Basic Programming in Ruby

Today's Topics: [whirlwind overview]

- Introduction
- Fundamental Ruby data types, "operators", methods
- Outputting text
  - print, puts, inspect
- Flow control
  - loops & iterators
  - conditionals
- Basic I/O
  - standard streams & file streams
  - reading & writing
- Intro to regular expression syntax
  - matching
  - substituting
- Writing custom methods (& classes)

Ruby Data Types

Ruby has essentially one data type: *Objects* that respond to *messages* (“methods”)

- all data is an Object
- variables are named locations that store Objects.

Let’s consider the classical “primitive” data types:

- **`Numeric`**: **`Fixnum`** (42) and **`Float`** (42.42, 4.242e1)
- Boolean: **`true`** and **`false`**
  - logically: **`nil`** & **`false`** are both treated as “false”, *all* other objects are considered “true”
- **`String`** (text)
  - double quotes (“ “) means interpolated: “\t#{52.2 * 45}\n\”Hi Mom\””
  - single quotoes (‘ ‘) means non-interpolated: ‘\t#{5*7}’
- **`Range`**
  - end-inclusive, a.k.a. “fully closed range”: 2..19
  - right-end-exclusive, a.k.a. “half-open”: 2…19

Ruby Data Types, “operators”, Methods

Key/special variables (predefined, part of language) :

- **nil** (the no-Object; NULL, undef, none)
  - technically an instance (object) of the class **`NilClass`**
- **self** (the current Object ; important later)

What about the standard bunch of operators?

- sure, but keep in mind these are all actually methods
- **`+, -, *, /, **`**
- **`<, >, <=, >=, ==, !=, <=>`**
- **`nil?()`**
- Other important methods many objects respond to:
  - `to_s() ; to_i() ; to_f() ; size() ; empty?()`
  - `reverse() ; reverse! ; include? ; is_a?`

More Complex, But Standard Data Types

Arrays

- **`anArr = Array.new() ; anArr = []`**
- 0-based indexes to access items in an Array using [] method à anArr[2] = “glycine” ; aA = anArr[2]
- objects of different classes can be stored within the Array
- responds to **`push(), pop(), shift(), unshift()`** methods
  - for adding & removing things from the end or the front of Arrays
- Array merging and subtraction via **+** and **–**
- Deleting items from Arrays via **`delete()`** & `delete_at()`
- Looking for items via `include?`
  - slow for huge arrays obviously, or if we use repeatedly on moderate sized ones

Hashes

- Look up items based on a key ß fast lookup
- `aHash = Hash.new() ; aHash = {} ; aHash = Hash.new {|hh,kk| hh[kk] = [] }`
- key-based access to items using [] method à aHash[“pros1”] = “chr14” ; chrom = aHash[“pros1”]
- key can be *any* object, as can the stored value
- check if there is already something within the array: `key?(“pros2”)`
- number of items stored: `size()`

Some Highlighted Methods

String:

- capitalize vs capitalize!
- downcase, upcase, chop, chomp, next, size, length, index, strip, strip!, reverse, tr, slice or [], split, empty?
- nil?, dup, is_a?, kind_of?, respond_to?, inspect ß inherited from *Object*

"aatCGc".reverse.tr('atcgATCG', 'tagcTAGC')

split() & strip() particularly useful for parsing text-based data files

Array:

- first, last, empty?, length, fill, max, min, max, reverse, transpose, uniq, sort
- pop/push, shift/unshift, << vs +,
- slice or []

Hash:

- key?, value?, include?
- keys, values, size, merge, entries, values_at

Kernel:

- puts, print, sprintf or %
- rand, exit, eval, sleep

Flow control: loops

Loops:

- boringly simple and often not what you need
- `loop { }`
- `while()`
- `break` keyword to end loop prematurely

Iteration:

- more useful…very common to want to iterate over a set of things
- iterator methods take `blocks`…a piece of code the iterator will call at each iteration, passing the current item to your piece of code
  - like a function pointer in C, function name callback in Javascript, anonymous methods in Java, etc
- `times {} ; upto {}`
- `each {} ; each_key {}`
- `each {}` probably the most useful… `Arrays, Strings, File` (`IO`), so many Classes support it
- look for specialty `each_XXX {}` methods like: `each_byte{}, each_index {},` etc

Result: no need for weird constructs like `for(ii=0; ii<arr.size;ii++)` nor `foreach…in…` nor `do-while`

Flow control: conditionals

Conditional Expressions:

- evaluate to **`true`** or to **`false`**
- usually involve a simple method call or a comparison
  - `nil? ; empty? ; include?`
  - `== ; != ; > ; <= ;` … etc…
- combine conditional expressions with boolean logic operators: `or , and , || , &&, !, not`
- remember: only **`nil`** and `false` evaluate to false, all other objects are true
  - `0, “0”, “”` evaluate to true (unlike in some other languages where they double as false values)

Use conditionals for flow control:

- `while() … end`
- `if() … elsif() … else … end`
- `unless() … else … end`
- single line `if()` and `unless()`
- Read about `case` statements, Ruby’s switch statement (a special form of if-elsif-else statements)

