Python Servlet Engine Manual

Release 3.0.6

Nicholas D. Borko

Christian Höltje

April 25, 2006

Abstract

The Python Servlet Engine (PSE) is an application server for web-based applications written in the Python programming language. It offers powerful templating tools, separation of presentation and implementation, and unlimited extensibility through its plugin and custom tag architectures. PSE includes some default plugins that allow automatic session management, persistent data and access to client and server information.

PSE is written in Python for use with mod_python in Apache or as a CGI script. Completed applications written for PSE can be compiled and distributed *without* the original Python source code, allowing application developers to protect their intellectual property.

CONTENTS

Introduction

PSE is an application server framework that lets you build complex web-based applications in Python. It works as an apache module built in conjunction with mod_python. The application uses compiled "servlets" written in Python to produce application output. This means that each page runs very fast because it is the result of executing compiled Python code — no need for parsing past the first run, as with PHP or other web application servers.

Each servlet has two components, the template and the code module. This kind of architecture allows the separation of presentation and implementation, i.e. the HTML designer and the Python programmer. The template file is a syntactically valid HTML file with processing instructions and/or custom tags defined by the programmer. This file is parsed and translated into an executable Python script, which is compiled and stored. A template file has the file extenson '.pt'.

The code module is a regular Python module, which is also compiled and stored in the same way that Python itself compiles and stores modules. When a web request is made, PSE executes the code module first in the global namespace, then executes the template servlet in the same global namespace as the code module. This allows any Python generated from the template to access everything that was created by the code module. A code module has the same base file name as the template, but with the extension '.py'.

It is possible to create a template without any code module. This frees the HTML designer to create working mock-ups of pages that can be filled in later by program logic created by the Python programmer in the code module. It is not possible to create a stand-alone code module without a template, although the template could be an empty file. This is for security reasons; it allows modules to be placed in the application directory so they can be imported by other modules, but never executed as a stand-alone script.

Template files get compiled to '.ptc' files. An intermediate file ending in '.pty' is stored to aid in debugging, which contains the source code for the compiled servlet. The code module is compiled to a file ending in '.pyc' or '.pyo', the same as normal python modules. In addition, output from a servlet execution can be cached according to whatever rules the programmer specifies, which results in a '.pto' file. When output is cached, the servlet will not be executed until the cached file expires, and only the cached file will be sent instead.

There are tools to pre-compile the template and code modules. If a directory only consists of '.ptc' and '.pyc' (or '.pyo') files, PSE will only use them and never look for the source files. This way users of the application will never have to experience a delay due to parsing and compiling of any files, and you can distribute only optimized compiled files to protect your intellectual property in the source code.

Installation

2.1 Requirements

2.1.1 Python

PSE requires Python version 2.3.1 or higher. Older versions are missing features that are needed to make PSE work right. Python is available under an Open Source license at http://python.org/

Under Mac OS X, users have reported success using python from Fink. Fink is available at http://fink.sf.net/.

2.1.2 Apache Web Server

PSE requires the Apache web server, version 1.3 or 2.0.

If your operating system has Apache, there is a good chance that it is a version that will work with PSE. If you want the latest version or if your operating system doesn't have apache, you can get it from http://httpd.apache.org/.

For the best performance and most stable experience, we suggest using Apache version 2.0.

2.1.3 mod_python

PSE requires a fairly recent version of mod_python. The exact version depends on your operating system and the version of Apache.

You may download the latest version of mod_python from http://modpython.org/.

2.2 Installing the files

PSE comes in several forms: Either as a source package or as a distribution package.

A source package is the most flexible way to install PSE. However, an appropriate distribution package might be the easiest way to install PSE.

2.2.1 Installation from Source Archive

You can unpack the PSE archive using following command:

```
$ tar zxf PSE-X.X.X.tar.gz
```

where *X.X.X* is the version number of PSE you are installing. You can also use your favorite unzipping program if you download the zipped version of the archive.

PSE will be extracted into a subdirectory named 'PSE-*X.X.X*' where *X.X.X* is the version number of PSE that you extracted. You should change directory (`cd`) to this directory.

Then, to actually install it, you should run 'setup.py' with the install command:

```
$ python setup.py install
```

You may wish to check out the options for install first by passing in the *–help* option to 'setup.py'.

2.2.2 Binary Installation for Windows

The file will be named 'PSE-X.X.X.win32.exe' where 'X.X.X' will be the actual version number of PSE.

Simply double click the file in Windows Explorer and follow the instructions in the installation wizard. The files should all be in the correct places.

Configuration

3.1 Configuring Apache

Before being able to use PSE with Apache, you must add a few directives to Apache's configuration. The following directives can be added to a `VirtualHost`, `Directory`, or `Location` section. If you don't have one of these sections, or if you prefer PSE to be available everywhere, then these directives can be added outside all sections.

3.1.1 Basic Setup

At the very least, you'll need to have the following lines in your Apache configuration file:

```
PythonHandler pse_handler
AddHandler python-program .pt
```

The first directive instructs mod_python to use `pse_handler` to handle all python requests. The second directive says that anything that ends in .pt should be handled by the python handler (`pse_handler`).

There are very very few situations where you would want to change these lines. This works pretty much universally.

3.1.2 Python Path

You can change the python path that mod_python uses. This is useful if you installed PSE someplace other than the default location.

To change the python path, use the `PythonPath` directive:

```
PythonPath "sys.path + [ '/path/to/PSE-X.X.X' ]"
```

The part between the quotes is actual python code. In this case, it says to append the list `['/path/to/PSE-X.X.X']` to the list `sys.path`. This tells python to search in '/path/to/PSE-X.X.X' for more python libraries.

You can use this directive to set different python paths for each PSE instance (see section 3.1.5).

3.1.3 Optimization Level

Another directive you might also wish to add is `PythonOptimize`:

```
PythonOptimize On
```

As you probably guessed, this is the same as passing the -O flag to python. Just like using the -O flag in python, it might prevent you from debugging code during development, so you probably only want to use it on production servers.

You can use this directive to set different optimizations for each PSE instance (see section 3.1.5).

3.1.4 Configuration File Location

Under UNIX operating systems, the default configuration file is '/etc/pse/pse.conf'. Under Windows, the default location is in 'etc/pse/pse.conf' in the directory Python was installed in.

You can change the configuration file that is being used by PSE using the Apache `PythonOption` directive. This lets you keep your configuration someplace else, such as under version control.

You can use this directive to have a different configuration for each PSE instance (see section 3.1.5).

```
PythonOption PSEConf /path/to/your/pse.conf
```

For more information on the configuration file and what it does, see section 3.2.

3.1.5 Multiple PSE Instances per Web-Server

Multiple PSE instances can be run within the same Apache server using the `PythonInterpPerDirective` or `PythonInterpPerDirectory` directives. Each interpreter can have a separate `PythonOption`, `PythonOptimize` and `PythonPath` directives.

Note: Each PSE instance *must* run in its own interpreter instance, separate from any other mod_python handlers. This is because PSE significantly alters the runtime environment in ways that may be unexpected in other handlers.

See the mod_python documentation for more information.

3.2 Configuring PSE

PSE's behavior is controlled by it's configuration file, 'pse.conf'. This file uses the standard `.ini` file format made popular in Windows 3.0. You can see how to change the location of the PSE configuration file in section 3.1.4.

The file is made up of sections and options. A section is a line with an open square bracket, the section name, and a close square bracket. An option is a line with the option name, an equal sign and the value of the option. Comments are lines that begin with a semi-colon or an octothorpe (aka a pound sign or hash symbol).

An example looks like this:

```
[Main]
;; MaxBuffer is the maximum buffer size for file reads
MaxBuffer = 131072

;; MemoryCacheSize is the maximum number of compiled files to keep in memory
;; per server process.  Note that templates and the modules that go with
;; them count as two separate files, even though they execute as a single
;; servlet.  You may disable caching by specifying a value of 0 here.
;; The default is 100.
MemoryCacheSize = 100
```

In that example, the section is `Main`. The option `MaxBuffer` is set to the value `131072`. The option `MemoryCacheSize` is set to `100`.

In the following sections, we'll talk about the different sections and the options in each. In addition, we'll try to provide some advice about how to set them correctly for your setup.

3.2.1   The [Main] section

The `Main` section is all about the PSE process and how it interacts with the system and Apache.

MaxBuffer

The *MaxBuffer* option is used to control the size of the file buffer when PSE reads or writes. Use this to prevent large requests from swamping PSE and the web server.

This applies only to PSE itself. If you read or write to files in your own code (a page template or a plugin) then you control the buffering there via normal python means (`pydoc file.read` for more info).

MemoryCacheSize

MemoryCacheSize controls the number of compiled files ('.pyc' and '.pty') that PSE keeps cached in *memory*.

Usually a page has two of these files, one for the template and one for the python code. So if you wanted to keep 50 pages cached in memory for maximum performance, you'd want to set *MemoryCacheSize* to 100.

