the same name exist, the method will be used.

You can restrict the methods and properties that can be accessed by listing them in an array as the third registration parameter.

By default, parameters passed to objects through the templates are passed the same way custom functions get them. An associative array is passed as the first parameter, and the smarty object as the second. If you want the parameters passed one at a time for each argument like traditional object parameter passing, set the fourth registration parameter to `FALSE`.

The optional fifth parameter has only effect with *`format`* being `TRUE` and contains a list of methods that should be treated as blocks. That means these methods have a closing tag in the template (`{foobar->meth2}...{/foobar->meth2}`) and the parameters to the methods have the same synopsis as the parameters for `block-function-plugins`: They get the four parameters *`$params`*, *`$content`*, *`&$smarty`* and *`&$repeat`* and they also behave like block-function-plugins.

Example 15.11. Using a registered or assigned object

```
<?php
// the object

class My_Object {
 function meth1($params, $smarty_obj) {
  return 'this is my meth1';
 }
}

$myobj = new My_Object;

// registering the object (will be by reference)
$smarty->register_object('foobar',$myobj);

// if we want to restrict access to certain methods or properties, list them
$smarty->register_object('foobar',$myobj,array('meth1','meth2','prop1'));

// if you want to use the traditional object parameter format, pass a boolean of f
$smarty->register_object('foobar',$myobj,null,false);

// We can also assign objects. assign_by_ref when possible.
$smarty->assign_by_ref('myobj', $myobj);

$smarty->display('index.tpl');
?>
```

And here's how to access your objects in `index.tpl`:

```
{* access our registered object *}
{foobar->meth1 p1='foo' p2=$bar}

{* you can also assign the output *}
{foobar->meth1 p1='foo' p2=$bar assign='output'}
the output was {$output}

{* access our assigned object *}
{$myobj->meth1('foo',$bar)}
```

See also `register_object()` and `assign()`.

Static Classes

You can directly access static classes. The syntax is the same as in PHP.

Note

Direct access to PHP classes is not recommended. This ties the underlying application code structure directly to the presentation, and also complicates template syntax. It is recommended to register plugins which insulate templates from PHP classes/objects. Use at your own discretion. See the Best Practices section of the Smarty website.

Example 15.12. static class access syntax

```
{assign var=foo value=myclass::BAR}  <--- class constant BAR

{assign var=foo value=myclass::method()}  <--- method result

{assign var=foo value=myclass::method1()->method2}  <--- method chaining

{assign var=foo value=myclass::$bar}  <--- property bar of class myclass

{assign var=foo value=$bar::method}  <--- using Smarty variable bar as class name
```

Prefilters

Template prefilters are PHP functions that your templates are ran through *before they are compiled*. This is good for preprocessing your templates to remove unwanted comments, keeping an eye on what people are putting in their templates, etc.

Prefilters can be either registered or loaded from the plugins directory by using `loadFilter()` function or by setting the *`$autoload_filters`* variable.

Smarty will pass the template source code as the first argument, and expect the function to return the resulting template source code.

Example 15.13. Using a template prefilter

This will remove all the html comments in the template source.

```
<?php
// put this in your application
function remove_dw_comments($tpl_source, $smarty)
{
    return preg_replace("/<!--#.*-->/U",'',$tpl_source);
}

// register the prefilter
$smarty->registerFilter('pre','remove_dw_comments');
$smarty->display('index.tpl');
?>
```

See also `registerFilter()`, postfilters and `loadFilter()`.

Postfilters

Template postfilters are PHP functions that your templates are ran through *after they are compiled*. Postfilters can be either registered or loaded from the plugins directory by using the `loadFilter()` function or by setting the *`$autoload_filters`* variable. Smarty will pass the compiled template code as the first argument, and expect the function to return the result of the processing.

Example 15.14. Using a template postfilter

```
<?php
// put this in your application
function add_header_comment($tpl_source, $smarty)
{
    return "<?php echo \"<!-- Created by Smarty! -->\n\"; ?>\n".$tpl_source;
}

// register the postfilter
$smarty->registerFilter('post','add_header_comment');
$smarty->display('index.tpl');
?>
```

The postfilter above will make the compiled Smarty template `index.tpl` look like:

```
<!-- Created by Smarty! -->
{* rest of template content... *}
```

See also `registerFilter()`, prefilters, outputfilters, and `loadFilter()`.

Output Filters

When the template is invoked via `display()` or `fetch()`, its output can be sent through one or more output filters. This differs from `postfilters` because postfilters operate on compiled templates before they are saved to the disk, whereas output filters operate on the template output when it is executed.

Output filters can be either registered or loaded from the plugins directory by using the `loadFilter()` method or by setting the *`$autoload_filters`* variable. Smarty will pass the template output as the first argument, and expect the function to return the result of the processing.

