

# Dynamic Web content and forms processing in Python



John W. Shipman

2008-06-05 14:41

## Abstract

Describes the techniques for generating Web pages dynamically, especially from the XHTML forms interface.

This publication is available in Web form<sup>1</sup> and also as a PDF document<sup>2</sup>. Please forward any comments to [tcc-doc@nmt.edu](mailto:tcc-doc@nmt.edu).

## Table of Contents

1. Introduction: Why do we need this technique? .....	2
2. References .....	2
3. Linking to a CGI script .....	3
3.1. Linking a form to a CGI script .....	3
3.2. Direct CGI links without use of a form .....	5
4. Writing your CGI script .....	6
4.1. The cgi module .....	6
5. anyform.cgi: An example CGI script to show all form data .....	8
5.1. XHTML to be generated .....	8
5.2. Prologue .....	8
5.3. main(): The main program .....	9
5.4. showControl(): Display one control's values .....	10
5.5. Epilogue .....	10
6. Cookies: Do I know you? .....	10
6.1. Design considerations for CGI with cookies .....	12
6.2. Cookie operations .....	12
7. reader.cgi: An example script with cookies .....	13
7.1. Design notes for reader.cgi: Sequencing .....	15
7.2. The user database .....	15
7.3. Classes used in this application .....	16
7.4. reader.cgi: Prologue .....	16
7.5. Imported modules .....	17
7.6. Manifest constants .....	17
7.7. main(): Main program .....	21
7.8. deleteCase(): Delete the cookie .....	22
7.9. deletionPage(): Generate the deletion-successful page .....	23
7.10. genericPage(): Build a generic Web page .....	24

<sup>1</sup> <http://www.nmt.edu/tcc/help/pubs/pycgi/>

<sup>2</sup> <http://www.nmt.edu/tcc/help/pubs/pycgi/pycgi.pdf>

7.11. <code>chapterCase()</code> : Present the next chapter .....	24
7.12. <code>updateDatabase()</code> : Update or create the user's database entry .....	26
7.13. <code>newUserRecord()</code> : Create the initial user record .....	27
7.14. <code>setCookie()</code> : Set a cookie .....	28
7.15. <code>generateChapter()</code> : HTML page generation .....	28
7.16. <code>generateForm()</code> : Build the form element .....	29
7.17. <code>navButtons</code> : Add navigational buttons .....	31
7.18. <code>radioGroup()</code> : Add the radiobutton group .....	31
7.19. <code>class Inputs</code> : All the script's input .....	32
7.20. <code>Inputs.__init__()</code> : Constructor .....	32
7.21. <code>class UserDatabase</code> .....	34
7.22. <code>UserDatabase.__init__()</code> : Constructor .....	35
7.23. <code>UserDatabase.__getitem__()</code> : Implement dictionary get .....	35
7.24. <code>UserDatabase.__setitem__()</code> .....	36
7.25. <code>UserDatabase.__delitem__()</code> .....	36
7.26. <code>UserDatabase.cleanup()</code> .....	36
7.27. <code>UserDatabase.close()</code> .....	37
7.28. <code>class UserRecord</code> .....	37
7.29. Epilogue .....	37

## 1. Introduction: Why do we need this technique?

---

Most Web pages are *static*: the same content appears each time you visit that URL. To put up static content, you simply place a file in or under a specific directory, and give it a name that ends in “.html”. If your page is hosted at the New Mexico Tech Computer Center (TCC), static pages must reside in subdirectory “public\_html” under your home directory.

This document describes technique for *dynamic* web pages: their content is generated on demand. This technique is called CGI, for Common Gateway Interface. Instead of writing the page's content, you will write a program that generates the content as its output. Such a program is called a *CGI script* (nowadays the terms “script” and “program” are pretty much interchangeable). The program can be written in just about any modern programming language. We'll describe how to do it in the Python language, a very good modern general-purpose programming language.

One of the commonest uses for dynamic web pages is to handle forms. For example, you might put up a web page displaying a form that allows the user to enter search terms for a database query. When they click on the *Submit* button, a new page appears that displays the results of their query.

However, dynamic web pages are useful for other applications besides forms handling. For example, you might put up a “Site Map” link that takes the user to an interactive site map. An example is the site map for the TCC Help System<sup>3</sup>.

## 2. References

---

This document describes everything you will need to do CGI programming in Python. If you would like to find out more, here are some relevant references.

- Wikipedia articles: Common Gateway Interface<sup>4</sup> and HTTP Cookie<sup>5</sup>.

<sup>3</sup> <http://infohost.nmt.edu/tcc/help/planview.cgi?h=/u/www/docs/tcc/help/&aa=site.pro&zz=site.epi>

<sup>4</sup> [http://en.wikipedia.org/wiki/Common\\_Gateway\\_Interface](http://en.wikipedia.org/wiki/Common_Gateway_Interface)

<sup>5</sup> [http://en.wikipedia.org/wiki/HTTP\\_Cookie](http://en.wikipedia.org/wiki/HTTP_Cookie)

- To encode Web pages, we recommend that you use the XHTML notation; see *Building web pages with XHTML 1.1*<sup>6</sup>.

We'll refer to this notation as HTML, a term which includes the current modern XHTML notation as well as a number of older variants.

- For more information on static web page policies on the TCC's web server, see *Web page hosting policies at www.nmt.edu*<sup>7</sup>.
- Python programming language official website<sup>8</sup>.
- *Python quick reference*<sup>9</sup>.
- *Python library reference*<sup>10</sup> describes Python's main module library. Pertinent modules include:
  - `cgi`<sup>11</sup>: Common Gateway Interface functions.
  - `Cookie`<sup>12</sup>: HTTP cookie handling.
  - `urllib`<sup>13</sup>: URL encoding and decoding.
  - `time`<sup>14</sup>: Time and date processing.
  - `gdbm`<sup>15</sup>: A persistent dictionary mechanism.

This document includes the actual code for all Web pages and Python scripts, a technique known as lightweight literate programming<sup>16</sup>. The code was developed using the Cleanroom<sup>17</sup> or zero-defect methodology.

## 3. Linking to a CGI script

---

CGI scripting is most commonly used to process the data from a form after it is filled out by a user. However, you can also place a link on a Web page that invokes the CGI script directly, without requiring a form; for that technique, see Section 3.2, "Direct CGI links without use of a form" (p. 5).

### 3.1. Linking a form to a CGI script

Any web page may contain any number of "fill-out forms", each represented by a `form` element in its HTML coding. For the complete details of form design, refer to the relevant section of *Building web pages with XHTML 1.1*<sup>18</sup>.

The design of an HTML form must be closely coordinated with the design of the CGI script that processes that form's data.

- Each form has a set of *controls* such as checkboxes, text entry fields, and buttons.

---

<sup>6</sup> <http://www.nmt.edu/tcc/help/pubs/xhtml/>  
<sup>7</sup> <http://www.nmt.edu/tcc/help/html/hosting.html>  
<sup>8</sup> <http://www.python.org/>  
<sup>9</sup> <http://www.nmt.edu/tcc/help/pubs/python/>  
<sup>10</sup> <http://docs.python.org/lib>  
<sup>11</sup> <http://docs.python.org/lib/module-cgi.html>  
<sup>12</sup> <http://docs.python.org/lib/module-Cookie.html>  
<sup>13</sup> <http://docs.python.org/lib/module-urllib.html>  
<sup>14</sup> <http://docs.python.org/lib/module-time.html>  
<sup>15</sup> <http://docs.python.org/lib/module-gdbm.html>  
<sup>16</sup> <http://www.nmt.edu/~shipman/soft/litprog/>  
<sup>17</sup> <http://www.nmt.edu/~shipman/soft/clean/>  
<sup>18</sup> <http://www.nmt.edu/tcc/help/pubs/xhtml/form-elt.html>

- Each control must have an internal *control name* that is used by the CGI script to retrieve the value of that control.
- The form must have a *Submit* button that causes the data on the form to be transmitted to the receiving CGI script.
- When the *Submit* button is clicked, the CGI script will receive the information from the form as a set of ordered pairs (*name*, *value*). The *name* is the control name you assign to that control in the form's HTML code. The *value* is a string of characters that describes what the user entered in that control.

Not all controls will transmit a name-value pair every time. For example, a checkbox control will transmit a name-value pair only when it is checked (turned on).

Also, be aware that some controls can transmit more than one value. For example, if your form contains a pull-down menu or pick list made with `select multiple='multiple'`, the user can select any or all of the items in the list.

- The connection from a form to the CGI script that processes it is specified by the `action` attribute of the HTML form element.

For the `method` attribute, we recommend `method='get'` in most situations. However, if your form includes a file control for uploading files, you must use `method='post'`.

