Last lecture

CMSC330

More Ruby

This lecture

- Ruby language
  - Regular expressions
    - Definition & examples
    - Back references
    - Scan
  - Arrays
  - Code blocks
  - Hash
  - File
  - Exceptions

String Operations in Ruby

- Consider `s.index('a',0), s.sub('a','b'), etc`
- All involve searching for a pattern
  - What if we wanted 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
  - perl, grep, Java, OCaml, C libraries, etc.
- Based on some really elegant theory
  - Next lecture

Regular Expression Example

- `/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"
- `/\(Ruby\|Regular\)\|/\(Ruby\|\(Ex\)\)\)/`
  - 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: `=~` method of String

```
line = gets
if line =~ /Ruby/ then
  puts "Found Ruby"
end
```

- Can use regular expressions in index, search, etc.

```
offset = line.index(/MAX\[MIN]/)
line.gsub(/Perl|Python)/, "Ruby") # replace
line.split(/\|\|/) # split at tab, space, newline
```

Repetition in Regular Expressions

- `(Ruby) *`:
  
  - `{`, "Ruby", "RubyRuby", "RubyRubyRuby", ...}
  
  - `*` means zero or more occurrences

- `(Ruby)+`
  
  - `{Ruby`, "Rubyy", "Rubyyy", ...}
  
  - `+` means one or more occurrence
  
  - so `/e+/` is the same as `/ee+/`

- `(Ruby)?`
  
  - `{`, "Ruby`}
  
  - `?` means optional, i.e., zero or one occurrence

Watch out for precedence

- `(Ruby)*` means `"", "Ruby", "RubyRuby", ...`
  
  - But `/Ruby*/` matches `"Rub", "Ruby", "Rubyy", ...`

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

  - Best to use parentheses to disambiguate

Using Regular Expressions

- Invert matching using `!~` method of String

  - Matches strings that don’t contain an instance of the regular expression

Character Classes

- `[/abcd/`
  
  - `{a", "b", "c", "d`}

- `//[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
^  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_]
Back Reference Example

- Extract information from a report
  
  ```ruby
  gets =~ /Min: (?<s>) Max: (?<d>)/
  min, max = $1, $2
  puts min + $1
  and max + $2
  ```

- Warning
  - Despite their names, $1 etc are local variables

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

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

```
I'm in ur fixx lab
testin string merry
```

Another Back Reference Example

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

```
def m(s)
  s =~ /\(he\)ll(o)/
  puts $1
  puts $2
  gets =~ /\(he\)ll(o)/
  puts $1
  puts $2
  gets =~ /hello/
  puts $1
end
```

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

scan form 1

- `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
  - 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.

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

scan form 1

- 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

```
m = "CMSC 330 Fall 2007"
m.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

```
> ls -l
-rw------- 2 corelle corelle 4096 Feb 18 18:05 bin
-rw------- 3 corelle corelle 674 Jun 1 15:27 calendar
```

Practice with Scan and Back-references

```
if line =~ /\S+/#
  name = $1 # "bin"
end
```
Array Standard Library

- 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> b = []; b[0] = 0; b[5] = 0;
    puts b.inspect
    [0, nil, nil, nil, nil, 0]
    - [ ] is the empty array, same as Array.new

Iteration and code blocks

- The Array class also has an **each** method
  - Takes a code block as an argument

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

- code block delimited by { }'s or do..end
- parameter name
- body

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)

Array Standard Library

- Arrays can also **shrink**
  - Contents shift left when you delete elements
    a = [1, 2, 3, 4, 5]
    a.delete_at(3) # delete at position 3;
    a.delete(2)    # delete element = 2;
    a.delete(0)    # a = [1,3,5]
- Can use arrays to model stacks and queues
  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

More code block examples

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

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

Still Another Example of Code Blocks

- 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

```
File.open("test.txt", "r") do |f|
  f.readlines.each { |line| puts line }
end
```
Using **`yield`** to call code blocks

```ruby
def count(x)
  for i in {1..x}
    puts i
    yield
  end
end

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

- 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

---

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

---

**`scan`** form 2

- Remember the **`scan`** method?
  - Executing returns an array of matches
- Can also take a code block as an argument
  ```ruby
  str.scan(regexp) { |match| block }
  ```
  - Applies the code block to each match
  - Short for
  ```ruby
  str.scan(regexp).each { |match| block }
  ```
  - The regular expression can also contain parenthesized subparts

