Last Lecture

- Many types of programming languages
  - Imperative, functional, logical, OO, scripting
- Many programming language attributes
  - Clear, orthogonal, natural...
- Programming language implementation
  - Compiled, interpreted

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

```
% wc *
271 674 5323 AST.c
100 392 3219 AST.h
117 1459 23878 AST_defs.c
371 884 9483 AST_defs.h
840 274 25873 Makefile
844 2052 27820 Makefile.am
117 1459 238788 AST.o
285 748 47461 AST_defs.c
1375 6307 53667 AST_defs.h
371 884 9483 AST_utils.c
810 2328 24589 AST_print.c
640 3070 33530 AST_types.h
285 748 47461 AST_utils.c
59 274 2158 AST_utils.h
50 400 28754 AST_utils.o
866 2757 25873 Makefile
866 2757 27820 Makefile.am
117 1459 238788 AST.o
2035 4516 47211 aloctypes.c
86 350 3286 aloctypes.h
104 1051 66848 aloctypes.o
38 175 1154 alloca.c
2035 4516 47211 aloctypes.c
86 350 3286 aloctypes.h
104 1051 66848 aloctypes.o
```

---

Climate Data for IAD in August, 2005

```
1 2 3 4 5 6A 6B 7 8 9 10 11 12 13 14 15 16 17 18
AVG MX 2MIN
DY MX MIN AVG DEP HDD CDD  WTR  SNW DPTH SPD SPD DIR MIN PSBL S
- S WX    SPD DR
```

```
1 87 66 77 1 0 12.00 0.0 0 2.5 9 200 M N 7 18 12 210
2 82 67 80 4 0 15.00 0.0 0 3.5 10 10 M N 3 18 17 320
3 83 69 81 5 0 16.00 0.0 0 4.1 13 360 M N 2 18 17 260
4 95 69 82 6 0 17.00 0.0 0 3.4 9 310 M N 3 18 12 290
5 94 73 84 8 0 19.00 0.0 0 5.9 18 10 M N 3 18 25 260
6 89 70 90 4 0 15.00 0.0 0 5.2 20 250 M N 4 135 23 225
7 89 69 79 3 0 14.00 0.0 0 3.6 14 200 M N 7 18 16 210
8 86 70 78 3 0 13.74 0.0 0 4.4 17 150 M N 10 18 23 150
9 76 70 73 -2 0 8.19 0.0 0 4.1 9 90 M N 5 18 13 90
10 87 71 79 4 0 14.00 0.0 0 2.3 8 260 M N 8 1 10 210
```

---

Raw Census 2000 Data for DC

```
u108_D.C.DO0.01.00000001,572059,72246,572059,12,6,572059,572059,572059,0,0,
G,572059,175004,34229,2250,14622,3821,14261,572059,572059,572059,0,0,
7,340061,1560,1460,28,168,10272,5615,1689,5122,446,157,32,2050,40
89,572059,26827,3362,3048,3170,3241,3504,3284,3270,5475,3939,3647,1755,
3040,2284,2915,2769,2835,2033,4254,5501,5217,3169,1558,4295,2N
16,23726,20721,18802,3523,12318,436,580,3425,4468,7109,5793,3260,2M
7,103232,3239,3078,2078,3249,3259,346,3407,1754,102,392,3036,27628
89,2028,2824,2624,2807,2871,6941,6389,5463,5550,11717,27675,24377,22888
21319,20991,19117,15800,5064,6795,6587,5517,10741,3462,8977,6175,57200
9,536771,370675,15966,55628,6030,57949,123464,112638,3754,3168,22484,
9667,46388,14110,26160,14584,62189,45944,13355,11717,68975,27013,13355,
3586,7273,28113,246508,15689,6499,60875,14021,115963,58020,22954,2
398,55753,10355,8050,6924,67556,22569,25424,15355,15709,70588,287
37,37121,21742,12627,6975,2753,2314,760,24625,80770,7469,73819,18177,
18312,13213,463,54,1881,24930,10985,40213,42747,21440,42601,49
47,87075,3979,4726,3924,25075,14657,105259,58283,22007,80141,21742,1727
61,2121,1546,9966,16938,56,3864,9325,27293,27661,1753,24950,2789
63,4999,32466,24605,24602,18229,12489,75M,1737,113487,11670,32465,12X1
9,521661,14232,12795,15631,4345,268300,116965,47538,28914,19259,1051247
48,3992,132627,108569,32624,2713,1210,599,2184,125
```