Basic I/O: Standard Streams

Reading & Writing: let's discuss the 3 standard I/O streams first:

Generally, 3 standard streams available to all programs: stdout, stderr, stdin

- most often, stdout à screen, stderr à screen, stdin ß data redirected into program
  - stdout and stderr sometimes redirected to files when running programs
- in Ruby, these I/O streams explicitly available via `$stdout, $stderr, $stdin`
- `puts()` ends up doing a `$stdout.puts()`
- explicit `$stderr.puts()` calls can be useful for debugging, program progress updates, etc

Reading:

- `each {} ; each_line{}` ß most useful (iteration again)
- `readline()`

Writing:

- we've been writing via `puts(), print()` already…these Object methods write to the standard output IO stream e.g. $stdout.puts()

Basic I/O: Working With Files

File Objects:

- open for reading à `file = File.open(fileName)`
- open for writing à `file = File.open(fileName, "w")` ß creates a new file or wipes existing file out
- open for appending à `file = File.open(fileName, "a+")`
- `read() ; readline() ; each {} ; each_line {}`
- `print() ; puts()`
- `seek() ; rewind() ; pos()`

Strings as IO:

- what if we have some big `String` in memory and want to treat it as we would a `File`?
- `require 'stringio'`
- `strio = StringIO.new(str)`
- go crazy and use file methods mentioned above
- `newStr = strio.string()` ß covert to regular String object

Interactive Programs:

- generally avoid…when working with and producing big data files, you want to write things that can run without manual intervention
- unless you are writing permanent tools for non-programmers to use (then also consider a GUI)
- `getc() ; gets() ;` *et alia*

Regular Expressions Intro

Regular Expressions are like powerful patterns, applied against `Strings`

Very useful for dealing with text data!

Ruby's regular expression syntax is like that of Perl, more or less. Also there is a pure object-oriented syntax for more complex scenarios or for Java programmers to feel happy about.

Key pattern matching constructs:

- `.` ß must match a single character
- `+` ß must match one or more of preceding thing [+ is 'one or more of preceding']
- `*` ß 0 or more of preceding thing match here [* is 'zero or more of preceding']
- `?` ß 0 or 1 of preceding thing match here [? is 'preceding may or may not be present']
- `[;_-]` ß match 1 of ; or _ or – (in this example)
- `[^;_-]` ß match any 1 character *except* ; or _ or – (in this example; ^ within [] means NOT)
- `[^;_-]+` ß match 1 or more of any character *except* ; or _ or – here
- `^` ß match must start at beginning of a line [$ anchors the end of a line]
- `\A` ß match must start at beginning of whole string [\Z anchors at end of string]
- `\d` ß match a digit [\D match any non-digit; note that \d+ would match 1 or more digits]
- `\w` ß match a word character [\W match any non-word character] [word is alpha-num and _]
- `\s` ß match a whitespace character [\S match any non-whitespace character]
- `foo|bar` ß match foo or bar ['match preceding or the following']

Regular Expressions Intro

Backreferences:

- `()` around any part of pattern will be captured for you to use later
- `/accNum=([^;]+)/`
- The text matched by each `()` is captured in a variable named `$1, $2, $3`, etc
- If the pattern failed to match the string, then your backreference variable will have `nil`

Syntax: (apply regex against a variable storing a String object)

- `aString =~ /some([a-zA-z]+$/` ß returns index where matches or `nil` (`nil` is false…)
- `aString !~ /some([a-zA-z]+$/` ß assert String *doesn't* match; returns `true` or `false`

Uses:

- In conditionals [`if(line =~ /gene|exon/) then geneCount += 1 ; end`]
- In String parsing
- In String alteration (like `s///g` operation in Perl or sed)
  - `gsub() ; gsub!()`

Special Note:

- Perl folks: where is `tr///` ?
- Right here: `newStr = oldStr.tr("aeiou", "UOIEA")`

Writing and Running Ruby Programs

Begin your Ruby code file with this line (in Unix/Linux/OSX command line):

```
#!/usr/bin/env ruby
```

Save your Ruby code file with ".rb" extension

Run your Ruby program like this on the command line:

```
ruby ./<yourRubyFile.rb> # windows may need .\rubycode.rb
```

Or make file executable (Unix/Linux/OSX) via `chmod +x <yourRubyFile.rb>` then:

```
./<yourRubyFile.rb>
```

Comment your Ruby code using "#" character

- Everything after the # is a comment and is ignored

Download and install Ruby here:

http://www.ruby-lang.org

Basic Programming in Ruby

More Help:

1. Try Ruby http://tryruby.hobix.com/
2. Learning Ruby http://www.math.umd.edu/~dcarrera/ruby/0.3/
3. Ruby Basic Tutorial http://www.troubleshooters.com/codecorn/ruby/basictutorial.htm
4. RubyLearning.com http://rubylearning.com/
5. Ruby-Doc.org http://www.ruby-doc.org/
   - Useful for looking up built-in classes & methods
   - E.g. In Google: “rubydoc String”
6. Ruby for Perl Programmers http://migo.sixbit.org/papers/Introduction_to_Ruby/slide-index.html