Lecture 19b: Regular expressions
Regular expressions are used for searching patterns of characters through text.

For example you might want to extract all email addresses from a file.

Regular expressions pop up in many places, not just PHP so they are well worth it to learn.

Applications of regular expressions include data validation, data manipulation.

You can match phone numbers, email addresses, url's, credit card numbers, social security numbers, zip codes, states, cities.....(the list never ends).

A huge script that is supposed to validate a user input and prepare it for a database can be reduced to only one line with the help of preg_replace.
Regular expressions are all about specifying a pattern.

You may have done a search on your computer for all *doc* files by searching for *.doc*. Regular expression patterns give you way more flexibility, but the idea is the same.

Example:

```
[A-Z0-9\._]+@[A-Z0-9\._]+\.[A-Z]{2,4}
```

Matches an email address.
Building blocks of patterns

The simplest match is a single character:

Example:

Pattern: a

Input string: This is a fine day.

Match will be found in the word a as well as in word day.

Single character is any character except: [\^$.|?*+()]

These are special characters that we will talk about shortly.

Any special character can be written as a regular match character if you escape it with a \
Simple patterns

Pattern: apple
Result: Matches any occurrence of apple. Including Snapple.

Pattern: ^apple
Result: Matches only words that begin with apple.

Pattern: apple$
Result: Matches only words that end with apple.
Character classes

Examples:

Pattern: [JTV]
Result: Matches a single character which is either J or T or V.

Pattern: [a-z]
Result: Matches any character a through z.

Pattern: [^ucla]
Result: Match any character that is not u,c,l,a.
## Predefined Character Classes

<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Literal value</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>[0-9]</td>
<td>&quot;0&quot;,&quot;1&quot;,&quot;2&quot;,...</td>
</tr>
<tr>
<td>\D</td>
<td>[^0-9]</td>
<td>Non-digit</td>
</tr>
<tr>
<td>\w</td>
<td>[A-Za-z_0-9]</td>
<td>Word character</td>
</tr>
<tr>
<td>\W</td>
<td>[^A-Za-z_0-9]</td>
<td>Not a word character</td>
</tr>
<tr>
<td>\s</td>
<td>[\r\t\n\f]</td>
<td>Whitespace</td>
</tr>
<tr>
<td>\S</td>
<td>[^\r\t\n\f]</td>
<td>Not a whitespace</td>
</tr>
</tbody>
</table>
Count indicators

Pattern: [1-5]*
Match: A number 1-5 may occur 0 or more times.

Pattern: [H-S]+
Match: A letter H-S must occur at least once.

Pattern: [v]?
Match: The letter v may occur once or not at all.

Pattern: [ucla]{2,3}
Match: Any of u,c,l,a may occur twice or three times.
More regular expression examples

US zip code:
[0-9]{5}

UCLA Bruin email:
[a-z][a-z0-9]{2,14}@ucla\..edu

Telephone number in (XXX)-XXX-XXXX form:
(\d\d\d)-\d\d\d-\d\d\d\d
"ab*": matches a string that has an a followed by zero or more b's ("a", "ab", "abbb", etc.);
"ab+": same, but there's at least one b ("ab", "abbb", etc.);
"ab?": there might be a b or not;
"a?b+$": a possible a followed by one or more b's ending a string.

You can also use bounds, which come inside braces and indicate ranges in the number of occurrences:

"ab{2}": matches a string that has an a followed by exactly two b's ("abb");
"ab{2,}": there are at least two b's ("abb", "abbb", etc.);
"ab{3,5}": from three to five b's ("abbb", "abbbb", or "abbbbb").

Note that you must always specify the first number of a range (i.e, "{0,2}", not "{,2}"). Also, as you might have noticed, the symbols '*', '+', and '?' have the same effect as using the bounds "{0,}", "{1,}", and "{0,1}", respectively.
More regular expression examples

"a(bc)*": matches a string that has an a followed by zero or more copies of the sequence "bc";
"a(bc){1,5}": one through five copies of "bc."

There's also the '|' symbol, which works as an OR operator:

"hi|hello": matches a string that has either "hi" or "hello" in it;
"(b|cd)ef": a string that has either "bef" or "cdef";
"(a|b)*c": a string that has a sequence of alternating a's and b's ending in a c;
A period ('.') stands for any single character:

"a.[0-9]": matches a string that has an a followed by one character and a digit;
"^.{3}$": a string with exactly 3 characters.

"[ab]": matches a string that has either an a or a b (that's the same as "a|b");
"[a-d]": a string that has lowercase letters 'a' through 'd' (that's equal to "a|b|c|d" and even "[abcd]");
"^[a-zA-Z]": a string that starts with a letter;
"[0-9]%": a string that has a single digit before a percent sign;
",[a-zA-Z0-9]$": a string that ends in a comma followed by an alphanumerical character.

