CMSC 330: Organization of Programming Languages

Ruby Regular Expressions
String Processing in Ruby

- Scripting languages provide many useful libraries for manipulating strings

- The Ruby `String` class provides useful methods that can
  - Concatenate two strings
  - Extract substrings
  - Search for a substring and Replace with something else
What if we want to find more complicated patterns? E.g.,
- Either Steve, Stephen, Steven, Stefan, or Esteve
- All words that have even number vowels

We need Regular Expressions
Regular Expressions

- A regular expression is a pattern that describes a set of strings. It is useful for:
  - Searching and matching
  - Formally describing strings
    - The symbols (lexemes or tokens) that make up a language

- Common to lots of languages and tools:
  - Syntax for them in sed, grep, awk, Perl, Python, Ruby, …
    - Popularized (and made fast) as a language feature in Perl

- Based on some elegant theory:
  - Future lecture
Ruby Regular Expressions

- Regular expressions are instances of Regexp
  - Surround regexp e with slashes: so /e/ has type Regexp

- Basic matching using =~ method of String

```ruby
line = gets  # read line from standard input
if line =~ /Ruby/ then  # =~ returns nil if regexp not matched
  puts "Read-in line contained Ruby"
end
```

- x =~ y is sugar for x.=~(y)
Example Regular Expressions in Ruby

- /Ruby/
  - Strings are matched exactly; here, the string "Ruby"

- /Ruby|OCaml/
  - e1|e2 means to match either e1 or e2
  - Here, matches either "Ruby" or "OCaml"

- /(ab)/
  - 0 or more occurrences of “ab”: matches “”, “ab”, “abab”, “ababab”,…
Repetition in Regular Expressions

The following are suffixes on a regular expression $e$

- $e^*$: zero or more occurrences of $e$
- $e^+$: one or more occurrences of $e$
  
  so $e^+$ is the same as $ee^*$

- $a^*$: ", "a", "aa", "aaa", ...

- $a^+$: "a", "aa", "aaa", ...

- $bc^*$: "b", "bc", "bcc", ...

- $a+b^*$: "a", "ab", "aa", "aab", "aabb", "aabb", "aaa", ...
Repetition in Regular Expressions

The following are suffixes on a regular expression $e$

- $e^*$: zero or more occurrences of $e$
- $e^+$: one or more occurrences of $e$
  
  so $e^+$ is the same as $ee^*$
- $e?$: exactly zero or one $e$
- $e\{x\}$: exactly $x$ occurrences of $e$
- $e\{x,\}$: at least $x$ occurrences of $e$
- $e\{x,y\}$: at least $x$ and at most $y$ occurrences of $e$
Watch Out for Precedence

- `/Ruby*/` means `""`, "Ruby", "RubyRuby", ...

- `/Ruby*/` means `"Rub", "Ruby", "Rubyy", ...

- Best to use parentheses to disambiguate
  - Note that parentheses have another use, to extract matches, as we’ll see later
Character Classes

- `/[abcd]/`
  - `{"a", "b", "c", "d"}` (Can you write this another way?)

- `/[a-zA-Z0-9]/`
  - Any upper- or lower-case letter or digit

- `/[^0-9]/`
  - Any character except 0-9 (the ^ means not, and must come first)

- `/	
`/`
  - Tab, newline or space

- `/[a-zA-Z_\$][a-zA-Z_\$0-9]*/`
  - Java identifiers ($ escaped...see next slide)
Special Characters

.  any character
^  beginning of line
$  end of line
\$  just a $
\d  digit, [0-9]
\s  whitespace, [\t\r\n\f ]
\w  word character, [A-Za-z0-9_]  
\D  non-digit, [^0-9]
\S  non-space, [^\t\r\n\f ]
\W  non-word, [^A-Za-z0-9_]
Potential Syntax Confusions

- [ ]
  - Inside regular expressions: character class
  - Outside regular expressions: array
    - Note: [a-z] does not make a valid array

- ^
  - Inside regex character class: *not*
  - Outside regex character class: beginning of line

- ( )
  - Inside character classes: literal characters ( )
    - Note /(0..2)/ does not mean 012
  - Outside character classes in regex: used for grouping

