Ruby Reference Sheet

Administrivia

⇒ Ruby has a interactive command line. In terminal, type `irb`.

⇒ To find your Ruby version, type `ruby --version`; or in a Ruby script:

```
RUBY_VERSION # ⇒  2.3.7

# In general, use backtics ‘···‘ to call shell commands
‘ls -ll ~‘ # ⇒  My home directory’s contents!
```

⇒ Single line comments marked by `#`.
Multi-line comments enclosed by `=begin` and `=end`.

⇒ Newline or semicolon is used to separate expressions; other whitespace is irrelevant.

⇒ Variables don’t have types, values do.

- ◇ *The type of a variable is the class it belongs to.*
- ◇ Variables do not need declarations.

Everything is an object!

Method calls are really message passing: `x ⊕ y` ≈ `x.⊕(y)` ≈ `x.send "⊕" , y`

Methods are also objects: `f x` ≈ `method(:f).call x`

Remember: Use `name.methods` to see the methods a name has access to. Super helpful to discover features!

```
"Hi".class                # ⇒  String
"Hi".method(:class).class # ⇒  Method
"Hi".methods              # ⇒  Displays all methods on a class ♡‿♡
2.methods.include?(:/)    # ⇒  true, 2 has a division method
```

Everything has a value —possibly `nil`.

- ◇ There’s no difference between an expression and a statement!

Functions – Blocks

Multiple ways to define anonymous functions; application can be a number of ways too.

Parenthesises are optional unless there’s ambiguity.

- ◇ The value of the last statement is the ‘return value’.
- ◇ Function application is right-associative.
- ◇ Arguments are passed in with commas.

```
fst = lambda { |x, y| x}
fst.call(1, 2)    # ⇒  1
fst.(1, 2)        # ⇒  1

# Supply one argument at a time.
always7 = fst.curry.(7)
always7.(42)              # ⇒  42

# Expplicitly curried.
fst = lambda {|x| lambda {|y| x}}
fst = ->(x) {->(y) {x}}
fst[10][20]   # ⇒  10
```

```
fst.(100).(200)   # ⇒  100

fst.methods # ⇒  arity, lambda?,
            #   parameters, curry
def sum x, y = 666, with: 0
  x + y + with end

sum (sum 1, 2) , 3 # ⇒  6
sum 1              # ⇒  667
sum 1, 2           # ⇒  3
sum 1, 22, with: 3  # ⇒  6
```

Notice that the use of ‘=’ in an argument list to mark arguments as **optional** with default values. We may use **keyword** arguments, by suffixing a colon with an optional default value to mark the argument as optional; e.g., omitting the `0` after `with:` makes it a necessary (keyword) argument. Such may happen in `|···|` for arguments to blocks.

Convention: Predicate names end in a `?`; destructive function names end in `!`.
That is, methods ending in `!` change a variable’s value.

Higher-order: We use `&` to indicate that an argument is a function.

```
def apply(x, &do_it) if block_given? then do_it.call(x) else x end end
apply (3) { |n| 2 * n }   # ⇒ 6, parens around ‘3’ are needed!
apply 3 do |n| 20 * n end # ⇒ 6
apply 3                   # ⇒ 3
```

In fact, all methods have an implicit, optional block parameter. It can be called with the `yield` keyword.

```
sum(1, 2) do |x| x * 0 end  # ⇒ 3, block is not used in “sum”

def sum’ (x, y) if block_given? then yield(x) + yield(y) else x + y end end
sum’(1, 2)                  # ⇒ 3
sum’(1, 2) do |n| 2 * n end  # ⇒ 6
sum’(1, 2) do end # ⇒ nil + nil, but no addition on nil: CRASHES!
sum’(1, 2) { 7 }  # ⇒ 14; Constanly return 7, ignoring arguments; 7 + 7 ≈ 14
```

Note: A subtle difference between `do/end` and `{/}` is that the latter binds tightly to the closest method; e.g., `puts x.y { z }` ≈ `puts (x.y do z end)`.