Setting *MemoryCacheSize* to 0 means that no compiled files will be cached in memory and that the file system will be consulted for each page access.

OuputCacheDir

Use *OuputCacheDir* to control where the cached output files are kept for servlets that explicitly set *pse.cache*. By default, it is the subdirectory 'pse_cache' in your system temporary directory (such as '/tmp' in UNIX).

Path

The *Path* option changes the `sys.path` inside a servlet page.

The `sys.path` is: the default python path + *path* + dirname(`servlet directory`)

If you aren't familiar with python's `sys.path`, it's the list of directories that python (or PSE in this case) will look for modules and files for `import` statements.

PreImport

PreImport allows you to import a module *before* the next page request.

The advantage of this is that modules that take a long time to load will all ready be loaded when a user requests a page. The module still needs to be explicitly imported, but the loading and initialization of the module will have already been completed. This can lead to increased peformance when importing heavy modules that take a long time to load and initialize.

There are three things to note about using the *PreImport* directive:

1. If the pre-imported module is updated, the effects will not be seen until after the *next* request, since the module has already been pre-imported.

2. The module(s) must be within default python import path (aka `sys.path`).

3. The module is not persistent; that is, each request sees a newly loaded and initialized module. Modules can be persisted by using a plugin or custom tag.

UserPluginPath

If you want to keep your PSE plugins in a different directory, then you can set *UserPluginPath* and then drop then in there.

This list is a comma separated list of directories. These directories must *only* contain plugins. Any files that aren't plugins can (and probably will) fail to load, and PSE will log these attempts.

IncludePath

The *IncludePath* option lets you have servlets that can be included or called in servlets using `pse.include` or `pse.call`. This list is a comma separated list of directories. Only the directories listed here will be searched.

By default, *IncludePath* is empty, meaning you cannot include or call external servlets. A typical configuration might include the current directory ('.'):

```
IncludePath = .
```

An example usage would be if all your pages had a common footer. You could create the file 'commonfooter.pt' and put it in a directory in *IncludePath* and then use `pse.include('commonfooter.pt')` to include it in every page.

Note: These directories don't need to be (and in fact probably shouldn't be) within the exposed web directories. However, it won't hurt if they are in the exposed web directories.

DefaultEncoding

DefaultEncoding specifies the default encoding for unicode strings. The default is `UTF-8`. This can be changed in a servlet using the `pse.setencoding` function.

3.2.2 The [Parser] section

TagHooksModule

The *TagHooksModule* specifies a module or list of modules that define custom tag hooks. You *must* specify a full, absolute path here. Refer to the Custom Tags API section in Developing with PSE.

3.2.3 The [Debug] section

ShowTrace

ShowTrace indicates whether or not to show tracebacks in the browser at all. The default is Yes.

TraceAll

Normally PSE only shows tracebacks if the source is available. Setting *TraceAll* to Yes will cause PSE to always attempt to show a traceback. This overrides the value of *ShowTrace*. The default is No.

ErrorLog

ErrorLog is the path to a file to log tracebacks, regardless of whether the traceback is shown in the browser. The default is to log to apache's `ErrorLog` if unset. The default is to log to apache's `ErrorLog`.

ApacheLogAll

ApacheLogAll indicates whether all errors should also be logged to the apache error log, even if ErrorLog is set. The default is No.

ExceptionHooksModule

ExceptionHooksModule defines functions to intercept uncaught exceptions instead of the builtin exception display. Refer to the Custom Exception Handling Hooks section in Developing with PSE.

3.2.4 The [UserOptions] section

The `[UserOptions]` section is different in that any arbitrary key/value pair can be specified here. These key/value pairs can be accessed in a servlet by referring to the key of the `pse.user_options` dictionary. All values in this dictionary are strings.

3.2.5 The [plugin.request] section

FormKeyError

This option effects `pse.plugins.request.form` object. `pse.plugins.request.form` is a dictionary-like object that contains the GET and POST arguments for the currently requested page.

Setting *FormKeyError* to `Yes` will cause `pse.plugin.request.form` to behave like a normal dictionary (raising a `KeyError`) if you try to access a non-existent key.

Setting *FormKeyError* to `No` will cause `pse.plugin.request.form` to return an empty string (`''`) if an invalid key is used. `has_key()`, `keys()`, etc. will still behave normally. By default, it is set to `No`.

See section 5.2.3 for more information.

3.2.6   The [plugin.session] section

PSE ships with three difference session plugins, each with identical APIs. The default is a traditional dbm implementation using a bsddb back-end, which ships with most versions of Python. Another uses an SQLite back-end, which is slower but only uses a single file. Finally, there is a PostgreSQL back-end for stateless load balanced servers.

For most low traffic, single machine sites, the default dbm session back-end is fast and reliable. For sites that process an extremely high volume of requests (more than 100 per second) or that use load balanced servers, the PostgreSQL back-end is recommended.

The SQLite session plugin is interesting insomuch as it can be used as an example for implementing your own SQL session back-ends. Since it is fairly slow and unreliable for high volume web sites, its use is discouraged. The dbm implementation is much more reliable and efficient for low to middle traffic sites, and high volume sites should consider a real enterprise RDBMS implementation.

Please read the PSE API documentation for more details on using the session plugin in general.

In order to use the SQLite session plugin for PSE, you need to compile and install both at least version 3 of the SQLite libraries and at least version 2.0 of the Python bindings (pysqlite2, from http://initd.org/tracker/pysqlite). For the PostgreSQL session plugin, you need the psycopg module (from http://initd.org/software/initd/psycopg). Many Linux systems already have these modules available, either included with the distribution CD-ROM or available for download.

For configuration information of the session plugin, refer to session module documentation in The PSE API.

3.2.7   Other [plugin.*] sections

In general, these sections contain configurable options for builtin and user plugins. These are defined in the plugin using the `plugin_config` dictionary in the plugin module. Refer to the Creating Plugins section in Developing with PSE for information on how to use plugin options.

There is a common option that is shared among all plugins, the *Enabled* option, which is `Yes` by default. You can explicitly disable plugins by setting *Enabled* to `No`. The required PSE plugins `request`, `response`, `client` and `server` cannot be disabled.

PSE Templates

PSE templates are the core of any PSE application. They allow developers to separate the application presentation (HTML markup) from the implementation (Python code) in different ways. First, XML processing instructions can be used to embed Python directly in the HTML template. This is very convenient for very rapid application development by someone knowledgable in both Python and HTML. Second, you can replace attribute values of HTML tags with Python expressions or statements. The final way is by using "Custom Tags" developed by programmers that completely insulate the HTML designers from the Python code.

It is important to note that while the resulting servlet output is made from the template, the template is actually used to create a Python script that is compiled and stored. The compiled code is then excecuted when a request is made to the web server. This results in fast execution, because the template only has to be processed once and the resulting servlet source compiled once. Templates can even be compiled in advance, ensuring that the user will never experience a syntax error or delay during processing.

4.1 Processing Instructions

Python scripts can be directly embedded in HTML using processing instruction tags, similar to PHP syntax. Because the parser validates the HTML syntax, you won't have to worry about hunting down a missing quote or tag symbol. PSE processing instructions look like this:

```
<? statement ?>
```

PSE understands processing instructions in three different ways. First is the most simple case, where raw Python statements appear in the tag. You can have one or as many statements as you like. However, if you use more than one statement, it is recommended that your first statement begins on a new line, separate from the opening `<?` tag, like this:

```
<?
statement 1
statement 2
 ...
statement n
?>
```

This is because Python uses indentation to recognize program blocks. Overall indentation is managed by PSE (you'll see how later), so you don't have to worry about making sure the code in any particular tag is lined up with any previous code. However, you *must* complete any block within a single tag, as indentation control is returned to PSE when it reaches the closing `?>`.

If you want to control indentation between processing instructions, then let PSE manage the indentation for you. If the last statement in an instruction ends in a colon (':'), this signals PSE to begin a new indentation level. Therefore, the instruction:

```
<? if condition: ?>
```

will generate the `if` statement and increase indentation by one level. Any subsequent code generated from proceding markup will be indented at the correct level, even code indented manually as described above.

If the first statement in a processing instruction *begins* with a colon, then the instruction will be treated as a "pseudo-instruction" and backdent one level. For example, to stop indentation generated by the `if` statement above, you could do the following:

```
<? :if condition ?>
```

What follows the inital colon is ignored by PSE, so you can use this as a comment space to help you remember which block it is you are closing. Since *all* Python blocks begin with a statement that ends in a colon, you can use this construct with `for`, `try` or any other statement that uses a block.