Example 15.15. Using a template outputfilter

```
<?php
// put this in your application
function protect_email($tpl_output, $smarty)
{
    $tpl_output =
       preg_replace('!(\S+)@([a-zA-Z0-9\.\-]+\.([a-zA-Z]{2,3}|[0-9]{1,3}))!',
                    '$1%40$2', $tpl_output);
    return $tpl_output;
}

// register the outputfilter
$smarty->registerFilter("output","protect_email");
$smarty->display("index.tpl');

// now any occurrence of an email address in the template output will have
// a simple protection against spambots
?>
```

See also `registerFilter()`, `loadFilter()`, *`$autoload_filters`*, postfilters and *`$plugins_dir`*.

Cache Handler Function

As an alternative to using the default file-based caching mechanism, you can specify a custom cache handling function that will be used to `read`, `write` and `clear` cached files.

Create a function in your application that Smarty will use as a cache handler. Set the name of it in the *`$cache_handler_func`* class variable. Smarty will now use this to handle cached data.

- The first argument is the action, which will be one of `read`, `write` and `clear`.
- The second parameter is the Smarty object.
- The third parameter is the cached content. Upon a `write`, Smarty passes the cached content in these parameters. Upon a `read`, Smarty expects your function to accept this parameter by reference and populate it with the cached data. Upon a `clear`, pass a dummy variable here since it is not used.
- The fourth parameter is the *`name`* of the template file, needed for read/write.
- The fifth parameter is the optional *`$cache_id`*.
- The sixth is the optional *`$compile_id`*.
- The seventh and last parameter *`$exp_time`* was added in Smarty-2.6.0.

Example 15.16. Example using MySQL as a cache source

```
 switch ($action) {
  case 'read':
   // read cache from database
   $results = mysql_query("select CacheContents from CACHE_PAGES where CacheID='$C
   if(!$results) {
    $smarty_obj->_triggerError_msg('cache_handler: query failed.');
   }
   $row = mysql_fetch_array($results,MYSQL_ASSOC);

   if($use_gzip && function_exists('gzuncompress')) {
    $cache_content = gzuncompress($row['CacheContents']);
   } else {
    $cache_content = $row['CacheContents'];
   }
   $return = $results;
   break;
  case 'write':
   // save cache to database

   if($use_gzip && function_exists("gzcompress")) {
    // compress the contents for storage efficiency
    $contents = gzcompress($cache_content);
   } else {
    $contents = $cache_content;
   }
   $results = mysql_query("replace into CACHE_PAGES values(
       '$CacheID',
       '".addslashes($contents)."')
      ");
   if(!$results) {
    $smarty_obj->_triggerError_msg('cache_handler: query failed.');
   }
   $return = $results;
   break;
  case 'clear':
   // clear cache info
   if(empty($cache_id) && empty($compile_id) && empty($tpl_file)) {
    // clear them all
    $results = mysql_query('delete from CACHE_PAGES');
   } else {
    $results = mysql_query("delete from CACHE_PAGES where CacheID='$CacheID'");
   }
   if(!$results) {
    $smarty_obj->_triggerError_msg('cache_handler: query failed.');
   }
   $return = $results;
   break;
  default:
   // error, unknown action
   $smarty_obj->_triggerError_msg("cache_handler: unknown action \"$action\"");
   $return = false;
   break;
 }
 mysql_close($link);
 return $return;

}

?>
```

Resources

The templates may come from a variety of sources. When you `display()` or `fetch()` a template, or when you include a template from within another template, you supply a resource type, followed by the appropriate path and template name. If a resource is not explicitly given, the value of *`$default_resource_type`* is assumed.

Templates from $template_dir

Templates from the *`$template_dir`* do not require a template resource, although you can use the `file:` resource for consistancy. Just supply the path to the template you want to use relative to the *`$template_dir`* root directory (no leading slash.)

Example 15.17. Using templates from the $template_dir

```
<?php
$smarty->display('index.tpl');
$smarty->display('file:index.tpl'); // same as above
?>
```

From within a Smarty template

```
{include file='index.tpl'}
{include file='file:index.tpl'} {* same as above *}
```

Templates from any directory

Templates outside of the *`$template_dir`* require the `file:` template resource type, followed by the absolute path to the template (with leading slash.)

Note

With security enabled, access to templates outside of the template_dir is not allowed.

Example 15.18. Using templates from any directory

```
<?php
$smarty->display('file:/export/templates/index.tpl');
$smarty->display('file:/path/to/my/templates/menu.tpl');
?>
```

And from within a Smarty template:

```
{include file='file:/usr/local/share/templates/navigation.tpl'}
```

Windows Filepaths

If you are using a Windows machine, filepaths usually include a drive letter (C:) at the beginning of the pathname. Be sure to use `file:` in the path to avoid namespace conflicts and get the desired results.

Example 15.19. Using templates from windows file paths

```
<?php
$smarty->display('file:C:/export/templates/index.tpl');
$smarty->display('file:F:/path/to/my/templates/menu.tpl');
?>
```

And from within Smarty template:

```
{include file='file:D:/usr/local/share/templates/navigation.tpl'}
```

Templates from strings

Smarty can render templates from a string by using the `string:` or `eval:` resource.

