NAME

CGI::HTML::Functions - Documentation for CGI.pm Legacy HTML Functionality

SYNOPIS

Nothing here - please do not use this functionality, it is considered to be legacy and essentially deprecated. This documentation exists solely to aid in maintenance and migration of legacy code using this functionality.

This functionality is likely to be removed in future versions of CGI.pm so you are strongly encouraged to migrate away from it. If you are working on new code you should be using a template engine. For more information see CGI::Alternatives.

DESCRIPTION

The documentation here should be considered an addendum to the sections in the CGI documentation - the sections here are named the same as those within the CGI perldoc.

Calling CGI.pm routines

HTML tag functions have both attributes (the attribute=''value'' pairs within the tag itself) and contents (the part between the opening and closing pairs). To distinguish between attributes and contents, CGI.pm uses the convention of passing HTML attributes as a hash reference as the first argument, and the contents, if any, as any subsequent arguments. It works out like this:

```
Code Generated HTML
---- --------------
h1() <h1 />
h1('some','contents'); <h1>some contents</h1>
h1({-align=>left}); <h1 align="LEFT">
h1({-align=>left},'contents'); <h1 align="LEFT">contents</h1>
```

Many newcomers to CGI.pm are puzzled by the difference between the calling conventions for the HTML shortcuts, which require curly braces around the HTML tag attributes, and the calling conventions for other routines, which manage to generate attributes without the curly brackets. Don't be confused. As a convenience the curly braces are optional in all but the HTML shortcuts. If you like, you can use curly braces when calling any routine that takes named arguments. For example:

```
print $q->header( { -type => 'image/gif', -expires => '+3d' } );
```

If you use warnings, you will be warned that some CGI.pm argument names conflict with built-in perl functions. The most frequent of these is the -values argument, used to create multi-valued menus, radio button clusters and the like. To get around this warning, you have several choices:

1. Use another name for the argument, if one is available. For example, -value is an alias for -values.
2. Change the capitalization, e.g. -Values
3. Put quotes around the argument name, e.g. '-values'

Function-oriented interface HTML exports

Here is a list of the HTML related function sets you can import:

:form

Import all fill-out form generating methods, such as ***textfield()***.

:html2

Import all methods that generate HTML 2.0 standard elements.

:html3

Import all methods that generate HTML 3.0 elements (such as <table>, <super> and <sub>).

:html4

Import all methods that generate HTML 4 elements (such as <abbrev>, <acronym> and <thead>).

:netscape

Import the <blink>, <fontsize> and tags.

:html
Import all HTML-generating shortcuts (i.e. 'html2', 'html3', 'html4' and 'netscape')

:standard
Import ''standard'' features, 'html2', 'html3', 'html4', 'ssl', 'form' and 'cgi'.

If you import any of the state-maintaining CGI or form-generating methods, a default CGI object will be created and initialized automatically the first time you use any of the methods that require one to be present. This includes ***param()***, ***textfield()***, ***submit()*** and the like. (If you need direct access to the CGI object, you can find it in the global variable **`$CGI::Q`**).

Pragmas

Additional HTML generation related pragms:

-nosticky
By default the CGI module implements a state-preserving behavior called ''sticky'' fields. The way this works is that if you are regenerating a form, the methods that generate the form field values will interrogate *param()* to see if similarly-named parameters are present in the query string. If they find a like-named parameter, they will use it to set their default values.

Sometimes this isn't what you want. The **-nosticky** pragma prevents this behavior. You can also selectively change the sticky behavior in each element that you generate.

-tabindex
Automatically add tab index attributes to each form field. With this option turned off, you can still add tab indexes manually by passing a -tabindex option to each field-generating method.

-no_xhtml
By default, CGI.pm versions 2.69 and higher emit XHTML (http://www.w3.org/TR/xhtml1/). The -no_xhtml pragma disables this feature. Thanks to Michalis Kabrianis <kabrianis@hellug.gr> for this feature.

If *start_html()*'s -dtd parameter specifies an HTML 2.0, 3.2, 4.0 or 4.01 DTD, XHTML will automatically be disabled without needing to use this pragma.

Special forms for importing HTML-tag functions

Many of the methods generate HTML tags. As described below, tag functions automatically generate both the opening and closing tags. For example:

```
print h1('Level 1 Header');
```

produces

```
<h1>Level 1 Header</h1>
```

There will be some times when you want to produce the start and end tags yourself. In this case, you can use the form start_*tag_name* and end_*tag_name*, as in:

```
print start_h1,'Level 1 Header',end_h1;
```

Creating the HTML document header

```
print start_html(
-title => 'Secrets of the Pyramids',
-author => 'fred@capricorn.org',
-base => 'true',
-target => '_blank',
-meta => {'keywords'=>'pharaoh secret mummy',
'copyright' => 'copyright 1996 King Tut'},
-style => {'src'=>'/styles/style1.css'},
-BGCOLOR => 'blue'
);
```

The *start_html()* routine creates the top of the page, along with a lot of optional information that controls the page's appearance and behavior.

This method returns a canned HTML header and the opening <body> tag. All parameters are optional. In the named parameter form, recognized parameters are -title, -author, -base, -xbase, -dtd, -lang and -target (see below for the explanation). Any additional parameters you provide, such as the unofficial BGCOLOR attribute, are added to the <body> tag. Additional parameters must be proceeded by a hyphen.

The argument **-xbase** allows you to provide an HREF for the <base> tag different from the current location, as in

```
-xbase => http://home.mcom.com/ -P "
```

All relative links will be interpreted relative to this tag.

The argument **-target** allows you to provide a default target frame for all the links and fill-out forms on the page. **This is a non-standard HTTP feature which only works with some browsers!**

```
-target => "answer_window"
```

All relative links will be interpreted relative to this tag. You add arbitrary meta information to the header with the **-meta** argument. This argument expects a reference to a hash containing name/value pairs of meta information. These will be turned into a series of header <meta> tags that look something like this:

```
<meta name="keywords" content="pharaoh secret mummy">
<meta name="description" content="copyright 1996 King Tut">
```

To create an HTTP-EQUIV type of <meta> tag, use **-head**, described below.

