Regular Expressions

> ***Some people, when confronted with a problem, think "I know, I'll use regular expressions."***
> ***Now they have two problems.*** — *Jamie Zawinski*

This quote illustrates two things about Regexes: their versatility and their complexity.

- A very powerful feature of PHP is *Regular Expressions*.
- Although they are built into the language, Regular Expressions (or *regexes*) are a whole language of their own. They define a *grammar* (in the computer-science sense) by which strings can be recognized.
- In this course, we will focus on practical use.
  - Regular expressions are primarily used for three things:
    1. Matching strings of text against *patterns* (does this string look like a phone number?)
    2. Parsing text to extract information (extract the area code from this phone number)
    3. Altering strings by deleting or swapping out a substring that matches a pattern (remove the area code to get the base phone number)
  - Regexes are far more robust than simply testing whether one string equals another, or using `substr` and `strpos` to pick apart a string.
- Note that PHP supports two versions of regexes: one called *POSIX* and one called *PCRE* (Perl-Compatible Regular Expressions). For some technical reasons, but mainly because many other programming languages support PCREs, I recommend using them. They are used exclusively in these notes.

The Regular Expression Grammar

The Basics

- A regex is represented as a string of text, surrounded by slashes.
  - One or more *pattern modifiers* can appear after the final slash; these will be covered later.
- Unless you use positional anchors (covered later) the pattern will match anywhere in the string. Therefore, the pattern
  ```
  /Bob/
  ```
  will match
  ```
  "Bob"
  "Bob is a guy"
  "My name is Bob"
  "I am called Bobby"
  "Let's see if Bob can go."
  ```
- In fact, it will match **any** string that has `"Bob"` as a substring.
- Note that regexes are (by default) **case-sensitive**. Thus, the above pattern will **not** match any of:
  ```
  "bob"
  "BOB"
  "bob is a guy."
  "My name is bob"
  "I am called bobby"
  "Let's see if BOB can go."
  ```

Quantifiers

- So far, we haven't replicated any functionality beyond `strpos`, but regular expressions can be far more complex. The next layer of functionality is *quantifiers*. These allow a *sub-pattern* (for example, a string or character) to match multiple times in a specific place.
- Use the `+` operator to match a sub-pattern **one or more times**:
  ```
  /(iss)+/
  ```
  will match any string that has the string `"iss"` in it one or more times:
  ```
  "Kissimmee"
  "Mississippi"
  "miss"
  "Miss Mississippi"
  ```
  but will not match any string that does not have `"iss"` in it.
  - Note in the final example that only the first `"iss"` was part of the match.

- Note that quantifiers apply to a single sub-pattern, which all else being equal is a single letter.
  - In other words, `/a|b+/` matches strings like `"x|a|b|x"`, `"x|a|bbb|x"`, and `"x|a|bbbbb|x"` (as well as, of course, strings like `"x|a|b|xabxabxabx"`).
  - You can create a sub-pattern with parentheses: `/(ab)+/` matches strings like `"x|ab|x"`, `"x|abab|x"`, and `"x|abababab|x"` (and `"x|abab|xababxababx"`).
- Combine the quantifier with other sub-patterns for more sophisticated matches:
  `/have a (very )+nice day/`
  will match
  `"have a very nice day"`
  `"have a very very nice day"`
  `"have a very very very nice day"`
  and so forth (as well as strings like `"I hope you have a very nice day today!"`), but will not match any of the following:
  `"have a nice day"`
  `"have a very very "`
  `"very very very nice day"`
  `"very very very "`
  all three sub-patterns must be matched for the overall regex to match.
  - The string `"nice day very have a "` also fails to match, because the sub-parts are all there, but **not in order**.
  - The string `"have a completely very good and nice day"` fails to match, because the sub-parts are all there, but **not adjacently**.
- Notice the space is inside the parentheses. Regexes are very sensitive to space and positioning, so be careful!
  - For example, consider the pattern
    `/have a (very)+ nice day/`
    Note the slight difference from the pattern above: the space is now outside the quantifier. The strings this pattern actually matches are:
    `"have a very nice day"`
    `"have a veryvery nice day"`
    `"have a veryveryvery nice day"`
  - Similarly, consider the pattern
    `/have a (very )+ nice day/`
    Here there is a space both inside and outside the quantifier. The strings this matches are:
    `"have a very  nice day"`
    `"have a very very  nice day"`
    `"have a very very very  nice day"`
    Note the double spaces. One is used to match the quantified sub-pattern; the other is needed to match the space outside the quantifier.
- Use the `*` operator to match a sub-pattern **zero or more times**:
  `/(iss)*/`
  will still match
  `"K|iss|immee"`
  `"M|ississ|ippi"`
  `"m|iss|"`
  and will **also** match strings with zero occurrence of `"iss"` in them. This is obviously not very useful by itself (because every string has zero or more occurrences of `"iss"` in it), but when used in combination with other sub-patterns...
  `/have a (very )*nice day/`
  will match strings containing
  `"have a very nice day"`
  `"have a very very nice day"`
  `"have a very very very nice day"`
  etc., and this time will also match:
  `"have a nice day"`
  but will still not match any of the following:
  `"have a very very "`
  `"very very very nice day"`
  `"very very very "`
  - Again, be careful of spacing:
    `/have a (very)* nice day/`
    matches
    `"have a very nice day"`
    `"have a veryvery nice day"`
    `"have a veryveryvery nice day"`
    and
    `"have a  nice day"` (note the double spaces)

