Lab 2: Fun with Regular Expressions

This lab is due by the end of the lab in which it is issued.
For spring 2015, **do not do part I**, since this is already set up for you. The instructions are just included for information purposes.

Objectives

Upon successful completion of this lab, you will have demonstrated the ability to:

- create an appropriate regular expression to validate a string
- use the `preg_match( )` function to validate form field data

Reference

- See lecture notes.

Required Materials

- None.

Lab Preparation

- Required reading posted on Moodle page. See the notes for an explanation of the set of special characters you need for regular expressions for this course.

Part I - Password protecting web directories

You can password protect directories through the Apache web server, which can be configured to support authentication. Before allowing a user agent to enter a directory, Apache will prompt the user for a user name and password. Authentication is often turned on for administrative programs (like the ones you will be writing for the assignments).

We would like to password protect our `~/public_html` directory.

1. Create a file with the **exact name** .htaccess **in your public_html directory** (note the dot at the beginning of the file name) and place the following code in that file (everything between, but not including, the dashed lines) and change 'username' to your zenit username :

   `AuthUserFile /home/`*`username`*`/.htpasswd`

```
AuthGroupFile /dev/null
AuthName "username's protected files"
AuthType Basic

<Limit GET>
require valid-user
</Limit>
-----------
```

Anyone requesting a web page or program from this directory (or any directories below) will now be prompted for a password. This also tells us where a list of valid users and their passwords can be found: `/home/username/.htpasswd`

2. Now we must create web users and passwords. In the .htaccess file we said we would keep a list of valid users in the `/home/username/.htpasswd file`. For security reasons, this file should never go on a web accessible path. So we will put it in our home directory. CD to *your home directory*.
3. Enter the following command to create the password file and add a user - substitute the _____ with a username of your choice. It can be the same as your zenit username if you want, but the password you choose will still change independently of zenit:

   `htpasswd -c .htpasswd _____`

   where _____ is the user ID you want to create. This is for a web user (someone who wants to browse to a web page or program), so make the name something different than your zenit username. You should be prompted for a password. Write down the user ID and password you created.

   NOTE 0: since Apache needs to access the `.htpasswd` file, and Apache runs as part of the others group, you must set permission for Apache (others) to a) enter or passthrough your home directory (x permission, so 711) and b) to read the .htpasswd file (r permission, so 644). Check to make sure these permissions are set properly before you try to test authorization. If they are not, use the `chmod` command to change permissions.

   NOTE 1: if you wish to add another user to the `.htpasswd` file, do **not** specify the `-c` option. `-c` creates a new file, thus deleting all the users in the old one. So all you need to add another user to the `.htpasswd` file is `htpasswd .htpasswd _____`

   NOTE 2: after you create at least one user id, you must manually edit the **.htpasswd** file, and add the following line to the file by copying and pasting.

   teacher:3BluTlKhWLb2w

   This line will allow me to log in to any student website. Note that the passwords in this file are encrypted, so they will not resemble the actual password you use to log in. The htpasswd program does the encryption for you.

You can test to see if this works by trying to browse to a page in the `public_html` directory. A login box should pop up. Note that once you login, Apache remembers that you are logged in and does not request a password for every page/program you access in the password protected directory and its subdirectories. However, as soon as you close your browser Apache will consider you logged out and will prompt you again the next time you try to browse to this directory.

Part II - Regular Expressions

Introduction

Regular expressions are often used to validate input (values input from the command line, input fields from a web form, command line arguments, etc) since they are a powerful and fast tool for searching a string for patterns. In fact, coding a function without using regular expressions might take several hundred lines of code to accomplish the same thing as one line which makes use of a regular expression!

In PHP, when we write a regular expression, we are always trying to strike a balance between usability (i.e. letting the user have some flexibility in how they enter a string) and security (i.e. not allowing malicious code to be injected through a form input field). For example, there is no reason not to allow your user to have blanks before and after a name entered in a form field. If the blanks bother you, you can use the `trim( )` function to remove them before you put the name in your DB (you should probably do this for all your data, just to be safe). However, if you don't allow the leading and trailing blanks, your user might puzzle over why the string "Boyczuk " does not validate - and since the trailing blank would be invisible to them, this would annoy and frustrate them, potentially losing you a client/member/customer. So you are better to allow them this flexibility, as long as it doesn't compromise security.