Here is an example of a very simple web page that contains a form with only three elements: a text entry field labeled *Favorite color*, a checkbox labeled *Color found in nature?*, and a *Submit* button.

example-form.html

```
<html xmlns='http://www.w3.org/1999/xhtml'>
  <head>
    <title>Form example</title>
  </head>
  <body>
    <h1>Form example</h1>
    <form method='get'
      action='http://infohost5.nmt.edu/~tcc/demo/pycgi/anyform.cgi'>
      <div>
        <input type='text' name='fave' id='fave-field' />
        <label for='fave-field'>Favorite color</label>
      </div>
      <div>
        <input type='checkbox' name='natural' id='nature-field' />
        <label for='nature-field'>Color found in nature?</label>
      </div>
      <div>
        <input type='submit' value='send' />
      </div>
    </form>
  </body>
</html>
```

Here is a screen shot of this page after a user has filled it out:

## Form example

Favorite color  
 Color found in nature?

When the user clicks the *send* button, two name-value pairs will be transmitted to the handler script `example.cgi`:

- ('fave', 'puce')
- ('natural', 'on'): The default value for a checkbox is the string 'on'.

If you like, you can click here to try this form yourself.<sup>19</sup>

To find out how to write a CGI script for this form, see Section 4, “Writing your CGI script” (p. 6).

Keep in mind that some forms can transmit multiple values for one name. For example, if a form has an element `select multiple='multiple' name='rainbow'` containing the colors of the rainbow, the user might pick the top three, so name 'rainbow' is associated with the values 'red', 'orange', and 'yellow'. The associated CGI script must anticipate the possibility of multiple values for such control names.

### 3.2. Direct CGI links without use of a form

You can link to a CGI script without using an HTML form element. On the TCC web server, your script's name must end in `.cgi`, `.py`, or `.pl`.

Here's the general form of a CGI link. It looks just like any other link. When a user clicks on it, your CGI script will be run, and its output will be displayed as a Web page.

```
<a href='whatever.cgi'>link text</a>
```

You can also pass information to the CGI script by appending a question mark (?) followed by a list of names and values. These name-value pairs are transmitted to the CGI script in exactly the same way as name-value pairs from an HTML form element. Here is the general form:

```
<a href='whatever.cgi?name1=value1&name2=value2&...'>link text</a>
```

The names and values in the list must be *URL-encoded*:

- Spaces must be replaced by plus characters (+).
- Special characters (that is, anything except letters, digits, hyphen, or underbar) must be expressed as “%HH” where HH is the hexadecimal code for the character.

Python's `urllib` module has a function called `quote_plus` that performs this encoding on a given string. Here is an interactive example:

```
>>> import urllib
>>> urllib.quote_plus('abc 123 +!@#$%^&*()|\\?/_-<>')
'abc+123+%2B%21%40%23%24%25%5E%26%2A%28%29%7C%5C%3F%2F_-%3C%3E'
>>>
```

<sup>19</sup> <http://infohost5.nmt.edu/~tcc/demo/pycgi/example-form.html>

As with forms, it is important to coordinate the design of the CGI script with the links to it, so that the names used in the link are names that the script expects to receive.

Here's an example link:

```
<a href='foo.cgi?fave=dark+green&natural=on'>Click here</a>
```

When the user clicks on this link, the script `foo.cgi` will receive two name-value pairs:

- ('fave', 'dark green')
- ('natural', 'on')

## 4. Writing your CGI script

---

Python's standard modules take care of a lot of the detail work associated with writing a CGI script. You don't have to worry about whether the script was launched from a `form` element or from a non-form link, or whether the form used `method='GET'` or `method='POST'`; the modules will handle all that for you.

All CGI scripts must write these three items to its standard output stream (`sys.stdout`) in sequence:

- One or more *header lines* that tell the Web server what you want to do.
- A blank line that tells the Web server you are done writing header lines.
- The Web page you want to display, encoded as HTML.

There are a number of different headers, but typically you will write a `Content-type` header with this format, to tell the Web server you are going to send it a Web page:

```
Content-type: text/html
```

Be sure to remember to write a blank line after the header.

### 4.1. The `cgi` module

Python's standard `cgi` module has a number of useful features. You'll want to import it like this:

```
import cgi
```

Here are some of the features of this module that you'll probably find useful.

#### `cgi.escape(s)`

Given a string `s`, this function returns a string with the three special characters "`<`", "`>`", and "`&`" replaced by their escape sequences "`&lt;`", "`&gt;`", and "`&amp;`", respectively. Here's an interactive example:

```
>>> import cgi
>>> cgi.escape ( 'abc < def & ghi >' )
'abc &lt; def &amp; ghi &gt;'
>>>
```

#### `cgi.FieldStorage()`

This constructor returns a `FieldStorage` instance that contains all the name-value pairs provided as input to your script.

The constructor finds these inputs regardless of whether the sending page used `method='GET'` or `method='POST'`, and regardless of whether the input came from a form or via the techniques described in Section 3.2, “Direct CGI links without use of a form” (p. 5).

Suppose `f` is a `FieldStorage` instance. Here are the operations you can perform on it:

**`f.getvalue(cName, default=None)`**

If `f` has one value for control name `cName`, this method returns that value as a string. If there are multiple values, the method returns them as a list of strings.

If `f` has no values for control name `cName`, it returns the `default` value.

**`f.getfirst(cName, default=None)`**

If `f` has any values for control name `cName`, it returns the first value as a string, otherwise it returns `default`.

**`f.getlist(cName)`**

Returns a list of the values for control name `cName`. This list may be empty or contain one or more values as strings.

**`f.keys()`**

Returns a list of the control names that have values in `f`.

**`f.has_key(cName)`**

Returns `True` if `f` has any values for control name `cName`; otherwise returns `False`.

**`f[cName]`**

If `f` has one name-value pair for control name `cName`, this method returns an instance that represents that name-value pair:

- If your form uses `method='post'` and `enctype='multipart/form-data'`, this method will return an instance of class `FieldStorage`. For more information on retrieving uploaded files from “`input type='file'`” controls, see Section 4.1.1, “Special considerations for uploading files” (p. 7).
- Otherwise, the method will return an instance of class `MiniFieldStorage`. The actual control value will reside in the `.value` attribute of that instance.

If `f` has multiple values for control name `cName`, the method will return a list of `MiniFieldStorage` instances, each containing one control value in its `.value` attribute.

If `f` has no value for control `cName`, this method will raise a `KeyError` exception.

### 4.1.1. Special considerations for uploading files

If you want your form to allow the user to upload a file, include an “`input type='file'`” control with `method='post'` and `enctype='multipart/form-data'`.

Suppose your file control's name is `'file-con'`. This code would set `fileItem` to a `FieldStorage` instance containing information about the file to be uploaded:

```
form = cgi.FieldStorage()
fileItem = form['file-con']
```

After this operation, `fileItem.filename` will contain the name of the uploaded file as a string. There are two ways to retrieve the contents of the file:

1. If the file is not too large, simply use `fileItem.value`, or `form.getvalue('file-con')`, either of which will return the file's entire contents as a single string.

2. You may instead simply read the file's contents from attribute `fileItem.file`, which is a readable Python file instance.

## 5. anyform.cgi: An example CGI script to show all form data

---

Here is an example of a working CGI script, `anyform.cgi`. It is intended to display all the data from an arbitrary form. The example form in Section 3.1, “Linking a form to a CGI script” (p. 3) uses this script as its handler.

### 5.1. XHTML to be generated

First let's look at the HTML we want to generate. Suppose a form has two controls: a text field named 'home' containing the string 'Eagles', and a multiple select menu named 'visitors' with two values: 'Animal Angst Project' and 'Bovine Wonder Horde'.

For simplicity, we'll use the `.getlist()` method on the `FieldStorage` instance to get any number of values for each control. Then we'll concatenate the values inside square brackets, with commas between them. Here's the output:

```
<html>
  <head>
    <title>Form output test</title>
  </head>
  <body>
    <h1>Form output test</h1>
    <div>home=[Eagles]</div>
    <div>visitor=[Animal Angst Project, Bovine Wonder Horde]</div>
  </body>
</html>
```

### 5.2. Prologue

We'll start off with the usual “pound-bang line” that makes the script self-executing, followed by a comment that points to this documentation.

```
anyform.cgi
#!/usr/bin/env python
#=====
# anyform.cgi:  General-purpose Web form handler script.
#   For documentation, see:
#     http://www.nmt.edu/tcc/help/pubs/pycgi/
#-----
```

We'll use the standard `cgi` and `urllib` modules described above. We also need the `sys` module to access the standard output stream, `sys.stdout`.

```
anyform.cgi
import cgi, urllib, sys
```

The author considers it a rather sloppy and dangerous practice to generate HTML using ordinary `print` statements. Hence, this example uses the `ElementTree` module, a good choice for generating XHTML,

and a component of the standard Python library. We'll refer to it internally as `et`. For full documentation on this module, see the author's *Python XML processing with lxml*<sup>20</sup>.

