

Last lecture

CMSC330

More Ruby

- Scripting languages
- Ruby language
 - Implicit variable declarations
 - Many control statements
 - Classes & objects
 - Strings

1

2

This lecture

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



3

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

6

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

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

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

Using Regular Expressions

- Invert matching using **!~** method of **String**
 - Matches strings that **don't** contain an instance of the regular expression



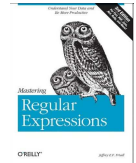
Repetition in Regular Expressions

- / (Ruby) */**
 - { "", "Ruby", "RubyRuby", "RubyRubyRuby", ... }
 - *** means **zero or more occurrences**
- / Ruby+ /**
 - { "Ruby", "Rubyy", "Rubyyyy", ... }
 - +** means **one or more occurrence**
 - so **/e+/** is the same as **/ee*/**
- / (Ruby) ? /**
 - { "", "Ruby" }
 - ?** means **optional**, i.e., zero or one occurrence

9

Repetition in Regular Expressions

- / (Ruby) {3} /**
 - { "RubyRubyRuby" }
 - {x}** means repeat the **search for exactly x occurrences**
- / (Ruby) {3,} /**
 - { "RubyRubyRuby", "RubyRubyRubyRuby", ... }
 - {x,}** means repeat the **search for at least x occurrences**
- / (Ruby) {3, 5} /**
 - { "RubyRubyRuby", "RubyRubyRubyRuby", "RubyRubyRubyRubyRuby" }
 - {x, y}** means repeat the **search for at least x occurrences and at most y occurrences**



10

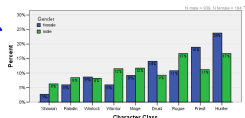
Watch out for precedence

- / (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

11

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)



12

Special Characters

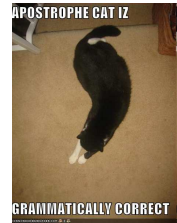
.	any character
^	beginning of line
\$	end of line
\\$	just a \$
\d	digit, [0-9]
\s	whitespace, [\t\r\n\f]
\w	word character, [A-Za-z0-9_]
\D	non-digit, [^0-9]
\S	non-space, [^\t\r\n\f]
\W	non-word, [^A-Za-z0-9_]



13

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 `()`
 - Note `/(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: subtraction



14

Regular Expression Practice

- Make Ruby regular expressions representing
 - All lines beginning with a or b
 - All lines containing at least two (only alphabetic) words separated by white-space
 - All lines where a and b alternate and appear at least once

15

Regular Expression Coding Readability

```
> 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 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
```

- What if we want to specify exactly?

```
/^(d|~)(x|~)(w|~)(x|~)(x|~)(w|~)(x|~)(x|~)(w|~)(x|~)
(\s+)(\d+)(\s+)(\w+)(\s+)(\w+)(\s+)(\d+)(\s+)(Jan|Feb
|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)(\s+)(\d\d)
(\s+)(\d\d:\d\d)(\s+)(\s+)$
```

This is illegible!

16

Regular Expression Coding Readability

- Write out separately and combine

```
oneperm_re = '((x|~)(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}:\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
```

18

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

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

```
sets min = $1
and max = $2
```

- Warning

- Despite their names, `$1` etc are **local** variables

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

19

Another Back Reference Example

- Warning 2

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

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

```
hello
h
ll
hello
e
nil
hello
nil
```

20

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



21

scan form 1



- `str.scan(regex)`

- 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.

22

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

```
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

23

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 name = line.scan(/\S+$/) # ["bin"]
```

- Back references

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

24

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):001:0> b = []; b[0] = 0; b[5] = 0;
puts b.inspect
[0, nil, nil, nil, nil, 0]
```
 - `[]` is the empty array, same as **Array.new**

25

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 = [1,2,3,5]
a.delete(2)    # delete element = 2;
               # 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"
```

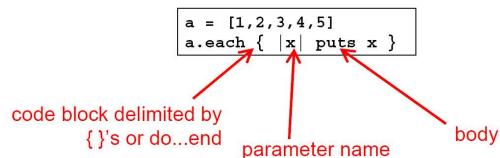
 - note: **push**, **pop**, **shift**, and **unshift** all permanently modify the array



26

Iteration and code blocks

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



27

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 }
```

28

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)

29

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
```

30

Using `yield` to call code blocks

```
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
```

- 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



31

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!

