Perl: Regular expressions

A powerful tool for searching and transforming text.

Motivation

- We have seen many operations involving string comparisons
- Several Perl built-in functions also help with operations on strings
  - split & join
  - substr
  - length
- There is a lot we can do with such functions
- Example:
  - Given a string holding some timestamp, extract out different parts of date & time

```
while (my $line = <STDIN>) {
    chomp $line;
    if ($line eq “BEGIN:VSTART”) {
    # ...
    }
}

# ...

my ($property, $value) = split /:/, $foo;
if ($property eq “DSTART) {
    # ... etc etc etc
}
```

```
@csv_fields = split /,/, $input_line;

$output = join “:”, @data;

$first_char = substr $input, 0, 1;

$width = length $heading;
print $heading, “\n:
print “-” x $width;
```

Motivation

- Recall:
  - iCalendar dates are used by iCal-like programs
  - The year, month, etc. portions of the code are fixed in position
- How could we use “substr” to help us?
- This code certainly obtains what we need.
  - But it can be a bit tricky to get right.
  - Adapting code to use another date/time format is not trivial...
  - ... and is bugbait!

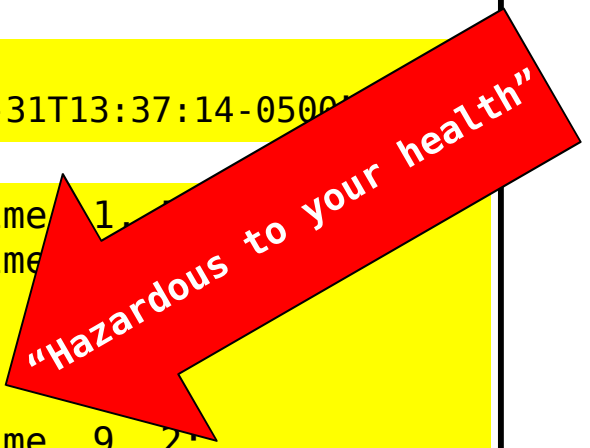
```
my $datetime = “20051225T053000”;
```

```
$year  = substr $datetime, 0, 4;
$month = substr $datetime, 4, 2;
$day   = substr $datetime, 6, 2;
$hour  = substr $datetime, 9, 2;
$min   = substr $datetime, 11, 2;
$sec   = substr $datetime, 13, 2;
```

```
# ISO 8601 time format
my $datetime = “i2003-10-31T13:37:14-0500”;

$year  = substr $datetime, 1, 
$month = substr $datetime

# coffee break
# ...
$day   = substr $datetime, 9, 2;
$hour  = substr $datetime, 12, 2;
$min   = substr $datetime, 14, 2;
$sec   = substr $datetime, 16, 2;
```



Motivation

- A better method is to indicate the string's pattern in a way the reflects the actual order of pattern components
  - The date begins at the start of the string.
  - The year is four digits.
  - The month follows (two digits)…
  - … and then the day.
  - The "T" character separates the date and time
  - Hour, minute and date follow, each two digits long.
- For the elder Perlmongers:

```
my $datetime = "20051225T053000";
```

```
my ($year, $month, $day,
    $hour, $minute, $second)
   = $datetime
       =~ m{ \A          # start of string
             (\d{4})     # year
             (\d{2})     # month
             (\d{2})     # day
             T           # literal T
             (\d{2})     # hour
             (\d{2})     # minute
             (\d{2})     # second
             \z          # end of string
         }xms;
```

```
if ($datetime =~
    /^(\d{4})(\d{2})(\d{2})T(\d{2})(\d{2})(\d{2})$/)
{
    ($year, $month, $day, $hour, $min, $sec)
        = ($1, $2, $3, $4, $5, $6);
}
```

Motivation

- Back to our "code modification" example
  - Now we have a different date format
  - Using a regular expression, we can greatly reduce the possibility of bugs
  - String begins with an "i" …
  - followed by year…
  - followed by a dash…
  - followed by month…
  - etc…

```
ISO 8601 time format
my $datetime = "i2003-10-31T13:37:14-0500";
```

```
my ($year, $month, $day,
    $hour, $minute, $second)
   = $ical_date
       =~ m{ \A          # start of string
             i           # literal i
             (\d{4})     # year
             -           # literal dash
             (\d{2})     # month
             -           # literal dash
             (\d{2})     # day
             T           # literal T
             (\d{2})     # hour
             :           # literal colon
             (\d{2})     # minute
             :           # literal colon
             (\d{2})     # second
             .+          # ignore remainder
             \z          # end of string
         }xms;
```

Topics

- Our coverage of regex syntax will be much more slowly paced that the "motivation" just shown!
  - Previous slides have been shown to give you a "flavour" of what regular expressions can achieve.
  - We will learn how to construct such expression over the next few lectures.
- We have a range of topics
- Regular expressions can seem complex and cryptic
  - However, slow and patient work with such expressions will improve your productivity.