- –
  - Inside regex character classes: range (e.g., a to z given by [a-z])
  - Outside regular expressions: subtraction
Summary

Let \( re \) represents an arbitrary pattern; then:

- \( /re/ \) – matches regexp \( re \)
- \( /(re_1|re_2)/ \) – match either \( re_1 \) or \( re_2 \)
- \( /(re)^*/ \) – match 0 or more occurrences of \( re \)
- \( /(re)^+/ \) – match 1 or more occurrences of \( re \)
- \( /(re)?/ \) – match 0 or 1 occurrences of \( re \)
- \( /(re)\{2\}/ \) – match exactly two occurrences of \( re \)
- \( /[a-z]/ \) – same as \((a|b|c|...|z)\)
- \( /[^0-9]// \) – match any character that is not 0, 1, etc.
- \( ^, $ \) – match start or end of string
Try out regexps at rubular.com
Regular Expression Practice

- Any string containing two consecutive \textit{ab}

- Any string containing \textit{a} or two consecutive \textit{b}
Regular Expression Practice

- Any string containing two consecutive `ab`
  
  `/(ab){2}/`

- Any string containing `a` or two consecutive `b`
  
  `/a|bb/`
Regular Expression Practice

Contains sss or ccc
Regular Expression Practice

Contains sss or ccc

/s\{3\}|c\{3\}/
Contains exactly 2 b's, not necessarily consecutive.

/\^ b b $/
Contains exactly 2 b's, not necessarily consecutive.

```
/^[^b]*b[^b]*b[^b]*$/
```

beginning

Any character not b

2 b's

end
Regular Expression Practice

- Starts with c, followed by one lowercase vowel, and ends with any number of lowercase letters

```
/^c[aouei][a-z]*$/
```

CMSC 330 Summer 2021
Regular Expression Practice

- Starts with c, followed by one lowercase vowel, and ends with any number of lowercase letters

/\^c [aouei] [a-z]* $/
Regular Expression Practice

- Starts with a and has exactly 0 or 1 letter after that
Regular Expression Practice

- Starts with a and has exactly 0 or 1 letter after that

/^a[A-Ża-z]?$/
Regular Expression Practice

- Only lowercase letters, in any amount, in alphabetic order
Regular Expression Practice

- Only lowercase letters, in any amount, in alphabetic order

/\^a*b*c*d*e*f*g*h*i*j*k*l*m*n*o*p*r*t*u*v*w*x*y*z*$\/
Regular Expression Practice

- Contains one or more ab or ba
Regular Expression Practice

- Contains one or more ab or ba

/\(ab|ba\)+/
Regular Expression Practice

- Precisely steve, steven, or stephen
Regular Expression Practice

Precisely steve, steven, or stephen

/^ste(ve|phen|ven)$/
Regular Expression Practice

- Even length string
Regular Expression Practice

- Even length string

```
/^([^a-zA-Z0-9]*$/
```

any two characters
Regular Expression Practice

- Even number of lowercase vowels
Regular Expression Practice

Contains even number of lowercase vowels

```
/^([^aouei]*[^aouei][^aouei]*[^aouei][^aouei]*)*$ /
```

Non-vowel  vowel
Regular Expression Practice

- Starts with anything but b, followed by one or more a’s and then no other characters
Regular Expression Practice

- Starts with anything but b, followed by one or more a’s and then no other characters

/^[^b]+a+$/
Quiz 1

How many different strings could this regex match?

/^Hello, Anyone awake?$/

A. 1
B. 2
C. 4
D. More than 4
Quiz 1

How many different strings could this regex match?

`/^
Hello, Anyone awake?
$/`

A. 1
B. 2
C. 4
D. More than 4
Quiz 2

Which regex is **not** equivalent to the others?

A. `^[cmSc]$`
B. `^c?m?s?c?$`
C. `(c|m|s|c)$`
D. `^([cm]|[sc])$`
Quiz 2

Which regex is not equivalent to the others?

A. `^[cmSc]$`
B. `^c?m?s?c?$`
C. `^(c|m|s|c)$`
D. `^([cm]|[sc])$`
Quiz 3

Which string does not match the regex?

