

Web Application Engineering

DR. YIHSIANG LIOW (JANUARY 9, 2019)

Contents

| | |
|---|-----------|
| 1 Prerequisites | 2 |
| 2 Introduction | 3 |
| 3 Installation | 5 |
| 4 Hello world | 7 |
| 5 HTML | 8 |
| 6 Configuration | 10 |
| 7 Hello world: First Python CGI Script | 11 |
| 8 Python Errors | 22 |
| 9 Apache Access and Error Logs | 29 |
| 10 Client–Server Interaction | 31 |
| 11 Server-Interpreter Communication | 38 |
| 12 Read and Write a File | 42 |
| 13 Forms: Getting a Value | 44 |
| 14 Forms: Getting a List of Values | 53 |
| 15 Forms: Getting a File | 55 |
| 16 Form: Hidden Field | 59 |
| 17 Building Your Own Query String | 63 |

| | |
|------------------------------------|-----------|
| 18 Query String and Caching | 67 |
| 19 Cookie | 69 |
| 20 CGI | 75 |
| 21 Session | 81 |

File: prerequisites.tex

1 Prerequisites

I assume the following:

- I assume you have taken CISS240, CISS245, and CISS350.
- You have downloaded my Fedora virtual machine. This set of notes uses my Fedora 14 virtual machine. However most of what's in this set of notes applies to all Fedora machines. In fact most of what's in the notes applies to most Unix systems. You are welcome to use your own Linux machines, but you'll be on your own.
- You have already read my *VMware Player/Workstation and Fedora Virtual Machine* notes.
- You have already read my *Unix Part 1* notes.
- I have a set of notes on *Emacs/XEmacs* but that's optional. You're OK as long as you know how to edit text files in linux.
- You have read my notes on *Python Programming*. Initially, I will only be using procedural programming so you can get into this set of notes quickly.

File: intro.tex

2 Introduction

The Apache HTTP Server, commonly referred to as Apache, is a web server software notable for playing a key role in the initial growth of the World Wide Web. In 2009 it became the first web server software to surpass the 100 million website milestone. Apache was the first viable alternative to the Netscape Communications Corporation web server (currently named Oracle iPlanet Web Server), and since has evolved to dominate other web servers in terms of functionality and performance. Typically Apache is run on a Unix-like operating system.

Apache is developed and maintained by an open community of developers under the auspices of the Apache Software Foundation. The application is available for a wide variety of operating systems, including Unix, FreeBSD, Linux, Solaris, Novell NetWare, Mac OS X, Microsoft Windows, OS/2, TPF, and eComStation. Released under the Apache License, Apache is open-source software.

Apache was originally based on NCSA HTTPd code. The NCSA code has since been removed from Apache, due to a rewrite.

Since April 1996 Apache has been the most popular HTTP server software in use. As of March 2012 Apache was estimated to serve 57.46% of all active websites and 65.24% of the top servers across all domains.

– wikipedia

This set of notes is an introduction to the engineering/development of web applications. I will start off using Apache and Python. However I will cover other web servers, web application servers, web frameworks, and web development using other languages including C++, Java, PHP, Perl, etc.

Exercise 2.1.

- Read the whole wikipedia page for Apache HTTP Server.
- Find the Apache HTTP Server homepage.
- Look for the user contributed Apache HTTP Server wiki [HINT: Refer to the previous question.]
- Look for the official Apache HTTP Server documentation page.
- What is the programming language used to write the Apache HTTP Server? [HINT: If you're a strong C++ programming, you should be able to study the Apache HTTP Server source code.] □

Exercise 2.2. Search for Apache HTTP Server web forums, mailing lists, newsgroups, message boards, etc. Look for one that is responsive. Besides personal study/research, it's extremely important that you learn to ask questions on the web through web forums. Once you're more knowledgeable, you should become a contributor and pay back. My personal rule: for each of your question that is answered, help to answer three questions. \square

This is an introduction to Apache HTTP Server. It's impossible to teach you everything about this complex software even from a user/administrator point of view. You can of course study this from the source code point of view – the source code is open to the public since this software is open source. Furthermore it's a complex evolving software.

One of the problems of many web programmers is that many of them only know how to use this or that library or framework or web programming tool to get this or that effect on the browser or on the server side. This is one of the problems when one starts web programming by using a heavy duty framework. When the unexpected happens, such a person is totally confused because the unexpected event might occurs deep inside the framework or in the network subsystem or the operating system. I will try to address some (not all – the whole stack of technologies is way too huge) of the subsystems so that you have a more complete understanding of web programming not only in using a web programming framework but also web programming in the context of network and operating systems. With this knowledge, you will see deeper things, debug better, and be better prepared to learn more about web and internet engineering on your own.

From now on, I will usually use “apache” to denote “Apache HTTP Server”.

File: installation.tex

3 Installation

As root, execute the following commands in a shell:

```
yum install -y httpd
/sbin/chkconfig httpd on
service httpd start
```

The above commands

- installs apache
- enables the httpd service (it will start on reboot)
- starts the httpd

To test that apache is running, point your browser to <http://localhost>. You should get the Fedora test page.

Exercise 3.1. The first question you should ask is “*What is this httpd?!?*” That’s the actual program name of the Apache HTTP Server. In fact, go to your shell and read the man pages for httpd. If you don’t know what I’m talking about ... it’s time to slap yourself 5 times and read my *Unix Part 1* right away. □

Without going into details, you can think of daemons and services as programs that can run in the background and is not attached to a regular user. “Regular user” as in like you or I when we log in. You can make them start up automatically when the system boots/reboots. The **d** in **httpd** stands for daemon. httpd is a daemon. (That’s good enough. For more information refer to my notes on OS.)

From now on, when I say apache, I might be referring to the software product/project Apache HTTP Server or I might be referring to the program httpd.

Exercise 3.2. What does **chkconfig** do? What happens when you execute

```
/sbin/chkconfig httpd off
```

□

Exercise 3.3. What does **service** do? What happens when you execute

```
service httpd stop
```

□

Exercise 3.4. You can also point your browser to <http://127.0.0.1>. What does “127.0.0.1”

mean?



File: helloworld.tex

4 Hello world

Open your browser and go to <http://localhost/helloworld.html>. You should get a “Not Found” error. That’s not surprising. You haven’t written the web page `helloworld.html` yet!!! Duh!!! You don’t expect web pages to spontaneously come into being, right?

The first question to ask is ... where do you put this web page?

Go to `/var/www/html` and add the following `helloworld.html` file as root:

```
<html>
  <head>
    <title>hello world</title>
  </head>

  <body>
    <h1>hello world</h1>
  </body>
</html>
```

Open your browser and go to <http://localhost/helloworld.html>. Now the browser should show you a hello world web page, i.e., the one you just created.

Exercise 4.1. How do you fire up your firefox browser from the command line? □

A really important thing to note is the following. Without any deep understand of the inner workings of apache, you see that apache maps the URL <http://localhost/helloworld.html> to the actual file `/var/www/html/helloworld.html`.

Remember that, unless otherwise stated, from now on all static web pages go into `/var/www/html`. When I say static web pages I mean web pages like the above where the content is static. Some web page (or web server pages) can contain program code and can look different each time you load the page.

Exercise 4.2. Write a couple of web pages of your own. Test them. □

File: html.tex

5 HTML

Here's out hello world again:

```
<html>
  <head>
    <title>hello world</title>
  </head>
  <body>
    <h1>hello world</h1>
  </body>
</html>
```

It's clear those `<...>` have special meaning. These are called HTML tags.

Try this version of hello world:

```
<html>
  <head>
    <title>hello world</title>
  </head>
  <body>
    <h1>hello world</h1>

    This is a test. <br>

    <p>
      Test is a <b>test</b>.
    </p>

    <p>
      Test is a <em>test</em>.
    </p>

    <ul>
      <li> This is a test.
      <li> This is a test.
      <li> This is a test.
    </ul>

    <ol>
```

```
<li> This is a test.  
<li> This is a test.  
<li> This is a test.  
</ol>  
  
<table border='1'>  
<tr><td>This </td><td>is </td><td>a test </td></tr>  
<tr><td>That </td><td>was </td><td>a test </td></tr>  
<tr><td>Those</td><td>were</td><td>the tests</td></tr>  
</table>  
</body>  
</html>
```

Exercise 5.1. Google HTML read about it, its history and use. Learn as much as you can about HTML – you have 5 hours. <http://www.w3schools.com> has lots of stuff that you should know – look for HTML, after your done, look for CSS. □

```
File: configuration.tex
```

6 Configuration

Exercise 6.1. What is SELinux? A short answer is good enough. (Hint: Google knows everything.) ☐

If SELinux is installed on your machine, occasionally some operations might be disallowed. You can then manually allow the operations one at a time. (The GUI that alerts you to the SELinux violation will give you instructions on how to do that.). The SELinux default policy is rather strict. For instance by default, your SELinux configuration might not allow web programs to write to files. However this might be a useful debugging activity.

SELinux maintain a list of boolean flags that indicates whether an activity is allowed or not. To show all these flags, do this in your shell:

```
getsebool -a | less
```

You can scroll up and down to see all of them. For instance one of the flags is `httpd_enable_cgi`. See it? Next to each flag is a boolean value which is either on or off. Type `q` to quit viewing the flags.

Exercise 6.2. Use the web and figure out how to turn the `httpd_enable_cgi` flag on or off permanently. ☐

Managing each flag of SELinux is very time consuming. So we will turn off SELinux on our machines just for development purposes. Note that when the web app is deployed on a live server, the server might have to run with SELinux turned on. In that case, one would need to go over the SELinux policy and see what security flag should be turned on or off.

To disable SELinux permanently open the file `etc/selinux/config` and make sure you have this line:

```
SELINUX=disabled
```

Save and exit the editor. Reboot. You can of course turn on SELinux again at a later point in time.

To disable SELinux using GUI, do System > Administration > SELinux Management, enter root password, Select Status (on the left) and do the obvious. Close the window and reboot.