The **-style** argument is used to incorporate cascading stylesheets into your code. See the section on CASCADING STYLESHEETS for more information.

The **-lang** argument is used to incorporate a language attribute into the <html> tag. For example:

```
print $q->start_html( -lang => 'fr-CA' );
```

The default if not specified is ‘‘en-US’’ for US English, unless the -dtd parameter specifies an HTML 2.0 or 3.2 DTD, in which case the lang attribute is left off. You can force the lang attribute to left off in other cases by passing an empty string (-lang=>’’).

The **-encoding** argument can be used to specify the character set for XHTML. It defaults to iso-8859-1 if not specified.

The **-dtd** argument can be used to specify a public DTD identifier string. For example:

```
-dtd => '-//W3C//DTD HTML 4.01 Transitional//EN')
```

Alternatively, it can take public and system DTD identifiers as an array:

```
-dtd => [
'-//W3C//DTD HTML 4.01 Transitional//EN',
       'http://www.w3.org/TR/html4/loose.dtd'
]
```

For the public DTD identifier to be considered, it must be valid. Otherwise it will be replaced by the default DTD. If the public DTD contains ’XHTML’, CGI.pm will emit XML.

The **-declare_xml** argument, when used in conjunction with XHTML, will put a <?xml> declaration at the top of the HTML header. The sole purpose of this declaration is to declare the character set encoding. In the absence of -declare_xml, the output HTML will contain a <meta> tag that specifies the encoding, allowing the HTML to pass most validators. The default for -declare_xml is false.

You can place other arbitrary HTML elements to the <head> section with the **-head** tag. For example, to place a <link> element in the head section, use this:

```
print start_html(
-head => Link({
-rel => 'shortcut icon',
-href => 'favicon.ico'
})
);
```

To incorporate multiple HTML elements into the <head> section, just pass an array reference:

```
print start_html(
-head => [
Link({
-rel => 'next',
                -href => 'http://www.capricorn.com/s2.html'
}),
Link({
-rel => 'previous',
                -href => 'http://www.capricorn.com/s1.html'
})
]
);
```

And here's how to create an HTTP-EQUIV <meta> tag:

```
print start_html(
-head => meta({
-http_equiv => 'Content-Type',
-content => 'text/html'
})
);
```

JAVASCRIPTING: The **-script**, **-noScript**, **-onLoad**, **-onMouseOver**, **-onMouseOut** and **-onUnload** parameters are used to add JavaScript calls to your pages. **-script** should point to a block of text containing JavaScript function definitions. This block will be placed within a <script> block inside the HTML (not HTTP) header. The block is placed in the header in order to give your page a fighting chance of having all its JavaScript functions in place even if the user presses the stop button before the page has loaded completely. CGI.pm attempts to format the script in such a way that JavaScript-naive browsers will not choke on the code: unfortunately there are some browsers that get confused by it nevertheless.

The **-onLoad** and **-onUnload** parameters point to fragments of JavaScript code to execute when the page is respectively opened and closed by the browser. Usually these parameters are calls to functions defined in the **-script** field:

```
$query = CGI->new;
print header;
$JSCRIPT = <<END;
// Ask a silly question
function riddle_me_this() {
var r = prompt(
"What walks on four legs in the morning, " +
"two legs in the afternoon, " +
"and three legs in the evening?"
);
response(r);
}
// Get a silly answer
function response(answer) {
if (answer == "man")
alert("Right you are!");
```

```
else
alert("Wrong! Guess again.");
}
END
print start_html(
-title => 'The Riddle of the Sphinx',
-script => $JSCRIPT
);
```

Use the **-noScript** parameter to pass some HTML text that will be displayed on browsers that do not have JavaScript (or browsers where JavaScript is turned off).

The <script> tag, has several attributes including ‘‘type’’, ‘‘charset’’ and ‘‘src’’. ‘‘src’’ allows you to keep JavaScript code in an external file. To use these attributes pass a HASH reference in the **-script** parameter containing one or more of -type, -src, or -code:

```
print $q->start_html(
-title => 'The Riddle of the Sphinx',
-script => {
-type => 'JAVASCRIPT',
-src => '/javascript/sphinx.js'}
);

print $q->(
-title => 'The Riddle of the Sphinx',
-script => {
-type => 'PERLSCRIPT',
-code => 'print "hello world!\n;"'
}
);
```

A final feature allows you to incorporate multiple <script> sections into the header. Just pass the list of script sections as an array reference. This allows you to specify different source files for different dialects of JavaScript. Example:

```
print $q->start_html(
-title => 'The Riddle of the Sphinx',
-script => [
{
-type => 'text/javascript',
-src => '/javascript/utilities10.js'
},
{
-type => 'text/javascript',
-src => '/javascript/utilities11.js'
},
{
-type => 'text/jscript',
-src => '/javascript/utilities12.js'
},
{
-type => 'text/ecmascript',
-src => '/javascript/utilities219.js'
}
]
);
```

The option ‘‘-language’’ is a synonym for -type, and is supported for backwards compatibility.

The old-style positional parameters are as follows:

Parameters:

1. The title
2. The author's e-mail address (will create a <link rev="MADE"> tag if present
3. A 'true' flag if you want to include a <base> tag in the header. This helps resolve relative addresses to absolute ones when the document is moved, but makes the document hierarchy non-portable. Use with care!

Other parameters you want to include in the <body> tag may be appended to these. This is a good place to put HTML extensions, such as colors and wallpaper patterns.

Ending the Html document:

```
print $q->end_html;
```

This ends an HTML document by printing the </body></html> tags.

