Regular Expressions

What are regular expressions?

- A means of searching, matching, and replacing substrings within strings.
- Very powerful
- (Potentially) Very confusing
- Fundamental to Perl

Let's get started…

- Matching:
- if (STRING =~ m/PATTERN/) { ... }
  - Searches for PATTERN within STRING.
  - If found, return true. If not, return false. (in scalar context)
- Substituting/Replacing/Search-and-replace:
- SCALAR =~ s/PATTERN/REPLACEMENT/;
  - Searches for PATTERN within SCALAR.
  - If found, replace first instance of PATTERN with REPLACEMENT, and return true
  - If not, leave SCALAR as it was, and return false.

Matching

- *most* characters match themselves. They 'behave'
- **`if ($string =~ m/foo/){`**
  `print "$string contains 'foo'\n";`
  `}`
- some characters 'misbehave'. They affect how other characters are treated:
- **`\ | ( ) [ { ^ $ * + ? .`**
  - These are occasionally known as “The Dirty Dozen”
- To match any of these, precede them with a backslash:
- **`if ($str =~ m/\+/){`**
  `print "$str contains a plus sign\n";`
  `}`

Substituting

- s/PATTERN/REPLACEMENT/
- REPLACEMENT is a plain (double quoted) string, not a RegExp pattern. Same 'dirty dozen' rules apply to the PATTERN, but not the REPLACEMENT
  - No need to escape the dirty dozen in the replacement.
  - You must escape any character you would normally have to escape in a double-quoted string
    - $ @ \
  - Must also always escape the delimiter
- **`$greeting =~ s/hello/goodbye/;`**
- **`$sentence =~ s/\?/./;`**
- **`$path =~ s/\\/\//;`**

Leaning Toothpicks

- that last example looks pretty bad.
- **`s/\\/\//;`**
- This can sometimes get even worse:
  - **`s/\/foo\/bar\//\\foo\\bar\\/;`**
- This is known as "Leaning toothpick" syndrome.
- To overcome, instead of **`/`**, use any non-alphanumeric, non-whitespace delimiters, just as you can with **`q()`** and **`qq()`**
- **`s#/foo/bar/#\\foo\\bar\\#;`**

No more toothpicks

- You must always backslash the delimiter in both the pattern and the replacement
  - even if it's not one of the dirty dozen
- If you choose "parentheses-like" characters as the delimiters (ie, **( ), [ ], { }, < >**)
  - Use the corresponding right-side character to close the pattern
    - Need not choose same delimiters for pattern & replacement
    - **s(egg)<larva>;**
- If you do use **/**, you can omit the '**m**' from m//
  - '**s**' cannot be omitted from s///
- **if ($string =~ /found/) { … }**
- **$sub =~ /hi/bye/; #WRONG!!**

One more special delimiter

- If you choose **?** as the delimiter:
- After match is successful, Perl will not attempt to perform the match again until a **reset** command is issued, or the program terminates
- So, if **$foo =~ ?hello?** is in a loop, program will not search $foo for hello any time in the loop after it's been found once
- This applies only to matching, not substitution

Binding and 'Negative' Binding

- **=~** is the 'binding' operator. Usually read "matches" or "contains".
  - **$foo =~ /hello/**
  - "Dollar foo contains hello"
- **!~** is the negative binding operator. Read "Doesn't match" or "doesn't contain"
  - **$foo !~ /hello/**
  - "Dollar foo doesn't contain hello"
  - equivalent of **!($foo =~ /hello/)**

No binding

- If no binding operator, the match or substitution is performed on $_
- ```
  if (/foo/){
    print "$_ contains foo\n";
  }
  ```
- `s/Java/Perl/;`
  - replaces $_'s first instance of 'Java' with 'Perl'

Interpolation

- Variable interpolation is done inside the pattern match/replace, just as in a double-quoted string
  - UNLESS you choose single quotes for your delimiters
- ```
  $foo1 = 'hello';
  $foo2 = 'goodbye';
  $bar =~ s/$foo1/$foo2/;
  ```
  - same as `$bar =~ s/hello/goodbye/;`
- ```
  $a = 'hi';
  $b = 'bye';
  $c =~ s'$a'$b';
  ```
  - this does NOT interpolate. Will literally search for `'$a'` in `$c` and replace it with `'$b'`