File: python-hello-world.tex

7 Hello world: First Python CGI Script

Go to `/var/www/cgi-bin/` and create the follow file (the filename is in the program):

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

# File: helloworld-1.py

import cgi
cgi.enable()

print r"""Content-Type: text/plain;charset=utf-8

hello world from python
"""
```

Now open your browser and point it to <http://localhost/cgi-bin/helloworld-1.py> ... and ... you see an error:



Make sure you read (and remember!) the error. Take note of the message at the top. You are getting an HTTP Error 500 Status Code. Also known as the internal server error. Remember this error!

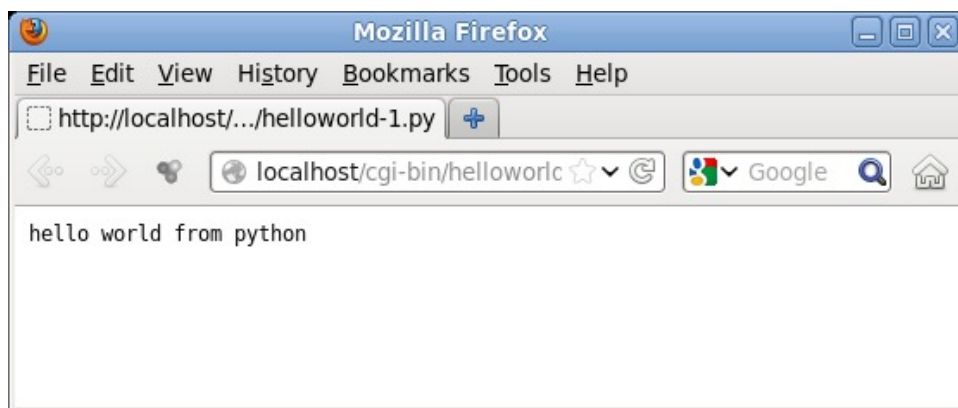
In this case, the problem is that the file is not executable. (There are other situations that can give rise to error 500.) We need to modify the file permission of `helloworld-1.py` so that it can be executed by the apache web server. In the shell do either this:

```
chmod a+x helloworld-1.py
```

or this:

```
chmod 755 helloworld-1.py
```

Now reload your browser and ... TADA! ... you see:



in the browser. You have just written your first python web program!!!

Your file need not end with `.py`. Change the file name to `helloworld-1`:

```
mv helloworld-1.py helloworld-1
```

Point your browser to <http://localhost/cgi-bin/helloworld-1> and again it works.

FILE EXTENSIONS ASIDE. It's up to you if you prefer to use file extensions (such as `.py`) or not. If you do, you are telling others that you're writing your web programs in python, something that hackers can use against you. In the world of security and cryptography, this is called security by obscurity. It's a very weak/lousy form of security. I won't address such issues in this set of notes or I will never finish web programming. Keeping the file extension allows some editors to figure out the programming language used and therefore provide correct syntax highlighting, auto-indentation, etc. I'll leave it to you to find out more about hiding file extensions on your own – You can actually keep the file extension but hide it from the web browser. I will continue to use file extensions. And now ... back to python web programming!!!

The above sends a text file to the browser (look at the `text/plain`). Let's change it so that the python program returns a web page:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: helloworld-2.py

import cgi
cgi.enable()

print """Content-Type: text/html;charset=utf-8

<html>
<h1>helloworld-2.py</h1>
<body>
hello world from python
</body>
</html>
"""
```

(Look at the text/html.) And we test it by pointing the browser now to <http://localhost/cgi-bin/helloworld-2.py> and we get



You can of course execute any computations you like using python and then use the results to build the web page. The next few exercises involves building web pages that requires basic computations.

Exercise 7.1. The following program shows you how to create a datetime object that represents the current datetime:

```
import datetime
now = datetime.datetime.now()
print now
```

Create a python script `helloworld-3.py` that gives you the following web page:



Of course the datetime on the web page will be different for you and if you continually reload the browser you will get different datetimes.

Answer on the next page ... peek only after you have tried the exercise ...

WARNING!!! ... SPOILERS ...

ANSWER:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

# File: helloworld-3.py

import cgi
cgi.enable()

import datetime
now = datetime.datetime.now()

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>helloworld-3.py</h1>
<body>
hello world from python ... it is currently %s
</body>
</html>
""" % now
```

Test: <http://localhost/cgi-bin/helloworld-3.py>

Exercise 7.2. This is a test that you can import modules within python scripts. Create the following python module in your `cgi-bin` directory:

```
# File: mydatetime.py

import datetime

def now():
    return datetime.datetime.now()
```

Using your answer to the previous exercise, write another script `helloworld-4.py` that uses this module.

WARNING ... Answer on next page ...

WARNING ... INCOMING SPOILERS !!! ...

ANSWER:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

# File: helloworld-4.py

import cgi
cgi.enable()

import datetime

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>helloworld-4.py</h1>
<body>
hello world from python ... it is currently %s
</body>
</html>
""" % datetime.datetime.now()
```

Test: <http://localhost/cgi-bin/helloworld-4.py>

Exercise 7.3. You know that when you open a Unix shell, you have two things: a user (so that the system knows what permissions you have) and your current working directory (so that the system knows your default path). Here's a python program to get the user name and the working directory.

```
import os, pwd
user = pwd.getpwuid(os.getuid())[0]
cwd = os.getcwd()
print user, cwd
```

Write a python web program `helloworld-5.py` that prints the user and cwd as above. Access `helloworld-5.py` with a browser and you will find out “who” is running the program and from “where”. (The “where” part is not surprising.)

The answer is on the next page ...

The “where” part is important because of relative paths. If you're at `/x/y/z/` and you try to read the file `abc.txt`, then you're really attempting to read using the full path `/x/y/z/abc.txt`. In the case of importing python modules, if you're at `/x/y/z/` and you execute

```
import xyz
```

the python interpreter will attempt to look for the file `abc.py` in several directories including your current working directory. The list of directories that python use to search for the file is stored in the variable `sys.path`. If you know that `abc.py` is actually in `/x/y/z/`, and you know that this directory is not in `sys.path`, then you should add the path to `sys.path` like this:

```
import sys
sys.path.append('/x/y/z/')
import xyz
```

***** SPOILERS !!! ***** SPOILERS !!! ***** SPOILERS !!! *****

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

# File: helloworld-5.py

import cgi
cgi.enable()

import os, pwd
user = pwd.getpwuid(os.getuid())[0]
cwd = os.getcwd()
print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>helloworld-5.py</h1>
user: %s
<br>
cwd: %s

</body>
</html>
""" % (user, cwd)
```

Test: <http://localhost/cgi-bin/helloworld-5.py>

Screenshot:



AHA! So apache is running httpd. Previously we allow global executable permission to our

python web program. If you want to have a tighter control, you can change the owner of the program to **apache** and only allow executable permission to **apache**.

Exercise 7.4. What are the commands to change the ownership of **helloworld-5.py** to **apache** and to allow executable permission to **apache**? □

Now we check if we can store scripts in subdirectories.

Exercise 7.5. In `cgi-bin`, create a directory named `x`. Copy `helloworld-2.py` to `x`. What happens when you open your browser to <http://localhost/cgi-bin/x/helloworld-2.py>? ☐

Exercise 7.6. Let's try another program: Copy `helloworld-4.py` in `cgi-bin` to `cgi-bin/x`. What happens when you open your browser to <http://localhost/cgi-bin/x/helloworld-4.py>? Read the error message carefully. What does it tell you? Fix the problem! The module used is in the parent directory. ☐

File: errors.tex

8 Python Errors

You can skip this section if you have never written bugs and debugging is something that you never do.

For the rest of us ... read on!!!

Let's focus on Python program errors first. (There are other types of errors, for instance do you remember the HTTP 500 error?)

In your python programs you see the two lines:

```
import cgitb
cgitb.enable()
```

You should always let these two lines be the first two python statements. If you prefer, you can also do the following since the semicolon is just a separator for python statements:

```
import cgitb; cgitb.enable()
```

The `cgitb` is a tool for catching and printing exceptions, tracebacks, some code, etc. (The `tb` in the name of the module stands for traceback, not tuberculosis.) It's to help you debug your program. You have already seen this in one of the exercises in the previous section. It's actually not necessary for your python web programs to work. Once your python web program works, you can comment out the above two lines. In fact you should do it when the web program goes live. Otherwise hackers will find out that you're using python and the more the hackers know about the internals of your web application, the more trigger happy they become.

Exercise 8.1. Go back to the previous section and pick any program. and comment out

```
import cgitb
cgitb.enable()
```

like this:

```
#import cgitb
#cgitb.enable()
```

Save it and load it from your browser. You will see that it still works. □

The above exercise shows you that as long as you have any script in any language that prints

`Content-Type: text/html; charset=utf-8`

(note the blank line!!!) follows by some web content, it will work as a web script.

Exercise 8.2. Save the following in `cgi-bin`:

Point your browser to it: <http://localhost/cgi-bin/python-error2-1.py>. (Did you change the file permission?) You should get an error. Comment out *one* line of code using the pound character (i.e., `#`) to remove the error and get the “OK!!!” message. Test it.

Exercise 8.3. Correct this program:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: python-errors-2.py

import cgi; cgi.enable()

initial = 1

x = initial
for i in range(100):
    x = (x + 2.0 / X) / 2.0

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>python-errors-2.py</h1>

Square root of 2 using the Babylonian algorithm with an initial guess of
%s is

<p style="text-align:center">%s</p>

</body>
</html>
""" % (initial, x)
```

(Test: <http://localhost/cgi-bin/python-errors-2.py>) so that it gives you this output in the browser:



Exercise 8.4. Correct and then complete this program:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: python-errors-3.py

