Ruby

Basic: Syntax, Comments, Variables, Functions

Collections: Arrays, Lists, Hashes

Object Oriented: Classes, Inheritance, Modules, Mixins

Unit Testing: Rspec

Ruby: Dynamic, OO (everything is object), Elegant, expressive & Declarative (Terse but extremely readable)

eg. 3.times { puts "Hello World" }

- 2 space indentation for each nested level is encouraged
- # for comments "pound"
- Everything is evaluated
- puts = put string
- p - ✓ eg. p "Got it" ⇒ Got it.
- Naming Conventions:
  - Variables: Lowercase seperated by underscore eg. snake_case
  - Constant: ALL_CAPS
  - Classes (& Modules): CamelCase
- Semicolons (;) not needed; or to cram several statements (discouraged)
- IRB - Interactive Ruby (type 'irb' in command line)
  - Console-based interactive Ruby interpreter
    - REPL (Read Evaluate Print Loop)
  - Comes with Ruby installation, let you experiment quickly

- Flow of Control
  - if/elsif/else ; case ; until/unless?; while/for
  - true ? false?
  - ===?

- if/elsif/else/unless
  - No () or {}
  - Use `end` to close flow control

eg.
```
a=5 # declare a var
if a ==3
  puts "a is 3"
elsif a ==5
  puts "a is 5"
else
  puts "a is not 3 or 5"
end
```
⇒ a is 5

eg.
```
a=5
unless a ==6
  puts "a is not 6"
end
```
⇒ a is not 6

- while, until

eg.
```
a=10
while a>9
  puts a
  a-=1 # a=a-1
end
```
⇒ 10

eg
```
a=9
until a>=10
  puts a
  a+=1
end
```
⇒ 9

- Flow of Control - Modifier Form : same line

eg.
```
a=5
b=0
puts "One liner" if a==5 and b==0
```
⇒ One liner

eg.
```
times_2=2
times_2 *2 while times_2<100
puts times_2
```
⇒ 128 # 64*2

- True/False
  - false & nil objects are false, Everything else is true!
    (false, nil) (0, "", "nil", "false")
    ↳ Warning.

- Triple Equal (===)
  - use (==) most of the time
  - A special kind of equal
    - eg /sera/ === "coursera" ⇒ true
      "coursera" === "coursera" ⇒ " (=== delegates to ==)
      Integer === 21 ⇒ true
      ↳ class ↳ value

- Case Expressions
  - 2 types:
    - 1) Similar to 'if' statements
      2) Specify a target next to `case` & each `when` clause is compared to target.
  - === → case equality operator
  - No fall-through logic (the only case that actually matched get executed)

eg.
```
age = 21
case
  when age >= 21
    puts "abc"
  when 1 == 0
    puts "def"
  else
    puts "default"
end
```
⇒ abc

↳ the case can be sth to do with age variables

eg.
```
name = 'Fisher'
case name
  when /fish/i then puts "abc"
  when 'Smith' then puts "def"
end
```
⇒ abc

↳ have actual target next to `case`, implicitly compare itself after when

- For loop
  - Hardly used in ruby
  - each/times are preferred

eg. for i in 0..2 ← range data type
  puts i
end
⟹ 0
1
2

Summary:
- list of options for flow of control; modifier form; Always true except nil & false

L3) Functions / Methods
- Def, How do you call them?, what & how do they return?, Default args
- How to make methods more expressive
- What is "splat"

Functions & Methods
- Technically, function: defined outside of a class
  method: defined inside a class

In Ruby, every function/method has at least 1 class it belongs to.
- Not always written inside a class

∴ Every function is really a method in Ruby

Methods
- () are optional both when defining & calling a method. It's used for clarity

eg.
```
def simple
  puts "abc"
end

def simple1()
  puts "def"
end
```

simple
simple() ⟹ abc

simple1
simple1() ⟹ def

- Return
  - Noneed to declare type of param; return whatever you want; `return` is optional (last executed line)

eg
```
def add(a,b)
  a+b
end

def divide(a,b)
  return "ZeroDivisionError" if b==0
  a/b
end
```

puts add(2,2) ⟹ 4
puts divide(2,0) ⟹ ZeroDivisionError
puts divide(12,4) ⟹ 3

Expressive Method Names

- Method names can end with:
  - '?' - Predicate methods
  - '!' - Dangerous side-effects.

eg.
```
def can_divide_by(number)
  return false if number.zero?
  true
end
puts can_divide_by? 3   ⇒ true
puts can_divide_by? 0   ⇒ false
```

