To try with a neighbor

Write a Ruby function to print all even numbers from 1 to some given value x.
Write a Ruby function to print all even numbers from 1 to some given value x.

```ruby
def even(x)
  for i in (1..x)
    if i % 2 == 0
      puts i
    end
  end
end
```
CMSC 330: Organization of Programming Languages

Ruby Regular Expressions
(and other topics)
Reminders

• Project 1 is due next Wednesday
• If you have questions about the project, please use the online forum
• Identify yourself on the forum
Counting to Five (Opinion Survey)

A) 
```ruby
i = 1
while i <= 5 do
  puts i;
  i += 1
end
```

B) 
```ruby
(1..5).each do |i|
  puts i
end
```

C) 
```ruby
for i in (1..5)
  puts i
end
```

D) 
```ruby
1 upto(5) do |i|
  puts i
end
```

Which form do you prefer?

http://www.surveymonkey.com/s/FTRPB8T
Why So Many Constructs?

• Is this a good idea?
• What are the advantages and disadvantages?

“syntactic sugar”
Command-line Arguments

• ARGV: a special global variable; it is an array of strings corresponding to the program’s command-line arguments

Ex: ruby validate test.in

ARGV = [“validate”, “test.in”]

• Extract particular arguments: ARGV[0]
• Iterate over all arguments:

ARGV.each do |opt|
  puts opt
end
Creating Strings in Ruby

- Substitution in double-quoted strings with `{}`
  - course = "330"; msg = "Welcome to #{course}"
  - "It is now #{Time.new}"
  - The contents of `{}` may be an arbitrary expression
  - Can also use single-quote to create strings ‘hi’
    - No expression substitution, fewer escaped characters

- Here-documents
  s = <<END
  This is a long text message
  on multiple lines
  and typing \n is annoying
END
  Can be any text
  no space
  no space
Creating Strings in Ruby (cont’d)

• Ruby also has `printf` and `sprintf`
  – `printf("Hello, %s\n", name);`
  – `sprintf("%d: %s", count, Time.now)`
    • Returns a string

• The `to_s` method returns a `String` representation of a class object
The **String** class has many useful methods

- `s.length`  # length of string
- `s1 == s2`  # “deep” equality (string contents)

```ruby
s = "A line\n"; s.chomp  # returns "A line"
```
- Return new string with s's contents except newline at end of line removed

```ruby
s = "A line\n"; s.chomp!
```
- Destructively removes newline from s

**Convention:** methods ending in `!` modify the object

**Another convention:** methods ending in `?` observe the object

```ruby
"r1\tr2\t\tr4".each("\t") { |rec| puts rec }
```
- Apply code block to each tab-separated substring
**Digression: Deep vs. Shallow Copy**

- Consider the following code
  - Assume an object/reference model like Java or Ruby
    - (Or even two pointers pointing to the same structure)

```
x = "groundhog" ; y = x
```

- Which of these occurs?

**Deep copy**

```
x (reference)  "groundhog" (object)
```

**Shallow copy**

```
x (reference)  "groundhog" (object)
y (reference)  "groundhog" (object)
```

```
x (reference)  "groundhog" (object)
y (reference)  "groundhog" (object)
```
Deep vs. Shallow Copy (cont’d)

• Ruby and Java would both do a shallow copy in this case
• But this Ruby example would cause deep copy:

```ruby
x = "groundhog"
y = String.new(x)
```

• In Java, this is done by implementing the cloneable interface and calling clone()
Deep vs. Shallow Equality

Consider these cases again:

- If we compare \( x \) and \( y \), what is compared?
  - The references, or the contents of the objects they point to?