- The `string:` resource behaves much the same as a template file. The template source is compiled from a string and stores the compiled template code for later reuse. Each unique template string will create a new compiled template file. If your template strings are accessed frequently, this is a good choice. If you have frequently changing template strings (or strings with low reuse value), the `eval:` resource may be a better choice.

- The `eval:` resource evaluates the template source every time a page is rendered. This is a good choice for strings with low reuse value. If the same string is accessed frequently, the `string:` resource may be a better choice.

Note

With a `string:` resource type, each unique string generates a compiled file. Smarty cannot detect a string that has changed, and therefore will generate a new compiled file for each unique string. It is important to choose the correct resource so that you do not fill your disk space with wasted compiled strings.

Example 15.20. Using templates from strings

```
<?php
$smarty->assign('foo','value');
$template_string = 'display {$foo} here';
$smarty->display('string:'.$template_string); // compiles for later reuse
$smarty->display('eval:'.$template_string); // compiles every time
?>
```

From within a Smarty template

```
{include file="string:$template_string"} {* compiles for later reuse *}
{include file="eval:$template_string"} {* compiles every time *}
```

Template inheritance defined by PHP script

The `extends:` resource is used to define child/parent relationships for template inheritance from the PHP script. For details see section of Template Interitance.

Example 15.21. Using template inheritance from the PHP script

```
<?php
$smarty->display('extends:parent.tpl|child.tpl|grandchild.tpl');
?>
```

Note

Use this when inheritance is required programatically. When inheriting within PHP, it is not obvious from the child template what inheritance took place. If you have a choice, it is normally more flexible and intuitive to handle inheritance chains from within the templates.

Templates from other sources

You can retrieve templates using whatever possible source you can access with PHP: databases, sockets, files, etc. You do this by writing resource plugin functions and registering them with Smarty.

See resource plugins section for more information on the functions you are supposed to provide.

Note

Note that you cannot override the built-in `file:` resource, but you can provide a resource that fetches templates from the file system in some other way by registering under another resource name.

Example 15.22. Using custom resources

```
<?php
// put these function somewhere in your application
function db_get_template ($tpl_name, &$tpl_source, $smarty_obj)
{
    // do database call here to fetch your template,
    // populating $tpl_source with actual template contents
    $tpl_source = "This is the template text";
    // return true on success, false to generate failure notification
    return true;
}

function db_get_timestamp($tpl_name, &$tpl_timestamp, $smarty_obj)
{
    // do database call here to populate $tpl_timestamp
    // with unix epoch time value of last template modification.
    // This is used to determine if recompile is necessary.
    $tpl_timestamp = time(); // this example will always recompile!
    // return true on success, false to generate failure notification
    return true;
}

function db_get_secure($tpl_name, $smarty_obj)
{
    // assume all templates are secure
    return true;
}

function db_get_trusted($tpl_name, $smarty_obj)
{
    // not used for templates
}

// register the resource name "db"
$smarty->register_resource("db", array("db_get_template",
                                       "db_get_timestamp",
                                       "db_get_secure",
                                       "db_get_trusted"));

// using resource from php script
$smarty->display("db:index.tpl");
?>
```

And from within Smarty template:

```
{include file='db:/extras/navigation.tpl'}
```

Default template handler function

You can specify a function that is used to retrieve template contents in the event the template cannot be retrieved from its resource. One use of this is to create templates that do not exist on-the-fly.

Example 15.23. Using the default template handler function

```
<?php
// put this function somewhere in your application

function make_template ($resource_type, $resource_name, &$template_source, &$templ
$smarty_obj)
{
 if( $resource_type == 'file' ) {
  if ( ! is_readable ( $resource_name )) {
   // create the template file, return contents.
   $template_source = "This is a new template.";
            require_once SMARTY_CORE_DIR . 'core.write_file.php';
            smarty_core_write_file( array( 'filename'=>$smarty_obj->template_dir .
   return true;
  }
    } else {
  // not a file
  return false;
    }
}

// set the default handler
$smarty->default_template_handler_func = 'make_template';
?>
```

See also `Streams`

Chapter 16. Extending Smarty With Plugins

Version 2.0 introduced the plugin architecture that is used for almost all the customizable functionality of Smarty. This includes:

- functions
- modifiers
- block functions
- compiler functions
- prefilters
- postfilters
- outputfilters
- resources
- inserts

With the exception of resources, backwards compatibility with the old way of registering handler functions via register_* API is preserved. If you did not use the API but instead modified the class variables `$custom_funcs`, `$custom_mods`, and other ones directly, then you will need to adjust your scripts to either use the API or convert your custom functionality into plugins.

How Plugins Work

Plugins are always loaded on demand. Only the specific modifiers, functions, resources, etc invoked in the templates scripts will be loaded. Moreover, each plugin is loaded only once, even if you have several different instances of Smarty running within the same request.

Pre/postfilters and output filters are a bit of a special case. Since they are not mentioned in the templates, they must be registered or loaded explicitly via API functions before the template is processed. The order in which multiple filters of the same type are executed depends on the order in which they are registered or loaded.

The plugins directory can be a string containing a path or an array containing multiple paths. To install a plugin, simply place it in one of the directories and Smarty will use it automatically.

