Introduction

• Ruby is an object-oriented, imperative scripting language
  – “I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That’s why I decided to design my own language.”
  – “I believe people want to express themselves when they program. They don’t want to fight with the language. Programming languages must feel natural to programmers. I tried to make people enjoy programming and concentrate on the fun and creative part of programming when they use Ruby.”
  – Yukihiro Matsumoto (“Matz”)

Books on Ruby

– Earlier version of Thomas book available on web
  • See course web page

Applications of Scripting Languages

• Scripting languages have many uses
  – Automating system administration
  – Automating user tasks
  – Quick-and-dirty development

• Major application: Text processing
Output from Command-Line Tool

```bash
% wc *
271 674 5323 AST.c
100 392 3219 AST.h
117 1459 238788 AST.o
1874 5428 47461 AST_defs.c
1375 6307 53667 AST_defs.h
371 884 9483 AST_parent.c
810 2328 24589 AST_print.c
640 3070 33530 AST_types.h
285 846 7081 AST_utils.c
59 274 2154 AST_utils.h
50 400 28756 AST_utils.o
866 2757 25873 Makefile
270 725 5578 Makefile.am
866 2743 27320 Makefile.in
38 175 1154 alloca.c
2035 4516 47721 aloctypes.c
86 350 3286 aloctypes.h
104 1051 68446 aloctypes.o
```

Climate Data for IAD in August, 2005

<table>
<thead>
<tr>
<th></th>
<th>AVG</th>
<th>MIN</th>
<th>AVG DEP</th>
<th>HDD</th>
<th>CDD</th>
<th>WTR</th>
<th>SNW</th>
<th>DPTH</th>
<th>SPD</th>
<th>SPD DIR</th>
<th>MIN</th>
<th>S-S</th>
<th>WX</th>
<th>SPD DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>87</td>
<td>66</td>
<td>12</td>
<td>0.00</td>
<td>0.0</td>
<td>2.5</td>
<td>9</td>
<td>200</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>18</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>92</td>
<td>67</td>
<td>8</td>
<td>0.00</td>
<td>0.0</td>
<td>3.5</td>
<td>10</td>
<td>10</td>
<td>17</td>
<td>320</td>
<td>3</td>
<td>18</td>
<td>17320</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>93</td>
<td>69</td>
<td>15</td>
<td>0.00</td>
<td>0.0</td>
<td>4.1</td>
<td>13</td>
<td>360</td>
<td>2</td>
<td>17</td>
<td>18</td>
<td>17</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>95</td>
<td>69</td>
<td>17</td>
<td>0.00</td>
<td>0.0</td>
<td>3.6</td>
<td>9</td>
<td>310</td>
<td>3</td>
<td>18</td>
<td>12</td>
<td>290</td>
<td>25360</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>94</td>
<td>73</td>
<td>19</td>
<td>0.00</td>
<td>0.0</td>
<td>5.9</td>
<td>9</td>
<td>10</td>
<td>3</td>
<td>18</td>
<td>3</td>
<td>25</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>89</td>
<td>70</td>
<td>15</td>
<td>0.00</td>
<td>0.0</td>
<td>5.3</td>
<td>20</td>
<td>200</td>
<td>6</td>
<td>17</td>
<td>6</td>
<td>138</td>
<td>23210</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>89</td>
<td>79</td>
<td>14</td>
<td>0.00</td>
<td>0.0</td>
<td>3.6</td>
<td>14</td>
<td>200</td>
<td>7</td>
<td>16</td>
<td>6</td>
<td>17</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>86</td>
<td>70</td>
<td>7</td>
<td>0.00</td>
<td>0.0</td>
<td>4.4</td>
<td>17</td>
<td>150</td>
<td>9</td>
<td>18</td>
<td>10</td>
<td>13</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>76</td>
<td>70</td>
<td>8</td>
<td>0.00</td>
<td>0.0</td>
<td>4.1</td>
<td>9</td>
<td>90</td>
<td>9</td>
<td>18</td>
<td>10</td>
<td>13</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>87</td>
<td>71</td>
<td>4</td>
<td>0.00</td>
<td>0.0</td>
<td>2.3</td>
<td>8</td>
<td>260</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>210</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Raw Census 2000 Data for DC

```
```
A Simple Example

