

# Smarty Manual

**by**

Monte Ohrt <monte at ohrt dot com> and Andrei Zmievski <andrei@php.net>

---

# Smarty Manual

Published 14-12-2005  
Copyright © 2001-2005 New Digital Group, Inc.

---

---

# Table of Contents

Preface .....	vi
I. Getting Started .....	1
1. What is Smarty? .....	2
2. Installation .....	3
Requirements .....	3
Basic Installation .....	3
Extended Setup .....	6
II. Smarty For Template Designers .....	8
3. Basic Syntax .....	10
Comments .....	10
Variables .....	11
Functions .....	11
Attributes .....	12
Embedding Vars in Double Quotes .....	12
Math .....	12
Escaping Smarty Parsing .....	13
4. Variables .....	14
Variables assigned from PHP .....	14
Variables loaded from config files .....	16
{\$smarty} reserved variable .....	17
5. Variable Modifiers .....	20
capitalize .....	21
cat .....	21
count_characters .....	22
count_paragraphs .....	22
count_sentences .....	23
count_words .....	24
date_format .....	24
default .....	26
escape .....	27
indent .....	28
lower .....	29
nl2br .....	30
regex_replace .....	30
replace .....	31
spacify .....	32
string_format .....	32
strip .....	33
strip_tags .....	33
truncate .....	34
upper .....	35
wordwrap .....	35
6. Combining Modifiers .....	38
7. Built-in Functions .....	39
{capture} .....	39
{config_load} .....	40
{foreach},{foreachelse} .....	42
{if},{elseif},{else} .....	44
{include} .....	45
{include_php} .....	47
{insert} .....	48
{ldelim},{rdelim} .....	49
{literal} .....	50

{php} .....	50
{section},{sectionelse} .....	51
{strip} .....	60
8. Custom Functions .....	62
{assign} .....	62
{counter} .....	63
{cycle} .....	64
{debug} .....	65
{eval} .....	65
{fetch} .....	66
{html_checkboxes} .....	67
{html_image} .....	68
{html_options} .....	69
{html_radios} .....	71
{html_select_date} .....	73
{html_select_time} .....	76
{html_table} .....	79
{mailto} .....	81
{math} .....	82
{popup} .....	84
{popup_init} .....	87
{textformat} .....	87
9. Config Files .....	91
10. Debugging Console .....	92
III. Smarty For Programmers .....	93
11. Constants .....	95
SMARTY_DIR .....	95
SMARTY_CORE_DIR .....	95
12. Smarty Class Variables .....	96
\$template_dir .....	96
\$compile_dir .....	96
\$config_dir .....	97
\$plugins_dir .....	97
\$debugging .....	97
\$debug_tpl .....	98
\$debugging_ctrl .....	98
\$autoload_filters .....	98
\$compile_check .....	98
\$force_compile .....	98
\$caching .....	98
\$cache_dir .....	99
\$cache_lifetime .....	99
\$cache_handler_func .....	99
\$cache_modified_check .....	99
\$config_overwrite .....	99
\$config_booleanize .....	100
\$config_read_hidden .....	100
\$config_fix_newlines .....	100
\$default_template_handler_func .....	100
\$php_handling .....	100
\$security .....	101
\$secure_dir .....	101
\$security_settings .....	101
\$trusted_dir .....	101
\$left_delimiter .....	101
\$right_delimiter .....	102
\$compiler_class .....	102
\$request_vars_order .....	102

\$request_use_auto_globals .....	102
\$error_reporting .....	102
\$compile_id .....	102
\$use_sub_dirs .....	103
\$default_modifiers .....	103
\$default_resource_type .....	103
13. Smarty Class Methods() .....	104
14. Caching .....	145
Setting Up Caching .....	145
Multiple Caches Per Page .....	147
Cache Groups .....	148
Controlling Cacheability of Plugins' Output .....	149
15. Advanced Features .....	151
Objects .....	151
Prefilters .....	152
Postfilters .....	152
Output Filters .....	153
Cache Handler Function .....	153
Resources .....	155
16. Extending Smarty With Plugins .....	159
How Plugins Work .....	159
Naming Conventions .....	159
Writing Plugins .....	160
Template Functions .....	160
Modifiers .....	162
Block Functions .....	163
Compiler Functions .....	164
Prefilters/Postfilters .....	165
Output Filters .....	166
Resources .....	166
Inserts .....	168
IV. Appendixes .....	169
17. Troubleshooting .....	170
Smarty/PHP errors .....	170
18. Tips & Tricks .....	172
Blank Variable Handling .....	172
Default Variable Handling .....	172
Passing variable title to header template .....	173
Dates .....	173
WAP/WML .....	174
Componentized Templates .....	175
Obfuscating E-mail Addresses .....	176
19. Resources .....	177
20. BUGS .....	178

---

# Preface

It is undoubtedly one of the most asked questions on the PHP mailing lists: how do I make my PHP scripts independent of the layout? While PHP is billed as "HTML embedded scripting language", after writing a couple of projects that mixed PHP and HTML freely one comes up with the idea that separation of form and content is a Good Thing [TM]. In addition, in many companies the roles of layout designer and programmer are separate. Consequently, the search for a templating solution ensues.

In our company for example, the development of an application goes on as follows: After the requirements docs are done, the interface designer makes mockups of the interface and gives them to the programmer. The programmer implements business logic in PHP and uses interface mockups to create skeleton templates. The project is then handed off to the HTML designer/web page layout person who brings the templates up to their full glory. The project may go back and forth between programming/HTML a couple of times. Thus, it's important to have good template support because programmers don't want anything to do with HTML and don't want HTML designers mucking around with PHP code. Designers need support for config files, dynamic blocks and other interface issues, but they don't want to have to deal with intricacies of the PHP programming language.

Looking at many templating solutions available for PHP today, most of them provide a rudimentary way of substituting variables into templates and do a limited form of dynamic block functionality. But our needs required a bit more than that. We didn't want programmers to be dealing with HTML layout at ALL, but this was almost inevitable. For instance, if a designer wanted background colors to alternate on dynamic blocks, this had to be worked out with the programmer in advance. We also needed designers to be able to use their own configuration files, and pull variables from them into the templates. The list goes on.

We started out writing out a spec for a template engine back in late 1999. After finishing the spec, we began to work on a template engine written in C that would hopefully be accepted for inclusion with PHP. Not only did we run into many complicated technical barriers, but there was also much heated debate about exactly what a template engine should and should not do. From this experience, we decided that the template engine should be written in PHP as a class, for anyone to use as they see fit. So we wrote an engine that did just that and SmartTemplate™ came into existence (note: this class was never submitted to the public). It was a class that did almost everything we wanted: regular variable substitution, supported including other templates, integration with config files, embedding PHP code, limited 'if' statement functionality and much more robust dynamic blocks which could be multiply nested. It did all this with regular expressions and the code turned out to be rather, shall we say, impenetrable. It was also noticeably slow in large applications from all the parsing and regular expression work it had to do on each invocation. The biggest problem from a programmer's point of view was all the necessary work in the PHP script to setup and process templates and dynamic blocks. How do we make this easier?

Then came the vision of what ultimately became Smarty. We know how fast PHP code is without the overhead of template parsing. We also know how meticulous and overbearing the PHP language may look to the average designer, and this could be masked with a much simpler templating syntax. So what if we combined the two strengths? Thus, Smarty was born... :-)

---

# Part I. Getting Started

## Table of Contents

1. What is Smarty? .....	2
2. Installation .....	3
Requirements .....	3
Basic Installation .....	3
Extended Setup .....	6

---

# 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 elements (HTML tables, background colors, font sizes, style sheets, etc.) One day the programmer needs to change the way the article content is retrieved (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 requires no changes 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, altering table row colors, upper-casing a variable, looping over an array of data and displaying it, etc. are all examples of presentation logic. This does not mean 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.

One of the unique aspects about Smarty is the template compiling. This means Smarty reads the template files and creates PHP scripts from them. Once they are created, they are executed from then on. Therefore there is no costly template file parsing for each request, and each template can take full advantage of PHP compiler cache solutions such as Zend Accelerator (<http://www.zend.com/>) or PHP Accelerator (<http://www.php-accelerator.co.uk>).

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 make custom functions and custom variable modifiers, so the template language is extremely extensible.
- Configurable template delimiter tag syntax, so you can use {}, {{}}, <!--{}-->, etc.
- The if/elseif/else/endif constructs are passed to the PHP parser, so the {if ...} expression syntax can be as simple or as complex as you like.
- Unlimited nesting of sections, ifs, etc. allowed.
- It is possible to embed PHP code right in your template files, although this may not be needed (nor recommended) since the engine is so customizable.
- Built-in caching support
- Arbitrary template sources
- Custom cache handling functions
- Plugin architecture



---

# Chapter 2. Installation

## Table of Contents

Requirements .....	3
Basic Installation .....	3
Extended Setup .....	6

## Requirements

Smarty requires a web server running PHP 4.0.6 or later.

## 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 they only get updated when you upgrade to a new version of Smarty.

### Example 2.1. Required Smarty library files

```
Smarty.class.php
Smarty_Compiler.class.php
Config_File.class.php
debug.tpl
/internals/*.php (all of them)
/plugins/*.php (all of them to be safe, maybe your site only needs a subset)
```

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`, Smarty will figure it out on its own. Therefore, if `Smarty.class.php` is not in your `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:

### Example 2.2. Create Smarty instance of Smarty

```
<?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 have to do one of the following:

**Example 2.3. Set SMARTY\_DIR constant manually**

```
<?php
// *nix style (note capital 'S')
define('SMARTY_DIR', '/usr/local/lib/php/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 assumed 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 2.4. Supply absolute path to library file**

```
<?php
// *nix style (note capital 'S')
require_once('/usr/local/lib/php/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 2.5. Add library directory to PHP include\_path**

```
<?php
// Edit your php.ini file, add the Smarty library
// directory to the include_path and restart web server.
// then the following should work:
require_once('Smarty.class.php');
$smarty = new Smarty();
?>
```

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.

Be sure you know the location of your web server document root. In our example, the document root is /web/www.example.com/docs/. 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 to place these directories *outside* of the document root.

For our installation example, we will be setting up the Smarty environment for a guest book application. We picked an ap-

plication 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 app. We'll place our Smarty directories under /web/www.example.com/smarty/guestbook/.

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

**Technical Note:** It is convenient to setup the web server so that 'index.php' can be identified as the default directory index, so if you access http://www.example.com/guestbook/, the 'index.php' script will be executed without adding 'index.php' to the URL. In Apache you can set this up by adding "index.php" onto the end of your *DirectoryIndex* setting (separate each entry with a space.) as in the httpd.conf example

```
DirectoryIndex index.htm index.html index.php index.php3 default.html index.cgi
```

Lets take a look at the file structure so far:

### Example 2.6. Example file structure

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

/web/www.example.com/smarty/guestbook/templates/
/web/www.example.com/smarty/guestbook/templates_c/
/web/www.example.com/smarty/guestbook/configs/
/web/www.example.com/smarty/guestbook/cache/

/web/www.example.com/docs/guestbook/index.php
```

Smarty will need **write access** (windows users please ignore) to the *\$compile\_dir* and *\$cache\_dir*, so be sure the web server user can write to them. 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 (usually in "/usr/local/apache/conf/") to see what user and group are being used.

### Example 2.7. Setting file permissions

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

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

**Technical 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 load. This will be located in the *\$template\_dir*.

### Example 2.8. Editing /web/www.example.com/smarty/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
// load Smarty library
require_once(SMARTY_DIR . 'Smarty.class.php');

$smarty = new Smarty();

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

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

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

**Technical Note:** In our example, we are setting absolute paths to all of the Smarty directories. If `/web/www.example.com/smarty/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 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. Editing `/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.

        $this->Smarty();

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

        $this->caching = true;
        $this->assign('app_name', 'Guest Book');
    }
}
?>
```

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

### **Example 2.11. Editing /web/www.example.com/docs/guestbook/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

3. Basic Syntax .....	10
Comments .....	10
Variables .....	11
Functions .....	11
Attributes .....	12
Embedding Vars in Double Quotes .....	12
Math .....	12
Escaping Smarty Parsing .....	13
4. Variables .....	14
Variables assigned from PHP .....	14
Variables loaded from config files .....	16
{\$smarty} reserved variable .....	17
5. Variable Modifiers .....	20
capitalize .....	21
cat .....	21
count_characters .....	22
count_paragraphs .....	22
count_sentences .....	23
count_words .....	24
date_format .....	24
default .....	26
escape .....	27
indent .....	28
lower .....	29
nl2br .....	30
regex_replace .....	30
replace .....	31
spacify .....	32
string_format .....	32
strip .....	33
strip_tags .....	33
truncate .....	34
upper .....	35
wordwrap .....	35
6. Combining Modifiers .....	38
7. Built-in Functions .....	39
{capture} .....	39
{config_load} .....	40
{foreach},{foreachelse} .....	42
{if},{elseif},{else} .....	44
{include} .....	45
{include_php} .....	47
{insert} .....	48
{l delim},{rdelim} .....	49
{literal} .....	50
{php} .....	50
{section},{sectionelse} .....	51
{strip} .....	60

8. Custom Functions .....	62
{assign} .....	62
{counter} .....	63
{cycle} .....	64
{debug} .....	65
{eval} .....	65
{fetch} .....	66
{html_checkboxes} .....	67
{html_image} .....	68
{html_options} .....	69
{html_radios} .....	71
{html_select_date} .....	73
{html_select_time} .....	76
{html_table} .....	79
{mailto} .....	81
{math} .....	82
{popup} .....	84
{popup_init} .....	87
{textformat} .....	87
9. Config Files .....	91
10. Debugging Console .....	92

---

# Chapter 3. Basic Syntax

## Table of Contents

Comments .....	10
Variables .....	11
Functions .....	11
Attributes .....	12
Embedding Vars in Double Quotes .....	12
Math .....	12
Escaping Smarty Parsing .....	13

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

For these examples, 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 --> They are useful for making internal notes in the templates.

### Example 3.1. Comments

```
<body>
{* a single line comment *}

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

{* include the header file here *}
{include file="header.tpl"}

{* Dev note: $includeFile is assigned in foo.php script *}
<!-- this html comment is sent to browser -->
{include file=$includeFile}

{include file=#includeFile#}

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



## Variables

Template variables start with a \$dollar sign. They can contain numbers, letters and underscores, much like a PHP variable [http://php.net/language.variables]. You can reference arrays that are indexed numerically or non-numerically. You can also reference object properties and methods. Config file variables are an exception to the dollar sign syntax. They can be referenced with surrounding #hashmarks#, or with the special \$smarty.config variable.