Naming Conventions

Plugin files and functions must follow a very specific naming convention in order to be located by Smarty.

plugin files must be named as follows:

```
type.name.php
```

- Where `type` is one of these plugin types:
  - function
  - modifier
  - block
  - compiler

  - prefilter
  - postfilter
  - outputfilter
  - resource
  - insert

- And `name` should be a valid identifier; letters, numbers, and underscores only, see php variables [http://php.net/language.variables].

- Some examples: `function.html_select_date.php`, `resource.db.php`, `modifier.spacify.php`.

plugin functions inside the PHP files must be named as follows:

```
smarty_type, _name
```

- The meanings of `type` and `name` are the same as above.

- An example modifier name `foo` would be `function smarty_modifier_foo()`.

Smarty will output appropriate error messages if the plugin file it needs is not found, or if the file or the plugin function are named improperly.

Writing Plugins

Plugins can be either loaded by Smarty automatically from the filesystem or they can be registered at runtime via one of the register_* API functions. They can also be unregistered by using unregister_* API functions.

For the plugins that are registered at runtime, the name of the plugin function(s) does not have to follow the naming convention.

If a plugin depends on some functionality provided by another plugin (as is the case with some plugins bundled with Smarty), then the proper way to load the needed plugin is this:

```
<?php
require_once $smarty->_get_plugin_filepath('function', 'html_options');
?>
```

As a general rule, Smarty object is always passed to the plugins as the last parameter with two exceptions:

- modifiers do not get passed the Smarty object at all

- blocks get passed *`$repeat`* after the Smarty object to keep backwards compatibility to older versions of Smarty.

Template Functions

```
void smarty_function_name($params, &$smarty);
```

```
array $params;
object &$smarty;
```

All attributes passed to template functions from the template are contained in the *$params* as an associative array.

The output (return value) of the function will be substituted in place of the function tag in the template, eg the `{fetch}` function. Alternatively, the function can simply perform some other task without any output, eg the `{assign}` function.

If the function needs to assign some variables to the template or use some other Smarty-provided functionality, it can use the supplied *$smarty* object to do so eg `$smarty->foo()`.

Example 16.1. function plugin with output

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     function.eightball.php
 * Type:     function
 * Name:     eightball
 * Purpose:  outputs a random magic answer
 * -------------------------------------------------------------
 */
function smarty_function_eightball($params, $smarty)
{
    $answers = array('Yes',
                     'No',
                     'No way',
                     'Outlook not so good',
                     'Ask again soon',
                     'Maybe in your reality');

    $result = array_rand($answers);
    return $answers[$result];
}
?>
```

which can be used in the template as:

```
Question: Will we ever have time travel?
Answer: {eightball}.
```

Example 16.2. function plugin without output

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     function.assign.php
 * Type:     function
 * Name:     assign
 * Purpose:  assign a value to a template variable
 * -------------------------------------------------------------
 */
function smarty_function_assign($params, $smarty)
{
    if (empty($params['var'])) {
        $smarty->triggerError("assign: missing 'var' parameter");
        return;
    }

    if (!in_array('value', array_keys($params))) {
        $smarty->triggerError("assign: missing 'value' parameter");
        return;
    }

    $smarty->assign($params['var'], $params['value']);
}
?>
```

See also: `registerPlugin()`, `unregisterPlugin()`.

Modifiers

Modifiers are little functions that are applied to a variable in the template before it is displayed or used in some other context. Modifiers can be chained together.

mixed **smarty_modifier_*name***(*$value*, *$param1*);

mixed *$value*;
[mixed *$param1*, ...];

The first parameter to the modifier plugin is the value on which the modifier is to operate. The rest of the parameters are optional, depending on what kind of operation is to be performed.

The modifier has to return [http://php.net/return] the result of its processing.

Example 16.3. A simple modifier plugin

This plugin basically aliases one of the built-in PHP functions. It does not have any additional parameters.

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     modifier.capitalize.php
 * Type:     modifier
 * Name:     capitalize
 * Purpose:  capitalize words in the string
 * -------------------------------------------------------------
 */
function smarty_modifier_capitalize($string)
{
    return ucwords($string);
}
?>
```

Example 16.4. More complex modifier plugin

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     modifier.truncate.php
 * Type:     modifier
 * Name:     truncate
 * Purpose:  Truncate a string to a certain length if necessary,
 *           optionally splitting in the middle of a word, and
 *           appending the $etc string.
 * -------------------------------------------------------------
 */
function smarty_modifier_truncate($string, $length = 80, $etc = '...',
                                  $break_words = false)
{
    if ($length == 0)
        return '';

    if (strlen($string) > $length) {
        $length -= strlen($etc);
        $fragment = substr($string, 0, $length+1);
        if ($break_words)
            $fragment = substr($fragment, 0, -1);
        else
            $fragment = preg_replace('/\s+(\S+)?$/', '', $fragment);
        return $fragment.$etc;
    } else
        return $string;
}
?>
```

See also `registerPlugin()`, `unregisterPlugin()`.

