

## **Programming in PHP**

**X52.9224 Spring 2005**

Instructor : David Mintz <[dmintz@davidmintz.org](mailto:dmintz@davidmintz.org)>

[http://davidmintz.org/php\\_course/](http://davidmintz.org/php_course/)

# **Session 3**

## **Outline**

Form input validation, string manipulation and regular expressions;  
(time permitting) code re-use with `require()`, and `include()`; setting the `include_path`;

Note: Sections of this document are stolen right out of the official PHP manual:

<http://php.net/manual/en/>

## **Input Validation**

Validating user input is one of the most essential tasks of any PHP application that uses user-supplied data. It can also be one of the most tedious – though there are some libraries that vastly reduce the pain. Tedious or not, the alternative is sloppy and insecure code that is buggy, vulnerable to attack by malicious users, and that ultimately produces a less satisfying experience for site users (and site owners) than would be the case if the developer had taken the care to sanitize external data.

A central precept is: *never trust user input*. It is presumed guilty – unacceptable – until it proves otherwise. Whether it's the result of malice or innocent mistake, assume that every piece of user input is invalid and check it.

How do you do that? By deciding up front what kind of data you expect and will accept.

If the user is supposed to choose from a <SELECT> menu whose options are *a*, *b* and *c*, then your program must reject anything that isn't *a*, *b* or *c*. (Just because you have presented the user with a <SELECT> does not guarantee that somebody is not submitting to your script with a home-grown form or even a program that isn't a web browser at all.) In this case, your friends are PHP functions like PHP function `in_array()`.

If a field value is supposed to be a 10-digit phone number, your program must ensure that it contains ten digits.

If a field value is supposed to look like a U.S. zip code, it should consist of five digits – or perhaps, five digits followed by a hyphen followed by four more digits.

If it supposed to be a last name, it should probably not exceed some reasonable length – it certainly should not be a string the length of *War and Peace*, so you need to enforce a reasonable limit.

If the input is supposed to be an email address, it should meet certain criteria that mean it looks like an email address.

If the input is supposed to be plain old words rather than HTML or Javascript, then your program needs to ensure that anything that looks suspicious is filtered out.

This may make for some tedious work, but fortunately it is not inordinately difficult, thanks to PHP's rich repertoire of string manipulation functions and regular expression support.

## XSS Attacks

This is but one relatively simple example of why sanitizing incoming data is essential. For further reading: <http://shiflett.org/articles/foiling-cross-site-attacks>:

```

```

Suppose you are the administrator of a sloppily coded BBS application, and you are logged in with full admin privileges. Assume your site uses cookies to pass the session id back and forth (as indeed many applications do) to identify you as a valid, logged in user. One of these cookie name/value pairs might look something like `PHPSESSID=0acec5d7f09a122443d5a3a0c02dfee9`. Someone from evil.com registers for your site and gets away with entering the above in the user profile. You view this user's profile and you happen to have Javascript enabled. You just gave your cookies for this domain to evil.com, perhaps emailing them to the attacker. Now an attacker has only to go to `yoursite.com?PHPSESSID=0acec5d7f09a122443d5a3a0c02dfee9` to hijack your session and take the keys to the kingdom.

## String manipulation

Here are a few of the dozens of string manipulation functions. Generally it is less computationally expensive to use these than a full-blown regular expression, they are preferred when a regex is not necessary. You can use these to enforce your data validation rules, and to examine, transform, filter and format your input and output.

int **strlen** ( string *string* )

Returns the length of the given *string*.

<?

```
php echo strlen('abcdef');
```

```
// or....
```

```
if (strlen($_POST('comments')) > 800) {  
    $errors[] = "Your comments exceed the maximum length of 800 characters"  
}  
?>
```

string **substr** ( string *string*, int *start* [, int *length*])

Returns the portion of string specified by the *start* and *length* parameters. The first character is number 0.

```
<?php  
// simply discard input in excess of 800 characteristics  
    $comments = substr($_POST['comments'],0,800);  
?>
```

string **trim** ( string *str* [, string *charlist*])

trims whitespace from beginning and end of a string – things like tab, space, newline. The optional second parameter is a list of characters to strip – usually the default is what you'll want.

**ltrim()** and **trim()**

same as above, but they operate on the left and right sides of the string, respectively.

string **htmlspecialchars** ( string *string* [, int *quote\_style* [, string *charset*]])

Certain characters have special significance in HTML, and should be represented by HTML entities if they are to preserve their meanings. This function returns a string with some of these conversions made; the translations made are those most useful for everyday web programming. If you require all HTML character entities to be translated, use **htmlentities()** instead.

