Regular Expressions

Regular expressions is a language you can use within perl to identify patterns in text.

A regular expression is a string template, or pattern, against which you can match a piece of text. They are something like shell wildcard expressions, but **much** more powerful.

Preview of Regular Expressions

A brief introduction of what a regular expression can do.

This bit of code loops through each line of a file. Finds all lines containing an EcoRI site, and bumps up a counter:

Code:

```
#!/usr/bin/perl -w
#file: EcoRI1.pl

use strict;

my $filename = "example.fasta";
open (FASTA , '<' , $filename ) or print "$filename does not exist\n";
my $sites;

while (my $line = <FASTA> ) {
  chomp $line;

  if ($line =~ /GAATTC/){
    print "Found an EcoRI site!\n";
    $sites++;
  }
}

if ($sites){
  print "$sites EcoRI sites total\n";
}else{
  print "No EcoRI sites were found\n";
}

#note: if $sites is declared inside while loop you would not be able to
#print it outside the loop
```

Output:

```
~]$ ./EcoRI1.pl
Found an EcoRI site!
Found an EcoRI site!
.
.
.
Found an EcoRI site!
Found an EcoRI site!
34 EcoRI sites total
```

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

(Pu-C-X-G) Methyation Site:

```
/[GA]C.?G/
```

> G or an A

> ```
> [GA]
> ```
>
> followed by a C
>
> ```
> C
> ```
>
> followed by one of anything, but could be nothing
>
> ```
> .?
> ```
>
> followed by a G
>
> ```
> G
> ```

Code:

```
#file:methy.pl
while (my $line = <FASTA>) {
    chomp $line;

    if ($line =~ /[GA]C.?G/){
          $sites++;
    }
}
if ($sites){
    print "$sites Methylation Sites total\n";
}else{
    print "No Methylation Sites were found\n";
}
```

Output:

```
~]$ ./methy.pl
723 Methylation Sites total
```

Atoms

Atoms are the content, or the bits and peices of a regular expression

A regular expression is normally delimited by two slashes ("/"). Everything between the slashes is a pattern to match. Patterns can be made up of the following **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)

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

Quantifiers quantify how many atoms are to be found.

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).

Quick Exercises:

> 1. Design a pattern to recognize an email address.
> 2. Design a pattern to recognize the id portion of a sequence in a FASTA file
>
> \>SEQ1
> ATGCTGCGCGTGCATGATGCT
> \>SEQ2
> CGCGTGCATGATGCTGCGCGT

Binding operator

Specifying the String to Match

The Binding operator (=~) is used to "bind" the string to be searched and the pattern.

```
$h = "Who's afraid of Virginia Woolf?";
$h =~ /Woo?lf/;
```

The one line version of the 'if statement' can be combined with a regular expression:

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

There's also an equivalent "not match" operator **!~**, which reverses the sense of the match:

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

Quick Exercises:

> 1. What happens if you do not use the `~` symbol in your pattern match?
> 2. Create a script with a regular expression within an if-statement.
> 3. Design the regular express to match an entire sentence, up to the ending period in the provided string.
>
>   my $str = "This is a paragraph. A Paragraph is usually made up of more than one sentence.";
>
> 4. Modify your regular expression to take '.' , '?', and '!; into account as ending punctuation.

Variable Patterns

Matching with a pattern stored in a variable

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

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

Quick Exercises:

> 1. Save your pattern from the last exercise in a variable. Replace the pattern portion of the regular expression with the varible.
> 2. Does your match still work?

Alternatives and Grouping

Sometimes you want to match "this" or "that" with your pattern

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

This pattern matches "wolf" or "sheep"

```
/wolf|sheep/;
```

Use parenthesis to group the alternative patterns:

This pattern matches "big bad wolf" or "big bad sheep"

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

Quick Exercises

> 1. Create a regular expression that will match a string with this pattern:
>   - ATG followed by a C or a T.
> 2. Test your regular expression with these two strings:
>   - GCTGATGCGTTA
>   - GCTATGGCT

Subpatterns

Sometimes a subset of the pattern is important and you would like to save it for later use.