- If references are compared the first would return false but the second true
- If objects are compared both would return true

\[
x \quad (reference) \quad \rightarrow \quad "groundhog" \quad (object) \\
y \quad (reference) \quad \rightarrow \quad "groundhog"
\]

\[
x \quad (reference) \quad \rightarrow \quad "groundhog" \quad (object) \\
y \quad (reference) \quad \rightarrow \quad "groundhog"
\]
String Equality

• In Java, `x == y` is shallow equality, always
  – Compares references, not string contents

• In Ruby, `x == y` for strings uses deep equality
  – Compares contents, not references
  – `==` is a method that can be overridden in Ruby!
  – To check shallow equality, use the `equal?` method
    • Inherited from the `Object` class

• It’s always important to know whether you’re doing a deep or shallow copy
  – And deep or shallow comparison
Standard Library: String

- "hello".index("l", 0)
  • Return index of the first occurrence of string “l” in “hello”, starting at 0
- "hello".sub("h", "j")
  • Replace first occurrence of "h" by "j" in string (not permanent)
  • Use gsub ("global" sub) to replace all occurrences
- "r1\tr2\t\tr3".split("\t")
  • Return array of substrings delimited by tab

• Consider these three examples again
  - All involve searching in a string for a certain pattern
  - What if we want to find more complicated patterns?
    • Find first occurrence of "a" or "b"
    • Split string at tabs, spaces, and newlines
Regular Expressions

• A way of describing patterns or sets of strings
  – Searching and matching
  – Formally describing strings
    • The symbols (lexemes or tokens) that make up a language

• Common to lots of languages and tools
  – awk, sed, perl, grep, Java, OCaml, C libraries, etc.

• Based on some really elegant theory
  – We’ll see that soon
Example Regular Expressions in Ruby

- `/Ruby/`
  - Matches exactly the string "Ruby"
  - Regular expressions can be delimited by /’s
  - Use \ to escape /’s in regular expressions

- `/((Ruby|OCaml|Java))/`
  - Matches either "Ruby", "OCaml", or "Java"

- `/((Ruby|Regular))` or `/R(uby|egular)/`
  - Matches either "Ruby" or "Regular"
  - Use ()’s for grouping; use \ to escape ()’s
Using Regular Expressions

- Regular expressions are instances of `Regexp`
  - we’ll see use of a `Regexp.new` later
- Basic matching using `=~` method of `String`

```ruby
line = gets               # read line from standard input
if line =~ /Ruby/ then    # returns nil if not found
  puts "Found Ruby"
end

offset = line.index(/(MAX|MIN)/)   # search starting from 0
line.sub(/(Perl|Python)/, "Ruby")  # replace
line.split(/(\t|\n| )/)            # split at tab, space, # newline
```

- Can use regular expressions in index, search, etc.
Using Regular Expressions (cont’d)

• Invert matching using `!~` method of String
  – Matches strings that don't contain an instance of the regular expression
Repetition in Regular Expressions

• /(Ruby)*/
  – {"", "Ruby", "RubyRuby", "RubyRubyRuby", ...}
  – * means zero or more occurrences

• /(Ruby)+/
  – {"Ruby", "RubyRuby", "RubyRubyRuby", ...}
  – + means one or more occurrence
  – so /e+/ is the same as /ee*/
Repetition in Regular Expressions

• /(Ruby)?/  
  – {"", "Ruby"}  
  – ? means *optional*, i.e., zero or one occurrence

• /(Ruby){3}/  
  – {“RubyRubyRuby”, “RubyRubyRubyRuby”, …}  
  – {x} means repeat the search for at least x occurrences

• /(Ruby){3, 5}/  
  – {“RubyRubyRuby”, “RubyRubyRubyRubyRuby”, “RubyRubyRubyRubyRuby”}  
  – {x, y} means repeat the search for at least x occurrences and at most y occurrences
Watch Out for Precedence

- `(Ruby)*` means `{""", "Ruby", "RubyRuby", ...}`
  - But `/Ruby*/` matches `{"Rub", "Ruby", "Rubyyy", ...}`

- In general
  - `* {n}` and `+` bind most tightly
  - Then concatenation (adjacency of regular expressions)
  - Then `|`

- Best to use parentheses to disambiguate
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 ^ is like not and must come first)
- `[/[t\n ]/]`
  - Tab, newline or space
- `/[a-zA-Z_\$][a-zA-Z_\$0-9]*/`
  - Java identifiers ($ escaped...see next slide)
Special Characters

- . any character (except for a newline \n)
- ^ beginning of line
- $ end of line
- \$ just a $
- \d digit, [0-9]
- \s whitespace, [\t\r\n ]
- \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 Character Class Confusions

^ inside character classes: not
outside character classes: beginning of line

[ ] inside regular expressions: character class
outside regular expressions: array
    note: [a-z] does not make a valid array,