Variadic number of arguments:

```
def sum” (*lots_o_stuff) toto = 0; lots_o_stuff.each{ |e| toto += e}; toto end
sum” 2 , 4 , 6 , 7  #⇒ 19

# Turn a list into an argument tuple using “splat”, ‘*’
nums = [2, 4, 6, 7, 8, 9]
# sum” nums  #⇒ Error: Array can’t be coerced into number
sum” *nums.first(4)  #⇒ 19
```

If a name is overloaded as a variable and as a function, then an empty parens must be used when the function is to be invoked.

```
w = "var"
def w; "func" end
"w: #{w}, but w(): #{w()}" # ⇒ w: var, but w(): func
```

How to furnish a single entity with features? *“Singleton methods/classes”!* You can attach methods to existing names whenever you like. (Instance vars are `nil` by default.)

```
x = "ni"
def x.upcase; "Knights who say #{self} × #{@count = (@count || 0) + 1}" end
x.upcase # ⇒ Knights who say ni × 1
x.upcase # ⇒ Knights who say ni × 2

# Other items are unaffected.
"ni".upcase  # ⇒ NI, the usual String capitalisation method
```

In general, the syntax `class « x ··· end` will attach *all usual class contents* “···” *only* for the entity `x`. (Undefined instance variables are always `nil`.)

We can redfine any method; including the one that handles missing method issues.

```
x.speak # ⇒ Error: No method ‘speak’
# Do nothing, yielding ‘nil’, when a method is missing.
def method_missing(id, *args) end
x.speak # ⇒ nil
```

A “ghost method” is the name of the technique to dynamically a create a method by overriding `method_missing`. E.g., by forwarding ghosts `get_x` as calls `get(:x)` *with* extra logic about them.

Operators are syntactic sugar and can be overrided. This includes the arithmetical ones, and `[]`, `[]=`; and unary ± via `+@`, `-@`.

```
def x.-(other); "nice" end
x - "two" # ⇒ "nice"
```

Forming aliases:

```
alias summing sum”
summing 1, 2, 3  # ⇒ 6
```

Methods as Values

Method declarations are expressions: A method definition returns the method’s name as a symbol.

```
def woah; "hello" end # ⇒ :woah

woah’ = method(:woah) # ⇒ #<Method: Object#woah>

woah’.call # ⇒ hello

method(:woah).call # ⇒ hello
```

Notice that using the operation `method` we can obtain the method associated with a symbol.

Likewise, `define_method` takes a name and a block, and ties those together to make a method. It overrides any existing method having that name.

The following is known as "decoration" or "advice"!

Besides decorating a function call to print a trace like below, it can be used to add extra behaviour such as caching expensive calls, mocking entities for testing, or doing a form of typing (Ruby is a Lisp).

```
define_method(:ni) {|x| x}

def notify(method_name)
  orginal = method(method_name)
  define_method(method_name) { |*args, &blk|
    p "#{method_name} running ... got #{orginal.call(*args, &blk)}"} end

notify def no_op (x) x end

no_op 1 # ⇒ no_op running ... got 1

# "x.singleton_class.include(M)" to wholesale attach module M's contents to x.
```

See here for a nifty article on methods in Ruby.

Variables & Assignment

Assignment '=' is right-associative and returns the value of the RHS.

```
# Flexible naming, but cannot use '-' in a name.
this_and_that = 1
𝑢ℕ𝑖ℂ∅𝒟𝐄      = 31

# Three variables x,y,z with value 2.
x = y = z = 2

# Since everything has a value, "y = 2" ⇒ 2
x = 1, y = 2  # Whence, x gets "[1, 2]"!
# Arrays are comma separated values; don't need [ and ]

x = 1; y = 2  # This is sequential assignment.

# If LHS as has many pieces as RHS, then we have simultenous assignment.
x , y = y , x # E.g., this is swap

# Destructuring with "splat" '*'
a , b, *more = [1, 2, 3, 4, 5] # ⇒ a ≈ 1; b ≈ 2; c ≈ [3, 4, 5]
first, *middle, last = [1, 2, 3, 4, 5] # ⇒ first ≈ 1; middle ≈ [2, 3, 4]; last = 5
last

# Without splat, you only get the head element!
a , b, c = [1, 2, 3, 4, 5] # ⇒ a ≈ 1; b ≈ 2; c ≈ 3
```

"Assign if undefined": `x ||= e` yields `x` if it's a defined name, and is `x = e` otherwise. This is useful for having local variables, as in loops or terse function bodies.

```
nope rescue "'nope' is not defined."
# nope ||= 1 # ⇒ nope = 1
# nope ||= 2 # ⇒ 1, since "nope" is defined
```

Notice: `B rescue R` ≈ Perform code `B`, if it crashes perform code `R`.