---
A Simple Example

Let’s start with a simple Ruby program

```ruby
# 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

Run Ruby, Run

There are several 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
  - Ruby filenames should end with `.rb` extension
- `irb` — launch interactive Ruby shell
  - Can type in Ruby programs one line at a time, and watch as each line is executed
  - Can load Ruby programs via `load` command
    - Form: `load string`
    - String must be name of file containing Ruby program
    - E.g.: `load 'foo.rb`
  - Ruby 1.9.3 is installed on linuxlab, Grace clusters

Run Ruby, Run (cont.)
Run Ruby, Run (cont.)

Suppose you want to run a Ruby script as if it were an executable (e.g. “double-click”, or as a command)
- **Windows**
  - Must associate .rb file extension with ruby command
  - If you installed Ruby using the Windows installer, this was done automatically
  - The Ruby web site has information on how to make this association

Run Ruby, Run (cont.)

Suppose you want to run a Ruby script as if it were an executable (cont.)
- *nix (Linux / Unix / etc.)
  - 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 of Ruby interpreter
    - /usr/local/bin/ruby vs. /usr/bin/ruby vs. /opt/local/bin/ruby etc.

Creating Ruby Programs

- As with most programming languages, Ruby programs are text files.
  - Note: there are actually different versions of “plain text”! E.g. ASCII, Unicode, Utf-8, etc.
  - You won’t need to worry about this in this course.
- To create a Ruby program, you can use your favorite text editor, e.g.
  - notepad++ (free, much better than notepad)
  - emacs (free, infinitely configurable)
  - vim
  - Eclipse (see web page for plugin instructions)
  - Many others

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
  - Ruby allows multi-assignment, too
    - x, y = 37, 5; y = x
    - x, y = 37, x+5 would have failed; x was not yet assigned
Tradeoffs?

<table>
<thead>
<tr>
<th>Explicit Declarations</th>
<th>Implicit Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>More text to type</td>
<td>Less text to type</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>Variable not held to a fixed type (could imagine variable declarations without types)</td>
</tr>
</tbody>
</table>

Methods in Ruby

Methods are declared with def...end

- List parameters at definition
- May omit parens on call
- Invoke method
- Like print, but
  - Adds newline

CMSC 330

Method Return Values

- Value of the return is the value of the last executed statement in the method
  - These are the same:
    ```ruby
    def add_three(x)
      return x+3
    end
    ```

- Methods can return multiple results (as a list)
  ```ruby
  def dep(x)
    return x, x
  end
  ```

CMSC 330

Terminology

- **Formal parameters**
  - Parameters used in the body of the method
  - `def sayN(message, n)` in our example

- **Actual parameters**
  - Arguments passed in to the method at a call
  - `x = sayN("hello", 3)` in our example
Style

- Names of methods that return a boolean should end in `?`.

- Names of methods that modify an object's state should end in `!`.

**Example:** suppose `x = [3,1,2]` (this is an array)
  - `x.member? 3` returns true since 3 is in the array `x`
  - `x.sort` returns a new array that is sorted
  - `x.sort!` modifies `x` in place

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 `method` 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
    ```

Ruby Conditionals Must End!

- All Ruby conditional statements must be terminated with the `end` keyword.

**Examples**

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

- `if grade >= 90 then
  puts "You got an A"
else
  puts "No A, sorry"
end`

What is True?

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

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

- 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"

- `until cond body end`
  - Same as "while not cond body end"

Using If and Unless as Modifiers

- Can write if and unless after an expression
  - `puts "You got an A" if grade >= 90`
  - `puts "You got an A" unless grade < 90`

- Why so many control statements?
  - Is this a good idea? Why or why not?
    - **Good**: can make program more readable, expressing programs more directly. In natural language, many ways to say the same thing, which supports brevity and adds style.
    - **Bad**: many ways to do the same thing may lead to confusion and hurt maintainability (if future programmers don’t understand all styles)

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 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!

Objects | Class (aka type)
---|---
10 | Fixnum
-3.30 | Float
"CMSC 330" | String
String.new | String
[a, b, c] | Array
Fixnum | Class

- Fixnum, Float, and String are objects of type Class
  - So is Class itself!

Two Cool Things to Do with Classes
- Since classes are objects, you can manipulate them however you like
  - Here, the type of y depends on p
    - Either a String or a Time object

- 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 (@ prefix used for fields)
  - 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
    - ::(main):006:0=>nil + 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++ program, it’s like the OS calls main(...)
- In Java, a program is...
  - A collection of class definitions
  - With some class (say, MyClass) containing a method
    - public static void main(String[] args)
  - When you run java MyClass, the main method of class MyClass 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)
puts(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

