Regular expressions

Regular expressions can be the key to powerful, flexible, and efficient text processing.

Regular expressions themselves are almost like a mini programming language, allowing the programmer to describe and parse text

Regular expressions are made up of two types of characters

- The special characters are called metacharacters
- All other characters are called literals, or normal text characters

Regular Expressions

- A regular expression is a string of characters (some with special meaning) that is to be matched against a string
  - A string either matches or doesn't match a given regular expression
  - Unless otherwise specified we are looking for the regular expression to be anywhere within the search string
- The =~ operator is used to determine whether the target string contains the regular expression

```
if ($input =~ m/test/)
{
   print “Found regular expression string\n”;
}
```

- Any non-alphanumeric character can be used to delimit the regular expression
  - If a “paired” character is used the opening character is used first, then the closing character; otherwise use the same character to start and end the regular expression
- The initial “m” before the first slash is the match operator. If you use slashes for the delimiters for the regular expression you don't need the “m”.

=~ operator

The =~ operator is sometimes called the binding operator because it binds whatever is on its left side to a regular expression operator in its right

If the target variable is $_ then the variable can be omitted

The syntax of the matching operator is flexible. The forward slashes are not required; any non-alphanumeric, non-whitespace character can be the delimiter. If you use the slashes the m operator is not required

The regular expression may also be contained in variables

The !~ operator can be used to determine if the regular expression is not in the search string

Single character patterns

A single literal character matches itself

The dot (.) matches any single character except the newline character

All matches are case-sensitive (ignoring the case is an option to be discussed later)

- h.t matches
- hit
- hat
- hat
- Shot

but does not match

- Hunt
- Htdocs
- Hotel
- hoot

Character classes

A character class is represented by a list of character enclose between [ ]

- For the pattern to match one and only one of these characters must be present
- [abcde]
- [aeiouAEIOU]

The dash character (-) can be used to specify ranges

- [0-9] is the same as [0123456789]
- [a-z] is the same as [abcdefghijklmnopqrstuvwxyz]

The dash character only has this special meaning when it is used to specify a range

- /-[0-9]/

The caret character (^) has a special meaning if it is the first character within square brackets. It this case it represents all characters except those that follow

- [^0-9] matches any line that doesn’t contain a number anywhere in the line

Predefined character classes

- Because certain character classes occur so often there are several predefined character classes
  - \d
    - A digit; [0-9]
  - \D
    - Not a digit; [^0-9]
  - \w
    - A word character; [a-zA-Z0-9_]
  - \W
    - Not a word character; [^a-zA-Z0-9_]
  - \s
    - Whitespace character; [\r\t\n\f]
  - \S
    - Not a whitespace character; [^\r\t\n\f]

Multiple character matches

You can use a set of curly braces ({}) to match multiple occurrences of characters. The numbers inside the curly braces correspond to the character indicated at the left of the curly braces

/[0-9]{3}/

To match a range of occurrences

/[0-9]{2,5}/ matches 2-5 digits

To match as least a certain number of occurrences

/[0-9]{4,}/ matches 4 or more digits

Built in multipliers

- *
  - 0 or more
  - Same as {0,}
  - /[0-9]{0,}/ is the same as /[0-9]*/
- +
  - 1 or more
  - Same as {1,}
  - /[0-9]{1,}/ is the same as /[0-9]+/
- ?
  - 0 or 1
  - Same as {0,1}
  - /[0-9]{0,1}/ is the same as /[0-9]?/
- *, + and ? represent themselves if inside square brackets. They are multipliers if outside the square brackets.

Alternation

To determine if a string contain one of a set of alternatives use the (|) symbol

```
/for|while|do/  #does this line contain part of a loop?
```

Alteration has the lowest precedence of all the symbols so use it with care

Parenthesis can be used to separate the alternated portion from the rest of the pattern

Anchoring

Without an anchor a regular expression can appear anywhere within the search string. To specify a specific location of the search string we use an anchor

^ matches the beginning of the string

$matches the end of the string

If you use both the pattern includes the entire string

Word boundary anchors

\b is an anchor that matches at the beginning or ending or a “word”
\B is an anchor that matches everywhere \b doesn’t

Words are runs of \w characters (letters, digits and underscores)

Backreferences

- Parenthesis do more than just grouping letter together; they are also used to “remember” the text that matches the regular expression

- Placing parenthesis around a portion of the regular expressions causes Perl to store the text that matched that portion of the regular expression
  - The first matched text is stored in \1
  - The second matched text is stored in \2
  - …

- These stored matches can be accessed after the match using $1 for \1, $2 for \2, …
  - Match variables generally hold their values until the next successful regular expression match

Automatic match variables

- The part of the string that actually matched is automatically stored in $&
  - The use of parenthesis doesn’t effect the $& variable
- The part of the string before what matched is in $`
- Everything after what matched is in $’

Quantifier greediness

- Quantifiers are inherently greedy.
  - A * or + matches the maximum number of occurrences of a pattern that satisfy the regular expression

- The ? indicates that quantifiers should be non-greedy
  - This means that the quantifier will match the least number of occurrences of a pattern that still satisfies the regular expression

Substitution

s/// is the substitution operator.

- Between the first pair of slashes place the regular expression to find
- Between the second pair of slashes place the text to replace the “found” text

- The substitution operator uses the binding operator just like the matching operator

- The substitution operator can use delimiters other than /

- Unlike the matching operator the “s” in the substitution operator is required

Modifiers

- Modifiers can be placed after the last slash to alter the effect of the binding operator

- The g modifier
  - Normally substitution only occurs on the first occurrence of the match
  - If you want the substitution to occur on all occurrences of the match you must use the g modifier
- The i modifier
  - The i modifier makes the match case insensitive

- The e modifier
  - The e modifier allows the replacement string to be evaluated as a Perl expression and the result of the expression is substituted for the target text

Other modifiers

The m modifier

- The m modifier forces Perl to treat a string as if it has several lines within it. The $ and ^ symbols are enforced at the beginning and end of each line rather than for the entire string

The x modifier

- The x modifier allows you to place comments and whitespace characters in the regular expression without those characters being interpreted as part of the regular expression
  - Remember that regular expressions use it’s own language

Quotemeta

- The quotemeta function puts a backslash in front of any character that isn’t a letter, digit, or underscore
  - Special characters like parentheses, asterisks, periods and so forth will not be interpreted as regular expression metacharacters

```
$chomp ($pattern = <STDIN>);
$quoted = quotemeta $pattern;

print “matches\n” if ($input =~/$quoted/);
print “matches\n” if ($input =~ /\Q$pattern\E/);
```

  - Alternate method - must have the \Q and \E surrounding the text to be “quotemeta”-ed

tr modifier

The tr modifier translates a target set of characters with a replacement set of characters

- Shortcuts can be used in expressing the character sets
- If the target set is larger than the replacement set then the last character of the replacement set is used as a replacement for the excessive target characters
- The d modifier causes tr to delete a set of characters

```
tr/0-9//d;
```

- The s modifier removes all but one of the repeated adjacent characters

```
$_ = “mississippi”;
tr/a-z//s;
```

- The c modifier replaces only those characters in the complement of the target set with a single character in the replacement set

```
tr/a-zA-Z/\n/cs;
```