### Example 3.2. Variables

```
{ $foo }          <-- displaying a simple variable (non array/object)
{ $foo[4] }       <-- display the 5th element of a zero-indexed array
{ $foo.bar }      <-- display the "bar" key value of an array, similar to PHP $foo['bar']
{ $foo.$bar }     <-- display variable key value of an array, similar to PHP $foo[$bar]
{ $foo->bar }      <-- display the object property "bar"
{ $foo->bar() }    <-- display the return value of object method "bar"
{ #foo# }         <-- display the config file variable "foo"
{ $smarty.config.foo } <-- synonym for {#foo#}
{ $foo[bar] }     <-- syntax only valid in a section loop, see {section}
{ assign var=foo value="baa" } { $foo } <-- displays "baa", see {assign}
```

Many other combinations are allowed

```
{ $foo.bar.baz }
{ $foo.$bar.$baz }
{ $foo[4].baz }
{ $foo[4].$baz }
{ $foo.bar.baz[4] }
{ $foo->bar($baz,2,$bar) } <-- passing parameters
{ "foo" }                 <-- static values are allowed
```

See also \$smarty reserved variables and Config Variables.

## Functions

Each Smarty tag either prints a variable or invokes some sort of function. Functions are processed and displayed by enclosing the function and its attributes into delimiters like so: {funcname attr1="val" attr2="val"}.

### Example 3.3. function syntax

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

{include file="header.tpl"}

{if $highlight_name}
    Welcome, <font color="{#fontColor#}">{$name}</font>
{else}
    Welcome, {$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}. They cannot be modified.

Custom functions are **additional** functions implemented via plugins. They can be modified to your liking, or you can add new ones. {html\_options} and {popup} are examples of custom functions.

## 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 recommended for literal strings. Variables may also be used, and should not be in quotes.

Some attributes require boolean values (true or false). These can be specified as either unquoted `true`, `on`, and `yes`, or `false`, `off`, and `no`.

### Example 3.4. function attribute syntax

```
{include file='header.tpl'}
{include file='header.tpl' attrib_name='attrib value'}
{include file=$includeFile}
{include file=#includeFile# title='Smarty is cool'}
{html_select_date display_days=yes}
{mailto address='smarty@example.com'}
<select name='company_id'>
  {html_options options=$companies selected=$company_id}
</select>
```

## Embedding Vars in Double Quotes

Smarty will recognize assigned variables embedded in double quotes so long as the variables contain only numbers, letters, underscores and brackets []. With any other characters (period, object reference, etc.) the variable must be surrounded by backticks. You cannot embed modifiers, they must always be applied outside of quotes.

### Example 3.5. embedded quotes syntax

```
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!

PRACTICAL EXAMPLES:
{include file="subdir/$tpl_name.tpl"} <-- will replace $tpl_name with value
{cycle values="one,two,`$smarty.config.myval`"} <-- must have backticks
```

See also `escape`.

## Math

Math can be applied directly to variable values.

**Example 3.6. 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`" }
```

See also the `{math}` function for complex equations.

## 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.

The simplest thing is to avoid the situation altogether by separating your Javascript and CSS code into their own files and then using standard HTML methods to access them.

Including literal content is possible using `{literal} .. {/literal}` blocks. Similar to HTML entity usage, you can use `{ldelim}`, `{rdelim}` or `{Smarty.ldelim}` to display the current delimiters.

It is often convenient to simply change Smarty's `$left_delimiter` and `$right_delimiter`.

**Example 3.7. changing delimiters example**

```
<?php
$smarty = new Smarty;
$smarty->left_delimiter = '<!--{';
$smarty->right_delimiter = '}->';
$smarty->assign('foo', 'bar');
$smarty->display('example.tpl');
?>
```

Where `example.tpl` is:

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

See also `escape modifier`

---

# Chapter 4. Variables

## Table of Contents

Variables assigned from PHP .....	14
Variables loaded from config files .....	16
{Smarty} reserved variable .....	17

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 function 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. Examples:

```
{ $Name }
{ $Contacts[row].Phone }
<body bgcolor="{ #bgcolor# }">
```

## Variables assigned from PHP

Variables that are assigned from PHP are referenced by preceding them with a dollar sign \$. Variables assigned from within a template with the {assign} function are also displayed this way.

### Example 4.1. assigned variables

php script

```
<?php
$smarty = new Smarty;

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

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

where the content of index.tpl is:

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

This will 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 that are assigned from PHP by specifying the key after the '.' (period) symbol.

### Example 4.2. 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');  
?>
```

where the content of index.tpl is:

```
{ $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 4.3. 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');  
?>
```

where index.tpl is:

```
{ $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 4.4. accessing object properties

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

this will output:

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

## 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 second syntax is useful for embedding into quoted attribute values.

### Example 4.5. config variables

foo.conf:

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

index.tpl:

```
{ 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>

```

index.tpl: (alternate syntax)

```

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

```

this will output for both examples:

```

<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 special template 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 4.6. displaying request variables

```

{* display value of page from URL ($_GET) http://www.example.com/index.php?page=foo *}
{$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 `{$_SCRIPT_NAME}` can be accessed directly, although `{$smarty.server.SCRIPT_NAME}` is the proposed way to access this value.

## **{\$smarty.now}**

The current timestamp [<http://php.net/function.time>] can be accessed with `{$smarty.now}`. The number reflects the number of seconds passed since the so-called Epoch (January 1, 1970) and can be passed directly to the `date_format` modifier for display purposes.

### **Example 4.7. using {\$smarty.now}**

```

{* 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

### **Example 4.8. using {\$smarty.const}**

```

{$smarty.const._MY_CONST_VAL}

```

## **{\$smarty.capture}**

The output captured via `{capture}..{/capture}` construct can be accessed using `{$smarty}` variable. See section on `{capture}` for an example.

## **{\$smarty.config}**

`{$smarty}` variable can be used to refer to loaded config variables. `{$smarty.config.foo}` is a synonym for `{#foo#}`. See the section on `{config_load}` for an example.

## **{\$smarty.section}, {\$smarty.foreach}**



{Smarty} variable can be used to refer to {section} and {foreach} loop properties.

## **{Smarty.template}**

This variable contains the name of the current template being processed.

## **{Smarty.version}**

This variable contains the version of Smarty the template was compiled with.

## **{Smarty.l delimiter}, {Smarty.r delimiter}**

This variable is used for printing the left-delimiter and right-delimiter value literally. See {l delimiter},{r delimiter}.

See also Variables and Config Variables

---

# Chapter 5. Variable Modifiers

## Table of Contents

capitalize .....	21
cat .....	21
count_characters .....	22
count_paragraphs .....	22
count_sentences .....	23
count_words .....	24
date_format .....	24
default .....	26
escape .....	27
indent .....	28
lower .....	29
nl2br .....	30
regex_replace .....	30
replace .....	31
spacify .....	32
string_format .....	32
strip .....	33
strip_tags .....	33
truncate .....	34
upper .....	35
wordwrap .....	35

Variable modifiers can be applied to variables, custom functions or strings. To apply a modifier, specify the value followed by the | (pipe) and the modifier name. A modifier may accept additional parameters that affect its behavior. These parameters follow the modifier name and are separated by : (colon).

### Example 5.1. modifier example

```
{* 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="me@domain.dom" }
```

If you apply a modifier to an array variable instead of a single value variable, the modifier will be applied to every value in that array. If you really want the modifier to work on an entire array as a value, you must prepend the modifier name with an

@ symbol like so: `{ $articleTitle|@count }` (this will print out the number of elements in the `$articleTitle` array.)

Modifiers can be autoloaded from your `$plugins_dir` (also see: Naming Conventions) or can be registered explicitly (see: `register_modifier`).

Additionally all php-functions can be used as modifiers implicitly. (The `@count` example above actually uses php's `count()` function and not a smarty-modifier). Using php-functions as modifiers has two little pitfalls: First: Sometimes the order of the function-parameters is not the desirable one (`{ "%2.f" |sprintf:$float }` actually works, but asks for the more intuitive. For example: `{ $float|string_format:"%2.f" }` that is provided by the Smarty distribution). Second: with `$security` turned on all php-functions that are to be used as modifiers have to be declared trusted in the `$security_settings['MODIFIER_FUNCS']`-array.

See also `register_modifier()`, `register_function()`, Extending Smarty with plugins and modifiers,

## capitalize

This is used to capitalize the first letter of all words in a variable.

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 template is:

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

This 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	cat	Description
1	string	No	<i>empty</i>	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." }
```

This 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 }
```

This 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\n
    Man is Fatally Slain. Death Causes Loneliness, Feeling of Isolation."
);
?>
```

Where template is:

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

This 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 }
```

This 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 `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 were 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.

### Example 5.8. date\_format

```
<?php
```

```
$smarty->assign('yesterday', strtotime('-1 day'));  
?>
```

Where template is (uses \$smarty.now):

```
{ $smarty.now | date_format }  
{ $smarty.now | date_format: "%D" }  
{ $smarty.now | date_format: "%I:%M %p" }  
{ $yesterday | date_format }  
{ $yesterday | date_format: "%A, %B %e, %Y" }  
{ $yesterday | date_format: "%H:%M:%S" }
```

This will output:

```
Feb 6, 2001  
02/06/01  
02:33 pm  
Feb 5, 2001  
Monday, February 5, 2001  
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 00 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 preceded 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

**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.

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

## default

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

Parameter Position	Type	Required	Default	Description
1	string	No	<i>empty</i>	This is the default value to output if the



Parameter Position	Type	Required	Default	Description
				variable is empty.

### Example 5.9. default

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

Where template is:

```
{ $articleTitle|default:"no title" }
{ $myTitle|default:"no title" }
```

This will output:

```
Dealers Will Hear Car Talk at Noon.
no title
```

See also [Default Variable Handling](#) and [Blank Variable Handling](#).

## escape

This is used to html escape, url escape, escape single quotes on a variable not already escaped, hex escape, hexentity or javascript escape. By default, the variable is html escaped.

Parameter Position	Type	Required	Possible Values	Default	Description
1	string	No	ur u r l p a t h i n f	html	This is the escape format to use.

Parameter Position	Type	Required	Possible Values	Default	Description
			,quotes,hex,hex-entity,javascript		
2	string	No	ISO-8859-1, UTF-8, ... any character set supported by htmlentities() [http://php.net/sprintf]	ISO-8859-1	The character set encoding passed to htmlentities() et. al.

**Example 5.10. escape**

```
<?php
$smarty->assign('articleTitle',
    "Stiff Opposition Expected to Casketless Funeral Plan"
);
?>
```

Where template is:

```
{ $articleTitle }
{ $articleTitle | escape }
{ $articleTitle | escape:"html" } { * escapes & " ' < > * }
{ $articleTitle | escape:"htmlall" } { * escapes ALL html entities * }
{ $articleTitle | escape:"url" }
{ $articleTitle | escape:"quotes" }
<a href="mailto:{$EmailAddress|escape:"hex"}">{$EmailAddress|escape:"hexentity"}</a>
```

This will output:

```
'Stiff Opposition Expected to Casketless Funeral Plan'
&#039;Stiff Opposition Expected to Casketless Funeral Plan&#039;
&#039;Stiff Opposition Expected to Casketless Funeral Plan&#039;
&#039;Stiff Opposition Expected to Casketless Funeral Plan&#039;
%27Stiff+Opposition+Expected+to+Casketless+Funeral+Plan%27
\'Stiff Opposition Expected to Casketless Funeral Plan\'
<a href="mailto:%62%6f%..snip..%65%74">&#x62;&#x6f;&#x62..snip..&#x65;&#x74;</a>
```

See also Escaping Smarty Parsing and Obfuscating E-mail Addresses.

## indent

This indents a string at 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. (Use "\t" for tabs.)

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 5.11. 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" }
```

this 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 and spacyfy.

## lower

This is used to lowercase a variable.

**Example 5.12. lower**

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

Where template is:

```
{ $articleTitle }
```

---

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

This will output:

```
Two Convicts Evade Moose, Jury Hung.
two convicts evade moose, jury hung.
```

See also upper and Capitalize.

## nl2br

All linebreaks will be converted to `<br />` tags in the given variable. This is equivalent to the PHP `nl2br()` [<http://php.net/nl2br>] function.

### Example 5.13. nl2br

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

Where template is:

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

This should 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 syntax for `preg_replace()` [[http://php.net/preg\\_replace](http://php.net/preg_replace)] from the PHP manual.

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

### Example 5.14. regex\_replace

```
<?php
```

```
$smarty->assign('articleTitle', "Infertility unlikely to\nbe passed on, experts say.");
?>
```

Where index.tpl is:

```
{* replace each carriage return, tab and new line with a space *}
{$articleTitle}
{$articleTitle|regex_replace:"/[\r\t\n]\/":" "}
```

This should 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 `str_replace()` [[http://php.net/str\\_replace](http://php.net/str_replace)] function.

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

### Example 5.15. replace

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

Where template is:

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

This should 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	<i>one space</i>	This what gets inserted between each character of the variable.

### Example 5.16. spacify

```
<?php
$smarty->assign('articleTitle', 'Something Went Wrong in Jet Crash, Experts Say. ');
?>
```

Where template is:

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

This should output:

```
Something Went Wrong in Jet Crash, Experts Say.
S o m e t h i n g   W . . . . s n i p . . . . s h ,   E x p e r t s   S a y .
S ^ ^ o ^ ^ m ^ ^ e ^ ^ t ^ ^ h ^ ^ i ^ ^ n ^ ^ g ^ ^ . . . . s n i p . . . . ^ ^ 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	<i>n/a</i>	This is what format to use. (sprintf)

### Example 5.17. string\_format

```
<?php
$smarty->assign('number', 23.5787446);
?>
```

Where template is:

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

This should 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 a supplied string.

**Note:** If you want to strip blocks of template text, use the `{strip}` function.

### Example 5.18. strip

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

where template is:

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

This 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.
```

## strip\_tags

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

Parameter Position	Type	Required	Default	Description
1	bool	No	true	This determines wether the tags are replaced by '' or by "

### Example 5.19. 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}
```

This will output:

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

## truncate

This truncates a variable to a character length, 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 NOT included in the truncation length setting.
3	boolean	No	false	This determines whether or not to truncate at a word boundary (false), or at the exact character (true).
4	boolean	No	false	This determines whether the truncation happens at the end of the string (false), or in the middle of the string (true). Note that if this setting is true, then word boundaries are ignored.

### Example 5.20. truncate



```
<?php
$smarty->assign('articleTitle', 'Two Sisters Reunite after Eighteen Years at Checkout Counter. ');
?>
```

where template is:

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

## upper

This is used to uppercase a variable.