This function is useful in preventing user-supplied text from containing HTML markup, such as in a message board or guest book application. The optional second argument, *quote\_style*, tells the function what to do with single and double quote characters. The default mode, **ENT\_COMPAT**, is the backwards compatible mode which only translates the double-quote character and leaves the single-quote untranslated. If **ENT\_QUOTES** is set, both single and double quotes are translated and if **ENT\_NOQUOTES** is set neither single nor double quotes are translated.

The translations performed are:

- '&' (ampersand) becomes '&amp;'
- '"' (double quote) becomes '&quot;'; when **ENT\_NOQUOTES** is not set.

- `'` (single quote) becomes `'&#039;` only when `ENT_QUOTES` is set.
- `<` (less than) becomes `&lt;`;
- `>` (greater than) becomes `&gt;`;

string **htmlentities** ( string *string* [, int *quote\_style* [, string *charset*]])

This function is identical to **htmlspecialchars()** in all ways, except with **htmlentities()**, all characters which have HTML character entity equivalents are translated into these entities.

string **strip\_tags** ( string *str* [, string *allowable\_tags*])

This function tries to return a string with all HTML and PHP tags stripped from a given *str*. Handy for cases where you want to allow a subset of HTML tags in your input – but it will not protect you from potentially malicious attribute values (eg `onmouseover`) that users could put in the tags you decide to allow.

```
<?php
$text = '<p>This is a <b>my</b> <em>test</em> paragraph.</p>
';
// take out everything
echo strip_tags($text);
echo "\n";
// allow benign tags
echo strip_tags($text, '<b><em><strong><i>');
?>
```

string **strstr** ( string *haystack*, string *needle*)

Returns part of *haystack* string from the first occurrence of *needle* to the end of *haystack*. If *needle* is not found, returns false.

```
<?php
if (strstr($_SERVER['HTTP_USER_AGENT'], 'Firefox')) {
    echo "nice choice!";
}
?>
```

string **stristr** ( string *haystack*, string *needle*)

Same as **strstr**, only case insensitive.

The following few string functions can be useful for clean up data that you present to the user, rather than vice-versa.

string **strtolower** ( string *string*)

Returns *string* with all alphabetic characters converted to lowercase.

string **strtoupper** ( string *string*)

Returns *string* with all alphabetic characters converted to uppercase.

string **nl2br** ( string string)  
Returns *string* with '<br />' inserted before all newlines.

string **number\_format** ( float number [, int decimals [, string dec\_point, string thousands\_sep]])

**number\_format()** returns a formatted version of *number*. This function accepts either one, two or four parameters (not three):

If only one parameter is given, *number* will be formatted without decimals, but with a comma (",") between every group of thousands.

If two parameters are given, *number* will be formatted with *decimals* decimals with a dot (".") in front, and a comma (",") between every group of thousands.

If all four parameters are given, *number* will be formatted with *decimals* decimals, *dec\_point* instead of a dot (".") before the decimals and *thousands\_sep* instead of a comma (",") between every group of thousands.

Only the first character of *thousands\_sep* is used. For example, if you use *foo* as *thousands\_sep* on the number *1000*, **number\_format()** will return *1foo00*.

```
<?php
$number = 24.78 * 12365.54;
echo 'Unformatted: ' , $number, '<br />';
echo 'US style: ' , number_format($number,2), '<br />';
echo 'Custom format: ' , number_format($number,3,'.','_'), '<br />';
?>
```

string **sprintf** ( string format [, mixed args [, mixed ...]])

Returns a string produced according to the formatting string *format*.

The format string is composed of zero or more directives: ordinary characters (excluding %) that are copied directly to the result, and *conversion specifications*, each of which results in fetching its own parameter. This applies to both **sprintf()** and **printf()**.

Each conversion specification consists of a percent sign (%), followed by one or more of these elements, in order:

1. An optional *sign specifier* that forces a sign (- or +) to be used on a number. By default, only the - sign is used on a number if it's negative. This specifier forces positive numbers to have the + sign attached as well, and was added in PHP 4.3.0.
2. An optional *padding specifier* that says what character will be used for padding the results to the right string size. This may be a space character or a 0 (zero character).

The default is to pad with spaces. An alternate padding character can be specified by prefixing it with a single quote ('). See the examples below.