Strings and %-Notation

Single quotes are for string literals, whereas double quotes are for string evaluation, 'interpolation'. Strings may span multiple lines.

```
you = 12
#  ⇒  12

"Me and \n #{you}"
#  ⇒  Me and ⟨⟨newline here⟩⟩ 12

'Me and \n #{you}'
#  ⇒  Me and \n #{you}

# “to string” and catenation
"hello " + 23.to_s  #  ⇒  hello 23
```

```
# String powers
"hello " * 3
#  ⇒  hello hello hello

# Print with a newline
puts "Bye #{you}"
#  ⇒  Bye 12  ⇒  nil

# printf-style interpolation
"%s or %s" % ["this" , "that"]
it = %w(this that); "%s or %s" % it
```

Strings are essentially arrays of characters, and so array operations work as expected!

There is a Perl-inspired way to quote strings, by using % along with any non-alpha-numeric character acting as the quotation delimiter. Now only the new delimiter needs to be escaped; e.g., " doesn't need escape.

A type modifier can appear after the % : q for strings, r for regexp, i symbol array, w string array, x for shell command, and s symbol. Besides x, s, the rest can be capitalised to allow interpolation.

```
%{ woah "there" #{1 + 2} }  #  ⇒  "woah \"there\" 3"
%w[ woah "there" #{1 + 2} ] #  ⇒  ["woah", "\"there\"", "\#{1", "+", "2}"]
%W[ woah "there" #{1 + 2} ] #  ⇒  ["woah", "\"there\"", "3"]
%i( woah "there" )          #  ⇒  [ :woah, :"there"]
```

See here for more on the %-notation.

Booleans

false, nil are both considered *false*; all else is considered *true*.

- Expected relations: ==, !=, !, &&, ||, <, >, <=, >=
- x <=> y returns 1 if x is larger, 0 if equal, and -1 otherwise.
- “Safe navigation operator”: x&.y ≈ (x && x.y).
- and, or are the usual logical operators but with lower precedence.
- They're used for control flow; e.g., s_0 and s_1 and ⋯ and s_n does each of the s_i until one of them is false.

Since Ruby is a Lisp, it comes with many equality operations; including =~ for regexps.

Arrays

Arrays are heterogeneous, 0-indexed, and [brackets] are optional.

```
array   = [1, "two", :three, [:a, "b", 12]]
again   =  1, "two", :three, [:a, "b", 12]
```

Indexing: x[±i] ≈ “value if i < x.length else nil” x[i] ⇒ The *i-th* element from the start; x[-i] ⇒ *i-th* element from the end.

```
array[1]     #  ⇒  "two"
array[-1][0] #  ⇒  :a
```

Segments and ranges:

x[m, k] ≈ [x_m, x_{m+1}, ..., x_{m+k-1}]
x[m..n] ≈ [x_m, x_{m+1}, ..., x_n] if $m \leq n$ and [] otherwise
x[m...n] ≈ x[m..n-1] —to exclude last value
a[i..j] = r ⇒ a ≈ a[0, i] + *r + a[j, a.length]
Syntactic sugar: x[i] ≈ x.[] i

Where *r is array coercion: Besides splicing, splat is also used to coerce values into arrays; some objects, such as numbers, don't have a to_a method, so this makes up for it.

```
a = *1      #  ⇒  [1]
a = *nil    #  ⇒  []
a = *"Hi"   #  ⇒  ["Hi"]
a = *(1..3) #  ⇒  [1, 2, 3]
a = *[1,2]  #  ⇒  [1, 2]

# Non-symmetric multiplication; x * y ≈ x.*(y)
[1,2,3] * 2    #  ⇒  [1,2,3,1,2,3]
[1,2,3] * "; " #  ⇒  "1; 2; 3"
```

As always, learn more with `array.methods` to see, for example, `first, last, reverse, push and « are both ‘‘snoc’’, include? ‘‘∋’’, map`. Functions `first` and `last` take an optional numeric argument n to obtain the *first* n or the *last* n elements of a list.