Block Functions

void **smarty_block_*name***(*$params*, *$content*, *&$smarty*, *&$repeat*);

array *$params*;
mixed *$content*;
object *&$smarty*;
boolean *&$repeat*;

Block functions are functions of the form: `{func} .. {/func}`. In other words, they enclose a template block and operate on the contents of this block. Block functions take precedence over custom functions of the same name, that is, you cannot have both custom function `{func}` and block function `{func}.. {/func}`.

- By default your function implementation is called twice by Smarty: once for the opening tag, and once for the closing tag. (See `$repeat` below on how to change this.)

- Only the opening tag of the block function may have attributes. All attributes passed to template functions from the template are contained in the *`$params`* variable as an associative array. The opening tag attributes are also accessible to your function when processing the closing tag.

- The value of the *`$content`* variable depends on whether your function is called for the opening or closing tag. In case of the opening tag, it will be `NULL`, and in case of the closing tag it will be the contents of the template block. Note that the template block will have already been processed by Smarty, so all you will receive is the template output, not the template source.

- The parameter *`$repeat`* is passed by reference to the function implementation and provides a possibility for it to control how many times the block is displayed. By default *`$repeat`* is `TRUE` at the first call of the block-function (the opening tag) and `FALSE` on all subsequent calls to the block function (the block's closing tag). Each time the function implementation returns with *`$repeat`* being `TRUE`, the contents between `{func}...{/func}` are evaluated and the function implementation is called again with the new block contents in the parameter *`$content`*.

If you have nested block functions, it's possible to find out what the parent block function is by accessing `$smarty->_tag_stack` variable. Just do a `var_dump()` [http://php.net/var_dump] on it and the structure should be apparent.

Example 16.5. block function

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     block.translate.php
 * Type:     block
 * Name:     translate
 * Purpose:  translate a block of text
 * -------------------------------------------------------------
 */
function smarty_block_translate($params, $content, $smarty, &$repeat)
{
    // only output on the closing tag
    if(!$repeat){
        if (isset($content)) {
            $lang = $params['lang'];
            // do some intelligent translation thing here with $content
            return $translation;
        }
    }
}
?>
```

See also: `registerPlugin()`, `unregisterPlugin()`.

Compiler Functions

Compiler functions are called only during compilation of the template. They are useful for injecting PHP code or time-sensitive static content into the template. If there is both a compiler function and a custom function registered under the same name, the compiler function has precedence.

mixed **smarty_compiler_*name***(*$tag_arg*, *&$smarty*);

string *$tag_arg*;
object *&$smarty*;

The compiler function is passed two parameters: the tag argument string - basically, everything from the function name until the ending delimiter, and the Smarty object. It's supposed to return the PHP code to be injected into the compiled template.

Example 16.6. A simple compiler function

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     compiler.tplheader.php
 * Type:     compiler
 * Name:     tplheader
 * Purpose:  Output header containing the source file name and
 *           the time it was compiled.
 * -------------------------------------------------------------
 */
function smarty_compiler_tplheader($tag_arg, $smarty)
{
    return "\necho '" . $smarty->_current_file . " compiled at " . date('Y-m-d H:M
}
?>
```

This function can be called from the template as:

```
{* this function gets executed at compile time only *}
{tplheader}
```

The resulting PHP code in the compiled template would be something like this:

```
<?php
echo 'index.tpl compiled at 2002-02-20 20:02';
?>
```

See also `registerPlugin()`, `unregisterPlugin()`.

Prefilters/Postfilters

Prefilter and postfilter plugins are very similar in concept; where they differ is in the execution -- more precisely the time of their execution.

```
string smarty_prefilter_name($source, &$smarty);

string $source;
object &$smarty;
```

Prefilters are used to process the source of the template immediately before compilation. The first parameter to the prefilter function is the template source, possibly modified by some other prefilters. The plugin is supposed to return the modified source. Note that this source is not saved anywhere, it is only used for compilation.

```
string smarty_postfilter_name($compiled, &$smarty);

string $compiled;
object &$smarty;
```

Postfilters are used to process the compiled output of the template (the PHP code) immediately after the compilation is done but before the compiled template is saved to the filesystem. The first parameter to the postfilter function is the compiled template code, possibly modified by other postfilters. The plugin is supposed to return the modified version of this code.

Example 16.7. prefilter plugin

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     prefilter.pre01.php
 * Type:     prefilter
 * Name:     pre01
 * Purpose:  Convert html tags to be lowercase.
 * -------------------------------------------------------------
 */
 function smarty_prefilter_pre01($source, $smarty)
 {
     return preg_replace('!<(\w+)[^>]+>!e', 'strtolower("$1")', $source);
 }
?>
```

Example 16.8. postfilter plugin

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     postfilter.post01.php
 * Type:     postfilter
 * Name:     post01
 * Purpose:  Output code that lists all current template vars.
 * -------------------------------------------------------------
 */
 function smarty_postfilter_post01($compiled, $smarty)
 {
     $compiled = "<pre>\n<?php print_r(\$this->getTemplateVars()); ?>\n</pre>" . $
     return $compiled;
 }
?>
```

See also `registerFilter()` and `unregisterFilter()`.