Here is a refresher on regular expressions:

- Special characters in regular expressions:
    - . matches one character, any character
    - ^ represents the start of a line
    - $ represents the end of a line
    - [ ] matches a character in the group specified
    - [^] matches anything but the characters in the group specified

- Character classes - these are often misunderstood and misused, and their use is not recommended
    - \s matches a white space character (blank, newline or tab)
    - \d matches a numeric character 0-9 **\D** matches any character except a numeric character
    - \w matches a word character (includes letters, numbers and underscore)

    - \W matches any character that is not a word character

- Quantifiers
    - * matches 0 or more of the specified character
    - + matches 1 or more of the specified character
    - ? matches 0 or 1 of the specified character
    - {n} matches n of the specified character
    - {n,m} matches from n to m occurrences of the specified character

- Choice
    - `( pattern1 | pattern2 )` matches `pattern1` or pattern2, but not both
    - you may have more than 2 choices: pattern1 | pattern2 | pattern3

- To escape special characters place a **** in front if you are trying to match one of the special characters - for example, the **`'\.'`** pattern matches a period, whereas the **`'.'`** pattern matches any character.
- See the PHP documentation for a full list of special characters - make sure you are not using a special character in your regular expression unless you mean to - otherwise the regex will not contain the pattern you think it does!

Some common errors to avoid when using regular expressions:

- If you do not use the `^` and `$` to designate the beginning and end of the string, then `preg_match` will match any substring within the string. That is, the pattern `/abc/` will match `xabcx` since `abc` is contained within `xabcx`. Using ^ and $ forces the user to enter a string that matches exactly what you've designated. So `/^abc$/` would **not** match `xabcx`. Note that allowing non-matching at the start or end of a string may enable someone to launch an injection attack by injecting non-valid strings before or after valid strings.
- Regular expressions make use of the special characters listed in the notes. If you need to pattern match one of these special characters, you must *escape* it so that `preg_match` (the function used to evaluate regular expressions) doesn't think you mean it as a special character. To do this, you put a backslash, \, in front of the character you wish to escape. For example, it is common for people new to regular expressions to forget to escape the period, which is also a special character which means "any" character. So the reg ex `/142.204/` would match `/142!204/` while `/142\.204/` would not.

PHP has a number of functions that work with regular expressions:

http://php.net/manual/en/book.pcre.php. `preg match` is the one we will be using most. The first argument to this function is a string containing a regular expression that defines the search pattern. The reg ex must be surrounded by delimiters - usually forward slashes, /, although the delimiter character can be any character as long as the first and last character of the string are the same. The delimiters show where the pattern begins and ends, but are not part of the pattern. The second argument is the string we are searching for the reg ex pattern. The function returns 1 if there is a match, 0 otherwise. Note that these functions are Perl compatible regular expressions (PCRE) functions. Perl was developed as a language for report (text) processing, and so has a comprehensive set of reg ex functions. The PHP functions at the above link are designed to be functionally equivalent with those Perl functions.

You may also include modifiers with PCRE regular expressions. These modifiers follow the closing delimiter character of the regex. For example, you might do the following:

```
/[a-z]+/i
```

The modifier "`i`" says to "ignore case" when matching. So this regular expression would match 1 or more upper or lowercase alphabetic characters.

The modifier "`m`" says to process multi-line strings - that is, strings which contain newlines (by default, preg_match would stop looking for a match at the first newline). This is often useful for text areas -- as in your assignment! For a full list of modifiers, see Pattern Modifiers.

Validating Input

Write the following programs - save them under different names so you can show them to me.

1. Write a program that takes a postal code from a form and uses a regular expression to determine if it is a valid postal code. When the submit button is pressed, have your program determine if the postal code is valid. A valid postal code has the following format

   X9X9X9

   where X is any upper or lower case alphabetic character, and 9 is a digit from 0 to 9. If the code is not valid, your program should repopulate the field and print an error message next to the input field. If the code is valid, your program should print out the postal code and a message that it is valid, but not reprint the form. Note there are no spaces allowed, or leading or trailing blanks, so this is a fairly restrictive regular expression.
2. Modify the last program so that a valid postal code can **optionally** have a space in the middle X9X 9X9. Note that there is exactly one space in the middle.
3. Modify the last program so that a valid postal code can **optionally** have leading and trailing blanks.
4. Modify the previous program so that it validates a Seneca Subject code. Codes have the following format:

   XXX999X
   XXX999XX

XXX999XXX

where X is an uppercase alphabetic character, and 9 is a digit from 0-9. Leading and trailing blanks are allowed.

5. Modify the previous program to validate a telephone number in the form 999-999-9999 where the dashes are required and 9 is a numeric character. Leading and trailing blanks are allowed.
6. Modify the last program so that the phone number can be in any of these forms (but only these forms!):

```
999-999-9999
999 999 9999
999 999-9999
9999999999
999 9999999
(999) 999-9999
(999) 999 9999
```

Where there is an embedded blank between digits, there may be zero or more embedded blanks, and all valid forms may have leading and trailing blanks. Hint: use alternate forms with the bar: `(form1) | (form2)`

Submitting Your Lab

Run the last program you completed successfully for me - part 6, I hope - before the end of class.