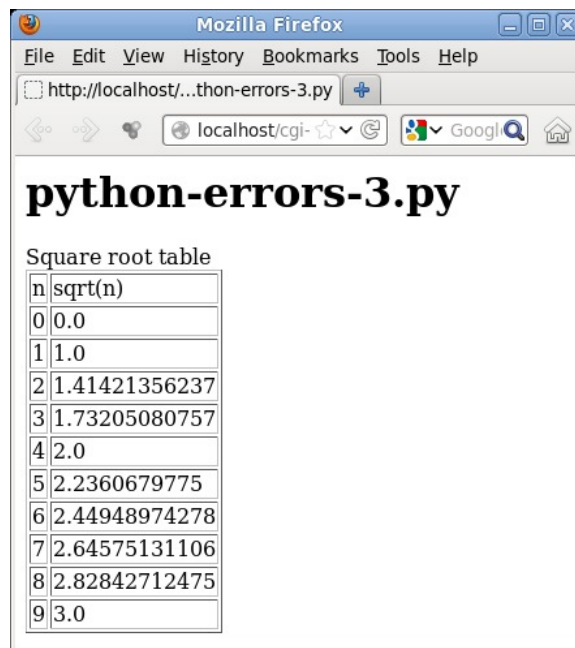
import cgi; cgi.enable()
import math

table = '<table border=1>\n'
table += '<tr><td>n</td><td>sqrt(n)</td></tr>\n'
for i in range(10):
    table += ' '
table += '</table>'

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>python-errors-3.py</h1>
Square root table
%s
</body>
</html>
""" % table
```

so that it gives you this output in the browser:



*** SPOILERS!!! *** SPOILERS!!! *** SPOILER!!! ***

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: python-errors-3-answer.py

import cgi; cgi.enable()

import math
table = '<table border=1>\n'
table += '<tr><td>n</td><td>sqrt(n)</td></tr>\n'

for i in range(10):
    table += '<tr><td>%s</td><td>%s</td></tr>' % (i, math.sqrt(i))
table += '</table>'

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>python-errors-3-answer.py</h1>
Square root table
%s
</body>
</html>
""" % table
```

Test: <http://localhost/cgi-bin/python-errors-3-answer.py>

Exercise 8.5. Note that `cgitb` cannot catch all errors. The following is an example where there's a (python) *syntax* error in the program but `cgitb` does not catch it. Save this as `python-errors-4.py` and point your browser and it out:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

# File: python-errors-4.py

import cgitb
cgitb.enable()

x = 42

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>pyhon-errors.py</h1>

The answer to life, universe and everything is ... %s
</body>
</html>
""" %
```

If you suspect that there's a python error and `cgitb` is not catching it (or you're trying to eliminate this possibility), you can do this: Run the program in the shell:

```
./python-errors-4.py
```

Do this now. Fix the problem! Test it!

Why is it that in this case, you don't see a nice printout of the python error? The reason is simple: The `cgitb` will attempt to catch exceptions, format a nice looking report, and send it to the browser. However that assumes that python is running properly in order for `cgitb` to work!!! When your browser attempts to go to <http://localhost/cgi-bin/python-errors-4.py>!, apache will locate the web program (script) `python-errors-4.py`. After finding it (if it exists), apache will realize that this is a python script (because of the first line of the script), and run the script with the python interpreter. However in this case, because the error is a *syntax* error and not a runtime error, the interpreter *can't even run* the program. So of course `cgitb` can't even catch an error to show to the browser. Get it?

File: other-errors.tex

9 Apache Access and Error Logs

For apache, there are two types of log files which are very important. They are found in directory:

/var/log/httpd

If you do an `ls` you will see something like this:

```
[root@localhost httpd]# ls -la
total 376
drwx-----.  2 root root   4096 Jul 22 12:38 .
drwxr-xr-x. 18 root root   4096 Jul 26 12:15 ..
-rw-r--r--.  1 root root 182760 Jul 26 22:53 access_log
-rw-r--r--.  1 root root 26729 Jan  6 2012 access_log-20120108
-rw-r--r--.  1 root root  3375 Mar 17 23:14 access_log-20120318
-rw-r--r--.  1 root root   712 Jul 14 23:48 access_log-20120715
-rw-r--r--.  1 root root 33366 Jul 20 09:41 access_log-20120722
-rw-r--r--.  1 root root 52039 Jul 26 22:53 error_log
-rw-r--r--.  1 root root  6607 Jul  1 09:21 error_log-20120701
-rw-r--r--.  1 root root  4881 Jul  8 09:17 error_log-20120708
-rw-r--r--.  1 root root 11485 Jul 15 08:37 error_log-20120715
-rw-r--r--.  1 root root 19545 Jul 22 12:38 error_log-20120722
```

Here's the last few line of `access_log`:

```
127.0.0.1 -- [27/Jul/2012:00:30:02 -0500] "GET /cgi-bin/python-errors-4.py HTTP/1.1" 200 121 "-" "Mozilla/5.0 (X11; Linux i686; rv:14.0) Gecko/20100101 Firefox/14.0.1"
```

It gives you information on browser information and when a browser access the apache server.

The `error_log` gives you error messages from apache and is more helpful for debugging program errors:

```
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] File "  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] /var/www/cgi-bin/python-errors-4.py  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] ", line  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] 21  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1]  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] """ %  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1]  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1]  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1]  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1]  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1]  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] ~  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] SyntaxError  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] :  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] invalid syntax  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1]  
[Thu Jul 26 22:53:02 2012] [error] [client 127.0.0.1] Premature end of script headers: python-errors-4.py
```

Notice that the last few lines are the python syntax errors from `python-errors-4.py` of the previous section.

File: client-server-interaction.tex

10 Client–Server Interaction

From the simple experiments and some comments above, it should be clear that the python web programs are executed by apache and therefore execute on the server side. The client (i.e., the browser) receives a string that the python web programs printed out and draws a web page from the string. Remember: the python web programs live on the server and the execution of the python web program occurs on the server. Not on the client!

Let me go over the client-server interaction in more detail. I will use `helloworld-1.py` (our first python web program) as our example:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

import cgi
cgi.enable()

print r"""Content-Type: text/plain;charset=utf-8

hello world from python
"""
```

Suppose the browser is running on a client machine far away (say Russia). Suppose also that both the client machine and my machine are connected to the internet. This means that both machine are given and are identified by IP addresses. (The actual situation is more complicated.) Suppose the client IP address is 1.2.3.4 and the IP address of my server is 5.6.7.8. The IP addresses are for machines to identify each other.

Each machine can run multiple processes. This means that for multiple processes on a machine to communicate with processes running on other machines, there has to be more than just one communication channels at each machine. (At least for modern day machines!) This is achieved by ports. Without going into details, you need to know that two processes, one at each of two machines, can open up ports and send messages (i.e., character arrays) through these ports so that the two processes can communicate with each other. Each machine can have many open ports at the same time. A process can open multiple ports.

Now suppose the user at the client machine enters the URL address

`http://5.6.7.8/cgi-bin/helloworld-1.py`

into the browser. The client sends a message (which is just a string ... as in a character array) to the machine with IP address 5.6.7.8 at a specific port. By default, the browser will send it to port 80. 80 is the default HTTP port, i.e., the default port used by a web server is 80. In other words, the browser will send the message to 5.6.7.8:80. The client machine might be using port say 12345. So we have 1.2.3.4:12345 communicating with 5.6.7.8:80:

You can think of the (IP address, port) pair as a (phone number, extension) pair for communication if you like. You can think of a bi-direction channel with one end attached to 1.2.3.4:12345 (the client side) and another attached to 5.6.7.8:80 (the server side). (Again, the real situation is more complicated.)

Exercise 10.1. Can you configure your apache server to listen to port 81 instead? ☐

By the way, we usually use a domain name such as `www.google.com` instead of an IP address. That's because it's easier to remember the domain name and also the IP address might change. If the address entered into the browser contains a domain name, then there's actually a translation from the domain names to IP addresses. This is called domain name translation.

Notice that in our examples in the previous sections we use `http://localhost`. The `localhost` means *this machine*. The IP address of `localhost` is by default 127.0.0.1. This is also called the local loopback. Again, on any machine *X*, `localhost` means *that machine X* itself.

Exercise 10.2. What does DNS stand for? How is the data for domain name translation stored? ☐

The message sent from the client is called an HTTP Request. This message of course includes the string `http://5.6.7.8/cgi-bin/helloworld-1.py`. The web server analyses the string and extracts `cgi-bin/helloworld-1.py` and knows that the request is to execute the file `helloworld-1.py` in the `cgi-bin` directory. The directory `cgi-bin` is the one in the directory `/var/www` which is the web server's configuration and can be changed. In other words, the mapping between `cgi-bin/helloworld-1.py` is a file on the server's file system can be configured.

Exercise 10.3. Can you configure your apache server so that

`http://localhost/cgi-bin/helloworld-1.py`

maps to, for instance, `/var/www/foobar/helloworld-1.py`? ☐

The first line of the script

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

import cgi
cgi.enable()

print r"""Content-Type: text/plain;charset=utf-8

hello world from python
"""
```

tells apache that the script should be executed by the python interpreter. The apache then executes the script and captures the output from the stdout file stream to get this string:

```
Content-Type: text/plain;charset=utf-8

hello world from python
```

or in C-string notation:

```
"Content-Type: text/plain;charset=utf-8\n\nhello world from python"
```

and sends this string to the (IP address, port) pair of 1.2.3.4:12345. The process at this point is of course the browser. The browser analyses the string and acts accordingly, i.e., it displays

```
hello world from python
```

in its window.

As I mentioned above the real picture is a lot more complicated. For instance (and you should have realized this problem), if the server is always at the same IP address and the port it uses is always 80, doesn't it mean that another client cannot connect to the same IP address and the same port? Doesn't that mean that while a client is communicating with the server no one else can even connect? Doesn't that mean that browser heavily used (i.e., popular) web site at a web server keep giving problems? Well ... actually the server maintains a *queue* of HTTP Requests. Of course, it's still possible to get some kind of "server is unavailable/busy" error when the queue is full.

Besides the presence of a queue of HTTP Request messages, there are other ways to make the web server responsive. For instance the web server can be (usually is!) a multithreaded program. This means that there might be a main process running, pulling requests from the queue and giving the actual work (i.e., executing the web script/program and sending the results back) to another process or to a thread. In general, a server (web server or database server or whatever-have-you-server) tends to have a main process that pulls out requests/work/tasks from a queue and then assign each task to the next available thread in

a pool of worker threads.

[Redirect, proxy, ...]

Note that in order for the client and server to make sense of the data in the strings passing back and forth, there must be an agreement on the “meaning” of the different substrings. For instance in the HTTP reply string

