Introduction to Pattern Matching 1

Pattern Matching In PHP

- Many programming problems require matching or manipulating patterns in string variables.
- For example, if you are expecting an XHTML form field to provide a U.S. telephone number as input, your script needs a way to verify that the input comprises a string of seven or ten digits.
- Another reason to match patterns arises when your script uses an input data file with fields that are delimited by characters such as colons or tabs.
- The process of searching text to identify matches—strings that **match** a regex's **pattern**—is **pattern matching.**
- Pattern matching in PHP is handled through regular expressions.

Pattern Matching In PHP

- Often in PHP we have to get data from files, or maybe through forms from a user.
- Before acting on the data, we:
  1. Need to put it in the format we require.
  2. Check that the data is actually valid.

Pattern Matching In PHP

- To achieve this, we need to learn about PHP functions that check values, and manipulate data.
  - Input PHP functions.
  - Regular Expressions (Regex).
- There are a lot of useful PHP functions to manipulate data.

PHP Functions

- Splitting
  - Often we need to split data into multiple pieces based on a particular character.
  - Use explode().

```
// expand user supplied date..
$input = '1/12/2007';
$bits  = explode('/',$input);
// array(0=>1,1=>12,2=>2007)
```

PHP Functions

- Trimming
  - Removing excess whitespace..
  - Use trim()

```
// a user supplied name..
$input = '   Rob    ';
$name  = trim($input);
// 'Rob'
```

PHP Functions

- String replace
  - To replace all occurrences of a string in another string use str_replace()

```
//allow user to use a number of date separators
$input  = '01.12-2007';
$clean = str_replace(array('.','-'),'/',$input);
    // 01/12/2007
```

PHP Functions

- cAsE
  - To make a string all uppercase use strtoupper().
  - To make a string all uppercase use strtolower().
  - To make just the first letter upper case use ucfirst().
  - To make the first letter of each word in a string uppercase use ucwords().

Pattern Matching In PHP

- Regular expressions (regex) are one of the black arts of practical mode Regular expressions (regex) in programming.
- Fundamentally, they are a way to describe *patterns* of text using a single set of strings.
- Unlike a simple search-and-replace operations, such as changing all instances of `"Marty"` to `"Mark"`, regex allow for much more flexibility
  - for example, finding all occurrences of the letters `"Mar"` followed by either `"ty"` or `"k"`, and so on.

Pattern Matching In PHP

- PHP actually supports both the POSIX standard and the Perl standard of regular expressions.
- The Perl version is known as PCRE (Perl-Compatible Regular Expressions).
- PCRE are much more powerful than their POSIX counterparts – and consequently more complex and difficult to use.

Pattern Matching In PHP

- First, the simplest regular expression is a single character.
- For example, the regex `a` would match the character "a" in the word "Mark".
- Next, single character regex can be grouped by placing them next to each other.
- Thus the regex `Mark` would match the word "Mark" in "Your instructor is Mark for CIS 4004."
- So far, regex are not very different from normal search operations.
  - However, this is where their similarities end.

Pattern Matching In PHP

- The Kleene Star can be used to create regex that can be repeated any number of times (including none).
- Consider the following string:

```
seeking the treasures of the sea
```

- The regex `se*` will be interpreted as "the letter s followed by zero or more instances of the letter e" and will match the following:
  - The letters "see" of the work "seeking", where the regex `e` is repeated twice.
  - Both instances of the letter s in "treasures", where `s` is followed by zero instances of `e`.
  - The letters "se" of the work "sea", where the `e` is present once.

Pattern Matching In PHP

- It's important to understand in the regex `se*` that only the expression e is considered with dealing with the star.
- Although its possible to use parentheses to group regular expressions, you should not be tempted to think that using `(se)*` is a good idea, because the regex compiler will interpret it as meaning "zero or more occurrences of se".
- If you apply this regex to the same string, you will encounter a total of 32 matches, because every character in the string would match the expression.
- (Remember? 0 or more occurrences!)

Pattern Matching In PHP

- You'll find parentheses are often used in conjunction with the pipe operator to specify alternative regex specifications.
- For example, the regex `gr(u|a)b` with the string: "grab the grub and pull" would match both "grub" and "grab".
- Although regular expressions are quite powerful because of the original rules, inherent limitations make their use impractical.
- For example, there is no regular expression that can be used to specify the concept of "any character".
- As a result of the inherent limitations, the practical implementations of regex have grown to include a number of other rules, most common of which are shown on the beginning of the next page.