anyform.cgi

```
import lxml.etree as et
```

### 5.3. `main()`: The main program

Execution starts here. The first order of business is to write a header line and blank line that informs the Web server that we intend to generate a page of HTML.

anyform.cgi

```
# - - -   m a i n

def main():
    '''Main program.
    ...
    print 'Content-type: text/html'
    print
```

We'll use the `ElementTree` module to build the output Web page. Refer to Section 5.1, "XHTML to be generated" (p. 8) for the structure of the HTML we are building here.

anyform.cgi

```
html = et.Element ( 'html' )
page = et.ElementTree ( html )
head = et.SubElement ( html, 'head' )
title = et.SubElement ( head, 'title' )
title.text = 'Form output test'

body = et.SubElement ( html, 'body' )
h1 = et.SubElement ( body, 'h1' )
h1.text = 'Form output test'
```

Next we'll get the `FieldStorage` instance and use its `.keys()` method to get a list of all the incoming control names. We'll sort the control names so they always appear in alphabetical order.

anyform.cgi

```
form = cgi.FieldStorage()
nameList = form.keys()
nameList.sort()
```

This loop goes through the control names and, for each one, calls the `showControl()` function to add a `div` element to the page body, displaying the value of that control.

anyform.cgi

```
for cName in nameList:
    showControl ( body, form, cName )
```

Finally, we use the `ElementTree.write()` method to send the HTML for the page to the standard output stream.

anyform.cgi

```
page.write ( sys.stdout )
```

<sup>20</sup> <http://www.nmt.edu/tcc/help/pubs/lxml>

## 5.4. showControl ( ) : Display one control's values

This function takes three arguments: `parent` is the `Element` instance under which the content is to be attached; `form` is the `FieldStorage` instance; and `cName` is the name of the control to be displayed.

This function is *not* designed to deal with `file` controls. For the special logic needed in that case, see Section 4.1.1, “Special considerations for uploading files” (p. 7).

anyform.cgi

```
# - - - s h o w C o n t r o l

def showControl ( parent, form, cName ):
    '''Display the value of one form control.
    ...
```

So that we don't have to worry about whether a control has no values, one value, or multiple values, we'll use the `.getList()` method to return a list of the values as strings. Then we'll join the elements of that list together with a comma and a space between each element to form the displayed string..

anyform.cgi

```
cValues = form.getList ( cName )
display = ', '.join ( cValues )
```

We're ready to generate XHTML: a `div` element containing the string “`cName=safe`”.

anyform.cgi

```
div = et.SubElement ( parent, 'div' )
div.text = '%s=[%s]' % (cName, display )
```

## 5.5. Epilogue

This epilogue starts the script up at `main()`.

anyform.cgi

```
#####
# Epilogue
#-----

main()
```

## 6. Cookies: Do I know you?

One significant problem with CGI scripting: when your script starts running, it has no memory of what went before. It has only the values that come in through `cgi.FieldStorage()`.

This can really complicate certain applications. For example:

- Suppose you publish a book in 19 chapters. Most users won't be able to read the entire thing in one sitting. You would like to remember how far each user has gotten, so that next time they come back to your site, you can take them back to where they left off.
- The term “portal” refers to a Web site where each user has some kind of login name and password. This is common for commercial Web applications, and also sites where you want to customize the presentation for each user.

Once a user logs in, you don't want to make them log in again every time they move to a different page on your site. You want to remember this individual user, but how do you know who it is?

To get around this problem, many applications use *cookies*.

- A cookie is an arbitrary string of characters that uniquely identify a *session*.
- A session is all the interactions between a particular server (Web site) and a particular browser (user). Each cookie is specific to one Web site and one user.
- A Web site can ask a user's browser to remember its cookie. Next time the browser goes to that Web site, it will send a copy of the cookie back to the Web site, so the Web site can recognize that user as someone who has been there before.

Cookies are data, not programs, and they are not very big (never more than 4096 bytes).

Here's an example of cookies in action. Let's suppose that you put up a site with your Great American Novel in 512 chapters. You don't require a login to read it, but you do want to remember which chapters each user has read, so you can start them back where they left off last time.

We'll present this as a little play with three actors: Pat User, a human; Pat's browser; and a CGI script that you have put up on your local server. As the play begins, Pat is visiting your Great American Novel CGI script for the first time.

*Pat:* Browser, I want to read this new Great American Novel.

*Browser:* Web site, please give me the Great American Novel page.

*Script:* Browser, do you have a cookie for me?

*Browser:* No, I've never seen your site before.

*Script:* [Generates a random character string, "apefmets388".] Okay, here's a cookie to store for me: apefmets388. And here's chapter one of the Great American Novel in HTML form.

*Browser:* [Stores the cookie away and remembers that it's for the Great American Novel site. Then it renders the HTML into a readable form].

*Pat:* [Clicks on the *Next Chapter* link.]

*Browser:* Web site, please give me the next chapter of the Great American Novel. Here is cookie apefmets388.

*Script:* Ah, I remember you, apefmets388. You must be ready for Chapter Two. Here it is.

That's basically how it works. The script asks the user's browser to store its cookie, so that the next time that script gets run by that user, it'll get its cookie back and can use it to look up that user and remember their specific information.

A number of important points about cookies:

- Cookies are data, not programs. They cannot infect anything.
- The lifetime of a cookie is based on several factors. The Web site can determine whether the cookie is short-lived, or whether it persists until the user kills their browser, or whether it persists even longer. The user can refuse or remove cookies from their browser.
- Cookies are not automatically secure from observation by the user or other parties. It is best not to include sensitive information (such as passwords) in cookies. Use a random string for your cookie, and relate its value to the real information through some mechanism available only to your script.
- The author is far from an authority on Web security. This document is not intended to be sufficient to build secure applications: that is a complex and ever-changing subject. Get some expert help if you are designing Web sites that handle money or sensitive information.

## 6.1. Design considerations for CGI with cookies

The sequencing of CGI scripts is complicated by the mechanics of cookies. Your script will read, set, or remove cookies by sending additional header lines before it sends the required “Content-type: text/html” header. So anything you do to cookies must be done *before* you start generating the HTML for the page.

Here is the overall sequence of a CGI script that uses cookies:

1. To check for an existing cookie, query the environmental variable “HTTP\_COOKIE”. You will need to import the `os` module. Its attribute `os.environ` is a dictionary containing all the currently defined environmental variables.

At this time you'll probably also want to call `cgi.FieldStorage()` to get the script's input name-value pairs.

2. Next, you may either set a new cookie or delete the cookie. This is done by writing special headers to the standard output stream.

If there was a cookie and you don't either set or delete it, the existing cookie will stay in place unless it expires or unless the user asks their browser to delete it.

3. Write to the standard output stream the required “Content-type: text/html” header, followed by a blank line.
4. Write the HTML for your page to the standard output stream.

## 6.2. Cookie operations

The standard Python `Cookie` module<sup>21</sup> has the operations you will need to handle cookies. Import it like this:

```
import Cookie
```

To create a new cookie `C` from a string `S`:

```
c = Cookie.SimpleCookie ( s )
```

Internally, a cookie is a set of key-value pairs. Most applications use only one name and one value, but it is possible to have several. Each key-value pair is kept in an instance of class `Morsel`.

Here are the operations on an instance `C` of class `SimpleCookie`.

### **C.has\_key ( key )**

Returns `True` if `C` has a key that matches `key`; otherwise, it returns `False`.

### **C[key]**

Return the `Morsel` instance for the given `key`, if there is one, otherwise raise `KeyError`.

### **C[key] = s**

Add a new `Morsel` to `C` with the given `key` and some string value `s`.

### **print C.output()**

Do this to set `C` as the current cookie, before you have sent any other headers such as the `Content-type` header.

Here are the operations on a `Morsel` instances `M`.

<sup>21</sup> <http://docs.python.org/lib/module-Cookie.html>

**M.key**

The key for this morsel.

**M.value**

The value for this morsel.

Instances of `Morsel` also act like dictionaries, except that only certain keys are allowed. You may read or set entries in this dictionary. Allowable keys include:

**'max-age'**

The cookie's maximum age in seconds. If you would like to log the user out of your site after a set period of time, you can use this attribute to determine when their cookie will expire.

You can set this value to a large interval, such as five years, to create what is called a *persistent cookie*. Such a cookie will live on even when the user kills and restarts their browser.

Setting this value to zero ('0') causes the cookie to be deleted.

**'path'**

Use this value to limit this cookie to only a certain directory on your site. It Specifies the URL of the top directory of your site, relative to the server. For example, if your site's pages are all at or below the URL `http://www.nmt.edu/~pat/cgi/`, the relative path would be `'/~pat/cgi/'`.

If you specify this attribute, users who come to your server but not inside your site will not see your cookies.

**'secure'**

Set this attribute to any value to request that cookies be carried only through secure channels such as the `https:` protocol. Disclaimer: This is not a complete security mechanism. Seek expert help if you are handling sensitive data.

