Ruby Basic

1. Paradigm: Procedural, Functional, OO
2. Convention
   1. shebang: #!/usr/bin/env ruby
   2. obj1.object\_id == obj2.object\_id 🡪 same obj
3. Array
   1. arr[start, length].

arr1 = 1, 2, 3, 4 arr[start, length]

arr2 = Array.new(3, "e") #=> ["e", "e", "e"]

arr3 = Array.new(arr2) # **same e, diff arr**

arr3[2] = 1; arr3 #=> [nil, nil, 1], created **hole**

%w{str} # str 🡪 **raw** string, split by space

%w{a b c}.combination(2).to\_a #3 comb. [[ ], [ ], [ ]]

%w{a b c}.permutation(2).to\_a #6 permutations

* 1. Logic

arr1 & arr2, arr1 | arr2, arr1 | arr2 - arr1 & arr2 # XOR

[1, 2] \* 2 #=> [1, 2, 1, 2] [1, 2] – [2, 3] # [1]

* 1. Methods: change arr directly

arr << e # append .pop(n)# n>1🡪Array, 1🡪element

.push(2, 3) .insert(index, val)

.shift(n) # left pop .unshift(n) # left push

* 1. Methods: not change arr

.last(n), .first(n), .length, .size,.empty?, .equal?(arr2)

[1, 2, 3].join(str) # "123", ⬄ arr \* str, default ""

str.split(str) #=> array, can be RegEx

* 1. Methods: only change arr in ! version

.sort # **ascend** .shuffle # random

.reverse .uniq # unique elements list

.rotate(n) # rotate to make [n] first, default 1

.slice(start, n) # w/o n🡪 arr[n] .slice(range)

.flatten(n) # flat n dim from outer, flat all dim w/o n

* 1. Methods: iteration

arr.each {|e|} arr.each\_index{|index|}

arr.each\_with\_index{|e, index|}

arr.map/collect{} # map=>arr, arr.select{} # filter🡪arr

arr.inject(init, :operator) # accumulate

arr.inject {|sum, x| sum … x} # block return as accu

1. Hash

h = {'key' => 'val'} || {:key => 'val'} || {key: 'val'}

h = Hash.new(1) # default val when key not=exist

a = Array.new(2, Hash.new)

# a[0].object\_id == a[1].object\_id

* 1. Methods: read only

.keys #=> list of keys .key(val)

.has\_key?(k) | h.include? | h.key? | h.member?

.values # val list .values\_at(k, l) # val | default list

.value?(k), .has\_value?(k) # bool

.default, .default = 2, h["notkey"] # default

h.size || h.length, h.empty?

h.to\_a # [[k, v], [k, v]], array

h.invert #=> {val=>key} pairs, de-dup

* 1. Methods: only change in ! version

h[:key].upcase!

.merge(h2) # keep h2 val

.merge(h2){|k v1 v2| v1} # keep h val

* 1. Methods: write

.clear # remove all .delete(key)#=> h[key]

.delete(k) {|k| p k} # pass key if not found, return val

.delete\_if {|k, val|} h.keep\_if {|k, val|}

.replace(h2) # h == h2 .shift # pop first pair

* 1. Methods: iteration

h.each || h.each\_pair {|key, val|}

h.each\_key {|key|} h.each\_value {|val|}

Ruby Method

1. Convention
   1. Name: can\_end\_with\_!?=.If start with Capital, must use () to call, Capital()
   2. () is optional: def m p1, p2; return ""; end, NOTHING before m if calling without ()
   3. def m(a, x:"1", y:"2"); p x, y; end, Keyword params must be at last
2. Nested
   1. Cannot contain class or module def
   2. Sub methods exist after sup-mthds called
3. Passed by:
   1. Val: immediate value: nil, true, false, Fixnums, Symbols, and some Floats.
   2. Ref: other objects.
4. puts = print + \n, puts a, b = puts a + puts b

def m(a, b=a\*a); puts "#{a}, #{b}"; end

> m(3) #=> 3, 9 > m(3, 4) #=> 3, 4

1. Splat: rest \*args

def m(a, \*b, c); puts "#{a}, #{b}, #{c}"; end

m(1, 2, 3, 4, 5) #=> 1, [2, 3, 4], 5

m("I", "do", f:"g", h:"i") #=> I, ["do"], {:f=>"g", :h=>"i"}

# hash must be at last

def m2(a, \*, c); puts "#{a}, #{c}"; end

m2(1, 2, 3, 4, 5) #=> 1, 5

1. Get hash:\*\*

def m(p, \*\*rest); p "#{p}, #{rest}"; end

def search(field, genre:nil, duration:120, \*rest)