Additional Syntax For Regex

- The special character "^" is used to identify the beginning of the string.
- The special character "$" is used to identify the end of the string.
- The special character "." is used to identify any character.
- Any nonnumeric character following the character "\" is interpreted literally (instead of being interpreted according to its regex meaning).
- Note that this escaping sequence is relative to the regex compiler and not to PHP.
- This means that you must ensure that an actual backslash character reaches the regex functions by escaping it as needed

Additional Syntax For Regex

- Any regular expression followed by a "+" character is a regular expression composed of one or more instances of that regular expression.
- Any regular expression followed by a "?" character is a regular expression composed of either zero or one instance of that regular expression.
- Any regular expression followed by an expression of the type {min [, |, max]} is a regular expression composed of a variable number of instances of that regular expression.

Additional Syntax For Regex

- The min parameter indicates the minimum acceptable number of instances, whereas the max parameter, if present, indicates the maximum acceptable number of instances.
- If only the comma is present, no upper limit exists.
- If only min is defined, it indicates the only acceptable number of instances.
- Square brackets can be used to identify groups of characters acceptable for a given character position.

Some Basic Regex Usage

- It's sometimes useful to be able to recognize whether a portion of a regular expression should appear at the beginning or the end of a string.
- For example, suppose you're trying to determine whether a string represents a valid HTTP URL.
- The regex `http://` would match both http://www.cs.ucf.edu, which is valid and nhttp://www.cs.ucf.edu which is not valid, and could easily represent a typo on the user's part.
- Using the special character "^", you can indicate that the following regular expression should only be matched at the beginning of the string.
- Thus, `^http://` will match only the first of our two strings.

Some Basic Regex Usage

- The same concept – although in reverse – applies to the end-of-string marker "$, which indicates that the regular expression preceding it must end exactly at the end of the string.
- Thus, `com$` will match "amazon.com" but not "communication".

Some Basic Regex Usage

- Consider the regex: `.+@.+\..+`
- This regex can be used to indicate:
  - At least one instance of any character, followed by
  - The @ character, followed by
  - At least one instance of any character, followed by
  - The "." character, followed by
  - At least one instance of any character
- Can you guess what sort of string this regex might validate?

Does this look familiar? markl@cs.ucf.edu
It's a very rough form of an email address. Notice how the backslash character was used to force the regex compiler to interpret the next to last "." as a literal character, rather than as another instance of the "any character" regular expression.

Some Basic Regex Usage

- The regex on the previous page is a fairly crude way of checking the validity of an email address.
- After all, only letters of the alphabet, the underscore character, the minus character, and digits are allowed in the name, domain, and extension of an email.

Some Basic Regex Usage

- This is where the range denominators come into play.
- As mentioned previously, anything within non-escaped square brackets represents a set of alternatives for a particular character position.
- For example, the regex `[abc]` indicated either an “a”, a “b”, or a “c” character.
- However, representing something like “any character” by including every possible symbol in the square brackets would give rise to some ridiculously long regular expressions.

Some Basic Regex Usage

- Fortunately, range denominators make it possible to specify a "range" of characters by separating them with a dash.
- For example `[a-z]` means "any lowercase character.
- You can also specify more than one range and combine them with individual characters by placing them side-by-side.
- For example, our email validation regex could be satisfied by the expression `[A-Za-z0-9_]`.
- Using this new tool our full email validation expression becomes:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]+
```

Some Basic Regex Usage

- The range specifications that we have seen so far are all *inclusive* – that is, they tell the regex compiler which characters *can* be in the string.
- Sometimes, its more convenient to use *exclusive* specification, dictating that any character *except* the characters you specify are valid.
- This is done by prepending a caret character (^) to the character specifications inside the square bracket.
- For example, `[^A-Z]` means any character except any uppercase letter of the alphabet.

Some Basic Regex Usage

- Going back to our email example, its still not as good as it could be because we know for sure that a domain extension must have a minimum of two characters and a maximum of four.
- We can further modify our regex by using the minimum-maximum length specifier.

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{2,4}
```

- Naturally, you might want to allow only email addresses that have a three-letter domain.
- This can be accomplished by omitting the comma and the `max` parameter from the length specifier, as in:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{3}
```

Some Basic Regex Usage

- On the other hand, you might want to leave the maximum number of characters open in anticipation of the fact that longer domain extensions might be introduced in the future, so you could use the regex:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{3,}
```

- Which indicates that the last regex in the expression should be repeated at least a minimum of three times, with no fixed upper limit.