- Simple matching
- Metacharacters
- Anchored search
- Character classes
- Range operators in character classes
- Matching any character
- Grouping
- Extracting Matches
- Search and Replace

Perl Regular Expressions

- Perl is renowned for its excellence at text processing.
- Handling of regular expressions plays a big factor in its fame.
- Mastering even the basics will allow you to manipulate text with ease.
- Regular expressions have a strong formalism (FSA).
- You have already used some and seen others.
- Other languages have some support for regexes, usually via some library.

```
% ls *.c

% ps aux | grep “s265s*” | less
```

```
Java:
import java.util.regex.*;

Python:
import re;


C#:
using System.Text.RegularExpressions;
```

Simple String Matching

- Regular expressions are usually used in conjunction with an "if"
  - "if < string matches this pattern> ..."
  - "... then > do something with that match> ."
- The simplest such match refers to a string
- But note: this is much different that using "eq"

```
my $line = <SOMEINPUT>;
chomp $line;

# Unbeknownst to programmer, the first line
# of the input is the line "Hello, World";

if ($line =~ m/World/xms) {
    print "Regexp matches!\n";
}
else {
    print "Oh, poop.\n";
}

if ($line eq "World") {
    print "line is equal to 'World'\n";
}
else {
    print "line sure ain't equal to 'World'\n";
}
```

A word about "m/yadayada/xms"

- The text between the two slashes is the regular expression ("regex").
- Leading "m" indicates the regex is used for a match
- Trailing "xms" are three regex options
  - "x": Extended formatting (whitespace in regex is ignored)
  - "m": For line boundaries (and eliminates a cause of some subtle bugs)
  - "s": ensures everything is matched by the "." symbol
- Why all of this verbiage instead of plain old "/yadayada/" as of old?

```
/'[^\\']*(?:\\.[^\\']*)*'/
```

- Also note: "m{}" or "m//"

```
m{ '            # an opening single quote
   [^\\']*      # any non-special chars
   (?:          # then all of..
       \\ .     #   any explicitly backslashed char
       [^\\']* #   followed by any non-special chars
   )*           # repeated zero of many times
   '            # a closing single quote
}xms
```

Another example

- The code on the right searches for a pattern in some dictionary file
  - Note that a command-line argument is being used for a regex!
  - Also note "<>" syntax: This takes the first unused command-line argument, and uses it as a filename for opening!

```
#!/usr/bin/perl

use strict;

my $regexp = shift @ARGV;
while (my $word = <>) {
    if ($word =~ m/$regexp/xms) {
        print $word;
    }
}
```

```
% ./search.pl pter /usr/share/dict/linux.words
abrupter
Acalypterae
acanthopteran
Acanthopteri
... <snip> ...
unchapter
unchaptered
underprompter
... <snip> ...
Zygopteris
zygopteron
zygopterous
%
```

Metacharacters

- Regexs obtain their power by describing sets of strings.
- Such descriptions involve the use of "metacharacters"
- Of course, some strings that we want to match will contain these strings.
  - Therefore we must "escape" them.

```
{ } [ ] ( )
  ^   $   .
  |   *   ?
    /   \
```

```
"2+2=4" =~ m/2+2/xms    # doesn't match

"2+2=4" =~ m/2\+2/xms   # does match

"The interval is [0,1)." =~
    m/[0,1)./xms        # syntax error

"The interval is [0,1)." =~
    m/\[0,1\)\./xms     # does match

"/usr/bin/perl"
   =~ m/\/usr\/bin/\/perl/xms  # matches

"/usr/bin/perl"
   =~ m{/usr/bin/perl}xms      # better

'C:\WINDOWS' =~ m/C:\\WINDOWS/ # matches
```