```
"Content-Type: text/plain;charset=utf-8\n\nhello world from python"
```

notice that there are two newline characters between the substring

```
"Content-Type: text/plain;charset=utf-8"
```

and

```
"hello world from python"
```

The part before the two newline characters makes up the so-called **HTTP Headers**. In particular the above is an HTTP Response Header. Notice the plural: “headers”. The HTTP Headers can have several parts, each part with a different meaning. Our examples up to this point has a very simple HTTP Headers section.

The *agreement* on how to interpret “parts” of the strings going between the client and server is an example of a communication **protocol**. In our context, this is the **HTTP Protocol**. I should really just say HTTP since the P stands for protocol.

Exercise 10.4. What does HTTP stand for? ☐

Exercise 10.5. Using any of our previous examples, instead of having two newline characters after the HTTP Headers, remove one newline. Save the file and point your browser to the script. Does the script work correctly? ☐

Understanding the HTTP will help you greatly in writing web applications. One of the tools provided by many web frameworks is basically to either take objects and forming HTTP string to be sent to the recipient or receiving an HTTP string, interpret the parts of the string, and form objects for the web developer.

For instance, you must have heard of “cookies”. They are just substrings going between client and server with the client storing it on the client machine. Once you have read the HTTP for the format of the cookie substring and where to put in the HTTP Response to be sent to the client, you would have complete grasp of the concept.

And how does a server figure out what browser you’re using? I’m sure you have the experience that some web sites sends you a message telling you to use the Internet Explorer or telling you to upgrade your Firefox to a newer version. (By the way, you know now that a web site cannot send you a message – it’s the web server that sends message.) Remember: only

strings go between the client and the server. The server *cannot* (absolutely cannot) look at the code of your browser to deduce the type of browser you use. Well ... it's because your browser sends this information to the server as part of the HTTP Request. (This is optional – a browser can choose not to send browser information.)

Exercise 10.6. Look for the HTTP specification on the web and study the cookie specification. □

Once you understand that HTTP communication is just about sending bytes between the client and the server and that bytes must follow the HTTP specification, then in fact you can communicate with apache even without a web browser. All you need is a software that allows you to send and receive data. I'll going to show you how to do that using **telnet**.

Exercise 10.7. Read up on telnet. □

Enter the following in your shell:

```
telnet localhost 80
```

and telnet will print this:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

and wait for your input. Enter the following *exactly* as shown below:

```
GET /helloworld.html HTTP/1.1
Host: localhost:80
```

Note the newlines!!!

The server responds with

```
HTTP/1.1 200 OK
Date: Mon, 13 Aug 2012 15:57:11 GMT
Server: Apache/2.2.17 (Fedora)
Last-Modified: Sat, 30 Jul 2011 22:49:32 GMT
ETag: "2e008f-b0-4a951361fa7e9"
Accept-Ranges: bytes
Content-Length: 176
Connection: close
Content-Type: text/html; charset=UTF-8

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>hello world</title>
  </head>

  <body>
    <h1>hello world</h1>
  </body>
</html>
Connection closed by foreign host.
```

The string you sent to the server is the HTTP Request called HTTP GET:

```
GET /helloworld.html HTTP/1.1
Host: localhost:80
```

The string you get back from the server is the HTTP Response:

```
HTTP/1.1 200 OK
Date: Mon, 13 Aug 2012 15:57:11 GMT
Server: Apache/2.2.17 (Fedora)
Last-Modified: Sat, 30 Jul 2011 22:49:32 GMT
ETag: "2e008f-b0-4a951361fa7e9"
Accept-Ranges: bytes
Content-Length: 176
Connection: close
Content-Type: text/html; charset=UTF-8

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>hello world</title>
  </head>

  <body>
    <h1>hello world</h1>
  </body>
</html>
```

The line:

```
Connection closed by foreign host.
```

is from telnet and is not part of the HTTP response from the server. If you're using a browser, the HTTP Response received is also the same except that the browser will only display this to the user (as a web page of course):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>hello world</title>
  </head>

  <body>
    <h1>hello world</h1>
  </body>
</html>
```

File: `environment-variables.tex`

11 Server-Interpreter Communication

In the previous section, I talked about some basic computer network communication and HTTP. That only explains the communication between the client and the server. The server still need to execute python web programs before delivering the resulting web pages to the client. So now I'm going to talk about how the server and communicates with your server/backend programs (or the interpreter that runs the programs.)

Recall from an earlier section, recall that I showed you how to get the user running httpd. (Go ahead and spend a couple of minutes looking for the example and studying it.) This can be considered a system level information. There are frequently other system level information that might be helpful to writing web programs.

The type of python web programs we have been writing are call CGI programs.

Once the web server received an HTTP Request, it must look for the CGI program and run the program with the appropriate interpreter (for us, at this point, that would be the python interpreter.) According to CGI specification, a CGI compliant web server makes a collection of variables available to CGI programs. These include some OS information, web server information, web client information, and HTTP communication data between server and client. Note that for instance if the HTTP Request is due to the submission of a web form, then the web form data is part of the HTTP Request. Therefore a CGI-compliant will make such data available to the CGI web program. Collectively, the server will pass system information and HTTP Request information to the executing CGI program. The system information are passed as what we call *environment* variables.

Different programming language will access the CGI environment variables in different ways. In the case of python, you can obtain the CGI environment variables using `os.environ`.

By the way, the linux operating system also has its own environment variables. Many complex systems do. So don't be confused between environment variables between different systems.

Exercise 11.1. Save this as `env.py` and point your browser to the program.

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: env.py
import cgi; cgi.enable()

import os
env = '<table border=1>'
keys = os.environ.keys()
keys.sort()
for key in keys:
    env += '<tr><td>%s</td><td>%s</td></tr>' % (key, os.environ[key])
env += '</table>'

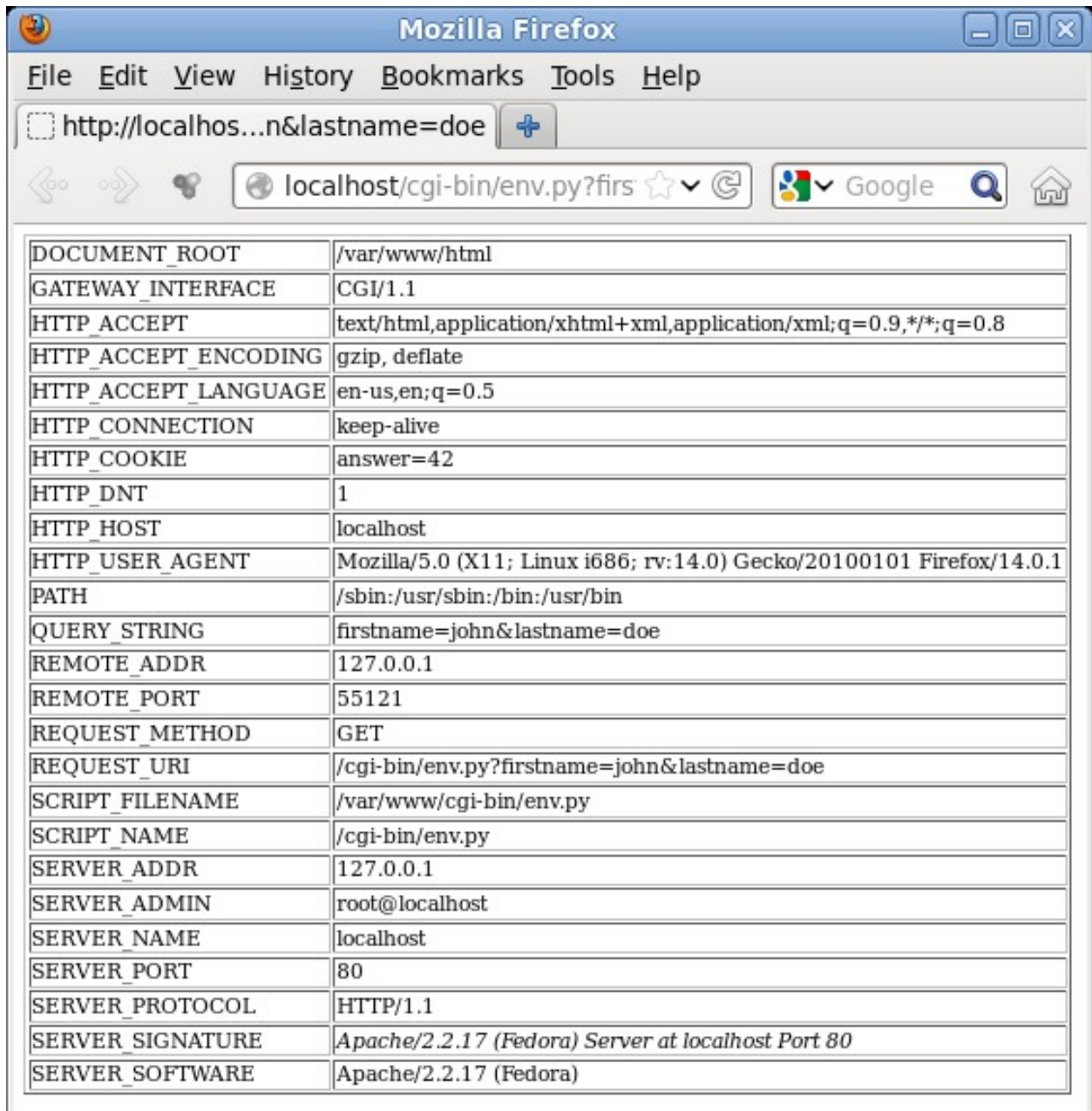
print r"""Content-Type: text/html;charset=utf-8