Now we're ready

- Up to this point, no real 'regular expressions'
  - string matching only
  - could as easily been done with `index()` and `substr()`
- Now we get to the heart of the beast
- recall 12 'misbehaving' characters:
  - `\ | ( ) [ { ^ $ * + ? .`
- Each one has specific purpose inside of regular expressions.
  - some even have more than one
  ```

Alternation

- simply: "or"
- use the vertical bar: |
  - similar (logically) to || operator
- **`$string =~ /Paul|David/`**
  - search $string for "Paul" or for "David"
- If either "Paul" or "David" is found in $string, return true
- If neither is found, return false

Beginnings of strings

- ^ matches the beginning of a string
- **`$string = "Hi Bob.  How goes it?"`**
- **`$string2 = "Bob, how are you?\n";`**
- **`if ($string =~ /^Bob/){ … }`**
  - false
- **`if ($string2 =~ /^Bob/) { … }`**
  - true

Ends of Strings

- **`$`** matches the end of a string
- **`$s1 = 'Go home';`**
- **`$s2 = 'Your home awaits';`**
- **`if ($s1 =~ /home$/) { … }`**
  - true
- **`if ($s2 =~ /home$/) { … }`**
  - false
- $ will also match immediately before a terminating newline.
- **`if ("foo bar\n" =~ /bar$/) { … }`**
  - true

Some meta-characters

- For complete list, see pg 161 of Camel
  - or **`perldoc perlre`**
- **`\d`** any digit: **`0-9`**
  - **`\D`** any non-digit
- **`\w`** any 'word' character: **`a-z, A-Z, 0-9, _`**
  - **`\W`** any 'non-word' character
- **`\s`** any whitespace: **`" ", "\n", "\t"`**
  - **`\S`** any non-whitespace character
- **`\b`** a word boundary
  - Matches the boundary of a word, but doesn't match any actual characters
    - Just like **`^`** is "true" at beginning of string, **`$`** at end
  - between **`\w`** and **`\W`**, **`\W`** and **`\w`**, **`^`** and **`\w`**, or **`\w`** and **`$`**
  - **`\B`** true when not at a word boundary

The . Wildcard

- A single period matches "any character".
  - Except the newline
    - usually.
- **`/filename\..../`**
  - matches filename.txt, filename.doc, filename.exe, etc etc

Character classes

- Brackets group characters that have a certain property
  - Either a list of specific characters, or a range
- Basically a sequence of one-character alternations
- **`/[aeiou]/`**
  - search $_ for a vowel
- **`/[a-nA-N]/`**
  - search $_ for any characters in the first half of the alphabet, in either case
- **`/[0-9a-fA-F]/`**
  - search $_ for any 'hex' digit.
- Matches only one of the members of the class
  - **`/[aeiou]/`** ↵ search for one single vowel
  - **`/[aeiou][aeiou][aeiou]/`** ↵ search for three vowels

Character class catches

- use ^ at very beginning of your character class to negate it:
  - **`/[^aeiou]/`**
  - Search $_ for any non-vowel (includes consonants, numbers, whitespace, and punctuation)
- Character classes have their own “dirty” characters. Normal dirty dozen lose their specialness.
  - **`/[\w\s.]/`**
  - Search $_ for a word character, a whitespace, or a dot
- to search for ']' or '-', within a character class, make sure you backslash them
  - If you want to include ^ in a character class, either make sure it’s not the first character, or backslash it.

Clustering

- To group tokens together so that other operators will affect the whole group, use (?:)
- **`/prob|n|r|l|ate/`**
  - matches 'prob' or 'n' or 'r' or 'l' or 'ate'
- **`/pro(?:b|n|r|l)ate/`**
  - matches 'probate' or 'pronate' or 'prorate' or 'prolate‘

Quantifiers

- "How many" of previous characters to match
- **`*`** 0 or more
- **`+`** 1 or more
- **`?`** 0 or 1
- **`{N}`** exactly N times
- **`{N,}`** at least N times
- **`{N,M}`** between N and M times
  - This one particular token will only match upto M characters
  - Does NOT (by itself) prevent pattern from succeeding if more characters follow!

Quantifier examples

- **`/Age: \d+$/`**
  - Search for one or more digits following "Age: ", followed by end of string
    - matches "Age: 5", "Age: 20", and "Age: 100"
- **`/[a-z]+'?[a-z]*/`**
  - Search for a word that may contain an apostrophe
    - 1 or more letters, a possible apostrophe, 0 or more letters
    - matches **`hello won't you'll students'`**
- **`/\d{1,3}(?:,\d{3})*/`**
  - Search for a properly commafied number
    - one to three digits, followed by 0 or more groups of a comma and three digits