### Example 5.21. upper

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

where template is:

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

This 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

**wordwrap** wraps a string to a column width, default is 80. As an optional second parameter, you can specify a string of text to wrap the text to the next line (default is carriage return \n). By default, {wordwrap} will attempt to wrap at a word bound-

ary. If you want to cut off at the exact character length, pass the optional third parameter of 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 5.22. 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:30:"\n":true }
```

This 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 kidney
from dad she hasn't seen 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 | spacyfy }
{ $articleTitle | lower | spacyfy | truncate }
{ $articleTitle | lower | truncate:30 | spacyfy }
{ $articleTitle | lower | spacyfy | truncate:30: ". . ." }
```

The above example will output:

```
Smokers are Productive, but Death Cuts Efficiency.
S M O K E R S   A R . . . . s n i p . . . .   H   C U T S   E F F I C I E N C Y .
s m o k e r s   a r . . . . s n i p . . . .   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

## Table of Contents

{capture} .....	39
{config_load} .....	40
{foreach},{foreachelse} .....	42
{if},{elseif},{else} .....	44
{include} .....	45
{include_php} .....	47
{insert} .....	48
{ldelim},{rdelim} .....	49
{literal} .....	50
{php} .....	50
{section},{sectionelse} .....	51
{strip} .....	60

Smarty comes with several built-in functions. Built-in functions are integral to the template language. You cannot create custom functions with the same names, nor can you modify built-in functions.

## {capture}

{capture} is used to collect the output of the template 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 special variable `$smarty.capture.foo` where "foo" is the value passed in the name attribute. If you do not supply a name attribute, then "default" will be used as the name. All {capture} commands must be paired with {/capture}. You can nest capture commands.

Attribute Name	Type	Required	Default	Description
name	string	no	<i>default</i>	The name of the captured block
assign	string	No	<i>n/a</i>	The variable name where to assign the captured output to

### 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 7.1. capturing template content

```
{* we don't want to print a table row unless content is displayed *}
{capture name=banner}
  {include file='get_banner.tpl'}
{/capture}

{if $smarty.capture.banner ne ''}
```

```

<table>
<tr>
  <td>
    {$smarty.capture.banner}
  </td>
</tr>
</table>
{/if}

```

### Example 7.2. capturing content to a variable

This example also demonstrates the {popup} function

```

{capture name=some_content assign=popText}
  .... some content ....
{/capture}

<a href="#" {popup caption='Help' text=$popText}>help</a>

```

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.

Attribute Name	Type	Required	Default	Description
file	string	Yes	<i>n/a</i>	The name of the config file to include
section	string	No	<i>n/a</i>	The name of the section to load
scope	string	no	<i>local</i>	How the scope of the loaded variables are treated, which must be one of local, parent or global. local 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.
global	boolean	No	<i>No</i>	Whether or not variables are visible to the parent template, same as scope=parent. NOTE: This attribute is deprecated by the scope attribute, but still supported. If scope is

Attribute Name	Type	Required	Default	Description
				supplied, this value is ignored.

### Example 7.3. {config\_load}

example.conf

```
#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"}

<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:** *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 7.4. function {config\_load} with section

```
{config_load file='example.conf' section='Customer'}
```

```
<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` for arrays of config variables

See also Config files, Config variables, `$config_dir`, `get_config_vars()` and `config_load()`.

## {foreach},{foreachelse}

`{foreach}` loops are an alternative to `{section}` loops. `{foreach}` is used to loop over a **single associative array**. The syntax for `{foreach}` is much easier than `{section}`, but as a tradeoff it **can only be used for a single array**. `{foreach}` tags must be paired with `{/foreach}` tags. Required parameters are `from` and `item`. The name of the `{foreach}` loop can be anything you like, made up of letters, numbers and underscores. `{foreach}` loops can be nested, and the nested `{foreach}` names must be unique from each other. The `from` variable (usually an array of values) determines the number of times `{foreach}` will loop. `{foreachelse}` is executed when there are no values in the `from` variable.

Attribute Name	Type	Required	Default	Description
from	array	Yes	<i>n/a</i>	The array you are looping through
item	string	Yes	<i>n/a</i>	The name of the variable that is the current element
key	string	No	<i>n/a</i>	The name of the variable that is the current key
name	string	No	<i>n/a</i>	The name of the foreach loop for accessing foreach properties

### Example 7.5. {foreach} - item

```
<?php
$arr = array( 1001,1002,1003);
$smarty->assign('custid', $arr);
?>
```

```
{* this example will print out all the values of the $custid array *}
{foreach from=$custid item=curr_id}
  id: {$curr_id}<br />
{/foreach}
```

The above example will output:

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

### Example 7.6. {foreach} - item and key

```
// The key contains the key for each looped value
// assignment looks like this:
<?php
$smarty->assign('contacts', array(
```



```

        array('phone' => '1',
              'fax' => '2',
              'cell' => '3'),
        array('phone' => '555-4444',
              'fax' => '555-3333',
              'cell' => '760-1234')
    ));

```

?>

```

{foreach name=outer item=contact from=$contacts}
  <hr />
  {foreach key=key item=item from=$contact}
    {$key}: {$item}<br />
  {/foreach}
{/foreach}

```

The above example will output:

```

<hr />
phone: 1<br />
fax: 2<br />
cell: 3<br />
<hr />
phone: 555-4444<br />
fax: 555-3333<br />
cell: 760-1234<br />

```

### Example 7.7. {foreach} - database example (eg PEAR or ADODB)

```

<?php
    $sql = 'select contact_id, name, nick from contacts order by contact';
    $smarty->assign("contacts", $db->getAssoc($sql));
?>

```

```

{foreach key=cid item=con from=$contacts}
  <a href="contact.php?contact_id={$cid}">{$con.name} - {$con.nick}</a><br />
{/foreach}

```

{foreach} loops also have their own variables that handle {foreach} properties. These are indicated like so: {Smarty.foreach.foreachname.varname} with foreachname being the name specified as the *name* attribute of foreach

See {section} for examples of the properties below as they are identical

## iteration

iteration is used to display the current loop iteration. Iteration always starts with 1 and is incremented by one on each iteration.

## first

*first* is set to true if the current foreach iteration is the first one.

## last

*last* is set to true if the current foreach iteration is the last one.

## show

*show* is used as a parameter to `foreach`. *show* is a boolean value, true or false. If false, the `foreach` will not be displayed. If there is a `foreachelse` present, that will be alternately displayed.

## total

*total* is used to display the number of iterations that this `foreach` will loop. This can be used inside or after the `foreach`.

See also `{section}` and `$smarty.foreach`.

## {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 an *{/if}*. *{else}* and *{elseif}* are also permitted. All PHP conditionals and functions are recognized, such as `//`, `or`, `&&`, `and`, `is_array()`, etc.

If `$security` is enabled then `IF_FUNCS` array in the `$security_settings` array.

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

### Example 7.8. {if} statements

```
{if $name eq "Fred"}
    Welcome Sir.
{elseif $name eq "Wilma"}
    Welcome Ma'am.
{else}
    Welcome, whatever you are.
{/if}

{* an example with "or" logic *}
{if $name eq "Fred" or $name eq "Wilma"}
```

```

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

{* 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}

```

## {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 attribute "file", which contains the template resource path.

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

All assigned variables' values are restored after the scope of the included template is left. This means you can use all variables from the including template inside the included template. But changes to variables inside the included template are not visible inside the including template after the `{include}` statement.

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

### Example 7.9. function `{include}`

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

{* body of template goes here *}
{include file="$tpl_name.tpl"} <-- will replace $tpl_name with value

{include file='page_footer.tpl'}
</body>
</html>
```

You can also pass variables to included templates as attributes. Any variables explicitly passed to an included template as attributes are only available within the scope of the included file. Attribute variables override current template variables, in the case they are named alike.

### Example 7.10. `{include}` passing variables

```
{include file='header.tpl' title='Main Menu' table_bgcolor='#c0c0c0'}

{* body of template goes here *}

{include file='footer.tpl' logo='http://my.example.com/logo.gif'}
```

where `header.tpl` could be

```
<table border='1' width='100%' bgcolor='{$table_bgcolor|default:"#0000FF"}'>
  <tr><td>
    <h1>{$title}</h1>
  </td></tr>
</table>
```

**Example 7.11. {include} and assign to variable**

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

```
<body>
{include file='nav.tpl' assign=navbar}
{include file='header.tpl' title='Main Menu' table_bgcolor='#effeef'}
{$navbar}

{* body of template goes here *}

{include file='footer.tpl' logo='http://my.example.com/logo.gif'}
{$navbar}
</body>
```

Use the syntax for template resources to include files outside of the \$template\_dir directory.

**Example 7.12. {include} template 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'}
```

See also {include\_php}, {php}, Template Resources and Componentized Templates.

## {include\_php}

**Technical Note:** {include\_php} is pretty much deprecated from Smarty, you can accomplish the same functionality via a custom template function. The only reason to use {include\_php} is if you really have a need to quarantine the php function away from the plugins directory or your application code. See the componentized template example for details.

Attribute Name	Type	Required	Default	Description
file	string	Yes	<i>n/a</i>	The name of the php file to include
once	boolean	No	<i>true</i>	whether or not to include the php file more than once if included multiple times
assign	string	No	<i>n/a</i>	The name of the variable that the output of

Attribute Name	Type	Required	Default	Description
				include_php will be assigned to

{include\_php} tags are used to include a php script in your template. If security is enabled, then the php script must be located in the \$trusted\_dir path. The {include\_php} tag must have the attribute "file", which contains the path to the included php file, either relative to \$trusted\_dir, or an absolute path.

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 \$this within the PHP script that you include.

### Example 7.13. function {include\_php}

load\_nav.php

```
<?php
// load in variables from a mysql db and assign them to the template
require_once("MySQL.class.php");
$sql = new MySQL;
$sql->query("select * from site_nav_sections order by name",SQL_ALL);
$this->assign('sections',$sql->record);
?>
```

Where index.tpl is:

```
{* absolute path, or relative to $trusted_dir *}
{include_php file="/path/to/load_nav.php"}

{foreach item="curr_section" from=$sections}
  <a href="{ $curr_section.url }">{ $curr_section.name}</a><br />
{/foreach}
```

See also {include}, {php}, {capture}, Template Resources and Componentized Templates

## {insert}

{insert} tags work much like {include} tags, except that {insert} tags are NOT cached when you have template caching 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)
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

Attribute Name	Type	Required	Default	Description
				script that is included before the insert function is called
[var ...]	[var type]	No	<i>n/a</i>	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 7.14. function `{insert}`

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

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 name-space 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 `$trusted_dir`. When security is enabled, the script must reside in `$trusted_dir`.

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.

**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.

## `{ldelim}`, `{rdelim}`

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

### Example 7.15. `{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>
```

See also [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 your {literal} block, consider using {ldelim}{rdelim} to escape the individual delimiters instead.

### Example 7.16. {literal} tags

```
{literal}
<script type="text/javascript">
<!--
function isblank(field) {
    if (field.value == '')
    { return false; }
    else
    {
        document.loginform.submit();
        return true;
    }
}
// -->
</script>
{/literal}
```

See also [Escaping Smarty Parsing](#).

## {php}

{php} tags allow php to be embedded directly into the template. They will not be escaped, regardless of the \$php\_handling setting. This is for advanced users only, not normally needed.

### Example 7.17. {php} tags



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

**Technical Note:** To access PHP variables in {php} blocks you may need to use the PHP global [<http://php.net/global>] keyword.

### Example 7.18. {php} tags with global

```
{php}
global $foo, $bar;
if($foo == $bar){
    // do something
}
{/php}
```

See also `$php_handling`, `{include_php}`, `{include}` and Componentized Templates.

## {section},{sectionelse}

Template sections are used for looping over **arrays of data** (just like {foreach}). All *{section}* tags must be paired with *{/section}* tags. Required parameters are *name* and *loop*. The name of the {section} can be anything you like, made up of letters, numbers and underscores. Sections can be nested, and the nested section names must be unique from each other. The loop variable (usually an array of values) determines the number of times the section will loop. 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.

Attribute Name	Type	Required	Default	Description
name	string	Yes	<i>n/a</i>	The name of the section
loop	mixed	Yes	<i>n/a</i>	Value to determine the number of loop iterations
start	integer	No	<i>0</i>	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	<i>1</i>	The step value that will be used to traverse the

Attribute Name	Type	Required	Default	Description
				loop array. For example, step=2 will loop on index 0,2,4, etc. If step is negative, it will step through the array backwards.
max	integer	No	<i>n/a</i>	Sets the maximum number of times the section will loop.
show	boolean	No	<i>true</i>	determines whether or not to show this section

### Example 7.19. {section}

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

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

Another couple of examples 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 7.20. {section} loop variable**

```
<?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 N 45th', '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 example assumes that $custid, $name and $address are all
 arrays containing the same number of values
*}
{section name=customer loop=$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 N 45th
</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>
```

**Example 7.21. {section} naming**

```
{*
 the name of the section can be anything you like,
 as it is used to reference the data within the section
*}
{section name=anything loop=$custid}
<p>
  id: {$custid[anything]}<br />
  name: {$name[anything]}<br />
  address: {$address[anything]}
</p>
{/section}
```

**Example 7.22. nested sections**

```
<?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 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);
?>
```

```
{*
 sections can be nested as deep as you like. With nested sections,
 you can access complex data structures, such as multi-dimensional
 arrays. In this example, $contact_type[customer] is an array of
 contact types for the current customer.
*}
{section name=customer loop=$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 7.23. sections and associative arrays**

```
<?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);
?>
```

```
{*
  This is an example of printing an associative array
  of data within a section
*}
{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>
```

Database example (eg using Pear or Adodb)

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

```
{*
  output database result in a 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>
{/section}
</table>
```

### Example 7.24. {sectionelse}

```
{* sectionelse will execute if there are no $custid values *}
{section name=customer loop=$custid}
  id: {$custid[customer]}<br />
{sectionelse}
  there are no values in $custid.
{/section}
```

Sections also have their own variables that handle section properties. These are indicated like so: `{$smarty.section.sectionname.varname}`

**Note:** As of Smarty 1.5.0, the syntax for section property variables has changed from `{%sectionname.varname%}` to `{$smarty.section.sectionname.varname}`. The old syntax is still supported, but you will only see examples of the new syntax.

## index

`index` is used to display the current array index, starting with zero (or the start attribute if given), and incrementing by one (or by the step attribute if given.)

**Technical Note:** If the step and start section properties are not modified, then this works the same as the iteration section property, except it starts at 0 instead of 1.

### Example 7.25. {section} property index

```
{* FYI, $custid[customer.index] and $custid[customer] are identical in meaning *}
{section name=customer loop=$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 used to display the previous loop index. on the first loop, this is set to -1.