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/
```

You can combine parenthesis and quantifiers to quantify entire subpatterns:

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

This matches "Who's afraid of the big bad wolf?" as well as "Who's afraid of the big wolf?"

This also shows how to literally match the special characters, **put a backslash `\` in front of them**.

Quick Exercises

> 1. A FASTA has the following format:
>   >ID Optional Descrption

> SEQUENCE
> SEQUENCE
> SEQUENCE
>
> 2. Create a regular expression that captures the sequence ID as a subpattern.

Using Subpatterns Inside the Regular Expression Match

This is helpful when you want to find a subpattern and then match the contents again

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

```
while ($line = <FILE>) {
  chomp $line;
  print "I'm scared!\n" if /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

Using the captured subpattern in code that follows the regular expression.

Outside the regular expression match statement, the matched subpatterns (if any) can be found in 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 = <FILE>) {
  chomp $line;
  next unless $line =~ /(.{50})TATTAT(.{25})/;
  my $upstream = $1;
  my $downstream = $2;
}
```

Quick Exercises

> 1. A FASTA header has the following format:
>   >ID Optional Descrption
>   SEQUENCE
>   SEQUENCE
>   SEQUENCE
> 2. Create a regular expression that captures the sequence ID as a subpattern.
> 3. If an ID is found, print it.

Extracting Subpatterns Using Arrays

Storing all the captured subpatterns in an array

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

Notice the `=` to the left of the string being searched. The subpatterns are returned and are assigned to the list of variables. This is another way to write it

```
my @nts = $line =~ /(.{50})TATTAT(.{25})/;
```

@nts will contain two values:

```
$nts[0] will have the upstream sequence
$nts[1] will have the downstream sequence
```

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

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:

Remember, if using the saved subpattern *outside* the pattern, or expression, use **$1, $2**, etc.

```
$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?"
```

Quick Exercises

> 1. Create a regular expression with the s/// function to find all ATG and replace with '-M-'.
>
> GCAGAGGTGATGGACTCCG
> TAATGGCCAAATGACACGT

Using a Variable in the Substitution Part

Yes you can:

```
$h = "Who's afraid of the big bad wolf?";
$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 = <FILE>) {
    chomp $line;   # assume one sequence per line
    my $count = $line =~ tr/Nn/Nn/;
    print "Sequence $line contains $count Ns\n";
  }
```

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
...
```

Regular Expression Options

Regular expression matches and substitutions have a whole set of options which you can toggle on by appending one or more of the **i, m, s, g, e** or **x** modifiers to the end of the operation.

See Programming Perl Page 153 for more information. Some example:

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

i

Case insensitive match.

g

Global match (see below).

e

Evalute right side of s/// as an expression.

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.

o

Defining that a variable used as a pattern will never change, so perl will not attempt to interpolate the variable.

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:

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

Output:

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

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;
```

The position of the most recent match can be determined by using the **pos** function. The pos function returns the position where the next attempt begins. Remember that pos will return in 0-base notation, the first postion is 0 not 1. Code:

```
#file:pos.pl
my $seq = "XXGGATCCXX";

if ( $seq =~ /(GGATCC)/gi ){
  my $pos = pos($seq);
  print "Our Sequence: $seq\n";
  print '$pos = ', "1st postion after the match: $pos\n";
  print '$pos - length($1) = 1st postion of the match: ',($pos-length($1)),"\n";
  print '($pos - length($1))-1 = 1st postion before the the match:',
    ($pos-length($1)-1),"\n";
}
```

Output:

```
~]$ ./pos.pl
Our Sequence: XXGGATCCXX
$pos = 1st postion after the match: 8
$pos - length(GGATCC) = 1st postion of the match: 2
($pos - length(GGATCC))-1 = 1st postion before the the match: 1
```

Quick Exercises

> 1. Create a regular expression that matches ATG.
> 2. Print "Found ATG" when an ATG is found.
> 3. Did you use a while loop?
>   - If not, try it with a while loop. How many ATGs do you find?
>   - If so, try it without a while loop. How many ATGs do you find?
>
> 4. Did you use the 'g' modifier? What happens in your while loop with it and without it?
> 5. Print the position of each of each of the ATGs.
>
> SEQUENCE:
> TAATGGCCAAATGACACGTGCAGAGGTGATGGACTCCG