CREATING STANDARD HTML ELEMENTS:

CGI.pm defines general HTML shortcut methods for many HTML tags. HTML shortcuts are named after a single HTML element and return a fragment of HTML text. Example:

```
print $q->blockquote(
"Many years ago on the island of",
                    },"Crete")," -P $q->a({href=>" -- http://crete.org/
"there lived a Minotaur named",
$q->strong("Fred."),
),
$q->hr;
```

This results in the following HTML code (extra newlines have been added for readability):

```
<blockquote>
Many years ago on the island of
  <a >Crete</a>" -P href=" -- http://crete.org/
there lived
a minotaur named <strong>Fred.</strong>
</blockquote>
<hr>
```

If you find the syntax for calling the HTML shortcuts awkward, you can import them into your namespace and dispense with the object syntax completely (see the next section for more details):

```
use CGI ':standard';
print blockquote(
"Many years ago on the island of",
     },"Crete")," -P a({href=>" -- http://crete.org/
"there lived a minotaur named",
strong("Fred."),
),
hr;
```

Providing arguments to HTML shortcuts

The HTML methods will accept zero, one or multiple arguments. If you provide no arguments, you get a single tag:

```
print hr; # <hr>
```

If you provide one or more string arguments, they are concatenated together with spaces and placed between opening and closing tags:

```
print h1("Chapter","1"); # <h1>Chapter 1</h1>"
```

If the first argument is a hash reference, then the keys and values of the hash become the HTML tag's attributes:

```
print a({-href=>'fred.html',-target=>'_new'},
"Open a new frame");

<a href="fred.html",target="_new">Open a new frame</a>
```

You may dispense with the dashes in front of the attribute names if you prefer:

```
print img {src=>'fred.gif',align=>'LEFT'};

<img align="LEFT" src="fred.gif">
```

Sometimes an HTML tag attribute has no argument. For example, ordered lists can be marked as COMPACT. The syntax for this is an argument that that points to an undef string:

```
print ol({compact=>undef},li('one'),li('two'),li('three'));
```

Prior to CGI.pm version 2.41, providing an empty ('') string as an attribute argument was the same as providing undef. However, this has changed in order to accommodate those who want to create tags of the form <img alt="">. The difference is shown in these two pieces of code:

```
CODE RESULT
img({alt=>undef}) <img alt>
img({alt=>''}) <img alt="">
```

The distributive property of HTML shortcuts

One of the cool features of the HTML shortcuts is that they are distributive. If you give them an argument consisting of a **reference** to a list, the tag will be distributed across each element of the list. For example, here's one way to make an ordered list:

```
print ul(
li({-type=>'disc'},['Sneezy','Doc','Sleepy','Happy'])
);
```

This example will result in HTML output that looks like this:

```
<ul>
<li type="disc">Sneezy</li>
<li type="disc">Doc</li>
<li type="disc">Sleepy</li>
<li type="disc">Happy</li>
</ul>
```

This is extremely useful for creating tables. For example:

```
print table({-border=>undef},
caption('When Should You Eat Your Vegetables?'),
Tr({-align=>'CENTER',-valign=>'TOP'},
[
th(['Vegetable', 'Breakfast','Lunch','Dinner']),
td(['Tomatoes' , 'no', 'yes', 'yes']),
td(['Broccoli' , 'no', 'no', 'yes']),
td(['Onions' , 'yes','yes', 'yes'])
]
)
);
```

HTML shortcuts and list interpolation

Consider this bit of code:

```
print blockquote(em('Hi'),'mom!'));
```

It will ordinarily return the string that you probably expect, namely:

```
<blockquote><em>Hi</em> mom!</blockquote>
```

Note the space between the element "Hi" and the element "mom!". CGI.pm puts the extra space there using array interpolation, which is controlled by the magic $" variable. Sometimes this extra space is not what you want, for example, when you are trying to align a series of images. In this case, you can simply change the value of $" to an empty string.

```
{
local($") = '';
print blockquote(em('Hi'),'mom!'));
}
```

I suggest you put the code in a block as shown here. Otherwise the change to $" will affect all subsequent code until you explicitly reset it.

Non-standard HTML shortcuts

A few HTML tags don't follow the standard pattern for various reasons.

comment() generates an HTML comment (<!-- comment -->). Call it like

```
print comment('here is my comment');
```

Because of conflicts with built-in perl functions, the following functions begin with initial caps:

```
Select
Tr
Link
Delete
Accept
Sub
```

In addition, *start_html()*, *end_html()*, *start_form()*, *end_form()*, *start_multipart_form()* and all the fill-out form tags are special. See their respective sections.

Autoescaping HTML

By default, all HTML that is emitted by the form-generating functions is passed through a function called *escapeHTML()*:

`$escaped_string` = escapeHTML("unescaped string");

Escape HTML formatting characters in a string. Internally this calls HTML::Entities (encode_entities) so really you should just use that instead - the default list of chars that will be encoded (passed to the HTML::Entities encode_entities method) is:

```
& < > " \x8b \x9b '
```

you can control this list by setting the value of `$CGI::ENCODE_ENTITIES:`

```
# only encode < >
$CGI::ENCODE_ENTITIES = q{<>}
```

if you want to encode **all** entities then undef `$CGI::ENCODE_ENTITIES:`

```
# encode all entities
$CGI::ENCODE_ENTITIES = undef;
```

The automatic escaping does not apply to other shortcuts, such as *h1()*. You should call *escapeHTML()* yourself on untrusted data in order to protect your pages against nasty tricks that people may enter into guestbooks, etc.. To change the character set, use *charset()*. To turn autoescaping off completely, use *autoEscape(0)*:

`$charset` = charset([$charset]);

Get or set the current character set.

`$flag` = autoEscape([$flag]);

Get or set the value of the autoescape flag.

CREATING FILL-OUT FORMS:

General note The various form-creating methods all return strings to the caller, containing the tag or tags that will create the requested form element. You are responsible for actually printing out these strings. It's set up this way so that you can place formatting tags around the form elements.

Another note The default values that you specify for the forms are only used the **first** time the script is invoked (when there is no query string). On subsequent invocations of the script (when there is a query string), the former values are used even if they are blank.

If you want to change the value of a field from its previous value, you have two choices:

(1) call the *param()* method to set it.

(2) use the -override (alias -force) parameter (a new feature in version 2.15). This forces the default value to be used, regardless of the previous value:

```
print textfield(-name=>'field_name',
-default=>'starting value',
-override=>1,
-size=>50,
-maxlength=>80);
```

Yet another note By default, the text and labels of form elements are escaped according to HTML rules. This means that you can safely use "<CLICK ME>" as the label for a button. However, it also interferes with your ability to incorporate special HTML character sequences, such as Á, into your fields. If you wish to turn off automatic escaping, call the *autoEscape()* method with a false value immediately after creating the CGI object:

```
$query = CGI->new;
$query->autoEscape(0);
```

Note that *autoEscape()* is exclusively used to effect the behavior of how some CGI.pm HTML generation functions handle escaping. Calling *escapeHTML()* explicitly will always escape the HTML.