**'comment'**

You may set a value for this attribute as a comment to explain what the cookie is for. Users can ask their browser to show them this comment while they are deciding whether or not to accept your cookie.

## 7. reader.cgi: An example script with cookies

---

In this section, we present a complete CGI application that uses cookies. It is a skeletal form of the reader for the Great American Novel described in Section 6, "Cookies: Do I know you?" (p. 10).

This script is not designed to be a polished product. Rather than presenting actual chapters of text, we'll display just the sentence "This is chapter *N*." The intent of the script is to show how cookies work in practice.

Here are the navigational features on the page:

- A *Next chapter* button and a *Previous chapter* button, so the user can move in both directions among the chapters.

These two buttons appear twice: once above the chapter text, and once below. This may seem silly, but in a production application, there might be quite a bit of text between them.

- A set of two radiobuttons that the user can select to control their cookie:
  - If they select *I'll be back* (the default), they will get a browser cookie, which will go away when their browser session ends.
  - If they select *Remember me*, they'll get a persistent cookie that expires in a week.
- If the user clicks a button labeled *Forget me*, their cookie will be deleted.

Here's a screen shot of the page on entry:

## The Great American Novel reader

[Next chapter](#) [Previous chapter](#)

This is chapter 1.

---

[Next chapter](#) [Previous chapter](#)

- I'll be back (remember me until my browser session ends)  
 Remember me (for a week)  
[Forget me](#)

And the XHTML behind this page:

mockread.html

```
<html xmlns='http://www.w3.org/1999/xhtml'>
  <head>
    <title>The Great American Novel reader</title>
  </head>
  <body>
    <h1>The Great American Novel reader</h1>
    <form method='get'
          action='http://infohost5.nmt.edu/~tcc/demo/pycgi/anyform.cgi'>
      <div>
        <input type='submit' name='next' value='Next chapter' />
        <input type='submit' name='prev' value='Previous chapter' />
      </div>
      <div>
        This is chapter 1.
      </div>
      <hr />
      <div>
        <input type='submit' name='next' value='Next chapter' />
        <input type='submit' name='prev' value='Previous chapter' />
      </div>
      <div>
        <input type='radio' name='request' value='s' id='req-b'
              checked='checked' />
        <label for='req-b'>I'll be back (remember me until my
        browser session ends)</label>
      </div>
      <div>
        <input type='radio' name='request' value='p' id='req-p' />
        <label for='req-p'>Remember me (for a week)</label>
      </div>
      <div>
        <input type='submit' name='erase' value='Forget me' />
      </div>
    </form>
  </body>
</html>
```

When the user is just reading their way through the chapters, the above page layout appears.

When the user clicks the *Forget me* button, however, the displayed page is a short announcement of success in erasing the user's cookie:

mockerase.html

```
<html xmlns='http://www.w3.org/1999/xhtml'>
  <head>
    <title>The Great American Novel reader</title>
  </head>
  <body>
    <h1>The Great American Novel Reader</h1>
    <div>
      Your records on this server have been erased.
    </div>
  </body>
</html>
```

Click here to try the page yourself<sup>22</sup>.

## 7.1. Design notes for reader.cgi: Sequencing

For reasons discussed in Section 6.1, “Design considerations for CGI with cookies” (p. 12), this script must carry out these three phases in sequence:

1. Gather all input. Most of the input comes from the `FieldStorage` instance containing the name-value pairs from the form. However, an incoming cookie, if any, comes from an environmental variable.
2. We must write the header or headers to standard output before we start generating HTML. A header line that sets or deletes a cookie must come first. In any case, the last header must be the standard “Content-type: text/html”, followed by a blank line.
3. There are two cases of HTML generation. If the user has just deleted their cookie, we'll generate the short confirmation page. Otherwise, we generate the general-purpose page, containing the requested chapter for this user.

## 7.2. The user database

The whole purpose of cookies in this application is to remember which chapter the user was reading, so we can take them back there next time.

We could put that information right in the cookie, and modify the cookie each time. However, this isn't very realistic. Because sites that use cookies often pass sensitive data to CGI scripts, it's unsafe to place user data right in the cookie. The preferred technique is to generate a cookie from random characters, and then use that as a key to look up the *real* user information somewhere.

Here is what we need to remember for each user:

- Which chapter they're currently reading.
- Whether they selected a session cookie (*I'll be back*) or a persistent cookie (*Remember me*). Once the user has selected one of these two radiobuttons, we'll want to continue to display their selected value. In an early version of this script, the session cookie radiobutton was always selected, but we don't want to change the user's preference. We want to leave it the way they last set it.

<sup>22</sup> <http://infohost5.nmt.edu/~tcc/demo/pycgi/reader.cgi>

For storing and retrieving small amounts of data, Python's built-in `gdbm` module<sup>23</sup> is a simple, appropriate tool. A `gdbm` file works like a Python dictionary: you give it a key and a value, and later you can give it the same key and you'll get back the same value. For simplicity's sake, both key and value must be strings.

With `gdbm`, we can just make each user's cookie the key, and keep the user's values in string form.

There is a subtle problem with using a `gdbm` database in this way: database bloat. Over time, if many users visit the system, the file may get quite large. We need some way to discard entries after a reasonable period. In this example application, we'll limit the lifetime of active cookies to a week, but in a production application, you'll want to think about how long you let a user stay around before you forget their information.

Hence, in each user's database entry, in addition to their current chapter number and their "persistent flag," we'll encode a third item: an expiration timestamp. Every time our script runs, we will go through all the entries in the database, and discard the ones that have expired. (In a production application where you are dealing with thousands of users, you might do this database cleaning in a separate program to be run periodically.)

The `gdbm` module requires that all keys and values must be strings. The cookie is a suitable key just as it is. As for the user's data, we'll encode the value like this:

```
C,P,T
```

- *C* is the user's current chapter number.
- *P* is 1 if the user last selected a persistent cookie, 0 if they selected a session cookie.
- *T* is the expiration time encoded as Unix "epoch time", that is, seconds since January 0, 1970.

### 7.3. Classes used in this application

Execution starts at Section 7.7, "main(): Main program" (p. 21). We'll also use three helper classes:

- Section 7.19, "class Inputs: All the script's input" (p. 32): This class encapsulates all the logic that deals with fetching the script's inputs from two sources: the `cgi.FieldStorage` instance that transmits the form's name-value pairs, and the environmental variable that contains the incoming cookie, if any.
- Section 7.21, "class UserDatabase" (p. 34): This class represents the file containing our records for each user. An instance is a container for `UserRecord` instances.
- Section 7.28, "class UserRecord" (p. 37): An instance of this class is an abstract data type containing the information for one user.

### 7.4. reader.cgi: Prologue

Here we begin the actual script. The first line makes the script self-executing. This is followed by a comment pointing back at this documentation.

```
reader.cgi
#!/usr/bin/env python
#=====
# reader.cgi: Example CGI script showing cookie processing
#   For documentation, see:
```

<sup>23</sup> <http://docs.python.org/lib/module-gdbm.html>

```
# http://www.nmt.edu/tcc/help/pubs/pycgi/  
#-----
```

## 7.5. Imported modules

From the standard Python library, we need `sys` for the standard output stream, `sys.stdout`. We need `os` for the environmental variable dictionary, `os.environ`. The `string` module supplies `string.letters` and `string.digits`, from which we pick the cookie's random characters, and the `random` module randomizes those choices. The `gdbm` module is the lightweight database that lets us store and retrieve user information. The `time` module is necessary to determine expiration times.

```
reader.cgi  
#=====br/># Imports  
#-----br/>import sys, os, string, random, gdbm, time
```

Specific to Web applications are: `Cookie` module, with handy functions for cookie work; `urllib` for escaping displayed values; and `cgi` for obtaining name-value pairs passed to the script. The `cgitb` module displays a stack traceback in case the script fails; in a production script, you would remove the `“cgitb.enable()”` that turns on this traceback feature.

```
reader.cgi  
import cgi, urllib, Cookie  
import cgitb; cgitb.enable()
```

The `lxml.etree` module handles XHTML generation; we import it here as `et`.

```
reader.cgi  
import lxml.etree as et
```

## 7.6. Manifest constants

Global names in capital letters are constants used in this script.

```
reader.cgi  
#=====br/># Manifest constants  
#-----
```

### 7.6.1. HTTP\_COOKIE

Name of the environmental variable that contains the user's existing cookie, if any.

```
reader.cgi  
HTTP_COOKIE = 'HTTP_COOKIE'
```

### 7.6.2. NEXT\_CONTROL

This is the name of the *Next chapter* control on the form. It must match the control name in the generated HTML.

```
reader.cgi  
NEXT_CONTROL = 'next'
```

### 7.6.3. NEXT\_LABEL

The label that appears on the *Next chapter* button.

