Programming via PHP Input validation

12.1. The `preg_match` function

Users submitting information to a Web site don't always behave quite as you would like. Usually, they make an idle mistake, such as misformatting what they send (perhaps neglecting to include the area code in a telephone number). But there are also inevitably some who try to abuse popular Web sites. For both sets of users, it is important for a PHP script to verify that what they send is correct. This process is called **input validation**.

One of the most useful tools for input validation is PHP's `preg_match` function. It isn't an easy function to learn, but it can be a powerful way to check whether something the user types fits into the format your Web site requires. Below is an example PHP fragment for confirming that a ZIP code submitted by the user is indeed a 5-digit number. (This would not work for people from countries other than the U.S.)

```
if(preg_match("/^[0-9]{5}$/", $form_zipcode)) {
    echo "The ZIP code must be a 5-digit number.";
}
```

The `preg_match` function takes two parameters. The first, called the *pattern*, describes the potential inputs. It is a string starting and ending with a slash ('`/`'), surrounding a *regular expression*. Right now, the particular regular expression appearing above (`^[0-9]{5}$`) will look nonsensical, but we'll study regular expressions later in this chapter. The second parameter is the string against which `preg_match` should compare the pattern. The function returns *true* or *false* depending on whether they match.

12.2. Regular expressions

A **regular expression** is a common way of describing a large set of possible strings. It can be seen as a miniature language, although it is expressed in a condensed form.

12.2.1. Matching subsequences

The simplest form of regular expression is as a sequence of letters or digits; in this case, `preg_match` will return 1 if that sequence appears anywhere in the string to be matched. For example, the pattern `amb` will match *amble* or *Cambridge*, but not *Amber* (since it is case-sensitive) or *aplomb* (since the letters *amb* must be adjacent).

But there are a number of special symbols one can use. One is the set of brackets '`[`' and '`]`', which indicates a choice among several characters. If we want to allow the*a* to be capitalized in the above example, then we could use `[Aa]mb` as our pattern. You can also indicate a range of characters inside the brackets using a hyphen; thus`a[a-z]b` will match any string containing an *a* followed by a lower-case letter followed by a *b*, as in *arbor* and *rhubarb*.

It is also sometimes useful to allow any character to appear in a particular spot in a regular expression. A period '`.`' will match any single character.

12.2.2. Matching the full string

Most often with user input, you'll want to match the entire string, not search for a particular substring. To say that the pattern must match starting at the string's beginning, you start the pattern with a caret ('`^`'). And to say that the pattern must match ending at the string's end, you terminate the pattern with a dollar sign ('`$`'). Most often, you would use both of these. For example, to confirm that a state code submitted by the user contains two capital letters, you could use the pattern `^[A-Z][A-Z]$`.

12.2.3. Repetitions

You can use a set of braces ('`{`' and '`}`') to specify that something appear a particular number of times within a regular expression. You would place the braces directly after the pattern that should be repeated that number of times, and the number would appear within the braces. Another way to write our two-letter state code pattern is as `^[A-Z]{2}$`. We also saw earlier that we can match 5-digit ZIP codes using the pattern `^[0-9]{5}$`.

A set of braces can also contain two numbers inside, specifying that the number of repetitions of the preceding pattern must be between those two numbers. The set of numbers between 1000 and 99999, for example, may be described with `^[1-9][0-9]{2,4}$`.

You can use a set of parentheses to specify an order of operations. This can be useful when you want a pattern of potentially several characters to repeat multiple times. We can extend our ZIP code pattern to allow either a 5-digit or 9-digit ZIP code using `^[0-9]{5}(-[0-9]{4}){0,1}$`. Here, we've used parentheses to enclose the final dash and four digits, saying that this portion is allowed to occur either 0 or 1 times.

There is a shorthand for saying 0 or 1 times: You can use a question mark ('`?`') in place of `{0,1}`. Similarly, you can use an asterisk ('`*`') to indicate that the preceding pattern can repeat any number of times (including the possibility that it may not appear at all); and you can use a plus sign ('`+`') to indicate that the preceding pattern must appear at least once. If I wanted to describe the set of strings describing integers, I could use the regular expression `^-?[0-9]+$`: It says that the string may begin with a negative sign, but that it must contain at least one digit.

12.2.4. Summary

Finally, you'll occasionally want a regular expression to mention one of the characters that have a special meaning within a regular expression. You can remove its special meaning to `preg_match` by preceding it with a backslash. Unfortunately, backslashes within strings also have a special meaning to PHP, so you should really precede it by two backslashes, so that `preg_match` will receive one of them. For instance, a regular expression one could use for describing e-mail addresses is `^[a-z]+@[a-z]+(\.[a-z]+)+$`. This says that there must be one or more letters before the '`@`' sign, followed by one or more letters, followed by one or more instances of a period followed by one or more letters. The period must be preceded by a backslash, because otherwise `preg_match` will allow any character in place of the period. (This regular expression is deficient because some institutions use periods in e-mail addresses, and some domain names contain hyphens.)

The above description isn't a complete description of the options available with regular expressions, but it is adequate for most purposes. Below is a summary of all of the special characters we have seen.

`()`	grouping
`[]`	range of characters
`.`	any character
`{}`	copies of the preceding pattern
`?`	zero or one of the preceding pattern
`*`	any number of the preceding pattern (including zero)
`+`	at least one of the preceding pattern
`^`	start of string
`$`	end of string
`\`	treat next character literally instead of as a special symbol

Source: http://www.toves.org/books/php/ch12-regex/index.html