- Use the **?** operator to match a sub-pattern **zero or one times**:
  `/have a (very )?nice day/`
  will match strings containing
  `"have a nice day"`
  `"have a very nice day"`
  and nothing else.
  - Final warning regarding spacing:
    `/have a (very)? nice day/`
    matches
    `"have a very nice day"`
    and
    `"have a  nice day"` (again, note the double spaces)
- Use braces for arbitrary quantification
  - **`sub-pattern{count}`** matches exactly **`count`** instances of **`sub-pattern`**
  - **`sub-pattern{min,max}`** matches between **`min`** and **`max`** instances of **`sub-pattern`**, inclusive
  - **`sub-pattern{min,}`** matches at least **`min`** instances of **`sub-pattern`**
  - **`sub-pattern{0,max}`** matches at most **`max`** instances of **`sub-pattern`** (or none)
  - **`sub-pattern{1,max}`** matches at most **`max`** instances of **`sub-pattern`** (but at least one)
  - Examples:
    ```
    /have a (very ){3}nice day/
    /have a (very ){0,3}nice day/
    /have a (very ){1,5}nice day/
    /have a (very ){3,5}nice day/
    /have a (very ){2,}nice day/
    ```
  - Note these can all be represented without braces:
    ```
    /have a (very )(very )(very )nice day/
    /have a (very )?(very )?(very )?nice day/
    /have a (very )(very )?(very )?(very )?(very )?nice day/
    /have a (very )(very )(very )?(very )?nice day/
    /have a (very )(very )+nice day/
    ```
    But the braces are much easier to interpret.
  - Also, note that the **+**, *****, and **?** quantifiers can be represented using braces:

the quantifier...	equivalent to...
`sub-pattern+`	`sub-pattern{1,}`
`sub-pattern*`	`sub-pattern{0,}`
`sub-pattern?`	`sub-pattern{0,1}`

    but again, the simple quantifiers are easier to understand in most cases.
    - And `/(subpattern)+/` is equivalent to `/(subpattern)(subpattern)*/`
  - Remember that patterns can match anywhere in the string, so the pattern `/x{3}/` will match any string with three **x**s in a row, including strings with **four** **x**s in a row (or more).
    - Typically quantifiers are used with positional anchors (see below) or are adjacent to other sub-patterns that limit the match.
    - For example, the pattern `/ax{3}a/` will match strings containing `"axxxa"`, (exactly three **x**s surrounded by **a**s). Four **x**s surrounded by **a**s (`axxxxa`) won't match.

Alternation

- *Alternation* allows you to choose one of a list of possible choices.
- Use a vertical bar (**|**) for alternation:
  `/man|bear|pig/`
  Will match all strings containing `"man"`, `"bear"`, or `"pig"`.
- Note that `/abc|xyz/` is equivalent to
  `/(abc)|(xyz)/`
  not
  `/(ab)(c|x)(yz)/`
- Spaces and alternation don't interact "intuitively": `/A B|C D/` is the same as
  `/(A B)|(C D)/`
  not
  `/(A )(B|C)( D)/`

- This may not be what you expect, given the above example `/|a|b+|/`, (which is equivalent to `/|(a)|(b+)|/`).
  - The explanation has to do with the precedence of operators in regexes.
  - When using alternation and spaces, (and in general when using regexes) parentheses are recommended for clarity.
  - In practical use, something like:
    `/|I live in |(ME|NH|VT|MA|RI|CT)|/`
    will match statements entered by people who live in New England.

Character Classes

- Several symbols represent a *character class*, or any of a group of characters:
  - `\s` means "any space character"
  - `\S` means "any non-space character"
  - `\d` means "any digit"
  - `\D` means "any non-digit"
  - `\w` means "any *word character* (letters, numbers, underscores)"
  - `\W` means "any non-word character"
  - `.` (a dot) means "any single character" except newline (unless you use the `s` pattern modifier, in which case it matches a newline as well).
- To define a custom character class, use brackets, and possibly dashes:
  - A single character means "this character is part of the class".
  - Two characters separated by a dash means "all the characters (ASCIIbetically) between the two, inclusively".
  - The class can contain multiple characters and ranges, which are simply placed adjacently.
  - Examples:
    - `[13579]` matches any single odd digit.
    - `[a-m]` matches any lowercase letter between `a` and `m`, while
    - `[A-M]` matches any uppercase letter in the same range.
    - `[A-Ma-m]` matches either
- A *complement* class matches any character **except** those specified. To specify a complement class, use a caret as the first character inside the brackets.
  Examples:
  - The character class
    `[^A-Z]`
    matches anything **except** an uppercase letter.
  - The character class
    `[^aeiouAEIOU]`
    matches anything **except** a vowel.
- Character classes can include the "named" classes:
  `[\dA-F]`
  matches any (uppercase) hex digit. The class
  `[^\d\s]`
  matches any character besides digits and whitespace.
- Be careful when mixing uppercase and lowercase letters.
  - For example,
    `[A-Za-z]`
    matches anything from `A` to `Z` or `a` to `z`, or in other words, any letter character.
  - However
    `[A-z]`
    matches any character between `A` (ASCII value 65) and `z` (ASCII value 122), which includes all upper- and lower-case letters **but also includes** brackets and other punctuation characters you probably did not intend to be part of the character class.
- Discussion question: what characters are in the class `[^\W_]`?