3. An optional *alignment specifier* that says if the result should be left-justified or right-justified. The default is right-justified; a - character here will make it left-justified.
4. An optional number, a *width specifier* that says how many characters (minimum) this conversion should result in.
5. An optional *precision specifier* that says how many decimal digits should be displayed for floating-point numbers. When using this specifier on a string, it acts as a cutoff point, setting a maximum character limit to the string.
6. A *type specifier* that says what type the argument data should be treated as. Possible types:

*%* - a literal percent character. No argument is required.

*b* - the argument is treated as an integer, and presented as a binary number.

*c* - the argument is treated as an integer, and presented as the character with that ASCII value.

*d* - the argument is treated as an integer, and presented as a (signed) decimal number.

*e* - the argument is treated as scientific notation (e.g. 1.2e+2).

*u* - the argument is treated as an integer, and presented as an unsigned decimal number.

*f* - the argument is treated as a float, and presented as a floating-point number.

*o* - the argument is treated as an integer, and presented as an octal number.

*s* - the argument is treated as and presented as a string.

*x* - the argument is treated as an integer and presented as a hexadecimal number (with lowercase letters).

*X* - the argument is treated as an integer and presented as a hexadecimal number (with uppercase letters).

Eyes glazing over? Not to worry. A good approach is to look at some examples, and move on. You can explore **printf** format strings more deeply as the need arises. Meanwhile, a couple examples:

```
<pre>
<?php
$phones = array("home"=>"201 420-4703", "office"=>"212 805-
0362", "mobile"=>"201 978-0608", );
foreach($phones as $key=>$val) {
    printf("%-10s%s\n", $key, $val);
}
?>
</pre>
```

Output:

```
home      201 420-4703
office    212 805-0362
mobile    201 978-0608
```

The above example uses whitespace to format the data in pretty columns. Handy for formatting something like a plain text email message containing tabular data. The next example computes and formats a hypothetical invoice.

```
<pre>
<?php
$items = array(

    'services rendered'=>18.2*62.75,
    'irritation surcharge'=>600);

// left column: string, 24 spaces wide, left-justified
// right column: float, 7 spaces wide, right-justified
$format = "%-24s%7.2f\n";
$total = 0;

foreach($items as $key=>$val) {
    $total += $val;
    printf($format,$key,$val);
}
printf($format,'TOTAL DUE',$total);

?>
</pre>
```

The next example uses printf number format conversion to turn integer RGB values into hexadecimal for use in HTML.

```
<?php
$r = 255;
$g = 0;
$b = 255;
$hex = sprintf("%02x%02x%02x",$r,$g,$b);
echo "this is what $hex looks like<br />";
?>
<div style="background-color:<?php echo $hex?>">
<p>&nbsp;</p>
</div>
```

mixed **str\_replace** ( mixed *search*, mixed *replace*, mixed *subject*)

This function returns a string or an array with all occurrences of *search* in *subject* replaced with the given *replace* value. If you don't need fancy replacing rules (like regular expressions), you should use this function instead of a regular expression function (more below).

As of PHP 4.0.5, every parameter in `str_replace()` can be an array.

```
<?php
// Result : <body text='black'>
$bodytag = str_replace("%body%", "black", "<body text='%body%'>");

// Result : Hll Wrld f PHP
$vowels = array("a", "e", "i", "o", "u", "A", "E", "I", "O", "U");
$onlyconsonants = str_replace($vowels, "", "Hello World of PHP");

// Result: You should eat pizza, beer, and ice cream every day
$phrase = "You should eat fruits, vegetables, and fiber every day.";
$healthy = array("fruits", "vegetables", "fiber");
$yummy = array("pizza", "beer", "ice cream");

$newphrase = str_replace($healthy, $yummy, $phrase);
```

For further information about PHP string manipulation functions:  
<http://www.php.net/manual/en/ref.strings.php>

## Regular Expressions

Pattern matching with regular expressions is an immensely powerful tool, and a subject deep enough to have entire books devoted to it, such as Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly (ISBN 1-56592-257-3). The following is merely a superficial introduction.

The Perl-Compatible Regular Expression (PCRE) library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5 (with just a few differences). (PHP has another family of functions having names like `ereg_XXX`; they are based on the so-called POSIX-extended regular expression syntax. PCRE is generally faster and more powerful.)

