Ruby Monk. (10/2/2020)

1.1 String Basics.

- String Interpolation "w # {var}"
  - * single quote(') doesn't support interpolation or escape character)
- .length ; .include?('w') ; .start_with?('w') ; .end_with?('w'), .index('w') ; .downcase ; .upcase ; .swapcase

1.2 Advanced String Operations

- splitting string : .split(' ')
- Concatenating Strings : a+b , .concat('w') ; a<<b ← append
  - created a 3rd string, beware of this for large scale string manipulations.
- Replacing a substring : .sub('a','b') ; .gsub('a','b')
  - sub → first occurrences
  - gsub → all
  - + RegEx : .gsub(/[aeiou]/,'1') ; gsub(/[A-Z]/,'0')
- Find a substring using RegEx : .match(/ ./)
  - find the characters from a string which are next to a whitespace. (first match)
  - for further match : .match(/ ./,9)

2.0 Boolean Expressions in Ruby

- ==, >=, <=, &&, ||,
- negating expression (!)

2.1 if..else construct
- if, elsif, else
- unless → unless x == if !x
- ternary operator
  - ? :
- Truthiness of objects in Ruby
  - 'false' & 'nil' equate to false; every other object like say 1, 0, "", are all evaluated to be 'true'

2.2 Loops in Ruby
- Loop are programming constructs that help you repeat an action an arbitrary number of times.
- Methods .each ; .select are most frequently used loops since the primary use of loops is to iterate over or transform a collection.
- 2 basic looping constructs :
  - ```
    loop do
      monk.meditate
      break if monk.nirvana?
    end.
    ```
  - ```
    def ring(bell, n)
      n.times do
        bell.ring
      end
    end
    ```

3.0 Intro to Arrays.
- Empty Arrays : [] ; Array.new
- Building Arrays : [1,2,3] ..
- Looking up data in Array :
  - [a, b, c, d, e]
    0 1 2 3 4
    -5 -4 -3 -2 -1
- Growing Arrays : In Ruby, array size is not fixed.
  - [1,2,3,4,5] << "woot"
  - [1, 2, 3, 4, 5].push("woot")

3.1 Basic Array Operations

- Transforming arrays: .map
  - eg. [1,2,3,4,5].map{|i| i*3}
    - ⇒ [3,6,9,12,15]
  - Ruby aliases the method .map to .collect, they can be used interchangeably.
- Filtering elements of an Array
  - .select
    - eg. [1,2,3,4,5,6].select {|number| number%2 == 0} ⇒ [2,4,6]
    - eg. .select {|i| i.length > 5}
- Deleting elements
  - [1,2,3,5,6,7].delete(5) ⇒ [1,2,3,6,7]
  - .delete_if
    - [1,2,3,4,5,6,7].delete_if {|i| i < 4} ⇒ [4,5,6,7]
    - [1,2,3,4,5,6,7,8,9].delete_if {|i| i%2 == 0} ⇒ [1,3,5,7,9]

* Doing these in C or Java would need a lot of boiler plate code. The beauty of Ruby is in its concise but readable code.

3.2 Iteration.

- Got 'for' loops? Yup, but interestingly, nobody uses it much.

eg.

```
array = [1,2,3,4,5]
for i in array
  puts i
end
```

```
def array_copy(source)
  destination = []
  for i in source
    destination.push(i) if i < 4
  end
  return destination
end
```

- looping with 'each'
  - 'each' method accepts a block to which element of the array is passed. You'll find that for loops are hardly used in Ruby. .each & its siblings are the de-facto standard.

eg.

```
array = [1,2,3,4,5]
array.each do |i|
  puts i
end
```

```
1
2
3
4
5
```

```
def array_copy(source)
  destination = []
  source.each do |i|
    destination.push(i) if i<4
  end
  return destination
end
```

4.0 Intro to Ruby Hashes

- Creating a Hash
  - Hash: a collection of key-value pairs; retrieve/create a new entry by referring to its key. Also called 'associative arrays', 'dictionary', 'HashMap' in other languages.
    - Declared using 2 curly braces {}
    - eg.

```
restaurant_menu = {
  "Ramen" => 3,
  "Dal Makhani" => 4,
  "Tea" => 2
}
```