```
# Ruby
x = 3
x = "foo"  # gives x a # new type
puts(x)
```

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 runtime
    // (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 when coding</td>
<td>Less work when coding</td>
</tr>
<tr>
<td>Helps prevent some subtle errors</td>
<td>Can use objects incorrectly and not discover until run time</td>
</tr>
<tr>
<td>Fewer programs type check</td>
<td>More programs type check</td>
</tr>
</tbody>
</table>
Arrays and Hashes

- Ruby data structures are typically constructed from Arrays and Hashes
  - Built-in syntax for both
  - Each has a rich set of standard library methods
  - They are integrated/used by methods of other classes

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> b = []; b[0] = 0; b[9] = 0; puts b.inspect [0, nil, nil, nil, nil, nil, nil, nil, nil]`
    - `[]` is the empty array, same as `Array.new`

Iterating Through Arrays

- It’s easy to iterate over an array with `while`
  - `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
  - Takes a code block as an argument

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

- We’ll consider code blocks generally a bit later

---

**Ranges**

- `1..3` is an object of class `Range`
  - Integers between 1 and 3 inclusively
- `1...3` also has class `Range`
  - Integers between 1 and 3 but not including 3 itself.
- Not just for integers
  - `'a'..'z'` represents the range of letters ‘a’ to ‘z’
  - `1.3..2.7` is the continuous range `[1.3,2.7)`
- `Ø(1.3…2.7).include? 2.0 ♯ => true`
- Discrete ranges offer the `each` method to iterate
  - And can convert to an array via `to_a`: e.g., `(1..2).to_a

---

**Other Useful Control Statements**

- `case x` when 1, 3.5 when 2, 6.8 end
- `def even(x)`
  - `def even(x)`
  - `def even(x)`
  - `for i in (0..x) x.times { |i| 0.upto(x-1) { |i|`
  - `if i % 2 == 0 if i % 2 == 0 if i % 2 == 0`
  - `puts i puts i puts i`
  - `end end end`
  - `end end end`
**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.)**

- 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 { |k| 
  print "key: ", k, " value: ", italy[k]
}

titaly.each { |k,v|
  print "key: ", k, " value: ", v
}
```

**Hash (cont.)**

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
```

**Defining Your Own Classes**

```ruby
class Point
  def initialize(x, y)
    @x = x
    @y = y
  end
  def add_x(x)
    @x += x
  end
  def to_s
    return "(" + @x.to_s + "," + @y.to_s + ")"
  end
end
p = Point.new(3, 4)
p.add_x(4)
puts(p.to_s)
```
No Access To Internal State

- Instance variables (with @) can be directly accessed only by instance methods
- Outside class, they require **accessors**:
  ```ruby
  def x
    @x
  end
  def x= (value)
    @x = value
  end
  ```
  A typical getter
  A typical setter

- Very common, so Ruby provides a shortcut
  ```ruby
  class ClassWithXandY
    attr_accessor "x", "y"
  end
  ```
  Says to generate the `x=` and `x` and `y=` and `y` methods

No Method Overloading in Ruby

- Thus there can only be one **initialize** method
  - A typical Java class might have two or more constructors
  - You can code your own overloading by using a variable number of arguments, and checking at run-time the number/types of arguments
  - Ruby does issue an exception or warning if a class defines more than one **initialize** method
  - But last `initialize` method defined is the valid one

Classes and Objects in Ruby (cont.)

- 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>
  ```
  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 #< Object
    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))
  ```
  ```ruby
  b.is_a? A true
  b.instance_of? A false
  ```
**super() in Ruby**

- **Within the body of a method**
  - Call to `super()` acts just like a call to that original method
  - Except that search for method body starts in the superclass of the object that was found to contain the original method

---

**Mixins**

- **Another form of code reuse is “mix-in” inclusion**
  - `include A` “inline” A’s methods at that point
  - Referred-to variables/methods captured from context
  - In effect: it adds those methods to the current class

