CherryPy Standard Library Reference

Release 0.10

Remi Delon

March 19, 2004

Email: remi@cherrypy.org

Abstract

CherryPy is a Python-based tool for developing dynamic web sites.

One of CherryPy's main feature is **CherryClass**: an extension to Python's regular classes. A CherryClass is a self-contained module that can contain both the algorithm to process the data and the templates to render the result.

CherryClasses are very powerful and easily reusable. CherryPy comes with a set of useful CherryClasses that will make your life easier.

These CherryClasses make the **CherryPy Standard Library**.

This standard library is still limited for now, but it is already very useful. We expect it to get much bigger very soon.

CONTENTS

1 Module list **1**
1.1 `Mail` — Simple smtplib wrapper to send e-mails. 1
1.2 `HttpAuthenticate` — Basic HTTP authentication. 2
1.3 `CookieAuthenticate` — Cookie-based authentication. 3
1.4 `CookieSessionAuthenticate` — Cookie and session-based authentication. 5
1.5 `Form` — Form handling. 8
1.6 `MySql` — Simple MySQLdb wrapper to access a MySql database. 14
1.7 `MaskTools` — Simple HTML patterns. 14

A History and License **17**
A.1 License 17

CHAPTER

ONE

Module list

1.1 `Mail` — Simple smtplib wrapper to send e-mails.

This module is a very simple module (the source code is only 20 lines) that allows you to send e-mails from your CherryPy program.

The module defines an abstract CherryClass called `Mail`, with one member variable called *smtpServer* and one method called *sendMail*.

To use it, just derive the *Mail* CherryClass, set *smtpServer* in the *__init__* method, and then call *sendMail* to send an e-mail:

variable: smtpServer
String containing the address of the Smtp server

function: sendMail(*sender, receiver, bcc, contentType, subject, msg*)
This functions sends an e-mail according to the parameters. All parameters must be a string. *contentType* should be either ”text/plain” or ”text/html”. Depending on *contentType*, *msg* should contain either plain text or html text. This functions uses Python’s *smtplib* library to send the e-mail. It uses the value of *smtpServer* to send the email.

function: sendHtmlMail(*sender, receiver, bcc, subject, txtmsg, htmlmsg*)
This functions sends an HTML e-mail according to the parameters. All parameters must be a string. This functions uses Python’s *smtplib* and *MimeWriter* modules to send the e-mail. It uses the value of *smtpServer* to send the email.

Exemple:

```
use Mail
CherryClass MyMail(Mail):
function:
    def __init__(self):
        self.smtpServer='smtp.site.com'
CherryClass Root:
mask:
    def index(self):
        <py-exec="myMail.sendMail('me@site.com', 'you@yourhost.com', '', 'text/plain', 'Hello',
        <html><body>
            Hi, I just sent an e-mail to you@yourhost.com
        </body></html>
```

1.2 `HttpAuthenticate` — Basic HTTP authentication.

1.2.1 Module

This module allows you to protect a part of your website with a login and a password, using a basic HTTP authentication scheme.

All you have to do is declare a CherryClass that inherits from *HttpAuthenticate*, and all your masks and views will be automatically protected.

To perform this magic, *HttpAuthenticate* uses AOP (aspect oriented programming). This basically means that it will add some extra code at the beginning of each of your masks and views.

You may override the following methods:

`function: getPasswordListForLogin`(*login*)
: This is where you specify what the valid login/password combinations are. The input value is the login that the user entered. The method should return a list of all valid passwords for this login. If the login is incorrect, just return an empty list.

  Note: Being able to return several matching passwords for a login allows you to keep a ”master key” password that works with all logins.

`mask or view: unauthorized`()
: This is the page that is displayed when the user entered an incorrect login/password 3 times in a row.

`variable: login`
: String containing the login of the user that is logged in. The string is empty if no-one is logged in.

Note: There is no ”logout” method. Users are automatically logged out when they close their browser window.

See Also:

Module `CookieAuthenticate` (section 1.3):
: Cookie-based authentication.