- No matter how many characters are in a class, a class only matches one character (it can be one of any of the characters in the class). So the pattern
  `/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/`
  despite having 26 characters in it, only matches a single uppercase letter.
  - If you want to match more than one of a class, you must replicate the class, as in
    `/[ABCDEFGHIJKLMNOPQRSTUVWXYZ][ABCDEFGHIJKLMNOPQRSTUVWXYZ]/`
    which matches two uppercase letters.
  - You may also (preferably) use quantification (see above).
- Classes can be used with repetition and alternation operators. Example:
  `/\d{5}(-\d{4})?/`
  matches a ZIP code.
  `/[aeiou]+/`
  matches one or more vowels.
- Another example:
  `/have a .*day/`
  (remember that dot is a character class) matches
  `"have a nice day"`
  `"have a very nice day"`
  `"have a Tuesday"`
  `"have a rotten day"`
  `"have a Ms3!a 54dPO2$ -)eEq1day"`
  `"have a day"`
  `"yo, dude, have a day!!!"`
- Classes are more efficient than alternation, so the pattern
  `/([aeiou])+/`
  will match much faster than the pattern
  `/(a|e|i|o|u)+/`
  even though they are functionally equivalent.

Anchors

- Anchors match conditions in the string but **do not match characters.**
- `^` matches the beginning of the string (or at the beginning of any "line", if you use the `m` pattern modifier and the string contains embedded newlines)
- `$` matches at the end of the string (or at the end of any "line", under the circumstances given above)
- The pattern `/^hi/` will match the strings
  `"hi"`
  `"hi there"`
  `"history"`
  but will not match
  `"this"`
  `"I said hi"`
  `" hi"` (note the space)
  `"Hillside"` (note the upper-case)
- Similarly, the pattern `/st$/` will match the strings
  `"st"`
  `"most"`
  `"mnopqrst"`
  `"Come here Fast"`
  but will not match
  `"state"`
  `"st "` (note the space)
  `"Come here FAST"` (note the upper-case)
- Using them together: Remember, patterns can match anywhere in the string being matched. Therefore, recall the pattern
  `/have a (very )+nice day/`
  will match
  `"have a very nice day"`
  `"have a very very very nice day"`
  `"DO NOT have a very nice day, you bonehead"`
  Perhaps that last one should be excluded. Add anchors to correct this:
  `/^have a (very )+nice day$/`
  will no longer match strings besides
  `"have a very nice day"`
  `"have a very very very nice day"`
  and so forth

- Note that `$` matches the end of the string, **before the newline** if any.
  - For example, the pattern `/^hello$/` matches both `"hello"` and `"hello\n"` (the latter being similar to what you would read in from a file).
- Word boundaries:
  - Consider the string `"ab_c .xyz"` (why you are considering this particular string is left to the imagination). Where are the "words"?
  - `\b` matches at a word boundary (between a `\w` character and a `\W` character, or at the beginning/end of a string if the first/last character is a word character):
    `"ab_c .xyz!!"`
  - `\B` matches at a non-word-boundary (between two `\w` characters or between two `\W` characters, or at the beginning/end of a string if the first/last character is a non-word character)):
    `"ab_c .xyz!!"`
  - As you recall, the pattern `/it/` will match any string with the letters `"it"` in it, including:
    `"it"`
    `"spit"`
    `"itch"`
    `"fritter"`
    `"this is it!"`
    `"is it here?"`
    `"hit it"` (Only one needs to match for the whole string to match.)
    If you are looking for strings with the word `"it"` by itself, use the `\b` anchor:
    `/\bit\b/`
    will match only the following from the above list:
    `"it"`
    `"this is it!"`
    `"is it here?"`
    `"hit it"`
    Alternatively,
    `/\Bit\B/`
    will match only the following from the above list:
    `"fritter"`
    This is because `\B` **does not match** at the beginning and end of strings. To match strings containing words with the letters `"it"` but not the word `"it"` itself, you must do:
    `/(it\B)|(\Bit)/`
- An important point: anchors do **not** match characters. They match **positions** or **conditions** within the string.

Escaped Characters

- To literally match one of the characters in the regex grammar, escape it with a backslash.
- Example 1:
  `/^\$[a-zA-Z_]\w*\[[0-9]+\]$/`
  matches legal PHP array lookups (with a numeric index), such as `$foo[3]`.
  - Note the backslash before the first dollar sign (because it's meant literally) and the lack of backslash before the second one (because it's matching the end of the string).
  - Note also which brackets are escaped (literal) and which are defining character classes.
- Example 2:
  `/^\d+\.\d+\.\d+\.\d+$/`
  matches IP addresses (you might then want to check that each part was between 0 and 255).
- Inside a character class, you do not need to escape anything except `[`, `]`, and `\`.
  - You don't need to escape `-`; instead put it as the first (after any initial `^`) or last character in the class and PHP will figure out what you mean.
  - You don't need to escape `^`; just put it anywhere besides first.
  - Bonus discussion question: what character(s) do the following classes match?
    `[--^]  [^-^]  [-^]`
    `[^--^]  [^-]  [^^]`
- You must also escape literal slashes.
- Recall also that regexes are stored as strings. So you must escape certain characters (primarily dollar signs, backslashes, and quotes) from the string itself. Thus, the regex
  `/^\$[a-z]$/`
  would actually be specified as
  `$regex = "/^\\\$[a-z]\$"`
  - The backslash that escapes the first dollar sign from the regex must be escaped from the string

- The first dollar sign (already escaped from the regex) must be escaped from the string
- The second dollar sign must be escaped from the string

You can use single-quotes to avoid most of this:

```
$regex = '/^\$[a-z]$'
```

but if you want to perform interpolation (see below) you must use double quotes and be wary of characters that must be escaped.