A Lurking Trap! Some of the form-element generating methods return multiple tags. In a scalar context, the tags will be concatenated together with spaces, or whatever is the current value of the $" global. In a list context, the methods will return a list of elements, allowing you to modify them if you wish. Usually you will not notice this behavior, but beware of this:

```
printf("%s\n",end_form())
```

end_form() produces several tags, and only the first of them will be printed because the format only expects one value.

<p>

Creating an isindex tag

```
print isindex(-action=>$action);

-or-

print isindex($action);
```

Prints out an <isindex> tag. Not very exciting. The parameter -action specifies the URL of the script to process the query. The default is to process the query with the current script.

Starting and ending a form

```
print start_form(-method=>$method,
-action=>$action,
-enctype=>$encoding);
<... various form stuff ...>
print end_form;

-or-

print start_form($method,$action,$encoding);
<... various form stuff ...>
print end_form;
```

start_form() will return a <form> tag with the optional method, action and form encoding that you specify. The defaults are:

```
method: POST
action: this script
enctype: application/x-www-form-urlencoded for non-XHTML
multipart/form-data for XHTML, see multipart/form-data below.
```

end_form() returns the closing </form> tag.

start_form()'s enctype argument tells the browser how to package the various fields of the form before sending the form to the server. Two values are possible:

application/x-www-form-urlencoded

This is the older type of encoding. It is compatible with many CGI scripts and is suitable for short fields containing text data. For your convenience, CGI.pm stores the name of this encoding type in **&CGI::URL_ENCODED**.

multipart/form-data

This is the newer type of encoding. It is suitable for forms that contain very large fields or that are intended for transferring binary data. Most importantly, it enables the ''file upload'' feature. For your convenience, CGI.pm stores the name of this encoding type in **&CGI::MULTIPART**

Forms that use this type of encoding are not easily interpreted by CGI scripts unless they use CGI.pm or another library designed to handle them.

If XHTML is activated (the default), then forms will be automatically created using this type of encoding.

The *start_form()* method uses the older form of encoding by default unless XHTML is requested. If you want to use the newer form of encoding by default, you can call ***start_multipart_form()*** instead of ***start_form()***. The method ***end_multipart_form()*** is an alias to ***end_form()***.

JAVASCRIPTING: The **-name** and **-onSubmit** parameters are provided for use with JavaScript. The -name parameter gives the form a name so that it can be identified and manipulated by JavaScript functions. -onSubmit should point to a JavaScript function that will be executed just before the form is submitted to your server. You can use this opportunity to check the contents of the form for consistency and completeness. If you find something wrong, you can put up an alert box or maybe fix things up yourself. You can abort the submission by returning false from this function.

Usually the bulk of JavaScript functions are defined in a <script> block in the HTML header and -onSubmit points to one of these function call. See *start_html()* for details.

Form elements

After starting a form, you will typically create one or more textfields, popup menus, radio groups and other form elements. Each of these elements takes a standard set of named arguments. Some elements also have optional arguments. The standard arguments are as follows:

-name
The name of the field. After submission this name can be used to retrieve the field's value using the *param()* method.

-value, **-values**
The initial value of the field which will be returned to the script after form submission. Some form elements, such as text fields, take a single scalar -value argument. Others, such as popup menus, take a reference to an array of values. The two arguments are synonyms.

-tabindex
A numeric value that sets the order in which the form element receives focus when the user presses the tab key. Elements with lower values receive focus first.

-id A string identifier that can be used to identify this element to JavaScript and DHTML.

-override
A boolean, which, if true, forces the element to take on the value specified by **-value**, overriding the sticky behavior described earlier for the **-nosticky** pragma.

-onChange, **-onFocus**, **-onBlur**, **-onMouseOver**, **-onMouseOut**, **-onSelect**
These are used to assign JavaScript event handlers. See the JavaScripting section for more details.

Other common arguments are described in the next section. In addition to these, all attributes described in the HTML specifications are supported.

Creating a text field

```
print textfield(-name=>'field_name',
-value=>'starting value',
-size=>50,
-maxlength=>80);
-or-

print textfield('field_name','starting value',50,80);
```

textfield() will return a text input field.

Parameters

1. The first parameter is the required name for the field (-name).
2. The optional second parameter is the default starting value for the field contents (-value, formerly known as -default).
3. The optional third parameter is the size of the field in characters (-size).
4. The optional fourth parameter is the maximum number of characters the field will accept (-maxlength).

As with all these methods, the field will be initialized with its previous contents from earlier invocations of the script. When the form is processed, the value of the text field can be retrieved with:

```
$value = param('foo');
```

If you want to reset it from its initial value after the script has been called once, you can do so like this:

```
param('foo',"I'm taking over this value!");
```

Creating a big text field

```
print textarea(-name=>'foo',
-default=>'starting value',
-rows=>10,
-columns=>50);

-or
```

```
print textarea('foo','starting value',10,50);
```

textarea() is just like textfield, but it allows you to specify rows and columns for a multiline text entry box. You can provide a starting value for the field, which can be long and contain multiple lines.

Creating a password field

```
print password_field(-name=>'secret',
-value=>'starting value',
-size=>50,
-maxlength=>80);
-or-

print password_field('secret','starting value',50,80);
```

password_field() is identical to *textfield()*, except that its contents will be starred out on the web page.

Creating a file upload field

```
print filefield(-name=>'uploaded_file',
-default=>'starting value',
-size=>50,
-maxlength=>80);
-or-

print filefield('uploaded_file','starting value',50,80);
```

filefield() will return a file upload field. In order to take full advantage of this *you must use the new multipart encoding scheme* for the form. You can do this either by calling ***start_form()*** with an encoding type of **&CGI::MULTIPART**, or by calling the new method ***start_multipart_form()*** instead of vanilla ***start_form()***.

Parameters

1. The first parameter is the required name for the field (-name).
2. The optional second parameter is the starting value for the field contents to be used as the default file name (-default).