%s
<br><br>
""" % env
```

Test:

- <http://localhost/cgi-bin/env.py>
- <http://localhost/cgi-bin/env.py?firstname=john&lastname=doe>

Here's the result of the second test above for me:



| | |
|----------------------|--|
| DOCUMENT_ROOT | /var/www/html |
| GATEWAY_INTERFACE | CGI/1.1 |
| HTTP_ACCEPT | text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 |
| HTTP_ACCEPT_ENCODING | gzip, deflate |
| HTTP_ACCEPT_LANGUAGE | en-us,en;q=0.5 |
| HTTP_CONNECTION | keep-alive |
| HTTP_COOKIE | answer=42 |
| HTTP_DNT | 1 |
| HTTP_HOST | localhost |
| HTTP_USER_AGENT | Mozilla/5.0 (X11; Linux i686; rv:14.0) Gecko/20100101 Firefox/14.0.1 |
| PATH | /sbin:/usr/sbin:/bin:/usr/bin |
| QUERY_STRING | firstname=john&lastname=doe |
| REMOTE_ADDR | 127.0.0.1 |
| REMOTE_PORT | 55121 |
| REQUEST_METHOD | GET |
| REQUEST_URI | /cgi-bin/env.py?firstname=john&lastname=doe |
| SCRIPT_FILENAME | /var/www/cgi-bin/env.py |
| SCRIPT_NAME | /cgi-bin/env.py |
| SERVER_ADDR | 127.0.0.1 |
| SERVER_ADMIN | root@localhost |
| SERVER_NAME | localhost |
| SERVER_PORT | 80 |
| SERVER_PROTOCOL | HTTP/1.1 |
| SERVER_SIGNATURE | Apache/2.2.17 (Fedora) Server at localhost Port 80 |
| SERVER_SOFTWARE | Apache/2.2.17 (Fedora) |

Note that although the environment variable name might be different for different libraries, the environment variable names themselves are standardized. For instance the **SERVER_ADDR** which gives you the IP address of the server is one of the standardized names.

Exercise 11.2. From the table above, what browser was used? What is the name of the CGI environment variable that provides this information?

Exercise 11.3. I've already mentioned that the server listens to its port 80. From the

above table, what is the CGI variable name that gives you this information? What about the client? Which port did the client use to communicate with the server?

Exercise 11.4. Occasionally you get a web site that will frighten you by saying something like: "You have been logged. We know everything about you. To prove that, here's your IP address 1.2.3.4. If you don't [... some kind of extortion ...], you will be reported." Write a similar web program.

File: read-write-file.tex

12 Read and Write a File

Create a directory `tmp/` in `cgi-bin`. We will write a program to save a file in the `tmp/` directory. This is obviously helpful in many circumstances including temporary debugging, logging, etc. (For building a large and complex system, obviously better debugging tools and logging subsystems should be used.)

Execute this as root in the shell

```
chown apache tmp
chmod u+rw
```

Save this:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: write-file.py

import cgi; cgi.enable()

f = file('tmp/abc.txt', 'w')
f.write("This is the first line.\nHello world.\nThis is the last line.")

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>write-file.py</h1>

Write to file tmp/abc.txt ...

</body>
</html>
"""
```

and point your browser to <http://localhost/cgi-bin/write-file.py>. Finally, go ahead and check that you do have a file `tmp/abc.txt` in your `cgi-bin`.

Reading a file is easy:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: read-file.py

import cgi; cgi.enable()

f = file('tmp/abc.txt', 'r')
s = f.read()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>read-file.py</h1>

Reading to file tmp/abc.txt.<br />

%s

</body>
</html>
""" % s
```

Test: <http://localhost/cgi-bin/read-file.py>.

File: form-getting-a-value.tex

13 Forms: Getting a Value

Here's what we're going to do in this section. We're going to write a web program that prompts the user for his/her first name. When the user clicks on the submit button, we want the browser to show a web page saying "Hello John" (or Mary or whatever firstname the user entered.)

First save this:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: input-text-1.py
import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>input-text-1.py</h1>
<body>

What is your first name?
<input type='text' value='' name='firstname' />
<input type='submit' />

</body>
</html>
"""
```

Test: <http://localhost/cgi-bin/input-text-1.py>

Point your browser to the script. After entering your first name, you click on the submit button, and ... *nothing happens!* Well ... of course ... that's because you need to tell the browser what to do when the button was clicked. Here's how you do it.

First we have to modify the above program. The new program will have to say something like "when the button is clicked, please execute another script using the firstname entered". Of course we'll have to write this second which will get the first name entered by the user and print it on the web page. Here's the correction to the above program:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: input-text-2.py
import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>input-text-2.py</h1>
<body>

<form method='get' action='input-text-3.py'>
What is your first name?
<input type='text' value='' name='firstname' />
<input type='submit' />
</form>

</body>
</html>
"""
```

And here's the web program that will execute when you click on the button on the web page from the above program:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: input-text-3.py
import cgi; cgi.enable()

import cgi
form = cgi.FieldStorage()
firstname = form.getfirst('firstname', '[NO FIRSTNAME FOUND]')
firstname = cgi.escape(firstname)

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>input-text-3.py</h1>
<body>

Hello %s
</body>
</html>
""" % firstname
```

Now we test this pair of programs by pointing the browser to the first: <http://localhost/cgi-bin/input-text-2.py>

Look at the navigation bar near the top of the browser. Notice that if you entered a name of john, the data in the navigation bar of the browser is

http://localhost/cgi-bin/input-text-3.py?firstname=john

The part of the above URL

firstname=john

is called the **query string**. Note that the URL to the program

http://localhost/cgi-bin/input-text-3.py

and the query string are separated by ?. This is important.

Now try the following which also contains a form like the previous program and also calls the same program to process the form. Do you see the difference between this and the previous program? (HINT: Look for the word `method`.)

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: input-text-4.py

import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>input-text-4.py</h1>
<body>

<form method='post' action='input-text-3.py'>
What is your first name?
<input type='text' value='' name='firstname' />
<input type='submit' />
</form>

</body>
</html>
"""
```

Test: <http://localhost/cgi-bin/input-text-4.py>

Notice that if you entered a first name of john, the data in the navigation bar of the browser is

```
http://localhost/cgi-bin/input-text-3.py
```

The important thing to note is that there is no query string when the action is **post**.

Later I'll talk about the HTTP GET and the HTTP POST actions in more details.

Notice that the data entered by the user can be retrieved by a subsequent web program. The important code that achieves that in the above example is:

```
import cgi
form = cgi.FieldStorage()
firstname = form.getfirst('firstname', '[NO FIRSTNAME FOUND]')
firstname = cgi.escape(firstname)
```

The first parameter of **getfirst** tells python which form variable to use for finding the user's input. The second parameter (if supplied) is the value returned if the form variable is not found.

The `escape` function modifies the string so that some special characters are replaced by special substrings that make HTML possible. For instance if the person enters `'<'` as part of the name, then `'<'` probably won't display correctly. Why? Because the character `'<'` has a special meaning: It's used in HTML tags such as `<input ...>`. To display the character `<` character in HTML, the browser must use `<`. You don't have to remember all the translations. Just use the `escape` function to convert substrings when you want to display it.

It also prevents something called injection attacks. (More on this later.)

By the way, why is the above function to retrieve a form variable called `getfirst`??? Is there a `getsecond`? No ... there's no `getsecond`. In HTTP forms, you can have any number of input tags containing the *same* form variable name. The `getfirst` simply returns the value of the variable whose name matches the first parameter of `getfirst`. The other method is `getlist` that will return *all* the values with variable names matching the first parameter of `getlist`. I'll give you some examples in the next second. For this section, I will only focus on `getfirst`.

It's important to remember that input values you obtain are strings. If the input is to be interpreted as an integer you need to do this:

```
form = cgi.FieldStorage()
age = form.getfirst('age', '-1')
age = int(age)
```

or if it's a float, you should do this:

```
form = cgi.FieldStorage()
height = form.getfirst('height', '-1')
height = float(height)
```

Exercise 13.1. Write a web app that prompts the user for a floating point number, say n and prints the square root of n if $n \geq 0$. If $n < 0$, the web app prints "I cannot do imaginary numbers ... yet." □

Exercise 13.2. Write a web app that prompts the user for two integers say m and n and prints the sum of m and n . If the inputs are not integers, the web app prints "The first input is not an integer!" if only the first input cannot be converted from the input (a string) into an integer. If only the second input cannot be converted to an integer, the web app prints "The second input is not an integer!" If both inputs cannot be converted to integers, the web app should print "Both inputs are not integers!" □

Exercise 13.3. Write a web app that prompts the user for the coefficients of a quadratic polynomial and prints the two roots. Handle the case where the roots are complex. □

Exercise 13.4. Write a web app called “Primality Testing Engine”. It prompts the user for an integer and then tells the user if the integer is a prime or not.

Here’s a series of tests:



Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost/cgi-bin/prime-testing.py

localhost/cgi-bin/prime-testing.py

Primality Testing Engine

Enter a positive integer for prime testing:



Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost/cgi-bin/prime-testing.py?n=1001

localhost/cgi-bin/prime-testing.py?n=1001

Primality Testing Engine

1001 is not a prime

Enter a positive integer for prime testing:



Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost/cgi-bin/prime-testing.py?n=97

localhost/cgi-bin/prime-testing.py?n=97

Primality Testing Engine

97 is a prime

Enter a positive integer for prime testing:

WARNING ... THE ANSWER IS ON THE NEXT PAGE ...

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: prime-testing.py
import cgi, math
import cgi, math
form = cgi.FieldStorage()
n = form.getfirst('n', None)
if n == None:
    result = ''
else:
    try:
        n = int(n)
        if n > 1:
            result = '%s is a prime' % n
            for i in range(2, int(math.sqrt(n) + 1)):
                if n % i == 0:
                    result = '%s is not a prime' % n
                    break
            elif n in [0, 1]:
                result = '%s is not a prime' % n
            else:
                result = 'INPUT ERROR: %s is not a positive integer' % n
    except:
        result = 'INPUT ERROR: "%s" is not an integer' % n
    result += '<hr>'

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>Primality Testing Engine</h1>
<body>
%s
<form method='get' action='prime-testing.py'>
Enter a positive integer for prime testing:
<input type='text' value='' name='n' />
<input type='submit' />
</form>
</body>
</html>
""" % result
```

Test: <http://localhost/cgi-bin/prime-testing.py>

Exercise 13.5. Modify the Prime Testing Engine (call it Prime Testing Engine 2) so that if an integer is not a prime and is greater than 2, it displays the factorization of the integer. Print it for instance in this format:

$$2^2 * 3^4 * 5 * 7^3$$

You will need to know how to do superscripts in HTML.