reader.cgi

```
NEXT_LABEL = 'Next chapter'
```

### 7.6.4. PREV\_CONTROL

This is the name of the *Previous chapter* control on the form.

reader.cgi

```
PREV_CONTROL = 'prev'
```

### 7.6.5. PREV\_LABEL

The label that appears on the *Previous chapter* button.

reader.cgi

```
PREV_LABEL = 'Previous chapter'
```

### 7.6.6. REQUEST\_GROUP

Name of the radiobutton group that the user selects to control their cookie.

reader.cgi

```
REQUEST_GROUP = 'request'
```

### 7.6.7. REQUEST\_SESSION

Value of the REQUEST\_GROUP control for a session cookie.

reader.cgi

```
REQUEST_SESSION = '0'
```

### 7.6.8. REQUEST\_PERSIST

Value of the REQUEST\_GROUP control for a persistent cookie.

reader.cgi

```
REQUEST_PERSIST = '1'
```

### 7.6.9. SESSION\_LABEL

The label on the radiobutton that selects a session cookie.

reader.cgi

```
SESSION_LABEL = ( "I'll be back (remember me until my browser "  
                  "session ends)" )
```

### 7.6.10. PERSIST\_LABEL

The label on the radiobutton that selects a persistent cookie.

```
PERSIST_LABEL = "Remember me (for a week)"
```

### 7.6.11. FORGET\_CONTROL

Value of the REQUEST\_GROUP control when the user wants to delete their cookie.

```
FORGET_CONTROL = 'd'
```

### 7.6.12. FORGET\_LABEL

The legend on the "Forget me" control.

```
FORGET_LABEL = 'Forget me'
```

### 7.6.13. DB\_NAME

Name of the gdbm file where we keep active user records.

```
DB_NAME = 'users.db'
```

### 7.6.14. PERSIST\_SECONDS

The number of seconds in a cookie's "maximum age" if the user has request a persistent cookie. This value is for one week: 60 seconds/minute times 60 minutes/hour times 24 hours/day times 7 days. In a production application, a year or more might be a better choice.

```
PERSIST_SECONDS = 60 * 60 * 24 * 7
```

### 7.6.15. SESSION\_SECONDS

When the user requests a session cookie, which will expire when they next kill their browser, we'll retain their database entry for this duration. Because this is just a demo program, we'll use a pretty short interval, one week. In a production application you would probably allow more time.

```
SESSION_SECONDS = PERSIST_SECONDS
```

### 7.6.16. CHAPTER\_COUNT

The total number of chapters. Since this is just a demo, we'll pick an arbitrary value of nine.

```
CHAPTER_COUNT = 9
```

### 7.6.17. MAX\_AGE

This constant defines the name of the maximum age attribute of a cookie.

reader.cgi

```
MAX_AGE = 'max-age'
```

### 7.6.18. DOCTYPE: Document type declaration

Placing this block at the top of each generated page makes the W3 validator happy.

reader.cgi

```
DOCTYPE = '''<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">'''
```

### 7.6.19. PAGE\_TITLE

The title that appears on all our generated Web pages.

reader.cgi

```
PAGE_TITLE = 'The Great American Novel reader'
```

### 7.6.20. HTML\_HEADER

The HTTP header that declares we are generating HTML dynamically.

reader.cgi

```
HTML_HEADER = 'Content-type: text/html'
```

### 7.6.21. XHTML\_NAMESPACE

Value of the xmlns attribute for a properly constructed XHTML page.

reader.cgi

```
XHTML_NAMESPACE = 'http://www.w3.org/1999/xhtml'
```

### 7.6.22. USER\_ID\_LENGTH

The length of a user's ID string. Too short and there might be a collision someday; too long and it eats up database space. Eight seems like a reasonable tradeoff.

reader.cgi

```
USER_ID_LENGTH = 8
```

### 7.6.23. MORSEL\_NAME

When creating a new cookie, this is the name used in the morsel containing the cookie's value.

reader.cgi

```
MORSEL_NAME = 'chapter'
```

### 7.6.24. BASE\_URL: The URL of the script's directory

This string, plus reader.cgi, is the URL where reader.cgi lives.

reader.cgi

```
BASE_URL = 'http://infohost5.nmt.edu/~tcc/demo/pycgi/'
```

## 7.7. main(): Main program

First we process the inputs; see Section 7.19, “class Inputs: All the script's input” (p. 32). We'll also open the database file. The 'c' mode argument causes it to be created if necessary; the 's' mode suffix requests that operations against be synchronized to the disk immediately. (If multiple copies of reader.cgi are running, gdbm will take care of locking the file.)

reader.cgi

```
# - - -   m a i n

def main():
    '''Main program.

    [ if cgi.FieldStorage() requests cookie deletion ->
      file DB_NAME := that file with no entry for the
        user's cookie, and with all expired entries deleted
      sys.stdout += (header to delete the cookie) +
        (HTML header) + (blank line) + (deletion-successful page)
    else if (the user has no cookie) or
      (the user's cookie is not a key in file DB_NAME) ->
      file DB_NAME := that file with a new entry whose key is a
        random string and whose value is chapter 1 and an
        expiration time as specified by cgi.FieldStorage(),
        and with expired entries deleted
      sys.stdout += (header to set that random string as
        the cookie) + (HTML header) + (blank line) +
        (page showing the chapter for that db entry)
    else ->
      file DB_NAME := that file with the entry for the
        user's cookie, updated to the chapter number
        specified by cgi.FieldStorage(), and with expired
        entries deleted
      sys.stdout += (HTML header) + (blank line) +
        (page showing the chapter for that db entry) ]
    ...
'''
```

First we'll transform all the inputs into an instance of the Inputs class, and open our database file.

reader.cgi

```
##- 1 -
# [ inputs := an Inputs instance representing all form
#   name-value pairs and relevant environmental variables
#   sent to this script
inputs = Inputs()
```

For the database file name, see Section 7.6.13, “DB\_NAME” (p. 19); see also Section 7.21, “class User-Database” (p. 34).

reader.cgi

```
##- 2 -
# [ if DB_NAME can be opened as a gdbm file ->
#   db := a UserDatabase instance representing that file
#   else ->
#   file DB_NAME := a new, empty gdbm file
#   db := a UserDatabase instance representing that file ]
db = UserDatabase()
```

If the user wants the cookie deleted, that's a special case with a different output page; see Section 7.8, “deleteCase(): Delete the cookie” (p. 22). Otherwise we generate the usual page.

reader.cgi

```
#-- 3 --
# [ if inputs.forget ->
#     db := db with the entry for key inputs.userId deleted
#     sys.stdout += (header to delete the cookie) +
#                 (HTML header) + (blank line) +
#                 (deletion-successful page)
# else if (inputs.userId is None) or (inputs.userId is not
# a key in db)->
#     db := db with a new entry whose key is a random
#         string and whose value is chapter 1 and an expiration time
#         as specified by inputs
#     sys.stdout += (header to set that random string as
#                 the cookie) + (HTML header) + (blank line) +
#                 (page showing the chapter for that db entry)
# else ->
#     db := db with the entry for inputs.userId updated
#         to the page number specified by inputs
#     sys.stdout += (HTML header) + (blank line) +
#                 (page showing the chapter for that db entry) ]
if inputs.forget:
    deleteCase(inputs, db)
else:
    chapterCase(inputs, db)
```

Finally, to prevent the database from growing without limit, we'll delete expired entries; see Section 7.26, “UserDatabase.cleanup()” (p. 36).

reader.cgi

```
#-- 4 --
# [ db := db with all expired entries deleted ]
db.cleanup()
db.close()
```

## 7.8. deleteCase(): Delete the cookie

This function handles the case where the user wants their cookie removed from our system.

reader.cgi

```
# - - -   d e l e t e C a s e

def deleteCase ( inputs, db ):
    '''Delete the user's cookie and database entry.

    [ (inputs is an Inputs instance) and
      (db is a UserDatabase instance) ->
      db := db with the entry for key inputs.userId deleted
      sys.stdout += (header to delete the cookie) +
                    (HTML header) + (blank line) +
                    (deletion-successful page) ]
    ...
```

Deleting an entry from a `gdbm` file works the same as if it were a normal dictionary: we use the `del` statement. However, there is no guarantee this entry exists in the database, and in that case the `del` statement will raise a `KeyError` exception, which we can happily ignore.

reader.cgi

```
#-- 1 --
# [ if db has an entry with key=inputs.userId ->
#     db := db with that entry removed
#     else -> I ]
try:
    del db[inputs.userId]
except KeyError:
    pass
```

Deleting the cookie from the user's browser is done by writing a header line that sets a cookie with a "max-age" attribute of zero. In this case, the cookie's name is ignored. See Section 7.6.17, "MAX\_AGE" (p. 19).