Module `CookieSessionAuthenticate` (section 1.4):
: Cookie/session-based authentication.

1.2.2 Example

The following code is an exemple that uses the HttpAuthenticate module:

```
use HttpAuthenticate
CherryClass Root(HttpAuthenticate):
function:
    def getPasswordListForLogin(login):
        if login==’mySecretLogin’: return [’mySecretPassword’]
        return []
mask:
    def index(self):
        <html><body>
            Hello <py-eval="self.login">, I see you know the secret login and password ...
        </body></html>
    def unauthorized(self):
        <html><body>
            Hey dude, get out ! You’re not allowed here if you don’t know the login/password
        </body></html>
```

1.3 `CookieAuthenticate` — Cookie-based authentication.

1.3.1 Module

A cookie-based authentication allows website users to login/logout using a username and a password.

While they are logged in, their session information is stored on their computer via a cookie.

If they are inactive for too long, they are automatically logged out.

This module provides an easy to use implementation of a cookie-based authentication.

Unlike many cookie-based authentication methods, it doesn't require any database on the server side to store session informations. It uses three cookies to store the session information:

- One cookie called *CherryLogin* that contains the login of the user
- One cookie called *CherryDate* that contains the time of the last action
- One cookie called *CherryPassword* that contains the password of the user, encrypted with the login and the time of the last action. This is to prevent someone from manually changing the last action time.

To use this module, you have to declare a CherryClass that inherits from *CookieAuthenticate*, and all your masks and views will be automatically protected.

To perform this magic, *CookieAuthenticate* uses AOP (aspect oriented programming). This basically means that it will add some extra code at the beginning of each of your masks and views.

You may use the following variables and methods:

`variable: loginCookieName`
String containing the name of the cookie where the *login* is stored. (default value is *CherryLogin*)

`variable: dateCookieName`
String containing the name of the cookie where the *last action time* is stored. (default value is *CherryDate*)

`variable: passwordCookieName`
String containing the name of the cookie where the *password* is stored. (default value is *CherryPassword*)

`variable: timeout`
Integers containing the timeout in minutes. If the user is inactive for that time, it will automatically be logged out. Default value ie 60. Set it to 0 if you want no timeout.

`function: getPasswordListForLogin`(*login*)
This is where you specify what the valid login/password combinations are. The input value is the login that the user entered. The method should return a list of all valid passwords for this login. If the login is incorrect, just return an empty list.

Note: Being able to return several matching passwords for a login allows you to keep a "master key" password that works with all logins.

`mask or view: loginScreen`(*message, fromPage, login=''*)
This is the page that is displayed when the user tries to access a protected page without being logged in.

message is a string containing the reason why no user is logged in. Possible values are:

- **timeoutMessage**: This means that someone was logged in, but they remained inactive for too long
- **wrongLoginPasswordMessage**: This means that someone is trying to log in, but the login and password they entered are incorrect
- **noCookieMessage**: This means that no informations are available: this is probably the first time the user is coming here

fromPage is a string containing the URL of the page the user was trying to access.

login is a string containing the login of the user if any. If the string is not empty, it means that the user already entered a login, but the password was incorrect, or that the user had a cookie with the login in it. This allows to display the login in the form so the user doesn't have to enter it each time.

The CherryClass comes with a default *loginScreen* mask. You'll probably want to overwrite it to customize it for your needs. All you have to do is define a form that calls the *doLogin* method with 3 parameters: *login*, *password* and *fromPage*. The first two are entered by the user. The third one should be a hidden field with the value that's passed to the function.

The following code is the default implementation of the *loginScreen* mask:

```
<html><body>
    Message: <div py-eval="message">message</div>
    <form method="post" action="doLogin">
        Login: <input type=text name=login py-attr="login" value="" length=10><br>
        Password: <input type=password name=password length=10><br>
        <input type=hidden name=fromPage py-attr="fromPage" value=""><br>
        <input type=submit>
    </form>
</body></html>
```

mask or view: logoutScreen()
This page is displayed after the user logged out. This method is called by the *doLogout* method. You may overwrite it to suit your needs.

view: doLogout()
This is the mask or view you should call to perform a logout. This method performs the logout, and then calls the *logoutScreen* method to display the logout screen.

variable: login
String containing the login of the user that is logged in. The string is empty if no-one is logged in.