( ) inside character classes: literal characters ( )
    /(0..2)/ does not mean 012
outside character classes: used for grouping

- inside character classes: range (e.g., a to z given by [a-z])
outside character classes: dash
Regular Expression Practice

With a neighbor, make regular expressions representing the following ideas:

• All lines beginning with a or b  \(^{/(a|b)/}\)

• All lines containing at least two (only alphabetic) words separated by white-space  \(^{/[a-zA-Z]+\s+[a-zA-Z]+/}\)

• All lines where a and b alternate and appear at least once  \(^{/^((ab)+ a?) | ((ba)+b?)$/}\)

• An expression which would match both of these lines (but not radically different ones):

  CMSC330: Organization of Programming Languages: Fall 2007
  CMSC351: Algorithms: Fall 2007
> 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 12:19 cmsg311

drwx------ 2 sorelle  sorelle  4096 Jun  4 17:31 cmsg330

drwx------ 1 sorelle  sorelle  4096 May 30 19:19 cmsg630

drwx------ 1 sorelle  sorelle  4096 May 30 19:20 cmsg631

What if we want to specify the format of this line exactly?

```
/^\(d\|\-\)\(r\|\-\)\(w\|\-\)\(x\|\-\)\(r\|\-\)\(w\|\-\)\(x\|\-\)
\(\s+\)\(\d+\)\(\s+\)\(\w+\)\(\s+\)\(\w+\)\(\s+\)\(\d+\)\(\s+\)\(\S+\)$/\n```

This is unreadable!
Regular Expression Coding Readability

Instead, we can do each part of the expression separately and then combine them:

```ruby
oneperm_re = '((r|-)(w|-)(x|-))'
permissions_re = '(d|-)' + oneperm_re + '{3}'
month_re = '(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)'
day_re = '\d{1,2}'; time_re = '(:\d{2})'
date_re = month_re + 's+' + day_re + 's+' + time_re
total_re = '\d+'; user_re = '\w+'; group_re = '\w+'
space_re = '\d+'; filename_re = '\S+'

line_re = Regexp.new('^' + permissions_re + 's+' + total_re + 's+' + user_re + 's+' + group_re + 's+' + space_re + 's+' + date_re + 's+' + filename_re + '$')

if line =~ line_re
  puts "found it!"
end
```
Method 1: Back-references

Two options to extract substrings based on R.E.’s:

• Use *back-references*
  
  – Ruby remembers which strings matched the parenthesized parts of r.e.’s
  
  – These parts can be referred to using special variables called back-references (named $1, $2,...)
Back-reference Example

• Extract information from a report

```ruby
gets =~ /^Min: (\d+) Max: (\d+)/
min, max = $1, $2
```

• Warning: Despite their names, $1 etc are local variables

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

sets min = $1 and max = $2
Another Back-reference Example

- Warning 2: If another search is done, all back-references are reset to nil

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

```
hello
h
ll
hello
e
nil
hello
nil
```
Method 2: String.scan

- Also extracts substrings based on regular expressions
- Can optionally use parentheses in regular expression to affect how the extraction is done
- Has two forms which 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* doesn't contain any parenthesized subparts, returns an array of matches
    - An array of all the substrings of *str* which matched

```
s = "CMSC 330 Fall 2007"
s.scan(/\S+ \S+/)
# returns array ["CMSC 330", "Fall 2007"]
```

- Note: these string are chosen sequentially from as yet unmatched portions of the string, so while “330 Fall” *does* match the regular expression above, it is *not* returned since “330” has already been matched by a previous substring.
First Form of the scan Method… part 2

- If `regexp` contains parenthesized subparts, returns an array of arrays
  - Each sub-array contains the parts of the string which matched one occurrence of the search
    ```ruby
    s = "CMSC 330 Fall 2007"
s.scan(/(\S+) (\S+)/) # [["CMSC", "330"],
                         # ["Fall", "2007"]
    ```
  - 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
  \[
  \text{name} = \text{line}.\text{scan}(/\S+/) \quad \# \text{[“bin”]}
  \]

- back-references
  \[
  \text{if line =~ /(\S+)//}
  \]
  \[
  \quad \text{name} = \$1 \quad \# \text{“bin”}
  \]
  \[
  \text{end}
  \]
Standard Library: Array

- Arrays of objects are instances of class `Array`
  - Arrays may be heterogeneous
    ```
    a = [1, "foo", 2.14]
    ```
  - C-like syntax for accessing elements, indexed from 0
    ```
    x = a[0]; a[1] = 37
    ```
- Arrays are *growable*
  - Increase in size automatically as you access elements
    ```
    irb(main):001:0> b = []; b[0] = 0; b[5] = 0; puts b.inspect
    [0, nil, nil, nil, nil, 0]
    ```
  - `[]` is the empty array, same as `Array.new`
Standard Library: Arrays (cont’d)

- Arrays can also shrink
  - Contents shift left when you delete elements
    
    ```python
    a = [1, 2, 3, 4, 5]
    a.delete_at(3)  # delete at position 3; a = [1,2,3,5]
    a.delete(2)    # delete element = 2; a = [1,3,5]
    ```

- Can use arrays to model stacks and queues