## index\_next

`index_next` is used to display 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 7.26. {section} property `index_next` and `index_prev`

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

{* FYI, $custid[cus.index] and $custid[cus] 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=cus loop=$custid}
  <tr>
    <td>{$smarty.section.cus.index}</td><td>{$custid[cus]}</td>
    <td>{$smarty.section.cus.index_prev}</td><td>{$custid[cus.index_prev]}</td>
    <td>{$smarty.section.cus.index_next}</td><td>{$custid[cus.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` is used to display the current loop iteration.

**Note:** This is not affected by the section properties `start`, `step` and `max`, unlike the `index` property. Iteration also starts with 1 instead of 0 like `index`. `rownum` is an alias to `iteration`, they work identical.

### Example 7.27. {section} property `iteration`

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

{section name=cu loop=$custid start=5 step=2}
```

```

iteration={$smarty.section.cu.iteration}
index={$smarty.section.cu.index}
id={$custid[cul]}<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 />

```

This example uses the iteration property to output a table header block every five rows (uses {if} with the mod operator).

```

<table>
{section name=co loop=$contacts}
  {if $smarty.section.co.iteration % 5 == 1}
    <tr><th>&nbsp;</th><th>Name</th><th>Home</th><th>Cell</th><th>Email</th></tr>
  {/if}
  <tr>
    <td><a href="view.php?id={$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>

```

## first

first is set to true if the current section iteration is the first one.

## last

last is set to true if the current section iteration is the last one.

### Example 7.28. {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 (uses section 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 is used to display the current loop iteration, starting with one. It is an alias to iteration, they work identically.

## loop

loop is used to display the last index number that this section looped. This can be used inside or after the section.

### Example 7.29. {section} property index

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

There were {\$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 were 3 customers shown above.

## show

*show* is used as a parameter to section. *show* is a boolean value, true or false. If false, the section will not be displayed. If there is a {sectionelse} present, that will be alternately displayed.

### Example 7.30. {section} attribute show

```
{*
  $show_customer_info (true/false) may have been passed from the PHP
  application, to regulate whether or not this section shows
*}
{section name=customer loop=$custid show=$show_customer_info}
  {$smarty.section.customer.rownum} id: {$custid[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 is used to display the number of iterations that this section will loop. This can be used inside or after the section.

### Example 7.31. {section} property total

```
{section name=customer loop=$custid step=2}
  {$smarty.section.customer.index} id: {$custid[customer]}<br />
{/section}

There were {$smarty.section.customer.total} customers shown above.
```

The above example will output:

```
0 id: 1000<br />
2 id: 1002<br />
4 id: 1004<br />

There were 3 customers shown above.
```

See also {foreach} 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.

**Technical Note:** {strip}{/strip} does not affect the contents of template variables. See the strip modifier.

### Example 7.32. {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 strip modifier

---

# Chapter 8. Custom Functions

## Table of Contents

{assign} .....	62
{counter} .....	63
{cycle} .....	64
{debug} .....	65
{eval} .....	65
{fetch} .....	66
{html_checkboxes} .....	67
{html_image} .....	68
{html_options} .....	69
{html_radios} .....	71
{html_select_date} .....	73
{html_select_time} .....	76
{html_table} .....	79
{mailto} .....	81
{math} .....	82
{popup} .....	84
{popup_init} .....	87
{textformat} .....	87

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

## {assign}

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

Attribute Name	Type	Required	Default	Description
var	string	Yes	<i>n/a</i>	The name of the variable being assigned
value	string	Yes	<i>n/a</i>	The value being assigned

### Example 8.1. {assign}

```
{assign var="name" value="Bob"}  
The value of $name is {$name}.
```

The above example will output:

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

### Example 8.2. Accessing {assign} variables from a PHP script.

To access {assign} variables from php use `get_template_vars()`. However, the variables are only available after/during template execution as in the following example

```
{* index.tpl *}
{assign var="foo" value="Smarty"}

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

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

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

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

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

?>
```

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

{capture}, {include}, {include\_php}, {insert}, {counter}, {cycle}, {eval}, {fetch}, {math}, {textformat}

See also `assign()` and `get_template_vars()`.

## {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 special "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	<i>default</i>	The name of the counter
start	number	No	<i>1</i>	The initial number to start counting from
skip	number	No	<i>1</i>	The interval to count by
direction	string	No	<i>up</i>	the direction to count (up/down)
print	boolean	No	<i>true</i>	Whether or not to print the value
assign	string	No	<i>n/a</i>	the template variable the output will be assigned to

### Example 8.3. 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 cycle through a set of values. This makes it easy to alternate for example between two or more colors in a table, or cycle through an array of values.

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

You can {cycle} through more than one set of values in your template by supplying a name attribute. Give each set of values a 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 special "assign" attribute, the output of the cycle function will be assigned to a template variable instead of being output to the template.

### Example 8.4. {cycle}

```
{section name=rows loop=$data}
<tr bgcolor="{cycle values="#e0e0e0,#d0d0d0"}">
  <td>{ $data[rows]}</td>
</tr>
{/section}
```

```
<tr bgcolor="#e0e0e0">
  <td>1</td>
</tr>
<tr bgcolor="#d0d0d0">
  <td>2</td>
</tr>
<tr bgcolor="#e0e0e0">
  <td>3</td>
</tr>
```

## {debug}

Attribute Name	Type	Required	Default	Description
output	string	No	<i>javascript</i>	output type, html or javascript

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

See also Debugging console

## {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 special "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	<i>n/a</i>	variable (or string) to evaluate
assign	string	No	<i>n/a</i>	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.

**Technical Note:** 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 8.5. {eval}

```

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 index.tpl 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 example will output:

```

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

## {fetch}

{fetch} is used to fetch 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. If the file name begins with "ftp://", the file will be fetched from the ftp server and displayed. For local files, the full system file path must be given, or a path relative to the executed php script.

If you supply the special "assign" attribute, the output of the {fetch} function will be assigned to this template variable instead of being output to the template. (new in Smarty 1.5.0)

Attribute Name	Type	Required	Default	Description
file	string	Yes	<i>n/a</i>	the file, http or ftp site to fetch
assign	string	No	<i>n/a</i>	the template variable the output will be assigned to

**Technical Note:** This will not support http redirects, be sure to include a trailing slash on your web page fetches where necessary.

**Technical Note:** If template security is turned on and you are fetching a file from the local file system, this will only allow files from within one of the defined secure directories. (\$secure\_dir)

### Example 8.6. fetch

```
{* 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"}

{* assign the fetched contents to a template variable *}
{fetch file="http://www.myweather.com/68502/" assign="weather"}
{if $weather ne ""}
  <b>{$weather}</b>
{/if}
```

See also {capture}, {eval} 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. Required attributes are values and output, unless you use options instead. All output is XHTML compatible.

Attribute Name	Type	Required	Default	Description
name	string	No	<i>checkbox</i>	name of checkbox list
values	array	Yes, unless using options attribute	<i>n/a</i>	an array of values for checkbox buttons
output	array	Yes, unless using options attribute	<i>n/a</i>	an array of output for checkbox buttons
selected	string/array	No	<i>empty</i>	the selected checkbox element(s)
options	associative array	Yes, unless using values and output	<i>n/a</i>	an associative array of values and output
separator	string	No	<i>empty</i>	string of text to separate each checkbox item
labels	boolean	No	<i>true</i>	add <label>-tags to the output

All parameters that are not in the list above are printed as name/value-pairs inside each of the created <input>-tags.

### Example 8.7. {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 Smith</label>
<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 8.8. Database example (eg PEAR or ADODB):

```
<?php
$sql = 'select type_id, types from types order by type';
$smarty->assign('types', $db->getAssoc($sql));

$sql = 'select * from contacts where contact_id=12';
$smarty->assign('contact', $db->getRow($sql));
?>
```

```
{html_checkboxes name="type" options=$types selected=$contact.type_id separator="<br />"}
```

See also {html\_radios} and {html\_options}

## {html\_image}

{html\_image} is a custom function that generates an HTML tag for an image. The height and width are automatically calculated from the image file if none are supplied.

Attribute Name	Type	Required	Default	Description
file	string	Yes	<i>n/a</i>	name/path to image
height	string	No	<i>actual image height</i>	height to display image

Attribute Name	Type	Required	Default	Description
width	string	No	<i>actual image width</i>	width to display image
basedir	string	no	<i>web server doc root</i>	directory to base relative paths from
alt	string	no	<i>""</i>	alternative description of the image
href	string	no	<i>n/a</i>	href value to link the image to
path_prefix	string	no	<i>n/a</i>	prefix for output path

`basedir` is the base directory that relative image paths are based from. If not given, the web server document root ( env variable `DOCUMENT_ROOT`) is used as the base. If `$security` is enabled, the path to the image must be within a secure directory.

`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 8.9. html\_image example

where `index.tpl` is:

```
-----
{html_image file="pumpkin.jpg"}
{html_image file="/path/from/docroot/pumpkin.jpg"}
{html_image file="../path/relative/to/currdir/pumpkin.jpg" }
```

possible output would be:

```



```

## {html\_options}

`{html_options}` is a custom function that creates html `<select><option>` group with provided data. It takes care of which item(s) are selected by default as well. Required attributes are values and output, unless you use options instead.

Attribute Name	Type	Required	Default	Description
values	array	Yes, unless using options attribute	<i>n/a</i>	an array of values for dropdown
output	array	Yes, unless using options attribute	<i>n/a</i>	an array of output for dropdown
selected	string/array	No	<i>empty</i>	the selected option ele-

Attribute Name	Type	Required	Default	Description
				ment(s)
options	associative array	Yes, unless using values and output	<i>n/a</i>	an associative array of values and output
name	string	No	<i>empty</i>	name of select group

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 output is XHTML compatible.

If the optional *name* attribute is given, the <select name="groupname"></select> tags will enclose the option list. Otherwise only the <option> list is generated.

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.

### Example 8.10. {html\_options}

#### Example 1:

```
<?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:

```
<select name="customer_id">
    {html_options values=$cust_ids output=$cust_names selected=$customer_id}
</select>
```

#### Example 2:

```
<?php
$smarty->assign('cust_options', array(
    1000 => 'Joe Schmoe',
    1001 => 'Jack Smith',
    1002 => 'Jane Johnson',
    1003 => 'Charlie Brown')
);
$smarty->assign('customer_id', 1001);
?>
```

where template is:

```
{html_options name=customer_id options=$cust_options selected=$customer_id}
```

Both the above examples will output:

```
<select name="customer_id">
```

```
<option label="Joe Schmoe" value="1000">Joe Schmoe</option>
<option label="Jack Smith" value="1001" selected="selected">Jack Smith</option>
<option label="Jane Johnson" value="1002">Jane Johnson</option>
<option label="Charlie Brown" value="1003">Charlie Brown</option>
</select>
```

**Example 8.11. {html\_options} - Database example (eg PEAR or ADODB):**

```
<?php
$sql = 'select type_id, types from types order by type';
$smarty->assign('types', $db->getAssoc($sql));

$sql = 'select contact_id, name, email, type_id
       from contacts where contact_id='.$contact_id;
$smarty->assign('contact', $db->getRow($sql));
?>
```

where the template is:

```
<select name="type_id">
  <option value='null'>-- none --</option>
  {html_options name="type" options=$types selected=$contact.type_id}
</select>
```

See also {html\_checkboxes} and {html\_radios}

## {html\_radios}

{html\_radios} is a custom function that creates html radio button group with provided data. It takes care of which item is selected by default as well. Required attributes are values and output, unless you use options instead. All output is XHTML compatible.

Attribute Name	Type	Required	Default	Description
name	string	No	<i>radio</i>	name of radio list
values	array	Yes, unless using options attribute	<i>n/a</i>	an array of values for radio buttons
output	array	Yes, unless using options attribute	<i>n/a</i>	an array of output for radio buttons
selected	string	No	<i>empty</i>	the selected radio element
options	associative array	Yes, unless using values and output	<i>n/a</i>	an associative array of values and output
separator	string	No	<i>empty</i>	string of text to separate each radio item

All parameters that are not in the list above are printed as name/value-pairs inside each of the created <input>-tags.

**Example 8.12. {html\_radios} : Example 1**

```
<?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 8.13. {html\_radios} : Example 2

```
<?php
$smarty->assign('cust_radios', array(
    1000 => 'Joe Schmoe',
    1001 => 'Jack Smith',
    1002 => 'Jane Johnson',
    1003 => 'Charlie Brown'));
$smarty->assign('customer_id', 1001);
?>
```

Where index.tpl 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" checked="checked" />Jack S
<label for="id_1002"><input type="radio" name="id" value="1002" id="id_1002" />Jane Johnson</label><br />
<label for="id_1003"><input type="radio" name="id" value="1003" id="id_1003" />Charlie Brown</label><br />
```

### Example 8.14. {html\_radios}- Database example (eg PEAR or ADODB):

```
<?php
$sql = 'select type_id, types from types order by type';
$smarty->assign('types', $db->getAssoc($sql));

$sql = 'select contact_id, name, email, type_id
  from contacts where contact_id='.$contact_id;
$smarty->assign('contact', $db->getRow($sql));
?>
```

and the template:

```
{html_radios name="type" options=$types selected=$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 for you. It can display any or all of year, month, and day.

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
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

Custom Functions

Attribute Name	Type	Required	Default	Description
				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
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



Attribute Name	Type	Required	Default	Description
				and "" as it's value. Note that you can use values like "YYYY-MM-" as time-attribute to indicate an unselected day.

All parameters that are not in the list above are printed as name/value-pairs inside the <select> tags of day, month and year.

### Example 8.15. {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="2001" selected="selected">2001</option>
</select>
```

### Example 8.16. {html\_select\_date}

```
{* 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}
```

This will output: (current year is 2000)

```
<select name="StartDateMonth">
<option value="1">January</option>
<option value="2">February</option>
<option value="3">March</option>
```

```

<option value="4">April</option>
<option value="5">May</option>
<option value="6">June</option>
<option value="7">July</option>
<option value="8">August</option>
<option value="9">September</option>
<option value="10">October</option>
<option value="11">November</option>
<option value="12" selected="selected">December</option>
</select>
<select name="StartDateYear">
<option value="1995">1995</option>
<option value="1996">1996</option>
<option value="1997">1997</option>
<option value="1998">1998</option>
<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 date tips.

## {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.

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
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

Attribute Name	Type	Required	Default	Description
				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

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>].