- You will note that most of my examples use single-quotes when specifying regexes–unless interoplation is being performed.

Case Sensitivity

- As mentioned, regular expressions are case-sensitive.
- It is always possible to write a regex that ignores case, by using `[a-zA-Z]` for a letter instead of `[a-z]` or `[A-Z]` and by specifying literal text as `[Pp][Ee][Tt][Ee][Rr]`.
- However, if you have long strings of literal text, this is very cumbersome.
- Instead, use the `i` modifier to cause the regex to match case-insensitively.
- Just add an `i` after the regex, like this: `$rgx = "/^regex$/i";`.
- Here is an example:

```
1  $str1 = "I am Bobby";
2  $str2 = "I AM BOBBY";
3
4  $rgx = '/Bob/';
5  print "using regex $rgx:\n";
6  print "  \"$str1\" " . (preg_match($rgx, $str1) ? "matches" : "doesn't match") . "\n";
7  print "  \"$str2\" " . (preg_match($rgx, $str2) ? "matches" : "doesn't match") . "\n\n";
8
9  $rgx = '/Bob/i';
10 print "using regex $rgx:\n";
11 print "  \"$str1\" " . (preg_match($rgx, $str1) ? "matches" : "doesn't match") . "\n";
12 print "  \"$str2\" " . (preg_match($rgx, $str2) ? "matches" : "doesn't match") . "\n";
```

outputs:

```
using regex /Bob/:
  "I am Bobby" matches
  "I AM BOBBY" doesn't match

using regex /Bob/i:
  "I am Bobby" matches
  "I AM BOBBY" matches
```

Rules For Matching

- In addition to the regex grammar itself, remember two things and you won't be steered wrong:
  1. Patterns always try to match the **LONGEST LEFTMOST** substring.
     - In other words, if a pattern can match two places in a string, it always matches as far to the left as possible.
     - If there are two possible matches at that leftmost spot, the longest one is the substring that is actually matched.

     Rule #1 only applies to PCREs. POSIX regexes follow different rules.

  2. When a pattern is matched against a string, the string must match all sub-patterns (be they individual characters, anchors, parenthetical expressions, alternations, or quantifications) **IN-ORDER, ADJACENTLY**.

Regex Functions

- The basic regex functions are:
  - `preg_match()` - Perform a regex match
  - `preg_match_all()` - Perform a global regex match
  - `preg_quote()` - Escape regex characters
  - `preg_replace()` - Perform a search and replace using a regex
  - `preg_split()` - Split a string using a regex

- **`preg_grep()`** - Return array items that match a regex

- **`preg_match($rgx, $str, [$matches_arr])`**
  - Test the string **`$str`** against the pattern **`$rgx`**.
  - Returns true or false depending on whether the match succeeded.
  - The optional **`$matches_arr`** argument is an array. If it is provided, and the match succeeded, the text which matched (it might or might not be the entire string) is placed in **`$matches_arr[0]`**.
    - If there were capturing parentheses (see below), their output will be placed in **`$matches_arr[1]`**, **`$matches_arr[2]`** and so on.
  - Example:

```
$strs = array(
                "432-8696",
                "1-508-393-0155",
                "603-424-2661",
                "424-2661 ",
                "(603) 432-8696"
              );

$rgx_phone = '/^(\d{3}-)?\d{3}-\d{4}$/';

foreach ($strs as $str) {
  print "testing \"$str\": ";
  if (preg_match($rgx_phone, $str)) {
    print "PHONE NUMBER\n";
  }
  else {
    print "not a phone number\n";
  }
}
```

  outputs:

```
testing "432-8696": PHONE NUMBER
testing "1-508-393-0155": not a phone number
testing "603-424-2661": PHONE NUMBER
testing "424-2661 ": not a phone number
testing "(603) 432-8696": not a phone number
```

- **`preg_match_all($rgx, $str, $matches_arr)`**
  - Test the string **`$str`** against the pattern **`$rgx`**.
  - If a match is found, the function keeps looking for more matches, starting from the end of the successful match.
  - Returns the number of times the pattern matched the string.
  - **`$matches_arr`** is required, and is used by the function to place the results of the match. It ends up being a multi-dimensional array with full-string matches an array in **`$matches_arr[0]`**, matches of first capturing parentheses in **`$matches_arr[1]`**, etc.
  - Example:

```
$text = "See Spot. See Spot run. Run, Spot, run!";
$rgx_sentence = '/[A-Z][^.]*\./';

preg_match_all($rgx_sentence, $text, $matches);
foreach ($matches[0] as $sentence) {
  print "found sentence: \"$sentence\"\n";
}
```

  outputs:

```
found sentence: "See Spot."
found sentence: "See Spot run."
```

- Well, that's not quite right. Here's how to fix it:

```
$text = "See Spot. See Spot run. Run, Spot, run!";
$rgx_sentence_fixed = '/[A-Z][^.!?]*[.!?]/';

preg_match_all($rgx_sentence_fixed, $text, $matches);
foreach ($matches[0] as $sentence) {
  print "found sentence: \"$sentence\"\n";
}
```

outputs:

```
found sentence: "See Spot."
found sentence: "See Spot run."
found sentence: "Run, Spot, run!"
```