Often you want to match an `if` or `try` with an `else` or `except`. This can be cumbersome with the method described, so there is a shortcut you can use in these cases. If you begin a statment with a colon *and* end it with a colon, PSE will backdent, use the statement, and re-indent again. This may sound confusing, but it should be clear with the following example:

```
<? try: ?>
other code and markup here
<? :except ValueError: ?>
exception handling
<? :except: ?>
maybe you will want catch more
<? :else: ?>
you can even do this; it doesn't matter as long as it's valid Python
<? :try (to backdent when finished) ?>
```

Finally, there is one other way to use the processing instructions. You can cause the resulting code to produce output of the value of any Python expression using the following syntax:

```
<?=valid python expression ?>
```

In this tag, white space can appear between the initial '?' and '=' or between the '=' and the expression, as long as '=' is the first character in the instruction. The resulting code does a `str()` on the expression, so you don't have to worry about `ValueError` exceptions being raised from non-string values.

4.2   HTML Attribute Substitution

In the last section, you saw how arbitrary Python could be embedded in HTML using processing instructions. Since the HTML template *must* be syntactically valid HTML, this presents a problem when you only want to manipulate a single attribute of an HTML tag. You can do this using attribute substitution, which looks very similar to the processing instruction syntax.

The most common method of attribute substitution is replacing the value of an HTML attribute with the value of a

---

Python expression. This can be accomplished by making the first character of the attribute the equals symbol ('=') as follows:

```
<body bgcolor="=color">
```

where the variable *color* has been previously defined in Python code. This is nearly identical to the behavior of processing instructions that begin with '=', except that the value is substituted for the tag's attribute.

There are cases where attributes do not take values, and are only present to indicate a certain condition. Examples of such attributes include `<COMPACT>`, `<DISABLED>`, `<NOWRAP>` and `<SELECTED>`. To conditionally include one of these attributes in a tag, begin the attribute value with a question mark ('?') as follows:

```
<option value="spam" selected="? selected_option == 'spam'">Spam</option>
```

Whatever follows the question mark must be a valid Python expression that evaluates to either `True` or `False`.

Note: In versions previous to 2.0, the question mark indicated that the value was to be evaluated as a Python statement. This behaviour was dubious as to its practical uses and is no longer supported by PSE.

4.3   Custom Tags

Custom tags are defined using the Custom Tags API, which is explained in the Developers manual. HTML designers can use custom tags just like any other HTML tag, resulting in a simple, easy to maintain template that doesn't require any knowledge of Python. Attributes of custom tags are interpreted as string arguments to the constructor of a class that defines the behavior of tag. If the attribute value begins with an equals symbol ('='), then the value will be interpreted instead as a python expression. The meaning of attributes is dependent on the custom tag class developed, so this varies from tag to tag within any given application.

That said, there is one attribute that is common to all custom tags, the `id` attribute. This is also a string that can be used to reference the object created by the tag from the Python code module. This is explained in detail in the Custom Tags API section of the Developers manual. If an `id` attribute is present, then the object is retrieved from an object cache if an object of the same `id` has been previously defined. If one hasn't been defined, then a new object will be used.

In this way an HTML designer can complete a template without having any supporting Python code module, and the template will be parsed, compiled and executed properly. The designer will be able to see instant results from his design without any knowledge of Python or what kind of code the programmer will write. Once a Python code module has been written by the programmer, the tag will be replaced by the object defined in the module, resulting in the correct output for the application.

"Anonymous" custom tags can be used; that is, a custom tag without an `id` attribute. In this case the resulting object will *not* be accessible from the Python code module, but this can be a useful tool for designing HTML pages that use a common look and feel that is determined programmatically.

Custom Tag objects have the following methods that can be used programmatically in the Python code module:

add(*thing*)
> Adds an arbitrary object *thing* to the things contained by the tag. Equivilant to nesting a custom tag that creates *thing* inside the current custom tag in the template file.

addString(*text*)
> Adds text to the contents of the tag. Equivilant to typing bare HTML inside the current custom tag in the template file.

There is a of restriction when using PSE template tags within the constructs of Custom Tags. If you include any

---

statements that will produce output using pse.write, print or sys.stdout.write or something similar (except for `<?=` `...` `?>` tags), then that output will appear before the tag output. An examination of the parsed servlet code will confirm this behavior. Also note that question mark '?' notation in Custom Tag attribute values is ignored, and the entire value is passed to the Custom Tag.

The PSE API

This chapter explains the Application Program Interface for PSE. There are three parts of the API that can be used to develop applications in PSE. The first is the `pse` module and plugin modules. The second is by creating your own plugin module that can be used as part of the `pse` module along side of the standard plugins. The third is using the Custom Tag API to allow HTML designers to build and use Python objects in HTML without having to learn any Python.

The standard `pse` and plugin modules are global and available without having to explicitly import them, although you can do so to import only certain parts or just from completeness of code sake. Third party plugins will also be available in the same manner.

The Custom Tag API is different in that it is read only once at startup time and is used by all requests. Therefore, you would want to explicitly import `pse` and any plugin module in your tag classes if you needed to use any attribute or method from those modules.

5.1 `pse` — The top level module

The `pse` module is a package that contains some useful functions and variables that are necessary for PSE servlets to run. Additionally, it contains the `plugins` package that contains all the plugin modules, and the `tags` module that contains any custom tag classes.

cache
> The `cache` variable controls output caching of the results of the servlet. The value of `cache` is a float that represents a timestamp of the same format returned by `time.time()`. When set to a time in the future, the output of the servlet will be cached until that time. Until then, the servlet will not be run again and only the cached output will be sent.
>
> Cached output is stored in the directory specified by the `OutputCacheDir` parameter in 'pse.conf'. By default, this is fileTMPDIR/pse_cache (where *TMPDIR* is your system's default temporary file location).
>
> **Note:** In order for caching to work, the web server process *must* have write access to the `OutputCacheDir` directory specified in the 'pse.conf' file. PSE will attempt to create this directory if it does not exist.

call (*filename,* $\big[$*name = value, ...* $\big]$)
> The `call` function can be used to execute an external servlet in a copy of the namespace of the currently running servlet. This has a similar effect of a nested scope, except that variable assignments will not affect the calling servlet's namespace. Any optional keyword arguments are variables to be added to the called servlet's namespace in addition to the caller's.
>
> Any output produced by the external servlet is returned by the `call` function, so the proper syntax should be:

```
<?=pse.call('filename.pt', name1 = value1, name2 = value2) ?>
```

The file is searched for on the path provided by the `IncludePath` parameter in the 'pse.conf' file. Any path information provided in the *filename* parameter will be ignored.

New in version 3.0.

copyright

A string containing the copyright information for PSE.

flush()

The first call to `flush` will send whatever HTTP headers have been set and flush output to the client connection. Subsequent calls will flush client output, but any operations that alter the HTTP headers will raise an exception.

New in version 3.0.

form

This is the same as `pse.plugins.request.form`.

Deprecated since release 3.0. You should reference `pse.plugins.request.form` instead of `pse.form` in your code, since *form* will no longer appear in the `pse` module in future versions of PSE. See section 5.2.3 for more information.

include(*filename*)

The `include` function can be used to execute an external servlet in the same namespace as the currently running servlet. Variable assignments will appear in the caller's namespace when the included servlet has finished executing. This call will be "nested" properly if used inside a servlet executed with `pse.call`.

Any output produced by the external servlet is returned by the `include` function, so the proper syntax should be:

```
<?=pse.include('filename.pt') ?>
```

The file is searched for on the path provided by the `IncludePath` parameter in the 'pse.conf' file. Any path information provided in the *filename* parameter will be ignored.

New in version 1.2.

Changed in version 2.0.

plugins

The pse.plugins package contains all the plugin modules that were loaded with PSE. See "Plugin Modules" below.

New in version 2.0.

setencoding([*encoding*])

The `setencoding` function can be used to change the default unicode encoding set by the `DefaultEncoding` parameter in the 'pse.conf' file for the currently running servlet. The *encoding* is specified using a valid encoding string. If *encoding* is omitted, the encoding will revert to the `DefaultEncoding` specified in the 'pse.conf' file.

New in version 3.0.

str(*value*)

The `str` function works just like Python's builtin `str` function/class, except that unicode strings are encoded using the `DefaultEncoding` specified in the 'pse.conf' file (or changed by calling `pse.setencoding`). In compiled servlets, expressions used in the `<?=expression ?>` construct gets passed through `pse.str` before being sent to the client as output via a `pse.write` call.

When explicitly producing output to the client, whether through `pse.write` or via `sys.stdout` or `print`, you should use the `pse.str` function to normalize client output. This is good practice in any case, since `pse.str` may do more work for you in future versions of PSE.

New in version 3.0.

tags

The pse.tags module is a module that contains all the custom tag classes defined in the module specified by the `TagHooksModule` option in the 'pse.conf' file. Normally all custom tag classes are available in the global namespace of the servlet, however if you want to be able to access the classes in modules that you import, they can be referenced here.

