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."
  - Yukihiko 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

Climate Data for IAD in August, 2005
Raw Census 2000 Data for DC

A Simple Example

- Let's start with a simple Ruby program

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

Language Basics

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

Run Ruby, Run (cont'd)

- Suppose you want to run a Ruby script as if it were an executable
  ```ruby
  #!/usr/local/bin/ruby -w
  print("Hello, world!
")
  ```
  - `filename` is run program
    - The first line ("shebang") tells the system where to find the program to interpret this text file
    - Must chmod u+x filename first
      - Or chmod a+x filename so everyone has exec permission
    - Warning: Not very portable
      - Depends on location `/usr/local/bin/ruby`

Explicit vs. Implicit Declarations

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

- In Ruby, variables are implicitly declared
  - First use of a variable declares it and determines type
    ```ruby
    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

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

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

List parameters at definition

May omit parens on call

Invoke method

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

Method (and Function) Terminology

- **Formal parameters**
  - The parameters used in the body of the method
    - `message, n` in our example

- **Actual parameters**
  - The arguments passed in to the method at a call
    - "hello", 3 in our example
  - Note: Some people make this distinction with “parameters” versus “arguments,” but I can never remember which is which...

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
```

What is True?

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

```ruby
if grade >= 90 then
  puts "You got an A"
```

- 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"
  else
    puts "You got a C"
  end
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`

```ruby
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

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

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

Classes and Objects

- Class names begin with an uppercase letter
- The “new” method creates an object
  - `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
  - `String.new`
    - 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></th>
<th>Object</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Fixnum</td>
<td>Class</td>
</tr>
<tr>
<td>3.30</td>
<td>Float</td>
<td>Class</td>
</tr>
<tr>
<td>&quot;CMSC 330&quot;</td>
<td>String</td>
<td>Class</td>
</tr>
<tr>
<td>String.new</td>
<td>String</td>
<td>Class</td>
</tr>
<tr>
<td>String</td>
<td>Class</td>
<td></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` (@ refers to a class field)
  - All uninitialized fields set to `nil` (@x refers to a class field)
  - => 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
    - `inb(main):011:0x @x = 2`  
    - `NoMethodError: undefined method `+` for nil:NilClass`

**What is a Program?**

- In 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
x = sayN("hello", 3)
p = x
```

**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

```ruby
# Ruby
x = 3
x = "foo"  # gives x a new type

# C
int x;
x = 3;
x = "foo"; /* not allowed */
```
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</td>
<td>Less work when writing code</td>
</tr>
<tr>
<td>writing code</td>
<td></td>
</tr>
<tr>
<td>Helps prevent some</td>
<td>Can use objects incorrectly</td>
</tr>
<tr>
<td>subtle errors</td>
<td>and not realize until execution</td>
</tr>
<tr>
<td>Fewer programs type</td>
<td>More programs type check</td>
</tr>
<tr>
<td>check</td>
<td></td>
</tr>
</tbody>
</table>

Classes and Objects in Ruby

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

Classes and Objects in Ruby (cont’d)

- Recall classes begin with an uppercase letter
- `inspect` converts any instance to a string
  ```ruby
  irb(main):033:0> p.inspect
  => "<Point:0x54574 @x=4, @y=7>"
  ```
  - Instance variables are prefixed with @
    - Compare to local variables with no prefix
    - 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

- Recall that every class inherits from `Object`

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

Global Variables in Ruby

- Ruby has two kinds of global variables
  - Class variables beginning with `@@`
  - Global variables across classes beginning with `$`

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

- 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 $_
  puts # 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`
  ```ruby
  printf("Hello, %s\n", name);  
  sprintf("%s: %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
  ```ruby
  - s.length  # length of string
  - s1 == s2   # "deep" equality (string contents)
  ```
  ```ruby
  - s = "A line\n"; s.chomp  # returns "A line"
  ```
  - Returns new string with s's contents except newline at end of line removed
  - 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
  - "r\ttr2\trt4\t.each("\t") { |rec| puts rec }
  ```
  - Apply code block to each tab-separated substring

Digression: Deep vs. Shallow Copy

- Consider the following code
  ```ruby
  x = "groundhog"; y = x
  ```
- (Or even two pointers pointing to the same structure)
- Which of these occurs?

- Deep copy
  ```ruby
  x = "groundhog"; y = x
  ```
- Shallow copy
  ```ruby
  x = "groundhog"; y = String.new(x)
  ```

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)
  ```
- Note: In Java, `new String(x)` is probably not useful; in Ruby, `String.new` might be. Why?
Deep vs. Shallow Equality

- Consider these cases again:
  - If objects are compared both would return true
  - If references are compared the first would return false but the second true
  - If objects are compared both would return true

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

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 (cont’d)

- "hello".index("y", 0)
  - Return index of the first occurrence of string in s, starting at n
- "hello".sub("h", "y")
  - Replace first occurrence of "h" by "y" 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)\(regular)\)/
  - Matches either "Ruby" or "Regular"
  - Use \(\)'s for grouping; use \ to escape \(\)'s

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

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
  if line =~ /Ruby/ then 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

More Repetition

- /Ruby\[3\]/
  - {"RubyRubyRuby"}
  - [n] means exactly n occurrences
- /Ruby\[3,\]/
  - {"RubyRubyRuby", "RubyRubyRubyRuby", ...}
  - [n,m] 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?