Output Filters

Output filter plugins operate on a template's output, after the template is loaded and executed, but before the output is displayed.

```
string smarty_outputfilter_name($template_output, &$smarty);

string $template_output;
object &$smarty;
```

The first parameter to the output filter function is the template output that needs to be processed, and the second parameter is the instance of Smarty invoking the plugin. The plugin is supposed to do the processing and return the results.

Example 16.9. An output filter plugin

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     outputfilter.protect_email.php
 * Type:     outputfilter
 * Name:     protect_email
 * Purpose:  Converts @ sign in email addresses to %40 as
 *           a simple protection against spambots
 * -------------------------------------------------------------
 */
 function smarty_outputfilter_protect_email($output, $smarty)
 {
     return preg_replace('!(\S+)@([a-zA-Z0-9\.\-]+\.([a-zA-Z]{2,3}|[0-9]{1,3}))!',
                         '$1%40$2', $output);
 }
?>
```

See also `registerFilter()`, `unregisterFilter()`.

Resources

Resource plugins are meant as a generic way of providing template sources or PHP script components to Smarty. Some examples of resources: databases, LDAP, shared memory, sockets, and so on.

There are a total of four functions that need to be registered for each type of resource. Every function will receive the requested resource as the first parameter and the Smarty object as the last parameter. The rest of parameters depend on the function.

```
bool smarty_resource_name_source($rsrc_name, &$source, &$smarty);

string $rsrc_name;
string &$source;
object &$smarty;

bool smarty_resource_name_timestamp($rsrc_name, &$timestamp, &$smarty);

string $rsrc_name;
int &$timestamp;
object &$smarty;

bool smarty_resource_name_secure($rsrc_name, &$smarty);

string $rsrc_name;
object &$smarty;

bool smarty_resource_name_trusted($rsrc_name, &$smarty);
```

```
string $rsrc_name;
object &$smarty;
```

- The first function, `source()` is supposed to retrieve the resource. Its second parameter *`$source`* is a variable passed by reference where the result should be stored. The function is supposed to return `TRUE` if it was able to successfully retrieve the resource and `FALSE` otherwise.

- The second function, `timestamp()` is supposed to retrieve the last modification time of the requested resource, as a UNIX timestamp. The second parameter *`$timestamp`* is a variable passed by reference where the timestamp should be stored. The function is supposed to return `TRUE` if the timestamp could be succesfully determined, or `FALSE` otherwise.

- The third function, `secure()`is supposed to return `TRUE` or `FALSE`, depending on whether the requested resource is secure or not. This function is used only for template resources but should still be defined.

- The fourth function, `trusted()` is supposed to return `TRUE` or `FALSE`, depending on whether the requested resource is trusted or not. This function is used for only for PHP script components requested by `{include_php}` tag or `{insert}` tag with the *`src`* attribute. However, it should still be defined even for template resources.

Example 16.10. resource plugin

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     resource.db.php
 * Type:     resource
 * Name:     db
 * Purpose:  Fetches templates from a database
 * -------------------------------------------------------------
 */
function smarty_resource_db_source($tpl_name, &$tpl_source, $smarty)
{
    // do database call here to fetch your template,
    // populating $tpl_source with actual template contents
    $tpl_source = "This is the template text";
    // return true on success, false to generate failure notification
    return true;
}

function smarty_resource_db_timestamp($tpl_name, &$tpl_timestamp, $smarty)
{
    // do database call here to populate $tpl_timestamp
    // with unix epoch time value of last template modification.
    // This is used to determine if recompile is necessary.
    $tpl_timestamp = time(); // this example will always recompile!
    // return true on success, false to generate failure notification
    return true;
}

function smarty_resource_db_secure($tpl_name, $smarty)
{
    // assume all templates are secure
    return true;
}

function smarty_resource_db_trusted($tpl_name, $smarty)
{
    // not used for templates
}
?>
```

See also `register_resource()`, `unregister_resource()`.

Inserts

Insert plugins are used to implement functions that are invoked by `{insert}` tags in the template.

`string` **`smarty_insert_`*`name`***`(`*`$params`*`, `*`&$smarty`*`);`

```
array $params;
object &$smarty;
```

The first parameter to the function is an associative array of attributes passed to the insert.

The insert function is supposed to return the result which will be substituted in place of the `{insert}` tag in the template.