See Also:

Module `CookieSessionAuthenticate` (section 1.4):
Cookie/session-based authentication.

Module `HttpAuthenticate` (section 1.2):
Basic HTTP authentication.

1.3.2 Example

The following code is an exemple that uses the CookieAuthenticate module:

```
use CookieAuthenticate

CherryClass MemberArea(CookieAuthenticate):
mask:
    def index(self):
        <html><body>
        Welcome to the member area, <py-eval="self.login"><br>
        If you want to log out, just click <a py-attr="self.getPath()+'/doLogout'" href="">here
        Otherwise, just click <a py-attr="request.base" href="">here</a> to go back to the home
        </body></html>
    def loginScreen(self, message, fromPage, login=''):
        <html><body>
            Welcome to the login page. Please enter your login and password below:
            <py-if="message==self.wrongLoginPasswordMessage">
                <br><font color=red>Sorry, the login or password was incorrect</font>
            </py-if>
            <form method="post" action="doLogin">
                Login: <input type=text name=login py-attr="login" value="" length=10><br>
                Password: <input type=password name=password length=10><br>
                <input type=hidden name=fromPage py-attr="fromPage" value=""><br>
                <input type=submit value="Login">
            </form>
        </body></html>
    def logoutScreen(self):
        <html><body>
            You have been logged out.<br>
            Click <a py-attr="request.base" href="">here</a> to go back to the home page.
        </body></html>
function:
    def getPasswordListForLogin(self, login):
        if login=="login": return ["password"]
        return []

CherryClass Root:
mask:
    def index(self):
        <html><body>
            Welcome to the site.<br>
            Click <a href='memberArea/index'>here</a> to access the
            member area.
        </body></html>
```

1.4 `CookieSessionAuthenticate` — Cookie and session-based authentication.

1.4.1 Module

A cookie-based authentication allows website users to login/logout using a username and a password.

While they are logged in, their session information is stored on their computer via a cookie.

If they are inactive for too long, they are automatically logged out.

This module provides an easy to use implementation of a cookie-based authentication.

This module is quite different from the *CookieAuthenticate* module because the login/password is only checked once (when the user first loggs in) and then the fact that this user is logged in is stored as a session.

To use this module, you have to declare a CherryClass that inherits from *CookieSessionAuthenticate*, and all your masks and views will be automatically protected.

To perform this magic, *CookieSessionAuthenticate* uses AOP (aspect oriented programming). This basically means that it will add some extra code at the beginning of each of your masks and views.

You may use the following variables and methods:

`variable: sessionIdCookieName`
: String containing the name of the cookie where the *login/session* informations are stored. (default value is *CherrySessionId*)

`variable: timeout`
: Integers containing the timeout in minutes. If the user is inactive for that time, it will automatically be logged out. Default value ie 60. Set it to 0 if you want no timeout.

`function: checkLoginAndPassword`(*login, password*)
: This is where you specify what the valid login/password combinations are. This method should return None if the login/password are ok and an error message such as ”Wrong login/password” or ”Account disabled” if the login/password are not ok.

`mask or view: loginScreen`(*message, fromPage, login=”*)
: This is the page that is displayed when the user tries to access a protected page without being logged in.

  message is a string containing the reason why no user is logged in. Possible values are:

  - **timeoutMessage**: This means that someone was logged in, but they remained inactive for too long
  - **wrongLoginPasswordMessage**: This means that someone is trying to log in, but the login and password they entered are incorrect
  - **noCookieMessage**: This means that no informations are available: this is probably the first time the user is coming here

  fromPage is a string containing the URL of the page the user was trying to access.

  login is a string containing the login of the user if any. If the string is not empty, it means that the user already entered a login, but the password was incorrect, or that the user had a cookie with the login in it. This allows to display the login in the form so the user doesn’t have to enter it each time.