- Let's start with a simple Ruby program

```
ruby1.rb:  # This is a ruby program
```

```
x = 37
y = x + 5
print(y)
print("\n")
```

```
% ruby -w ruby1.rb
42
```

Language Basics

- Comments begin with #, go to end of line
- Variables need not be declared
- Line break separates expressions (can also use ";" to be safe)
- No special main() function or method

```
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

Run Ruby, Run

- There are three ways to run a Ruby program
  - `ruby -w filename` — execute script in `filename`
    - Tip: The `-w` will cause Ruby to print a bit more if something bad happens
  - `irb` — launch interactive Ruby shell
    - Can type in Ruby programs one line at a time, and watch as each line is executed

```
irb(main):001:0> 3+4
=> 7
irb(main):002:0> print("hello\n")
hello
=> nil
```

Run Ruby, Run (cont’d)

- Suppose you want to run a Ruby script as if it were an executable

```
#!/usr/local/bin/ruby -w
print("Hello, world!\n")
```

- Explicit vs. Implicit Declarations

- Java and C/C++ use explicit variable declarations
  - Variables are named and typed before they are used
    - `int x, y; x = 37; y = x + 5;`

- In Ruby, variables are implicitly declared
  - First use of a variable declares it and determines type
    - `x = 37; y = x + 5;`
      - `x, y` exist, will be integers
Tradeoffs?

<table>
<thead>
<tr>
<th>Explicit Declarations</th>
<th>Implicit Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher overhead</td>
<td>Lower overhead</td>
</tr>
<tr>
<td>Helps prevent typos</td>
<td>Easy to mistype variable name</td>
</tr>
<tr>
<td>Forces programmer to document types</td>
<td>Figures out types of variables automatically</td>
</tr>
</tbody>
</table>

Methods in Ruby

Methods are declared with def...end
- List parameters at definition
- May omit parens on call
- Invoke method

```ruby
def sayN(message, n)
i = 0
while i < n
  puts message
  i = i + 1
end
return i
end
```

x = sayN("hello", 3)
puts(x)

(Methods must begin with lowercase letter and be defined before they are called)

More Control Statements in Ruby

- A control statement is one that affects which instruction is executed next
  - We’ve seen two so far in Ruby
    - while and function call
  - Ruby also has conditionals

```ruby
if grade >= 90 then
  puts "You got an A"
elsif grade >= 80 then
  puts "You got a B"
elsif grade >= 70 then
  puts "You got a C"
else
  puts "You’re not doing so well"
end
```

(Methods must begin with lowercase letter and be defined before they are called)
What is True?

• The guard of a conditional is the expression that determines which branch is taken

```ruby
if grade >= 90 then ...
```

Guard

• The true branch is taken if the guard evaluates to anything except
  – false
  – nil
• Warning to C programmers: 0 is not false!

Yet More Control Statements in Ruby

• unless cond then stmt-f else stmt-t end
  – Same as “if not cond then stmt-t else stmt-f end”

```ruby
unless grade < 90 then
  puts "You got an A"
else unless grade < 80 then
  puts "You got a B"
end
```

• until cond body end
  – Same as “while not cond body end”

```ruby
until i >= n
  puts message
  i = i + 1
end
```

Even More Control Statements in Ruby

• Can write if and unless after an expression
  – puts "You got an A" if grade >= 90
  – puts "You got an A" unless grade < 90
• Case is a multi-way branch

```ruby
case grade
  when 90..100
    puts "You got an A"
  when 80..89
    puts "You got a B"
  when 70..79
    puts "You got a C"
  else
    puts "You failed"
end
```

Why So Many Conditionals?

• Is this a good idea?
• Advantages? Disadvantages?
Looping with while

- Basic loop construct is `while..end`