32

`scan` form 2



- Remember the `scan` method?
 - Executing returns an **array** of matches
- Can also take a code block as an argument


```
str.scan(regex) { |match| block }
```

 - Applies the code block to each match
 - Short for


```
str.scan(regex).each { |match| block }
```
 - The regular expression can also contain parenthesized subparts

33

Example of `scan` form 2

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

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

```
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)
```

converts the string
to an integer

Sums up three columns of numbers

34

Standard Library: `Hash`

- A **hash** acts like an **associative array**
 - Elements can be indexed by any kind of values
 - 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


```
- italy["population"] = 58103033
- italy["continent"] = "europe"
- italy[1861] = "independence"
```

35

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


```
- italy.keys.each {
  |key|
    puts("key: #{key}, value: #{italy[key]}")
}
```

36

More Hash

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

```
credits = {
  "cmssc131" => 4,
  "cmssc330" => 3,
}

x = credits["cmssc330"] # x now 3
credits["cmssc111"] = 3
```

37

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 f
- `$stdin`, `$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

38

Exceptions



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

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

Class of exception to catch

Local name for exception

Always happens

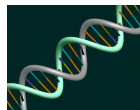
Command Line Arguments

- Can refer to as predefined global constant `ARGV`
- Example
 - If
 - Invoke `test.rb` as `"ruby test.rb a b c"`
 - Then
 - `ARGV[0] = "a"`
 - `ARGV[1] = "b"`
 - `ARGV[2] = "c"`

40

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)
- ```
gcggcattcagcacccgtatactgttaagcaatccagattttgtgtata
acataccggccatactgaagcattcattgaggtagcgtgataacagta
gcgctaacaatgggggaatgtggcaatcgggtcgattactaagagccgg
gaccacacaccccgtaaggtgagcgtggttaacataataatccgttcaa
gcagtgggcgaaggtggagatgtccagtaagaatagtggggcctacta
ccatggtacataattaagagatcgtcaatcttgagacggtcaatggtac
cgagactatatactcaactccggacgtatgcgttactggtcacctcgt
tactgacgga
```



41

## Practice: Amino Acid counting in DNA

```
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
```

get the file handle

array of lines from the file

for each line in the file

for each triplet in the line

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

get an array of triplets in the line

42

## 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...

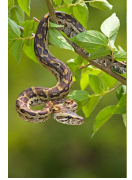
Makes it  
quick to  
write small  
programs

Hallmark of  
scripting  
languages

43

## 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/execution
- 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



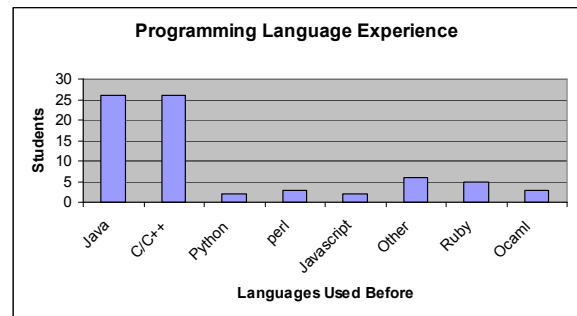
44

## Example Python Program

```
#!/usr/bin/python
import re
list = ("deep", "deer", "duck")
x = re.compile("^s{3,5}.[aeiou]")
for i in list:
 if re.match(x, i):
 print i
 else:
 print
```

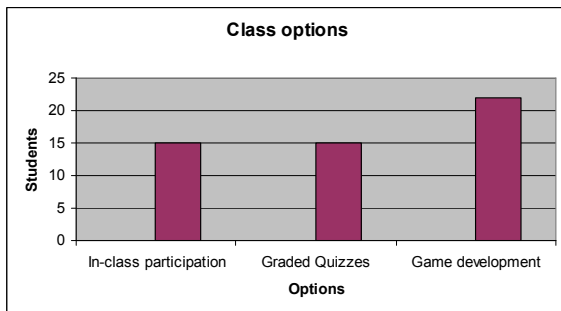
45

## Discussion Yesterday



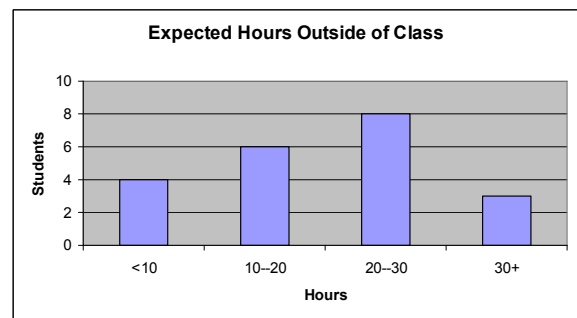
46

## Discussion Yesterday



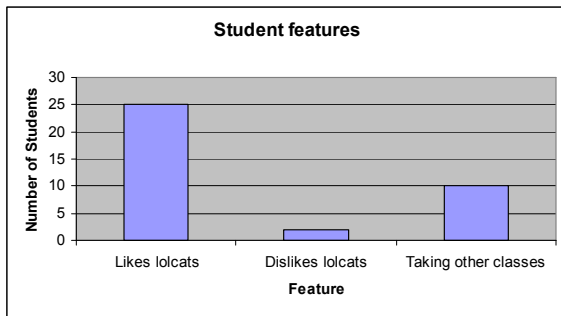
47

## Discussion Yesterday



48

## Discussion Yesterday



49

## 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

50