CSCI 4152/6509
Natural Language Processing

Lab 2:

Perl Tutorial 2

Faculty of Computer Science

Dalhousie University

Lab Overview

- Use of Regular Expressions in Perl
- This topic is discussed in class, we will see some more examples in this lab
- The second part of the lab includes some practice with Regular Expressions
- Practice with processing Character N-grams

Lab Evaluation

- The lab will be evaluated as a part of an assignment with the same submission deadline as the assignment, which will be at least one week after the lab.
- Files to be submitted by the end of the lab are:

1. `lab2-matching.pl`
2. `lab2-matching-data.pl`
3. `lab2-word-counter.pl`
4. `lab2-replace.pl`
5. `lab2-line-count.pl`

Some References about Regular Expressions in Perl

- To read more (e.g., on timberlea):
  - `man perlrequick`
  - `man perlretut`
  - `man perlre`
- Same information on:
  `http://perldoc.perl.org/perlrequick.html`
  `http://perldoc.perl.org/perlretut.html`
  `http://perldoc.perl.org/perlre.html`
- Used for string matching, searching, transforming
- Built-in Perl feature

Introduction to Regular Expressions

- A simple example:

```
if ("Hello World" =~ /World/) {
  print "It matches\n";
} else {
  print "It does not match\n";
}
```

Regular Expressions: Basics

- A simple way to test a regular expression:

  ```
  while (<>)
  { print if /book/ }
  ```

  prints lines that contain substring ‘book’
- /chee[sp]eca[rk]e/ would match: cheesecare, cheepecare, cheesecake, cheepecake
- option /i matches case variants; i.e., /book/i would match Book, BOOK, bOoK, etc., as well
- Beware that substrings of words are matched, e.g., "That hat is red" =~ /hat/; matches ‘hat’ in ‘That’

RegEx — No match

```
if ("Hello World" !~ /World/) {
  print "It doesn’t match\n";
} else {
  print "It matches\n";
}
```

Character Classes (1)

`/200[012345]/`	match one of the characters
`/200[0-9]/`	character range
`/From[^:!]/`	match any character but `:` or `!`
`/[^a]at/`	does not match ‘aat’ or just ‘at’ but does ‘bat’, ‘cat’, ‘0at’, ‘%at, etc.
`/[a^]at/`	matches ‘aat’ or ‘^at’
`/[^a-zA-Z]the[^a-zA-Z]/`	multiple ranges
`/[0-9ABCDEFa-f]/`	match a hexadecimal digit

Character Classes (2)

`.` (period) any character but new-line

`\d` any digit; i.e., same as `[0-9]`

`\D` any character but digit

`\s` any whitespace character; e.g., space, tab, newline

`\S` any character but whitespace; i.e., printable

`\w` any word character (letter, digit, underscore)

`\W` any non-word character; i.e., any except word characters

Some more examples:

`/\d\d:\d\d:\d\d/` matches a hh:mm:ss time format

`/[\d\s]/` matches any digit or whitespace

`/\w\W\w/` matches a word char, followed by non-word char, followed by word char

`/..rt/` matches any two chars followd by ‘rt’

`/end\./` matches ‘end.’

Word Boundary Anchor (`\b`)

- `\b` is word boundary anchor. It matches inter-character position where a word starts or ends; e.g., between `\w` and `\W`
- Examples:

```
$x = "Housecat catenates house and cat";
$x =~ /cat/          matches cat in ‘housecat’
$x =~ /\bcat/        matches cat in ‘catenates’
$x =~ /cat\b/        matches cat in ‘housecat’
$x =~ /\bcat\b/      matches ‘cat’ at end of string
```

Anchors ^ and $

```
"housekeeper" =~ /keeper/;    # match
"housekeeper" =~ /^keeper/;   # no match
"housekeeper" =~ /keeper$/;   # match
"housekeeper\n" =~ /keeper$/; # match

"keeper" =~ /^keep$/;    # no match
"keeper" =~ /^keeper$/;  # match

"" =~ /^$/; # ^$ matches an empty
            # string
```

Matching: Alternatives (Choices)

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

"cab" =~ /a|b|c/ # matches "c"
                 # /a|b|c/ == /[abc]/
/(a|b)b/;    # matches "ab" or "bb"
/(ac|b)b/;   # matches "acb" or "bb"
/(^a|b)c/;   # matches "ac" at start, "bc" anywhere
/(a|[bc])d/; # matches "ad", "bd", or "cd"
/house(cat|)/; # matches "housecat" or "house"
/house(cat(s|)|)/; # matches  "housecats", "housecat"
                   # or "house". Groups can be nested.
/(19|20|)\d\d/; # match years 19xx, 20xx, or xx

"20" =~ /(19|20|)\d\d/; # matches null alternative
    # /(19|20)\d\d/ would not match
```

Repetitions