### Example 8.17. {html\_select\_time}

template code:

```
{html_select_time use_24_hours=true}
```

This will output:

```
<select name="Time_Hour">
<option value="00">00</option>
<option value="01">01</option>
<option value="02">02</option>
<option value="03">03</option>
<option value="04">04</option>
<option value="05">05</option>
<option value="06">06</option>
<option value="07">07</option>
<option value="08">08</option>
<option value="09" selected>09</option>
<option value="10">10</option>
<option value="11">11</option>
<option value="12">12</option>
<option value="13">13</option>
<option value="14">14</option>
<option value="15">15</option>
<option value="16">16</option>
<option value="17">17</option>
<option value="18">18</option>
<option value="19">19</option>
<option value="20">20</option>
<option value="21">21</option>
<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>
<option value="02">02</option>
<option value="03">03</option>
<option value="04">04</option>
<option value="05">05</option>
<option value="06">06</option>
<option value="07">07</option>
<option value="08">08</option>
<option value="09">09</option>
<option value="10">10</option>
<option value="11">11</option>
<option value="12">12</option>
<option value="13">13</option>
<option value="14">14</option>
```

```
<option value="15">15</option>
<option value="16">16</option>
<option value="17">17</option>
<option value="18">18</option>
<option value="19">19</option>
<option value="20" selected>20</option>
<option value="21">21</option>
<option value="22">22</option>
<option value="23">23</option>
<option value="24">24</option>
<option value="25">25</option>
<option value="26">26</option>
<option value="27">27</option>
<option value="28">28</option>
<option value="29">29</option>
<option value="30">30</option>
<option value="31">31</option>
<option value="32">32</option>
<option value="33">33</option>
<option value="34">34</option>
<option value="35">35</option>
<option value="36">36</option>
<option value="37">37</option>
<option value="38">38</option>
<option value="39">39</option>
<option value="40">40</option>
<option value="41">41</option>
<option value="42">42</option>
<option value="43">43</option>
<option value="44">44</option>
<option value="45">45</option>
<option value="46">46</option>
<option value="47">47</option>
<option value="48">48</option>
<option value="49">49</option>
<option value="50">50</option>
<option value="51">51</option>
<option value="52">52</option>
<option value="53">53</option>
<option value="54">54</option>
<option value="55">55</option>
<option value="56">56</option>
<option value="57">57</option>
<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>
<option value="02">02</option>
<option value="03">03</option>
<option value="04">04</option>
<option value="05">05</option>
<option value="06">06</option>
<option value="07">07</option>
<option value="08">08</option>
<option value="09">09</option>
<option value="10">10</option>
<option value="11">11</option>
<option value="12">12</option>
<option value="13">13</option>
<option value="14">14</option>
<option value="15">15</option>
<option value="16">16</option>
<option value="17">17</option>
<option value="18">18</option>
<option value="19">19</option>
<option value="20">20</option>
<option value="21">21</option>
<option value="22">22</option>
<option value="23" selected>23</option>
```

```

<option value="24">24</option>
<option value="25">25</option>
<option value="26">26</option>
<option value="27">27</option>
<option value="28">28</option>
<option value="29">29</option>
<option value="30">30</option>
<option value="31">31</option>
<option value="32">32</option>
<option value="33">33</option>
<option value="34">34</option>
<option value="35">35</option>
<option value="36">36</option>
<option value="37">37</option>
<option value="38">38</option>
<option value="39">39</option>
<option value="40">40</option>
<option value="41">41</option>
<option value="42">42</option>
<option value="43">43</option>
<option value="44">44</option>
<option value="45">45</option>
<option value="46">46</option>
<option value="47">47</option>
<option value="48">48</option>
<option value="49">49</option>
<option value="50">50</option>
<option value="51">51</option>
<option value="52">52</option>
<option value="53">53</option>
<option value="54">54</option>
<option value="55">55</option>
<option value="56">56</option>
<option value="57">57</option>
<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 date tips.

## {html\_table}

`{html_table}` is a custom function that dumps an array of data into an HTML table. 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.

Attribute Name	Type	Required	Default	Description
loop	array	Yes	<i>n/a</i>	array of data to loop through
cols	integer	No	3	number of columns in the table. 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

Attribute Name	Type	Required	Default	Description
				display to be just enough cols to display all elements. If both, rows and cols, are omitted cols defaults to 3.
rows	integer	No	<i>empty</i>	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	<i>cols</i>	direction of consecutive elements in the loop-array to be rendered. <i>cols</i> means elements are displayed col-by-col. <i>rows</i> means elements are displayed row-by-row.
table_attr	string	No	<i>border="1"</i>	attributes for table tag
tr_attr	string	No	<i>empty</i>	attributes for tr tag (arrays are cycled)
td_attr	string	No	<i>empty</i>	attributes for td tag (arrays are cycled)
trailpad	string	No	<i>&amp;nbsp;</i>	value to pad the trailing cells on last row with (if any)
hdir	string	No	<i>right</i>	direction of each row to be rendered. possible values: <i>right</i> (left-to-right), and <i>left</i> (right-to-left)
vdir	string	No	<i>down</i>	direction of each column to be rendered. possible values: <i>down</i> (top-to-bottom), <i>up</i> (bottom-to-top)

**Example 8.18. html\_table**

```

php code:
-----
<?php
require('Smarty.class.php');
$smarty = new Smarty;
$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');

```

?>

template code:

```
-----
{html_table loop=$data}
{html_table loop=$data cols=4 table_attr='border="0"'}
{html_table loop=$data cols=4 tr_attr=$tr}
```

The above example will output:

```
<table border="1">
<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>
</table>
<table border="0">
<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>
</table>
<table border="1">
<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>
</table>
```

## {mailto}

{mailto} automates the creation of mailto: links and optionally encodes them. Encoding e-mails makes it more difficult for web spiders to lift e-mail addresses off of your site.

Attribute Name	Type	Required	Default	Description
address	string	Yes	<i>n/a</i>	the e-mail address
text	string	No	<i>n/a</i>	the text to display, default is the e-mail address
encode	string	No	<i>none</i>	How to encode the e-mail. Can be one of none, hex, javascript or javascript_charcode.
cc	string	No	<i>n/a</i>	e-mail addresses to carbon copy. Separate entries by a comma.
bcc	string	No	<i>n/a</i>	e-mail addresses to blind carbon copy. Separate entries by a comma.
subject	string	No	<i>n/a</i>	e-mail subject.
newsgroups	string	No	<i>n/a</i>	newsgroups to post to. Separate entries by a comma.
followupto	string	No	<i>n/a</i>	addresses to follow up to. Separate entries by a comma.

Attribute Name	Type	Required	Default	Description
extra	string	No	<i>n/a</i>	any extra information you want passed to the link, such as style sheet classes

**Technical Note:** javascript is probably the most thorough form of encoding, although you can use hex encoding too.

### Example 8.19. {mailto}

```
{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...%6f%6d">&#x6d;&#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%20they@example.com" >me@example.com</a>

{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`, `Obfuscating E-mail Addresses` and `{textformat}`

## {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`, `srand` and `tan` are all valid operators. Check the PHP documentation for further information on these math functions.

If you supply the special "assign" attribute, the output of the math function will be assigned to this template variable instead of being output to the template.

Attribute Name	Type	Required	Default	Description
equation	string	Yes	<i>n/a</i>	the equation to execute
format	string	No	<i>n/a</i>	the format of the result (sprintf)



Attribute Name	Type	Required	Default	Description
var	numeric	Yes	<i>n/a</i>	equation variable value
assign	string	No	<i>n/a</i>	template variable the output will be assigned to
[var ...]	numeric	Yes	<i>n/a</i>	equation variable value

**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 PHP and `assign()` the results to the template. Definately avoid repetitive `{math}` function calls, eg within `{section}` loops.

### Example 8.20. `{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:

## {popup}

{popup} is used to create javascript popup windows. {popup\_init} MUST be called first for this to work.

Attribute Name	Type	Required	Default	Description
text	string	Yes	<i>n/a</i>	the text/html to display in the popup window
trigger	string	No	<i>onMouseOver</i>	What is used to trigger the popup window. Can be one of on-MouseOver or onClick
sticky	boolean	No	<i>false</i>	Makes the popup stick around until closed
caption	string	No	<i>n/a</i>	sets the caption to title
fgcolor	string	No	<i>n/a</i>	color of the inside of the popup box
bgcolor	string	No	<i>n/a</i>	color of the border of the popup box
textcolor	string	No	<i>n/a</i>	sets the color of the text inside the box
capcolor	string	No	<i>n/a</i>	sets color of the box's caption
closecolor	string	No	<i>n/a</i>	sets the color of the close text
textfont	string	No	<i>n/a</i>	sets the font to be used by the main text
captionfont	string	No	<i>n/a</i>	sets the font of the caption
closefont	string	No	<i>n/a</i>	sets the font for the "Close" text
textsize	string	No	<i>n/a</i>	sets the size of the main text's font
captionsize	string	No	<i>n/a</i>	sets the size of the caption's font
closesize	string	No	<i>n/a</i>	sets the size of the "Close" text's font
width	integer	No	<i>n/a</i>	sets the width of the box
height	integer	No	<i>n/a</i>	sets the height of the box
left	boolean	No	<i>false</i>	makes the popups go to the left of the mouse
right	boolean	No	<i>false</i>	makes the popups go to the right of the mouse
center	boolean	No	<i>false</i>	makes the popups go to

Custom Functions

Attribute Name	Type	Required	Default	Description
				the center of the mouse
above	boolean	No	<i>false</i>	makes the popups go above the mouse. NOTE: only possible when height has been set
below	boolean	No	<i>false</i>	makes the popups go below the mouse
border	integer	No	<i>n/a</i>	makes the border of the popups thicker or thinner
offsetx	integer	No	<i>n/a</i>	how far away from the pointer the popup will show up, horizontally
offsety	integer	No	<i>n/a</i>	how far away from the pointer the popup will show up, vertically
fgbackground	url to image	No	<i>n/a</i>	defines a picture to use instead of color for the inside of the popup.
bgbackground	url to image	No	<i>n/a</i>	defines a picture to use instead of color for the border of the popup. NOTE: You will want to set bgcolor to "" or the color will show as well. NOTE: When having a Close link, Netscape will re-render the table cells, making things look incorrect
closetext	string	No	<i>n/a</i>	sets the "Close" text to something else
noclose	boolean	No	<i>n/a</i>	does not display the "Close" text on stickies with a caption
status	string	No	<i>n/a</i>	sets the text in the browsers status bar
autostatus	boolean	No	<i>n/a</i>	sets the status bar's text to the popup's text. NOTE: overrides status setting
autostatuscap	string	No	<i>n/a</i>	sets the status bar's text to the caption's text. NOTE: overrides status and autostatus settings
inarray	integer	No	<i>n/a</i>	tells overLib to read text from this index in the ol_text array, located in overlib.js. This parameter can be used instead of text

Custom Functions

Attribute Name	Type	Required	Default	Description
caparray	integer	No	<i>n/a</i>	tells overLib to read the caption from this index in the ol_caps array
capicon	url	No	<i>n/a</i>	displays the image given before the popup caption
snapx	integer	No	<i>n/a</i>	snaps the popup to an even position in a horizontal grid
snapy	integer	No	<i>n/a</i>	snaps the popup to an even position in a vertical grid
fixx	integer	No	<i>n/a</i>	locks the popups horizontal position Note: overrides all other horizontal placement
fixy	integer	No	<i>n/a</i>	locks the popups vertical position Note: overrides all other vertical placement
background	url	No	<i>n/a</i>	sets image to be used instead of table box background
padx	integer,integer	No	<i>n/a</i>	pads the background image with horizontal whitespace for text placement. Note: this is a two parameter command
pady	integer,integer	No	<i>n/a</i>	pads the background image with vertical whitespace for text placement. Note: this is a two parameter command
fullhtml	boolean	No	<i>n/a</i>	allows you to control the html over a background picture completely. The html code is expected in the "text" attribute
frame	string	No	<i>n/a</i>	controls popups in a different frame. See the overlib page for more info on this function
timeout	string	No	<i>n/a</i>	calls the specified javascript function and takes the return value as the text that should be displayed in the popup window
delay	integer	No	<i>n/a</i>	makes that popup be-

Attribute Name	Type	Required	Default	Description
				have like a tooltip. It will popup only after this delay in milliseconds
hauto	boolean	No	<i>n/a</i>	automatically determine if the popup should be to the left or right of the mouse.
vauto	boolean	No	<i>n/a</i>	automatically determine if the popup should be above or below the mouse.

### Example 8.21. {popup}

```
{* popup_init must be called once at the top of the page *}
{popup_init src="/javascripts/overlib.js"}

{* create a link with a popup window when you move your mouse over *}
<a href="mypage.html" {popup text="This link takes you to my page!"}>mypage</a>

{* you can use html, links, etc in your popup text *}
<a href="mypage.html" {popup sticky=true caption="mypage contents"
text="<ul><li>links</li><li>pages</li><li>images</li></ul>"
snapx=10 snapy=10}>mypage</a>
```

See also {popup\_init} and overLib [<http://www.bosrup.com/web/overlib/>].

## {popup\_init}

{popup} is an integration of overLib [<http://www.bosrup.com/web/overlib/>], a library used for popup windows. These are used for context sensitive information, such as help windows or tooltips. {popup\_init} must be called once at the top of any page you plan on using the {popup} function.

overLib [<http://www.bosrup.com/web/overlib/>] was written by Erik Bosrup, and the homepage/download is located at <http://www.bosrup.com/web/overlib/>.

### Example 8.22. {popup\_init}

```
<head>
{* popup_init must be called once at the top of the page *}
{popup_init src="javascripts/overlib.js"}
</head>
```

## {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	<i>n/a</i>	preset style
indent	number	No	<i>0</i>	The number of chars to indent every line
indent_first	number	No	<i>0</i>	The number of chars to indent the first line
indent_char	string	No	<i>(single space)</i>	The character (or string of chars) to indent with
wrap	number	No	<i>80</i>	How many characters to wrap each line to
wrap_char	string	No	<i>\n</i>	The character (or string of chars) to break each line with
wrap_cut	boolean	No	<i>false</i>	If true, wrap will break the line at the exact character instead of at a word boundary
assign	string	No	<i>n/a</i>	the template variable the output will be assigned to

**Example 8.23. {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.

This config file example 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 function `{config_load}` (see also `config_load()`).

You can hide variables or entire sections by prepending the variable name or section name with a period. 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 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 for debug.tpl (this is in `SMARTY_DIR` by default.) When you load the page, a javascript console window should 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 templates, 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, `$error_reporting` and `trigger_error()`.