Some definitions

```
'robert@example.com'
```

Actual data string

```
'/^[a-z\d\._-]+@([a-z\d-]+\.)+
[a-z]{2,6}$/i'
```

Definition of the pattern (the 'Regular Expression')

PHP functions to *do something* with data and regular expression.

```
preg_match(), preg_replace()
```

Regex: Delimiters

- The regex definition is always bracketed by delimiters, usually a '/':

```
pattern: '/php/';

Matches: 'php', 'I love php', 'phpphp'
Doesn't match: 'PHP', 'I love ph'
```

The whole regular expression has to be matched, but the whole data string doesn't have to be used.

Regex: Case insensitive

- Extra switches can be added after the last delimiter.
- The **'i'** switch makes comparisons case insensitive

```
$regex = '/php/i';

Matches: 'php', 'I love pHp', 'PHP'
Doesn't match: 'I love ph', 'p h p'
```

Will it match 'phpPHP'?

Regex: Character groups

- A regex is matched character-by-character. You can specify multiple options for a character using square brackets:

```
$regex = '/p[huo]p/';

Matches: 'php', 'pup', 'pop'
Doesn't match: 'phup', 'ppp', 'pHp'
```

Will it match 'phpPHP'?

Regex: Character groups

- You can also specify a digit or alphabetical range in square brackets:

```
$regex = '/p[a-z1-3]p/';

Matches: 'php', 'pup',
         'ppp', 'pop', 'p3p'
Doesn't match: 'PHP', 'p5p', 'p p'
```

Will it match 'pa3p'?

Regex: Predefined Classes

\d	Matches a single character that is a digit (0-9)
\s	Matches any whitespace character (includes tabs and line breaks)
\w	Matches any alphanumeric character (A-Z, 0-9) or underscore.

Regex: Predefined classes

```
$regex = '/p\dp/';
Matches: 'p3p', 'p7p',
Doesn't match: 'p10p', 'P p'

$regex = '/p\wp/';
Matches: 'p3p', 'pHp', 'pop', 'p_p'
Doesn't match: 'phhp', 'p*p', 'pp'
```

Regex: the Dot

- The special dot character matches any character except for a line break:

```
$regex = '/p.p/';

Matches: 'php', 'p&p',
         'p(p', 'p3p', 'p$p'
Doesn't match: 'PHP', 'phhp'
```

Regex: Repetition

- There are a number of special characters that indicate the character group may be repeated:

`?`	Zero or 1 times
`*`	Zero or more times
`+`	1 or more times
`{a,b}`	Between a and b times

Regex: Repetition

```
$regex = '/ph?p/';
Matches: 'pp', 'php',
Doesn't match: 'phhp', 'pbp'

$regex = '/ph*p/';
Matches: 'pp', 'php', 'phhhhp'
Doesn't match: 'pop', 'phhohp'
```

Will it match 'phHp'?

Regex: Bracketed repetition

- The repetition operators can be used on bracketed expressions to repeat multiple characters:

```
$regex = '/(php)+/';
Matches: 'php', 'phpphp',
         'phpphpphp'
Doesn't match: 'ph', 'popph'
```

Will it match 'phpph'?

Regex: Repetition

```
$regex = '/ph+p/';
Matches: 'php', 'phhhhp',
Doesn't match: 'pp', 'phyhp'

$regex = '/ph{1,3}p/';
Matches: 'php', 'phhhp'
Doesn't match: 'pp', 'phhhhp'
```

Will it match 'pHHp'?

Regex: Anchors

- So far, we have matched anywhere within a string. We can change this behaviour by using anchors:

^	Start of the string
$	End of string

Regex: Anchors

- With NO anchors:

```
$regex = '/php/';

Matches: 'php', 'php is great',
         'I love php'
Doesn't match: 'pop'
```

Regex: Anchors

- With *start* anchor:

```
$regex = '/^php/';

Matches: 'php', 'php is great
Doesn't match:
         'I love php', 'pop'
```

Will it match 'PHP rocks!'?