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
- /[^0-9]/
  - Tab, newwine or space
- /[a-zA-Z_\$][a-zA-Z_\$0-9\$]/
  - Java identifiers ($ escaped...see next slide)

Replication in Regular Expressions

- /Ruby*/
  - {"", "Ruby", “RubyRuby”, “RubyRubyRuby”, ...}
  - * means zero or more occurrences
- /Ruby+/
  - {"Ruby", “RubyRuby”, “Rubyyyyy”, ...}
  - + 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 |
  - Best to use parentheses to disambiguate

Special Characters

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

- All lines where `a` and `b` alternate and appear at least once
  `- /(ab)+|((ba)+)/$`

Regular Expression Coding Readability

```
> ls -l
drwx------  2 sorelle sorelle 4096 Feb 18 18:05 bin
drwx-------  1 sorelle sorelle 674 Jun 1 15:27 calendar
drwx-------  3 sorelle sorelle 4096 May 11 12:12 cmc311
drwx-------  2 sorelle sorelle 4096 Jun 4 17:31 cmc330
drwx-------  1 sorelle sorelle 4096 May 30 19:19 cmc331
```

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

```
/^\(\d\)-\(\d\)\[\s\][\d\s]+\[\s\][\a-zA-Z]+\[\s\][\d\s]+\[\s\][\d\s]+\[\s\][\d\s]+\[\s\][\d\s]+\[\s\][/J|\s\]\)$
```

This is unreadable!

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:
  - `/"Status: (.*)/`
    - Capture all chars to the right on lines beginning with "Status"
  - `/\(\d\)+ Max: \(\d\)+$/`
    - Capture digits following "Min" and "Max"

Backreference Example

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

- Warning: Despite their names, `$1` etc are local variables
  ```
def m(s)
  a = /\{Foo\}/
  puts $1  # prints Foo and
  m('Foo')
  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)l(l)o/
pus$h $1   # h
pus$h $2   # ll
"hello" =~ /h(e)l(l)o/
pus$h $1   # e
pus$h $2   # nil
"hello" =~ /h(e)l(l)o/
pus$h $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> 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"
```

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| pus$h x }
  ```

Iterating through Arrays

- **It’s easy to iterate over an array with while**
  ```ruby
  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

More Examples of Code Blocks

- **Sum up the elements of an array**
  ```ruby
  a = [1,2,3,4,5]
s = 0
  a.each { |x| s = s + x }
  printf("sum is %d\n", s)
  ```
- **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| pus$h x }
    ```
    ("delimiter" = symbol used to denote boundaries)
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

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

```ocaml
def count x(x)
  for i in 1..x
    puts i
    yield
  end
end
```

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)`

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

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

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

Example of Using scan

Sums up three columns of numbers

```ocaml
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
- Sums up three columns of numbers
**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"
  ```

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

**Convenient syntax for creating literal hashes**

- Use `{ key => value, ... }` to create hash table
  ```
  credits = {
    "cmsc330" => 3,
    "cmsc311" => 4,
  }
  x = credits["cmsc330"] # x now 3
  ```

**Exceptions**

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

```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
```

**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"
  ```

**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` # 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`
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)

gccatgacctccatcactgtaaagatcccatatattggatataatcaccacacccgctgaagatgtgtgcacacatataactggtacacgagaacgctacactacctccagtgcagctgcaaatcctcgtaaccatcgtcagctgtgtgatagctgaagctacactcctacagtacagcttattcctagctgaccatcttcgtaacagtacatccttagtaagcactaatctgctatccagtatatcc

def countAa(filename):
    file = File.new(filename, "r")
    lines = file.readlines
    hash = Hash.new
    for each line in lines:
        acids = line.scan(/. . ./)
        for each acid in acids:
            if hash[acid] == nil
                hash[acid] = 1
            else
                hash[acid] += 1
        end
    end
    return hash

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/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

Example Perl Program

```perl
#!/usr/bin/perl
foreach (split(/, $ARGV[0])) {
    if ($G{$_}) {
        $SRE = "\"" . $G{$_};
    } else {
        $SRE = "$N ? "(?!!)" .
        join("\"", values($G)) . '\}\w' : '\w';
        $G{$_} = ++$N;
    }
}
```

Example Python Program

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