**REGULAR EXPRESSIONS**

Think of regular expressions as an elaborate system of matching patterns. You first write the pattern and then use one of PHPs built-in functions to apply the pattern to a value. Regular expressions are applied to strings, even it that means a string with a numeric value.

**Creating a test script**

The application of the pattern to a value is accomplished using one of a handful of functions. The most important being: preg\_match(). This function returns 0 or 1, indicating whether or not the pattern matches the string:

Preg\_match(pattern, subject);

To see if the word cat has the letter a, you would code:

If(preg\_match(‘/a/’, ‘cat’)) { }

The forward slashes (/) are called *delimiters* and must be user to mark the beginning & end of a pattern. It doesn’t have to be a forward slash, it can be any non-alphanumeric character, besides a backslash (\). So if you need to match a forward slash then use a different delimiter e.g. ‘|’ or ‘!’.

Example **1.1**

If($\_SERVER[‘REQUEST\_METHOD] == “POST”) {

//trim the strings from the input boxes

$pattern = trim($\_POST[‘pattern’]);

$subject = trim($\_POST[‘subject’]);

Echo “The result of checking $pattern against $subject is: “;

//test

If(preg\_match($pattern, $subject)) {

Echo “TRUE”;

} else {

Echo “FALSE”;

}

}

<form action = “index.php” method = “post”>

Pattern: <input type=”text” name=”pattern” value=” <?php if(isset($pattern)) echo htmlentities($pattern); ?> ”/>

subject: <input type=”text” name=”subject” value=” <?php if(isset($subject)) echo htmlentities($subject); ?> ”/>

<input type=”submit” name = “submit” value=”test” />

</form>

Because the two values might include quotation marks and other characters that would conflict with the forms ‘stickiness’, each variables value is sent through htmlentities().

**Defining simple patterns**

The first type of character you will use for defining patterns is a literal. A literal is a value that is written exactly as it is interpreted. E.g. *rom* will match ‘CD-ROM’, ‘Rommel crossed the dessert’ and ‘I’m writing a roman a clef’.

Along with literals, your patterns will use meta-characters. These are special symbols that have a meaning beyond their literal value.

|  |  |
| --- | --- |
| **Character** | **Meaning** |
| \ | Escape character |
| ^ | Indicates the beginning of a string |
| $ | Indicates the end of a string |
| . | Any single character except newline |
| | | Alternatives (or) |
| [ | Start of a class |
| ] | End of a class |
| ( | Start of a sub pattern |
| ) | End of a sub pattern |
| { | Start of a quantifier |
| } | End of a quantifier |

E.g. the period (.) will match any single character except for a new line (e.g. a, b, c, the underscore, a space etc... just not \n).

To match any meta-character, you will need to escape it (e.g. \.). if not escaped, 1.99 will match: 1.99, 1B99, 1299 etc… but 1\.99 only matches 1.99.

Two meta-characters specify where certain characters must be found (^) marks the beginning, ($) marks the end. E.g. ^a – match any string beginning with an *a*. $a – match any string ending with an *a*.

You can use parenthesis to group characters into more involved patterns, (abc) will match abc.

Example (using example 1.1)

Check if a string contains the word *color* or *colour*: /col(o|ou)r/

Check if a string starts with *cat*: /^cat/

**Using quantifiers**

|  |  |
| --- | --- |
| **Character** | **Meaning** |
| ? | 0 or 1 |
| \* | 0 or more |
| + | 1 or more |
| {x} | Exactly x occurences |
| {x,y} | Between x and y (inclusive) |
| {x,} | At least x occurences |

Example (using example 1.1)

Check if a string contains the letters *c* and *t*, with one or more letters inbetween: /c.+t/

Check if a string matches either *cat* or *cats*: /^cats?$/

Check if a string ends with .33, .333, or .3333: /\.3(2,4)$/

Match a five digit number: /^(0|1|2|3|4|5|6|7|8|9){5}$/

**Using character classes**

|  |  |  |
| --- | --- | --- |
| **Class** | **Shortcut** | **meaning** |
| [0-9] | \d | Any digit |
| [\f\r\t\n\v] | \s | Any white space |
| [A-Za-z0-9\_] | \w | Any word character |
| [^0-9] | \D | Not a digit |
| [^\f\r\t\n\v] | \S | Not a white space |
| [^A-Za-z0-9\_] | \W | Not a word character |

Character classes are used to make *or* statements and literals a bit easies:

e.g. [a,e,i,o,u] is equivalent to (a|e|i|o|u)

[a-z] is equivalent to alphabet lowercase

Within classes, most of the meta-characters are treated literally, except for:

\ - still the escape

^ - negation operator

- - indicates a range

] – closing bracket (obvz!)

A class can have both ranges and literals.

Example (using example 1.1)

Check if a string is formatted as a value US zip code (a US zip code always starts with 5 digits but a valid zip code could also have a dash followed by another 4 digits):

/^(\d{5})(-\d{4})?$/

Check if a string contains no spaces: /^\S$/

Validate an email address: /^[\w.-]+@[\w.-]+\.[A-Za-z]{2,6}$/

**Finding all matches**

Preg\_match() has been used to see if a pattern matches a value or not. To find out what actually matches you can use a third argument:

preg\_match(pattern, subject, $match);

the $match variable will contain the first match found to find every match, use:

preg\_match\_all(pattern, subject, $matches);

This function will return the number of matches made (or FALSE if none). It will also assign to $matches evert match made.

Example **1.2** (using 1.1)

\**use 1.1 but save as different file*\*

\*make subject box a text area\*

Change preg\_match(), to:

If(preg\_match\_all($pattern, $subject, $matches) ) {

Echo ‘<pre>’ .print\_r($matches,1).’</pre>’;

*Now test the script*

**Using modifiers**

One final type of special character is the pattern modifier:

|  |  |
| --- | --- |
| **Character** | **Result** |
| A | Anchors the pattern to the beginning of the string |
| i | #enables case-insensitive mode |
| m | Enables multiline matching |
| s | Has the period match every character, including newline |
| x | Ignores most white space |
| U | Performs a non-greedy match |

Pattern modifiers are different than other meta-characters in that they are placed after the closing delimiters.

Example (using 1.2)

Validate a list of email addresses:

/^[\w.-]+@[\w.-]+\.[A-Za-z]{2,6}\r?$/m

**Matching and replacing patterns**

To search & replace: preg\_replace(pattern, replacement, subject);

This function takes an optional 4th argument limiting the number of replacements made.

To replace all instances of *cat* with dog, you would use:

$str = preg\_replace(‘/cat/’, ‘dog’, ‘I like my cat’);

Example (using 1.2)

\*add a replacement input field and store in $replace variable (and trim)\*

Change the code which checks for a match to:

If(preg\_match($pattern, $subject) ) {

Echo preg\_replace($pattern, $replace, $subject);

}else{

Echo “The pattern was not found”;

}

**Test**

**Part 1**

1. What is the main function used with regular expressions?
2. Give a brief example using the above method using a basic pattern and string
3. Create a test script \*\*
4. Look at the *defining simple patterns* table and revise/practice
5. Check if a string starts with a certain word/letter(s)
6. Check if a word contains *color* or *colour*
7. Look at the *quantifiers* table and revise/practice

**Part 2**

1. Look at the *character classes* table and revise/practice
2. What 4 meta-characters are not treated literally with classes
3. Validate an email address
4. What function is used to find match/es of patterns
5. Display all matches
6. Look at the *modifiers* table and revise/practice
7. Validate a list of email addresses
8. What method is used to match and replace? Give example