  The CherryClass comes with a default *loginScreen* mask. You’ll probably want to overwrite it to customize it for your needs. All you have to do is define a form that calls the *doLogin* method with 3 parameters: *login*, *password* and *fromPage*. The first two are entered by the user. The third one should be a hidden field with the value that’s passed to the function.

  The following code is the default implementation of the *loginScreen* mask:

```
<html><body>
    Message: <div py-eval="message">message</div>
    <form method="post" action="doLogin">
        Login: <input type=text name=login py-attr="login" value="" length=10><br>
        Password: <input type=password name=password length=10><br>
        <input type=hidden name=fromPage py-attr="fromPage" value=""><br>
        <input type=submit>
    </form>
</body></html>
```

`mask or view: logoutScreen`()
: This page is displayed after the user logged out. This method is called by the *doLogout* method. You may overwrite it to suit your needs.

`view: doLogout()`
: This is the mask or view you should call to perform a logout. This method performs the logout, and then calls the *logoutScreen* method to display the logout screen.

`variable: request.login`
: String containing the login of the user that is logged in. The string is empty if no-one is logged in. The reason this is stored in the *request* global variable is to make it thread-safe.

See Also:

Module `CookieAuthenticate` (section 1.3):
: Cookie-based authentication.

Module `HttpAuthenticate` (section 1.2):
: Basic HTTP authentication.

1.4.2 Example

The following code is an exemple that uses the CookieAuthenticate module:

```
use CookieSessionAuthenticate

CherryClass MemberArea(CookieSessionAuthenticate):
mask:
    def index(self):
        <html><body>
        Welcome to the member area, <py-eval="request.login"><br>
        If you want to log out, just click <a py-attr="self.getPath()+'/doLogout'" href="">here
        Otherwise, just click <a py-attr="request.base" href="">here</a> to go back to the home
        </body></html>
    def loginScreen(self, message, fromPage, login=''):
        <html><body>
            Welcome to the login page. Please enter your login and password below:
            <py-if="message==self.wrongLoginPasswordMessage">
                <br><font color=red>Sorry, the login or password was incorrect</font>
            </py-if>
            <form method="post" action="doLogin">
                Login: <input type=text name=login py-attr="login" value="" length=10><br>
                Password: <input type=password name=password length=10><br>
                <input type=hidden name=fromPage py-attr="fromPage" value=""><br>
                <input type=submit value="Login">
            </form>
        </body></html>
    def logoutScreen(self):
        <html><body>
            You have been logged out.<br>
            Click <a py-attr="request.base" href="">here</a> to go back to the home page.
        </body></html>
function:
    def checkLoginAndPassword(self, login, password):
        if login == 'login' and password == 'password': return
        else: return "Wrong login/password"


CherryClass Root:
mask:
    def index(self):
        <html><body>
            Welcome to the site.<br>
            Click <a href='memberArea/index'>here</a> to access the
            member area.
        </body></html>
```

Note that you need to enable sessions in your configuration file. For instance, if you want to have session data stored in RAM, you need to put this in your config file:

```
[session]
storageType = ram
```

1.5 `Form` — Form handling.

1.5.1 Introduction

Handling complicated forms can really be a pain sometimes, especially if you want to handle user errors.

The Form module can save you a lot of time and trouble, once you've learned how to use it.

Most of the time, you'll want this:

- Your form has all sorts of fields: text fields, textareas, checkboxes, radio buttons, ...
- By default, some fields are empty, and some have default values.
- Some fields are mandatory, some aren't. Some fields can only have certain values (ex: birthdate, price, ...)