p [field, genre, duration, rest]; end

1. Alias new old: preserve original method.
2. obj.send(:m | "m", var): invoke dynamically

Ruby Expression Assignment

1. Parallel

a = (b = 2\*3) + 4 # a = 10, b = 6

a = 1,2,3 # [1,2,3] a, = 1, 2, 3 # a=1

a, b = 1, 2, 3 # a =1, b =2

* 1. Splat on right

a, b, c = (1..3) # a=1, b=c=nil

a, b, c, d = \*(1..2), \*(3..4) # a=1, b=2, c=3, d=4

a, b = [1, 2], [3, 4] # a=[1,2], b=[3,4]

a, b = \*[1, 2], [3, 4] # a=1, b=2

a, b = \*"abc" # a=["abc"], b=nil

a, b = {1=>2, 3=>4} # a={1=>2, 3=>4}, b=nil

a, b = \*{1=>2, 3=>4} # a=[1, 2], b=[3, 4], hash🡪array

* 1. Splat on left (**only 1 \* allowed**)

a, \*b = 1, 2, 3 # a=1, b=[2, 3]

\*a, b = 1, 2, 3 # a=[1, 2], b=3

a, \*b, c = 1, 2, 3, 4 # a=1, b=[2, 3], c=4

a, \*b, c = 1, 2 # a=1, b=[ ], c=2

\*, last = 1, 2, 3 # last=3

* 1. Unit: treated as 1 var

a, (b, c) = 1, 2, 3 # a=1, b=2, c=nil

a, (b, c) = 1, [2, 3], 4 # a=1, b=2, c=3

# decompose first: ( ) ⬄ var

Ruby Control

1. Condition (case, if returns last exec exp)

n = if … # use 'then' for branch at same line

# puts "word" if age > 10

# unless: if not, puts "young" unless age > 60

n = case var; when …; …; else; …; end

# "===" used for "when" (Range:between, Obj: ==, **RegEx**: match, Module: is an instance of it)

1. Loop:
   1. break: : stop, next: skip

while; …, end | until # only built-in loop primitives

# **no i++, i-- in Ruby**

Loop do; …, end # block

for # ".each" method

10.times {}, 1.upto(10), "word".each\_char {}

1. Block
   1. Params in yield are passed into block.
   2. yield( ) returns block val
   3. Scope: var def inside not avail outside, var def outside **AVAIL** inside, |; y| to block.

# call block and pass vals

class Hash; def each; len, x = self.length, 0

while x < len

yield(self.keys[x], self.values[x]) # pass |k, v|

x += 1; end; end; end

# block return value through yield

class Array; def find; self.each do |val|

puts val if **yield(val, 'hello')** # get block return val

end; end; end

[1,2,3].find{|x| x>1} #=> 2, 3

# pass block

def merge!(h, &block)

self.merge(h, &block) .each {|key, val| self[key] = val}

end; end

* 1. if block\_given? block provided/not
  2. block vs. do; end: higher precedence

**upto 10** do |x|; end # "upto" binded with "10" first

upto 10 {do |x|} # syntax err, 10 binded with {}

# should be upto(10) {do |x|}

Ruby Class

1. Convention
   1. MultiWorkdClassName
   2. Class definition **automatically** executed.
   3. self is current scope.
   4. Inst var in diff scopes are DIFFERENT

class C

puts "A def" # class def, auto exec

attr\_accessor: var # obj inst accessor

@var = 10 # class inst var, **diff scope**

class << self

attr\_accessor: var; end # class inst accessor

def initialize(v) # initializer

@var = v # obj inst var, **won't change 10**

self.class.m; end # call class mtd from inst mtd

(def C.set | self.set # class (inst) mtd, class var setter

@ var = 15; end # **change 10 to 15, not change v)**

(def get # obj (inst) mtd, obj var getter

@var; end # **get v** instead of 10/15)

class D < class C; def new; …; end; end # inheritance

class D = Class.new(C) do; def new; …; end; end

E = Struct.new(:a, :b) # class only have attr

|  |  |  |
| --- | --- | --- |
|  | Obj instance | Class instance |
| Var | In obj inst | In class inst |
| Method | In obj class | In singleton class |

