7) Regular Expressions

Matt Webster
IMBIM, BMC
matthew.webster@imbim.uu.se

Regular expressions

- Very powerful and useful part of perl
- similar to find and replace in word, or grep in unix
- used for pattern matching

```
$_="ACGCGCTGCGTAGACATGGTAGACGAT";
if (/ATG/) {
   print "I found a start codon\n";
}
```

- a simple regular expression is something inside `//`
- you can use all the normal backlash codes e.g.

```
/TG\tchr10\t128182\n/
```

a snp and it's position?

Metacharacters

characters with a special meaning

`.`
a dot is the wildcard (matches one of everything except `\n`)

`\`
backslash changes any metacharacter into a normal character

`/./`
matches anything

`/\./`
matches a dot

`/\\/`
matches a backlash

`/\t/`
matches tab

`/\\t/`
matches `\t`

`/.\t./`
matches two characters with a tab in the middle

More metacharacters

`*`

a star matches previous characters zero or more time

`+`

a plus matches previous characters one or more times

`?`

question mark matches previous character zero or one time

`/Maja*/`

matches `Maj` or `Maja` or `Majaaaaa`

`/Maj+a/`

matches `Maja` or `Majjjjjja`

`/M?aja/`

matches `Maja` or `aja`

More metacharacters

`( )`
parentheses can be used to group parts of the pattern

`/(Maja)*/`
matches nothing or `Maja` or `MajaMaja` or `MajaMajaMajaMajaMaja`

`/(Maj)+a/`
matches `Maja` or `MajMajMajMaja`

`/M*(aja)+/`
matches `aja` or `MMMaja` or `MMMMajaajaajaajaaja`

More metacharacters

`\1 \2`

backslash number is used to refer back to a parentheses that matches

```
$_ = "Emma Ivansson";
```

`/(.)\1/`

matches two characters next to each other `'mm'`

`/(E)mm(a).+\2/`

matches `'Emma Iva'`

`/(E)mm(a).+\1/`

doesn't match

character classes

`[ ]`
a list of possible characters goes inside square brackets

`/[a-z]/`
matches one lowercase letter (not åäö)

`/[a-zA-Z]/`
matches one upper or lowercase letter (not ÅÖÄåöä)

`/[ATGC]+/`
matches a DNA sequence of any length

`</[^A-Z]+/`
matches anything except capital letters (^ negates)

`/[A\-Z]+/`
matches strings containing `'A'`, `'-'` and `'Z'` (backslash stops dash being a special character)
e.g. `AAA---A---A----Z---Z----A----A-Z-AAAZAZ`

character classes

`\d`
a digit

`\s`
a space

`\r \n  \R`
linebreaks (\R means any line break – new feature)

`\w` (or `[a-zA-Z0-9_]`)
word (although includes numbers)

`/\d+\s\w+/`
matches `'123134 BICF2837482'` for example

`/\w+\s\w+\s\w+\n/`
matches three words followed by linebreak

negating character classes

`\D` `[^\d]`
not a digit

`\S` `[^\s]`
not a space

`\W` `[^\w]`
not a word

matches with `m//`

`/ /`
usually used for pattern matching

`m/ /`
is the same thing

`m% %`
you can change the separators

`m%http://%`
is easier to read than

`/http:\/\//`
lots of backslashes can be confusing (but still correct)

modifiers

`/ /a`
specify ASCII text encoding

`/ /i`
case insensitive

`/ /s`
includes newlines into the dot character class

`/ /x`
ignores spaces within the pattern

`/maja/i`
matches `'Maja' 'maja' 'maJA' 'mAjA'`

`/Maja.+Emma/s`
matches `'MajaEmma' 'Maja Elin Emma' 'Maja\nElin\nEmma'`

`/Maja.*Emma/is`
matches `'maJaEMMa' 'mAJa elin emMA' 'MAJA\nELIN\nEMMA'`
modifiers can be combined

`/Maja Emma/x`
matches `'MajaEmma'` but not `'Maja Emma'`

ASCII and Unicode

- ASCII

ASCII is an acronym for "american standard code for information interchange". used for storing/ displaying text, simple formatting and a few other control characters.
characters numbered 0-255

- Unicode

ASCII has been superseded by unicode, a double byte character system designed to store and display a much wider range of letters.(65,536) the extra include foreign languages and mathematical/scientific symbols, plus space for future expansion.