```
i = 0
while i < 5
  puts i.to_s
  i = i + 1
end
```

- Inside of while
  - `break` exits the while loop
  - `next` jumps to the next iteration of the loop
  - `redo` “restarts” the current iteration
    - i.e., jumps back to the top of the loop

Other Looping Constructs

- Ruby also has “for”
  - Though it’s just syntactic sugar, as we’ll see later

```
for elt in [1, "math", 3.4]
  puts elt.to_s
end
```

```
for i in 1..3
  puts i
end
```

Classes and Objects

- Class names begin with an uppercase letter
- The “new” method creates an object
  - `s = String.new` creates a new `String` and makes `s` refer to it
- Every class inherits from `Object`

Everything is an Object

- In Ruby, everything is in fact an object
  - `(-4).abs`
    - integers are instances of `Fixnum`
  - `3 + 4`
    - infix notation for “invoke the + method of 3 on argument 4”
  - "programming".length
    - strings are instances of `String`
  - `String.new`
    - classes are objects with a `new` method
  - `(4.13).class`
    - use the `class` method to get the class for an object
    - floating point numbers are instances of `Float`
Objects and Classes

• Objects are data
• Classes are types (the kind of data which things are)
• But in Ruby, classes themselves are objects!

<table>
<thead>
<tr>
<th>Object</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Fixnum</td>
</tr>
<tr>
<td>-3.30</td>
<td>Float</td>
</tr>
<tr>
<td>&quot;CMSC 330&quot;</td>
<td>String</td>
</tr>
<tr>
<td>String.new</td>
<td>String</td>
</tr>
<tr>
<td>Fixnum</td>
<td>Class</td>
</tr>
<tr>
<td>String</td>
<td>Class</td>
</tr>
</tbody>
</table>

• Fixnum, Float, String, etc., (including Class), are objects of type Class

Two Cool Things to Do with Classes

• Since classes are objects, you can manipulate them however you like
  – if p then x = String else x = Time end  # Time is another class
    y = x.new  # creates a String or a Time, depending upon p
• You can get names of all the methods of a class
  – Object.methods
    • => ["send", "name", "class_eval", "object_id", "new", "autoload?", "singleton_methods", ... ]

The nil Object

• Ruby uses a special object nil
  – All uninitialized fields set to nil (@ refers to a class field)
    ```ruby
    irb(main):004:0> @x
    => nil
    ```
  – Like NULL or 0 in C/C++ and null in Java
• nil is an object of class NilClass
  – It’s a singleton object – there is only one instance of it
    • NilClass does not have a new method
  – nil has methods like to_s, but not other methods that don’t make sense
    ```ruby
    irb(main):006:0> @x + 2
    NoMethodError: undefined method `+' for nil:NilClass
    ```

What is a Program?

• In C/C++, a program is...
  – A collection of declarations and definitions
  – With a distinguished function definition
    • int main(int argc, char *argv[]) { ... }
  – When you run a C/C++ program, it’s like the OS calls main(...)
• In Java, a program is...
  – A collection of class definitions
  – With a class Cl that contains a method
    • public static void main(String[] args)
  – When you run java Cl, the main method of class Cl is invoked
A Ruby Program is...

• The class **Object**
  – When the class is loaded, any expressions not in method bodies are executed

```ruby
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end
```

*defines a method of Object*