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern *the quick brown fox* matches a portion of a subject string that is identical to itself.

```
<?php
$pattern = "/the quick brown fox/";
$subject = "I hear the quick brown fox jumped over the lazy dog.";
$result = preg_match($pattern,$subject) ? "matches" : "does not
match";
```

```
echo "<p>pattern $pattern $result <em>$subject</em></p>";
```

## Delimiters

You just saw an example of a pattern delimited with / (slashes). This is a convention but you can use just about any character as a pattern delimiter. The only rule is that if you want to have a literal occurrence of your delimiter inside the pattern, you have to escape it with a backslash.

## Meta-characters

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *metacharacters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta-characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the metacharacters are as follows:

\	general escape character with several uses
^	assert start of subject (or line, in multiline mode)
\$	assert end of subject (or line, in multiline mode)
.	match any character except newline (by default)
[	start character class definition
]	end character class definition
/	start of alternative branch
(	start subpattern
)	end subpattern
?	extends the meaning of (, also 0 or 1 quantifier, also quantifier minimizer
*	0 or more quantifier
+	1 or more quantifier
{	start min/max quantifier

} end min/max quantifier

Part of a pattern that is in square brackets is called a *character class*. In a character class the only meta-characters are:

\ general escape character

^ negate the class, but only if the first character

- indicates character range

] terminates the character class

Forget character classes for the moment, and let's go back to the other metacharacters mentioned above, looking at each in turn.

## . (the dot)

The dot `.` matches any character except newline (by default – meaning you can change this behavior with a *pattern modifier* (-:)). Example:

```
<?php
$pattern = "/the qu.ck brown fox/";

$subject = "I hear the quack brown fox jumped over the lazy dog.";

$result = preg_match($pattern,$subject) ?
    "matches" : "does not match";
echo "<p>pattern $pattern $result <em>$subject</em></p>";
```

## ?

In its most common usage as a *quantifier*, the `?` means *zero or one* of the character or subpattern than precedes it. So the following match succeeds:

```
<?php
$pattern = "/the qui?ck brown fox/";

$subject = "I hear the quck brown fox jumped over the lazy dog.";

$result = preg_match($pattern,$subject) ?
    "matches" : "does not match";
echo "<p>pattern $pattern $result <em>$subject</em></p>";
```

Here, the `?` means zero or one occurrence of the letter `i`, so the match succeeds.

+

The + means *one or more* of the character or subpattern that precedes it. The following match would succeed.

```
$pattern = "/the qui+ck brown fox/";  
$subject = "I hear the quiiick brown fox jumped over the lazy dog.";
```

Note that you can combine these metacharacters, as follows, so that this match would succeed:

```
$pattern = "/the .+ brown fox/";  
$subject = "I hear the like whatever dude! brown fox jumped over the  
lazy dog.";
```

Above, \$pattern means "the string the, followed by a space, followed one or more of any character except a newline, followed by a space, followed by brown fox

\*

The asterisk quantifier means *zero or more* of whatever precedes it.

```
$pattern = "/the.* brown fox/";  
$subject = "I hear the like whatever dude! brown fox jumped over the  
lazy dog.";
```

The above pattern match would succeed; it would also succeed against "the brown fox jumped..."

## Quantifying with { } (min/max)

This is easiest explained with examples.

```
/x{2,4}/
```

means *at least two but no more than four* occurrences of x, while

```
/x{2,}/
```

means *at least two* occurrences of x. Note that the second argument is optional but the comma is not. The following are semantically identical:

- \* is the same as {0,}
- + is the same as {1,}
- ? is the same as {0,1}

The following \$pattern matches \$subject:

```
$pattern = "/the qu.{2,8}ck brown fox/";
$subject = "I hear the quaasydyck brown fox jumped over the lazy dog.";
```

## backslash

The backslash character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a "\*" character, you write "\\*" in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphanumeric with "\" to specify that it stands for itself. In particular, if you want to match a backslash, you write "\\".

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner: typically, things like tab (\t) and newline (\n).

NOTE: In the name of brevity and comprehensibility, we skipping some information about the backslash. For the full story please see <http://www.php.net/manual/en/reference.pcre.pattern.syntax.php>

## Character classes

Character classes are for matching single characters. They begin with [ and end with ]. Inside the character class some metacharacters take on special meanings while others lose their special meanings. (Yes, it is a bit confusing. Fear not!)