Methods yield new arrays; updates are performed by methods ending in “!”.

```
x = [1, 2, 3]  # A new array
x.reverse      # A new array; x is unchanged
x.reverse!     # x has changed!

# Traverse an array using “each” and “each_with_index”.
x.each do |e| puts e.to_s end
```

Catenation `+`, union `|`, difference `-`, intersection `&`.
Here is a cheatsheet of array operations in Ruby.

What Haskell calls `foldl`, Ruby calls `inject`;
e.g., `xs.inject(0) do |sofar, x| sofar + x end` yields the sum of `xs`.

Symbols

Symbols are immutable constants which act as *first-class variables.*

- ◇ Symbols evaluate to themselves, like literals `12` and `"this"`.

```
:hello.class # ⇒ Symbol
# :nice = 2 # ⇒ ERROR!
```

```
# Conversion from strings
"nice".to_sym == :nice  # ⇒ true
```

Strings occupy different locations in memory even though they are observationally indistinguishable. In contrast, all occurrences of a symbol refer to the same memory location.

```
:nice.object_id == :nice.object_id   # ⇒ true
"this".object_id == "this".object_id # ⇒ false
```

Control Flow

We may omit `then` by using `;` or a newline, and may contract `else if` into `elsif`.

```
# Let 𝒞 ∈ {if, unless}
𝒞 :test_1 then :this else :that end
this 𝒞 test  ≈  𝒞 test then this else nil end
```

```
  (1..5).each do |e| puts e.to_s end
≈ 1 .upto 5 do |e| puts e end
≈ 5 .downto 1 do |e| puts 6 - e end
≈ for e in 1..5 do puts e.to_s end
≈ e = 1; while e <= 5 do puts e.to_s; e += 1 end
≈ e = 1; begin puts e.to_s; e += 1 end until e > 5
≈ e = 1; loop do puts e.to_s; e += 1; break if e > 5 end
```

Just as `break` exits a loop, `next` continues to the next iteration, and `redo` restarts at the beginning of an iteration.

There’s also times for repeating a block a number of times, and step for traversing over every n-th element of a collection.

```
n.times   S ≈ (1..n).each S
c.step(n) S ≈ c.each_with_index {|val, indx| S.call(val) if indx % n == 0}
```

See here for a host of loop examples.

Hashes

Also known as finite functions, or ‘dictionaries’ of key-value pairs —a dictionary matches words with their definitions.

Collections are buckets for objects; hashes are labelled buckets: The label is the key and the value is the object. Thus, hashes are like objects of classes, where the keys are slots that are tied to values.

```
hash = { "jasim" => :farm, :qasim => "hockey", 12 => true}

hash.keys     # ⇒ ["jasim", :qasim, 12]
hash["jasim"] # ⇒ :farm
hash[12]      # ⇒ true
hash[:nope]   # ⇒ nil
```

Simpler syntax when all keys are symbols.

```
oh = {this: 12, that: "nope", and: :yup}
oh.keys  #⇒  [:this, :that, :and]
oh[:and] #  ⇒  :yup

# Traverse an array using “each” and “each_with_index”.
oh.each do |k, v| puts k.to_s end
```

As always, learn more with `Hash.methods` to get `keys, values, key?, value?, each, map, count, ...` and even the “safe navigation operator” `dig`: `h.dig(:x, :y, :z)` ≈ `h[:x] && h[:x][:y] && h[:x][:y][:z]`.

We may pass in any number of keyword arguments using `**`.

```
def woah (**z) z[:name] end

woah name: "Jasim" , work: "Farm" #⇒  Jasim
```

Hashes can be used to model (rose) trees:

```
family = {grandpa: {dad:   {child1: nil, child2: nil},
                    uncle: {child3: nil, child4: nil},
                    scar:  nil}}

# Depths of deepest node.
def height t
    if not t
    then 0
    else t.map{|k, v| height v}.map{|e| e + 1}.max
    end end

height family #  ⇒  3
```

Classes

Classes are labelled product types: They denote values of tuples with named components. Classes are to objects as cookie cutters (templates) are to cookies.