Regex: Anchors

- With *start* and *end* anchors:

```
$regex = '/^php$/';

Matches: 'php',
Doesn't match: 'php is great',
          'I love php', 'pop'
```

Will it match 'php is php'?

Regex: Escape special characters

- We have seen that characters such as ?,.,$,*,+ have a special meaning. If we want to actually use them as a literal, we need to escape them with a backslash.

```
$regex = '/p\.p/';
Matches: 'p.p'
Doesn't match: 'php', 'p1p'
```

Will it match 'p..p'?

So.. An example

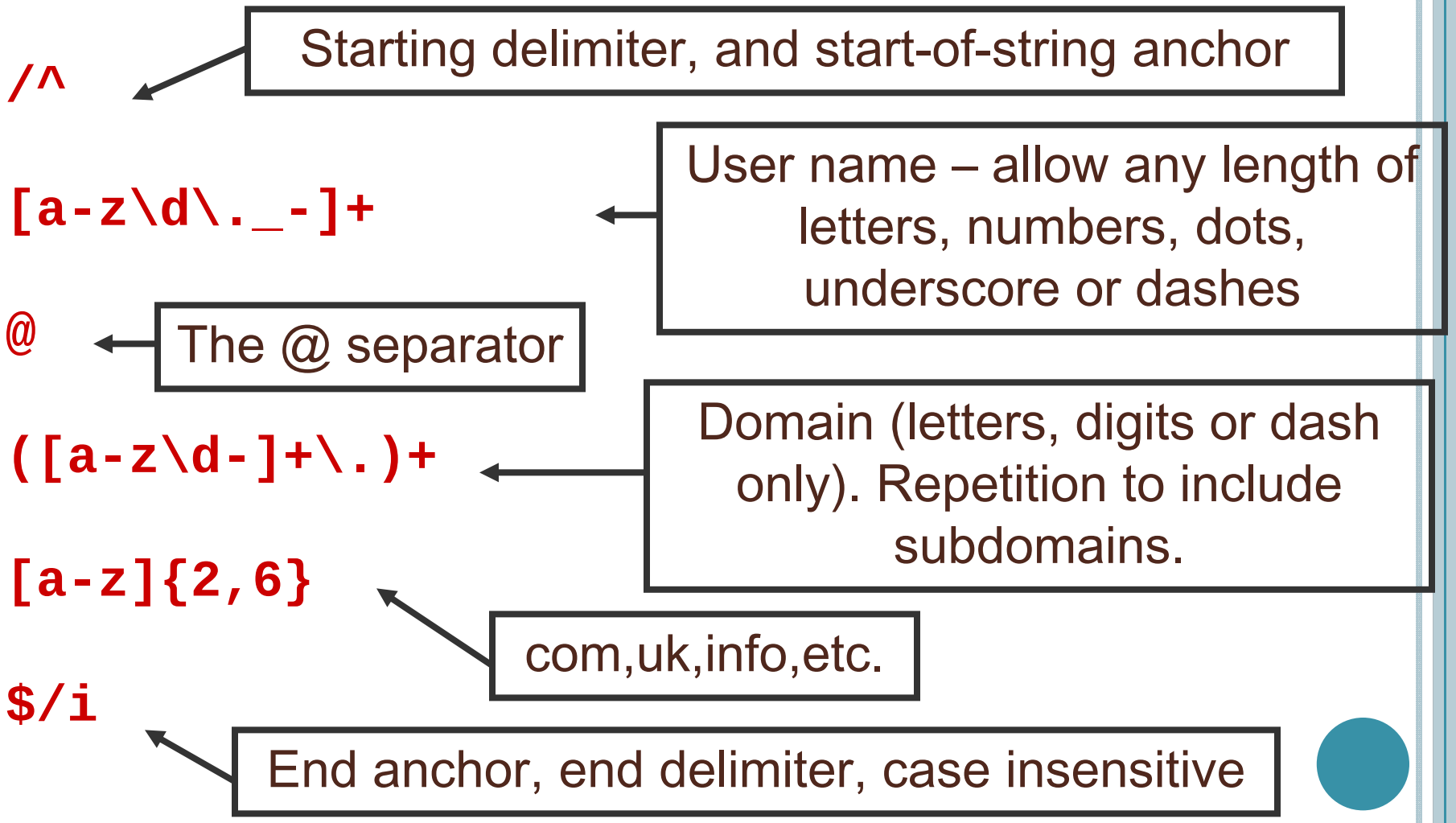
- Lets define a regex that matches an email:

```
$emailRegex = '/^[a-z\d\._-]+@([a-z\d-]+\.)+[a-
                    z]{2,6}$/i';

    Matches: 'rob@example.com',
             'rob@subdomain.example.com'
                 'a_n_other@example.co.uk'

    Doesn't match: 'rob@exam@ple.com'
                   'not.an.email.com'
```

So.. An example



Now What?