Anchoring

- We may wish to “anchor” a match to certain locations
  - “^ ” matches the beginning of a line.
  - “$” matches the end of a line.
  - “\A” matches the beginning of a string.

```
“housekeeper” =~ m/keeper/xms              # matches
“housekeeper” =~ m/^keeper/xms             # does not match
“housekeeper” =~ m/keeper/xms              # matches
“housekeeper” =~ m/keeper\n/xms            # also matches

“keeper”      =~ m/^keep$/xms              # does not match
“keeper”      =~ m/^keeper$/xms            # matches
“keeper”      =~ m{\A keeper \z}xms        # matches
```

```
my $text ="Here is one line.\nIt is followed by\nAnother line!\n";

if ($text =~ m{line\. $}x) { print "Gotcha\n"; } else { print "Oh dear\n"; }

if ($text =~ m{line\. $}xm) { print "Gotcha\n"; } else { print "Oh dear\n"; }
```

Character classes

- These allow sets of possible characters to be matched
- Used at desired points within a regex.

```
m/cat/xms                  # matches ‘cat’
m/[bcr]at/xms              # matches ‘bat, ‘cat’, or ‘rat’
m/item[0123456789]/xms     # matches ‘item0’, .. ‘item9’

“abc” =~ m/[cab]/xms       # matches ‘a’
m/[yY][eE][sS]/xms         # matches case-insensitive YES
m/yes/xmsi                 # simpler way, using “i”
m/(?i)yes/xms              # same

m/[\]c]def/xms             # matches ‘]def’ or ‘cdef’

$x =‘bcr’
m/[$x]at/xms               # matches ‘bat’, ‘cat’, ‘rat’
m/[\$x]at/xms              # matches ‘$at’ or ‘xat’
m/[\\$x]at/xms             # matches ‘\at’, ‘bat, ‘cat’,
                             or ‘rat’
```

Range operators

- Ranges can eliminate some ugly code
  - [0123456789] becomes [0-9]
  - [abcdefghijklmnopqrstuvwxyz] becomes [a-z]
- If "-" is the first or last character in a character class, it is treated as an ordinary character

```
m/item[0-9]/xms   # item0, item1, ... item9
m/[0-9bx-z]aa/xms # '0aa', ..., '9aa',
                  #  'baa', 'xaa', 'yaa',
                  # or 'zaa'
m/[0-9a-fA-F]/xms # matches hex digit
m/[a-z]/i         # matches a "word" char
```

```
# all are equivalent

m/[-ab]/xms
m/[ab-]/xms
/[a\-b]/xms
```

Negated character classes

- The special character ^ in the first position of a character class denotes a negated character class
- Matches any character but those in the brackets

```
m/[^a]at/xms
            # doesn't match 'aat' or 'at', but
            #   matches all other 'bat', 'cat,
            #  '0at', '%at', etc.

m/[^0-9]/xms
            # matches a non-numeric character


m/[a^]at/xms
            # matches 'aat' or '^at'; here '^'
            #  is ordinary
```

Matching *any* character

- The period **'.'** matches any character but "\n"
- A period is a metacharacter, it needs to be escaped to match as an ordinary period.

```
m/..rt/xms          # matches any 2 chars, followed by 'rt'
m/end\./xms         # matches 'end.'
m/end[.]/xms        # same thing, matches only end.
"" =~ m/./xms       # doesn't match - needs a character
"a" =~ m/^.$/xms    # matches

"" =~ m/^.$/xms     # doesn't match - needs a character
"\n" =~ m/^.$/xms   # doesn't match - needs a character
                    # other than \n
"a\n" =~ m/^.$/xms  # matches, ignores the \n
```

Matching this or that

- We would like to match different possible words or character strings
- We use the *alternation* character | (pipe)

```
"cats and dogs" =~ /cat|dog|bird/ # matches "cat"
"cats and dogs" =~ /dog|cat|bird/ # matches "cat"
```

Grouping Things Together

- Sometimes we want alternatives for part of a regular expression.

```
/(a|b)b/      # matches 'ab' or 'bb'
/(ac|b)b/     # matches 'acb' or 'bb'
/(^a|b)c/     # matches 'ac' at start of string or
              #    'bc' anywhere
/(a|[bc])d/   # matches 'ad', 'bd', or 'cd'

/house(cat|)/          # matches either 'housecat'
                       #   or 'house'
/house(cat(s|)|)/      # matches either 'housecats' or
                       # 'housecat' or 'house'.
                       # Note groups can be nested.
```

