Regular Expressions

Aims

- Understand what a regular expression is
- Use match, substitute and transliterate
- Know about using i,m,g flags
- Understand some metacharacters
- Capture
- Greedy and non-greedy match
- Count with transliterate

Regular Expressions

- Perl's speciality
- Heavily optimised to be as efficient as possible
- Reliable
- Fully featured (as yet, I haven't found anything it is missing)

Regular Expressions

- 3 types of regular expressions
- Match
  - Does the variable contain
- Substitute
  - If match, then replace with something
- Transliterate
  - Swap characters

Match

- Problem: I want to see it a sequence contains the codon for Methionine
- So far, we could look to see if there is an index returned from

  say index $seq, 'ATG'
- However, that isn't very obvious, and what if the sequence is lower case?

  bin/01-match.pl

Match

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
say $sequence;
if ( $sequence =~ m/ATG/i ) {
  say 'Sequence contains Methionine';
}
```

- This tests to see if it can find ATG at all in the sequence, and lets us know if true

Match - Syntax

- Lets break down the match

  if ( ) {

  }
- This is the block form of if, we'll look at that more in the next section. For now, accept that it is just allowing us to 'visualise' the match

Match - Syntax

$sequence =~ m/ATG/i

- This is the syntax for a match

  $sequence
- Our variable to act upon

  =~
- The 'true' match operator. The return value of the expression is a boolean, so this sets match as boolean true, fail as boolean false

  (No match to be true would use !~)

Match - Syntax

$sequence =~ m/ATG/i

m/ATG/i

- m tells perl we want to do a match regular expression operation
- /xxx/ delimits the thing we want to match
- i is a flag options

Match

- Obviously, we can just swap ATG for whatever we want to search for

  AAA

  Hello World

  01223834244

  \n (\t)
- There are also a set of metacharacters we can use, to match out of a set.

Match - Metacharacters

```
if ( $sequence =~ m/\d+/ ) {
  say 'Sequence contain a number';
}
```

- \d matches a digit
- + matches 1 or more of the preceeding character or metacharacter

```
if ( $sequence !~ m/\d+/ ) {
  say 'Sequence does not contain a number';
}
```

Match - Metacharacters

```
if ( $sequence =~ m/\w+/ ) {
  say 'Sequence contains word characters';
}
```

- \w matches a word character
  - Alpha
  - Underscore
  - Digits

Match - Metacharacters

```
my $lyrics = 'I see a little silhoutte of a man';
if ( $lyrics =~ m/\s/ ) {
  say 'Lyrics contain whitespace';
}
```

- \s matches a whitespace character
  - Space
  - Line break
  - Tab

Match - Metacharacters

- These are the most common metacharacters you will see along with

  \A – match the start of the string

  \z – match the end of the string

  . - match any character

Match - Capture

- As well as matching, you can capture part or all of what you are trying to match, by placing brackets around the bit to capture

```
my ( $silhoutte ) = $lyrics =~ m/\s(silhoutte)\s/i;
say $silhoutte;
```

- In this case, if we match the word silhoutte, surrounded by whitespace, then put that word into the variable

Match - Capture

- How about capturing the first word:

```
my ( $first_word ) = $lyrics =~ m/(\w+)\s+/;
say $first_word;
```

- Or the last:

```
my ( $last_word ) = $lyrics =~ m/\s(\w+)\s*\z/m;
say $last_word;
```

- * means zero or more

Match - Greedy

- * means zero or more, + means 1 or more
- However, they like to grab more than less

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
my ( $greedy_match ) = $sequence =~ /(\w+ACC)/;
say $greedy_match;
```

- How do we stop this?

Match – Non-Greedy

- Use the ? metacharacter

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
my ( $non_greedy_match ) = $sequence =~ /(\w+?
ACC)/;
say $non_greedy_match;
```

- ? tells the regex to stop matching after the first it can match

Match – User defined

- Our user wants to see if their favourite codon is present

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
my $user_wants_to_find_this_sequence = 'ttt';
```

- How?

Match – User defined

- We can put their variable in the match

```
if ( $sequence =~ /
$user_wants_to_find_this_sequence/i ) {
  say "User sequence
$user_wants_to_find_this_sequence found.";
}
```

- The regular expression is allowed to contain variables.
- Much more useful than forcing only to match developer designated expression

Match – i and m

- What do the i and m mean?
- They are flags which tell the regex to work in particular ways (note, these are here for completeness, not because you need to learn them all)
- i – this tells the regex to ignore case

Match – i and m

- m - makes perl match newlines/end of strings the same as other languages
- In most languages:
  - ^ matches the start of a line, but in perl the start of the string
  - $ matches the end of a line, but in perl the end of a string
- This makes it a little confusing, so m changes their meaning
- Example at end of bin/01-match.pl

Substitute

- Problem: We want to replace a codon with another in the sequence.

bin/02-substitute.pl

```
my $sequence =
'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG
CGCCGTCACGCTCGCAGTC';
unless ( $sequence =~ m/cat/i ) {
  say 'we do not have a cat';
}
```

Substitute

- Lets replace cac with cat

```
$sequence =~ s/cac/cat/i;
if ( $sequence =~ m/cat/i ) {
  say 'we now have a cat';
}
```