And you'll probably want your form to behave like this:

- When the form is first displayed, all fields are either empty or they have a default value
- The user fills the form in and hit the submit button
- (At this point, you may want to use a few lines of javascript to catch trivial errors. But if your form is really big, you'll probably want to catch these errors on the server side ...)
- The data is sent to the server, which analyzes it
- If the data is correct (no missing field, no wrong value, ...), everything continues normally
- In case some fields have incorrect values, you'll probably want the following:
  - Redisplay the form, but keep all values that the user entered (that's the painful part ...)
  - Display a message that stands out at the top of the form to notify the user that some fields need to be changed
  - Display a message next to each field that has an error

We'll see how the Form module can help you do that ...

1.5.2 Module

This module defines 4 CherryClasses:

FormField

A FormField instance is used for each of the form fields.

function: `__init__`(*label, name, typ, mask=None, mandatory=0, size=15, optionList=[], defaultValue='', validate=None*)

label is a string that will be displayed next to the field.

name is a string containing the name of the field.

typ is a string containing the type of the field. It can be one of the following: text, password, file, hidden, submit, image, select, textarea, radio, checkbox

mask is a mask used to render the field. The default value is *defaultFormMask.defaultMask*. The mask will receive the FormField instance as an argument and it should return some HTML to render the field.

mandatory is an integer that indicates whether the field is mandatory or not.

size is an integer that indicates the size of the field.

mandatory is an integer that indicates whether the field is mandatory or not (it is only used for some of the fields like text or password).

optionList is a list of strings containing the different options for a fied (is is only used for radio and checkbox fields).

defaultValue is a string containing the default value for the field.

validate is a function used to validate the field. The function will receive the value of the field as an argument, and it should return *None* if the value is correct, or a string containing the error message if the value is not.

FormSeparator

A FormSeparator instance is used to display some text or images between the different fields of the form.

function: **__init__**(*label, mask*)

label is a string that will be used by the mask to know what to display.

mask is a mask used to render the field. The mask will receive the FormSeparator instance as an argument and it should return some HTML to render the separator.

DefaultFormMask

This CherryClass contains a default implementation of a mask for the fields. You'll probably want to use your own masks for your own design. The next section explains how to write your own field masks.

Form

The is the main CherryClass of the module. To create a form, you should declare a CherryClass that inherits from Form.

You may use the following variables and methods:

variable: **method**

String containing the *method* attribute of the form tag. It may be *send* or *post*. The default value is *post*

variable: **enctype**

String containing the *enctype* attribute of the form tag. For instance, for a form that allows the user to upload files, you would use *multipart/form-data* The default value is an empty string, which means that the *enctype* attribute wile be omitted.

variable: **fieldList**

List containing instances of the FormField and FormInstance CherryClasses. This list determines which fields and separators will be displayed, and in which order. *fieldList* should be set in the *__init__* method of the CherryClass.

function: **formView**(*leaveValues=0*)

This function returns the HTML code for the form. if *leaveValues* is false, it will use the default value for each of the fields. If *leaveValues* is true, it will use the values that are in *request.paramMap* (in other words, the values that were entered by the user)

function: **validateFields**()

This function should be overwritten if you need to perform some validation that involves several fields at the same time (for instance, checking that 2 passwords match).

If a field has an error, the function should set the *errorMessage* member variable of the FormField instance.

function: **setFieldErrorMessage**(*fieldName, errorMessage*)

Sets the *errorMessage* member variable of the FormField instance whose name is *fieldName*.

function: getFieldOptionList(*fieldName*)
Returns the *optionList* member variable of the FormField instance whose name is *fieldName*.

function: getFieldDefaultValue(*fieldName*)
Returns the *defaultValue* member variable of the FormField instance whose name is *fieldName*.

function: setFieldDefaultValue(*fieldName, defaultValue*)
Sets the *defaultValue* member variable of the FormField instance whose name is *fieldName*.

function: getFieldNameList(*exceptList=[]*)
Returns the list of field names, based on the *fieldList* member variable. Names that are in *exceptList* are omitted.

function: validateForm()
This function checks if the data that the user entered is correct or not. It returns 1 if it is, 0 otherwise.

view: postForm(***kw*)
This view is automatically called when the user submits the form. You should overwrite this view and add your own code to handle the form data. Typical code for this view looks like this:

```
def postForm(self, **kw):
    if self.validateForm():
        # Yes, the data is correct
        # Do what you want here
        pass
    else:
        # No, the data is incorrect
        # Redisplay the form and tell the user to fix the errors:
        return "<html><body><font color=red>Fill out missing fields</font>"+self.formView(1
```

1.5.3 Writing a form mask

The module comes with a default mask for forms, but you'll probably want to change it to use your own design. All you have to do is write your own form mask.