   For security reasons, browsers don't pay any attention to this field, and so the starting value will always be blank. Worse, the field loses its ''sticky'' behavior and forgets its previous contents. The starting value field is called for in the HTML specification, however, and possibly some browser will eventually provide support for it.
3. The optional third parameter is the size of the field in characters (-size).
4. The optional fourth parameter is the maximum number of characters the field will accept (-maxlength).

JAVASCRIPTING: The **-onChange**, **-onFocus**, **-onBlur**, **-onMouseOver**, **-onMouseOut** and **-onSelect** parameters are recognized. See *textfield()* for details.

Creating a popup menu

```
print popup_menu('menu_name',
['eenie','meenie','minie'],
'meenie');

-or-

%labels = ('eenie'=>'your first choice',
'meenie'=>'your second choice',
'minie'=>'your third choice');
%attributes = ('eenie'=>{'class'=>'class of first choice'});
print popup_menu('menu_name',
```

```
['eenie','meenie','minie'],
'meenie',\%labels,\%attributes);

-or (named parameter style)-

print popup_menu(-name=>'menu_name',
-values=>['eenie','meenie','minie'],
-default=>['meenie','minie'],
-labels=>\%labels,
-attributes=>\%attributes);
```

popup_menu() creates a menu. Please note that the -multiple option will be ignored if passed - use *scrolling_list()* if you want to create a menu that supports multiple selections

1. The required first argument is the menu's name (-name).
2. The required second argument (-values) is an array **reference** containing the list of menu items in the menu. You can pass the method an anonymous array, as shown in the example, or a reference to a named array, such as ''\@foo''.
3. The optional third parameter (-default) is the name of the default menu choice. If not specified, the first item will be the default. The values of the previous choice will be maintained across queries. Pass an array reference to select multiple defaults.
4. The optional fourth parameter (-labels) is provided for people who want to use different values for the user-visible label inside the popup menu and the value returned to your script. It's a pointer to an hash relating menu values to user-visible labels. If you leave this parameter blank, the menu values will be displayed by default. (You can also leave a label undefined if you want to).
5. The optional fifth parameter (-attributes) is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name as the key and the attribute's value as the value.

When the form is processed, the selected value of the popup menu can be retrieved using:

```
$popup_menu_value = param('menu_name');
```

Creating an option group

Named parameter style

```
print popup_menu(-name=>'menu_name',
-values=>[qw/eenie meenie minie/,
optgroup(-name=>'optgroup_name',
-values => ['moe','catch'],
-attributes=>{'catch'=>{'class'=>'red'}})],
-labels=>{'eenie'=>'one',
'meenie'=>'two',
'minie'=>'three'},
-default=>'meenie');

Old style
print popup_menu('menu_name',
['eenie','meenie','minie',
optgroup('optgroup_name', ['moe', 'catch'],
{'catch'=>{'class'=>'red'}})],'meenie',
{'eenie'=>'one','meenie'=>'two','minie'=>'three'});
```

optgroup() creates an option group within a popup menu.

1. The required first argument (**-name**) is the label attribute of the optgroup and is **not** inserted in the parameter list of the query.

2. The required second argument (**-values**) is an array reference containing the list of menu items in the menu. You can pass the method an anonymous array, as shown in the example, or a reference to a named array, such as \@foo. If you pass a HASH reference, the keys will be used for the menu values, and the values will be used for the menu labels (see -labels below).

3. The optional third parameter (**-labels**) allows you to pass a reference to a hash containing user-visible labels for one or more of the menu items. You can use this when you want the user to see one menu string, but have the browser return your program a different one. If you don't specify this, the value string will be used instead (''eenie'', ''meenie'' and ''minie'' in this example). This is equivalent to using a hash reference for the -values parameter.

4. An optional fourth parameter (**-labeled**) can be set to a true value and indicates that the values should be used as the label attribute for each option element within the optgroup.

5. An optional fifth parameter (-novals) can be set to a true value and indicates to suppress the val attribute in each option element within the optgroup.

   See the discussion on optgroup at W3C (http://www.w3.org/TR/REC-html40/interact/forms.html#edef-OPTGROUP) for details.

6. An optional sixth parameter (-attributes) is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name as the key and the attribute's value as the value.

Creating a scrolling list

```
print scrolling_list('list_name',
['eenie','meenie','minie','moe'],
['eenie','moe'],5,'true',{'moe'=>{'class'=>'red'}});
-or-

print scrolling_list('list_name',
['eenie','meenie','minie','moe'],
['eenie','moe'],5,'true',
\%labels,%attributes);

-or-

print scrolling_list(-name=>'list_name',
-values=>['eenie','meenie','minie','moe'],
-default=>['eenie','moe'],
-size=>5,
-multiple=>'true',
-labels=>\%labels,
-attributes=>\%attributes);
```

scrolling_list() creates a scrolling list.

Parameters:

1. The first and second arguments are the list name (-name) and values (-values). As in the popup menu, the second argument should be an array reference.

2. The optional third argument (-default) can be either a reference to a list containing the values to be selected by default, or can be a single value to select. If this argument is missing or undefined, then nothing is selected when the list first appears. In the named parameter version, you can use the synonym ''-defaults'' for this parameter.

3. The optional fourth argument is the size of the list (-size).

4. The optional fifth argument can be set to true to allow multiple simultaneous selections (-multiple). Otherwise only one selection will be allowed at a time.

5. The optional sixth argument is a pointer to a hash containing long user-visible labels for the list items (-labels). If not provided, the values will be displayed.

6. The optional sixth parameter (-attributes) is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name as the key and the attribute's value as the value.

   When this form is processed, all selected list items will be returned as a list under the parameter name 'list_name'. The values of the selected items can be retrieved with:

```
@selected = param('list_name');
```

Creating a group of related checkboxes

```
print checkbox_group(-name=>'group_name',
-values=>['eenie','meenie','minie','moe'],
-default=>['eenie','moe'],
-linebreak=>'true',
-disabled => ['moe'],
-labels=>\%labels,
-attributes=>\%attributes);

print checkbox_group('group_name',
['eenie','meenie','minie','moe'],
['eenie','moe'],'true',\%labels,
{'moe'=>{'class'=>'red'}});

HTML3-COMPATIBLE BROWSERS ONLY:

print checkbox_group(-name=>'group_name',
-values=>['eenie','meenie','minie','moe'],
-rows=2,-columns=>2);
```

checkbox_group() creates a list of checkboxes that are related by the same name.