In the pattern

```
/the qu[aeiou]ck brown fox/
```

the character class stands for any occurrence of a, e, i, o or u. That means the following \$pattern matches \$subject:

```
$pattern = "/the qu[aeiou]+ck brown fox/";
$subject = "the quaaaaeeeeiiiiiiick brown fox jumped over the lazy dog.";
```

If we removed the + from \$pattern, would the match succeed or fail?

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

Thus means the following \$pattern matches \$subject:

```
$pattern = "/the qu[a-z]+ck brown fox/";  
$subject = "I hear the quaaaeiiiiiiiick brown fox jumped over the  
lazy dog.";
```

Suppose \$subject were

"I hear the quaaae eei iiiiiiick brown fox jumped over the lazy dog.";

Would it match this same pattern?

The hyphen can also be used to signify a range of arabic numerals. Therefore /[0-9]/ means *one occurrence of either 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9*

You can combine these like so:

```
/[a-zA-Z0-9]+/
```

to mean *any upper or lower case letter from a to z, or any number.*

## Negation with ^

It gets even more interesting. The ^ (circumflex) means *negation* when it appears at the beginning of a character class. Thus [^aeiou] matches any character that is *not* a lower case vowel.

The metacharacters ^ - \ and ] are the only ones that have special meaning inside a character class. All the rest ( + , \* , ? , etc.) mean their literal selves.

## Shortcut character classes

PCRE provides some shortcuts that aid legibility. The third use of backslash (discussed above) is for specifying generic character types:

\d any decimal digit

\D any character that is not a decimal digit

- `\s` any whitespace character
- `\S` any character that is not a whitespace character
- `\w` any "word" character
- `\W` any "non-word" character

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a Perl "word". The definition of letters and digits is controlled by PCRE's character tables, and may vary if locale-specific matching is taking place.

## Anchors

The anchors metacharacters `^` and `$` constrain the match to the beginning and end, respectively, of the subject.

```
$pattern = '/^[0-9]{5}$/'
```

only matches a subject that *begins* and *ends* with five digits in a row.

## Subpatterns: grouping with () (parentheses)

You can use parentheses for *grouping* or forming *subpatterns*, which is handy when you want to follow a subpattern by a quantifier. Example:

```
$pattern = '/^[0-9]{5}(-[0-9]{4})?$/'
```

would match a string that looked like either a 5-digit or a 9-digit zip code. That is, five digits, *followed by zero or one occurrences of hyphen-digit-digit-digit-digit*. (If you think it's a little painful to read patterns like this, I think you'd be right. There's a *modifier* that lets you use whitespace and comments, and turn a regular expression into an essay if you're so inclined.)

However, using the shortcuts mentioned above, you can rewrite this pattern as

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

which you might (*might*) find more legible.

## Alternation with | (vertical bar)

The vertical bar like saying 'or' and can be combined with the grouping parens to powerful effect. You use vertical bar characters to separate alternative patterns. For example, the pattern *gilbert/*

*sullivan* matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern, "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

```
$pattern = '/the (quick|agile|nimble) brown fox/';
```

You can probably see where this is going. The above matches any of the following:

```
the quick brown fox
the agile brown fox
the nimble brown fox
```

## Modifiers

You can tack certain flags onto the end of a Perl-compatible regex to change its behavior. These are known as modifiers.

### i

Perhaps the one you will use most often is *i*, meaning caseless (mnemonic: *i* as in case insensitive). You put the modifier after the ending delimiter, like so

```
<?php
$pattern = '/the (quick|agile|nimble) brown fox/i';

// Now you could use ransom-note-case in $subject, and it will match

$subject = "tHE qUiCk bRoWn fOX";

$result = preg_match($pattern,$subject) ?
    "matches" : "does not match";
echo "pattern: $pattern<br />subject: $subject<br />result: <b>$result</b>";
?>
```

Your assignment will make use of a regular expression to match a syntactically-valid-appearing email address, and it will use this modifier.

### S

The *s* modifier means make the dot in the regex it modifies match any character *including* newline (without the *s* modifier, dot excludes newline).

For more information about modifiers:

<http://www.php.net/manual/en/reference.pcre.pattern.modifiers.php>

## Capturing subpatterns

Another meaning of the parentheses is that it tells PHP to *remember* whatever string matched that subpattern. Each successive parenthesized match becomes available in the special variables \$0, \$1, \$2 and so on up to \$99. These are sometimes known as *backreferences*, and come in handy when you want to re-arrange the pieces that you capture. This will make more sense in the context of the following section.