- Fetch values from a Hash
  - You can retrieve values from a Hash object using [] operator.
    - eg. `restaurant_menu['Ramen']`
- Modifying a Hash
  - eg. `restaurant_menu['Ramen'] = 11.21`

4.1 Hashes, in and out

Iterating over a Hash

- ~~When you~~ .each, passes 2 values to the block: key & value
- eg.

```
restaurant_menu = {"Ramen" => 3, "Dal Makhani" => 4, "Coffee" => 2}
restaurant_menu.each do | item, price |
  restaurant_menu[item] = price * 1.1
end
```

- Ideally, any transformation of a collection should produce a new collection with the original unchanged making the code easier to understand & manage.
- However, speed & memory considerations often (& usually wrongly) trump maintainability & so the approach above is used quite frequently.

Extracting the keys & values from a Hash.

- 2 methods → .keys & .values, which returns an array of all keys / values

eg. `restaurant_menu.keys` → `["Ramen", "Dal Makhani", "Coffee"]`

Newer, faster

- Some little-known shortcuts, which provides a slight convenience

eg.

```
normal = Hash.new
was_not_there = normal[:zig]      } set default value
puts "Wasn't there:"
p was_not_there                   } generate a hash directly from pre-existing key-value pairs.
usually_brown = Hash.new("brown")
pretending_to_be_there = usually_brown[:zig]   } set default value
puts "Pretending to be there:"
p pretending_to_be_there
```

```
Wasn't there:
nil
Pretending to be there:
"brown"
```

→ a "normal" hash always return `nil` by default, specifying a default in the Hash constructor will always ~~return~~

* A "normal" hash always return `nil` by default, specifying a default in Hash constructor will always return your custom default for any failed lookups on that hash instance.

The other 2 shortcuts actually use the `Hash` class's convenience method: `Hash::[]`. They're fairly straight-forward. the first takes a flat list of parameters, arranged in pairs The second takes just one parameter: an array containing arrays which are themselves key-value pairs

eg.
```
chuck_norris = Hash[:punch, 99, :kick, 98, :stop_bullets_with_hands, true]
p chuck_norris
```
=>
```
{:punch=>99, :kick=>98, :stop_bullets_with_hands=>true}
```

eg. Build Hash out of array:

```
def artax
  a = [:punch, 0]
  b = [:kick, 72]
  c = [:stops_bullets_with_hands, false]
  key_value_pairs = [a, b, c]
  Hash[key_value_pairs]
end
p artax
```
=> `{:punch=>0, :kick=>72, :stops_bullets_with_hands=>false}`

5.0 Classes

- Ruby allows one to define groups of objects, or, to use object oriented jargon, classes of objects.
- One may look up the class of any object by simply calling the `class` method on it.

eg.
```
puts 1.class        → Fixnum (special case of integer)
puts "".class       → String
puts [].class       → Array
```

- Turn this interrogation around by using .is_a (<class>)

eg.
```
puts 1.is_a? (Integer)   → true
puts 1.is_a? (String)    → false
```

- Classes are people too
  - One important feature of Ruby → "everything is an object", unlike other languages like C++ where classes are special constructs that cannot be interacted with like normal objects.
  - In Ruby, classes themselves are simply objects that belongs to the class `Class`
  - eg. `1.class.class` → Class

- What do classes do?
  - Classes act as the factories that build objects. An object built by a certain class is called 'an instance of that class'. Typically, calling `new` method on a class results in an instance being created.
    - eg. `Object.new` → An instance of Object is created.

& This object, or instance - can't do very much. However, instances of more powerful classes like Array & Strings can do lot more. You'll learn to create your own classes which you can then use to build bigger & more powerful object.

5.1 Building your own classes.

- A class needs a job or a role that its instances fulfill.
- All classes shall have names that begin with a capital letter.
- eg.
```
class Rectangle
end
```
  - Incomplete For a class to justify its existence < State : defines attributes of its instances
    Behavior : methods for interaction with its state.