---

# Part III. Smarty For Programmers

## Table of Contents

11. Constants .....	95
SMARTY_DIR .....	95
SMARTY_CORE_DIR .....	95
12. Smarty Class Variables .....	96
\$template_dir .....	96
\$compile_dir .....	96
\$config_dir .....	97
\$plugins_dir .....	97
\$debugging .....	97
\$debug_tpl .....	98
\$debugging_ctrl .....	98
\$autoload_filters .....	98
\$compile_check .....	98
\$force_compile .....	98
\$caching .....	98
\$cache_dir .....	99
\$cache_lifetime .....	99
\$cache_handler_func .....	99
\$cache_modified_check .....	99
\$config_overwrite .....	99
\$config_booleanize .....	100
\$config_read_hidden .....	100
\$config_fix_newlines .....	100
\$default_template_handler_func .....	100
\$php_handling .....	100
\$security .....	101
\$secure_dir .....	101
\$security_settings .....	101
\$trusted_dir .....	101
\$left_delimiter .....	101
\$right_delimiter .....	102
\$compiler_class .....	102
\$request_vars_order .....	102
\$request_use_auto_globals .....	102
\$error_reporting .....	102
\$compile_id .....	102
\$use_sub_dirs .....	103
\$default_modifiers .....	103
\$default_resource_type .....	103
13. Smarty Class Methods() .....	104
14. Caching .....	145
Setting Up Caching .....	145
Multiple Caches Per Page .....	147
Cache Groups .....	148
Controlling Cacheability of Plugins' Output .....	149
15. Advanced Features .....	151
Objects .....	151
Prefilters .....	152

Postfilters .....	152
Output Filters .....	153
Cache Handler Function .....	153
Resources .....	155
16. Extending Smarty With Plugins .....	159
How Plugins Work .....	159
Naming Conventions .....	159
Writing Plugins .....	160
Template Functions .....	160
Modifiers .....	162
Block Functions .....	163
Compiler Functions .....	164
Prefilters/Postfilters .....	165
Output Filters .....	166
Resources .....	166
Inserts .....	168

---

# Chapter 11. Constants

## Table of Contents

SMARTY_DIR .....	95
SMARTY_CORE_DIR .....	95

## SMARTY\_DIR

This should be the **full system path** to the location of the Smarty class files. If this is not defined, then Smarty will attempt to determine the appropriate value automatically. If defined, the path **must end with a slash**.

### Example 11.1. SMARTY\_DIR

```
<?php
// set path to Smarty directory *nix style
define('SMARTY_DIR', '/usr/local/lib/php/Smarty/libs/');

// path to Smarty windows style
define('SMARTY_DIR', 'c:/webroot/libs/Smarty/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

## SMARTY\_CORE\_DIR

This should be the full system path to the location of the Smarty core files. If not defined, Smarty will default this constant to the *internals/* sub-directory below SMARTY\_DIR. If defined, the path must end with a slash. Use this constant when manually including any of the core.\* files.

### Example 11.2. SMARTY\_CORE\_DIR

```
<?php
// load core.get_microtime.php
require_once(SMARTY_CORE_DIR . 'core.get_microtime.php');
?>
```

See also `$smarty.const`

---

# Chapter 12. Smarty Class Variables

## Table of Contents

\$template_dir .....	96
\$compile_dir .....	96
\$config_dir .....	97
\$plugins_dir .....	97
\$debugging .....	97
\$debug_tpl .....	98
\$debugging_ctrl .....	98
\$autoload_filters .....	98
\$compile_check .....	98
\$force_compile .....	98
\$caching .....	98
\$cache_dir .....	99
\$cache_lifetime .....	99
\$cache_handler_func .....	99
\$cache_modified_check .....	99
\$config_overwrite .....	99
\$config_booleanize .....	100
\$config_read_hidden .....	100
\$config_fix_newlines .....	100
\$default_template_handler_func .....	100
\$php_handling .....	100
\$security .....	101
\$secure_dir .....	101
\$security_settings .....	101
\$trusted_dir .....	101
\$left_delimiter .....	101
\$right_delimiter .....	102
\$compiler_class .....	102
\$request_vars_order .....	102
\$request_use_auto_globals .....	102
\$error_reporting .....	102
\$compile_id .....	102
\$use_sub_dirs .....	103
\$default_modifiers .....	103
\$default_resource_type .....	103

## \$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 it will look for the compile directory in the same directory as the executing php script. **This directory must be writable by the web server** (see install). 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 `$compile_id` and `$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 `cwd` (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 `cwd`.

### 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 template page. It does not show variables assigned within a template with `{assign}`.

Also see `$debugging_ctrl` on how to enable debugging from the url.

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 Debugging console

## \$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.

See also Debugging console.

## \$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 `register_outputfilter()`, `register_prefilter()`, `register_postfilter()` and `load_filter()`

## \$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 (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` or `clear_compiled_tpl()`.

## \$force\_compile

This forces Smarty to (re)compile templates on every invocation. This setting overrides `$compile_check`. By default this is disabled. This is handy for development and debugging. It should never be used in a production environment. If `$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 `$cache_dir`. By default this is set to 0, or disabled. If your templates generate redundant content, it is advisable to turn on `$caching`. This will result in significant performance gains. You can also have multiple caches for the same template. A value of 1 or 2 enables caching. 1 tells Smarty



to use the current `$cache_lifetime` variable to determine if the cache has expired. A value of 2 tells Smarty to use the `$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 `is_cached()`.

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` and Caching section.

## `$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). 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 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. `$caching` must be set to "true" for `$cache_lifetime` to have any purpose. A 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 = false`.)

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.

**Technical Note:** If you want to give certain templates their own cache lifetime, you could do this by setting `$caching = 2`, then set `$cache_lifetime` to a unique value just before calling `display()` or `fetch()`.

## `$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 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 Caching section.

## `$config_overwrite`

If set to true (by default), 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 12.3. 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}, config files, get\_config\_vars(), clear\_config() and config\_load().

## \$config\_booleanize

If set to true, config file 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. true by default.

## \$config\_read\_hidden

If set to true, hidden sections (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 (\r and \r\n) in config files are converted to \n when they are parsed. true by default.

## \$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, 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.

## \$security

\$security true/false, default is false. Security is good for situations when you have untrusted parties editing the templates (via ftp for example) and you want to reduce the risk of system security compromises through the template language. Turning on security enforces the following rules to the template language, unless specifically overridden with \$security\_settings:

- If \$php\_handling is set to SMARTY\_PHP\_ALLOW, this is implicitly changed to SMARTY\_PHP\_PASSTHRU
- PHP functions are not allowed in {if} statements, except those specified in the \$security\_settings
- templates can only be included from directories listed in the \$secure\_dir array
- local files can only be fetched from directories listed in the \$secure\_dir array using {fetch}
- {php}{/php} tags are not allowed
- PHP functions are not allowed as modifiers, except those specified in the \$security\_settings

## \$secure\_dir

This is an array of all local files and directories that are considered secure. {include} and {fetch} use this when security is enabled.

See also Security settings, and \$trusted\_dir.

## \$security\_settings

These are used to override or specify the security settings when security is enabled. These are the possible settings:

- PHP\_HANDLING - true/false. If set to true, the \$php\_handling setting is not checked for security.
- IF\_FUNCS - This is an array of the names of permitted PHP functions in IF statements.
- INCLUDE\_ANY - true/false. If set to true, any template can be included from the file system, regardless of the \$secure\_dir list.
- PHP\_TAGS - true/false. If set to true, {php}{/php} tags are permitted in the templates.
- MODIFIER\_FUNCS - This is an array of the names of permitted PHP functions used as variable modifiers.
- ALLOW\_CONSTANTS - true/false. If set to true, constants via {\$smarty.const.name} are allowed in the templates. The defaults is set to "false" for security.

## \$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 `$HTTP_*_VARS[]` (`$request_use_auto_globals=false`) or `$_*[]` (`$request_use_auto_globals=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.

## **`$error_reporting`**

When this value is set to a non-null-value it's value is used as php's `error_reporting` [[http://php.net/error\\_reporting](http://php.net/error_reporting)] level inside of `display()` and `fetch()`. When debugging is enabled this value is ignored and the error-level is left untouched.

See also `trigger_error()`, `debugging` and `Troubleshooting`.

## **`$compile_id`**

Persistent 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 should use the current language as `$compile_id` and you will get a set of compiled templates for each language you use.

Another example would be to use the same compile directory across multiple domains / multiple virtual hosts.

### **Example 12.4. `$compile_id` in a virtual host enviroment**

```
<?php
$smarty->compile_id = $_SERVER['SERVER_NAME'];
$smarty->compile_dir = '/path/to/shared_compile_dir';
?>
```

## \$use\_sub\_dirs

Smarty will create subdirectories under the templates\_c and cache directories if \$use\_sub\_dirs is set to true. 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. The default is false (disabled). Sub directories are more efficient, so use them if you can.

Theoretically you get much better performance 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.

**Note:** Since Smarty-2.6.2 \$use\_sub\_dirs defaults to false.

See also \$compile\_dir, and \$cache\_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 details.

---

# Chapter 13. Smarty Class Methods()

## Table of Contents

append()	105
append_by_ref()	106
assign()	107
assign_by_ref()	108
clear_all_assign()	109
clear_all_cache()	110
clear_assign()	111
clear_cache()	112
clear_compiled_tpl()	113
clear_config()	114
config_load()	115
display()	116
fetch()	118
get_config_vars()	120
get_registered_object()	121
get_template_vars()	122
is_cached()	123
load_filter()	124
register_block()	125
register_compiler_function()	126
register_function()	127
register_modifier()	128
register_object()	129
register_outputfilter()	130
register_postfilter()	131
register_prefilter()	132
register_resource()	133
trigger_error()	134
template_exists()	135
unregister_block()	136
unregister_compiler_function()	137
unregister_function	138
unregister_modifier()	139
unregister_object()	140
unregister_outputfilter()	141
unregister_postfilter()	142
unregister_prefilter()	143
unregister_resource()	144

## append()

append()append an element to an assigned array

append()

## 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 `array_merge()` [[http://php.net/array\\_merge](http://php.net/array_merge)] function of PHP which wipes out numerical keys and renumbers them.

### Example 13.1. append

```
<?php
// passing name/value pairs
$smarty->append("Name", "Fred");
$smarty->append("Address", $address);

// passing an associative array
$smarty->append(array('city' => 'Lincoln', 'state' => 'Nebraska'));
?>
```

See also `append_by_ref()`, `assign()` and `get_template_vars()`

## append\_by\_ref()

append\_by\_ref() append values by reference

append\_by\_ref()

## Description

void **append\_by\_ref** (string varname, mixed var [, bool merge])

This is used to append values to the templates by reference. If you append a variable by reference then change its value, the appended value sees the change as well. For objects, `append_by_ref()` also avoids an in-memory copy of the appended object. See the PHP manual on variable referencing for an in-depth explanation. 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 `array_merge()` [[http://php.net/array\\_merge](http://php.net/array_merge)] function of PHP which wipes out numerical keys and renumbers them.

### Example 13.2. append\_by\_ref

```
<?php
// appending name/value pairs
$smarty->append_by_ref('Name', $myname);
$smarty->append_by_ref('Address', $address);
?>
```

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



## assign()

assign() assign values to the templates

assign()

## 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.

### 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 a row from a database (eg adodb)
$sql = 'select id, name, email from contacts where contact = '.$id;
$smarty->assign('contact', $db->getRow($sql));
?>
```

Access these in the template with

```
{* note the vars are case sensitive like php *}
{$Name}
{$Address}
{$city}
{$state}

{$contact.id}, {$contact.name}, {$contact.email}
```

For more complex array assignments see {foreach} and {section}

See also assign\_by\_ref(), get\_template\_vars(), clear\_assign(), append() and {assign}

## assign\_by\_ref()

assign\_by\_ref() assign values by reference

assign\_by\_ref()

## Description

void **assign\_by\_ref** (string varname, mixed var)

This is used to assign() values to the templates by reference instead of making a copy. See the PHP manual on variable referencing for an explanation.

**Technical Note:** This is used to assign values to the templates by reference. If you assign a variable by reference then change its value, the assigned value sees the change as well. For objects, assign\_by\_ref() also avoids an in-memory copy of the assigned object. See the PHP manual on variable referencing for an in-depth explanation.

### Example 13.4. assign\_by\_ref()

```
<?php
// passing name/value pairs
$smarty->assign_by_ref('Name', $myname);
$smarty->assign_by_ref('Address', $address);
?>
```

See also assign(), clear\_all\_assign(), append() and {assign}

## clear\_all\_assign()

clear\_all\_assign() clears the values of all assigned variables

clear\_all\_assign()

## Description

void **clear\_all\_assign** (void)

### Example 13.5. clear\_all\_assign()

```
<?php
// passing name/value pairs
$smarty->assign('Name', 'Fred');
$smarty->assign('Address', $address);

// will output above
print_r( $smarty->get_template_vars() );

// clear all assigned variables
$smarty->clear_all_assign();

// will output nothing
print_r( $smarty->get_template_vars() );
?>
```

See also [clear\\_assign\(\)](#), [clear\\_config\(\)](#), [assign\(\)](#) and [append\(\)](#)

## clear\_all\_cache()

clear\_all\_cache() clears the entire template cache

clear\_all\_cache()

## Description

void **clear\_all\_cache** ([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. clear\_all\_cache

```
<?php
// clear the entire cache
$smarty->clear_all_cache();
?>
```

See also clear\_cache(), is\_cached() and caching

## clear\_assign()

clear\_assign() clears the value of an assigned variable

clear\_assign()

## Description

void **clear\_assign** (mixed var)

This can be a single value, or an array of values.

### Example 13.7. clear\_assign()

```
<?php
// clear a single variable
$smarty->clear_assign('Name');

// clears multiple variables
$smarty->clear_assign(array('Name', 'Address', 'Zip'));
?>
```

See also clear\_all\_assign(), clear\_config(), get\_template\_vars(), assign() and append()

## clear\_cache()

clear\_cache() clears the cache for a specific template

clear\_cache()

## Description

void **clear\_cache** (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. clear\_cache()

```
<?php
// clear the cache for a template
$smarty->clear_cache('index.tpl');

// clear the cache for a particular cache id in an multiple-cache template
$smarty->clear_cache('index.tpl', 'CACHEID');
?>
```

See also clear\_all\_cache() and caching.

## clear\_compiled\_tpl()

clear\_compiled\_tpl() clears the compiled version of the specified template resource

clear\_compiled\_tpl()

## Description

void **clear\_compiled\_tpl** ([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. clear\_compiled\_tpl()

```
<?php
// clear a specific template resource
$smarty->clear_compiled_tpl("index.tpl");

// clear entire compile directory
$smarty->clear_compiled_tpl();
?>
```

## clear\_config()

clear\_config() clears assigned config variables

clear\_config()

## Description

void **clear\_config** ([string var])

This clears all assigned config variables. If a variable name is supplied, only that variable is cleared.