Modifiers: `public, private, protected`

- Everything is public by default.
- One a modifier is declared, by itself on its own line, it remains in effect until another modifier is declared.
- Public ⇒ Inherited by children and can be used without any constraints.
- Protected ⇒ Inherited by children, and may be occur freely *anywhere* in the class definition; such as being called on other instances of the same class.
- Private ⇒ Can only occur stand-alone in the class definition.

These are forms of advice.

`Class` is also an object in Ruby.

```
  class C ⟨⟨contents⟩⟩ end
≈
  C = Class.new do ⟨⟨contents⟩⟩ end
```

Instance attributes are variables such that each object has a different copy; their names must start with `@` —“at” for “at”tribute.

Class attributes are variables that are mutually shared by all objects; their names must start with `@@` —“at all” ≈ attribute for all.

`self` refers to the entity being defined as a whole; `name` refers to the entities string name.

```
class Person

  @@world = 0 # How many persons are there?
  # Instance values: These give us a reader “x.field” to see a field
  # and a writer “x.field = ...” to assign to it.
  attr_accessor :name
  attr_accessor :work

  # Optional; Constructor method via the special “initialize” method
  def initialize (name, work) @name = name; @work = work; @@world += 1 end

  # See the static value, world
  def world
    @@world
  end
  # Class methods use “self”;
  # they can only be called by the class, not by instances.
  def self.flood
    puts "A great flood has killed all of humanity"; @@world = 0 end

end

jasim = Person.new("Qasim", "Farmer")
qasim = Person.new("", "")
jasim.name = "Jasim"

puts "#{jasim.name} is a #{jasim.work}"
puts "There are #{qasim.world} people here!"
Person.flood
puts "There are #{qasim.world} people here!"
```

- See here to learn more about the new *method*.

Using `define_method` along with `instance_variable_set("@#namehere", value)` and `instance_variable_get("@#namehere")`, we can elegantly form a number of related methods from a list of names; e.g., recall `attr_accessor`. Whence **design patterns become library methods!**

In Ruby, just as methods can be overriden and advised, classes are open: They can be extended anytime. This is akin to C# extension methods or Haskell's typeclasses.

```
# Open up existing class and add a method.
class Fixnum
  def my_times; self.downto 1 do yield end end
end

3.my_times do puts "neato" end # ⇒ Prints “neato” thrice
```

- We can freely add and alter class continents long after a class is defined.
- We may even alter core classes.
- Useful to extend classes with new functionality.

Modules & Mixins

Single parent inheritance: `class Child < Parent ··· end`, for propagating behaviour to similar objects.

A `module` is a collection of functions and constants, whose contents may become part of any class. Implicitly, the module will depend on a number of class methods —c.f., Java interfaces— which are used to implement the module's contents. This way, we can *mix in* additional capabilities into objects regardless of similarity.

Modules:

- Inclusion binds module contents to the class instances.
- Extension binds module contents to the class itself.

```
# Implicitly depends on a function “did”
module M; def go; "I #{did}!" end end

# Each class here defines a method “did”; Action makes it static.
# Both include the module; the first dynamically, the second statically.
class Verb; include M;  def did; "jumped" end end
class Action; extend M; def self.did; "sat"  end end

puts "#{Verb.new.go} versus #{Action.go}"
# ⇒ I jumped! versus I sat!
```

For example, a class wanting to be an `Enumerable` must implement `each` and a class wanting to be `Comparable` must implement the 'spaceship' operator `<=>`. In turn, we may then use `sort, any?, max, member?, ...`; run `Enumerable.instance_methods` to list many useful methods.

Modules are also values and can be defined anywhere:

```
mymod = Module.new do def talk; "Hi" end end
```

Todo COMMENT more

```
# Print with p, puts, print
p 1 , 2, :three # ⇒ 1, 2, :three
```

```
# Like PHP and Perl, “heredoc” quotes long strings: Left quote is <<XYZ and right
# quote is XYZ, where XYZ is any sequence of characters. Use <<'XYZ' for the text
# to not be interpolated.
b = <<XXX
hello
XXX
c = <<'abc'
ddd hello
abc
```

In pure OO, such as Smalltalk, conditional constructs are not part of the language but are instead merely defined behaviour for the Boolean class. That is, one sends a message to a Boolean object on how to proceed, rather than taking action by inspecting it.