- **`preg_quote($str, $delim)`**
  - Returns **`$str`**, modified so that special regex characters are escaped.
  - If you embed a string in a regex, any special characters in the string are treated **as regex grammar**.
  - By escaping the string's regex characters, the embedded string will be matched literally.
  - The **`$delim`** argument should be set to the character used to delimit the regex (usually `'/'`).
  - Example:

```
$award_grade = "A+";
$rgx_award = "/$award_grade/";

$students = array("Bill: A", "Jill: B+", "Will: B-", "Gil: A+", "Phil: F", "Fran: A-");

foreach ($students as $student) {
  if (preg_match($rgx_award, $student)) {
    print "$student\n";
  }
}

print "\nregex is: \"$rgx_award\"\n";
```

outputs:

```
Bill: A
Gil: A+
Fran: A-

regex is: "/A+/"
```

    - Note this did not work. The **`+`** in the regex is a treated as the "one or more" quantifier, so any string with one or more **`A`** characters will be matched.

  - Here is the correct way to do it:

```
$award_grade = "A+";
$rgx_award = "/" . preg_quote($award_grade) . "/";

$students = array("Bill: A", "Jill: B+", "Will: B-", "Gil: A+", "Phil: F", "Fran: A-");

foreach ($students as $student) {
  if (preg_match($rgx_award, $student)) {
    print "$student\n";
  }
}

print "\nregex is: \"$rgx_award\"\n";
```

outputs:

```
Gil: A+

regex is: "/A\+/"
```

Note the **`+`** is escaped with a backslash now.

    - Of course it's possible to just manually escape special characters—and it's probably clearer to do so with a very short regex like this example—but the **`preg_quote`** function won't forget any.

- **`preg_replace($rgx, $repl, $str, [$limit], [$count])`**
  - Matches the string **`$str`** against the pattern **`$rgx`**.
  - If the pattern is matched, the substring matched by **`$rgx`** is replaced by the (literal) string **`$repl`**.
  - Does a global substitution by default; to limit the number of substitutions, use the optional **`$limit`** variable.
  - If you pass the optional **`$count`** variable, it will be set to the number of replacements actually made.
  - Returns the string with replacements done (which is the string unchanged if the match failed).
  - Example:

```
$str = "Bahama mama";
$rgx_work = '/(ma)+/';
$rgx_fail = '/^x/';
$repl = "pop";

$new_str_work1 = preg_replace($rgx_work, $repl, $str);
$new_str_work2 = preg_replace($rgx_work, $repl, $str, 1);
$new_str_fail = preg_replace($rgx_fail, $repl, $str);

print "original string: \"$str\"\n";
print "modified string 1 (worked): \"$new_str_work1\"\n";
print "modified string 2 (worked): \"$new_str_work2\"\n";
print "modified string (failed): \"$new_str_fail\"\n";
```

    outputs:

```
original string: "Bahama mama"
modified string 1 (worked): "Bahapop pop"
modified string 2 (worked): "Bahapop mama"
modified string (failed): "Bahama mama"
```

- **`preg_split($rgx, $str, [$limit])`**
  - Split a string **`$str`** into pieces wherever the delimiter pattern **`$rgx`** matches.
    - This is superior to **`explode`** when the delimiter is variable.
  - If the optional **`$limit`** argument is provided, then a maximum of that many pieces will be returned. The last piece will be everything after the **`$limit-1`**th delimiter is found and may have additional delimiters embedded in it.
  - Example:

```
$data = "1, 32 , 8,36,84 ,72";

$bad_results = explode(",", $data);
$good_results = preg_split('/\s*,\s*/', $data);

print "bad results:\n";
foreach ($bad_results as $result) {
  print "    \"$result\"\n";
}

print "good results:\n";
foreach ($good_results as $result) {
  print "    \"$result\"\n";
}
```

    outputs:

```
bad results:
    "1"
    " 32 "
    " 8"
    "36"
    "84 "
    "72"
good results:
    "1"
    "32"
    "8"
    "36"
    "84"
    "72"
```