### Example 13.10. clear\_config()

```
<?php
// clear all assigned config variables.
$smarty->clear_config();

// clear one variable
$smarty->clear_config('foobar');
?>
```

See also [get\\_config\\_vars\(\)](#), [config variables](#), [config files](#), [{config\\_load}](#), [config\\_load\(\)](#) and [clear\\_assign\(\)](#)



## config\_load()

config\_load() loads config file data and assigns it to the template

config\_load()

## Description

void **config\_load** (string file [, string section])

This loads config file data and assigns it to the template. This works identical 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 config\_load() are always global scope. Config files are also compiled for faster execution, and respect the \$force\_compile and \$compile\_check settings.

### Example 13.11. config\_load()

```
<?php
// load config variables and assign them
$smarty->config_load('my.conf');

// load a section
$smarty->config_load('my.conf', 'foobar');
?>
```

See also {config\_load}, get\_config\_vars(), clear\_config(), and config variables

## display()

display() displays the template

display()

## 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 13.12. display()

```
<?php
include("Smarty.class.php");
$smarty = new Smarty;
$smarty->caching = true;

// only do db calls if cache doesn't exist
if(!$smarty->is_cached("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);
}

// display the output
$smarty->display("index.tpl");
?>
```

Use the syntax for template resources to display files outside of the \$template\_dir directory.

### Example 13.13. function display() template resource examples

```
<?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 `template_exists()`.

## fetch()

fetch() returns the template output

fetch()

## 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 13.14. fetch()

```

<?php
include('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = true;

// only do db calls if cache doesn't exist
if(!$smarty->is_cached('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 13.15. 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}/index.php?page=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 have
violated that, then you hereby owe the sender 10 GBP for each even
numbered word you have read
{/textformat}
```

and 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_id;
$contact = $db->getRow($sql);
$smarty->assign('contact', $contact);

mail($contact['email'], 'Subject', $smarty->fetch('email_body.tpl'));

?>
```

See also {fetch} display(), {eval}, and template\_exists().

## get\_config\_vars()

get\_config\_vars() returns the given loaded config variable value

get\_config\_vars()

## Description

array **get\_config\_vars** ([string varname])

If no parameter is given, an array of all loaded config variables is returned.

### Example 13.16. get\_config\_vars()

```
<?php
// get loaded config template var 'foo'
$foo = $smarty->get_config_vars('foo');

// get all loaded config template vars
$config_vars = $smarty->get_config_vars();

// take a look at them
print_r($config_vars);
?>
```

See also `clear_config()`, `{config_load}`, `config_load()` and `get_template_vars()`.

## get\_registered\_object()

get\_registered\_object() returns a reference to a registered object

get\_registered\_object()

## Description

array **get\_registered\_object** (string object\_name)

This is useful from within a custom function when you need direct access to a registered object. See objects for more info.

### Example 13.17. get\_registered\_object()

```
<?php
function smarty_block_foo($params, &$smarty)
{
    if (isset($params['object'])) {
        // get reference to registered object
        $obj_ref = &$smarty->get_registered_object($params['object']);
        // use $obj_ref is now a reference to the object
    }
}
?>
```

See also register\_object(), unregister\_object() and objects section

## get\_template\_vars()

get\_template\_vars() returns assigned variable value(s)

get\_template\_vars()

## Description

array **get\_template\_vars** ([string varname])

If no parameter is given, an array of all assigned variables are returned.

### Example 13.18. get\_template\_vars

```
<?php
// get assigned template var 'foo'
$foo = $smarty->get_template_vars('foo');

// get all assigned template vars
$tpl_vars = $smarty->get_template_vars();

// take a look at them
print_r($tpl_vars);
?>
```

See also [assign\(\)](#), [{assign}](#), [assign\\_by\\_ref\(\)](#), [append\(\)](#), [clear\\_assign\(\)](#), [clear\\_all\\_assign\(\)](#) and [get\\_config\\_vars\(\)](#)



## is\_cached()

is\_cached() returns true if there is a valid cache for this template

is\_cached()

## Description

bool **is\_cached** (string template [, string cache\_id [, string compile\_id]])

This only works if \$caching is set to true. See also the caching section.

### Example 13.19. is\_cached()

```
<?php
$smarty->caching = true;

if(!$smarty->is_cached('index.tpl')) {
    // do database calls, assign vars here
}

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

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 you do not want to pass a cache id but want to pass a \$compile\_id you have to pass *null* as cache id.

### Example 13.20. is\_cached() with multiple-cache template

```
<?php
$smarty->caching = true;

if(!$smarty->is_cached('index.tpl', 'FrontPage')) {
    // do database calls, assign vars here
}

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

**Technical Note:** If `is_cached()` 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 `is_cached()` and to `display()` in the example above. This also means calls to `clear_cache()` and other changes of the cache-settings may have no effect after `is_cached()` returned true.

See also `clear_cache()`, `clear_all_cache()`, and caching section.

## load\_filter()

load\_filter()load a filter plugin

load\_filter()

## Description

void **load\_filter** (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, for example, 'trim'.

### Example 13.21. loading filter plugins

```
<?php
// load prefilter named 'trim'
$smarty->load_filter('pre', 'trim');

// load another prefilter named 'datefooter'
$smarty->load_filter('pre', 'datefooter');

// load output filter named 'compress'
$smarty->load_filter('output', 'compress');

?>
```

See also register\_prefilter(), register\_postfilter(), register\_outputfilter(), \$autoload\_filters and Advanced features.

## register\_block()

register\_block() dynamically register block functions plugins

register\_block()

## Description

void **register\_block** (string name, mixed impl, bool cacheable, mixed cache\_attrs)

Use this to dynamically register block functions plugins. Pass in the block function name, followed by the PHP function callback that implements it.

The php-function callback *impl* can be either (a) a string containing the function name or (b) an array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name or (c) an array of the form `array(&$class, $method)` with `$class` being a classname and `$method` being a class method of that class.

*cacheable* and *cache\_attrs* can be omitted in most cases. See [Controlling Cacheability of Plugins' Output](#) on how to use them properly.

### Example 13.22. register\_block()

```
<?php
$smarty->register_block('translate', 'do_translation');

function do_translation ($params, $content, &$smarty, &$repeat)
{
    if (isset($content)) {
        $lang = $params['lang'];
        // do some translation with $content
        return $translation;
    }
}
?>
```

where the template is:

```
{* template *}
{translate lang="br"}
Hello, world!
{/translate}
```

See also `unregister_block()` and [Plugin Block Functions](#).

## register\_compiler\_function()

register\_compiler\_function() dynamically register a compiler function plugin

register\_compiler\_function()

### Description

bool **register\_compiler\_function** (string name, mixed impl, bool cacheable)

Pass in the compiler function name, followed by the PHP function that implements it.

The php-function callback *impl* can be either:

- a. a string containing the function name
- b. an array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name
- c. an array of the form `array(&$class, $method)` with `$class` being a classname and `$method` being a class method of that class.

*cacheable* can be omitted in most cases. See Controlling Cacheability of Plugins' Output on how to use it properly.

See also `unregister_compiler_function()` and Plugin Compiler Functions.

## register\_function()

register\_function() dynamically register template function plugins

register\_function()

## Description

void **register\_function** (string name, mixed impl [, bool cacheable [, mixed cache\_attrs]])

Pass in the template function name, followed by the PHP function name that implements it.

The php-function callback *impl* 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 a classname and `$method` being a class method of that class.

*cacheable* and *cache\_attrs* can be omitted in most cases. See [Controlling Cacheability of Plugins' Output](#) on how to use them properly.

### Example 13.23. register\_function()

```
<?php
$smarty->register_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"}
```

See also `unregister_function()` and [Plugin functions](#) .

## register\_modifier()

register\_modifier() dynamically register modifier plugin

register\_modifier()

## Description

void **register\_modifier** (string name, mixed impl)

Pass in the template modifier name, followed by the PHP function that it implements it.

The php-function callback *impl* can be either

- a. a string containing the function name
- b. an array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name
- c. an array of the form `array(&$class, $method)` with `$class` being a classname and `$method` being a class method of that class.

### Example 13.24. register\_modifier()

```
<?php
// let's map PHP's stripslashes function to a Smarty modifier.
$smarty->register_modifier('slash', 'stripslashes');
?>
```

template

```
<?php
{* use 'slash' to strip slashes from variables *}
{$var|slash}
?>
```

See also `unregister_modifier()`, `register_function()`, `modifiers`, `Extending Smarty with plugins` and `Creating Plugin modifiers`,

## register\_object()

register\_object()register an object for use in the templates

register\_object()

## Description

void **register\_object** (string object\_name, object object, array allowed\_methods\_properties, boolean format, array block\_methods)

See the objects section for examples.

See also get\_registered\_object(), and unregister\_object().

## register\_outputfilter()

register\_outputfilter() dynamically register outputfilters

register\_outputfilter()

## Description

void **register\_outputfilter** (mixed function)

Use this to dynamically register outputfilters to operate 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.

The php-function callback *function* can be either

- a. a string containing the function name
- b. an array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name
- c. an array of the form `array(&$class, $method)` with `$class` being a classname and `$method` being a class method of that class.

See also `unregister_outputfilter()`, `register_prefilter()`, `register_postfilter()`, `load_filter()`, `$autoload_filters` and template output filters.



## register\_postfilter()

register\_postfilter() dynamically register postfilters

register\_postfilter()

## Description

void **register\_postfilter** (mixed function)

Use this to dynamically register postfilters to run templates through after they are compiled. See [template postfilters](#) for more information on how to setup a postfiltering function.

The php-function callback *function* can be either

- a. a string containing the function name
- b. an array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name
- c. an array of the form `array(&$class, $method)` with `$class` being a classname and `$method` being a class method of that class.

See also [unregister\\_postfilter\(\)](#), [register\\_prefilter\(\)](#), [register\\_outputfilter\(\)](#), [load\\_filter\(\)](#), [\\$autoload\\_filters](#) and [template output filters](#).

## register\_prefilter()

register\_prefilter() dynamically register prefilters

register\_prefilter()

## Description

void **register\_prefilter** (mixed function)

Use this to dynamically register prefilters to run templates through before they are compiled. See template prefilters for more information on how to setup a prefiltering function.

The php-function callback *function* can be either

- a. a string containing the function name
- b. an array of the form `array(&$object, $method)` with `&$object` being a reference to an object and `$method` being a string containing the method-name
- c. an array of the form `array(&$class, $method)` with `$class` being a classname and `$method` being a class method of that class.

See also `unregister_prefilter()`, `register_postfilter()`, `register_outputfilter()`, `load_filter()`, `$autoload_filters` and template output filters.

## register\_resource()

register\_resource() dynamically register resources

register\_resource()

## Description

void **register\_resource** (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 13.25. register\_resource()

```
<?php
$smarty->register_resource('db', array(
    'db_get_template',
    'db_get_timestamp',
    'db_get_secure',
    'db_get_trusted' )
);
?>
```

See also [unregister\\_resource\(\)](#) and [template resources](#)

## trigger\_error()

trigger\_error() output an error message

trigger\_error()

## Description

void **trigger\_error** (string error\_msg [, int level])

This function can be used to output an error message using Smarty. *level* parameter can be one of the values used for the PHP `trigger_error()` [[http://php.net/trigger\\_error](http://php.net/trigger_error)] function, i.e. `E_USER_NOTICE`, `E_USER_WARNING`, etc. By default it's `E_USER_WARNING`.

See also `$error_reporting`, `debugging` and `Troubleshooting`.

## template\_exists()

template\_exists() checks whether the specified template exists

template\_exists()

## Description

bool **template\_exists** (string template)

It can accept either a path to the template on the filesystem or a resource string specifying the template.

### Example 13.26. template\_exists()

This example uses \$\_GET['page'] to include a content template. If the template doesn't exist then an error page is displayed instead.

The page\_container.tpl

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

{* include middle content page *}
{include file=$page_mid}

{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( !$smarty->template_exists($mid_template) ){
    $mid_template = 'page_not_found.inc.tpl';
}
$smarty->assign('page_mid', $mid_template);

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

See also display(), fetch(), {include} and {insert}

## unregister\_block()

unregister\_block() dynamically unregister block function plugins

unregister\_block()

### Description

void **unregister\_block** (string name)

Use this to dynamically unregister block function plugins. Pass in the block function *name*.

See also register\_block() and Block Functions Plugins.

## **unregister\_compiler\_function()**

unregister\_compiler\_function() dynamically unregister a compiler function

unregister\_compiler\_function()

### **Description**

void **unregister\_compiler\_function** (string name)

Pass in the *name* of the compiler function.

See also register\_compiler\_function() and Plugin Compiler Functions.

## unregister\_function

unregister\_function dynamically unregister template function plugin

unregister\_function

### Description

void **unregister\_function** (string name)

Pass in the template function name.

#### Example 13.27. unregister\_function

```
<?php
// we don't want template designers to have access to system files
$smarty->unregister_function('fetch');
?>
```

See also [register\\_function\(\)](#) .



## unregister\_modifier()

unregister\_modifier() dynamically unregister modifier plugin

unregister\_modifier()

### Description

void **unregister\_modifier** (string name)

Pass in the template modifier name.

#### Example 13.28. unregister\_modifier()

```
<?php
// we don't want template designers to strip tags from elements
$smarty->unregister_modifier('strip_tags');
?>
```

See also register\_modifier() and Plugin modifiers,

## **unregister\_object()**

unregister\_object() dynamically unregister an object

unregister\_object()

### **Description**

void **unregister\_object** (string object\_name)

See also register\_object() and objects section

## **unregister\_outputfilter()**

unregister\_outputfilter() dynamically unregister an output filter

unregister\_outputfilter()

### **Description**

void **unregister\_outputfilter** (string function\_name)

Use this to dynamically unregister an output filter.

See also register\_outputfilter() and template output filters.

## **unregister\_postfilter()**

unregister\_postfilter() dynamically unregister a postfilter

unregister\_postfilter()

### **Description**

void **unregister\_postfilter** (string function\_name)

See also register\_postfilter() and template post filters.

## **unregister\_prefilter()**

unregister\_prefilter() dynamically unregister a prefilter

unregister\_prefilter()

### **Description**

void **unregister\_prefilter** (string function\_name)

See also register\_prefilter() and pre filters.