- How do we use Regular Expressions?
  - The **ereg()** function searches a string specified by string for a string specified by pattern, returning true if the pattern is found, and false otherwise. The search is case sensitive in regard to alphabetical characters
  - **preg_match()** tests to see whether a string matches a regex pattern. (For PCRE)
  - **ereg_replace()** is used to replace a string that matches a regex pattern
  - **preg_replace()** is used to replace a string that matched a regex pattern. (For PCRE)
- ereg() has been DEPRECATED as of PHP 5.3.0 and REMOVED as of PHP 6.0.0. Relying on this feature is highly discouraged.
- preg_match(), which uses a Perl-compatible regular expression syntax, is **often a faster alternative to ereg().**

POSIX Regular Expressions

- The simplest form or regex matching is performed through the `ereg()` function which has the following form:

```
ereg(pattern, string[, matches]);
```

- The `ereg()` function works by compiling the regular expression stored in `pattern` and then comparing it against `string`.
- If the regex is matched against string, the result value of the function is `true` – otherwise, it is `false`.
- If the `matches` parameter is specified, it is filled with an array containing all the references specified by `pattern` that were found in string.
- Position 0 in this array represents the entire matched string.

Pattern match

- We can use the function **preg_replace**() to replace any matching strings.
  - int **preg_match** (string $pattern , string $subject [, array &$matches [, int $flags = 0 [, int $offset = 0]]])
- **Pattern:** The pattern to search for, as a string.
- **Subject:** The input string.
- **Matches:** If matches is provided, then it is filled with the results of search. *$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.

Pattern match

- **Flags:** can be the following flag:
  - **PREG_OFFSET_CAPTURE** If this flag is passed, for every occurring match the appendant string offset will also be returned.
  - Note that this changes the value of matches into an array where every element is an array consisting of the matched string at offset *0* and its string offset into subject at offset *1*.
- **Offset:** Normally, the search starts from the beginning of the subject string.
  - The optional parameter offset can be used to specify the alternate place from which to start the search (in bytes).

PREG_MATCH

- We can use the **preg_match**() function to test whether a string matches or not.

```
// match an email
$emailRegex  =  '/^[a-z\d\._-]+@([a-z\d-]+\.)+[a-
  z]{2,6}$/i' ;
$input = ‘rob@example.com';
if (preg_match($emailRegex,$input) {
     echo 'Valid email';
} else {
     echo 'Invalid email';
}
```

A Practice Exercise

- See if you can create a POSIX based regex that will validate a string representing a date in the format `mm/dd/yyyy`.
  - In other words, 04/05/2011 would be matched but 4/5/11 would not.
- Step 1: form a basic regex. A regex such as .+ (one or more characters) is a bit too vague even as a starting point.
- So how about something like this?

```
[[:digit:]]{2}/[[:digit:]]{2}/[[:digit:]]{4}
```

- This will work and validate 04/05/2011. However, it will also validate 99/99/2011 which is not a valid date, so we still need some refinement.

A Practice Exercise (continued)

- For the month component of our regex, the first digit must always be either a 0 or a 1, but the second digit can be any of 0 through 9.
- Similarly, for the day component of the regex, the first digit can only be 0, 1, 2, or 3.
- Our final regex now becomes:

```
[0-1][[:digit:]]/[0-3][[:digit:]]/[[:digit:]]{4}
```

- This will work and validate 04/05/2011.

Using RegEx in Validation Functions

1. Write a function validZip that returns true if an input contains exactly 5 digits. Test the validZip function on an array of zip codes.
2. Write a function validText that returns true if an input contains only text, no numbers or symbols. Test the validText function on an array of strings.
3. Write a function validSid that returns true if an input contains a student ID in the form 880-88-3322. Test the validSid function on an array of SIDs.