Exercise 13.6. Write a web app called “Prime Table Generator”. It prompts the user m , n , and c and prints a table of primes p such that $m \leq p \leq n$. The table is made up of c columns.

Here are some screenshots:

Prime Table Generator

m n c

Prime Table Generator

m n c

m=20, n=100, c=10

| | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|
| 23 | 29 | 31 | 37 | 41 | 43 | 47 | 53 | 59 | 61 |
| 67 | 71 | 73 | 79 | 83 | 89 | 97 | | | |

Number of primes: 17

File: form-getting-a-list-of-values.tex

14 Forms: Getting a List of Values

If you have select input tags with the same name, then you can get a list of values for that name. Here's an example using checkboxes:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: get-list-checkbox-1.py

import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>get-list-checkbox-1.py</h1>

<form method='get' action='get-list-checkbox-2.py'>
Where have you been?<br/>
<input type='checkbox' name='visited' value='Earth' />Earth<br>
<input type='checkbox' name='visited' value='Barsoom' />Barsoom<br>
<input type='checkbox' name='visited' value='Middle Earth' />Middle Earth<br>
<input type='submit' /> <br>
</form>

</body>
</html>
"""
```

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: get-list-checkbox-2.py

import cgi
import cgi
form = cgi.FieldStorage()
visited = form.getlist('visited')

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>get-list-checkbox-2.py</h1>

%s <br/>

</body>
</html>
""" % visited
```

Point your browser to <http://localhost/cgi-bin/get-list-checkbox-1.py> and test the above.

Exercise 14.1. Write a web program that prompts the user for three hobbies using input fields of text type with the same form variable name, say `hobbies`. Let the program call another that prints the values of `hobbies`.

File: form-getting-a-file.tex

15 Forms: Getting a File

If the form includes at least one file upload, you do it this ways. Notice that for the form, you must include an `enctype` attribute with value `'multipart/form-data'`. Furthermore, the method must be `post`.

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: get-file-1.py

import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>get-file-1.py</h1>

<form enctype='multipart/form-data'
      method='post'
      action='get-file-2.py'>
Upload a text file:
<input type='file' name='somefile' /><br>
<input type='submit' /> <br>
</form>

</body>
</html>
"""
```



```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: get-file-2.py

import cgi; cgi.enable()

import cgi
form = cgi.FieldStorage()
somefile = form['somefile']
filename = somefile.filename
contents = somefile.file.read()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>get-file-2.py</h1>

The file uploaded has name %s and it contains:<br/><br/>

<pre>
%s
</pre>

</body>
</html>
""" % (filename, contents)
```

Point your browser to <http://localhost/cgi-bin/get-file-1.py> and test that it works. Note that you can also retrieve the name of the file uploaded although there is no way to get the full path of the file.

Exercise 15.1. Test to see if you can perform two file uploads in the same form. ☐

Exercise 15.2. Test to see if you can have a form that allows you to enter a text in an input and a file upload. ☐

It's common if the upload is a huge file you might want to process the file chunks at a time. This will also let you set a maximum upload. This has nothing to do with web programming. Instead of reading the whole file like this:

```
somefile.file.read()
```

you simply read the file specifying a maximum amount to be read at a time:

```
while 1:
    chunk = somefile.file.read(1024) # read 1 KB at a time
    if not chunk: break
    # do some computation with chunk
```

For instance if you want to save the file:

```
import os
f = file('tmp/%s' % os.path.basename(somefile.filename), 'w')
while 1:
    chunk = somefile.file.read(1024) # read 1 KB at a time
    if not chunk: break
    f.write(chunk)
f.close()
```

The previous code will read the whole file. And if the file is extremely huge, this will result in `chunk` using a huge amount of memory. Note that `os.path.basename` will return the filename without the directory part. This will prevent directory traversal attacks. (Most web servers would have already removed the directory part of the filename.)

Or if you want to limit upload file size to 1000000 bytes, you do this:

```
f = file('tmp/%s' % somefile.filename, 'w')
while 1:
    chunk = somefile.file.read(1024) # read 1 KB at a time
    size += len(chunk)
    if not chunk or size > 1000000: break
    f.write(chunk)
f.close()
```

Or if your web app policy is not to allow size > 1000000, then you should obviously delete the file and sends an error web page to the user.

Exercise 15.3. Write a web app that allows a user to upload a text file and perform the following statistical analysis: prints the top five most frequently occurring characters. ☐

Exercise 15.4. Write a web app that allows a user to upload a text file and enter a string. On submission of the form, the web app prints the number of times the string occurs in the text file. ☐

Exercise 15.5. Write a web app that allows the user to upload an image file and allows user to select a new format from a list of radio buttons. The web app convert the file to the new format and returns a web page that contains a link for the user to download the new file. ☐

Exercise 15.6. Write a web app that allows the user to upload a C++ program and an input file and then print the output of the program or errors or warnings provided by the g++ compiler if there are errors in the program.

File: form-hidden-field.tex

16 Form: Hidden Field

All the previous form examples involves data supplied by the user, either through entering the data or file data from the user's storage. There's another type of field that's actually supplied by *not* the user but by the *server*.

Why would you do that? Well think about a shopping cart. Each time a use clicks on an item to buy, the server remembers this item, and then sends a web page to the user to ask him to buy more stuff or to checkout. The web program can continually update the hidden field to include more and more items the user wants to buy.

Here's how to do it. First here's a program that contains a hidden field:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: test-hidden-1.py

import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>test-hidden-1.py</h1>
<body>
<form method='get' action='test-hidden-2.py'>
What is your first name?
<input type='hidden' name='answer' value='42' />
<input type='text' name='firstname' value='' />
<input type='submit' />
</form>
</body>
</html>
"""
```

and here's the script to test that the hidden field was also submitted:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: test-hidden-2.py

import cgi; cgi.enable()

import cgi
form = cgi.FieldStorage()
firstname = form.getfirst('firstname', '[NO FIRSTNAME FOUND]')
firstname = cgi.escape(firstname)
answer = form.getfirst('answer', '[answer NOT FOUND]')
answer = cgi.escape(answer)

print r"""Content-Type: text/html;charset=utf-8

<html>
<h1>test-hidden-2.py</h1>
<body>
firstname: %s <br />
answer: %s <br />
</body>
</html>
""" % (firstname, answer)
```

Test: <http://localhost/cgi-bin/test-hidden-1.py>

You can use `post` for hidden fields too. One thing to note is that if you do an HTTP GET, the form variables are encoding in the URL and can be viewable by others looking at your browser if the navigation bar is opened!

The big picture is that the above method allows you to send two type of information to a web program:

- data entered by the user (in the form of course)
- data from a previous web program (in the query string)

Exercise 16.1. Write a simple addition web app. This consists of 3 programs. The first program prompts the user for x . After the use submits, the second program prompts the user for y . After the user submits the second program, the third program prints x and y and $x + y$.

Exercise 16.2. Modify the previous program so that if the user enters a value that cannot

be converted to a floating point number, the program reprompts the user. This applies both to x and y . Use an HTTP GET.

Exercise 16.3. You have decided to have a startup for people who need psychiatric help. Write a web app that plays the role of a (very simplistic Rogerian) psychiatrist. Ask the user for his/her name, ask him/her how you can help. No matter what he/she, just reply by saying “[Firstname], what do you mean when you say [whatever he/she says]”. For instance if his name is John and he says

I am not happy.

your web app should say

John, why do you say "I am not happy."

To end the psychiatrist session, the user enter “quit” in which case, the web app prints a thank you message. Use HTTP POST.

Exercise 16.4. Modify the previous program by getting your web app to randomly say

[Firstname], can you elaborate?

or

[Firstname], why do you say [whatever he/she says]

or

[Firstname], can you tell me why you say [whatever he/she says]

Etc.

Exercise 16.5. Modify the previous so that your web app can remember more. Make it reference an earlier statement made by the user. For instance:

[Firstname], earlier you said [whatever he/she says].
Can you elaborate on that?

or

[Firstname], let's go back to an earlier point.
Why do you say [whatever he/she says]?

Exercise 16.6. Modify the above. If the user is John and he says

I am not happy.

your web app should say something like

John, why do you say you are not happy?

or if he says

I feel like doing a bungee jump.

your web app should say something like

John, why do you say you feel like doing a bungee jump?

Exercise 16.7. How far can you go with the above psychiatrist web app? For instance to make your psychiatrist more intelligent, you might ask the user his/her age, gender, ... or build a psychological profile of him/her with questions like

John, do you chew your fingernails?

or

John, how often do you have dreams?

or

John, how often do you shower?

or

John, how much time do you spend per week in an online chatroom?

With more information you should be able to build better (interesting? bizarre?) responses. For instance

John, why do you say you are sad since you take three showers a day?

File: building-your-own-query-string.tex

17 Building Your Own Query String

Recall from the previous section that hidden variables allow you to send two type of information to a web program:

- data entered by the user in a form
- data from a previous web program embedded in the query string of the action

Now I'm going to talk about another method for achieving the same goal.

Recall that for a HTTP GET, form variables are attached to the URL as a query string. We can extract (name,value) pairs from the URL using the `cgi` library. Now if you think about it, we can attach (name,value) pairs onto the URL ourselves.

Try the following experiment in the python interpreter. It involves attaching two (name,value) pairs to a URL.