## unregister\_resource()

unregister\_resource() dynamically unregister a resource plugin

unregister\_resource()

### Description

void **unregister\_resource** (string name)

Pass in the name of the resource.

#### Example 13.29. unregister\_resource()

```
<?php
$smarty->unregister_resource('db');
?>
```

See also register\_resource() and template resources

---

# Chapter 14. Caching

## Table of Contents

Setting Up Caching .....	145
Multiple Caches Per Page .....	147
Cache Groups .....	148
Controlling Cacheability of Plugins' Output .....	149

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 weather map 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 = true;

$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`. Upon 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" extension, they are not really executable php scripts. Do not edit these files!

Each cached page has a limited lifetime determined by `$cache_lifetime`. The default value is 3600 seconds, or 1 hour. After that time expires, the cache is regenerated. It is possible to give individual caches their own expiration time by setting `$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 = 2; // 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 `$compile_check` is enabled, every template file and config file that is involved with the cache file is checked for modification. If any of the files have been modified since the cache was generated, the cache is immediately regenerated. This is a slight overhead so for optimum performance, leave `$compile_check` set to false.

### Example 14.3. enabling `$compile_check`

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = true;
$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 = false` (or 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 = true;

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 right side of the page. By using an {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 (or groups) with the `clear_cache()` function.

### Example 14.5. clearing the cache

```
<?php
require('Smarty.class.php');
$smarty = new Smarty;

$smarty->caching = true;

// clear out all cache files
$smarty->clear_all_cache();

// clear only cache for index.tpl
$smarty->clear_cache('index.tpl');

$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 = true;

$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_ids` 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 10 characters and is made up of alpha-numeric 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 = true;

$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 = true;

// 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 = true;

// 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

Since Smarty-2.6.0 plugins the cacheability of plugins can be declared when registering them. The third parameter to `register_block()`, `register_compiler_function()` and `register_function()` is called *\$cacheable* and defaults to true which is also the behaviour of plugins in Smarty versions before 2.6.0

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
require('Smarty.class.php');
$smarty = new Smarty;
$smarty->caching = true;

function remaining_seconds($params, &$smarty) {
    $remain = $params['endtime'] - time();
    if ($remain >=0)
        return $remain . " second(s)";
    else
        return "done";
}

$smarty->register_function('remaining', 'remaining_seconds', false, array('endtime'));

if (!$smarty->is_cached('index.tpl')) {
    // fetch $obj from db and assign...
    $smarty->assign_by_ref('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
require('Smarty.class.php');
$smarty = new Smarty;
$smarty->caching = true;

function smarty_block_dynamic($param, $content, &$smarty) {
    return $content;
}
$smarty->register_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

## Table of Contents

Objects .....	151
Prefilters .....	152
Postfilters .....	152
Output Filters .....	153
Cache Handler Function .....	153
Resources .....	155

## Objects

Smarty allows access to PHP objects [<http://php.net/object>] through the templates. 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 (beginning with "\_"). 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 4 parameters *\$params*, *\$content*, *&\$smarty* and *&\$repeat* and they also behave like block-function-plugins.

### Example 15.1. 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 false
$smarty->register_object('foobar',$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()`

## 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 `load_filter()` 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 15.2. using a template prefilter

This will remove all the 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->register_prefilter('remove_dw_comments');
$smarty->display('index.tpl');
?>
```

See also `register_prefilter()`, `postfilters` and `load_filter()`

## 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 `load_filter()` function or by setting `$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 15.3. 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->register_postfilter('add_header_comment');
$smarty->display('index.tpl');
?>
```

This will make the compiled Smarty template index.tpl look like:

```
<!-- Created by Smarty! -->
{* rest of template content... *}
```

See also `register_postfilter()`, `prefilters` and `load_filter()`

## 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, and 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 `load_filter()` function or by setting `$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 15.4. 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->register_outputfilter("protect_email");
$smarty->display("index.tpl");

// now any occurrence of an email address in the template output will have
// a simple protection against spambots
?>
```

## 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 `cache_id` (optional), and the sixth is the `compile_id` (optional).

Note: The last parameter (`$exp_time`) was added in Smarty-2.6.0.

### Example 15.5. example using MySQL as a cache source

```
<?php
/*
example usage:

include('Smarty.class.php');
include('mysql_cache_handler.php');

$smarty = new Smarty;
$smarty->cache_handler_func = 'mysql_cache_handler';

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

mysql database is expected in this format:

create database SMARTY_CACHE;

create table CACHE_PAGES(
CacheID char(32) PRIMARY KEY,
CacheContents MEDIUMTEXT NOT NULL
);
*/

function mysql_cache_handler($action, &$smarty_obj, &$cache_content, $tpl_file=null, $cache_id=null, $C
{
    // set db host, user and pass here
    $db_host = 'localhost';
    $db_user = 'myuser';
    $db_pass = 'mypass';
    $db_name = 'SMARTY_CACHE';
    $use_gzip = false;

    // create unique cache id
    $CacheID = md5($tpl_file.$cache_id.$compile_id);

    if(! $link = mysql_pconnect($db_host, $db_user, $db_pass)) {
        $smarty_obj->trigger_error_msg("cache_handler: could not connect to database");
        return false;
    }
    mysql_select_db($db_name);

    switch ($action) {
        case 'read':
            // read cache from database
            $results = mysql_query("select CacheContents from CACHE_PAGES where CacheID='".$C
            if(!$results) {
                $smarty_obj->trigger_error_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->_trigger_error_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->_trigger_error_msg("cache_handler: query failed.");
    }
    $return = $results;
    break;
default:
    // error, unknown action
    $smarty_obj->_trigger_error_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 `$template_dir` do not require a template resource, although you can use the `file:` resource for consistency. Just supply the path to the template you want to use relative to the `$template_dir` root directory.

### Example 15.6. using templates from `$template_dir`

```
<?php
```

```

$smarty->display("index.tpl");
$smarty->display("admin/menu.tpl");
$smarty->display("file:admin/menu.tpl"); // same as one above
?>

{* from within Smarty template *}
{include file="index.tpl"}
{include file="file:index.tpl"} {* same as one above *}

```

## Templates from any directory

Templates outside of the `$template_dir` require the `file:` template resource type, followed by the absolute path and name of the template.

### Example 15.7. 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 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 15.8. 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 other sources

You can retrieve templates using whatever possible source you can access with PHP: databases, sockets, LDAP, and so on. 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 15.9. 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
    $sql = new SQL;
    $sql->query("select tpl_source
                from my_table
                where tpl_name='$tpl_name'");
    if ($sql->num_rows) {
        $tpl_source = $sql->record['tpl_source'];
        return true;
    } else {
        return false;
    }
}

function db_get_timestamp($tpl_name, &$tpl_timestamp, &$smarty_obj)
{
    // do database call here to populate $tpl_timestamp.
    $sql = new SQL;
    $sql->query("select tpl_timestamp
                from my_table
                where tpl_name='$tpl_name'");
    if ($sql->num_rows) {
        $tpl_timestamp = $sql->record['tpl_timestamp'];
        return true;
    } else {
        return false;
    }
}

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 re-

source. One use of this is to create templates that do not exist on-the-fly.

### Example 15.10. using the default template handler function

```
<?php
// put this function somewhere in your application

function make_template ($resource_type, $resource_name, &$amp;template_source, &$amp;template_timestamp,
&$smarty_obj)
{
    if( $resource_type == 'file' ) {
        if ( ! is_readable ( $resource_name )) {
            // create the template file, return contents.
            $template_source = "This is a new template.";
            $template_timestamp = time();
            $smarty_obj->_write_file($resource_name,$template_source);
            return true;
        }
    } else {
        // not a file
        return false;
    }
}

// set the default handler
$smarty->default_template_handler_func = 'make_template';
?>
```

---

# Chapter 16. Extending Smarty With Plugins

## Table of Contents

How Plugins Work .....	159
Naming Conventions .....	159
Writing Plugins .....	160
Template Functions .....	160
Modifiers .....	162
Block Functions .....	163
Compiler Functions .....	164
Prefilters/Postfilters .....	165
Output Filters .....	166
Resources .....	166
Inserts .....	168

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.

The 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).

Some examples: `function.html_select_date.php`, `resource.db.php`, `modifier.spacify.php`.

The plugin functions inside the plugin files must be named as follows:

```
smarty_type, _name()
```

The meanings of *type* and *name* are the same as before.

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 and 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 (**fetch()** function, for example). Alternatively, the function can simply perform some other task without any output (**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.

See also: `register_function()`, `unregister_function()`.

### 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->trigger_error("assign: missing 'var' parameter");
        return;
    }
}
```

```

    if (!in_array('value', array_keys($params))) {
        $smarty->trigger_error("assign: missing 'value' parameter");
        return;
    }

    $smarty->assign($params['var'], $params['value']);
}
?>

```

## 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 supposed to operate. The rest of the parameters can be optional, depending on what kind of operation is supposed to be performed.

The modifier has to return the result of its processing.

See also `register_modifier()`, `unregister_modifier()`.

### Example 16.3. 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;
}
?>

```

## 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 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 `$params` as an associative array. You can access those values as e.g. `$params['start']`. The opening tag attributes are also accessible to your function when processing the closing tag.

The value of `$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 block 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()` on it and the structure should be apparent.

See also: `register_block()`, `unregister_block()`.

### 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, &$repeat)
{
    if (isset($content)) {
        $lang = $params['lang'];
        // do some intelligent translation thing here with $content
        return $translation;
    }
}
?>

```

## 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.

See also `register_compiler_function()`, `unregister_compiler_function()`.

### Example 16.6. 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';

```

?>

## 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->get_template_vars()); ?>\n</pre>" . $compiled;
    return $compiled;
}
?>
```

?>

## 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. 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);
}
?>
```

## 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 4 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 is supposed to retrieve the resource. Its second parameter is a variable passed by reference where the res-

ult 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 is supposed to retrieve the last modification time of the requested resource (as a UNIX timestamp). The second parameter is a variable passed by reference where the timestamp should be stored. The function is supposed to return `true` if the timestamp could be successfully determined, and `false` otherwise.

The third function 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 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 `src` attribute. However, it should still be defined even for template resources.

See also `register_resource()`, `unregister_resource()`.

### 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
    $sql = new SQL;
    $sql->query("select tpl_source
                from my_table
                where tpl_name='$tpl_name'");
    if ($sql->num_rows) {
        $tpl_source = $sql->record['tpl_source'];
        return true;
    } else {
        return false;
    }
}

function smarty_resource_db_timestamp($tpl_name, &$tpl_timestamp, &$smarty)
{
    // do database call here to populate $tpl_timestamp.
    $sql = new SQL;
    $sql->query("select tpl_timestamp
                from my_table
                where tpl_name='$tpl_name'");
    if ($sql->num_rows) {
        $tpl_timestamp = $sql->record['tpl_timestamp'];
        return true;
    } else {
        return false;
    }
}

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
}
?>
```

## 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->trigger_error("insert time: missing 'format' parameter");
        return;
    }

    $datetime = strftime($params['format']);
    return $datetime;
}
?>
```

---

# Part IV. Appendixes

## Table of Contents

17. Troubleshooting .....	170
Smarty/PHP errors .....	170
18. Tips & Tricks .....	172
Blank Variable Handling .....	172
Default Variable Handling .....	172
Passing variable title to header template .....	173
Dates .....	173
WAP/WML .....	174
Componentized Templates .....	175
Obfuscating E-mail Addresses .....	176
19. Resources .....	177
20. BUGS .....	178

---

# Chapter 17. Troubleshooting

## Table of Contents

Smarty/PHP errors ..... 170

## 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 occurred.

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 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, `$error_reporting` and `trigger_error()`.

---

# Chapter 18. Tips & Tricks

## Table of Contents

|                                                 |     |
|-------------------------------------------------|-----|
| Blank Variable Handling .....                   | 172 |
| Default Variable Handling .....                 | 172 |
| Passing variable title to header template ..... | 173 |
| Dates .....                                     | 173 |
| WAP/WML .....                                   | 174 |
| Componentized Templates .....                   | 175 |
| Obfuscating E-mail Addresses .....              | 176 |

## 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 table backgrounds work properly. Many would use an `{if}` statement to handle this, but there is a short-hand way with Smarty, using the *default* variable modifier.

### 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 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 $title was empty, it now contains the value "no title" when you print it *}
{$title}
```

See also default 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 when it is included.

### Example 18.3. Passing the title variable to the header template

mainpage.tpl

```
{include file="header.tpl" title="Main Page"}
{* template body goes here *}
{include file="footer.tpl"}
```

archives.tpl

```
{config_load file="archive_page.conf"}
{include file="header.tpl" title=#archivePageTitle#}
{* template body goes here *}
{include file="footer.tpl"}
```

header.tpl

```
<html>
<head>
<title>{$title|default:"BC News"}</title>
</head>
<body>
```

footer.tpl

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

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. When the archives page is drawn, the title will be "Archives". Notice in the archive example, we are using a variable from the archive\_page.conf file instead of a hard coded variable. Also notice that "BC News" is printed if the \$title variable is not set, using the default variable modifier.

## Dates

As a rule of thumb, always pass dates to Smarty as timestamps. This allows template designers to use date\_format for full control over date formatting, and also makes it easy to compare dates if necessary.

**Note:** As of Smarty 1.4.0, you can pass dates to Smarty as unix timestamps, mysql timestamps, or any date parsable by strtotime() [<http://php.net/strtotime>].

### Example 18.4. using date\_format

```
{$startDate|date_format}
```

This will output:

```
Jan 4, 2001
```

```
{startDate|date_format:"%Y/%m/%d"}
```

This will output:

```
2001/01/04
```

```
{if $date1 < $date2}
  ...
{/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 easiest 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/wml_1.1.xml">
<!-- 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="YHOO" 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 E-mail address gets on so many spam mailing lists? One way spammers collect E-mail addresses is from web pages. To help combat this problem, you can make your E-mail address show up in scrambled javascript in the HTML source, yet it will look and work correctly in the browser. This is done with the {mailto} plugin.

### Example 18.8. Example of Obfuscating an E-mail Address

```
{* in index.tpl *}
Send inquiries to
{mailto address=$EmailAddress encode="javascript" subject="Hello"}
```

**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.

See also escape and {mailto}.

---

# Chapter 19. Resources

Smarty's homepage is located at <http://smarty.php.net/>. You can join the mailing list by sending an e-mail to [smarty-general-subscribe@lists.php.net](mailto:smarty-general-subscribe@lists.php.net). An archive of the mailing list can be viewed at <http://marc.theaimsgroup.com/?l=smarty-general&r=1&w=2>.

---

# Chapter 20. BUGS

Check the `BUGS` file that comes with the latest distribution of Smarty, or check the website.