Example 16.11. insert plugin

```
<?php
/*
 * Smarty plugin
 * -------------------------------------------------------------
 * File:     insert.time.php
 * Type:     time
 * Name:     time
 * Purpose:  Inserts current date/time according to format
 * -------------------------------------------------------------
 */
function smarty_insert_time($params, $smarty)
{
    if (empty($params['format'])) {
        $smarty->triggerError("insert time: missing 'format' parameter");
        return;
    }
    return strftime($params['format']);
}
?>
```

Part IV. Appendixes

Table of Contents

Chapter 17. Troubleshooting

Smarty/PHP errors

Smarty can catch many errors such as missing tag attributes or malformed variable names. If this happens, you will see an error similar to the following:

Example 17.1. Smarty errors

```
Warning: Smarty: [in index.tpl line 4]: syntax error: unknown tag - '%blah'
       in /path/to/smarty/Smarty.class.php on line 1041

Fatal error: Smarty: [in index.tpl line 28]: syntax error: missing section name
       in /path/to/smarty/Smarty.class.php on line 1041
```

Smarty shows you the template name, the line number and the error. After that, the error consists of the actual line number in the Smarty class that the error occured.

There are certain errors that Smarty cannot catch, such as missing close tags. These types of errors usually end up in PHP compile-time parsing errors.

Example 17.2. PHP parsing errors

```
Parse error: parse error in /path/to/smarty/templates_c/index.tpl.php on line 75
```

When you encounter a PHP parsing error, the error line number will correspond to the compiled PHP script, NOT the template itself. Usually you can look at the template and spot the syntax error. Here are some common things to look for: missing close tags for `{if}{/if}` or `{section}{/section}` , or syntax of logic within an `{if}` tag. If you can't find the error, you might have to open the compiled PHP file and go to the line number to figure out where the corresponding error is in the template.

Example 17.3. Other common errors

```
Warning: Smarty error: unable to read resource: "index.tpl" in...
or
Warning: Smarty error: unable to read resource: "site.conf" in...
```

- The `$template_dir` is incorrect, doesn't exist or the file `index.tpl` is not in the `templates/` directory
- A `{config_load}` function is within a template (or `config_load()` has been called) and either `$config_dir` is incorrect, does not exist or `site.conf` is not in the directory.

```
Fatal error: Smarty error: the $compile_dir 'templates_c' does not exist,
or is not a directory...
```

- Either the `$compile_dir`is incorrectly set, the directory does not exist, or `templates_c` is a file and not a directory.

```
Fatal error: Smarty error: unable to write to $compile_dir '....
```

- The `$compile_dir` is not writable by the web server. See the bottom of the installing smarty page for more about permissions.

```
Fatal error: Smarty error: the $cache_dir 'cache' does not exist,
or is not a directory. in /..
```

- This means that `$caching` is enabled and either; the `$cache_dir` is incorrectly set, the directory does not exist, or `cache/` is a file and not a directory.

```
Fatal error: Smarty error: unable to write to $cache_dir '/...
```

- This means that `$caching` is enabled and the `$cache_dir` is not writable by the web server. See the bottom of the installing smarty page for permissions.

See also debugging.

Chapter 18. Tips & Tricks

Blank Variable Handling

There may be times when you want to print a default value for an empty variable instead of printing nothing, such as printing `&nbsp;` so that html table backgrounds work properly. Many would use an `{if}` statement to handle this, but there is a shorthand way with Smarty, using the `default` variable modifier.

Note

“Undefined variable” errors will show if PHP xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx and a variable had not been assigned to Smarty.

Example 18.1. Printing &nbsp; when a variable is empty

```
{* the long way *}
{if $title eq ''}
   &nbsp;
{else}
   {$title}
{/if}

{* the short way *}
{$title|default:'&nbsp;'}
```

See also `default` modifier and default variable handling.

Default Variable Handling

If a variable is used frequently throughout your templates, applying the `default` modifier every time it is mentioned can get a bit ugly. You can remedy this by assigning the variable its default value with the `{assign}` function.

Example 18.2. Assigning a template variable its default value

```
{* do this somewhere at the top of your template *}
{assign var='title' value=$title|default:'no title'}

{* if $title was empty, it now contains the value "no title" when you use it *}
{$title}
```

See also `default` modifier and blank variable handling.

Passing variable title to header template

When the majority of your templates use the same headers and footers, it is common to split those out into their own templates and {include} them. But what if the header needs to have a different title, depending on what page you are coming from? You can pass the title to the header as an attribute when it is included.

Example 18.3. Passing the title variable to the header template

mainpage.tpl - When the main page is drawn, the title of “Main Page” is passed to the header.tpl, and will subsequently be used as the title.

```
{include file='header.tpl' title='Main Page'}
{* template body goes here *}
{include file='footer.tpl'}
```

archives.tpl - When the archives page is drawn, the title will be “Archives”. Notice in the archive example, we are using a variable from the archives_page.conf file instead of a hard coded variable.

```
{config_load file='archive_page.conf'}

{include file='header.tpl' title=#archivePageTitle#}
{* template body goes here *}
{include file='footer.tpl'}
```

header.tpl - Notice that “Smarty News” is printed if the $title variable is not set, using the default variable modifier.

```
<html>
<head>
<title>{$title|default:'Smarty News'}</title>
</head>
<body>
```

footer.tpl

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

Dates

As a rule of thumb, always pass dates to Smarty as timestamps [http://php.net/time]. This allows template designers to use the `date_format` modifier for full control over date formatting, and also makes it easy to compare dates if necessary.