- ◇ Smalltalk syntax: b ifTrue: [· · ·] .
- ◇ Ruby syntax: if b then · · · end

Looking at, for example, true.methods does not seem that a conditional operation is defined for the Boolean class.

Likewise for loops, and chains of and's and or's.

How can control flow be construed as message passing in Ruby? —Without adding it ‘after the fact’ with something like this:

```
class TrueClass
  def ifTrue; yield; end
end

class FalseClass
  def ifTrue; self; end
end

puts ((1 == 0 + 1).ifTrue do "hi" end) # ⇒  "hi". Need parens for some reason; why!?
puts  (1 == 0).ifTrue do "hi" end      # ⇒  "false"
```

```
hi
false
false
```

(Incidentally, how do I get at the Boolean expressions themselves? How can I obtain, say 1 = 0, in the second example above and for example print it as ‘‘1 = 0” rather than “false”. Moreover, “1 = 0’’ is really ‘‘1.send(:, 0)”, so how can I get access to the object 1, the method ==, and the other argument, 0?)

I'm nearly a week new into Ruby; thanks for being patient.

https://www.reddit.com/r/ruby/comments/cz2v9t/ruby_is_pure_oo_99_true/

In mathematics, the “η-rule”, a form of extensionality, says

$$(\lambda x \to f x) \approx f$$

In almost this form, this rule holds for Haskell and Lisp. **How does this rule hold for Ruby?**

With the help of some code, see below, I have arrived at this rule:

`h do |e| g e end ≈ h(&method(:g))`

Provided `h` takes an explicit block and `g` is defined using `def`. If `g` is defined using `lambda`, then rhs is `h(&g)`.

Below is some code to back this up. However, **can this rule be stated more tersely?** My observation does not account for the case `h` takes an implicit block. How is that treated?

```
def go; "Hello" end
go' = lambda { "Hola" }
def go¹ (x) "Hey, #{x}" end
go'¹ = lambda {|x| "Yo, #{x}"}

def explicit (&blk) blk.call end
explicit do go end     # ⇒ Hello
explicit(&method(:go)) # ⇒ Hello
explicit(&go')         # ⇒ Hola'
#
# “&:” not uniform?
explicit(&:go) rescue "crashes"
explicit(&:go') rescue "crashes"
[1,2,3].map(&:to_s)    # ⇒ ["1", "2", "3"]

def explicit¹ (&blk) blk.call(1) end
explicit¹ do |x| x.to_s end # ⇒  "1"
explicit¹(&:to_s)           # ⇒  "1"
explicit¹(&method(:go¹))    # ⇒  "Hey, 1"
```

```
explicit¹(&:go¹) rescue "crashes"
explicit¹(&:go’¹) rescue "crashes"
explicit¹(&go’¹) # ⇒ "Yo, 1"

def implicit; yield end
implicit do go end # ⇒ Hello
# implicit (&method(:go)) ⇒ Syntax error!

def implicit¹; yield(1) end
implicit¹ do |x| go¹ x end   # ⇒ "Hey, 1"
# implicit¹ (&method(:go¹))  # ⇒ Syntax error
```

I’m nearly a week new into Ruby; thanks for being patient.

https://www.reddit.com/r/ruby/comments/cz2msl/etaconversion_in_ruby_what_do/

Reads

- ◇ Ruby Monk — Interactive, in browser, tutorials
- ◇ Ruby Meta-tutorial — ruby-lang.org
- ◇ The Odin Project
- ◇ Learn Ruby in ~30 minutes — https://learnxinyminutes.com/
- ◇ contracts.ruby — Making assertions about your code
- ◇ Algebraic Data Types for Ruby
- ◇ Community-driven Ruby Coding Style Guide
- ◇ Programming Ruby: The Pragmatic Programmer’s Guide
- ◇ Learn Ruby in One Video – Derek Banas’ Languages Series
- ◇ Learn Ruby Using Zen Koans
- ◇ Metaprogramming in Ruby —also some useful snippets
- ◇ Seven Languages in Seven Weeks