reader.cgi

```
#-- 2 --
# [ antiCookie := a Cookie.SimpleCookie instance with
#     name 'ignore' and max-age=0 ]
antiCookie = Cookie.SimpleCookie()
antiCookie['ignore'] = 'ignore'
antiMorsel = antiCookie['ignore']
antiMorsel[MAX_AGE] = '0'

#-- 3 --
# [ sys.stdout += header that sets antiCookie ]
print antiCookie.output()
```

For the HTML generation of the deletion message page, see Section 7.9, "deletionPage(): Generate the deletion-successful page" (p. 23).

reader.cgi

```
#-- 4 --
# [ sys.stdout += (HTML header) + (blank line) +
#     (deletion-successful page) ]
deletionPage()
```

## 7.9. deletionPage(): Generate the deletion-successful page

For the HTML generated here, see Section 7, "reader.cgi: An example script with cookies" (p. 13). This function uses Section 7.10, "genericPage(): Build a generic Web page" (p. 24), which generates a general-purpose HTML page. See also Section 7.6.19, "PAGE\_TITLE" (p. 20).

reader.cgi

```
# - - -   d e l e t i o n P a g e

def deletionPage():
    '''Write the deletion-successful page.

    [ sys.stdout += (HTML header) + (blank line) +
      (deletion-successful page) ]
    ...
#-- 1 --
```

```

# [ page := a generic HTML page as an et.ElementTree,
#     with title PAGE_TITLE
#     body := the body element of that page ]
page, body = genericPage ( PAGE_TITLE )

```

Next we add the body content.

```

#-- 2 --
# [ body += content of the deletion-successful page ]
div = et.SubElement ( body, 'div' )
div.text = 'Your records on this server have been erased.'

```

reader.cgi

Finally, write the HTML header, blank line, and page content; see Section 7.6.20, “HTML\_HEADER” (p. 20) and Section 7.6.18, “DOCTYPE: Document type declaration” (p. 20).

```

#-- 3 --
# [ sys.stdout += (HTML header) + (blank line) +
#     (page as HTML) ]
print HTML_HEADER
print
print DOCTYPE
page.write ( sys.stdout, pretty_print=True )

```

reader.cgi

## 7.10. genericPage(): Build a generic Web page

This function builds a Web page as an et.ElementTree.

```

# - - -   g e n e r i c P a g e

def genericPage ( titleText ):
    '''Build a generic Web page frame.

    [ titleText is a string ->
      return (a new et.ElementTree containing a minimal Web
      page, the body element of that page) ]
    ...

    root = et.Element ( 'html', xmlns=XHTML_NAMESPACE )
    page = et.ElementTree ( root )
    head = et.SubElement ( root, 'head' )
    title = et.SubElement ( head, 'title' )
    title.text = titleText
    body = et.SubElement ( root, 'body' )
    h1 = et.SubElement ( body, 'h1' )
    h1.text = titleText
    return (page, body)

```

reader.cgi

## 7.11. chapterCase(): Present the next chapter

This function handles the generation of the normal output page, showing one simulated “chapter”.

```
# - - -   c h a p t e r C a s e

def chapterCase ( inputs, db ):
    '''Generate the usual page displaying one chapter.

    [ (inputs is an Inputs instance) and
      (db is a gdbm instance) ->
        if (inputs.userId is None) or
          (inputs.userId is not a key in db) ->
            db := db with a new entry whose userId is a random
                  string, whose chapterNo is 1, and whose
                  expiration time is as specified by inputs
            sys.stdout += (header to set that random string as
                          a session cookie) + (HTML header) + (blank line) +
                          (page showing the chapter for that db entry)
        else ->
            db := db with the entry for inputs.userId updated
                  as specified by inputs
            sys.stdout += (header to set inputs.userId as the
                          cookie, with persistent specified by inputs) +
                          (HTML header) + (blank line) +
                          (page showing the chapter for that db entry) ]

    ...
```

This function has a lot to do. We'll basically break it down into three steps: updating the database, generating the cookie header, and generating the actual HTML.

First, the database update step: see Section 7.12, “updateDatabase(): Update or create the user's database entry” (p. 26).

```
#-- 1 --
# [ if (inputs.userId is None) or (inputs.userId is not in db) ->
#     db := db with a new UserRecord whose userId is a random
#         string, whose chapterNo is 1, and whose
#         expiration time is as specified by inputs
#     userRecord := that new UserRecord
# else ->
#     db := db with the UserRecord for inputs.userId updated
#         as specified by inputs
#     userRecord := that updated UserRecord ]
userRecord = updateDatabase ( inputs, db )
```

The updated userRecord has all the information we need to set the new cookie.

```
#-- 2 --
# [ sys.stdout += header to set a cookie as per userRecord ]
setCookie ( userRecord )
```

The rest is the generation of the actual HTML page.

```
#-- 3 --
# [ sys.stdout += (HTML header) + (blank line) +
```

```
# (page displaying the chapter specified by userRecord) ]
generateChapter ( userRecord )
```

## 7.12. updateDatabase(): Update or create the user's database entry

This function update's the user's database entry if there is one. If the user has no database entry, we create one here.

reader.cgi

```
# - - -   u p d a t e D a t a b a s e

def updateDatabase ( inputs, db ):
    '''Generate a cookie if necessary; update the database.

    [ (inputs is an Inputs instance) and
      (db is a UserDatabase instance) ->
        if (inputs.userId is None) or (inputs.userId is not in db) ->
            db := db with a new UserRecord whose userId is a random
                string, whose chapterNo is 1, and whose
                expiration time is as specified by inputs
            return that new UserRecord
        else ->
            db := db with the UserRecord for inputs.userId updated
                as specified by inputs
            return that updated UserRecord ]
    ...
```

First we see if the database has a UserRecord for this user. If not, we create a new one; see Section 7.13, “newUserRecord(): Create the initial user record” (p. 27).

reader.cgi

```
#-- 1 --
# [ if db has an entry for inputs.userId ->
#   userRecord := the corresponding value
# else ->
#   userRecord := a new UserRecord with a randomy
#                 userId, chapterNo=1, persist=0, and expiration=0 ]
try:
    userRecord = db[inputs.userId]
except KeyError:
    userRecord = newUserRecord()
```

Adjust the chapter number up or down if this is a repeat visit, but no lower than 1 and no higher than the value specified in Section 7.6.16, “CHAPTER\_COUNT” (p. 19). If this is the user's first visit, neither `inputs.next` nor `inputs.prev` will be set.

reader.cgi

```
#-- 2 --
if inputs.next:
    userRecord.chapterNo = min ( userRecord.chapterNo + 1,
                                CHAPTER_COUNT )
elif inputs.prev:
    userRecord.chapterNo = max ( userRecord.chapterNo - 1,
                                1 )
```

Next we adjust the expiration time. See Section 7.6.14, “PERSIST\_SECONDS” (p. 19) and Section 7.6.15, “SESSION\_SECONDS” (p. 19).

reader.cgi

```
#-- 3 --
# [ now := current epoch time as an integer in seconds ]
now = int ( time.time() )

#-- 4 --
if inputs.persist:
    userRecord.persist      = 1
    userRecord.expiration  = now + PERSIST_SECONDS
else:
    userRecord.persist      = 0
    userRecord.expiration  = now + SESSION_SECONDS
```

Place or replace the user's record in the database.

reader.cgi

```
#-- 5 --
db[userRecord.userId] = userRecord

#-- 6 --
return userRecord
```

## 7.13. newUserRecord(): Create the initial user record

The default state of a new user is to be on chapter 1, with a session cookie.

reader.cgi

```
# - - -   n e w U s e r R e c o r d

def newUserRecord():
    '''Create a new UserRecord instance.

    [ return a new UserRecord with a random userId,
      chapterNo=1, persist=0, and expiration=0 ]
    ...
```

To generate a random `userId`, we'll use the `random.choice()` function, which picks a random member of a sequence. We'll pick the characters from a concatenation of `string.letters`, which contains the 52 upper- and lowercase letters, and `string.digits`, the ten digits. The length of the user ID is defined in Section 7.6.22, “USER\_ID\_LENGTH” (p. 20).

reader.cgi

```
#-- 1 --
# [ userId := USER_ID_LENGTH random letters or digits ]
c = string.letters + string.digits
L = [random.choice(c) for i in range(USER_ID_LENGTH)]
userId = "".join ( L )
```

The expiration time will be adjusted by the caller, so we don't need to worry about it here. See Section 7.28, “class UserRecord” (p. 37).

reader.cgi

```
#-- 2 --
# [ userRecord := a new UserRecord instance with
```

```

#     userId=(userId), chapterNo=1, persist=0, and
#     expiration=0 ]
userRecord = UserRecord ( userId, 1, 0, 0 )

#-- 3 --
return userRecord

```

## 7.14. setCookie(): Set a cookie

This function creates a cookie according to the values in the given `userRecord`, and then writes to the standard output the header that sets that cookie.