A form mask takes a FormField instance as an input and returns some HTML code as output. Don't forget that your mask should be setting the value of the field according the the *currentValue* member variable. Moreover, it should handle the field differently if the *errorMessage* is set.

For instance, a mask for a text field could look like this:

```
if field.typ=='text':
    result='%s: <input type=text name="%s" value="%s" size="%s">'%(
        field.label, field.name, field.currentValue, field.size)
    if field.errorMessage:
        result+=' <font color=red>%s</font>'%field.errorMessage
return result+'<br>'
```

Things are a bit trickier for select boxes, radio buttons or checkboxes because you have to loop over the *optionList* member variable and match each value against *currentValue*.

For instance, for a select box, the mask could look like this:

```
if field.typ==’select’:
    result=’%s: <select name="%s" size="%s">’%(field.label, field.name, field.size)
    for optionValue in optionList:
        if optionValue==field.currentValue: checked=’ checked’
        else: checked=’’
        result+=’<option%s>%s</option>’%(checked,optionValue)
    result+=’</select>
    if field.errorMessage:
        result+=’ <font color=red>%s</font>’%field.errorMessage
return result+’<br>’
```

1.5.4 Putting it together

Let’s see how we use all these CherryClasses, variables and methods to build a nice form.

We are going to build a form where users choose a login and a password, enter their e-mail, their country and their hobbies.

We need 6 fields:

- One text field for the login (this field is mandatory)
- Two password fields for the password (which they must enter twice)
- One text field for their e-mail (this field is optional)
- One select field for their country (the default value is USA)
- One checkbox list for their hobbies (this field is optional)

Plus we’ll add one line between the e-mail field and the country field.

Here is what the code could be:

```
use Form, MaskTools

# We start by creating a CherryClass that inherits from Form
# This CherryClass will hold all the informations about the form we want to create
CherryClass MyForm(Form):
function:
    def __init__(self):
        # Instantiate all fields plus 3 separators (one at the beginning, one for the line and
        headerSep=FormSeparator('', defaultFormMask.defaultHeader)
        login=FormField(label='Login:', name='login', mandatory=1, typ='text')
        password=FormField(label='Password:', name='password', mandatory=1, typ='password')
        password2=FormField(label='Confirm password:', name='password2', mandatory=1, typ='pass
        email=FormField(label='E-mail:', name='email', typ='text', validate=self.validateEmail)
        lineSep=FormSeparator('', self.lineSeparator)
        country=FormField(label='Country:', name='country', typ='select', optionList=['USA', 'A
        hobbies=FormField(label='Hobbies:', name='hobbies', typ='checkbox', optionList=['Using
        submit=FormField(label='', name='Submit', typ='submit')
        footerSep=FormSeparator('', defaultFormMask.defaultFooter)
        self.fieldList=[headerSep, login, password, password2, email, lineSep, country, hobbies

    # Function that checks if an e-mail is correct or not
    def validateEmail(self, email):
        try:
            before, after=email.split('@')
            if not before or after.find('.')==-1: raise 'Error'
        except: return "Wrong email"

    # Function that performs general validation of the form. In our case, we need to check
    # that the passwords match
    def validateFields(self):
        # Warning: paramMap could have no "password" or "password2" key if the user didn't fill
        if request.paramMap.get('password','')!=request.paramMap.get('password2',''):
            # Set errorMessage for password fields
            self.setFieldErrorMessage('password', 'Not matching')
            self.setFieldErrorMessage('password2', 'Not matching')

mask:
    # Line separator used to draw a line between the email field and the country field
    def lineSeparator(self, label):
        <tr><td colspan=3 height=1 bgColor=black py-eval="maskTools.x()"></td></tr>

view:
    def postForm(self, **kw):
        if self.validateForm():
            return root.formOk()
        else:
            return "<html><body><font color=red>Please correct the errors (fields in red)</font


# Now we just have to create a regular Root CherryClass, that will call some of MyForm's method
CherryClass Root:
mask:
    def index(self):
        <html><body>
            Welcome, please fill out the form below:
            <py-eval="myForm.formView()">
        </body></html>
    def formOk(self):
        <html><body>
            Thank you for filling out the form.<br>
            and values were correct
        </body></html>
```