`/\w+/a`
matches ASCII word characters

`/\w+/u`
matches Unicode word characters

`/\w+/l`
matches locale word characters (different versions of ASCII for different countries)

You need to consider this when matching special characters like Å € ç Ω œ®
but you probably never need to do that for biology!

Anchors

- Patterns usually slide along the string until they match (or not)
- you can use an anchor to match the beginning or end

`\A or ^`
matches start of string

`\Z or $`
matches end of string

`\z`
matches end of string (ignoring `\n`)

`/\Ahanna/`
matches `'hannas'` but not `'johannas'`

`/\.txt\Z/`
matches a word ending in `'.txt'` (a file for example)

`/^hanna.txt$/`
matches `'hanna.txt'` but not `'johanna.txt.gz'`

matching a variable with =~

=~

can be used to match any scalar variable instead of the default $_

```
$student_name = "johanna";

if ($student_name =~ /hanna/) {
    print "hanna matches $student_name!\n";
}

if ($student_name =~ /\Ahanna\Z/) {
    print "hanna is the same as $student_name!\n";
}
```

interpolating in patterns

- variables can also be placed inside patterns

```
$dna_sequence = "AGGAGGAGGATTTTTGCGCTAGCGTAGCGGATGAA";
$dna_match = "ATG";

if ($dna_sequence =~ /$dna_match/) {
    print "found $dna_match in the sequence!\n";
}

if ($dna_sequence =~ /($dna_match)*/) {
    print "might have found $dna_match in the sequence!\n";
}
```

quantifiers

`{n}`
can be used to specify the number of consecutive matches

```
if ($dna_sequence =~ /T{5}/) {
    print "$dna_sequence contains TTTTT!\n";
}

if ($dna_sequence =~ /(TG){2}T{5}/) {
    print "$dna_sequence contains TGTGTTTTT\n";
}
```

match variables

`$1 $2 $3 ...`

These store the contents of the 1st 2nd 3rd ... pair of parentheses

```
$dna_sequence = "AGGAGGAGGATTTTTGCGCTAGCGTAGGGGATGAA";

$dna_sequence =~ /(\w{2})(T{5})(\w{2})/;

# $1 will contain GA
# $2 will contain TTTTT
# $3 will contain GC

$dna_sequence =~ /(GC)+/;

# $1 will contain GCGC


$matched_motif = $1;
# you should put the match variable somewhere else before you lose it!
```

precedence

`( )`
parenthesis are more important than

`* + ? {n}`
quantifiers, which are more important than

`^ $ \A \Z`
anchors, which are more important than

`|`
alternation, which is more important than

`abcd [] \d \w`
atoms (characters and character classes)

pattern test program

```
#!/usr/bin/perl
while (<>) {
   chomp;
   if (/YOUR_PATTERN_GOES_HERE/) {
      print "Matched: |$`<$&>$'|\n";
   } else {
      print "No match: |$_|\n";
   }
}
```

summary so far (1)

Pattern	Result
`.`	Matches any character except newline
`[a-z0-9]`	Matches any single character of set
`[^a-z0-9]`	Matches any single character *not* in set
`\d`	Matches a digit, same as [0-9]
`\D`	Matches a non-digit, same as [^0-9]
`\w`	Matches an alphanumeric (word) character [a-zA-Z0-9_]
`\W`	Matches a non-word character [^a-zA-Z0-9_]
`\s`	Matches a whitespace character (space, tab, newline...)
`\S`	Matches a non-whitespace character
`\n`	Matches a newline
`\r`	Matches a return
`\t`	Matches a tab
`\c`*`X`*	Matches an ASCII control character
`\`*`metachar`*	Matches the character itself (|,\.,*...)