Extracting Matches

- The grouping metacharacters () also serve another completely different function: they allow the extraction of the parts of a string that matched.
- For each grouping, the part that matched inside goes into the special variables $1, $2, etc.

```
# extract hours, minutes, seconds
$time =~ /(\d\d):(\d\d):(\d\d)/  # match hh:mm:ss format

# \d is equivalent to [0-9]
$hours = $1;
$minutes = $2;
$seconds = $3;

# More compact code, equivalent code
($hours,$minutes,$second) = ($time =~/(\d\d):(\d\d):
(\d\d)/)
```

Matching Repetitions

- We would like to be able to match multiple times:
  - **a?** = match 'a' 0 or 1 times (~ optional)
  - **a*** = match 'a' 0 or more times, i.e., any number of times
  - **a+** = match 'a' 1 or more times, i.e., at least once
  - **a{n,m}** = match at least n times, but not more than m times.
  - **a{n,}** = match at least n or more times.
  - **a{n}** = match exactly n times

```
$year =~ /\d{2,4}/  # make sure year is at least 2 but
                    #   not more than 4 digits

/[a-z]+\d*/i        # match a word and any number of digits

/y(es)?/i           # matches 'y', 'Y',
                    #   or a case-insensitive 'yes'
```

Search and Replace

- Regular expressions also play a role in search and replace operations in Perl
- Search and replace is accomplished with the **`s///`** operator
- General form:

s/regexp/replacement/modi ers

```
$x = "Time to feed the cat!";
if ( $x =~ s/cat/hamster/ ) {
    print $x;  # “Time to feed the hamster!”
}
```

More Search and Replace Commands

```
$y = "'quoted words'";
$y =~ s/^'(.*)'$/<<$1>>/ # strip single quotes, $y
                         # contains "<<quoted words>>"

$x = "I batted 4 for 4";
$x =~ s/4/four/          # doesn't do it all:
                         #   $x contains
                         #   "I batted four for 4"
 $x = "I batted 4 for 4";
 $x =~ s/4/four/g        # /g modifier does it all:
                         #   $x contains
                         #   "I batted four for four"
```

A few more regexp topics

- Advanced uses of matches
- Escape sequences
- List and scalar context, e.g., phone numbers
- Finding all instances of a match
- Parenthesis
- Substituting with s///
- **tr**, the translate function

Advanced uses of matches

- You can assign ***pattern memory*** directly to your own variable names (***capturing***):

```
($phone) = $value =~ /^phone\:(.+)$/;
```

  - Read from right to left. Apply this pattern to the value in **$value**, and assign the results to the ***list*** on the left.

```
($front,$back) = /^phone\:(\d{3})-(\d{4})/;
```

  - Apply this pattern to **$_** and assign the results to the ***list*** on the left.

Meaning of backslash letters

- \n : newline
- \r: carriage return
- \t: tab
- \f: formfeed
- \d: a digit (same as [0-9])
- \D: a non-digit
- \w: an alphanumeric character, same as [0-9a-z_A-Z]
- \W: a non-alphanumeric character
- \s: a whitespace character, same as [\t\n\r\f]
- \S: a non-whitespace character

Reminder: list or scalar context?

- A pattern match returns 0 (false) or 1 (true) in ***scalar context***, and a list of matches in ***array context***.
- Recall: There are a lot of functions that do different things depending on whether they are used in scalar or list context.

```
# returns the number of elements
$count = @array

# returns a reversed string
$revString = reverse $string

# returns a reversed list
@revArray = reverse @array
```

- You must always be cautious of this behaviour.

Practical Example of Context

```
$phone = $string =~ /^.+\:(.+)$/;
```

- **$phone** contains 1 if pattern matches, 0 otherwise

```
($phone) = $string =~ /^.+\:(.+)$/;
```

- **$phone** contains the matched string

Finding all instances of a match

- Use the **'g'** modifier with a regular expression

```
@sites = $sequence =~ /(TATTA)/g;
```

  - think `g` for ***global***
  - Returns a list of all the matches (in order), and stores them in the array
  - If you have ***n pairs*** of parentheses, the array looks like the following:
    - `($1,$2,…$n,$1,$2,…$n,…)`

Perl is Greedy

- Perl regular expressions try to match the largest possible string which matches your pattern:

  `"lalaaaaagag" =~ /(la.*ag)/`

  - **`/la.*ag/`** matches **`laag, lalag, laaaaaag`**
  - **`$1`** contains **`"lalaaaaagag"`**
- If this is not what you wanted to do, use the **`'?'`** modifier:

  `"lalaaaaagag" =~ /(la.+?ag)/`

  - **`/(la.+?ag)/`** matches as few characters as possible to find matching pattern
  - **`$1`** contains **`"lalaaaaag"`**

Making parentheses forgetful

- Sometimes you need parentheses to make your regular expression work, but you don't actually want to keep the results. You can still use parentheses for grouping.
- **`/(?:group)/`**
  - Certain characters are overloaded; recall:
    - **`\d?`** means 0 or 1 instances
    - **`\d+?`** means the fewest non zero number of digits
    - **`(?:group)`** means look for the group of atoms in the string, but don't remember them

Example of "*forgetting*"

```
#!/usr/bin/perl
# Method 1

if (@ARGV && $ARGV[0] eq "-x") {
        $mod = "?:";
} else {
        $mod = "";
}

$pat1 = "\\w+";
$pat2 = "\\d+";

while (<STDIN>) {
    $_ =~ /($mod$pat1) ($pat2)/;
    print $1, "\n";
}
```

```
#!/usr/bin/perl
# Method 2

if (@ARGV && $ARGV[0] eq "-x") {
        $ignore = 1;
} else {
        $ignore = 0;
}

while (<STDIN>) {
    $_ =~ /(\w+) (\d+)/;
    if ($ignore) {
        print $2, "\n";
    }
    else {
        print $1, "\n";
    }
}
```

More examples using s///

- Substituting one word for another

```
$string =~ s/dogs/cats/
```

  - If **$string** was "I love dogs", it is now "I love cats"

- Removing trailing white space

```
$string =~ s/\s+$//
```

  - If **$string** was 'ATG ', it is now 'ATG'

- Adding 10 to every number in a string

```
$string =~ /(\d+)/$1+10/ge
```

  - Note ***pattern memory***
  - **g** means **global** (just like in regular expressions)
  - **e** is specific to **s**, ***e***valuate the expression on the right

tr function

- ***tr***anslate or ***tr***ansliterate
- General form:

  tr/list1/list2/
- Even less like a regular expression than ***s***
- substitutes characters in the first list with characters from the second list:

  $string =~ tr/a/A/
  - every ‘a’ to translated to an ‘A’
  - No need for a global modifier using ***tr***.

More examples of tr

- converting named scalar to lowercase

```
$ARGV[1] =~ tr/A-Z/a-z/
```

- count the number of "*" in $_

```
$cnt = tr/*/*/
$cnt = $_ =~ tr/*/*/
```

- change all non-alphabetic characters to spaces

```
tr/a-zA-Z/ /c
```

  - notice space + **c** = complement search string

- delete all non-alphabetic characters completely

```
tr/a-zA-Z//cd
```

  - **d** = delete found but unreplaced characters

Using the results of matches within a pattern

- `\1, \2, \3` refer to what a previous set of parentheses matched

```
"abc abc" =~ /(\w+) \1/  # matches
"abc def" =~ /(\w+) \2/  # doesn't match
```

- Can also use **`$1, $2`**, etc. to perform some interesting operations:

```
s/^([^ ]*) *([^ ]*)/$2 $1/  #swap first two words
/(\w+)\s*=\s*\1/            # match "foo = foo"
```

- other default variables used in matches
  - **`$`\`** : returns everything before matched string
  - **`$&`** : returns entire matched string
  - **`$'`** : returns everything after matched string

Example: Celsius Fahrenheit

```
#! /usr/bin/perl -w
print "Enter temperature: \n";
$line = <STDIN>;
chomp($line);

if ( $line =~ /^([-+]?[0-9]+(?:\.[0-9]*)?)\s*([CF])$/i ) {
     $temp = $1;
     $scale = $2;
     if ( $scale =~ /c/i ) {
         $cel = $temp;
         $fah = ($cel * 9 / 5) + 32;
     }
     else {
         $fah = $temp;
         $cel = ($fah - 32) * 5 / 9;
     }
     printf( "%.2f C is %.2f F\n", $cel, $fah );
}
else {
    printf( "Bad format\n" );
}
```

Regex on command line

- We can execute simple regular expressions on the command line:

```
$ perl –p –i –e 's/kat/cat/g' in.txt
```

  - **p** : apply program to each line in file **in.txt**
  - **i** : write changes back to **in.txt**
  - **e** : program between '...'