---

**Example of **`scan`** form 2

Sums up three columns of numbers

---

**Standard Library: **`Hash`

- A **`hash`** acts like an associative array
  - Every Ruby object can be used as a hash key, because the **`Object`** class has a **`hash`** method
- Elements are referred to using [ ] like array elements, but **`Hash.new`** is the **`Hash`** constructor
  ```ruby
  italy["population"] = 58103033
  italy["continent"] = "Europe"
  italy[1861] = "independence"
  ```

---

**More Hash**

- **Hash methods**
  - **values** returns array of a hash’s values (in some order)
  - **keys** returns an array of a hash’s keys (in some order)
- Iterating over a hash
  ```ruby
  italy.keys.each {
    |key| puts("key: #{key}, value: #{italy[key]}")
  }
  ```
More Hash

- Convenient syntax for creating literal hashes
  - Use `key => value, ...` to create hash table

```perl
credits = {
    "cmoc330" => 4,
    "cmoc330" => 3,
}
x = credits["cmoc330"] # x now 3
credits["cmoc311"] = 3
```

Standard Library: **File**

- Lots of convenient methods for IO
  - `File.new("file.txt", "rw")` # open for rw access
  - `f.readline` # reads the next line from a file
  - `f.readlines` # returns an array of all file lines
  - `f.eof` # return true if at end of file
  - `f.close` # close file
  - `f << object` # convert object to string and write to file

- `$stdin, $stdout, $stderr` are global variables for standard UNIX IO
  - By default stdin reads from keyboard, and stdout and stderr both write to terminal

- `File` inherits some of these methods from IO

---

Exceptions

- Use `begin...rescue...ensure...`
  - Like `try...catch...finally` in Java

```ruby
begin
  f = File.open("test.txt", "r")
  while !f.eof
    line = f.readline
    puts line
    rescue Exception => e
      puts "Exception: " + e.to_s + ": " + (e.class.name + ":")
      ensure
        f.close
    end
end
```

Command Line Arguments

- Can refer to as predefined global constant `ARGV`

- Example
  - If
    - Invoice `test.rb` as "ruby test.rb a b c"
  - Then
    - `ARGV[0] = "a"
    - `ARGV[1] = "b"
    - `ARGV[2] = "c"

---

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)

    ```
    gaggcattccgaacccgctataagttataaatcactcagattttttagtaatacataccgccatactgtagcttagctataaoagtagatagctagctagttacatcatacaatggggaatgtggcataagtgtgcattactaatagagccgagcacaacaccgcctgaatgggtgactaatcatattcagtcccga tgtggggaagttggagatttcagacagtaagggggctattaaccatggttatataactctgcttac cagtcttggtcacttaaaagaggtcgtagctcacttctggagccgtacatagtctagtctctgcttac
gagagtataactcaactccagctgtagcttagctcacttggtatgtagctacttgccttg
```

---

Practice: Amino Acid counting in DNA

```ruby
def countaa(filename)
  file = File.new(filename, "r")
  lines = file.readlines
  hash = Hash.new
  lines.each do |line|
    acids = line.scan(/\S\S\S/)
    acids.each { |aa|
      if hash[aa] == nil
        hash[aa] = 1
      else
        hash[aa] += 1
      end
    }
  end
  hash
end
```
Ruby Summary

- Interpreted
- Implicit declarations
- Dynamically typed
- Built-in regular expressions
- Easy string manipulation
- Object-oriented
  - Everything (!) is an object
- Code blocks
  - Easy higher-order programming!
  - Get ready for a lot more of this...

Other scripting languages

- Perl and Python are also popular scripting languages
  - Also are interpreted, use implicit declarations and dynamic typing, have easy string manipulation
  - Both include optional “compilation” for speed of loading/executing
- Will look fairly familiar to you after Ruby
  - Lots of the same core ideas
  - All three have their proponents and detractors
  - Use whichever language you personally prefer

Example Python Program

```python
#!/usr/bin/python
import re
list = ("deep", "deed", "duck")
x = re.compile("\$[1,5],[aeiou]")
for i in list:
    if re.match(x, i):
        print i
    else:
        print
```

Discussion Yesterday

Programming Language Experience

Languages Used Before

Discussion Yesterday

Class options

Students

Options

Discussion Yesterday

Expected Hours Outside of Class

Hours

Students
Discussion Yesterday

Reminders

- Discussion tomorrow
  - More Ruby examples
    - Regular expressions and hash
    - Available on schedule
  - Project 1 due next Wednesday
    - You have all the material you need for the project