- How many 'cat's?

```
say $sequence;
```

- 1, but there were 3 'cac's

Substitute - Syntax

- Lets look at the syntax, comparing with match

```
$sequence =~ s/cac/cat/i;
$sequence =~ m/cat/i
```

- =~ That's the same. We are informing perl that we wish to perform a regex operation on $sequence

Substitute - Syntax

```
$sequence =~ s/cac/cat/i;
$sequence =~ m/cat/i
```

- s instead of m
  - We tell perl that we want to do a substitution type of regex
- i – as before we want to do case insensitive matching

Substitute - Syntax

```
$sequence =~ s/cac/cat/i;
$sequence =~ m/cat/i
```

- /*match*/*replace*/ instead of /*match*/
  - match is identical, and you can use all the same matching tech
  - But, perl needs to know what to replace the match with, so we put that in a second 'column'

Substitute 'g'lobally

- How many 'cat's?

  say $sequence;
- 1, but there were 3 'cac's
- It only substituted the first one it found!
- How do we replace all of them?

  $sequence =~ s/cac/cat/gi;

  say $sequence;
- The g flag tells perl to keep matching and substituting until the end

Substitute – User defined

- You can have variables as the match, and the replacement

  my $favourite_codon = 'aaa'; # it's all the first letter of my name;

  my $least_favourite_codon = 'cat'; # I don't actually like cats

  $sequence =~ s/$least_favourite_codon/ $favourite_codon/gi;

  say $sequence;

Substitute – How many Substitutions made?

- The return value is the number of substitutions made, or undef if 0

  say $sequence =~ s/aaa/cac/i; # 1 (we didn't specify global)

  say $sequence =~ s/aaa/cac/gi; # 2 (the remaining ones)

  say $sequence =~ s/cat/aaa/gi; # undef, we had already cleared away all the cats
- This means you can use it as a boolean

Substitute – Don't be greedy!

- Because the match is exactly the same as with a match regex, + and * are still greedy

  my $clone = $sequence;

  *$sequence =~ s/g.*cac/aaa/i;*

  say $sequence;

  $clone =~ s/g.?cac/aaa/i;*

  say $clone;
- You would not have been able to do multiple substitutions, as the first match ate everything between the first g and last cac

Match and Substitute

- There are other metacharacters, and ways of constructing really powerful short regexes with other flags
- However, everything you have seen here will form the basis of any match you do, and you should be able to construct virtually any match you want
  - 90% of my regexes use nothing more than shown here

Transliterate

- The 3rd type of regex is transliterate. It stands a little aside from match and substitute, as it is more literal (i.e. less metacharacter and flag driven)
- Problem: I want to know the reverse strand of my DNA sequence
- You can't use substitute easily
  - s/T/A/gi
  - s/A/T/gi

Transliterate

bin/03-transliterate.pl

my $sequence = 'CCGGATCACTATGACCTGCTTTCGCACCTGCTCG CGCCGTCACGCTCGCAGTC';

my $reverse_strand = $sequence;

$reverse_strand =~ tr/ACGT/TGCA/;

say $sequence;

say $reverse_strand;

- In-place substitution of the dna letters

Transliterate - Syntax

- Let's look at the syntax

  reverse_strand =~ tr/ACGT/TGCA/;
- It's quite similar to substitute, but there are important differences
- =~ We now know what this means
- tr – tells perl to perform a transliterate operation on *$reverse_strand*

Transliterate - Syntax

reverse_strand =~ tr/ACGT/TGCA/;

- /ACGT/TGCA/

  This is the important bit. Whereas before we had match 'patterns' and substitution strings, we don't here.
- As it reads *$reverse_strand*, each time it finds a letter in the first part, it replaces with the equivalent letter in the second
  - Find an A, replace with T
  - Find a G, replace with a C

Transliterate – Return Value

- As with substitute telling you the number of substitutions made, transliterate returns the number of transliterations made (or undef)

```
$reverse_strand = $sequence;
say $reverse_strand =~ tr/ACGT/TGCA/;
my $bees = 'BBBBBBBBBBBBB';
say $bees =~ tr/ACGT/TGCA/;
```

Transliterate – Coding a Message

- A bit of fun, lets use this to apply a simple cypher

```
my $original_message = 'The quick brown fox jumped over the lazy dog.';
say $original_message;
my $coded_message = $original_message;
$coded_message =~ tr/a-zA-Z ./N-ZA-Mn-za-m. /;
say $coded_message;
```

- In regexes, we can shorten sequences with a dash, they get expanded by the parser.

Transliterate – Counting Characters

- You can replace a character with itself, to get a count of that character in the string, without changing the string

```
say $sequence;
say "The number of A's in above = " . ( $sequence =~ tr/Aa/Aa/ );
say $sequence;
```

- It is unchanged, but we know how many A's are present

Transliterate – GC percentage

- With a bit of playing, you could work out the percentage GC content

```
say "percentage GC = " .
        ( ( $sequence =~ tr/CGcg/CGcg/ ) * 100 /
         (  $sequence =~ tr/ACGTacgt/ACGTacgt/ ) );
say $sequence;
```

- Again, unchanged, but we know what the GC percentage is

Quick play with regexes

- Lets just spend a few minutes playing with regexes.

  Produce a regex which will capture a name and a phone number into variables from the following:

  Andy Brown: 01234 567890

- Bonus points for
  - Using metacharacters
  - name parts and phone parts