1. Singleton mtd vs singleton var (duck typing)
   1. Obj: in singleton class (anonymous)
   2. Class: all class mtd are singleton

obj1 = MyClass.new

def obj1.mtd; …; end # mtd for "obj1" only

# or class << obj1; def mtd; ..;end; end

obj1.instance\_variable\_set(:var, "1") # add inst var

1. Open Class: reopen same name Class.

Ruby Module

1. Same as Class, but no .new() and super, and can’t be a super class

> Module.ancestors

=> [Module, Object, Kernel, BasicObject]

> MyModule = Module.new

> MyModule.new || > MyModule.superclass => **error**

1. Usage: organize CONSTANTS (class, module, method)
2. Constant: anything start w/ capital letter
   1. Val can be changed (w/ a warning)
3. Scope:
   1. Enter new scope: Module, Class, Method
   2. List
      1. Var: puts local\_variables
      2. CONST: puts MyModule.constants
   3. Variable: invisible to **other(👆/👇)** scope.
   4. Access constant (CONST, Class, Method)
      1. From outside: M::Cls::C2
      2. From inside: ::C1 (get top C1 in Cls.run)

puts M::C1 #=> 1

puts M::Cls::C2 #=> 2

# class method

puts M::Cls.run #=> [1, 0]

# or M::Cls::run

M::C0 = 1

puts M::C0 #=> 1

C1 = 0

module M

C1 = 1

class Cls

C2 = 2

def Cls.run

return C1, ::C1

end

end

end

1. Module method => Class method
2. Mix-in
   1. include 🡪 mix-in instance methods to upper level, can modify inst var.
   2. prepend: include to lower level, won’t be overwritten if same method exists.
   3. extend🡪 add **inst** as **class methods**.

module MyModule; def my; p "#{self}"; end; end

class MyClass; **include** MyModule; end

obj = MyClass.new; obj.my #=> "#<MyClass:…>"

class NewClass; **extend** MyModule; end

NewClass.my #=> "NewClass"

* 1. include in main:🡪 instance methods available at top level.

module Math; def my; p "#{self}"; end; end

include Math

my #=> "main"

* 1. Inst method 🡪 module method: module\_function(:symbol)

module Math; def my; p "#{self}"; end

module\_function(:my); end

Math.my #=> "Math"

1. Load modules
   1. System module: require “module”
   2. Own file: require\_relative “folder/module”
2. Subclassing vs mix-in
   1. Sub: special case of parent🡪 car < vehicle
   2. Mix-in: mix other cases in  
      🡪 def Human; include Disease; end

RegEx

1. str.scan(RegEx) => list of match
2. escape: \

r = /re/ ⬄ Regexp.new(re) ⬄ %r{re} # last not escape!

# anchor 🡪 immediate char/group

/abc/i # case /(C|D|)og/ # ⬄ /[cd]og/

+ ⬄{1, } \* ⬄ {0, } ? ⬄ {0, 1} # {min, max}

. ⬄ any 1 char

\d: digit ⬄ ^\D, \w:[z-aA-Z\_0-9] ⬄^\W

\s: space, \t, \n ⬄ ^\S

\A:^, \z:$

\b, ^\B 🡪 boundary (not), one side

str.scan(re) # matched list

re =~ str # position | nil re !~ str # not match

s.sub(re, t) # replace first s.gsub(re, t) # replace all

.sub! | .gsub! returns modified or **nil**

# // method

.source: no / /, .inspect: /.source/, .to\_s: (?-mix:.source)

m = /e/.match("new")

m.pre\_match-m[0]-m.post\_match # n-e-w

# goruping

m[n] 🡪 n-th matched GROUP(, m[0] is matching str

# $1…$n is the same

/(w+)\1/ # \1🡪same as group 1

/(**?<nm>**\w+)**\k<nm>**/: same, nm also be local var

str = "it is a good class" # max/min match

/\s.\*\s/ 🡪 greedy, " is a good ", /\s.\*?\s/🡪lazy, " is "

File IO

1. Methods

putc: char

gets # readline: screen or (file in ARGV)

readlines # ["each", "line"]