- **`preg_grep($rgx, $strings_arr)`**
  - Returns an array of the elements of **`$strings_arr`** which match the pattern **`$rgx`**.
  - Example:

```
$strs = array(
                "432-8696",
                "1-508-393-0155",
                "603-424-2661",
                "424-2661 ",
                "(603) 432-8696"
              );

$rgx_phone = '/^(\d{3}-)?\d{3}-\d{4}$/';
$phone_nums = preg_grep($rgx_phone, $strs);

print "phone numbers:\n";
foreach ($phone_nums as $result) {
  print "    \"$result\"\n";
}
```

  outputs:

```
phone numbers:
    "432-8696"
    "603-424-2661"
```

Regex Memory

- As seen, parentheses are used in regexes for *grouping* but they also have *memory*.
- What is matched inside the parentheses is *captured*, and can be accessed after the match, depending on the function used to do the matching.
- This is, for example, accomplished using the **`preg_match()`** function. If this function is passed an array as a third argument, the captured substrings are put in it.
- Example:

```
$str = "phone number is 603-432-8696.";
$rgx_phone = '/(\d{3})-(\d{3})-(\d{4})/';

print "Searching the string \"$str\"\n";
if (preg_match($rgx_phone, $str, $parts)) {
  print "Found a phone number.  \$parts contains: ";
  print_r($parts);
}
else {
  print "Didn't find a phone number\n";
}
```

  outputs:

```
Searching the string "phone number is 603-432-8696."
Found a phone number.  $parts contains: Array
(
    [0] => 603-432-8696
    [1] => 603
    [2] => 432
    [3] => 8696
)
```

- Note that **`$parts[0]`** contains the portion of the string which matched, which in this case is **not** the whole string.
- Subsequent elements of **`$parts`** correspond to each parenthetical expression in the pattern, in order of appearance of the open-parenthesis, left-to-right.
- You can also use the submatches in the replacement string by referring to them as **`$1`**, **`$2`**, etc.

- Example:

```
$str = 'name = Mr. John Smith';
$rgx_name = '/([A-Z][a-z]+\.)\s+[A-Z][a-z]+\s+([A-Z][a-z]+)/';

print "string before: \"$str\"\n";
$str = preg_replace($rgx_name, "$1 $2", $str);
print "string after: \"$str\"\n";
```

outputs:

```
string before: "name = Mr. John Smith"
string after: "name = Mr. Smith"
```

- Finally, you can even use the submatches later in the same pattern by referring to then as `\1`, `\2`, etc.
  - Example:

```
$text_ok = "This is the hour of our discontent";
$text_rpt = "This is the the hour of our discontent";
$rgx_rpt_words = '/\b([a-z]+)\s+\1/i';

foreach (array($text_ok, $text_rpt) as $text) {
  print "- searching for repeated words in string:\n";
  print "    \"$text\"\n";
  if (preg_match($rgx_rpt_words, $text, $repeats)) {
    print "  found repeated word \"$repeats[1]\"\n";
  }
  else {
    print "  no repeated words found\n";
  }
}
```

outputs:

```
- searching for repeated words in string:
    "This is the hour of our discontent"
  no repeated words found
- searching for repeated words in string:
    "This is the the hour of our discontent"
  found repeated word "the"
```

  - This is known as *backreferencing*.
- What a match variable contains depends on how its corresponding parenthetical expression figures into the overall match.
  - If a sub-pattern matches multiple times because it (or a sub-match that contains it) is followed by a quantifier, the match variable associated with that expression will contain the **last thing that matched**.
  - If a sub-pattern is not included in the match (for instance, because it is on one side of an alternation, or is followed by a "zero-or-more" or "zero-or-one" quantifier), then the associated match variable will be undefined.
  - For example, the code

```
$regex = "/^((a)|(b))(\\d+)([a-z])+\$/";

$strings = array("a432ncjkds", "b63jaq");
foreach ($strings as $string) {
  if (preg_match($regex, $string, $parts)) {
    print "\"$string\" matched. Parts are: ";
    print_r($parts);
  }
}
```

outputs:

```
"a432ncjkds" matched. Parts are: Array
(
    [0] => a432ncjkds
    [1] => a
    [2] => a
    [3] =>
    [4] => 432
    [5] => s
)
"b63jaq" matched. Parts are: Array
(
    [0] => b63jaq
    [1] => b
    [2] =>
    [3] => b
    [4] => 63
    [5] => q
)
```

- Pay attention to what items 1, 2, and 3 are based on whether the string started with an **a** or a **b**.
- Pay attention to the difference between what 4 and 5 capture.

- If you want to group sub-patterns in a regular expression but don't want to capture them, you can use *non-capturing* parentheses.
  - They are specified as **(?: ...)**.
  - They group like ordinary parentheses, but their contents are not captured.
  - For example:

```
1  $names = array("Mr. John Smith", "Bill Jackson", "Mrs. White", "Bill");
2  $rgx_lastname = '/^(?:[A-Z][a-z]+\.)?\s*(?:[A-Z][a-z]+)?\s+([A-Z][a-z]+)/';
3
4  foreach ($names as $name) {
5    if (preg_match($rgx_lastname, $name, $matches)) {
6      print "For the name \"$name\", got last name \"{$matches[1]}\"\n";
7    }
8    else {
9      print "The name \"$name\" did not match\n";
10   }
11 }
```

outputs:

```
For the name "Mr. John Smith", got last name "Smith"
For the name "Bill Jackson", got last name "Jackson"
For the name "Mrs. White", got last name "White"
The name "Bill" did not match
```

- Note in this example, even though there are three sets of parentheses, only one set captures anything. This simplifies subsequent operations.

  - Noncapturing parentheses are usually good to use unless you know you will be capturing.
    - However, they make a regular expression harder to read because they are more complicated. For this reason, these notes use regular parentheses.