Parameters:

1. The first and second arguments are the checkbox name and values, respectively (-name and -values). As in the popup menu, the second argument should be an array reference. These values are used for the user-readable labels printed next to the checkboxes as well as for the values passed to your script in the query string.

2. The optional third argument (-default) can be either a reference to a list containing the values to be checked by default, or can be a single value to checked. If this argument is missing or undefined, then nothing is selected when the list first appears.

3. The optional fourth argument (-linebreak) can be set to true to place line breaks between the checkboxes so that they appear as a vertical list. Otherwise, they will be strung together on a horizontal line.

The optional **-labels** argument is a pointer to a hash relating the checkbox values to the user-visible labels that will be printed next to them. If not provided, the values will be used as the default.

The optional parameters **-rows**, and **-columns** cause *checkbox_group()* to return an HTML3 compatible table containing the checkbox group formatted with the specified number of rows and columns. You can provide just the -columns parameter if you wish; checkbox_group will calculate the correct number of rows for you.

The option **-disabled** takes an array of checkbox values and disables them by greying them out (this may not be supported by all browsers).

The optional **-attributes** argument is provided to assign any of the common HTML attributes to an individual menu item. It's a pointer to a hash relating menu values to another hash with the attribute's name

as the key and the attribute's value as the value.

The optional **-tabindex** argument can be used to control the order in which radio buttons receive focus when the user presses the tab button. If passed a scalar numeric value, the first element in the group will receive this tab index and subsequent elements will be incremented by one. If given a reference to an array of radio button values, then the indexes will be jiggered so that the order specified in the array will correspond to the tab order. You can also pass a reference to a hash in which the hash keys are the radio button values and the values are the tab indexes of each button. Examples:

```
-tabindex => 100 # this group starts at index 100 and counts up
-tabindex => ['moe','minie','eenie','meenie'] # tab in this order
-tabindex => {meenie=>100,moe=>101,minie=>102,eenie=>200} # tab in this order
```

The optional **-labelattributes** argument will contain attributes attached to the <label> element that surrounds each button.

When the form is processed, all checked boxes will be returned as a list under the parameter name 'group_name'. The values of the ''on'' checkboxes can be retrieved with:

```
@turned_on = param('group_name');
```

The value returned by *checkbox_group()* is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

```
@h = checkbox_group(-name=>'group_name',-values=>\@values);
&use_in_creative_way(@h);
```

Creating a standalone checkbox

```
print checkbox(-name=>'checkbox_name',
-checked=>1,
-value=>'ON',
-label=>'CLICK ME');

-or-

print checkbox('checkbox_name','checked','ON','CLICK ME');
```

checkbox() is used to create an isolated checkbox that isn't logically related to any others.

Parameters:

1. The first parameter is the required name for the checkbox (-name). It will also be used for the user-readable label printed next to the checkbox.
2. The optional second parameter (-checked) specifies that the checkbox is turned on by default. Synonyms are -selected and -on.
3. The optional third parameter (-value) specifies the value of the checkbox when it is checked. If not provided, the word ''on'' is assumed.
4. The optional fourth parameter (-label) is the user-readable label to be attached to the checkbox. If not provided, the checkbox name is used.

The value of the checkbox can be retrieved using:

```
$turned_on = param('checkbox_name');
```

Creating a radio button group

```
print radio_group(-name=>'group_name',
-values=>['eenie','meenie','minie'],
-default=>'meenie',
-linebreak=>'true',
-labels=>\%labels,
-attributes=>\%attributes);
```

```
-or-

print radio_group('group_name',['eenie','meenie','minie'],
'meenie','true',\%labels,\%attributes);


HTML3-COMPATIBLE BROWSERS ONLY:

print radio_group(-name=>'group_name',
-values=>['eenie','meenie','minie','moe'],
-rows=2,-columns=>2);
```

radio_group() creates a set of logically-related radio buttons (turning one member of the group on turns the others off)

Parameters:

1. The first argument is the name of the group and is required (-name).
2. The second argument (-values) is the list of values for the radio buttons. The values and the labels that appear on the page are identical. Pass an array *reference* in the second argument, either using an anonymous array, as shown, or by referencing a named array as in ‘‘\@foo’’.
3. The optional third parameter (-default) is the name of the default button to turn on. If not specified, the first item will be the default. You can provide a nonexistent button name, such as ‘‘-’’ to start up with no buttons selected.
4. The optional fourth parameter (-linebreak) can be set to ’true’ to put line breaks between the buttons, creating a vertical list.
5. The optional fifth parameter (-labels) is a pointer to an associative array relating the radio button values to user-visible labels to be used in the display. If not provided, the values themselves are displayed.

All modern browsers can take advantage of the optional parameters **-rows**, and **-columns**. These parameters cause *radio_group()* to return an HTML3 compatible table containing the radio group formatted with the specified number of rows and columns. You can provide just the -columns parameter if you wish; radio_group will calculate the correct number of rows for you.

To include row and column headings in the returned table, you can use the **-rowheaders** and **-colheaders** parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don’t reorganize the interpretation of the radio buttons — they’re still a single named unit.

The optional **-tabindex** argument can be used to control the order in which radio buttons receive focus when the user presses the tab button. If passed a scalar numeric value, the first element in the group will receive this tab index and subsequent elements will be incremented by one. If given a reference to an array of radio button values, then the indexes will be jiggered so that the order specified in the array will correspond to the tab order. You can also pass a reference to a hash in which the hash keys are the radio button values and the values are the tab indexes of each button. Examples:

```
-tabindex => 100 # this group starts at index 100 and counts up
-tabindex => ['moe','minie','eenie','meenie'] # tab in this order
-tabindex => {meenie=>100,moe=>101,minie=>102,eenie=>200} # tab in this order
```

The optional **-attributes** argument is provided to assign any of the common HTML attributes to an individual menu item. It’s a pointer to a hash relating menu values to another hash with the attribute’s name as the key and the attribute’s value as the value.