## The preg\_XXX Functions

Please see <http://www.php.net/manual/en/ref.pcre.php> for full documentation of all the built-in PCRE functions. We will only look at a couple of them in this discussion

mixed **preg\_replace** ( mixed *pattern*, mixed *replacement*, mixed *subject* [, int *limit*])

Searches *subject* for matches to *pattern* and replaces them with *replacement*. If *limit* is specified, then only *limit* matches will be replaced; if *limit* is omitted or is -1, then all matches are replaced.

*Replacement* may contain references of the form \$n. Every such reference will be replaced by the text captured by the n'th parenthesized pattern. n can be from 0 to 99, and \$0 refers to the text matched by the whole pattern. Opening parentheses are counted from left to right (starting from 1) to obtain the number of the capturing subpattern.

```
<?php
$pattern = '/reptiles|dogs|fish/';
$replacement = "cats";
$subject = "John likes dogs because he thinks dogs are pleasant.";
$result = preg_replace($pattern, $replacement, $subject);
echo "<p>$result</p>";
?>
```

Here's another more practical example. Suppose you are expecting a phone number and you want to strip out all the digits so you can call **strlen()** on the resulting string to count how many digits there are (which may be helpful for validating user input). Perhaps you are also planning to save the phone number in a database in a compact, format-agnostic way.

```
<?php
$subject = "(212) 805-0362";
$just_digits = preg_replace("/\D/", "", $subject);
echo "<p>The number $just_digits has ". strlen($just_digits) ." digits</p>";
?>
```

An example with back-references: suppose you need to change the date format in some text, without knowing the exact date in question.

```
<?php
```

```

$subject = "my date format is March 17, 2004, hope that's ok";
$pattern = "/\b(jan|feb|mar)\S* (\d{1,2}), (\d{4})/i";
$replacement = "$2-$1-$3";
$result = preg_replace($pattern, $replacement, $subject);
echo "<p>$result</p>";
?>

```

Another noteworthy feature of `preg_replace()` is that *pattern* and *replacement* can be arrays, in which case `$patterns[0]` will replace `$replacements[0]`, `$patterns[1]` will replace `$replacements[1]`, and so forth.

mixed **preg\_match** ( string *pattern*, string *subject* [, array *&matches* [, int *flags* [, int *offset*]])

Searches *subject* for a match to the regular expression given in *pattern*.

If *matches* is provided, then it is filled with the results of the search. `$matches[0]` will contain the text that matched the full pattern, `$matches[1]` will have the text that matched the first captured parenthesized subpattern, and so on.

If you had an entire HTML document in a string `$html`, this is a way to parse out the document body:

```

<?php
preg_match("/<body[^>]*>(.)</body>/si", $html, $matches);
// now $matches[1] contains everything between
// the opening and closing body tags.
?>

```

int **preg\_match\_all** ( string *pattern*, string *subject*, array *&matches* [, int *flags* [, int *offset*]])

Searches *subject* for all matches to the regular expression given in *pattern* and puts them in *matches* in the order specified by *flags*.

So this would be another way to figure out how many digits are in a string:

```

<?php
$string = "(212) 805-3062";
$count = preg_match_all("/\d/", $string, $matches);
echo "<p>There are $count digits in $string</p>";
?>

```

## Assignment

Download [http://davidmintz.org/php\\_course/3/week3.assignment.txt](http://davidmintz.org/php_course/3/week3.assignment.txt) and save it as `week3.php`

The first section contains some function declarations. One of them is called

`validate_form($data)` and its body is almost empty of code, but there is a comment block describing the validation rules. *Replace the comment block with the code that enforces the validation rules.* Use conditionals to test the input against the validation criteria. Append an appropriate error message to the `$errors` array for each test that fails. Tip: your friends are likely to include `array_key_exists()`, `preg_match()`, `strlen()`, `preg_replace()`, `trim()`

*Change only the `validate_form()` function body. Do not change anything else in the script.*

To see how your solution should behave, try  
[http://davidmintz.org/php\\_course/3/week3.solution.php](http://davidmintz.org/php_course/3/week3.solution.php)

## Reading

Sklar (*Learning PHP 5*): Appendix B

Lerdorf (*Programming PHP*): Chapter 4

Atkinson (*Core PHP Programming*) : Chapter 12.8

The PHP manual at <http://php.net/manual/en> : string functions, regex functions

Any of last week' s reading that you didn' t do ( - :