More Regex Features

Extended Regexes

- By using the **x** modifier, you can insert whitespace and comments (beginning with a hash character).
- This makes regular expressions much easier to use.
- For example, consider the regular expression

```
$rgx_int = '/^([-+])?(\d+)(?:[eE](\d{1,3}))?$/';
```

- Using extended regexes, this regex can be written as

```
$rgx_int = '/
             ^               # beginning of string

             (               # begin capture $1: sign
               [-+]          #   plus or minus
             )               # end capture $1
             ?               # sign is optional

             (               # capture $2: coefficient
               \d+           #   one or more digits
             )               # end capture $2

             (?:             # group
               [eE]          # literal e or E
               (             # capture $3: exponent
                 \d{1,3}     #   one to three digits
               )             # end capture $3
             )
             ?               # exponent is optional

             $               # end of string
           /x';
```

- These regular expressions are equivalent, but the second is easier to comprehend.
- To include spaces or hash characters in an extended regex, escape them or include them in a character class.

Interpolation

- Regular expressions are just strings, stored in string variables. As such, you can interpolate variables (such as other strings) into them.
- We have already seen that when you interpolate string data into a regular expression, it is often a good idea to escape any regex characters in the substrings with **`preg_quote()`**.
- However, you might want to preserve the regex functionality of a string when interpolating it into a regular expression, if it is a regular expression itself.
- This allows you to build a complicated expression using a *top-down* method.
  - Define a complex regex simply by using sub-regexes. Then define the sub-regexes, possible with more sub-regexes. Continue until you have built a complex regular expression much more easily than by starting from scratch.
  - For example, a regular expression to define any possible email address would be highly complex. So instead, start with the following:

```
$rgx_email = '/^($rgx_user)@($rgx_host)$/';
```

  - Then define each of the two subregexes:

```
$rgx_host = '(?:$rgx_dns_name|$rgx_ipaddr)';
$rgx_user = '(?:[a-zA-Z0-9._-]+)';

$rgx_email = '/^($rgx_user)@($rgx_host)$/';
```

  - Keep going until you have defined all sub-levels:

```
$rgx_octet = '(?:\d{1,2}|[01]\d{2}|2[0-4]\d|25[0-5])';
$rgx_ipaddr = '(?:(?:$rgx_octet\.){3}$rgx_octet)';
$rgx_dns_tld = '(?:[a-zA-Z]{2,4})';
$rgx_dns_comp = '(?:[a-zA-Z0-9]+[a-zA-Z0-9-]*[a-zA-Z0-9]+|[a-zA-Z0-9]+)';
$rgx_dns_name = '(?:(?:$rgx_dns_comp\.)+$rgx_dns_tld)';
$rgx_host = '(?:$rgx_dns_name|$rgx_ipaddr)';
$rgx_user = '(?:[a-zA-Z0-9._-]+)';
$rgx_email = '/^($rgx_user)@($rgx_host)$/';
```

  - Note how much easier this is to read than if it were all combined into one monstrous regular expression:

```
$rgx_email = '/^([a-zA-Z0-9._-]+)@((?:(?:(?:[a-zA-Z0-9]+[a-zA-Z0-9-]*
              [a-zA-Z0-9]+|[a-zA-Z0-9]+)\.)+(?:[a-zA-Z]{2,4}))|
              (?:(?:(?:\d{1,2}|[01]\d{2}|2[0-4]\d|25[0-5])\.){3}
              (?:\d{1,2}|[01]\d{2}|2[0-4]\d|25[0-5])))$/';
```

- The *bottom-up* method is the same, except you define the pieces you know you will need first, then glue them together.
  - In practice, you will often come at a complex regex from both sides at the same time.
- There are a couple of caveats to remember when using interpolation to build complex regular expressions:
  - Remember that you only want to surround the overall regex with slashes. Sub-regexes should **not** be enclosed in slashes.
  - It is generally a good idea to enclose regexes to be interpolated in (non-capturing) parentheses in order to prevent unexpected interactions between adjacent interpolations.
  - Remember that positional anchors (beginning and ending of string) are intepreted in the context of the overall regex. Placing positional anchors in sub-regexes will not normally work.
  - Capturing parentheses are numbered left to right in the overall interpolated regex. When interpolating, use capturing parentheses only when you need to actually capture something.

Constructing Regexes Programmatically

- String operations such as `join` and concatenation work on regexes as well.
- This allows you to create very long and cumbersome regexes in code using some data structure as a base (perhaps even using user-supplied data).
- Here is a simple example:

```
1 $months = array("Jan", "Feb", "Mar", "Apr", "May", "Jun",
2                 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec");
3
4 $month_abbrs = join("|", $months);
5 $rgx_date = '/^(\d{4})-($month_abbrs)-(\d{2})$/i';
6
7 print "Regex to match dates is $rgx_date\n";
```

  which outputs

```
Regex to match dates is /^(\d{4})-($month_abbrs)-(\d{2})$/i
```

- Also, remember that regexes can be passed to and returned from subroutines.