Any external modules that want to access the custom tags must import this module, using one of the following examples or something similar:

```
import pse.tags
from pse import tags
from pse.tags import *
```

The form used will determine how the local namespace is affected by the import. The last form shown above is normally used, since the `pse.tags` module contains *only* the custom tag classes that were defined by the progammer.

user_options

A dictionary of options set by the user in the 'pse.conf' file under the section UserOptions. You can pass customized configuration information from the 'pse.conf' file to the running servlet by assigning a value to an arbitrary configuration option under the UserOptions section. These options/values will be converted to a key/value pair in the `user_options` dictionary. Note that any option name will be converted to all lower case, regardless of the format in the configuration file. All values will be returned as strings.

version

A string containing the number of the currently running version of PSE.

write(*output*)

The `write` function is used internally by PSE to send output to the client. In compiled servlets, all template information is translated into a series of `pse.write` calls. This function should be used in servlets when you want to send output to the client, in lieu of using `sys.stdout` or `print`.

While those methods *can* be used to send output (most Python library modules use them), it will always be safer for your servlet to call `pse.write` instead, since `pse.write` may do more work for you in future versions of PSE

New in version 3.0.

5.2   Plugin Modules

The PSE plugin architecture allows you to add functionality to PSE applications through extension modules. Plugin modules are not normal modules in the sense that they are files that are imported, but instead they are created through a specific plugin API for every web request.

PSE comes with some standard plugins that are always available in any PSE application. These plugins are sub-modules contained in the `pse` package and do not need to be imported, just like the `pse` module, but they can be imported for completeness sake or to reference with an alias.

5.2.1   `pse.plugins.application` — persistent, per server application data

The `pse.plugins.application` module can be used to store information between requests on a per server basis. This module can be used to track persistent database connections or anything other data you want to reuse for every request.

Application data can be stored in a global namespace or a namespace of your choosing to prevent accidental name collision in different parts of the application or between different applications. Namespace identifiers can be any

hashable type for the purpose of dictionary keys. The only restriction is that namespaces cannot begin with the underscore character ('_').

Any data stored using the `pse.plugins.application` module is specific to a particular apache child process and is lost when the process dies, either because it has server the maximum number of requests, apache itself was shut down or restarted, or the process died from a segfault. Therefore, you cannot depend on the fact that every request to the application will see the same data, unless apache is configured to use only one child process.

new([*namespace*])
> Return a new application namespace dictionary. If *namespace* is specified, a persistent dictionary is returned. If the namespace has already previously been created, then that same dictionary will be returned, otherwise an empty dictionary will be returned.
>
> If *namespace* is omitted, then the global application namespace will be returned.

delete(*namespace*)
> Delete the specified namespace. If the namespace did not previously exist, then the `delete` function will silently do nothing. If you don't specify the namespace, the global namespace will be cleared. This is the same as calling the `clear` method on the global namespace dictionary itself.

5.2.2 `pse.plugins.client` — information about the browser/client

Generally speaking you can get all the client information you want from CGI variables, which are available in the `pse.plugins.request` module. The `pse.plugins.client` module gives you an easier and programmatically clearer way to access that information. Additionally, the `pse.plugins.client` provides an interface to manage cookies maintained by the browser.

address
> A tuple of the remote IP address and the port.

class CookieClass(*name, value, path, expires*)
> *name* is the name used for the cookie , and *value* is the value to set. The *path* argument defaults to `'/'`. The *expires* argument is the number of seconds the cookie should live, and a value of `None` (default) indicated that the cookie should live only for the current browsing session.
>
> A `CookieClass` object has the attributes `name`, `value`, `path`, `expires` and `modified`. Only `value`, `path`, and `expires` are writable. To discard a cookie, set `expires` to `0`. To rename a cookie, discard it and create a new CookieClass.
>
> **Note:** In versions prior to 3.0, `CookieClass` was named `CookieDef`, but since the class itself was not exposed to create new cookies, it shoudn't pose any upgrading problems.
>
> New in version 3.0.

cookies
> A dictionary containing `CookieClass` objects (see above). Upon initialization, the `cookies` dictionary contains all cookies that were sent from the browser. Since the browser does not specify the expiration of the cookie, the *expires* attribute will be initially set to `None`. The keys correspond to the cookie names provided by the browser. New cookies do not have to match the key, so duplicate cookie names can be created with different paths.
>
> Only cookies that have been modified are sent back to the browser at the end of the request. Therefore, if you modify a cookie sent by the browser, you should also alter the *expires* value, unless the cookie is to be discarded at the end of the browsing session.
>
> **Note:** In versions prior to 3.0, `CookieClass` was named `CookieDef`, but its interface was not available so that you could actually create new cookies. This has been fixed for 3.0, as well as a name change. Since the class itself was not exposed to create new cookies, it shoudn't pose any upgrading problems.
>
> New in version 2.0.
>
> Changed in version 3.0.

ip
> Similar to `address`, but it is a string containing only the IP address.

ident
> A string containing information obtained from the client using RFC1413 ident, if available.

password
> If an authorization challenge was issued for the request, then this is the password the user entered.

user_agent
> This is the browser's User Agent string. Note that this can be spoofed by some browsers.

username
> If an authorization challenge was issued for the request, then this is the username the user entered.

5.2.3 `pse.plugins.request` — information about the current request

The `pse.plugins.request` module provides you with information about the current request being processed.

CGI
> A dictionary containing all the values you would see as environment variables in a CGI program. In addition to the variables in the Common Gateway Interface specification, some apache-specific variables are also available. See the apache documentation for details.

args
> The arguments passed in the URL as part of the request. This is the same as `pse.plugins.request.CGI['HTTP_QUERY_STRING']`.

content
> The `content` file contains the raw HTTP POST data, if any, from the client request. This can be used for applications that require access to the raw HTTP POST data, such as XML-RPC.
>
> New in version 3.0.

id
> The unique identifier apache assigns to each request. If the client has HTTP Keep-Alive capability, then this identifier may be the same for multiple requests.
>
> **Note:** If using Apache 1.3, the mod_unique_id module must be loaded and active. If this module is not present, PSE will attempt to assign a unique identifier for the request, but multiple requests through the same connection using HTTP Keep-Alive will all receive different identifiers.

form
> A dictionary of form variables submitted to the page, by GET and/or POST methods. Each key in the dictionary is from the name of a form element from the calling script, and the value is either a string, list or file object. What kind of object is used is dependent on the nature of the form elements used.
>
> If there is only one form element with the same name, and it only has one value submitted, then the value is a string. Actually, this is a subclass of `str`, called PSEstr, which can be used as a normal Python `str` object in every way. It has the additional properties `str` and `list`, which return the string value as an actual string and a singleton list, respectively.
>
> If there are multiple elements with the same name or a single element with multiple values, then the value is a list. Actually, this is a subclass of `list`, called PSElist, which can be used as a normal `list` object in every way. It has the additional propertes `str` and `list`, which return the first element of the list as a string (or '' of the list length is 0) and a copy of the PSElist as an actual list, respectively.
>
> If the form encoding was multipart/form-data, and the form element was a file, then an open file is returned. Actually, this is a file-like object, called PSEfile, which can be used as a normal `file` object in every way. It has additional properties as follows. The `filename` property that is the file name from the form, which is also returned by the `str` property. The `list` property returns an iterator for the file, which can be used like a list, but does not read the entire file into memory at once.

The `form` attribute differs from the normal `cgi.FieldStorage` object in that values are automatically extracted and collected from the different `Field` objects. Also, blank values are always preserved, which result in empty strings.

By default `form` will never raise a `KeyError` for accessing an undefined key. Instead, it will return an empty string. However, the `has_key` function will still work as expected, returning `False` if the key was not defined in the form. You can change this behavior by setting `FormKeyError` to `True` in the 'pse.conf' file.

Changed in version 3.0.

method
>   The method used for the request, such as "GET" or "POST." This is the same as
>   `pse.plugins.request.CGI['REQUEST_METHOD']`.

mtime
>   The modification time of the resource being accessed as a long integer.

path_info
>   The information the appears after the path of the URI, but before the arguments of a the requested URI. This is the same as `pse.plugins.request.CGI['PATH_INFO']`.

protocol
>   The protocol being used for the request as a string. For example, `'HTTP/1.1'`.

uri
>   The path-only portion of the URI being requested.

url
>   The entire URI that was requested, including any path information and arguments.

5.2.4   `pse.plugins.response` — information that is returned in the response

content_type
>   By setting the `content_type`, you can control the mime type of the output of the servlet. This defaults to `'text/html'`.

add_header(*header, content*)
>   Add a response header to be sent to the client. *header* is the header you want to send without the trailing colon (':'), and *content* is the content of the header. Both *header* and *content* must be strings, so if you specify `'Content-Length'` as the header, you need to represent the content as a string, even though it is an integer number. The same *header* can be added multiple times, in which case multiple headers will be sent.
>
>   Calling this function after `pse.flush` has been called will raise a `HeadersSentError`.

authenticate(*[realm = 'PSE Application']*)
>   The `authenticate` function can be called to immediately deny access and call for Basic authentication from the client, without further processing of the servlet. The *realm* argument can be used to specify different authentication realms for different servlets in the application.
>
>   Calling this function after `pse.flush` has been called will raise a `HeadersSentError`.
>
>   New in version 3.0.

redirect(*url[, method = `pse.plugins.response.MOVED_TEMPORARILY`]*)
>   Immediatly quit running the servlet and redirect to a new URL. The value of *url* must be a valid URL, either internal or external, to redirect the browser.
>
>   The value of *method* must be one of `MOVED_TEMPORARILY`, `MOVED_PERMANENTLY`, `SEE_ALSO`, `TEMPORARY_REDIRECT` or `REFRESH`. The first four correspond to HTTP results codes; normally this should be `MOVED_TEMPORARILY`, but your needs may require a different response. The `REFRESH` method differs in that it inserts a Refresh header in the response to refresh the page location, with a delay of 0 seconds.
>
>   Calling this function after `pse.flush` has been called will raise a `HeadersSentError`.