    - 534 1,431 10,234,053

Capturing

- Any parts of the match enclosed in parentheses without the **`?:`** are saved ('captured') in the numerical variables **`$1`**, **`$2`**, **`$3`**, etc
- Order is read left-to-right by *opening* parenthesis.
  - **`/(([a-z]+)=(\d+))/`**
  - $1 ← the whole match
    $2 ← the letters on the left of the equal sign
    $3 ← the digits on the right
- These variables are reset immediately after the next successful pattern match
  - regardless of whether or not that match uses ()

Regexp Pitfall #1 Greediness

- All quantifiers are 'greedy' by nature. They match as much as they possibly can.
  - Starting from left to right,
  - Will never prevent entire match from succeeding
- They can be made non-greedy by adding a **`?`** at the end of the quantifier
- **`$string = "hello there!"`**
- **`$string =~ /e(.*)e/;`**
  - $1 gets "llo ther";
- **`$string =~ /e(.*?)e/;`**
  - $1 gets "llo th";
- This applies to all quantifiers: **`*?`** prefers to match 0, **`+?`** prefers to match 1, **`{N,M}?`** prefers to match N, **`??`** prefers to match 0

Regexp Pitfall #2
Quantifier maximums

- A quantifier of the form {5} or {2,3} does not prevent more than the maximum from appearing in the string:
- **`"aaaaa" =~ /a{3}/`**
  - true!! The a{3} token simply matches the first three a's in the string
- **`"aaaaa" =~ /^a{3}$/`**
  - false. Can't find beginning of string, three a's, end of string.
- **`"10000" =~ /1\d{0,2}/`**
  - true!! The \d{0,2} token matches the first two digits after the 1 in the string (because it's greedy)
- **`"10000" =~ /1\d{0,2}(?:\D|$)/`**
  - false. Cannot find a 1, zero through two 0's, followed by either a non-digit or the end of string.

Regexp Pitfall #3
Contains vs Is

- The ^ and $ anchors are useful for determining whether a string "is" a pattern, or whether it merely "contains" that pattern.
- **`$name =~ /Bob|Robert/`**
  - Will match Bob, Robert, Bobby, BillyBob, Roberta, etc
- **`$name =~ /^(?:Bob|Robert)$/`**
  - Will match Bob or Robert only

Regexp Pitfall #4
Variables in your pattern

- Perl code with a pattern matching operation actually undergoes three levels of parsing.
  - The perl code is parsed to find that **`m/…/`** is a pattern match operation
  - The contents of the /.../ is parsed as a double quoted string, interpolating any variables
  - The resultant string is then parsed by the RegExp engine.
- Result being that if your pattern includes a variable with any of the dirty dozen, the RegExp Engine will see them, and treat them as special.

Regexp Pitfall #4 <continued>
Clean the Dirty Dozen

- **`my $item = 'This costs $5940.32';`**
  `my $price = '5.40';`
  `if ($item =~ /$price/) {`
  `  print "Item contains $price\n";`
  `}`
- We could attempt to use s/// to backslash every dirty dozen character.
  - But that would be time consuming and error prone.
- **`if ($item =~ /\Q$price\E/) { … }`**
  - \Q auto-backslashes all following dirty-dozen characters.
  - **`if ($item =~ /5\.40/) { ... }`**

Regexp Pitfall #5
Unexpected use of regexps

- Recall the **`split`** function from the first week
- Previously, we said it takes a string to use as a separator. This was not quite true.
- The first argument is actually a regular expression.
  - In our examples, we just used ' as the regexp delimiter
- **`my $str = "81alpha32beta0gamma";`**
  `my @greek = split /\d+/, $str;`
  - @greek ₤ ('', 'alpha', 'beta', 'gamma')

A bit more split

- split PATTERN, STRING, LIMIT
- if LIMIT is not given, removes all trailing empty fields
  - **`my @f = split /:/, 'a:b:c:d:e::::';`**
    - @f ₤ ('a', 'b', 'c', 'd', 'e');
- if LIMIT is given and positive, resulting list will have maximum of that many fields, remainder are left untouched and appended to final field
  - **`my @f = split /:/, 'a:b:c:d:e::::', 3;`**
    - @f ₤ ('a', 'b', 'c:d:e::::')
- if LIMIT is negative, retains all trailing empty fields
  - **`my @f = split/:/, 'a:b:c:d:e::::', -1;`**
    - @f ₤ ('a', 'b', 'c', 'd', 'e', '', '', '', '',)

TMI

- That's (more than) enough for now.
- next week, more information and configurations for regular expressions.
- Also, the transliteration operator.
  - doesn't use Reg Exps, but does use binding operators. Go figure.
- We'll look at some more built-in functions as well.