1.6 `MySql` — Simple MySQLdb wrapper to access a MySql database.

This module is a very simple module. The source code is the following:

```
import MySQLdb

#################
CherryClass MySql abstract:
#################
function:
    def openConnection(self, host, user, passwd, db):
        self.connection=MySQLdb.connect(host, user, passwd, db)
    def query(self, query):
        c=self.connection.cursor()
        c.execute(query)
        res=c.fetchall()
        c.close()
        return res
```

All it does is it provides a CherryClass wrapper to the Python MySQLdb module

All you have to do to use it is declare a CherryClass that inherits from MySql, call the *openConnection* method in the *__init__* method, and use *query* to execute a query and get the result.

The connection will be automatically opened when the server gets started (when your CherryClass gets instantiated), and it will remain open until the server dies.

The following code is an example on how to use the module:

```
use MySql
CherryClass MyDb(MySql):
function:
    def __init__(self):
        self.openConnection('host', 'user', 'password', 'database')

CherryClass Root:
mask:
    def index(self):
        <html><body>
            Hello, there are currently <py-eval="myDb.query('select count(*) from user')[0][0]"
        </body></html>
```

1.7 `MaskTools` — Simple HTML patterns.

This module is a simple module that contains a few commonly used HTML patterns. Its main purpose is to show you that it ease very easy to create your own masks and reuse them all over your website.

`mask: x()`

Returns the code for a transparent pixel. This is very useful when you want to draw 1-pixel wide lines.

The way it is used is the following:

```
# Draw a 1px by 100px blue line
<table border=0 cellspacing=0 cellpadding=0><tr><td width=100 height=1 bgColor=blue py-eval
```

mask: displayByColumn(*dataList, numberOfColumns=2, columnWidth=0, gapWidth=50, tdClass=”*)

This function displays a list of data on several columns.

dataList is a list of strings that you want to display

numberOfColumns is the number of columns that you want to use to display the data

columnWidth is used if you want to use a specific with for the columns (in pixels)

gapWidth is the number of pixels between each column

tdClass is the style sheet class to use to display the strings

Example:

```
# Display integers from 1 to 102 in 7 columns with 20 pixels between each column:
<py-eval="maskTools.displayByColumn(map(str,range(1,103)), 7, 0, 20)">
```

mask: displayByLine(*dataList, numberOfLines=2, lineHeight=0, gapHeight=50*)

This function displays a list of data on several lines.

dataList is a list of strings that you want to display

numberOfLines is the number of lines that you want to use to display the data

lineHeight is used if you want to use a specific height for the lines (in pixels)

gapHeight is the number of pixels between each line

tdClass is the style sheet class to use to display the strings

Example:

```
# Display integers from 1 to 102 in 7 lines with 5 pixels between each line:
<py-eval="maskTools.displayByLine(map(str,range(1,103)), 7, 0, 5)">
```

mask: textInBox(*text, boxColor=”black”, insideColor=”white”*)

This function displays a text in a box

text is the text to display inside the box

boxColor is the color of the border of the box

insideColor is the color of the background of the box

Example:

```
<py-eval="maskTools.textInBox('This is some text displayed in a red box filled with yellow'
```

APPENDIX

A

History and License

A.1 License

CherryPy is released under the GPL license.