```ruby
x = sayN("hello", 3)
puts(x)
```

*invokes self.sayN (part of Object)*

Ruby is Dynamically Typed

• Recall we don’t declare types of variables
  – But Ruby does keep track of types at run time
    ```ruby
    x = 3; x.foo
    NoMethodError: undefined method 'foo' for 3:Fixnum
    ```

• We say that Ruby is **dynamically typed**
  – Types are determined and checked at run time

• Compare to C, which is **statically typed**

<table>
<thead>
<tr>
<th>Ruby</th>
<th>/* C */</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 3</code></td>
<td><code>int x;</code></td>
</tr>
<tr>
<td><code>x = &quot;foo&quot;</code></td>
<td><code>x = 3;</code></td>
</tr>
</tbody>
</table>
| `# gives x a    | `x = "foo"; /* not allowed */`
| `# new type`    |         |

Types in Java and C++

• Are Java and C++ statically or dynamically typed?
  – A little of both
  – Many things are checked statically
    ```java
    Object x = new Object();
    x.println("hello"); // No such method error at compile time
    ```

  – But other things are checked dynamically
    ```java
    Object o = new Object();
    String s = (String) o; // No compiler warning, fails at run time
    // (Some Java compilers may be smart enough to warn about above cast)
    ```

Tradeoffs?

<table>
<thead>
<tr>
<th>Static types</th>
<th>Dynamic types</th>
</tr>
</thead>
<tbody>
<tr>
<td>More work to do when writing code</td>
<td>Less work when writing code</td>
</tr>
<tr>
<td>Helps prevent some subtle errors</td>
<td>Can use objects incorrectly and not realize until execution</td>
</tr>
<tr>
<td>Fewer programs type check</td>
<td>More programs type check</td>
</tr>
</tbody>
</table>
Classes and Objects in Ruby

```ruby
class Point
  def initialize(x, y)
    @x = x
    @y = y
  end
  def addX(x)
    @x += x
  end
  def to_s
    return "(" + @x.to_s + "," + @y.to_s + ")"
  end
end
p = Point.new(3, 4)
p.addX(4)
puts(p.to_s)
```

- Recall classes begin with an uppercase letter
- `inspect` converts any instance to a string
  ```ruby
  irb(main):033:0> p.inspect
  => "<Point:0x54574 @y=4, @x=7>"
  ```
- Instance variables are prefixed with `@`
  - Cannot be accessed outside of class
- The `to_s` method can be invoked implicitly
  - Could have written `puts(p)`
    - Like Java’s `toString()` methods

Inheritance

```ruby
class A
  def add(x)
    return x + 1
  end
end
class B < A
  def add(y)
    return (super(y) + 1)
  end
b = B.new
puts(b.add(3))
```

- Recall that every class inherits from `Object`
- `extend superclass`
- `invoke add method of parent`

Global Variables in Ruby

```ruby
class Global
  @@x = 0
  def Global.inc
    @@x = @@x + 1; $x = $x + 1
  end
def Global.get
    return @@x
  end
end
$y = 0
Global.inc
$y = $y + 1
Global.inc
puts(Global.get)
puts($x)
```

- Ruby has two kinds of global variables
  - Class variables beginning with `@@`
  - Global variables across classes beginning with `$`
- Define a class ("singleton") method
Special Global Variables

- Ruby has a bunch of global variables that are implicitly set by methods
- The most insidious one: `$_`
  - Default method return, argument in many cases
- Example:
  ```ruby
  gets  # implicitly reads input into $_
  print # implicitly writes $_
  ```
- Using `$_` leads to shorter programs
  - And confusion
  - It's suggested you avoid using it

Creating Strings in Ruby

- Substitution in double-quoted strings with `#{}`
  - `course = "330"; msg = "Welcome to #{course}"`
  - "It is now #{Time.now}"`
  - The contents of `#{}` may be an arbitrary expression
  - Can also use single-quote as delimiter
    - No expression substitution, fewer escaping characters
- Here-documents
  ```ruby
  s = <<END
  This is a long text message
  on multiple lines
  and typing \n is annoying
  END
  ```

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

Standard Library: String

- The `String` class has many useful methods
  - `s.length` # length of string
  - `s1 == s2` # “deep” equality (string contents)
  - `s = "A line\n"; s.chomp` # returns "A line"
    - Return new string with s's contents except newline at end of line removed
  - `s = "A line\n"; s.chomp!` # destructively removes newline from s
  - `"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?

<table>
<thead>
<tr>
<th>Deep copy</th>
<th>Shallow copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>x (reference)</td>
<td>&quot;groundhog&quot; (object)</td>
</tr>
<tr>
<td>y (reference)</td>
<td>&quot;groundhog&quot; (object)</td>
</tr>
</tbody>
</table>

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:

\[
x = "groundhog"
y = String.new(x)
\]

- Note: In Java, \texttt{new String(x)} is probably not useful; in Ruby, \texttt{String.new} might be. Why?

Deep vs. Shallow Equality

- Consider these cases again:

<table>
<thead>
<tr>
<th>Deep copy</th>
<th>Shallow copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>x (reference)</td>
<td>&quot;groundhog&quot; (object)</td>
</tr>
<tr>
<td>y (reference)</td>
<td>&quot;groundhog&quot; (object)</td>
</tr>
</tbody>
</table>

- 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

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 \texttt{equal?} method
    - Inherited from the \texttt{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 (cont’d)

- "hello".index("l", 0)
  - Return index of the first occurrence of string in s, starting at n
- "hello".sub("h", "j")
  - Replace first occurrence of "h" by "j" in string
  - Use gsub ("global" sub) to replace all occurrences
- "r1\tr2\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

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
  - But you won’t often use its methods
• 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
```