Pattern replacement

- We can use the function **`preg_replace`**() to replace any matching strings.
  - mixed **preg_replace** (mixed $pattern , mixed $replacement , mixed $subject [, int $limit = -1 [, int &$count]])**Pattern:** The pattern to search for, as a string.
- **pattern** The pattern to search for. It can be either a string or an array with strings.
- **replacement** The string or an array with strings to replace. If there are fewer elements in the replacement array than in the pattern array, any extra patterns will be replaced by an empty string.
- **subject** The string or an array with strings to search and replace.
  - If subject is an array, then the search and replace is performed on every entry of subject, and the return value is an array as well.

Pattern replacement

- **limit** The maximum possible replacements for each pattern in each subject string. Defaults to *-1* (no limit).
- **count** If specified, this variable will be filled with the number of replacements done.
- **Return Values:** preg_replace() returns an array if the subject parameter is an array, or a string otherwise.
  - If matches are found, the new subject will be returned, otherwise subject will be returned unchanged or **NULL** if an error occurred.

Pattern replacement

- We can use the function **preg_replace**() to replace any matching strings.

```
//  replace  two  or  more  spaces  with
// a single space
$input = 'Some     comment string';
$regex = '/\s\s+/';
$clean = preg_replace($regex,' ',$input);
//'Some comment string'
```

POSIX Regular Expressions

- POSIX (**P**ortable **O**perating **S**ystem **I**nterface for uni**X**) is a collection of standards that define some of the functionality that a Unix operating system should support.
- One of these standards defines two flavors of regular expressions.
  - BRE (Basic Regular Expressions) standardizes a flavor similar to the one used by the traditional Unix `grep` command. This is probably the oldest regular expression flavor still in use today.
  - ERE (Extended Regular Expressions) standardizes a flavor similar to the one used by the Unix `egrep` command. Most modern regex flavors are extensions of the ERE flavor.
- The POSIX standard is the simplest form of regex available in PHP (as opposed to the PCRE), and as such is the best way to learn regular expressions.

POSIX Regular Expressions

- In addition, the POSIX regex standard defines the concept of character classes as a way to make it even easier to specify character ranges.
- Character classes are always preceded by colon characters (:) and must be enclosed in square brackets.
- There are 12 character classes defined in the POSIX standard. These are listed in the table on the following page.

Character class	Description
alpha	Represents a letter of the alphabet (either lower or upper case). Equivalent to `[A-Za-z]`
digit	Represents a digit between 0 and 9. Equivalent to `[0-9]`
alnum	Represents an alphanumeric character. Equivalent to `[0-9A-Za-z]`
blank	Represents "blank" characters, normally space and tab
cntrl	Represents "control" characters, such as DEL, INS, and so on
graph	Represents all printable characters except the space
lower	Represents lowercase letters of the alphabet only
upper	Represents uppercase letters of the alphabet only
print	Represents all printable characters
punct	Represent punctuation characters such as ".", or ","
space	Represents the whitespace
xdigit	Represents hexadecimal digits

The POSIX character classes

POSIX Regular Expressions

- Rewriting our previous email regex using the POSIX standard notation the following:

  ```
  [A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{2,4}
  ```

  becomes:

  ```
  [[:alnum:]_]+@[[:alnum:]_]+\.[[:alnum:]_]{2,4}
  ```

- This notation is a bit simpler, and it unfortunately also makes mistakes a little less obvious.

POSIX Regular Expressions

- Another important concept introduced by the POSIX standard is the reference.
- Recall that we discussed the use of parentheses to group regular expressions.
- When you use parentheses in a POSIX regex, when the expression is executed the interpreter assigns a numeric identifier to each grouped expression that is matched.
- This identifier can be used in various operations – such as finding and replacing.
- Consider the example on the following page:

POSIX Regular Expressions

- Suppose we have the string: `markl@cs.ucf.edu` and the regex:

  `([[:alnum:]_]+)@([[:alnum:]_]+)\.([[:alnum:]_]{2,4})`
- The regex should match the email address string.
- However, because we have grouped the username, domain name, and the domain extensions, they will each become a reference, as shown in the table below:

Reference Number	Value
0	`markl@cs.ucf.edu` (string matches the entire regex)
1	`markl`
2	`cs.ucf`
3	`edu`