Regular Expressions

Sofia Robb

What is a regular expression?

A regular expression is a string template against which you can match a piece of text.

They are something like shell wildcard expressions, but **much** more powerful.

Examples of Regular Expressions

This bit of code loops through @ARGV files or STDIN. Finds all lines containing an EcoRI site, and bumps up a counter:

```
my $sites = 0;
 while (my $line = <>) {
   chomp $line;
   if ($line =~ /GAATTC/){
      print "Found an EcoRI site!\n";
      $sites++;
   }
 }
 print "$sites EcoRI sites total.\n"
```

Examples of Regular Expressions

This does the same thing, but counts one type of methylation site (Pu-C-X-G) instead:

```
my $sites = 0;
  while (my $line = <>) {
    chomp $line;
    if ($line =~ /[GA]C.?G/) {  # more conventional if block
       print "Found a methylation site!\n";
       $sites++;
    }
  }
  print "$sites methylation sites total.\n"
```

Specifying the String to Search

To specify which string variable to search, use the **=~** operator:

```
my $h = "Who's afraid of Virginia Woolf?";
print "I'm afraid!\n" if $h =~ /Woo?lf/;
```

Regular Expression Atoms

A regular expression is normally delimited by two slashes ("/"). Everything between the slashes is a pattern to match. A pattern is composed of one or more atoms:

```
1.Ordinary characters: a-z, A-Z, 0-9 and some punctuation. These
  match themselves.
2.The "." character, which matches everything except the newline.
3.A bracket list of characters, such as [AaGgCcTtNn], [A-F0-9], or
  [^A-Z] (the last means anything BUT A-Z).
4.Certain predefined character sets:
  \d The digits [0-9]
  \w A word character [A-Za-z_0-9]
  \s White space [ \t\n\r]
  \D A non-digit
  \W A non-word
  \S Non-whitespace
5.Anchors:
  ^ Matches the beginning of the string
  $ Matches the end of the string
  \b Matches a word boundary (between a \w and a \W)
```

Regular Expression Atoms

Examples

- /g..t/ matches "gaat", "goat", and "gotta get a goat" (twice)
- /g[gatc][gatc]t/ matches "gaat", "gttt", "gatt", and "gotta get an agatt" (once)
- /\d\d\d-\d\d\d\d/ matches 376-8380, and 5128-8181, but not 055-98-2818.
- /^\d\d\d-\d\d\d\d/ matches 376-8380 and 376-83801, but not 5128-8181.
- /^\d\d\d-\d\d\d\d$/ only matches telephone numbers.
- /\bcat/ matches "cat", "catsup" and "more catsup please" but not "scat".
- /\bcat\b/ only text containing the word "cat".

Quantifiers

By default, an atom matches once. This can be modified by following the atom with a quantifier:

?	atom matches zero or exactly once
*	atom matches zero or more times
+	atom matches one or more times
{3}	atom matches exactly three times
{2,4}	atom matches between two and four times, inclusive
{4,}	atom matches at least four times

Examples:

- /goa?t/ matches "goat" and "got". Also any text that contains these words.
- /g.+t/ matches "goat", "goot", and "grant", among others.
- /g.*t/ matches "gt", "goat", "goot", and "grant", among others.
- /^\d{3}-\d{4}$/ matches US telephone numbers (no extra text allowed.

Alternatives and Grouping

A set of alternative patterns can be specified with the | symbol:

```
/wolf|sheep/;   # matches "wolf" or "sheep"

/big bad (wolf|sheep)/;   # matches "big bad wolf" or "big bad sheep"
```

You can combine parenthesis and quantifiers to quantify entire subpatterns:

```
/Who's afraid of the big (bad )?wolf\?/;

# matches "Who's afraid of the big bad wolf?" and
#         "Who's afraid of the big wolf?"
```

This also shows how to literally match the special characters -- put a backslash (\) in front of them. There's also an equivalent "not match" operator !~, which reverses the sense of the match:

```
$h = "Who's afraid of Virginia Woolf?";
print "I'm not afraid!\n" if $h !~ /Woo?lf/;
```

Matching with a Variable Pattern

You can use a scalar variable for all or part of a regular expression. For example:

```
$pattern = '/usr/local';
print "matches" if $file =~ /^$pattern/;
```

See the o flag for important information about using variables inside patterns.

Subpatterns

You can extract and manipulate subpatterns in regular expressions.

To designate a subpattern, surround its part of the pattern with parenthesis (same as with the grouping operator). This example has just one subpattern, (.+) :

```
/Who's afraid of the big bad w(.+)f/
```

Matching Subpatterns

Once a subpattern matches, you can refer to it later within the same regular expression. The first subpattern becomes \1, the second \2, the third \3, and so on.

```
while (my $line = <>) {
  chomp $line;
  print "I'm scared!\n" if $line =~ /Who's afraid of the big bad w(.)\1f/
}
```

This loop will print "I'm scared!" for the following matching lines:

- Who's afraid of the big bad woof
- Who's afraid of the big bad weef
- Who's afraid of the big bad waaf

but not

- Who's afraid of the big bad wolf
- Who's afraid of the big bad wife

In a similar vein,

```
/\b(\w+)s love \1 food\b/
```

will match "dogs love dog food", but not "dogs love monkey food".

Using Subpatterns Outside the Regular Expression Match

Outside the regular expression match statement, the matched subpatterns (if any) can be found the variables **$1, $2, $3,** and so forth.