Changed in version 3.0.

5.2.5 `pse.plugins.server` — information about the server process handling the request

As well as providing you with some basic information about the server, the `pse.plugins.server` module also gives you an interface to apache's logging facility.

address
> A tuple of the server's IP address and port.

child_num
> The number of the server's child process currently handling the request.

ip
> Similar to `address`, but it is a string containing only the IP address.

log_level
> The current logging level set for the server. This can be tested against the logging levels listed below.

log (*message*[, *level*])
> This function makes a log entry *message* in apache's ErrorLog file using *level* notification level. These levels are analogous to the internal apache notification levels.

The logging levels available for use in the `log` function and for comparison to `log_level` are as follows:

INFO
> This can be used to provide general information about the state of the application. This is the lowest non-debug logging level and is usually not logged by apache by default.

NOTICE
> This is just a normal log entry for notification of certain events that do not necessarily indicate an error. This is usually the lowest level that is logged by apache by default. This is the default level for the `log` function if no *level* is speficied.

WARNING
> This level can indicate a potential problem without necessarily indicating an error in the application.

ERROR
> This level is almost always logged by apache and indicates an error in the application.

DEBUG
> This is the lowest level that apache will log and usually has to be explicitly turned on in the apache configuration file to see messages logged at this level. This can be used to generate messages to aid in debugging an application that you would not normally see in a production environment.

5.2.6 `pse.plugins.session` — persistent, per user session data

The `pse.plugins.session` module can be used to store information between requests on a per user session basis. This module can be used to keep persistent data available for a specific user using cookies, even after the user has closed the browser and begun a new browsing session.

As with the `pse.plugins.application` module, session data can be stored in a global namespace or a namespace of your choosing to prevent accidental name collision in different parts of the application or between different applications. Namespace identifiers can be any hashable type for the purpose of dictionary keys. The only restriction is that namespaces cannot begin with the underscore character ('_').

The following values are configurable in the plugins.session section of the 'pse.conf' file:

CollectionProbability The chance in 1000 that an access to a session environment will initiate a garbage collection in the session database. Default value: `1`

CookieExpires The life of the session cookie since the last page access in minutes. Default value: `360`

DSN The psycopg DSN connect string of the database to store the session table. Default: `dbname=pse`

> **Note:** This is only for the psycopg session type. See `File` for other session types' parameters.

File The path of the file used to store the session database, which must be readable and writable by the apache process. If you want to destroy all user session information, simply delete this file. Note that for dbm sessions, `File` is actually a directory that holds the database and a Berkeley Database environment subsystem for locking support. Default: '/tmp/pse_session'

> **Note:** This option is only for the dbm and sqlite session types. See `DSN` and `Table` for psycopg parameters.

Table The name of the table that stores the session data. Default: `pse_session`

> **Note:** This is only for the psycopg session type. See `File` for other session types' parameters.

Timeout The minimum length of time a session will be stored, in minutes, in the database before being garbage collected. Default: `360`

Type The type of session back-end to use, which can be one of `dbm`, `sqlite` or `psycopg`. Default: `dbm`

> New in version 3.0.

Data can be stored either in a BSD-style DBM hash file, an SQLite database or a PostgreSQL database. The BSD-style DBM session type requires the bsddb module, which builds by default with Python. The SQLite session type requires at least version 3 of the SQLite libraries and version 2 of Python module (pysqlite2). The psycopg (PostgreSQL) type requires the `psycopg` module, available from http://initd.org/software/initd/psycopg. Whichever module is used, all have the same interface as follows:

new ([*namespace*])

> Return a new session namespace dictionary. If *namespace* is specified, a persistent dictionary is returned. If the namespace has already previously been created, then that same dictionary will be returned, otherwise an empty dictionary will be returned.
>
> If *namespace* is omitted, then the global session namespace will be returned.

delete (*namespace*)

> Delete the specified namespace. If the namespace did not previously exist, then the `delete` function will silently do nothing. You cannot delete the global namespace — use `del` on individual elements in the dictionary instead.

Developing with PSE

6.1 `pse_handler.api` — The PSE Handler API

The `pse_handler.api` module provides some interfaces to help you develop PSE extensions. It is usually safe to do:

```
from pse_handler.api import *
```

The members are described below.

apache
    This is the `mod_python.apache` module, imported for your convenience.

class CustomTag(*id, *arg, **kw*)
    This is the base class for all custom tags. Refer to Custom Tag API below for more information.

htmlquote(*text*)
    This convenience function takes a string argument and returns the HTML encoded equivalent, with special characters replaced by their entity codes.

log_traceback(*message, exc_info*)
    This function will log a traceback denoted by the *exc_info* tuple (as from `sys.exc_info()`) using the set PSE logging rules, prepended by the string in *message*.

    For example, the plugin loader uses this function to report errors. If *message* is `"plugin bad_plugin not imported due to error(s)"`, then the resulting output might be

```
[Wed Oct 19 17:38:51 2005] plugin bad_plugin not imported due to error(s): (traceback)
  File "/usr/lib/python2.4/site-packages/pse_handler/plugins/__init__.py", line
67, in load_plugins
    new_mod.initialize and new_mod.finalize
AttributeError: 'module' object has no attribute 'initialize'
```

write_error(*text,* [*level*])
    This function makes a log entry using the set PSE logging rules. *text* is the message to log, and *level* is the severity. The severity is one of the apache logging levels defined in `mod_python.apache`, which defaults to `mod_python.apache.APLOG_ERR`. Note that the `apache` module is also accessible in `pse_handler.api`.

6.2 Creating Plugins

Using the PSE plugin API, you can create your own modules that appear under the `pse.plugins` package for use in your applications. Some examples of user created plugins include:

- A new "session" module that uses an SQL database instead of DBM.

- A module to handle pooling of persistent database connections.

- A special Form handling module that can read and write form elements in different ways.

Since plugin modules have access to the original apache request through mod_python, you are only limited by the limits of mod_python itself when writing your own plugins.

6.2.1 Plugin Basics

Plugins are Python modules that reside in the 'pse/plugins' directory. Modules are loaded in lexigraphical order, so you can control the order in which plugins load by prepending a number value to the file name. The file name cannot start with the underscore ('_') character. You can easily temporarily disable a module from loading at all by prepending the file name with an underscore.

In order for the module to be used by PSE, the module has to define certain attributes and functions. These are described below.

Plugin Name

The name of the plugin as it appears in the pse.plugins package is determined by the global module variable *plugin_name*, which must be a string.

Note: In versions prior to PSE 3.0, the plugin name was derived from the file name of the plugin itself. While your PSE 2.x plugins will still function in PSE 3.0, you need to update your plugin to define *plugin_name* to be compatible with future versions of PSE.

New in version 3.0.

Plugin Initialization

The plugin initialization routine is only called once during the life of a PSE server process, when all the plugins are scanned and loaded by the PSE engine. In order to create an initialization routine, create a function with the following signature:

plugin_init()
> This function takes no arguments and can be used to initialize any internal structures needed for the plugin. Returns `True` upon successful initialization, `False` if the initialization failed (and the plugin will not appear in the `pse.plugins` package).

Note: The `plugin_init` function was previously named `init_plugin` in versions prior to PSE 3.0 and was optional. The `plugin_init` function is no longer optional as of PSE 3.0. While your PSE 2.x plugins will still function in PSE 3.0, you need to update your plugins to include a `plugin_init` function, and it needs to return either `True` or `False`.

Changed in version 3.0.

Module Initialization

The module initialize function is called before each servlet request is processed. This function initializes the plugin module that will be seen within the servlet. Create a function with the following signature:

initialize(*mod, req*)

> *mod* and *req* represent the module object and the apache request object respectively. The module object is an empty module with the name of the plugin to be initialized, if necessary, with all the properties and methods you will require to expose to servlets. The request object is defined in the mod_python documentation, provided to allow your plugin to interact with the apache request.

If the module does not contain an `initialize` function, then the plugin will not be loaded.

Module Finalization

The finalize function is called after each servlet is run, regardless of the exception status. This function can be used to perform actions after the servlet has run, and it can perform different actions based on the exception status of the servlet. Create a function with the following signature:

finalize(*mod, req, success*)

> *mod*, *req* and *success* represent the module object, the apache request object, and a boolean flag respectively. The module object is the same module object that was passed in the initialize function above. The request object is defined in the mod_python documentation, provided to allow your plugin to interact with the apache request. The success flag is true if there was no uncaught exception at the end of the servlet, false if there was an uncaught exception.