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

```ruby
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 (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", "Rubyy", "Rubyyy", ...}
  – + means one or more occurrence
  – so /e+/ is the same as /ee*/
• /(Ruby)?/
  – {"", "Ruby"}
  – ? means optional, i.e., zero or one occurrence

More Repetition

• /(Ruby){3}/
  – {"RubyRubyRuby"}
  – {n} means exactly n occurrences
• /(Ruby){3,}/
  – {"RubyRubyRuby", "RubyRubyRubyRuby", ...}
  – {n,} means n or more occurrences
• /(Ruby){2,4}/
  – {"RubyRuby", "RubyRubyRuby", "RubyRubyRubyRuby"}
  – {n,m} means at least n through at most m occurrences
• Do these add any new power to regexps?

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 |
**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
- `/[\t\n ]/`
  - Tab, newline or space
- `/[a-zA-Z_\$][a-zA-Z_\$0-9]*/`
  - Java identifiers ($ escaped...see next slide)

**Special Characters**

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>any character</td>
</tr>
<tr>
<td>^</td>
<td>beginning of line</td>
</tr>
<tr>
<td>$</td>
<td>end of line</td>
</tr>
<tr>
<td>$</td>
<td>just a $</td>
</tr>
<tr>
<td>\d</td>
<td>digit, [0-9]</td>
</tr>
<tr>
<td>\s</td>
<td>whitespace, [\t\n\f]</td>
</tr>
<tr>
<td>\w</td>
<td>word character, [A-Za-z0-9_]</td>
</tr>
<tr>
<td>\D</td>
<td>non-digit, [^0-9]</td>
</tr>
<tr>
<td>\S</td>
<td>non-space, [^\t\n\f]</td>
</tr>
<tr>
<td>\W</td>
<td>non-word, [^A-Za-z0-9_]</td>
</tr>
</tbody>
</table>

**Potential Character Class Confusions**

- `^`
  - Inside char classes: not
  - Outside char classes: beginning of line
- `[]`
  - Inside regexps: character class
  - Outside regexps: Ruby Array
- `( )`
  - Inside char classes: literal characters ( and )
    - Note `/(0..2)/` does not mean {0, 1, 2}
  - Outside char classes: used for grouping
- `-`
  - Inside char classes: range
  - Outside char classes: subtraction

**Regular Expression Practice**

- 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?) $/`
Regular Expression Coding Readability

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

```
> 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 cmisc311
drwx------ 2 sorelle sorelle 4096 Jun  4 17:31 cmisc330
drwx------ 1 sorelle sorelle 4096 May 30 19:19 cmisc360
drwx------ 1 sorelle sorelle 4096 May 30 19:20 cmisc631
```

This is unreadable!

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