/^[a-z]{4}\d{3}$/

A. “cmsc\d\d\d”
B. “cmsc330”
C. “hellocmsc330”
D. “cmsc330world”
Quiz 3

Which string does not match the regex?

```
[a-zA-Z]{4}\d{3}[

A. “cmsc\d\d\d”
B. “cmsc330”
C. “hellocmsc330”
D. “cmsc330world”
```
Extracting Substrings based on Regexp
Method 1: Back References

Two options to extract substrings based on Regexp:

- Use back references
  - Ruby remembers which strings matched the parenthesized parts of a Regexp
  - These parts can be referred to using special variables called back references (named $1, $2,…)
Back Reference Example

gets =~ /^Min:\((\d+)\) Max:\((\d+)\)$/

min, max = $1, $2
puts “mini=#{min} maxi=#{max}”

Input

Min: 1 Max: 27
Min: 10 Max: 30
Min: 11 Max: 30
Min: a Max: 24

Output

mini=1 maxi=27
mini=10 maxi=30
mini= maxi=
mini= maxi=

Extra space messes up match
Not a digit; messes up match
What is the output of the following code?

```ruby
s = "Help I'm stuck in a text editor"
s =~ /([A-Z]+)/
puts $1
```

A. H  
B. Help  
C. I  
D. I’m stuck in a text editor
What is the output of the following code?

```ruby
s = "Help I’m stuck in a text editor"
s =~ /[A-Z]+/;
puts $1
```

A. H  
B. Help  
C. I  
D. I’m stuck in a text editor
Quiz 5

What is the output of the following code?

```
"Why was 6 afraid of 7?" =~ /\d\s(\w+).*(\d)/
puts $1
```

A. afraid
B. Why
C. 6
D. (empty string)
Quiz 5

What is the output of the following code?

```
“Why was 6 afraid of 7?” =~ /\d\s(\w+).*\d\s/ 
puts $1
```

A. afraid  
B. Why  
C. 6  
D. (empty string)
Back References are Local

- Warning
  - Despite their names, $1 etc are local variables
  - (Normally, variables starting with $ are global)

```ruby
def m(s)
  s =~ /(Foo)/
  puts $1 # prints Foo
end
m("Foo")
puts $1    # prints nil
```
Back References are Reset

- Warning #2
  - If another search is performed, all back references are reset to nil

```
gets =~ /\(h\)e(ll)o/ gets =~ /h(e)llo/ gets =~ /hello/ puts $1 puts $2 puts $1
```

<table>
<thead>
<tr>
<th>hello</th>
<th>hello</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>e</td>
</tr>
<tr>
<td>ll</td>
<td>nil</td>
</tr>
<tr>
<td>hello</td>
<td>nil</td>
</tr>
</tbody>
</table>

CMSC 330 Summer 2021
Method 2: String.scan

- Also extracts substrings when matching a Regexp
  - Can optionally use parentheses in Regexp to affect how the extraction is done

- Has two forms that differ in what Ruby does with the matched substrings
  - The first form returns an array
  - The second form uses a code block
    - We’ll see this later
First Form of the Scan Method

- `str.scan(regexp)`
  - If `regexp` does not contain any parenthesized subparts, returns an array of matches
    - An array of all the substrings of `str` which matched

```ruby
s = "CMSC 330 Spring 2021"
s.scan(/\S+ \S+/)
# returns array ["CMSC 330", "Spring 2021"]
```

```ruby
s.scan(/\S{2}/)
# => ["CM", "SC", "33", "Sp", "ri", "ng", "20", "21"]
```
First Form of the Scan Method (cont.)

- **str.scan**(regexp)
  - If regexp does contain parenthesized subparts, returns an array of arrays
    - Each sub-array contains the parts of the string which matched one occurrence of the search
      - `s = "CMSC 330 Spring 2021"
        
        s.scan(/([^S]+) ([S]+)/) # ["CMSC", "330"],
        # ["Spring", "2021"]

    - Each sub-array has the same number of entries as the number of parenthesized subparts
    - All strings that matched the first part of the search (or $1 in back-reference terms) are located in the first position of each sub-array
Practice with Scan and Back-references