reader.cgi

```

# - - -   s e t C o o k i e

def setCookie ( userRecord ):
    '''Generate the header to set a cookie.

    [ userRecord is a UserRecord instance ->
      sys.stdout += header to set a cookie as per userRecord ]
    ...
#-- 1 --
# [ cookie := a new Cookie.SimpleCookie() with no morsels ]
cookie = Cookie.SimpleCookie()

```

For the name of the morsel where the user ID is kept, see Section 7.6.23, “MORSEL\_NAME” (p. 20).

reader.cgi

```

#-- 2 --
# [ cookie += a morsel whose name is MORSEL_NAME and whose
#           value is userRecord.userId ]
cookie[MORSEL_NAME] = userRecord.userId

```

For the duration of persistent cookies, see Section 7.6.14, “PERSIST\_SECONDS” (p. 19).

reader.cgi

```

#-- 3 --
# [ if userRecord.persist ->
#   cookie := cookie with its only morsel's maximum age
#           set to PERSIST_SECONDS
#   else -> I ]
if userRecord.persist:
    morsel = cookie[MORSEL_NAME]
    morsel[MAX_AGE] = PERSIST_SECONDS

#-- 4 --
# [ sys.stdout += the header that sets cookie=(cookie) ]
print cookie.output()

```

## 7.15. generateChapter(): HTML page generation

This function writes the usual “Content - type” header and blank line, then generates the chapter page according to its database entry. For the generated HTML, see Section 7, “reader.cgi: An example script with cookies” (p. 13).

```
# - - -   g e n e r a t e C h a p t e r

def generateChapter ( userRecord ):
    '''Generate the usual HTML displaying one chapter.

        [ dbValue is a string of the form "(chapterNo),(expiration)" ->
          sys.stdout += (HTML header) + (blank line) +
            (page showing chapter (chapterNo)) ]
    ...
```

We'll use Section 7.10, “genericPage(): Build a generic Web page” (p. 24) to set up an empty Web page. For the page titles, see Section 7.6.19, “PAGE\_TITLE” (p. 20).

```
#-- 1 --
# [ page := a minimal Web page as an et.ElementTree,
#       carrying title PAGE_TITLE
#       body := the body element of that page ]
page, body = genericPage ( PAGE_TITLE )
```

The form element has all the remaining content; it is created by Section 7.16, “generateForm(): Build the form element” (p. 29). Writing the headers (see Section 7.6.20, “HTML\_HEADER” (p. 20)), and the generated page, works just the same as in Section 7.9, “deletionPage(): Generate the deletion-successful page” (p. 23).

```
#-- 2 --
# [ body += a form element displaying the chapter as
#         specified by userRecord ]
generateForm ( body, userRecord )
```

See Section 7.6.20, “HTML\_HEADER” (p. 20) and Section 7.6.18, “DOCTYPE: Document type declaration” (p. 20).

```
#-- 3 --
# [ sys.stdout += (HTML header) + (blank line) +
#               (page as HTML) ]
print HTML_HEADER
print
print DOCTYPE
page.write ( sys.stdout, pretty_print=True )
```

## 7.16. generateForm(): Build the form element

For a normal chapter page, this function builds the HTML form element and everything underneath it. For the HTML to be generated, refer to Section 7, “reader.cgi: An example script with cookies” (p. 13).

```
# - - -   g e n e r a t e F o r m

def generateForm ( body, userRecord ):
    '''Generate the form element and its content.
```

```

    [ (body is an et.Element) and
      (userRecord is a UserRecord instance) ->
        body += a form element displaying the chapter as
                specified by userRecord ]
    ...

```

The form element has two attributes: `method='get'` and an `action` attribute whose value is the URL where `reader.cgi` lives; see Section 7.6.24, “BASE\_URL: The URL of the script’s directory” (p. 20).

reader.cgi

```

#-- 1 --
# [ body += a new form element with method='get' and
#   action=(BASE_URL)+'reader.cgi'
#   form := that new element ]
form = et.SubElement ( body, 'form', method='get',
                       action=(BASE_URL+'reader.cgi') )

```

The navigation buttons are added next; see Section 7.17, “navButtons: Add navigational buttons” (p. 31).

reader.cgi

```

#-- 2 --
# [ form += next/previous chapter buttons ]
navButtons ( form )

```

Here is the simulated chapter content.

reader.cgi

```

#-- 3 --
# [ form += a div containing the simulated content for
#   chapter (userRecord.chapterNo) ]
contentDiv = et.SubElement ( form, 'div' )
contentDiv.text = 'This is chapter %d.' % userRecord.chapterNo

```

Next is a horizontal rule and a duplicate set of navigation buttons.

reader.cgi

```

#-- 4 --
# [ form += (horizontal rule) + (next/previous chapter
#   buttons) ]
et.SubElement ( form, 'hr' )
navButtons ( form )

```

For the logic that adds the radiobutton group, see Section 7.18, “radioGroup(): Add the radiobutton group” (p. 31). This function needs the `userRecord` so that it can set the radiobuttons the way the user last set them.

reader.cgi

```

#-- 5 --
# [ form += the radiobutton group in the same state as
#   userRecord.persist ]
radioGroup ( form, userRecord )

```

Last on the page is the *Forget me* button.

reader.cgi

```

#-- 6 --
# [ form += a submit button with name=FORGET_CONTROL
#   and value=FORGET_LABEL ]

```

```

div = et.SubElement ( form, 'div' )
forgetButton = et.SubElement ( div, 'input', type='submit',
                               name=FORGET_CONTROL, value=FORGET_LABEL )

```

## 7.17. navButtons: Add navigational buttons

This function adds the *Next chapter* and *Previous chapter* buttons to the given parent. For control names and labels, see Section 7.6.2, “NEXT\_CONTROL” (p. 17), Section 7.6.3, “NEXT\_LABEL” (p. 18), Section 7.6.4, “PREV\_CONTROL” (p. 18), and Section 7.6.5, “PREV\_LABEL” (p. 18).

reader.cgi

```

# - - -   n a v B u t t o n s

def navButtons ( parent ):
    '''Add the next/previous chapter buttons.

    [ parent is an et.Element ->
      parent += a div element containing two submit
                buttons, one for next chapter, one for previous ]
    ...

    div = et.SubElement ( parent, 'div' )
    nextButton = et.SubElement ( div, 'input', type='submit',
                                 name=NEXT_CONTROL, value=NEXT_LABEL )
    prevButton = et.SubElement ( div, 'input', type='submit',
                                 name=PREV_CONTROL, value=PREV_LABEL )

```

## 7.18. radioGroup(): Add the radiobutton group

This function adds the group of two radiobuttons that allow the user to select whether they get a persistent cookie or a session cookie.

reader.cgi

```

# - - -   r a d i o G r o u p

def radioGroup ( parent, userRecord ):
    '''Add the REQUEST_GROUP radiobutton group.

    [ parent is an et.Element ->
      parent += the radiobuttons in REQUEST_GROUP in the
                same state as userRecord.persist ]
    ...

```

For control names and labels, see Section 7.6.6, “REQUEST\_GROUP” (p. 18) and adjacent entries.

reader.cgi

```

#-- 1 --
# [ parent += two new 'div' children
#   div1 := the first child
#   div2 := the second child ]
div1 = et.SubElement ( parent, 'div' )
div2 = et.SubElement ( parent, 'div' )

#-- 2 --
# [ div1 += a new radiobutton with name REQUEST_GROUP and

```

```

#             value REQUEST_SESSION
# radioSession := that radiobutton ]
radioSession = et.SubElement ( div1, 'input', type='radio',
                               name=REQUEST_GROUP, value=REQUEST_SESSION, id='req-s' )
radioLabel = et.SubElement ( div1, 'label' )
radioLabel.attrib['for'] = 'req-s'
radioLabel.text = SESSION_LABEL

#-- 3 --
# [ div2 += a new radiobutton with name REQUEST_GROUP and
#             value REQUEST_PERSIST
# radioPersist := that radiobutton ]
radioPersist = et.SubElement ( div2, 'input', type='radio',
                               name=REQUEST_GROUP, value=REQUEST_PERSIST, id='req-p' )
radioLabel = et.SubElement ( div2, 'label' )
radioLabel.attrib['for'] = 'req-p'
radioLabel.text = PERSIST_LABEL

#-- 4 --
# [ if userRecord.persist ->
#     radioPersist := radioPersist, checked initially
# else ->
#     radioSession := radioSession, checked initially ]
if userRecord.persist:
    radioPersist.attrib['checked'] = 'checked'
else:
    radioSession.attrib['checked'] = 'checked'

```

## 7.19. class Inputs: All the script's input

An instance of this class contains all the inputs to the script. Here is the interface:

### **I = Inputs()**

Returns an instance containing all the inputs to the script.

### **I.userId**

If the user had an existing cookie for us, its user ID is in this attribute as a string; otherwise it is an empty string.

### **I.next**

True if the user clicked the *Next chapter* button, otherwise False.

### **I.prev**

True if the user clicked the *Previous chapter* button, otherwise False.