If the module does not contain a `finalize` function, then the plugin will not be loaded.

Using Parameters

To get parameter settings from the 'pse.conf' file for your module, create a dictionary named *plugin_config* in the global namespace of your module. The keys are the name of the configuration paramaters, and the values are the default values for the parameters. These will be replaced with values that appear in the `[plugins.pluginname]` section, where *pluginname* is the name of the plugin, from the 'pse.conf' file before the plugin is initialized. The values can be either a string, integer, float or list, and any value from the conf file will be required to be and converted into the type of the default value. Do not make a default value of None; instead, use `''`, `0`, `0.0` or `[]`.

Note: The *plugin_config* variable was previously named just *config* in versions prior to PSE 3.0; While your PSE 2.x plugins will still function in PSE 3.0, you need to update your plugin to use variable name *plugin_config* instead of *config*.

Changed in version 3.0.

There is a special "reserved" parameter for plugins called `Enabled`, which is by default set to true. You can disable a user plugin from showing up in the `pse.plugins` package by specifiying `Enabled = no` in the plugin's configuration section in the 'pse.conf' file.

6.2.2   Usage within Servlets

Custom plugins can be used the same way the builtin plugins are within running servlets. In fact, there is no difference between a builtin plugin and one you've made yourself, except for the fact that the builtin plugins came with PSE when you installed it.

If you cannot see your plugin within a servlet for some reason, check the apache error log for an ImportError exception and traceback.

6.2.3 Development Tips

To make functions for your module, use local function definitions in the `initialize` function, and assign them as members of *mod*.

Example:

```
plugin_name = 'example'

def plugin_init():
    return True

def initialize(mod, req):
    mod.last_result = None

    def method(a, b):
        mod.last_result = a + b
        return a + b

    mod.method = method

def finalize(mod, req, success):
    pass
```

In this example, method takes 2 parameters and returns the sum. It also stores the sum as the property last_result.

6.3 Custom Tag API

The Custom Tag API is a way for programmers to provide access to Python objects to HTML designers in their templates without having to know any Python code. The down side of this is that you will be introducing your own set of non-standard HTML tags into the markup, making it a little more difficult for transitioning projects to people who don't know what all your tags mean. On the other hand, it can be a very powerful tool to enable HTML designers to completely prototype an application without requiring Python code modules.

6.3.1 How Custom Tags Work

To understand how the Custom Tags API can be used, you need to know how the PSE parser handles custom tags. Whenver the opening tag of first custom tag is encountered in the template, the constructor for the CustomTag class is called, creating an object. When the parser reaches the closing tag, the parser causes code to be generated that displays the output of a `str()` on the object, i.e. the object's `__str__` method is called. The default `__str__` method returns a string composed by `joining` the `str()` of each object contained in the object's `_contents` attribute. What gets put in the `_contents` list is explained below.

If a custom tag object is contained inside another custom tag in the template, instead of printing the output of `__str__` when the closing tag is encountered, the object is passed to the containing custom tag object's `add` method. The default action of the `add` method is to add to the containing object's `_contents` list whatever was passed as an argument. The `add` method has the following signature:

add(*self, thing*)
    The argument *thing* is the custom tag object being passed to be added to the `self._contents` list.

If bare HTML is encountered inside the custom tag, then the object's `addString` method is called, which appends the string to the object's `_contents` list. The `addString` method has the following signature:

addString (*self, text*)
> The argument *text* is the bare HTML being added to the `self._contents` list.

To use the Custom Tag API, you need to subclass `CustomTag` from the `pse_handler.api` module and override the methods described above. Usually your overridden methods will eventually call the base methods.

Finally, when using the Custom Tag API you will usually want to override the `__init__` method to initialize the object. The `__init__` method has the following signature:

__init__ (*self, id = None, *arg, **kw*)
> *id* is the `id` attribute from the tag in the template file. It is also the way to refer to the same tag in the Python code module. In the base implementation, **arg* and ***kw* are ignored.
>
> **Note:** Parameters are passed from PSE templates in all lower case. Therefore, the named parameters for the constructor should all be in lower case.

Since objects created using the Custom Tag API are cached and reused, your overridden `__init__` method should look something like this:

```
def __init__(self, id = None, *arg, **kw):
    if self.needs_init:
        CustomTag.__init__(self, id)
        # put other initialization here
```

This way, if the custom tag object is created twice using the same *id*, the object will only be registered and initialized on the first call. You *could* allow overriding of certain attributes regardless of how many times the custom tag object was created, and that would appear outside the `if` statement. This is an advanced usage of the API and special care should be taken when doing so.

In order to make your custom tags usable in your application, you need to make a module that defines all your custom tag classes. By default, the corresponding tag in the template will have the same name as the class itself, but you can override this by setting the `_name` class attribute. In the 'pse.conf' file in the `[Parser]` section, set `TagHooksModule` to the full path of your custom tags module. The `TagHooksModule` parameter can be a list of modules if you want to use more than one module containing custom tag classes. You will need to restart apache for the changes to take effect.

Finally, sometimes it is desirable to suspend PSE's processing of data contained between the start and end of a custom tag within a PSE template. This can be done by setting the `_suspend_processing` class attribute to `False`. This value is `True` by default. New in version 3.0.

6.3.2 Custom Tag Example

Here is an example of a custom tag module that does the basic functionality of the HTML tags `<UL>` and `<LI>`.

```
from pse_handler.api import CustomTag

class List(CustomTag):

    # this is the same as the default, but this is how you could define
    # a different tag name from the class name
    _name = 'list'

    def __init__(self, id = None, type = 'square'):
        # this is similar to the <UL> tag but defaults to "square"
        if self.needs_init:
            CustomTag.__init__(self, id)
            self.type = type

    def add(self, thing):
        # only add bona fide Item objects
        if isinstance(thing, Item):
            CustomTag.add(self, thing)

    def addString(self, text):
        # ignore bare HTML not in an Item tag
        pass

    def __str__(self):
        # construct the unordered list html
        result = [ '<ul type="%s">' % self.type ]
        result.append(CustomTag.__str__(self))
        result.append('</ul>')
        return '\n'.join(result)

class Item(CustomTag):

    def __str__(self):
        # make a list item from the _contents
        result = [ '  <li>' ]
        result.append(CustomTag.__str__(self))
        result.append('</li>')
        return ''.join(result)
```

Sample usage is as follows:

```
<list>
<item>This is the first item</item>
<item>This is the second item</item>
</list>
```

which produces the following output:

```
<ul type="square">
  <li>This is the first item</li>
  <li>This is the second item</li>
</ul>
```

6.3.3   Helpful Hints for Developing Custom Tags

- Your custom tag module is imported when PSE itself is initialized, and is therefore a long-living module. Keep in mind that any modules imported globally will only be initialized *once* for each Apache process.

- The `pse` module (and all plugins) is only available when a servlet is running, so you cannot import it globally. Import `pse` and any plugins you want to access inside your custom tag methods.

- The easiest way to get all the content between your custom tag is to call `CustomTag.__str__(self)` in your `__str__` method. You can also manually loop through the `self._contents` list if you want to do something specific for each bit that was added.

- Don't forget that you can always override the methods `add` and `addString` to do what you want. For example, if you wanted to ignore nested custom tags, just do `pass` for `add`. Or, keep the bare HTML as a separate list in `addString`. You can always make a `super` call to add functionality while retaining the default actions of the `CustomTag` class.

- As an instructional aid, you may want to look at the '.pty' file that PSE generates to see how custom tags are converted to Python code.

6.4   Custom Exception Handling Hooks

Normally PSE will catch and display information about uncaught exceptions raised in servlets. You can control how and when this happens by setting options in the 'pse.conf' file, however sometimes it is desirable to act on uncaught exceptions in a customized manner. For example, you may want to automatically send an e-mail to an administrator or developer when servlets raise an uncaught exception. PSE provides a way to do this through exception hooks.

To create an exception hook, you must define two functions in an external module to handle exceptions that should be formatted in either HTML or text only. These functions must have the following signature:

text_traceback(*req, exception, traceback, frames*)

html_traceback(*req, exception, traceback, frames*)
    The *req* argument is the request object from mod_python. This can be used to find information about the request that may be helpful in providing debugging information. Please refer to the mod_python documentation for details.

    The *exception* argument is the exception only portion of the traceback, as a list, similar to the result of the `format_exception_only` function from the `traceback` module.

    The *traceback* argument is a list of "pre-processed" stack entries from the traceback, similar to teh result of the `extract_tb` function from the `traceback` module. However, only relevant stack entries are returned, which do not include references to internal PSE functions. Refer to the Python `traceback` module documentation for details.

    The *frames* argument is a list of `frame` objects, similar to the result of a call to the `getinnerframes` function from the `inspect` module. A frame object can be used to obtain information about each frame in the stack, such as the current function, module, line number, and source code (if available). Refer to the Python `inspect` module documentation for details.