  ```python
  a = [1, 2, 3]
  a.push("a")    # a = [1, 2, 3, "a"]
  x = a.pop       # x = "a"
  a.unshift("b") # a = ["b", 1, 2, 3]
  y = a.shift     # y = "b"
  ```

  **note:** push, pop, shift, and unshift all permanently modify the array
Iterating through Arrays

• It's easy to iterate over an array with `while`

```ruby
a = [1,2,3,4,5]
i = 0
while i < a.length
  puts a[i]
i = i + 1
end
```

• Looping through all elements of an array is very common
  – And there’s a better way to do it in Ruby
Iteration and Code Blocks

- The **Array** class also has an **each** method, which takes a code block as an argument.

```ruby
a = [1,2,3,4,5]
a.each { |x| puts x }
```

- The code block is delimited by `{}`'s or `do...end`.
- The parameter name is `x`.
- The body of the code block is `puts x`.

---

**Notes:**
- **Code block**: A block of code that is treated as a single entity.
- **Parameter name**: The variable name passed to the code block.
- **Body**: The actions performed within the code block.
More Examples of Code Blocks

• Sum up the elements of an array

```ruby
a = [1,2,3,4,5]
sum = 0
a.each { |x| sum = sum + x }
printf("sum is %d\n", sum)
```

• Print out each segment of the string as divided up by commas (commas are printed trailing each segment)
  – Can use any delimiter

```ruby
s = "Student, Sally, 099112233, A"
s.each(',,') { |x| puts x }
```

("delimiter" = symbol used to denote boundaries)
Yet More Examples of Code Blocks

3.times { puts "hello"; puts "goodbye" }
5.upto(10) { |x| puts(x + 1) }
[1,2,3,4,5].find { |y| y % 2 == 0 }
[5,4,3].collect { |x| -x }

- n.times runs code block n times
- n.upto(m) runs code block for integers n..m
- a.find returns first element x of array such that the block returns true for x
- a.collect applies block to each element of array and returns new array (a.collect! modifies the original)
Still Another Example of Code Blocks

```ruby
File.open("test.txt", "r") do |f|
  f.readlines.each { |line| puts line }
end
```

- **open** method takes code block with file argument
  - File automatically closed after block executed
- **readlines** reads all lines from a file and returns an array of the lines read
  - Use **each** to iterate
Using Yield to Call Code Blocks

- Any method can be called with a code block. Inside the method, the block is called with `yield`.
- After the code block completes, control returns to the caller after the yield instruction.

```ruby
def countx(x)
    for i in (1..x)
        puts i
        yield
    end
end

countx(4) { puts "foo" }
```

```
1
foo
2
foo
3
foo
4
foo
```
So What are Code Blocks?

- A code block is just a special kind of method
  - \{ |y| x = y + 1; puts x \} is almost the same as
  - def m(y) x = y + 1; puts x end

- The each method takes a code block as an argument
  - This is called higher-order programming
    - In other words, methods take other methods as arguments
    - We’ll see a lot more of this in OCaml

- We’ll see other library classes with each methods
  - And other methods that take code blocks as arguments
  - As we saw, your methods can use code blocks too!
Second Form of the scan Method

• Remember the scan method?
  – Gave back an array of matches
  – Can also take a code block as an 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

```ruby
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