Example 18.4. Using date_format

```
{$startDate|date_format}
```

This will output:

```
Jan 4, 2009
```

```
{$startDate|date_format:"%Y/%m/%d"}
```

This will output:

```
2009/01/04
```

Dates can be compared in the template by timestamps with:

```
{if $order_date < $invoice_date}
   ...do something..
{/if}
```

When using `{html_select_date}` in a template, the programmer will most likely want to convert the output from the form back into timestamp format. Here is a function to help you with that.

Example 18.5. Converting form date elements back to a timestamp

```
<?php

// this assumes your form elements are named
// startDate_Day, startDate_Month, startDate_Year

$startDate = makeTimeStamp($startDate_Year, $startDate_Month, $startDate_Day);

function makeTimeStamp($year='', $month='', $day='')
{
   if(empty($year)) {
       $year = strftime('%Y');
   }
   if(empty($month)) {
       $month = strftime('%m');
   }
   if(empty($day)) {
       $day = strftime('%d');
   }

   return mktime(0, 0, 0, $month, $day, $year);
}
?>
```

See also `{html_select_date}`, `{html_select_time}`, `date_format` and *`$smarty.now`*,

WAP/WML

WAP/WML templates require a php Content-Type header [http://php.net/header] to be passed along with the template. The easist way to do this would be to write a custom function that prints the header. If you are using caching, that won't work so we'll do it using the `{insert}` tag; remember `{insert}` tags are not cached! Be sure that there is nothing output to the browser before the template, or else the header may fail.

Example 18.6. Using {insert} to write a WML Content-Type header

```
<?php

// be sure apache is configure for the .wml extensions!
// put this function somewhere in your application, or in Smarty.addons.php
function insert_header($params)
{
   // this function expects $content argument
   if (empty($params['content'])) {
       return;
   }
   header($params['content']);
   return;
}

?>
```

your Smarty template *must* begin with the insert tag :

```
{insert name=header content="Content-Type: text/vnd.wap.wml"}

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wapforum.org/DTD/w

<!-- begin new wml deck -->
<wml>
 <!-- begin first card -->
 <card>
  <do type="accept">
   <go href="#two"/>
  </do>
  <p>
   Welcome to WAP with Smarty!
   Press OK to continue...
  </p>
 </card>
 <!-- begin second card -->
 <card id="two">
  <p>
   Pretty easy isn't it?
  </p>
 </card>
</wml>
```

Componentized Templates

Traditionally, programming templates into your applications goes as follows: First, you accumulate your variables within your PHP application, (maybe with database queries.) Then, you instantiate your Smarty object, `assign()` the variables and `display()` the template. So lets say for example we have a stock ticker on our template. We would collect the stock data in our application, then assign these variables in the template and display it. Now wouldn't it be nice if you could add this stock ticker to any application by merely including the template, and not worry about fetching the data up front?

You can do this by writing a custom plugin for fetching the content and assigning it to a template variable.

Example 18.7. componentized template

`function.load_ticker.php` - drop file in *`$plugins directory`*

```
<?php

// setup our function for fetching stock data
function fetch_ticker($symbol)
{
   // put logic here that fetches $ticker_info
   // from some ticker resource
   return $ticker_info;
}

function smarty_function_load_ticker($params, $smarty)
{
   // call the function
   $ticker_info = fetch_ticker($params['symbol']);

   // assign template variable
   $smarty->assign($params['assign'], $ticker_info);
}
?>
```

`index.tpl`

```
{load_ticker symbol='SMARTY' assign='ticker'}

Stock Name: {$ticker.name} Stock Price: {$ticker.price}
```

See also `{include_php}`, `{include}` and `{php}`.

Obfuscating E-mail Addresses

Do you ever wonder how your email address gets on so many spam mailing lists? One way spammers collect email addresses is from web pages. To help combat this problem, you can make your email address show up in scrambled javascript in the HTML source, yet it it will look and work correctly in the browser. This is done with the `{mailto}` plugin.

Example 18.8. Example of template the Obfuscating an email address

```
<div id="contact">Send inquiries to
{mailto address=$EmailAddress encode='javascript' subject='Hello'}
</div>
```

Technical Note

This method isn't 100% foolproof. A spammer could conceivably program his e-mail collector to decode these values, but not likely.... hopefully..yet ... wheres that quantum computer :-?.

See also `escape` modifier and `{mailto}`.

Chapter 19. Resources

Smarty's homepage is located at http://www.smarty.net/

- You can join the mailing list by sending an e-mail to `ismarty-discussion-subscribe@googlegroups.com`. An archive of the mailing list can be viewed at here [http://groups.google.com/group/smarty-discussion]
- Forums are at http://www.smarty.net/forums/
- The wiki is located at http://smarty.incutio.com/
- Join the chat at irc.freenode.net#smarty [http://smarty.incutio.com/]
- FAQ's are here [http://smarty.incutio.com/?page=SmartyFrequentlyAskedQuestions] and here [http://www.smarty.net/forums/viewforum.php?f=23]

Chapter 20. BUGS

Check the BUGS file that comes with the latest distribution of Smarty, or check the website.