> ls -l

- drwx------ 2 sorelle sorelle 4096 Feb 18 18:05 bin
- -rw------- 1 sorelle sorelle 674 Jun 1 15:27 calendar
- drwx------ 3 sorelle sorelle 4096 May 11 2006 cmsc311
- drwx------ 2 sorelle sorelle 4096 Jun 4 17:31 cmsc330
- drwx------ 1 sorelle sorelle 4096 May 30 19:19 cmsc630
- drwx------ 1 sorelle sorelle 4096 May 30 19:20 cmsc631

Extract just the file or directory name from a line using

- scan

  \[
  \text{name} = \text{line} . \text{scan} (\text{/S+$/}) \quad \# \quad \text{["bin"]}
  \]

- back-references

  \[
  \text{if} \ \text{line} \ \text{=~} \ (\text{/S+$/})/ \\
  \quad \text{name} = \$1 \quad \# \quad \text{"bin"} \\
  \text{end}
  \]
What is the output of the following code?

```ruby
s = "Hello World"
t = s.scan(/\w\{2}\/).length
puts t
```

A. 3
B. 4
C. 5
D. 6
Quiz 6

What is the output of the following code?

```ruby
s = "Hello World"
t = s.scan(/ \w{2} /).length
puts t
```

A. 3
B. 4
C. 5
D. 6
What is the output of the following code?

```ruby
s = "To be, or not to be!"
a = s.scan(/(\S+) (\S+)/)
puts a.inspect
```

A. ["To", "be,", "or", "not", "to", "be!"]
B. [["To", "be,"], ["or", "not"], ["to", "be!"]]
C. ["To", "be,"]
D. ["to", "be!"]
What is the output of the following code?

```ruby
s = "To be, or not to be!"
a = s.scan(/(\S+) (\S+)/)
puts a.inspect
```

A. `["To", "be", ",", "or", ",", "not", ",", "to", ",", "be!" ]`
B. `[["To", "be," ], ["or", ",", "not" ], ["to", ",", "be!" ] ]`
C. `["To", ",", "be," ]`
D. `["to", ",", "be!" ]`
Second Form of the Scan Method

- Can take a **code block** as an optional argument

- `str.scan(regexp) { |match| block }`
  - Applies the code block to each match
  - Short for `str.scan(regexp).each { |match| block }`
  - The regular expression can also contain parenthesized subparts
Example of Second Form of Scan

Sums up three columns of numbers

```
12 34 23
19 77 87
11 98 3
2 45 0
```

Example of Second Form of Scan

```
sum_a = sum_b = sum_c = 0
while (line = gets)
    line.scan(/(\d+)\s+(\d+)\s+(\d+)/) { |a,b,c|
        sum_a += a.to_i
        sum_b += b.to_i
        sum_c += c.to_i
    }
end
printf("Total: %d %d %d\n", sum_a, sum_b, sum_c)
```

input file:
will be read line by line, but
column summation is desired

converts the string
to an integer
Practice: Amino Acid counting in DNA

Write a function that will take a filename and read through that file counting the number of times each group of three letters appears so these numbers can be accessed from a hash.

(assume: the number of chars per line is a multiple of 3)

gcggcattcagcaccgtataactgttaagcaatccagatgtgtgtataacatcacciaccggc
catactgaagcattcattgaggctagcggctgataaacaagtagcgttaacaatggggtggtgtgtgtgtataacataacacacccgtaagagtggggagcgttgg
tggcaataacgggtgattctactaagagccggggaccacacacacacaccccgtaaggatggagacgttgg
taacataataatccgttcaagcagtggtggcgaaggtggagatgttccagtaagaatagttgg
gggcctactaccatgttgctacaattagagatcgtcaatctgtgagacggtcaatggtgtc
cgagactatatateactcactccggagcgtatggtcgttactgtgacccctcgttactgtacgga
def countaa(filename)
    file = File.new(filename, "r")
    lines = file.readlines
    hash = Hash.new
    lines.each { |line|
        acids = line.scan(/.../)
        acids.each { |aa|
            if hash[aa] == nil
                hash[aa] = 1
            else
                hash[aa] += 1
            end
        }
    }
end

initialize the hash, or you will get an error when trying to index into an array with a string.

get the file handle.

array of lines from the file.

for each line in the file.

for each triplet in the line.

get an array of triplets in the line.

get the file handle.