# File

file = File.open("f.txt", "r") do |f| # r/r+/w/w+/a/a+

while line = f.gets; puts "#{line}" # reading

# or f.each\_line {|line| puts "#{line}"}

f.puts "…" # writing

# file.close auto called if use block

Puts File.read("file")

File.rename(s, s1), .size

#IO .foreach('file') {|line| …}

str = IO.read # into str arr = IO.readlines # into array

STDOUT << str << …

ARGV: args array # ARGV.each {|arg|}

ARGF.lineno, ARGF.filename, ARGF.file.lineno, line

Dir: .pwd, .chdir(str), .entries(path) # all file

Dir.foreach(path), Dir.home(usr)

CSV.foreach(filename){|row|…}

1. Socket

require 'socket'

sock\_serv = Socket.new(:INET, STREAM || DGRAM)

serv\_addr =Socket.pack\_sockaddr\_in((1025..48999), ip)

sock\_serv.bind(addr)

sock\_serv.listen(n) # default 5

loop {conn = sock\_serv.accept; conn.close}

#S: sock\_serv = TCPServer.new(ip, port)

Socket.accept\_loop(sock\_serv) {|conn| conn.close}

sock\_clnt = Socket.new(:INET, STREAM || DGRAM)

sock\_clnt.connect(serv\_addr)

# read socket

Socket.tcp\_server\_sockets(host=nil, port) {|conn|

puts conn.gets; conn.close}

# write socket

sock\_clnt = TCPSocket.new(ip, port)

sock\_clnt.write(str)

1. STD Lib

require 'net/http'

site - %{domainName.com}; path = "/"

response = Net::HTTP.get\_response(site, path)

puts "Code = #{response.code}",

puts "Message = #{response.message}"

response.each {|key, value|

printf "%-15s = %-100s\n", key, value}

p response.body[0, 500]

// require 'open-uri'

CGI/Template Concept



Str = <<END <html> END

print "Content-Type: text/html;charset=UTF-8\n"

print "Content-Length: #{str.size}\n"

print "Connection: close\n\n"; puts

print str

1. CGI#params collect request as hash

url/script?a="1"&a="2"&b=3

require 'cgi'

cgi = CGI.new; cgi.user\_agent

cgi["a"] # ["1", "2"], cgi.keys # ["a", "b"]

cgi.params # {"a"=>["1", "2"], "b"=>"3"}

------------------

p cgi.params["a"] if cgi.params["a"] != ""

cgi.out do; cgi.html do

cgi.head {cgi.title {"this is a cgi program"}} +

cgi.body do

cgi.h1 {"your submit from the form are:"} +

cgi.p {p1} + cgi.p {p2}; end; end; end

1. Cookie: CGI::Cookie, save in browser

require 'cgi'; cgi = CGI.new("html5")

c\_name, c\_val = "visit, cig.cookie[c\_name] # access

mycookie = CGI::Cookie.new(c\_name, (c\_val.to\_i+1).to\_s)

mycookie.expires = Time.now + 30\*24)3600

cgi.out("cookie" => mycookie) {msg}

1. Session:

require 'cgi'; require 'cgi/session'

cgi = CGI.new("html5")

session = CGI::Session.new(cgi,

"session\_key" => "mysession", "session\_expires"=> …)

session['visit'] = (session['visit']||0).to\_i + 1

session.close # store on server

1. Template
   1. ERB: filename.ouput\_type.erb

<% code %>, % whole line code,

<%= expression %>, <% #comment %>

require 'erb'

class List; attr\_accessor :items

def initialize(items); items = items; end

def bind; binding(); end; end # filter can access inst v

list = List.new(["egg", "milk"])

render = ERB.new(template)

puts output = render.result(list.bind)

$> erb template.html.erb > output.html

* 1. HAML (div can be ommited)

%:tag, =: replaced by val of ruby expression

-: execute without val replacement

<strong class="code" id="message">Hello!</strong>

%strong{:class => "code", :id => "message"} Hello!

%strong .code#message Hello!

<strong><%= item.title %></strong>

%strong= item.title

<div class='item' id='item<%= item.id %>'><%= item.

require 'haml'

engine = Haml::Engine.new("%p Haml code!")

# "<p>Haml code!</p>\n"



