Regular Expressions

What are regular expressions?

- A means of searching, matching, and replacing substrings within strings.
- Very powerful
- (Potentially) Very confusing
- Fundamental to Perl
- Something C/C++ can’t even begin to accomplish correctly

Let’s get started…

- Matching:
  - STRING =~ m/PATTERN/;
  - Searches for PATTERN within STRING.
  - If found, return true. If not, return false. (in scalar context)
- Substituting/Replacing/Search-and-replace:
  - STRING =~ s/PATTERN/REPLACEMENT/;
  - Searches for PATTERN within STRING.
  - If found, replace PATTERN with REPLACEMENT, and return number of times matched
  - If not, leave STRING as it was, and return false.
Matching
• *most* characters match themselves. They ‘behave’ (according to our text)
  if ($string =~ m/foo/){
    print "$string contains ‘foo’\n”;
  }  
• some characters ‘misbehave’. They affect how other characters are treated:
  \ | ( ) [ { ^ $ * + ? .
  – To match any of these, precede them with a backslash:
  if ($string =~ m/\+/){
    print "$string contains a plus sign\n”;
  }

Substituting
• same rules apply
  $greeting =~ s/hello/goodbye/;
  $sentence =~ 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.
• Perl has a way around this: instead of /, use any non-alphanumeric, non-whitespace delimiters
  s#foo/bar/#\foo\bar#;
No more toothpicks

• Any non-alphanumeric, non-whitespace characters can be used as delimiters.
• If you choose brackets, braces, parens:
  – close each part
  – Can choose different delimiters for second part
  – $s(egg)<larva>;
• If you do use / (front slash), can omit the ‘m’ (but not the ‘s’) 
• $string =~ /found/;

One more special delimiter

• If you choose ? as the delimiter:
• After match is successful, will not attempt to perform the match again until reset command is issued, or 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 string is given to bind to (either via =~ or !~), the match or substitution is taken out on $_

```perl
if (/foo/) {
    print "$_ contains the string foo\n";
}
```

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

```perl
$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 string $c and replace with '$b'
```

Saving your matches

• parts of your matched substring can be automatically saved for you.
• Group the part you want to save in parentheses
• matches saved in $1, $2, $3, …

```perl
if ($string =~ /(Name)=(Paul)/) {
    print "First match = $1, Second match = $2\n";
}
```
• prints
  – “First match = Name, Second match = Paul”
Now we’re ready

- Up to this point, no real ‘regular expressions’
  - pattern matching only
- Now we get to the heart of the beast
- recall 12 ‘misbehaving’ characters:
  - \ | ( ) { ^ $ * + ? .
- Each one has specific meaning inside of regular expressions.
  - We’ve already seen 3…

Alternation

- simply: “or”
- use the vertical bar: |
  - similar (logically) to || operator
- $\text{string =~ }/ (Paul|Justin)/$
  - search $\text{string}$ for “Paul” or for “Justin”
  - return first one found in $1$
- /$\text{Name=(Robert(o|a))}$/
  - search $_$ for “Name=Roberto” or “Name=Roberta”;
  - return either Roberto or Roberta in $1$
  - (also returns either o or a in $2$)

Capturing and Clustering

- We’ve already seen examples of this, but let’s spell it out:
- Anything within the match enclosed in parentheses are returned (‘captured’) in the numerical variables $1$, $2$, $3$
- Order is read left-to-right by *Opening* parenthesis.
  - /$((\$foo)=($name))/$
  - $1$ → “$foo=$name”, $2$ → “$foo””, $3$ → “$name”;
Clustering

- Parentheses are also used to ‘cluster’ parts of the match together.
  - similar to the function of parens in mathematics

- `/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”

Clustering without Capturing

- For whatever reason, you might not want to ‘capture’ the matches, only cluster something together with parens.
- use `(?: )` instead of plain `( )`
- in previous example:
  - `/pro(?:b|n|r|l)ate/`
  - matches “probate” or “pronate” or “prorate” or “prolate”
  - this time, $1$ does not get value of b, n, r, or l

Beginnings and Ends of strings

- `^` matches the beginning of a string
- `$` matches the end of a string
  - `$string = “Hi, Bob. How’s it going?”`
  - `$string2 = “Bob, how are you?\n”;`
  - `$string =~ /^Bob/;`
    - returns false
  - `$string2 =~ /^Bob/;`
    - returns true
- `$` matches ends in the same way.
*Some* meta-characters

- For complete list, see pg 161 of Camel
- \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: space, \n, \t
  - \S → any non-whitespace character
- \b → a word boundary
  - this is “zero-length”. It’s simply “true” when at the boundary of a word, but doesn’t match any actual characters
  - \B → true when not at a word boundary

The . Wildcard

- A single period matches “any character”.
  - Except the new line
    - usually.
  - /filename\..../
    - matches filename.txt, filename.doc, filename.exe, etc etc

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} → at least N times, no more than M times
Greediness

- Quantifiers are ‘greedy’ by nature. They match as much as they possibly can.
- 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";

Character classes

- Use [ ] to match characters that have a certain property
  - Can be either a list of specific characters, or a range
    - /[aeiou]/
      - search \$_ for a vowel
    - /([a-nA-N])/ 
      - search \$_ for any characters in the 1st half of the alphabet, in either case
    - /([0-9a-fA-F])/ 
      - search \$_ for any ‘hex’ digit.

Character class catches

- use ^ at very beginning of your character class to negate it
  - /[^aeiou]/
    - Search \$_ for any non-vowel
      - Careful! This matches consonants, numbers, whitespace, and non-alpha-numeric too!
- . wildcard loses its specialness in a character class
  - /(/\w\s\.\)/
    - Search \$_ for a word character, a whitespace, or a dot
- to search for ’\’ or ‘\^’, make sure you backslash them in a character class
TMI

- That’s (more than) enough for now.
- go over the material, play with it.
- next week, more information and trivialities about regular expressions.
- Also, the transliteration operator.
  – doesn’t use Reg Exps, but does use binding operators. Go figure.