The optional **-labelattributes** argument will contain attributes attached to the <label> element that surrounds each button.

When the form is processed, the selected radio button can be retrieved using:

```
$which_radio_button = param('group_name');
```

The value returned by *radio_group()* is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

```
@h = radio_group(-name=>'group_name',-values=>\@values);
&use_in_creative_way(@h);
```

Creating a submit button

```
print submit(-name=>'button_name',
-value=>'value');

-or-

print submit('button_name','value');
```

submit() will create the query submission button. Every form should have one of these.

Parameters:

1. The first argument (-name) is optional. You can give the button a name if you have several submission buttons in your form and you want to distinguish between them.
2. The second argument (-value) is also optional. This gives the button a value that will be passed to your script in the query string. The name will also be used as the user-visible label.
3. You can use -label as an alias for -value. I always get confused about which of -name and -value changes the user-visible label on the button.

You can figure out which button was pressed by using different values for each one:

```
$which_one = param('button_name');
```

Creating a reset button

```
print reset
```

reset() creates the ‘‘reset’’ button. Note that it restores the form to its value from the last time the script was called, NOT necessarily to the defaults.

Note that this conflicts with the perl *reset()* built-in. Use *CORE::reset()* to get the original reset function.

Creating a default button

```
print defaults('button_label')
```

defaults() creates a button that, when invoked, will cause the form to be completely reset to its defaults, wiping out all the changes the user ever made.

Creating a hidden field

```
print hidden(-name=>'hidden_name',
-default=>['value1','value2'...]);

-or-

print hidden('hidden_name','value1','value2'...);
```

hidden() produces a text field that can’t be seen by the user. It is useful for passing state variable information from one invocation of the script to the next.

Parameters:

1. The first argument is required and specifies the name of this field (-name).
2. The second argument is also required and specifies its value (-default). In the named parameter style of calling, you can provide a single value here or a reference to a whole list

Fetch the value of a hidden field this way:

```
$hidden_value = param('hidden_name');
```

Note, that just like all the other form elements, the value of a hidden field is ‘‘sticky’’. If you want to replace a hidden field with some other values after the script has been called once you’ll have to do it manually:

```
param('hidden_name','new','values','here');
```

Creating a clickable image button

```
print image_button(-name=>'button_name',
-src=>'/source/URL',
-align=>'MIDDLE');

-or-

print image_button('button_name','/source/URL','MIDDLE');
```

image_button() produces a clickable image. When it’s clicked on the position of the click is returned to your script as ‘‘button_name.x’’ and ‘‘button_name.y’’, where ‘‘button_name’’ is the name you’ve assigned to it.

Parameters:

1. The first argument (-name) is required and specifies the name of this field.
2. The second argument (-src) is also required and specifies the URL
3. The third option (-align, optional) is an alignment type, and may be TOP, BOTTOM or MIDDLE

Fetch the value of the button this way: $x = param(’button_name.x’); $y = param(’button_name.y’);

Creating a javascript action button

```
print button(-name=>'button_name',
-value=>'user visible label',
-onClick=>"do_something()");

-or-

print button('button_name',"user visible value","do_something()");
```

button() produces an `<input>` tag with `type="button"`. When it’s pressed the fragment of JavaScript code pointed to by the **-onClick** parameter will be executed.

WORKING WITH FRAMES

It’s possible for CGI.pm scripts to write into several browser panels and windows using the HTML 4 frame mechanism. There are three techniques for defining new frames programmatically:

1. Create a <Frameset> document

   After writing out the HTTP header, instead of creating a standard HTML document using the *start_html()* call, create a <frameset> document that defines the frames on the page. Specify your script(s) (with appropriate parameters) as the SRC for each of the frames.

   There is no specific support for creating <frameset> sections in CGI.pm, but the HTML is very simple to write.

2. Specify the destination for the document in the HTTP header

   You may provide a **-target** parameter to the *header()* method:

   ```
   print header(-target=>'ResultsWindow');
   ```

   This will tell the browser to load the output of your script into the frame named ‘‘ResultsWindow’’. If a frame of that name doesn’t already exist, the browser will pop up a new window and load your script’s document into that. There are a number of magic names that you can use for targets. See the HTML `<frame>` documentation for details.

3. Specify the destination for the document in the <form> tag
   You can specify the frame to load in the FORM tag itself. With CGI.pm it looks like this:

```
print start_form(-target=>'ResultsWindow');
```

   When your script is reinvoked by the form, its output will be loaded into the frame named ''ResultsWindow''. If one doesn't already exist a new window will be created.

The script ''frameset.cgi'' in the examples directory shows one way to create pages in which the fill-out form and the response live in side-by-side frames.

SUPPORT FOR JAVASCRIPT

The usual way to use JavaScript is to define a set of functions in a <SCRIPT> block inside the HTML header and then to register event handlers in the various elements of the page. Events include such things as the mouse passing over a form element, a button being clicked, the contents of a text field changing, or a form being submitted. When an event occurs that involves an element that has registered an event handler, its associated JavaScript code gets called.

The elements that can register event handlers include the <BODY> of an HTML document, hypertext links, all the various elements of a fill-out form, and the form itself. There are a large number of events, and each applies only to the elements for which it is relevant. Here is a partial list:

onLoad
The browser is loading the current document. Valid in:

```
+ The HTML <BODY> section only.
```

onUnload
The browser is closing the current page or frame. Valid for:

```
+ The HTML <BODY> section only.
```

onSubmit
The user has pressed the submit button of a form. This event happens just before the form is submitted, and your function can return a value of false in order to abort the submission. Valid for:

```
+ Forms only.
```

onClick
The mouse has clicked on an item in a fill-out form. Valid for:

```
+ Buttons (including submit, reset, and image buttons)
+ Checkboxes
+ Radio buttons
```

onChange
The user has changed the contents of a field. Valid for:

```
+ Text fields
+ Text areas
+ Password fields
+ File fields
+ Popup Menus
+ Scrolling lists
```

onFocus
The user has selected a field to work with. Valid for:

```
+ Text fields
+ Text areas
+ Password fields
+ File fields
+ Popup Menus
+ Scrolling lists
```

onBlur

The user has deselected a field (gone to work somewhere else). Valid for:

```
+ Text fields
+ Text areas
+ Password fields
+ File fields
+ Popup Menus
+ Scrolling lists
```

onSelect

The user has changed the part of a text field that is selected. Valid for:

```
+ Text fields
+ Text areas
+ Password fields
+ File fields
```

onMouseOver

The mouse has moved over an element.

```
+ Text fields
+ Text areas
+ Password fields
+ File fields
+ Popup Menus
+ Scrolling lists
```

onMouseOut

The mouse has moved off an element.

```
+ Text fields
+ Text areas
+ Password fields
+ File fields
+ Popup Menus
+ Scrolling lists
```