You can also list which characters you DON'T want -- just use a '^' as the first symbol in a bracket expression
(i.e., "%^[a-zA-Z]%" matches a string with a character that is not a letter between two percent signs).
Recall that our pattern is a string composed of a regular expression.

It looks something like:

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

Any guess what the above matches?

Next we want to know how to search for this pattern in a given text.

In PHP we have to add slashes around this expression.

```
$pattern = "/^\d{5}(-\d{4})?$";
```
PHP and regular expressions

`preg_match($pattern, $string )`

- Returns 1 if it finds a $pattern match in string, 0 otherwise.
- Stops looking in $string after the first match of $pattern.
- Returns FALSE if an error occurred.
PHP and regular expressions

Example:

$url = "http://www.math.ucla.edu/~virtanen";
if (preg_match('/^(http|https|ftp)://[A-Z0-9][A-Z0-9_\-]*(?:.[A-Z0-9][A-Z0-9_\-]*)+):?(d+)?/i', $url))
{
    echo "Your url is ok.";
}
else
{
    echo "Invalid url.";
}
PHP and regular expressions

preg_match($pattern, $string, $matches )

Can also be used to return the first occurrence of the matched pattern.

$matches is an array where 0 index contains the first match.

```php
<?php
$s="The zip code is: 976435";

preg_match('/^\d{5}(-\d{4})?$/', $s, $matches);

$zip = $matches[0];

?>
```
PHP and regular expressions

Sub-patterns

Inside a pattern we can specify a sub-pattern by using parenthesis. The sub-patterns can themselves be used to find a specific text inside a matched pattern.

`preg_match($pattern, $string, $matches)`

$matches is an array where 0 index contains the first match and subsequent indexes contain sub-matches.

```php
<?php
$url="http://www.pic.ucla.edu/~virtanen";

preg_match('/http:\/[\/[\^\/]++/\', $url, $matches);

$domain = $matches[1];

print "Domain is: $domain";
?>

Output: Domain is: www.pic.ucla.edu
PHP and regular expressions

preg_match_all($pattern, $string, $matches )

•Searches for all matches to $pattern in $string and stores the substrings which matches in array $matches.

Note: $matches[0] IS an array!! This is bit confusing but all the strings we want are contained in $matches[0]. Read the PHP manual for clarification.
<?php

$str = 

$Phone_Pattern = "/(\d)?(\s|-)?(\()?(\d){3}(\))?(\s|-){1}(\d){3}
(\s|-){1}(\d){4}/";

preg_match_all($Phone_Pattern,$str,$phone);

for($i=0; $i < count($phone[0]); $i++)
{
    echo $phone[0][$i]."<br>";
}

?>
PHP and regular expressions

preg_grep($pattern, $arraytosearch)

- Returns the array entries that match $pattern. Values are indexed using the keys of $arraytosearch.
PHP and regular expressions

preg_replace($pattern, $replacement, $string)

• Replaces all substrings of $string which match $pattern by $replacement.
• Any of the parameters above can either be a string or an array of strings.
• Returns an array or string with replacements depending on whether $string is a string or an array of strings.
• If $pattern is an array and $replacement is a string, then PHP replaces any substring that matches any pattern with the same $replacement.
• If they both are arrays, PHP replaces substrings of $string that match pattern element at index i with replacement element at index i.
Example

This is a bit beyond what we have time for, but I include the following example for completeness. Feel free to read up on this on the web.

```php
<?php
echo preg_replace("/\([Cc]\)opyright 20(09|10|11)/", "$1 2012", "Copyright 2009");
?>
```

In the above example we use back references in the replacement string. Back references make it possible for you to use part of a matched pattern in the replacement string. To use this feature, you should use parentheses to wrap any elements of your regular expression that you might want to use. You can refer to the text matched by subpattern with a dollar sign ($) and the number of the subpattern. For instance, if you are using subpatterns, $0 is set to the whole match, then $1, $2, and so on are set to the individual matches for each subpattern.
Patterns remain the same. There are different functions for matching patterns.

3 string methods
- \texttt{search(/bruin/)\texttt{}}
  - returns the index of the first substring that matches the pattern /bruin/ or -1 if no match.
- \texttt{match(/bruin/)\texttt{}}
  - returns the first substring that matches the pattern /bruin/ or undefined if no match
- \texttt{replace(/trojan/, "bruin")\texttt{}}
  - replaces the first substring that matches the pattern /trojan/ with the string "bruin"

Example: "Hello".search(/He/); //returns 0 since He is found at index 0.
Submitting forms via JS

<form action="process_form.php" onsubmit="return validateForm()"
method="post">
First name: <input id="fname" type="text" name="fname">
<input type="submit" value="Submit">
</form>

function validateForm()
{
value = document.getElementById("fname").value;

if (value.search(/[a-zA-Z]{2,100}/)==-1){
    alert("Nope!!");
    return false;
}
else {alert("ok!");
    return true;
}