```
a?     means: match "a" 1 or 0 times

a*     means: match "a" 0 or more times;
              i.e., any number of times
a+     means: match "a" 1 or more times;
              i.e., at least once

a{n,m}    means: match at least n times and
                 not more than m times.
a{n,}     means: match at least n or more times
a{n}      means: match exactly n times

/[a-z]+\s+\d*/  letters a-z, spaces, and maybe digits
/(\w+)\s+\1/    match doubled words (back reference)
/y(es)?/i       'y', 'Y', or case-insensitive 'yes'
```

Extractions (or Captures)

```
# extract hours, minutes, seconds
if ($time =˜ /(\d\d):(\d\d):(\d\d)/)
{ # match hh:mm:ss format
  $hours = $1;
  $minutes = $2;
  $seconds = $3;
}

# Another way to capture substrings:
($h, $m, $s) = ($time =˜ /(\d\d):(\d\d):(\d\d)/);

/(ab(cd|ef)((gi)|j))/;
 1  2      34          # opening parentheses order

/\b(\w\w\w)\s\1\b/;    # use of backreferences
```

Selective Grouping

```
# may want to use grouping but no substring capture
# use modified grouping: (?:regex)

# E.g.: match a number, $1-$4 are set, but we want $1
/([+-]?\ *(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?)/;

# match a number faster, only $1 is set:
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][+-]?\d+)?)/;

# match a number, get $1 = entire num., $2 = exp.
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE]([+-]?\d+))?)/;
```

Greediness in regex Matching

```
# by default: left−most longest match (greedy)

$x = "the cat in the hat";

$x =˜ /ˆ(.∗)(at)(.∗)$/;
  # matches:
  # $1 = ’the cat in the h  (left−most longest)
  # $2 = ’at
  # $3 = ’’ (0 characters match)

$x =˜ /ˆ(.∗?)(at)(.∗)$/; # first group shortest match
  # matches:
  # $1 = ’the c
  # $2 = ’at
  # $3 = ’ in the hat’
```

Shortest Matches (Minimizing Greediness)

```
a??      # match ’a’ 0 or 1 times. Try 0 first, then 1.

a*?      # match ’a’ 0 or more times, but as few times
         # as possible

a+?      # match ’a’ 1 or more times, but as few times
         # as possible

a{n,m}? # match at least n and not more than m times,
         # but as as few times as possible

a{n,}?  # match at least n times, but as few times as
         # possible

a{n}?    # match exactly n times; so a{n}? is equivalent
         # to a{n}
```

Look-aheads, Look-behinds

```
$x = "I catch the housecat 'Tom-cat' with catnip";

$x =~ /cat(?=\s)/; # look-ahead
      # matches 'cat' in 'housecat'
@catwords = ($x =~ /(?<=\s)cat\w+/g); # look-behind
      # matches:
      # $catwords[0] = 'catch'
      # $catwords[1] = 'catnip'
$x =~ /\bcat\b/;
      # matches 'cat' in 'Tom-cat'
$x =~ /(?<=\s)cat(?=\s)/;
      # doesn't match; no isolated 'cat' in
      # middle of $x
$x =~ /(?<!\s)foo(?!bar)/; # negative look-behind and
                           # negative look-ahead
```

Replacements: s/regex/replacement/

```
# General format: s/regexp/replacement/modifiers
# 1-letter modifiers, also called flags or options

$x = "Time to feed the cat!";
$x =~ s/cat/hacker/;
  # $x now contains "Time to feed the hacker!"

$strong = 1 if $x =~ s/^(Time.*hacker)!$/$1 now!/;

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

$x =~ s/(?<=\s)cat(?=\s)/dog/g; # modifier ’g’ used
                                # to replace all matches
```

More Replacement Examples

```
$x = "I batted 4 for 4";
$x =~ s/4/four/; # does not replace all 4s:
                 # $x contains "I batted four for 4"

$x = "I batted 4 for 4";
$x =~ s/4/four/g; # flag "g" (global) replaces all:
                  # $x contains "I batted four for four"

$x = "Bill the cat";
$x =~ s/(.)/$ch{$1}++;$1/eg; # flag "e" (evaluate)
         # counts characters, and final $1 simply
         # replaces char with itself

# Printing characters by frequency, sorted:
print "frequency of ’$_’ is $ch{$_}\n"
  for sort {$ch{$b} <=> $ch{$a}} keys %ch;
```

End of Regular Expressions

- We end review of regular expressions in Perl here
- Hands-on Exercises to follow

Step 1. Logging in to server timberlea

1-a: Login to the server `timberlea`

1-b: Check permissions of your course directory `csci4152` or `csci6509`:

```
 ls -ld csci4152
```
or
```
ls -ld csci6509
```

1-c: Change directory to `csci4152` or `csci6509`

1-d: Create directory `lab2` and enter it:

```
    mkdir lab2
    cd lab2
```

Step 2: Testing Regular Expressions

- Create file called `lab2-matching.pl` with the content provided in the notes
- Make it executable and run it
- Enter some input lines including the word ‘book’ and not
- End input with Control-d (`C-d`)
- Submit `lab2-matching.pl` using `submit-nlp`

Step 3: Using DATA

- Write a program called `lab2-matching-data.pl` with the content provided in the notes
- Notice use of keywords: `DATA` and `__DATA__`
- Use of variables: `` $` ``, `$&`, and `$'`
- Test it
- You can extend it if you want
- Submit it using `submit-nlp`

Step 4: Counting words

- Write a program called `lab2-word-counter.pl` with the content provided in the notes
- It is a simple program for counting words
- `g` modifier after match is used to continuously match for new words in the loop
- Test it
- Submit it using `submit-nlp`

Step 5: Simple Task 1

- Write a program called `lab2-replace.pl` as specified in the notes
- Read the comments and fill the missing line in the code
- It is about replacing any case-insensitive string ‘book’ with the strictly lowercase version
- Test it
- Submit it using `nlp-submit`

Some String Functions

- Side note: `man perlfunc` gives a lot of information about different Perl functions
- **chomp** *string*; removes trailing newline from the string if it exists
- Like all predefined Perl functions, **chomp** can be used with parentheses as well, as in:
  chomp(*string*);
- **chomp;** applies `chomp` to the default variable (`$_`), like most other functions
- **length** *string*; string length
- **index***(str,substr[,offset])* returns position of the substring *substr* in the string *str*, starting from offset *offset*; if *offset* is not included, 0 is assumed; returns -1 if substring not found
- **substr***(str,begin[,len])* returns substring of string *str* starting from *begin*, with length *len*; if *len* is missing, returns to the end of string *str*

Some String Functions: sprintf

- **sprintf***(format, @arguments)* an elaborate function to create a string based on a given format with provided list of arguments; similar to the C function `printf`, more information provided in `man perlfunc`

Review: Standard Input and Standard Output

- Remember that *standard input* and *standard output* (and *standard error*) have a precise meaning in the Linux or Unix environment
- When a program reads *standard input* it reads keyboard by default
- When a program writes to *standard output* it prints to the screen terminal
- Redirection operators such as ‘<’ and ‘>’ can be used to redirect standard input from a file, or standard output to a file
- Redirection operators are used in the command line and do not depend on a programming language

Basic I/O in Perl

- We have seen basic “diamond” operator `<>` for reading input
- The diamond operator `<>` behaves in a special way:
  - if the program is not given arguments, the diamond operator reads the standard input
  - if the program is given arguments, the diamond operator treats the first argument as the file name, opens the file, and reads it; when finished, it will open the next file using the next argument as the file name
- For output, we can use `print`
- `printf` can be used for formatted output
- We can also explicitly open and close files using command `open` and `close`
- `print` can be used to print to a file
- Let us look at some examples

Some I/O Code Snippets

We can read the standard input, or from files specified in the command line and print using the following code snippet:

```
while ($line = <>) { print $line }
```

or using the default variable `$_`:

```
while (<>) { print }
```

The following two lines show different behaviour of `<>` depending on the context:

```
$line = <>;  # reads one line
@lines = <>; # reads all lines,

print "a line\n"; # output, or
printf "%10s %10d %12.4f\n", $s, $n, $fl;
     # formatted output
```

Reading from a File

```
my $filename = ’file.txt’;

#using file handle $fh

open(my $fh, ’<’, $filename);

my $line = <$fh>;

print $line;

close $fh;
```

Reading from a File, with Error Check after Opening

```
my $filename = ’file.txt’;

#using file handle $fh

open(my $fh, ’<’, $filename)
    or die "Cannot open file $filename: $!";

my $line = <$fh>;

print $line;

close $fh;
```

Writing to a File

```
my $filename = ’file.txt’;

#using file handle $fh

open(my $fh, ’>’, $filename)
    or die "Cannot open file $filename $!";

print $fh "new first line\n";

close $fh;
```

Appending to a File

```
my $filename = ’file.txt’;

#using file handle $fh

open(my $fh, ’>>’, $filename)
    or die "Cannot open file $filename $!";

print $fh "new last line\n";

close $fh;
```

Step 6: Count Number of Lines

- Write a program `lab2-line-count.pl`
- Usage: `./lab2-line-count.pl file.txt`
- Output: `file.txt has 124 lines`
- Remember to include a file header comment
- Submit `lab2-line-count.pl` using `nlp-submit`

Step 7: End of the Lab

- Make sure that you submitted all required files:
  `lab2-matching.pl`,
  `lab2-matching-data.pl`,
  `lab2-word-counter.pl`,
  `lab2-replace.pl`,
  `lab2-line-count.pl`
- End of the lab.