Ruby Reference Sheet

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.

```
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
```

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.

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.

Convention: Predicate names end in a `?`; destructive function names end in `!`.

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
```

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]
```

```ruby
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.

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

"Singleton methods": You can attach methods to existing names whenever you like.

```ruby
x = "ni"
def x.upcase; "The knights who say #{self}" end
x.upcase # ⇒ The knights who say ni

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

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

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

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

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

Forming aliases:

```ruby
alias summing sum''
summing 1, 2, 3  # ⇒ 6
```

Variables & Assignment

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

```ruby
# 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]"!

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

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

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

# Variable scope is determined by name decoration.
# Constants are names that begin with a captial letter.
$a = 2; @a = 3; @aa = 4; A = 5
[defined? a, defined? $a, defined? @a, defined? @@a, defined? A]
# ⇒ [local-variable , global-variable , instance-variable , hline, constant]
```

Strings

Single quotes are for string literals, whereas double quotes are for string evaluation, 'interpolation'.

```ruby
you = 12          # ⇒ 12
"Me and #{you}"   # ⇒ Me and 12
'Me and #{you}'   # ⇒ Me and #{you}

# String catenation
"This " + "That"
"This " << "That"
```

```ruby
# "to string" function
"hello " + 23.to_s  # ⇒ hello 23

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

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

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.
- `and, or` are the usual logical operators but with lower precedence.
- They're used for control flow; e.g., $\texttt{s}_0$ `and` $\texttt{s}_1$ `and` $\cdots$ `and` $\texttt{s}_n$ does each of the $\texttt{s}_i$ until one of them is false.

Arrays

Arrays are heterogeneous and 0-indexed.

```ruby
array = [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.

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

Inclusive Subsegment using `..`, excluding upper index using `,`.

`x[0..2]` ≈ `x[0, 3]` ≈ $[\texttt{x}_0, \texttt{x}_1, \texttt{x}_2]$

Syntactic sugar: `x[i]` ≈ `x.[] i`

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 "!".

```ruby
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
```

Symbols

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

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

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

```ruby
# 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.

```ruby
: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`.

```ruby
if :test1 then :this else if :test2 then :that end  end
```

```ruby
  (1..5).each do |e| puts e.to_s 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
```

Hashes

Finite functions, or 'dictionaries' of key-value pairs.

```
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

# As always, learn more with
# hash.methods   ⇒  keys, values, key?, value?, each, map, count

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

Classes

Instance fields are any `@` prefixed variables.

- ◇ Class fields, which are shared by all instances, are any `@@` prefixed variables.

```
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!"
```

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.

Classes are open!

- ◇ 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.

`Class` is also an object in Ruby.

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

```
C = Class.new do attr_accessor :hi end

c = C.new
c.hi = 12
puts "#{c.hi} is neato"
```

Modules & Mixins

Inheritance: `class Child < Parent ··· end`.

Modules:

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

```
module M; def go; "I did it!" end end

class Verb; include M end
class Action; extend M end

puts "#{Verb.new.go} versus #{Action.go}"
```

```
I did it! versus I did it!
```

Reads

- Ruby Monk — Interactive, in browser, tutorials
- Ruby Meta-tutorial — ruby-lang.org
- 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