    The return value is a string message to be sent to the browser, which should contain either HTML or plain text, as indicated by the function name.

The module *must* contain *both* functions for the hooks to be initialized by PSE. In the 'pse.conf' file in the `[Debug]` section, set `ExceptionHooksModule` to the full path of the module containing your custom exception hooks functions. You will need to restart apache for the changes to take effect.

Using the Command Line Utilities

PSE comes with some command line utilities to aid in development. The **psecompile** utility can compile and/or copy an entire directory tree, suitable for creating an optimized, binary only distribution of a PSE application. The **pserun** utility can be used to run servlets from the command line without the need of a web server or browser.

7.1 psecompile

Usage: **psecompile [OPTIONS] SOURCE [DESTINATION]**

Recursively compiles all PSE source files in a directory tree to optimized servlets. Normal Python modules that are not associated with a PSE template are compiled to '.pyo' files. Any other file is simply copied.

Optimized compiled servlets have a '.pto' extension. This helps to distinguish them from normal compiled templates ending in '.ptc'. If a '.pto' file exists, any other '.pt'/'.ptc' plus '.py' pair will be ignored, and only the '.pto' file will be executed, regardless of the relative timestamps. If you alter either of the servlet source files, you must either remove the compiled servlet or recompile it using the **psecompile** utility.

OPTIONS can be:

-h, --help This help message.

-c, --config=file Use a different pse.conf file than the default.

-v, --verbose Produce information about each file processed.

SOURCE is the source directory that contains '.pt' and '.py' files to be compiled.

DESTINATION is the destination directory for the compiled '.ptc' files. If the directory does not exist, then it will be created.

If **DESTINATION** is not specified, the **SOURCE** directory is used.

7.2 pserun

Usage: **pserun [OPTIONS] FILE**

Runs the specified servlet file as though it was requested through the web server, and prints the output.

OPTIONS can be:

-a,--args=QUERY_STRING Specify query arguments for the servlet.

-c,--config=FILE Use a different pse.conf file than the default.

-h,--help This help message.

-m,--method=GET—POST Specify the retrieval method for the servlet.

-n,--noheaders Do not output HTTP headers.

-o,--cookie=COOKIE Specify a Cookie according to RFC 2109.

-s,--sessionid=ID Specify session ID to simulate a PSE session.

FILE is the path to a servlet source file ('.pt' file), even if only the compiled version is available (as in HTTP requests).

If **--method=POST** is specified, then POST data will be read from standard input, which must be in application/x-www-form-urlencoded format.

PSE Changelog

Changelog for PSE version 3.0.6

Changes for version 3.0.6

- The exception text is now monospace, allowing syntax errors to be read correctly.

- Pre-compiled .pto files no longer return 404.

- psecompile fixes.

Changes for version 3.0.5

- Improved import functionality.

- It is now possible to disable non-essential pre-packaged plugins.

- Prevented FutureWarning messages from appearing in Python 2.3.

- Various unicode fixes.

- Minor code fixes/enhancements.

Changes for version 3.0.4

- Added TEMPORARY_REDIRECT (307) to response.redirect.

- Fixed possible failure to remove import hook on errors.

- Errors during import should now be properly reported instead of a "module not found" error.

- Fixed possible sys.modules contamination under high load.

- Fixed expiration of user cookies defined in client.cookies.

- Minor code fixes/enhancements.

Changes for version 3.0.3

- Fixed unicode handling inside custom tags.

- Fixed form parsing bug under certain circumstances.

- Fixed typo that prevented user_options from working.

- Importer fixes and improvements (better performance).

- Added an option to pserun to allow profiling of code.

Changes for version 3.0.2

- Minor bug fixes.

Changes for version 3.0.1

- FormKeyError was in the wrong place in the pse.conf file, and it didnt work anyway. It is now in the plugin.request section where it belongs.

- PSE will no longer die if a custom tag import fails.

- Session cookies now expire properly according to pse.conf.

- api.write_error will no longer fail if a string was not passed.

- psecompile will now successfully compile/copy symlinked directories.

Changes for version 3.0.0

- PSE now requires at least Python 2.3.1. Python 2.3.0 and older versions of Python are no longer supported.

- PSE is now thread safe for use with multithreaded apache 2.0 mpm modules. If you decide to use a multithreaded mpm, be sure your plugins, custom tags and other modules used in your applications are also thread safe.

- The name of the pse handler for mod_python has changed to pse_handler. See UPGRADING.TXT for more information.

- PSE no longer uses the PythonAccessHandler directive in the apache configuration file. You must delete these references.

- Authentication hooks have been removed from PSE. See the manual for use of the new pse.plugins.response.authenticate function.

- A new function pse.call() works like pse.include(), but it runs code in a separate namespace. Please see the PSE Manual for details.

- PSE handles unicode better than previous versions. The default encoding can be set with DefaultEncoding in pse.conf, which can be changed per servlet using the pse.setencoding() function. Please see the pse.conf file and the PSE Manual for details.

- A new funtion pse.flush() will flush client output immediately, but any operations to alter headers will then raise an exception.

- The plugin architecture has changed significantly. While plugins made for PSE 2.x will continue to work, much of the functionality is deprecated. See UPGRADING.TXT and the PSE Manual for details.

- The pse.conf file has been reorganized. Specifically, many options were moved to the Debug section. Please see the pse.conf file for details.

- User plugins can be placed in directories other than in the PSE distribution package directory; see pse.conf for details.

- The session plugin type can now be configured in the pse.conf file rather than manipulating the modules in the PSE plugins directory; see the PSE Manual for details.

- The SQLite session plugin now requires at least SQLite 3.0 and version 2.0 or higher of the Python bindings (pysqlite2).

- User plugins can be disabled in the pse.conf file by specifying "Enabled = no" in that plugin's configuration section.

- pse.plugins.response.redirect can now redirect requests in different ways. See the PSE Manual for details.

- Access to form data using pse.form is deprecated; use pse.plugins.request.form instead.

- The cookie interface in pse.plugins.client was broken in 2.x; CookieDef was not available to create new cookies. The class name has been changed to CookieClass and is now exposed to be able to create and store new cookies. See the PSE Manual for details.

- Custom tag classes can cause PSE to suspend its processing of text within the tag; see the PSE Manual for more information

- Raw HTTP POST data can now be obtained from pse.plugins.request.content, a file-like object, for use with XML-RPC or other applications.

- Unicode output is now passed through instead of being converted to a string.

- Cached output is now stored in a separate directory from the PSE template files; see pse.conf and the PSE Manual for details.

- Servlets compiled with the psecompile utility now compile to .pto files, not to be confused with the old "cache" .pto files.

- Tracebacks no longer contain internal PSE frames leading up to the running servlet's first frame.

- Added TraceAll to pse.conf to force PSE to attempt to show a traceback even when the source is not available.

- A new api module was added to aid in the development of plugins and custom tags. Please see the PSE Manual for details.

- Many bug fixes and code optimizations were completed.

- PSE includes an experimental CGI script that can be used when mod_python is not available. WARNING: THIS IS EXPERIMENTAL!! Please report bugs to pse.info@borko.org.

Changes for version 2.2.2

- Fixed negative line numbers in tracebacks.

- Tracebacks are collapsable in browsers supporting Javascript and CSS.

- Fixed file upload bug in forms.

Changes for version 2.2.1

- Fixed bugs in output caching and error reporting for includes.

- Fixed pse.include() to be able to include the compiled servlet if the original source .pt file is absent.

- Included debian package stuff, suitable for building debs for unstable(sid) debian.

Changes for version 2.2.0

- pse.conf is now expected to be in /etc/pse/ or in a similar location, depending on where Python is installed (e.g. /usr/local/etc if Python was installed in /usr/local), as opposed to being in the pse package directory.

- The compile utility has been renamed to psecompile.

- _dbhash-session.py no longer exists in plugins. It is now the default session.py modules.

- PSE will no longer log the perror string for most log entries.

- Fixed bug with ¡br /¿ being convertedto ¡br¿¡/br¿.

- The pserun utility no longer requires a fake _apache.py module.

- pse modules can be now imported outside of an apache environment.

- Added working setup.py to allow easy installation and source tarballs to be made.

- Added a Windows installer.

- Various documentation updates and general clean ups.

Changes for version 2.1.1

- Parser fix for tag attributes with empty strings as values.

- Corrected sample dbpool plugin.

- Updated Troubleshooting guide.

Changes for version 2.1.0

- Changes to bring the code up to date for use with mod_python 3.1.

- Uncaught exceptions (not apache.SERVER_RETURN) in plugin finalize routines now get reported.

- PSE now supports XML processing tags beginning with "¡?xml".

- Session cookies will now always be set for each request, regardless of error state.

- PSE has been tested with Apache on Windows. The dbhash session plugin has been modified to work in Windows. The Win32 extensions for Python must be installed to work correctly.