Default Arguments

- If a value is passed in - use that value
- Otherwise - use the default value provided.

eg.
```
def factorial(n)
  n == 0 ? 1 : n * factorial(n-1)
end

puts factorial 5  ⇒ 120
```

```
def factorial_with_default(n=5)
  n = 0 ? 1 : n * factorial_with_default(n-1)
end
puts factorial_with_default  ⇒ 120
puts factorial_with_default(3) ⇒ 6
```

* Ternary operator:
  condition ? true : false

Splat (*)

- * prefixes parameter inside method definition
  - Can even apply to middle parameter, not just the last.

eg.
```
def max(one_param, *numbers, another)
  numbers.max  # return max & for unlimited param & become an array
end          # value of numbers array
puts max("sth", 7, 32, -4, "more")
```

⇒ 32 # 32 is the maximum value among [7, 32, -4]

Summary:

Dynamic (↑ param type passed in or return); `return` is optional;

- You can construct methods with variable number of arguments or default arguments.
  (*)

- 4) Blocks
  - Blocks, How they are used, How to incorporate them into your own methods

- Blocks
  - Chunks of code, enclosed by either curly braces {} or do & end
  - Passed to methods as last "parameter"
  - Convention:
    - Use {} for single line block
    - Use `do` & `end` for multiple lines block
  - Often used as iterators
    - Can accept arguments

```
1.times { puts "Hello World!" }  => Hello World!
2.times do |index| <- often accept params between ||
  if index > 0
    puts index                   => 1 # iteration are 0 & 1, only 1 is printed
  end
end

2.times {|index| puts index if index > 0 } => 1 # same
```

  - Coding with blocks
    - 2 ways to configure a block in your own method
      - Implicit
        - Use block_given? to see if block was passed in
        - Use yield to "call" the block
      - Explicit "ampersand"
        - Use `&` in front of the last param
        - Use call method to call the block.

- Implicit
  - Need to check "block_given?", otherwise, an exception is thrown

```
def two_times_implicit
  return "No block" unless block_given?
  yield
  yield
end
puts two_times_implicit { print "Hello" } ⇒ Hello
                                            Hello
puts two_times_implicit ⇒ No block
```

- Explicit
  - Should check if the block is nil?

```
def two_times_explicit (&i_am_a_block)
  return "No block" if i_am_a_block.nil?
  i_am_a_block call
  i_am_a_block.call
end

puts two_times_explicit ⇒ No block
two_times_explicit { puts "Hello" } ⇒ Hello
                                      Hello
```

  * Explicit is more direct

Summary

- Blocks are just code to be passed into methods. When incorporating them into your own methods, either use blocks implicitly, or call them explicitly

5) Files

Reading & writing to files, Exceptions, Reading values from environment variables.

Reading from File

test.txt

```
a b c
d e f
g
```

```
File.foreach('test.txt') do |line|
  puts line
  p line
  p line.chomp # chops off newline char (\n)
  p line.split # array of words in line
end
```

```
⇒ a b c                  # puts line
"a b c\n"                # p line
"a b c"                  # p line.chomp
["a", "b", "c"]          # p line.split
d e f
"d e f\n"
"d e f"
["d", "e", "f"]
g
"g\n"
"g"
["g"]
```

Program ⇒ Abruptly ends

Reading from Non-Existing File ⇒ Exception Error (Errno::ENOENT)

Handling Exceptions

```
begin
  File.foreach('no_file.txt') do |line|
    puts line.chomp
  end
rescue Exception => e
  puts e.message
  puts "Let's pretend this didn't happen..."
end
```

⇒ No such file ... Let's pretend this didn't happen...

- Alternative to Exceptions
  - `if File.exist? 'w.txt'  # check if file exists`
    `File.foreach('test.txt') do |line|`
    `puts line.chomp`
    `end`
    `end`
  - ↵
  - ↳ it won't help you where a real exception happens, like sth wrong with ur network..
  - ↳ use it if it's simple case of file not being found.
- Writing to File
  - Automatically closed after the block executes
  - `File.open("test1.txt","w") do |file|`
    `file.puts "abc"`
    `file.puts "def"`
    `end`

    ⇒

    test1.txt
    abc
    def
  - ↵
- Environment Variables
  - `puts ENV["EDITOR"]` ⇒ subl # for sublime editor.

Summary
- Files auto closed at the end of block
- Either use exception handling or check for existence of file before accessing