Example. Extract 50 base pairs upstream and 25 base pairs downstream of the TATTAT consensus transcription start site:

```
while (my $line = <>) {
  chomp $line;
  next unless $line =~ /(.{50})TATTAT(.{25})/;
  my $upstream = $1;
  my $downstream = $2;
}
```

Extracting Subpatterns Using Arrays

If you assign a regular expression match to an **array**, it will return a list of all the subpatterns that matched. Alternative implementation of previous example:

```
while (my $line = <>) {
    chomp $line;
    my ($upstream,$downstream) = $line =~ /(.{50})TATTAT(.{25})/;
  }
```

If the regular expression doesn't match at all, then it returns an empty list. Since an empty list is FALSE, you can use it in a logical test:

```
while (my $line = <>) {
  chomp $line;
  next unless my ($upstream,$downstream) = $line =~ /(.{50})TATTAT(.{25})/;
  print "upstream = $upstream\n";
  print "downstream = $downstream\n";
}
```

Grouping without Making Subpatterns

Because parentheses are used both for grouping (a|ab|c) and for matching subpatterns, you may match subpatterns that don't want to. To avoid this, group with (?:pattern):

```
/big bad (?:wolf|sheep)/;
```

# matches "big bad wolf" or "big bad sheep",
# but doesn't extract a subpattern.

Subpatterns and Greediness

By default, regular expressions are "greedy". They try to match as much as they can. For example:

```
$h = 'The fox ate my box of doughnuts';
$h =~ /(f.+x)/;
$subpattern = $1;
```

Because of the greediness of the match, **$subpattern** will contain "fox ate my box" rather than just "fox".

To match the minimum number of times, put a **?** after the qualifier, like this:

```
$h = 'The fox ate my box of doughnuts';
$h =~ /(f.+?x)/;
$subpattern = $1;
```

Now **$subpattern** will contain "fox". This is called *lazy* matching.
Lazy matching works with any quantifier, such as +?, *?, ?? and {2,50}?.

String Substitution

String substitution allows you to replace a pattern or character range with another one using the **s///** and **tr///** functions.

The s/// Function

s/// has two parts: the regular expression and the string to replace it with: s/*expression*/*replacement*/.

```
$h = "Who's afraid of the big bad wolf?";
$i = "He had a wife.";

$h =~ s/w.+f/goat/;  # yields "Who's afraid of the big bad
goat?"
$i =~ s/w.+f/goat/;  # yields "He had a goate."
```

If you extract pattern matches, you can use them in the replacement part of the substitution:

```
$h = "Who's afraid of the big bad wolf?";

$h =~ s/(\w+) (\w+) wolf/$2 $1 wolf/;
# yields "Who's afraid of the bad big wolf?"
```

Using a Variable in the Substitution Part

Yes you can:

```
$animal = 'hyena';
$h =~ s/(\w+) (\w+) wolf/$2 $1 $animal/;
# yields "Who's afraid of the bad big hyena?"
```

Translating Character Ranges

The **tr///** function allows you to translate one set of characters into another. Specify the source set in the first part of the function, and the destination set in the second part:

```
$h = "Who's afraid of the big bad wolf?";
$h =~ tr/ao/AO/; # yields "WhO's AfrAid Of the big bAd
wOlf?";
```

tr/// returns the number of characters transformed, which is sometimes handy for counting the number of a particular character without actually changing the string.

This example counts N's in a series of DNA sequences:

Code:

```
while (my $line = <>) {
    chomp $line;   # assume one sequence per line
    my $count = $line =~ tr/Nn/Nn/;
    print "Sequence $line contains $count Ns\n";
  }
```

Input:

```
AGCTGGGAAAGT
AGCNGNNAAAGT
TAGCNGTTAAAT
GAATCAGCTGGG
. . .
```

Output:

```
(~) 50% count_Ns.pl
sequence_list.txt
Sequence 1 contains 0 Ns
Sequence 2 contains 3 Ns
Sequence 3 contains 1 Ns
Sequence 4 contains 0 Ns
...
```

Common Regular Expression Modifiers

Regular expression matches and substitutions have a whole set of options which you can use by appending one or more modifiers to the end of the operation.

i
Case insensitive match.

g
Global match.

Case insensitive Matches

```
my $string = 'Big Bad WOLF!';
print "There's a wolf in the closet!" if $string =~ /wolf/i;
#case insensitive match
```

Global Matches

Adding the g modifier to the pattern causes the match to be global. Called in a scalar context (such as an if or while statement), it will match as many times as it can.

This will match all codons in a DNA sequence, printing them out on separate lines:

Code:

```
my $sequence = 'GTTGCCTGAAATGGCGGAACCTTGAA';
while ( $sequence =~ /(.{3})/g ) {
  print $1,"\n";
}
```

Output:

```
GTT
GCC
TGA
AAT
GGC
GGA
ACC
TTG
```

The pos() function retrieves the position where the next attempt begins

```
$position_of_next_attempt = pos($sequence)
```

If you perform a global match in a **list** context (e.g. assign its result to an array), then you get a list of all the subpatterns that matched from left to right. This code fragment gets arrays of codons in three reading frames:

```
@frame1 = $sequence =~ /(.{3})/g;
@frame2 = substr($sequence,1) =~ /(.{3})/g;
@frame3 = substr($sequence,2) =~ /(.{3})/g;
```

Additional regular expression modifiers

o
Only compile variable patterns once.

m
Treat string as multiple lines. ^ and $ will match at start and end of internal lines, as well as at beginning and end of whole string. Use \A and \Z to match beginning and end of whole string when this is turned on.

s
Treat string as a single line. "." will match any character at all, including newline.

x
Allow extra whitespace and comments in pattern.