- The global session namespace can now be cleared by calling pse.session.delete with no parameters.

- For custom tag modules, tag_hooks is no longer required, and its use is now depricated. Please see the PSE Manual for more information.

- Form processing now results in PSEstr and PSElist objects instead of normal strings and lists. Please see the PSE Manual for more information.

- Numerous bug fixes and documentation updates.

- NEW: Added PreImport option in pse.conf to pre-import modules for each request (modules must be on sys.path), to reduce page response time.

- NEW: Compiled code memory caching can be controlled using CacheSize option in pse.conf.

- NEW: Accessing the pse.form dictionary with an invalid key no longer raises a KeyError by default. You can revert this by specifying FormKeyError=Yes in pse.conf. Please see the PSE Manual for more information.

Changes for version 2.0.0

- This is the first Open Source version of PSE.

- This version contains some MAJOR CHANGES that may break existing servlets written for PSE 1.2.x and below. !!!! Please read UPGRADING.TXT for more information. !!!!

- MAJOR CHANGE: The leading ? indicator templates from version 1.2 and below longer evaluates Python code in tag attribute values. Instead, it indicates a condition in which the attribute appears at all. This is for boolean attributes such as SELECTED and DISABLED. Please read the documentation for more details.

- MAJOR CHANGE: All plugins now appear in the pse.plugins package as opposed to appearing directly as sub-modules of the pse module

- MAJOR CHANGE: The pse.include() function now returns a string instead of printing output directly as a result to allow better integration with Custom Tags

- Custom tag magic has been changed to work better with includes; you must delete all compiled servlets and/or recompile them to take advantage

- Newer custom tag modules cause servlets to be recompiled

- Fixes to be compatible with apache 2 and mod_python 3

- The standard client plugin module now has the ability to manipulate browser cookies; see the manual for more information

- The compile utility has changed to work directly with a pse.conf file instead of specifying options and manipulating paths on the command line, use the –help option for more information

- A utility to run offline servlets has been added, pserun

- More examples were added to the contrib directory

- Many miscellaneous bug fixes

- Documentation updated to reflect the changes and new features of PSE 2.0

Changes for version 1.2.1

- TagHooksModule setting in pse.conf can be a list instead of only a single file

Changes for version 1.2.0

- Added the ability to have in-line execution of external servlets via the pse.include() function

- Added the ability to create authentication hooks in a custom Python module

- Added the ability to create custom exception handling hooks in a custom Python module

- Plugins that conflict with reserved attributes of the pse module will no longer load and overwrite those attributes

- Fixed a minor parsing bug that ate the last character in ¡? ... ?¿ tags

- The compile tool has been substantially upgraded in functionality and now includes a -h/–help option

- Documentation has been extensively updated, and now includes a chapter on installation and configuration and an index

- A 1.1 to 1.2 transition document was added to the documentation

- Distribution now has a contrib directory containing some sample plugins and other modules for use with PSE

Changes for version 1.1.1

- Fixed Syntax Error during parsing if the module file does not end with an EOL but ends in white space

Changes for version 1.1.0

- Made many documentation fixes, including an erroneous line in the License text. This change did not affect the license terms.

- Added ShowTrace configuration option to pse.conf to control display of tracebacks on errors

- You can no longer erroneously "outdent" to a negative indentation level

- Single ¡? : ?¿ expressions no longer re-indent, only "outdent"

- Prepending an equals symbol (=) to an attribute value of a custom tag will cause the value to be treated as a python expression instead of a string value

Changes for version 1.0.3

- Session modules now use SimpleCookie

Changes for version 1.0.2

- Plugins will not import if there are no initialize and finalize functions

- Developer documentation added

- Added more items to doc/Troubleshooting.txt

Changes for version 1.0.1

- Error from importing custom tag modules now shows traceback in apache log

- Session modules use BasicCookie instead of SmartCookie (security)

- Fixed HTML translation for tracebacks

Changes for version 1.0.0

- Official 1.0 release

Changes for version 0.3.7

- fixed bug where modules imported in custom tag modules would be deleted

- fixed parsing bug where would not be quoted in final output

Changes for version 0.3.6

- PSE no longer requires the Python Cryptography Toolkit

- Troubleshooting.txt added to the doc directory

- Debug code for PGSQL session module was accidently left in 0.3.5; removed

Changes for version 0.3.5

- fixed list parsing for config file

- removed redundant curdir in path, moved Path to the front of sys.path

- fixed sessions not being saved if pse.response.redirect is called

- from pse.tags import * now works in servlet modules

- miscellaneous minor bug fixes

Changes for version 0.3.4

- added the "tags" module to the pse module, containing custom tag classes

Changes for version 0.3.3

- added the ability to log tracebacks in a log file (see pse.conf)

- enhance and log HTML parse errors

Changes for version 0.3.2

- fixed column length of session key for sqlite

- added postgresql session module

- updated pse.conf to reflect config info for the pgsql session

Changes for version 0.3.1

- fixed devastating execution bug where servlet would not execute if the .py module was not present

Changes for version 0.3.0

- the content type is always set, regardless of the success of the servlet

- corrected parser errors from changes in custom tag parsing

- added custom traceback module, cgitb no longer used

Changes for version 0.2.0

- added Changelog to distribution

- white space no longer removed from bare data when adding to custom tags

- added hwaddr module, now required to validate license key

- license is now keyed to the host, requires license upgrade

Changes for version 0.1.5

- preliminary cache control via the response plugin

- More fixes and documentation of CustomTags

Changes for version 0.1.4

- Fixes for custom tag parsing in parser.py and error logging in config.py

Changes for version 0.1.3

- added user_options to pse module

- updated documentation

Changes for version 0.1.2

- .py files should now compile and load sanely

- fixed some errors and close session db after initial test

- set default values for all config variables

- comment out all config variables by default

- added init hook for plugin modules

- bail session modules if can't access db

- added form to the req structure for plugins to use

- moved id from client to request and made it work without mod_unique_id

- added ability to configure plugins from pse.conf

- added configuration variables to the session plugins

- updated sample pse.conf

- updated documentation

Changes for version 0.1.1

- documentation updated to aid installation

- fixed bug in logging

Changes for version 0.1.0

- Initial version

Copying and License

PSE and documentation is copyright ©2003–2005 by Nicholas D. Borko and Christian Höltje. The software is distributed via the LGPL license. A copy of this license is included below as well as being including with the source and packages.

LGPL, LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

Copyright © 1981, 1999 Free Software Foundation, Inc.

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the Lesser General Public License because it does *Less* to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) The modified work must itself be a software library.

   (b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   (c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   (d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

   In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

(a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

(b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

(c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

(d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

(e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   (a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   (b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

INDEX

Symbols

`__init__()` (in module), 27

A

`add()` (in module), 13, 26
`add_header()` (in module pse.plugins.response), 20
`address`
    data in pse.plugins.client, 18
    data in pse.plugins.server, 21
`addString()` (in module), 13, 27
`apache` (data in pse_handler.api), 23
`args` (data in pse.plugins.request), 19
`authenticate()` (in module pse.plugins.response), 20

C

`cache` (data in pse), 15
`call()` (in module pse), 15
`CGI` (data in pse.plugins.request), 19
`child_num` (data in pse.plugins.server), 21
`content` (data in pse.plugins.request), 19
`content_type` (data in pse.plugins.response), 20
`CookieClass` (class in pse.plugins.client), 18
`cookies` (data in pse.plugins.client), 18
`copyright` (data in pse), 16
`CustomTag` (class in pse_handler.api), 23

D

`DEBUG` (data in pse.plugins.server), 21
`delete()`
    in module pse.plugins.application, 18
    in module pse.plugins.session, 22

E

`ERROR` (data in pse.plugins.server), 21

F

`finalize()` (in module), 25
`flush()` (in module pse), 16
`form`
    data in pse, 16
    data in pse.plugins.request, 19

H

`html_traceback()` (in module), 29
`htmlquote()` (in module pse_handler.api), 23

I

`id` (data in pse.plugins.request), 19
`ident` (data in pse.plugins.client), 19
`include()` (in module pse), 16
`INFO` (data in pse.plugins.server), 21
`initialize()` (in module), 25
`ip`
    data in pse.plugins.client, 19
    data in pse.plugins.server, 21

L

`log()` (in module pse.plugins.server), 21
`log_level` (data in pse.plugins.server), 21
`log_traceback()` (in module pse_handler.api), 23

M

`method` (data in pse.plugins.request), 20
`mtime` (data in pse.plugins.request), 20

N

`new()`
    in module pse.plugins.application, 18
    in module pse.plugins.session, 22
`NOTICE` (data in pse.plugins.server), 21

P

`password` (data in pse.plugins.client), 19
`path_info` (data in pse.plugins.request), 20
`plugin_init()` (in module), 24
`plugins` (data in pse), 16
`protocol` (data in pse.plugins.request), 20
`pse` (extension module), **15**
`pse.plugins.application` (extension module), **17**
`pse.plugins.client` (extension module), **18**