```
import urllib
params = urllib.urlencode({'firstname':'john', 'lastname':'doe'})
print "params:", params
url = 'http://localhost/cgi-bin/x.py?' + params
print "url:", url
```

Look at the above output carefully. Notice that there are special characters used in forming the query string. This is called URL encoding. These special characters have special meanings in HTTP.

Exercise 17.1. What is the character used to separate (name,value) pairs in a query string?

It's possible to read the HTTP specification and then craft your own query strings. But this is error prone. It's better to use `urllib.urlencode` to do it for you.

Exercise 17.2. Using the program above, answer the following questions:

1. What does a space encodes as?
2. What about the less-than character <?
3. What about the greater-than character >?

Now what's the point of knowing the above??? After all, doesn't HTTP GET attach (name,value) pairs from the form for you?

To answer the question, here are two programs:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: build-query-string-1.py

import cgi; cgi.enable()

import urllib
params = urllib.urlencode({'firstname':'john', 'lastname':'doe'})
url = 'http://localhost/cgi-bin/build-query-string-2.py?' + params

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>build-query-string-1.py</h1>

<form method='post' action='%s'>
<input type='text' name='middlename' /><br>
<input type='submit'>
</form>

</body>
</html>
""" % url
```

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: build-query-string-2.py

import cgi; cgi.enable()

import cgi
form = cgi.FieldStorage()
firstname = form.getfirst('firstname', 'no firstname')
lastname = form.getfirst('lastname', 'no lastname')
middlename = form.getfirst('middlename', 'no middlename')

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>
<h1>build-query-string-2.py</h1>

firstname: %s<br />
middlename: %s<br />
lastname: %s<br />

</body>
</html>
""" % (firstname, middlename, lastname)
```

Test: <http://localhost/cgi-bin/build-query-string-1.py>.

Notice that the form in build-query-string-1.py has action

<http://localhost/cgi-bin/build-query-string-2.py?lastname=doe&firstname=john>

Note also that the method is post – this is important.

Notice that although the first program prompts the user for the `middlename`, the second program actually processes `firstname`, `middlename`, and `lastname`. Why? Because the action attribute

<http://localhost/cgi-bin/build-query-string-2.py?lastname=doe&firstname=john>

contains two (name,value) pairs. The method used is HTTP POST. Therefore in the execution of the second program, the program can access two (name, value) pairs from the query string and one (name, value) pair from stdin.

Exercise 17.3. What happens when you use HTTP GET in the above example? ☐

Exercise 17.4. Write a web app that prompts in the first web page the how many hobbies the user has and then if the user enters 5, the web app will prompt him for his 5 hobbies, one on each page, and after collecting all the hobbies, the last page returned is the list of 5 hobbies. Your web app should work for any positive integer besides 5. If on the first page the user did not enter data that can be converted to a positive integer, go back to the first page. Use query strings. ☐

Exercise 17.5. Write web app to play tic-tac-toe against a human player. Make your web app smart enough to play intelligently (you should know what I mean by that). Use the table tag and text to draw the tic-tac-toe board. Use query strings and not hidden variables.

Exercise 17.6. Modify the above by replacing the text-based board with images for each cell of your (suitably sized) table. ☐

Exercise 17.7. Modify the previous program so that it prompts the user for n and then play an n -by- n game of tic-tac-toe with the player. Use query strings and not hidden variables. ☐

File: query-string-and-caching.tex

18 Query String and Caching

Here's a very common problem that causes a lot of frustration to newbie web developer.

Sometimes, when you're debugging your web programs, you will find that no matter what you do to your web program, on reloads, your browser keeps displays the same error again and again.

Of course one of the obvious reasons might be that you're modifying the *wrong* file. (Have you *never* entered the wrong filenames?) But if when you look at your editor it says the filename of the file you're editing *is* indeed the one that you're debugging, then another problem might be that you have *two* files with the same name but they live in different directories (and your editor only displays the filename but not the full path.)

Besides the above obvious (silly!) mistakes there's another one that is slightly more devious and always trip newcomers.

[TODO]

Your browsers might be reading from cached results.

What I mean is that when you type `http://a.b.com/d/e/f/h.html`, not only does the browser fetch the web page at this URL for you, but it also saves the web page together with address `http://a.b.com/d/e/f/h` (on the client's storage or course) so that if you were to reference this page again, instead of performing an HTTP Request for this page, it actually reads the web page from thge local storage. In other words, there was actually *no* HTTP Request, and therefore no matter how you change your web program associated with the address `http://a.b.com/d/e/f/h`, you browser will always display the same content!!!

There are two things you can do. The first is obvious: You would expect your browser to be configurable so that caching can be turned off. All you need to do is to turn off caching.

There's a problem with this approach.

You might not be able to control the caching behavior of other browsers access your web app once it goes live.

You can send an HTTP Response to a browser telling the browser not to cache the result. But there's no guarantee that it will be honored by that (rogue) browser.

Another method is to vary URI. Of course you can't continually change the name of your web program from `x1.py` to `x2.py` to ... That would be absurd. However you can attach some random query string to the URI.

For instance when you generate a HTTP Response that contains a form, instead of doing this:

```
<form method='post'
      action='cgi-bin/googlestockpricetomorrow'>
...
</form>
```

you might have something like this:

```
<form method='post'
      action='cgi-bin/googlestockpricetomorrow?rand=123asdq3412eqwdqwd'>
...
</form>
```

where the `rand` value is randomly generated by your web program. Or, if you're presenting a list of clickable links, you might want to do this:

```
<a href='cgi-bin/howtowalk?rand=123r2ede12eq3i23'> Walk </a><br/>
<a href='cgi-bin/howtoeat?rand=de312sqe12eq3jj3'> Eat </a><br/>
<a href='cgi-bin/howtoswim?rand=1zx312eqe12eq3490'> Swim </a><br/>
<a href='cgi-bin/howtoshower?rand=12tfueqe12eq34mn'>Shower</a><br/>
```

Etc. You get the picture.

Exercise 18.1. How do you turn off web page caching for firefox and IE?

File: cookie.tex

19 Cookie

So far you have seen two different ways for a web program to communicate with another one:

1. Hidden form variables
2. Query strings

Now for a third ... cookies. No, not the edible type.

A cookie is just a small file that a browser can store on the client's storage (example: harddrive). The request and the information to store is initiated on the server side. This instruction is of course in the HTTP Headers section. Let me show you how to do this.

(You can also initiate the command to store cookies on the client side. I won't talk about it here.)

The following will set a cookie with the name of **answer** with value 42:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: cookie-1.py

import cgi; cgi.enable()

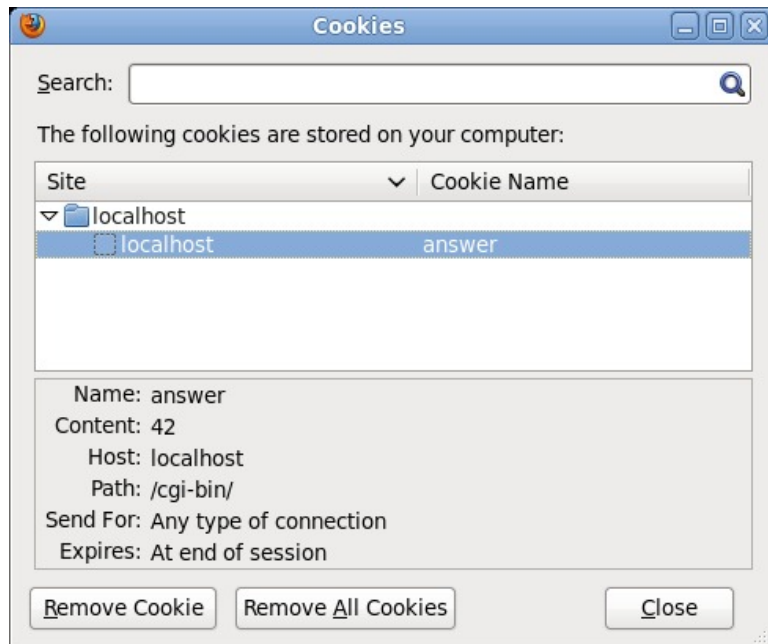
print r"""Set-Cookie: answer=42
Content-Type: text/html; charset=utf-8

<html><body>
<h1>cookie-1.py</h1>

Setting cookie answer=42.

</body></html>
"""
```

Now point your browser to <http://localhost/cgi-bin/cookie-1.py>. After the web page is loaded in your browser, view the cookies for **localhost** in your browser. There should be one with the name of **answer** with value 42. For Firefox, do Edit > Preferences > Privacy > remove individual cookies, click on the left of localhost host and then click on answer and you'll see



So at this point, the browser has stored the cookie **answer=42** for **localhost**. Do not close your browser or the cookie is deleted. We'll need it for the next program.

Every time the browser sends an HTTP Request to the server, all cookies are included in the HTTP Headers. Therefore the server, after analyzing the HTTP Headers, has access to all cookies. The cookies can be retrieved from `os.environ` as shown in the following.

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: cookie-2.py

import cgi; cgi.enable()
import os

print r"""Content-Type: text/html;charset=utf-8

<html><body>
<h1>cookie-2.py</h1>
%s
</body></html>
""" % os.environ.get('HTTP_COOKIE', 'no cookie')
```

Test: <http://localhost/cgi-bin/cookie-2.py>.

Exercise 19.1. What do you get if you do the following in sequence: access <http://>

localhost/cgi-bin/cookie-1.py, close the browser, and finally access <http://localhost/cgi-bin/cookie-2.py>?

Exercise 19.2. Write a program `cookie-3.py` that sets two cookies

`answer=42 and marvin=paranoid.`

(There's only one – maybe two – obvious way to do this. If you can't get it right, do some research on the web.) Access <http://localhost/cgi-bin/cookie-3.py> and check that your program works. Write another program `cookie-4.py` that read the cookie string. Access <http://localhost/cgi-bin/cookie-4.py> and check that it works. What does the cookie string look like?

Exercise 19.3. In the previous exercise, note that there's only *one* string representing *both* cookies. Modify `cookie-4.py` by saving the cookie string as a text file like this

```
cookie = os.environ.get('HTTP_COOKIE', 'no cookie')
f = file('cookie.txt')
f.write(cookie)
f.close()
```

After running `cookie-4.py`, read the file `cookie.txt` and analyze the contents of the file. Write a simple function that returns a dictionary of cookies:

```
def cookie_dictionary(http_cookie):
    cookie = {}
    # ... TODO ...
    return cookie
```

so that you can access the individual cookies easily like this:

```
http_cookie = os.environ.get('HTTP_COOKIE', 'no cookie')
cookie = cookie_dictionary(http_cookie)
# cookie['answer'] = '42' and cookie['marvin'] = 'paranoid'
```

[TODO: Check permission]

Exercise 19.4. Can cookie (on the client side of course) be set several times? In other words if `x.py` sets `a` to 0 and then `y.py` sets `a` to 1, do you get an error? (Are cookies meant to be “constant”?) Or is there no error but the value cannot be changed? Or is there no error and the value is indeed changed?

Exercise 19.5. Suppose a web program sets two cookies (on the client side) and this is followed by setting the value of the first of the above two cookies. Will the second cookie stay or be wiped out?

Exercise 19.6. Is there a limit to the amount of space allowed for cookie storage?

In the above, the cookies expire (i.e., the browser deletes them) immediately when the browser is closed. You can actually specify when the cookie should expire. (Don't forget that users can manually remove cookies any time they like!) There are two ways of doing this. The following program sets a cookie and expires it *60 seconds* after the creation of the cookie. Make sure you test it.

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: cookie-max-age.py

import cgi; cgi.enable()

print r"""Set-Cookie: answer=42; Max-Age=60
Content-Type: text/html; charset=utf-8

<html><body>
<h1>cookie-1.py</h1>

Setting cookie answer=42 with expiration in 60 seconds.

</body></html>
"""
```

You can also set the cookie to expire with an absolute datetime:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: cookie-expires.py

import cgi; cgi.enable()

print r"""Set-Cookie: answer=42; Expires=Mon, 6 Aug 2012 10:50:38 AM CDT
Content-Type: text/html;charset=utf-8

<html><body>
<h1>cookie-1.py</h1>

Setting cookie answer=42 with expiration at
Mon, 6 Aug 2012 10:50:38 AM CDT

</body></html>
"""
```

Note the format of the datetime. Beside “CDT”, you can also use, for instance, “GMT” (Greenwich mean time).

Exercise 19.7. On the server, when you retrieve the cookie string, do you see the expiration datetime?

Of course you would expect python to come with a library for managing cookies. (In fact there are several.) Open your python interpreter and run this

```
import Cookie, time

cookie = Cookie.SimpleCookie()
cookie['lastvisit'] = str(time.time()) # (name,value) pair
cookie['lastvisit']['expires'] = 30 * 24 * 60 * 60
cookie['lastvisit']['path'] = '/cgi-bin/'
cookie['lastvisit']['comment'] = 'The last user\'s visit date'
cookie['lastvisit']['domain'] = '.localhost'
cookie['lastvisit']['max-age'] = 30 * 24 * 60 * 60
cookie['lastvisit']['httponly'] = 1
cookie['lastvisit']['secure'] = 1
cookie['foo'] = 'bar'
print cookie
```

All the library does is to format the cookie string for HTTP Header.

Exercise 19.8. Write a web program to set a cookie using the Cookie library. Test it. There

are several extra cookie attributes in the above example not mentioned earlier. What are they for? (Especially the `secure` attribute.)

Exercise 19.9. use the above library to set two cookies.

The library can also convert HTTP_COOKIE strings into dictionaries. First point your browser to our first cookie program <http://localhost/cgi-bin/cookie-1.py>

```
cookie = Cookie.SimpleCookie()
cookie.load(os.environ.get('HTTP_COOKIE', ''))
#
# The cookie values are:
# cookie['lastvisit'].value
# cookie['answer'].value
#
```

Exercise 19.10. Write a program `cookie-simplecookie-read.py` that uses the above code to get the cookies.

By the way, notice that what is called cookie in the HTTP Request is a string that contains possible several cookies. This can be confusing. So sometimes some people will call the whole HTTP_COOKIE string the cookie but call an individual (name,value) pair a cookie crumb. However it's also common to call an individual (name,value) a cookie. Just watch the context.

There's a big advantage in using cookies compared to hidden variables or query strings: the data stored can last a long time. If the expiration datetime is far enough into the future, you can close the browser and reopen it after a week and continue what you were going at the particular web site.

File: `cgi.tex`

20 CGI

While HTTP is a protocol for communication between a browser and a web server, notice that the construction of the web page lies not with the web server but through the script. A web server that support CGI programming provides two features (there are many others):

First, it provides a way to translate each URL to an actual fullpath of a script. (Besides scripts, the web server can also translate URL/URI to other types of files such as static HTML web pages, image files, and Javascript files.)

Second, it executes the script, passing to it various information. The web server also captures the output from the script, through the stdout file stream, and sends it back to the web browser.

Viewing the script that is executed almost like a function, of course the script needs input which is obtained in two different ways depending on the HTTP method:

- HTTP POST: The script retrieves the user submitted data from stdin. This includes the (name, value) pairs from input tags and, if present, file uploads (which is more complicated.) [CHECK: The form data is sent through the HTTP Headers.]
- HTTP GET: The script accesses the user submitted data from some execution environment provided by the operating system (therefore it depends on the operating system). The cgi-compliant web server places relevant data in this environment. This environment is usually available in most programming languages. For instance, in the case of python the variable `os.environ` is a dictionary providing the environment data. [CHECK: The form data is attached as query string.]

Of course there is always data in the environment not related to user submitted data. For instance browser information is available in the environment.

Exercise 20.1. What does CGI stand for?

Here's an example where the first script performs an HTTP POST and the second script reads the stdin:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: test-cgi-post-stdin-1.py

import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>

<form method='post' action='test-cgi-post-stdin-2.py'>
<input type='text' name='firstname' value='John' />
<input type='text' name='lastname' value='Doe' />
<input type='submit' />
</form>

</body>
</html>
"""
```

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: test-cgi-post-stdin-2.py

import cgi; cgi.enable()

import sys
stdin = sys.stdin.read()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>

stdin: %s

</body>
</html>
""" % stdin
```

You will see the stdin gives the (name, value) pairs from the form. Here's the output (if I

don't change the values already in the fields:

```
stdin: firstname=John&lastname=Doe
```

Now let's see what's in the stdin for the case of HTTP GET.

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: test-cgi-get-stdin-1.py

import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>

<form method='get' action='test-cgi-get-stdin-2.py'>
<input type='text' name='firstname' value='John' />
<input type='text' name='lastname' value='Doe' />
<input type='submit' />
</form>

</body>
</html>
"""
```

The above script calls the following (which is really the same as the second script above):

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: test-cgi-get-stdin-2.py

import cgi; cgi.enable()

import sys
stdin = sys.stdin.read()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>

stdin: %s

</body>
</html>
""" % stdin
```

In this case the stdin gives nothing but an empty string. As mentioned above, for the case of HTTP GET, the form data is retrieved from `os.environment` in the case of using python.

Finally for the case of HTTP POST where the form uses “multipart/form-data” for the enctype attribute is slightly more complicated. The data is still placed in stdin but the data is in several “parts”. Run the following and everything will be clear. Here’s the first script:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: test-cgi-post-multipart-stdin-1.py

import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>

<form enctype="multipart/form-data"
      method='post' action='test-cgi-post-multipart-stdin-2.py'>
<input type='text' name='firstname' value='John' /> </br>
<input type='text' name='lastname' value='Doe' /> </br>
<input type='file' name='file' /> </br>
<input type='submit' />
</form>

</body>
</html>
"""
```



```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
# File: test-cgi-post-multipart-stdin-2.py

import cgi; cgi.enable()

print r"""Content-Type: text/html;charset=utf-8

<html>
<body>

<form enctype="multipart/form-data"
      method='post' action='test-cgi-post-stdin-2.py'>
<input type='text' name='firstname' value='John' /> </br>
<input type='text' name='lastname' value='Doe' /> </br>
<input type='file' name='file' /> </br>
<input type='submit' />
</form>

</body>
</html>
"""
```

This is what I get on the web page:

```
-----98254644411098721642106434066 Content-Disposition: form-data; name="firstname" John
-----98254644411098721642106434066 Content-Disposition: form-data; name="lastname" Doe
-----98254644411098721642106434066 Content-Disposition: form-data; name="file"; filename="afile.txt" Content-Type: text/plain
This is the beginning of the file. Hello world. This is the end of the file.
-----98254644411098721642106434066--
```

Exercise 20.2. Write your own CGI library for python CGI programming using the above information without using the CGI module that comes with your python installation.

File: session.tex

21 Session

HTTP is called a stateless protocol. This simply means that each HTTP communication between the client and server is independent and does not know anything about the previous communications.

This implies that if you need sequence of communications to complete a task, you need to somehow keep track of where you are.

For instance if the web server need to go through a sequence of steps to approve an online home mortgage application, somehow there must be a way for the server to say “I am at step 3 and have verified that the data at this step is correct. Now I’m going to prompt the user for the data for step 4.”

You have already seen that the server can set cookies in the browser and when the browser performs an HTTP GET or POST, the cookies are also sent back to the server. Therefore cookies can be used to coordinate the sequencing of the subtasks and also to keep track of data collected so far in order to complete the task. You can also use hidden form variables and query strings. The advantage of cookies is that they can stay around for a long time.

Note that in both the cookie, hidden form variable, and query string method for creating a stateful computation, the data to be used for the computation is passed between client and server. This will have an impact on the performance if the data is huge.

One way to get around this problem is to store the state of the computation on the server side and just pass some kind of ID to link the sequence of communications to the state of the computation stored on the server side. Remember that web app is usually used by many people. Even if the web app is used by one person, the person might be carrying out many sessions with the web server. For instance the person might be playing 3 tic-tac-toe games simultaneously on a personal web app running on his own laptop.

The sequence of communication between the client and server in when bunched up together to do something meaningful and useful is called a session. The data used for the session is called session data. Again this is usually stored on the server side. (There are exceptions.) The ID used to link the session to the session data on the server is obviously called the session ID.

For instance suppose you’re applying to join the new MARS colony project, you might have entered your firstname, lastname, date of birth, gender, home address, and skills. You still

have to enter your educational background, health, and a host of other ridiculous questions. The communication might pass an ID of 42 back and forth between the server and your browser. On the server side, the information you have already entered (firstname, lastname, date of birth, gender, home address, and skills) is already stored together with the ID value of 42. Suppose the next web page you see asks you for your educational background (since kindergarten!!!) You faithfully entered all that and clicks the submit button. The server receives the information from you as well as information from 1000s of other applicants. Your HTTP Request includes the ID (either as a cookie or as part of the query string).