Greed

- The repetition operators `*` and `+` are *greedy*.
  - This means they match as much of the string as possible. Example:

```
1 $html_line = 'Here is a <B>bold</B> thing and an <I>italic</I> thing';
2 print "HTML is: \"$html_line\"\n";
3 print "(Trying to strip HTML tags)\n";
4 $text_line = preg_replace('/<.+>/', '', $html_line);
5 print "TEXT is: \"$text_line\"\n";
```

    outputs:

```
HTML is: "Here is a <B>bold</B> thing and an <I>italic</I> thing"
(Trying to strip HTML tags)
TEXT is: "Here is a  thing"
```

  - Oh no! What happened?
    - The pattern is three sub-patterns:
      - subpat1: (`<`) Match a literal `<`
      - subpat2: (`.+`) Match one or more characters
      - subpat3: (`>`) Match a literal `>`
    - On the first pass, subpat1 matches the `<` in `<B>`
    - The pattern match continues from that point with subpat2, which is `.+`. This is intended to match everything inside the HTML tag (the `B`, in the first case, the `/B` in the second, but instead matches as much as possible, including the `>` ending the `<B>` tag, the entire `</B>` tag, the `<I>` tag and the `</I` towards the end, plus all the text in between.
    - Then subpat3 tries to match. It succeeds because there is another `>` after `</I`.
    - All three sub-patterns have matched so the overall pattern is a match.
      - However, the text actually matched within the string was: `<B>bold</B> thing and an <I>italic</I>`.
      - Therefore when the substitution is done, much more of the string is removed than was intended.

- To avoid greed, use the **+?** and ***?** *minimal match* operators instead. They still match one or more repetitions, or zero or more repetitions, but they gobble up as little as is required for the pattern to match:

```
1 $html_line = 'Here is a <B>bold</B> thing and an <I>italic</I> thing';
2 print "HTML is: \"$html_line\"\n";
3 print "(Trying to strip HTML tags)\n";
4 $text_line = preg_replace('/<.+?>/', '', $html_line);
5 print "TEXT is: \"$text_line\"\n";
```

outputs:

```
HTML is: "Here is a <B>bold</B> thing and an <I>italic</I> thing"
(Trying to strip HTML tags)
TEXT is: "Here is a bold thing and an italic thing"
```

  - That's what we intended.

Summary of Pattern Modifiers

modifier	effect
/i	match case-**I**nsensitively
/s	include **\n** in the dot class (force string to be handled as a **S**ingle line)
/m	**^** and **$** can match before or after an ebmedded newline (force string to be handled as **M**ultiple lines)
/x	e**X**tended regex (can include comments and whitespace)

Tips

- Develop regular expressions one piece at a time, such as from the inside to the outside, and break them up if needed.
  - Use interpolation to make regular expressions easier to understand.
- If you use **$1**, **$2**, etc. remember that **all** regular parentheses in a regex memorize things.
  - This is particularly important to remember when interpolating regexes into one another.
- Remember that *, +, ? quantifiers are greedy.
- Remember that word boundaries and other anchors are **not** characters.
- Don't reinvent the wheel - if you need to use a regex for a URL, email address, U.S. Postal address, or anything else common but complicated, search the Web for a solution before you develop your own.
- Keep regexes as simple as possible.
- Keep regexes single-functional.
- Give clear names to regular expressions that are stored in variables.
  - Generally they should start with the **$rgx** prefix or something similar.
  - Give them names based on what they are supposed to do, not how they are supposed to do it.

When not to use Regular Expressions

- Regexes are very powerful, but they are slower and more difficult to use than some alternatives and are not always the best solution for a problem involving interpreting a string.
- Therefore, don't use regexes to:
  - Determine if a string is exactly the same as another string (use **==**/**===** or **strcmp**).
  - Determine if a string is exactly the same as another string, disregarding case (use **strcasecmp** or **==**/**===** with **strtolower**).
  - Determine if a string has a given prefix or suffix (use **strncmp** or **==**/**===** with **substr**).
  - Determine if a string contains a given substring (use **strpos** or **strstr**, or **stripos** or **stristr** for case-insensitive search).
  - Chop up a string by length (use **substr**)
  - Split a string with a constant delimiter (use **explode**).
- In general, if the problem can be solved with one or more string functions, it will probably be faster and easier to do so.
  - But don't use some tortured combination of multiple string functions when a simple regex will work.

Non-PCRE Regexes (POSIX)

- Because you may see POSIX regexes in others' code, you should be familiar with them.
- The major differences are:
  - Patterns are not enclosed in slashes.
  - POSIX does not support non-greedy matching.
  - POSIX uses a different syntax for character classes. For example, instead of **`\d`** for digits, POSIX uses **`[[:digit:]]`**.
  - The POSIX regex functions are **`ereg()`**, **`eregi()`**, **`ereg_replace()`**, **`eregi_replace()`**, **`split()`**, and **`spliti()`**.
  - POSIX regexes match the longest possible substring instead of the longest leftmost.
  - See PHP.net for more information.

Answers to Discussion Questions:

- What characters are in the class **`[^\W_]`**? The caret means this is an complement character class. The characters specified are non-word characters and underscores. Characters other than these (and therefore part of the class) are word characters except underscore; i.e. upper- and lowercase letters and digits.
- Members of the following classes:
  - **`[-^]`** matches a dash or caret
  - **`[^-^]`** matches any character except a dash or caret
  - **`[--^]`** matches a character between dash (ASCII #45) and caret (ASCII #94), inclusive
  - **`[^--^]`** matches any character except those between dash and caret
  - **`[^-]`** matches any character except a dash
  - **`[^^]`** matches any character except a caret