summary so far (2)

Pattern	Result
`(abc)`	Rembers the match for later backreferences
`\1`	Matches whatever the first set of parens matched
`\2`	Matches whatever the second set of parens matched
`\3`	and so on...
x`?`	Matches 0 or 1 *x*'s, where *x* is any of the above
x`*`	Matches 0 or more *x*'s
x`+`	Matches 1 or more *x*'s
x`{m,n}`	Matches at least *m* *x*'s but nore more than *n*
`abc`	Matches all of a, b, and c in order
`fee|fie|foe`	Matches one of fee, fie, or foe

substitutions with `s///`

- like search and replace in word

```
s/BMC/Biomedical Centre/;
```
replaces BMC with Biomedical Centre (one time)

```
s/BMC/Biomedical Centre/g;
```
replaces BMC with Biomedical Centre (global = all occurrences)

```
while (<FILEHANDLE>) {
    s/BMC/Biomedical Centre/g;
    print;
}
```
prints out a new file where BMC is changed to Biomedical Centre

```
if (s/BMC/Biomedical Centre/) {
    print "changed BMC to Biomedical Centre\n";
}
```
the operation is true if it makes a replacement

transliteration with `tr///`

```
tr///
# maps the first characters to the second characters

$student="Sangeet";
$student=~tr/ae/XY/;
# $student becomes "SXngYYt"

$dna_sequence = "AGGAGGAGGATTTTTGCGCTAGCGTAGCGGATGAA";
$dna_sequence =~ tr/ACGT/acgt/;
# $dna_sequence becomes  "aggaggaggatttttgcgctagcgtagcggatgaa"

$a = ($dna_sequence =~ tr/a/a/);
# $a is the number of a's in the sequence
```

split

```
@array = split(/PATTERN/,$string);
```
very useful command for breaking up strings into useful units

```
$student_list = "Sangeet Malin Emma Daniel";
@students = split(/\s/,$student_list);
```
each element of `@students` has a different name

```
$student_list = "Sangeet    Malin Emma            Daniel";
@students = split(/\s+/,$student_list);
```
still works

```
$id_code = "BICF28372_id2847_rs18736_E529DY4";
@code_part = split(/_/,$id_code);
```
`$code_part[0]` will be `BICF28372`
`$code_part[1]` will be `id2847`
...

split

- 90% of my perl scripts start something like this:

```
#!/usr/bin/perl –w

open IN,$file;
while (<IN>) {
    chomp;
    @data = split(/\t/,$_);
    ...
    ...
```

this is very useful for reading a tab-delimited file line by line and placing the columns into `$data[0] $data[1] $data[2]` etc...
once you have done this, you can output them again in any way you like!

join

- goes in the opposite direction to split

```
my $result = join ("glue",@pieces);

$student_line = join ("\t",@students);
# $student_line will have a list of names separated by tab
```

The first argument of `join` is a string
Unlike `split`, it is not pattern matching

non-greedy quantifiers

- if you place a ? after a quantifier it changes from greedy (matching as much as possible) to non-greedy (match as little as possible)

`+?`
matches one or more times (prefers one)

`*?`
matches zero or more times (prefers zero)

`??`
matches one or zero times (prefers zero)

```
$dna_sequence = "AGGAGGAGGATTTTTGCGCTAGCGTAGCGGATGAA";

$dna_sequence =~ /T+?/;
#matches T

$dna_sequence =~ /GA(T+?)GC/;
#matches GATTTTTGC
```

summary: regular expressions

```
/ /   m/ /                  # pattern matching

\n \t \\                    # newline tab backslash

\d \w \s . \D \S \W         # built-in character classes

[a-z]                       # define character class

+ ? * {n}                   # quantifiers

/i /s /x /g                 # modifiers

\A \Z ^ $                   # anchors

\1 \2                       # back references

$1 $2 $& $` $'              # match variables

s/// tr///                  # substitutions

split join                  # split and join by pattern match
```