```ruby
class OneDPoint
  attr_accessor "x"
  include Comparable
  def <=>(other)
    # used by Comparable
    if @x < other.x then return -1
    elsif @x > other.x then return 1
    else return 0
    end
  end
end
```

```ruby
p = OneDPoint.new
p.x = 1
q = OneDPoint.new
q.x = 2
x < y    # true
puts [y,x].sort
```

---

**Global Variables in Ruby**

- **Ruby has two kinds of global variables**
  - Class variables beginning with `@@` (static in Java)
  - Global variables across classes beginning with `$`

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

```ruby
$p = OneDPoint.new
$p.x = 1
$q = OneDPoint.new
$q.x = 2
$x < $y  # true
puts [$y,$x].sort
```

```ruby
define a class ("singleton") method
```

---

**Special Global Variables**

- **Ruby has a special set of global variables that are implicitly set by methods**
  - The most insidious one: `$_`
    - Last line of input read by `gets` or `readline`

  ```ruby
  print    # implicitly prints out $_
  puts    # implicitly reads input line into $_
  ```

- **Using `$_` leads to shorter programs**
  - And confusion
  - We suggest 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
  - s = <<END
    This is a text message on multiple lines
    and typing \n is annoying
  END

Creating Strings in Ruby (cont.)

- 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 # structural equality (string contents)
  - s = "A lineln"; s.chomp # returns "A line"
    - Return new string with s's contents except newline at end of line removed
  - s = "A lineln"; s.chomp!
    - Destructively removes newline from s
  - Convention: methods ending in ! modify the object
  - Another convention: methods ending in ? observe the object
  - "r1\nr2\nr3\nr4".each_line { |rec| puts rec }
    - Apply code block to each newline-separated substring

Standard Library: String (cont.)

- "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\nr2\n\tr3\n".split("\r")
  - Return array of substrings delimited by tab
- Consider these 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!
Object Copy vs. Reference Copy

Consider the following code
- Assume an object/reference model like Java or Ruby
  - Or even two pointers pointing to the same structure
    \[ x = \text{"groundhog"}; y = x \]

Which of these occur?

- Object copy
- Reference copy

For
- Ruby and Java would both do a reference copy
  \[ x = \text{"groundhog"}; y = x \]
- But for
  \[ x = \text{"groundhog"} \]
  \[ y = \text{String.new}(x) \]
  - Ruby would cause an object copy
  - Unnecessary in Java since Strings are immutable

Physical vs. Structural Equality

Consider these cases again:
- The references, or the contents of the objects they point to?
- If references are compared (physical equality) the first would return false but the second true
- If objects are compared both would return true

String Equality

- In Java, \( x == y \) is physical equality, always
  - Compares references, not string contents
- In Ruby, \( x == y \) for strings uses structural equality
  - Compares contents, not references
  - \( == \) is a method that can be overridden in Ruby!
  - To check physical equality, use the \texttt{equal?} method
    - Inherited from the \texttt{Object} class
- It’s always important to know whether you’re doing a reference or object copy
  - And physical or structural comparison
### Comparing Equality

<table>
<thead>
<tr>
<th>Language</th>
<th>Physical equality</th>
<th>Structural equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td><code>a == b</code></td>
<td><code>a.equals(b)</code></td>
</tr>
<tr>
<td>C</td>
<td><code>a == b</code></td>
<td><code>*a == *b</code></td>
</tr>
<tr>
<td>Ruby</td>
<td><code>a.equal?(b)</code></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td>Ocaml</td>
<td><code>a == b</code></td>
<td><code>a = b</code></td>
</tr>
<tr>
<td>Python</td>
<td><code>a is b</code></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td>Scheme</td>
<td><code>(eq? a b)</code></td>
<td><code>(equal? a b)</code></td>
</tr>
<tr>
<td>Visual Basic .NET</td>
<td><code>a ls b</code></td>
<td><code>a = b</code></td>
</tr>
</tbody>
</table>

### Summary

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