```
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 = '[0-9]{1,2}';
time_re = '[0-9]{2}:[0-9]{2}';
date_re = month_re + ' ' + day_re + ' ' + time_re
total_re = '\d+';
user_re = '\w+';
group_re = '\w+';
space_re = '\d+';
filename_re = '\S+';
line_re = Regexp.new('^' + permissions_re + ' ' + total_re + ' ' + user_re + ' ' + group_re + ' ' + space_re + ' ' + date_re + ' ' + filename_re + '$')
```

If line =~ line_re
puts "found it!"
end

Extracting Substrings Based on r.e.'s

- Ruby remembers which strings matched the parenthesized parts of r.e.'s
- These parts can be referred to using special variables called backreferences (named $1, $2,…)
- Examples:
  - `/^\(Min: (\d+) Max: (\d+)$/`
    - Capture digits following “Min” and “Max"

Backreference Example

- Extract information from a report
```
gets =~ /^Min: (\d+) Max: (\d+)/
min, max = $1, $2
```

- **Warning:** Despite their names, $1 etc are local variables
```
def m(s)
s = '/(Foo)/'
p = m("foo")
puts $1   # prints Foo
end
def m(s)
puts $1     # prints nil
```
Another Back Reference Example

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

```
“hello” =~ /(h)e(ll)o/
puts $1  # h
puts $2  # ll
“hello” =~ /h(e)llo/
puts $1  # e
puts $2  # nil
“hello” =~ /hello/
puts $1  # nil
```

Standard Library: Array

• Arrays of objects are instances of class Array
  – Arrays may be heterogeneous
    
    ```ruby
    a = [1, "foo", 2.14]
    ```
  – C-like syntax for accessing elements, indexed from 0
    
    ```ruby
    x = a[0]; a[1] = 37
    ```
  • Arrays are growable
    – Increase in size automatically as you access elements
      
      ```ruby
      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
    
    ```ruby
    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
  
  ```ruby
  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 }
```

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

Yet More Examples of Code Blocks

- `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

```ruby
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 }
```

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

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

```ruby
countx(4) { puts "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
  - Your own methods can also take code block args

Code blocks and the scan Method

- `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

- (There are also some other ways to call `scan`)

Example of Using `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
", sum_a, sum_b, sum_c)
```
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

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

Hash (cont’d)

- The Hash method values returns an array of a hash’s values (in some order)
- And 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]}")
}
```

Hash (cont’d)

Convenient syntax for creating literal hashes

- Use { key => value, ... } to create hash table

```ruby
credits = {
  "cmsc131" => 4,
  "cmsc330" => 3,
}
x = credits["cmsc330"] # x now 3
```

Standard Library: File

- Lots of convenient methods for IO

```ruby
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 # 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...end`
  - Like `try...catch...finally` in Java

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

Command Line Arguments

- Stored in predefined array variable `$*`
  - 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"

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)

```ruby
def countaa(filename)
  file = File.new(filename, "r")
  lines = file.readlines
  hash = Hash.new
  lines.each{ |line|
    acids = line.scan(/.../)  # get the array of triplets in the line
    acids.each{ |aa|  # for each triplet in the line
      if hash[aa] == nil  # initialize the hash, or you will get an error when trying to index into an array with a string
        hash[aa] = 1  # get an array of triplets in the string
      else
        hash[aa] += 1  # for each line in the file
      end
    }
  }
end
```
Considering Ruby Again

- Interpreted
- Implicit declarations
- Dynamically typed
  - These three make it quick to write small programs
- Built-in regular expressions and easy string manipulation
  - This and the three above are the hallmark of scripting languages
- 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/execution
- Will look fairly familiar to you after Ruby
  - Lots of the same core ideas
  - All three have their proponents and detractors
  - Use whichever one you like best