### **I.forget**

True if the user clicked on *Forget me*, otherwise False.

### **I.persist**

True if the user selected a persistent cookie, otherwise False.

## 7.20. Inputs.\_\_init\_\_(): Constructor

Here is the beginning of the class, and the constructor.

```
# - - - - - c l a s s   I n p u t s

class Inputs:
    '''Container for all the script's inputs.
    ...

# - - -   I n p u t s .   _ _   i n i t   _ _

    def __init__ ( self ):
        '''Constructor for Inputs.
        ...
```

The existing cookie value comes from the environmental variable whose name is given in Section 7.6.1, “HTTP\_COOKIE” (p. 17). If that variable has a value, we use the `Cookie.SimpleCookie()` constructor to convert it to a `SimpleCookie` instance, and then we extract the user ID from that.

```
#-- 1 --
# [ if os.environ has a key HTTP_COOKIE ->
#     self.userId := the value for key MORSEL_NAME
#                 of a cookie made from os.environ[HTTP_COOKIE]
# else ->
#     self.userId := '' ]
try:
    rawCookie = os.environ[HTTP_COOKIE]
    bakedCookie = Cookie.SimpleCookie ( rawCookie )
    self.userId = bakedCookie[MORSEL_NAME].value
except KeyError:
    self.userId = ''
```

Now convert the various form elements into the equivalent attributes.

```
#-- 2 --
# [ form := a cgi.FieldStorage instance containing
#         any name-value pairs passed to this script ]
form = cgi.FieldStorage()
```

For control names, see Section 7.6.4, “PREV\_CONTROL” (p. 18), Section 7.6.2, “NEXT\_CONTROL” (p. 17), Section 7.6.11, “FORGET\_CONTROL” (p. 19), and Section 7.6.6, “REQUEST\_GROUP” (p. 18).

```
#-- 3 --
# [ if form has a PREV_CONTROL value ->
#     self.prev := True
# else -> I ]
if form.has_key ( PREV_CONTROL ):
    self.prev = True
else:
    self.prev = False

#-- 4 --
# [ simile ]
if form.has_key ( NEXT_CONTROL ):
    self.next = True
```

```

else:
    self.next = False

#-- 5 --
if form.has_key ( FORGET_CONTROL ):
    self.forget = True
else:
    self.forget = False

```

Next we get the value of the REQUEST\_GROUP radiobutton group. If neither radiobuttons was set, we won't get a value, but this can happen only if the form is not designed correctly (with neither of the radiobuttons carrying a checked='checked' attribute). In that case, implement the default value here. For the control values of the radiobuttons, see Section 7.6.7, "REQUEST\_SESSION" (p. 18) and Section 7.6.8, "REQUEST\_PERSIST" (p. 18).

reader.cgi

```

#-- 5 --
# [ if form has a REQUEST_GROUP value ->
#   requestValue := that value
#   else ->
#   requestValue := REQUEST_SESSION ]
try:
    requestValue = form[REQUEST_GROUP].value
except KeyError:
    requestValue = REQUEST_SESSION

if requestValue == REQUEST_PERSIST:
    self.persist = True
else:
    self.persist = False

```

## 7.21. class UserDatabase

This class encapsulates all of the `gdbm` protocol for saving and retrieving user records. It is a container class for `UserRecord` instances; see Section 7.28, "class UserRecord" (p. 37). Its interface is dictionary-like, with user ID values as the keys.

reader.cgi

```

# - - - - - c l a s s   U s e r D a t a b a s e

class UserDatabase:
    '''Represents the file of user records.

    Exports:
    UserDatabase():
        [ file DB_NAME can be opened read-write ->
          return a UserDatabase instance representing that file ]
    __getitem__ ( self, userId ):
        [ if self contains an entry for user (userId) ->
          return that user's data as a UserRecord instance
          else -> raise KeyError ]
    __setitem__ ( self, userId, userRecord ):
        [ (userId is a string) and
          (userRecord is a UserRecord instance) ->

```

```

        self := self with userRecord stored under key (userId) ]
    .__delitem__ ( self, userId ):
        [ userId is a string ->
          self := self with nothing stored under key (userId) ]
    .cleanup():
        [ self := self minus all entries whose expiration is
          in the past ]
    .close():
        [ self := self closed ]

State/Invariants:
    .__db: [ a gdbm.gdbm instance connected to file DB_NAME ]
    ...

```

## 7.22. UserDatabase.\_\_init\_\_(): Constructor

The first argument to the `gdbm.open()` method is the name of the database file. The second argument is a mode string. We use 'c' to open the file read/write, and create it if necessary. The 's' flag instructs `gdbm` to synchronize transactions to the disk right away.

reader.cgi

```

# - - -   U s e r D a t a b a s e . _ _ i n i t _ _

def __init__ ( self ):
    '''Constructor for UserDatabase.
    ...
    self.__db = gdbm.open ( DB_NAME, 'cs' )

```

## 7.23. UserDatabase.\_\_getitem\_\_(): Implement dictionary get

This method retrieves the string-valued contents for a given user ID, and returns those values as a `UserRecord` instance.

reader.cgi

```

# - - -   U s e r D a t a b a s e . _ _ g e t i t e m _ _

def __getitem__ ( self, userId ):
    '''Return self[userId] as a UserRecord.
    ...

    #-- 1 --
    # [ if self.__db has an entry for key (userId) ->
    #   rawValue := corresponding value
    #   else -> raise KeyError ]
    rawValue = self.__db[userId]

```

The three values—chapter number, persistent flag, and expiration time—are separated by commas.

reader.cgi

```

#-- 2 --
rawChapterNo, rawPersist, rawExpiration = rawValue.split(',')

#-- 3 --
return UserRecord ( userId, int(rawChapterNo),
                   int(rawPersist), int(rawExpiration) )

```

## 7.24. UserDatabase.\_\_setitem\_\_()

Converts a UserRecord back to string form and stores it into the gdbm file.

reader.cgi

```
# - - -   U s e r D a t a b a s e . _ _ s e t i t e m _ _

def __setitem__ ( self, userId, userRecord ):
    '''Implement self[userId] = userRecord.
    ...
    rawRecord = ( "%d,%d,%d" %
                  (userRecord.chapterNo, userRecord.persist,
                  userRecord.expiration) )
    self.__db[userId] = rawRecord
```

## 7.25. UserDatabase.\_\_delitem\_\_()

Deletes a database entry.

reader.cgi

```
# - - -   U s e r D a t a b a s e . _ _ d e l i t e m _ _

def __delitem__ ( self, userId ):
    '''Implements del self[userId]. We don't care if it fails.
    ...
    try:
        del self.__db[userId]
    except KeyError:
        pass
```

## 7.26. UserDatabase.cleanup()

Removes expired entries. The gdbmkeys() function generates all the keys in the database.

reader.cgi

```
# - - -   U s e r D a t a b a s e . c l e a n u p

def cleanup ( self ):
    '''Remove expired entries.
    ...
    #-- 1 --
    # [ now := current epoch time as an int ]
    now = int ( time.time() )

    #-- 2 --
    # [ self := self minus all entries whose expiration
    #     times are less than now ]
    for userId in self.__db.keys():
        #-- 2 body --
        # [ if self.__db[userId] has an expiration time
        #     less than now ->
        #     self.__db := self.__db minus that entry ]
        #-- 2.1 --
        # [ userRecord := a UserRecord made from
```

```

#             self.__db[userId] ]
userRecord = self[userId]

#-- 2.2 --
if userRecord.expiration < now:
    del self.__db[userId]

```

## 7.27. UserDatabase.close()

reader.cgi

```

# - - -   U s e r D a t a b a s e . c l o s e

def close ( self ):
    '''Close self's database.
    ...
    self.__db.close()

```

## 7.28. class UserRecord

An instance of this class holds everything we need to know about one user.

reader.cgi

```

# - - - - -   c l a s s   U s e r R e c o r d

class UserRecord:
    '''Record for one user.

    Exports:
    UserRecord ( userId, chapterNo, persist, expiration ):
        [ (userId is a string) and
          (chapterNo is an positive integer) and
          (persist is 1 for persistent cookie, 0 for session
           cookie) and
          (expiration is the user's expiration timestamp as
           an integer epoch time) ->
          return a new UserRecord with those values ]
    .userId:          [ as passed to constructor, read-write ]
    .chapterNo:       [ as passed to constructor, read-write ]
    .persist:         [ as passed to constructor, read-write ]
    .expiration:     [ as passed to constructor, read-write ]
    ...

    def __init__ ( self, userId, chapterNo, persist, expiration ):
        '''Constructor for UserRecord
        ...

        self.userId      = userId
        self.chapterNo   = chapterNo
        self.persist     = persist
        self.expiration  = expiration

```

## 7.29. Epilogue

This code starts execution of the main program.

```
#=====
# Epilogue
#-----
main()
```