In order to register a JavaScript event handler with an HTML element, just use the event name as a parameter when you call the corresponding CGI method. For example, to have your *validateAge()* JavaScript code executed every time the textfield named ‘‘age’’ changes, generate the field like this:

```
print textfield(-name=>'age',-onChange=>"validateAge(this)");
```

This example assumes that you’ve already declared the *validateAge()* function by incorporating it into a <SCRIPT> block. The CGI.pm *start_html()* method provides a convenient way to create this section.

Similarly, you can create a form that checks itself over for consistency and alerts the user if some essential value is missing by creating it this way: print start_form(-onSubmit=>‘‘validateMe(this)’’);

See the javascript.cgi script for a demonstration of how this all works.

LIMITED SUPPORT FOR CASCADING STYLE SHEETS

CGI.pm has limited support for HTML3’s cascading style sheets (css). To incorporate a stylesheet into your document, pass the *start_html()* method a **-style** parameter. The value of this parameter may be a scalar, in

which case it is treated as the source URL for the stylesheet, or it may be a hash reference. In the latter case you should provide the hash with one or more of **-src** or **-code**. **-src** points to a URL where an externally-defined stylesheet can be found. **-code** points to a scalar value to be incorporated into a <style> section. Style definitions in **-code** override similarly-named ones in **-src**, hence the name ‘‘cascading.’’

You may also specify the type of the stylesheet by adding the optional **-type** parameter to the hash pointed to by **-style**. If not specified, the style defaults to ’text/css’.

To refer to a style within the body of your document, add the **-class** parameter to any HTML element:

```
print h1({-class=>'Fancy'},'Welcome to the Party');
```

Or define styles on the fly with the **-style** parameter:

```
print h1({-style=>'Color: red;'},'Welcome to Hell');
```

You may also use the new ***span()*** element to apply a style to a section of text:

```
print span({-style=>'Color: red;'},
h1('Welcome to Hell'),
"Where did that handbasket get to?"
);
```

Note that you must import the ‘‘:html3’’ definitions to have the ***span()*** method available. Here’s a quick and dirty example of using CSS’s. See the CSS specification at http://www.w3.org/Style/CSS/ for more information.

```
use CGI qw/:standard :html3/;

#here's a stylesheet incorporated directly into the page
$newStyle=<<END;
<!--
P.Tip {
margin-right: 50pt;
margin-left: 50pt;
color: red;
}
P.Alert {
font-size: 30pt;
font-family: sans-serif;
color: red;
}
-->
END
print header();
print start_html( -title=>'CGI with Style',
                  -style=>{-src=>'http://www.capricorn.com/style/st1.css',
-code=>$newStyle}
);
print h1('CGI with Style'),
p({-class=>'Tip'},
"Better read the cascading style sheet spec before playing with this!"),
span({-style=>'color: magenta'},
"Look Mom, no hands!",
p(),
"Whooo wee!"
);
print end_html;
```

Pass an array reference to **-code** or **-src** in order to incorporate multiple stylesheets into your document.

Should you wish to incorporate a verbatim stylesheet that includes arbitrary formatting in the header, you may pass a -verbatim tag to the -style hash, as follows:

print start_html (-style => {-verbatim => '@import url(''/server-common/css/'.$cssFile.''');', -src => '/server-common/css/core.css'});

This will generate an HTML header that contains this:

```
<link rel="stylesheet" type="text/css" href="/server-common/css/core.css">
<style type="text/css">
@import url("/server-common/css/main.css");
</style>
```

Any additional arguments passed in the -style value will be incorporated into the <link> tag. For example:

```
start_html(-style=>{-src=>['/styles/print.css','/styles/layout.css'],
-media => 'all'});
```

This will give:

```
<link rel="stylesheet" type="text/css" href="/styles/print.css" media="all"/>
<link rel="stylesheet" type="text/css" href="/styles/layout.css" media="all"/>
```

<p>

To make more complicated <link> tags, use the *Link()* function and pass it to *start_html()* in the -head argument, as in:

```
@h = (Link({-rel=>'stylesheet',-type=>'text/css',-src=>'/ss/ss.css',-media=>'all
Link({-rel=>'stylesheet',-type=>'text/css',-src=>'/ss/fred.css',-media=>'paper'}
print start_html({-head=>\@h})
```

To create primary and ''alternate'' stylesheet, use the **-alternate** option:

```
start_html(-style=>{-src=>[
{-src=>'/styles/print.css'},
{-src=>'/styles/alt.css',-alternate=>1}
]
});
```

Dumping out all the name/value pairs

The *Dump()* method produces a string consisting of all the query's name/value pairs formatted nicely as a nested list. This is useful for debugging purposes:

```
print Dump
```

Produces something that looks like:

```
<ul>
<li>name1
<ul>
<li>value1
<li>value2
</ul>
<li>name2
<ul>
<li>value1
</ul>
</ul>
```

As a shortcut, you can interpolate the entire CGI object into a string and it will be replaced with the a nice HTML dump shown above:

```
$query=CGI->new;
print "<h2>Current Values</h2> $query\n";
```

BUGS

Address bug reports and comments to: <https://github.com/leejo/CGI.pm/issues>

The original bug tracker can be found at: <https://rt.cpan.org/Public/Dist/Display.html?Queue=CGI.pm>

However as stated this functionality is no longer being maintained and is considered deprecated. Any feature requests, bug reports, issues, pull requests, etc, for this functionality will almost certainly be rejected without any action being taken place - this includes fixes to utterly broken page rendering, invalid HTML, nonsensical output, and annoyances.

SEE ALSO

CGI - The original source of this documentation / functionality