- State & Behavior.
  - def to create methods.
    - eg.
```
class Rectangle
  def perimeter
    2*(@length + @breath)
  end
end
```
      - @ : a convention - to design them as being a part of the state of the class, or, they are the "instance variables of the class"
        - every instance of Rectangle class will have its own unique copies of these variables.
  - @length & @breath don't have value yet.
    -
```
class Rectangle
  def initialize(length, breath)
    @length = length
    @breath = breath
  end
  def perimeter
    2*(@length + @breath)
  end
end
```
      → + def area
        @length * @breadth
        end

6.1 Being Methodical

What are methods, really?

- Objects collaborate through methods. Method, is simply programming jargon for sth one object can do for another.
- eg. `puts 1.next` ⏎ ⇝ 2
- In brief, the data an object contains is what it is & its methods are what it can do. The abilities of an object are limited to the methods it exposes.

Objectifying methods

- Methods aren't exempt from Ruby's "everything is an object" rule. The methods exposed by any object are themselves objects.
- All objects in Ruby expose the eponymous `method` method that can be used to get hold of any of its methods as an object.
- eg. `puts 1.method("next")` ⏎ ⇝ `#<Method: Fixnum(Integer)#next>`
  - ↳ here, we ask the object (integer 1) to give us the instance of the method `next`
- The method object still maintains a relationship with the object to which it belongs, so you can still call it using the eponymous `call` method & it responds like a normal invocation of that method.

eg.
```
next_method_object = 1.method("next")
puts next_method_object.call
```
⏎ ⇝ 2

↳ In normal course ~~of this it's~~, don't need do that IT.

Make it so

- write a method called `reverse_sign` that takes one object - an `Integer` - & changes a positive value to a negative one

```
def reverse_sign(an_integer)
  return 0 - an_integer
end

puts reverse_sign(100)   → -100
puts reverse_sign(-5)    →  5
```

⇝ * method names should be lower case.
* A method must always return exactly one object.

eg. ⌜def do_nothing
end
puts do_nothing.class ⌟ ⇝ printing a 'nil' returns an empty string.

└ Be cautious when using return - calling return also exits the method at that pt. No code in the method after the 'return' statement is executed..

└ Calling 'return' without specifying an object to return results in a 'nil', which is returned by default.

~~└ An excellent~~

└ Beat me up, Scotty

⌜def add_two(n)
n+2
end

6.2 Calling a method

└ Cooperative objects (> 1 parameter)

eg. 2 param:

⌜def add(a, b)
a+b
end ⇝ 3
puts add(1,2) ⌟

eg. params with default value

⌜def add(a, b, c=0)
a+b+c
end
puts add(1,2) ⌟ ⇝ 3

Arranging your arguments

- The list of parameters passed to an object is available as a list by using splat operator → asterisk (*)
  - used to handle methods which have a variable parameter list
- we use 'inject' method to iterate over arguments
- eg.

```
def add(*numbers)
  numbers.inject(0) { |sum, number| sum+number }
end

puts add(1)          → 1
puts add(1,2)        → 3
puts add(1,2,3)      → 6
puts add(1,2,3,4)    → 10
```

- The splat operator works both ways ← convert arrays to parameter lists / convert para list to an array
- eg. how to splat array into a para list

```
def add(a, b, c)
  a+b+c
end
numbers_to_add = [1,2,3]
puts add(*numbers_to_add)   → 6
```

= ☆ → splatting an array into items.

- eg: To allow for the sum to be printed as a part of a message. We know the msg but don't know how many numbers to be added.

```
def add(*n)
  n.inject(0) { |sum, number| sum + number }
end
def add_with_msg(msg, *n)
  "#{msg}: #{add(*n)}"
end
puts add_with_msg("The Sum is", 1,2,3)   → 6
```

ex: a method called 'introduction' that accepts a person's age, gender & any number of names, returns a 'string'

```
def introduction(age, gender, *names)
  "Meet #{names.join(' ')}, who's #{age} and #{gender}"
end
```

↳ concatenate names using '.join' method

Naming parameters

eg.

```
def add(a, b, options = {})
  sum = a+b
  sum = sum.abs if options[:absolute]
  sum = sum.round(options[:precision]) if options[:round]
  sum
end
puts add(1.0134, -5.568)
puts add(1.0134, -5.568, absolute: true)
puts add(1.0134, -5.568, absolute: true, round: true, precision: 2)
```

2 param - 2 numbers

3 param - 2 numbers & a hash

Hash

↳
```
-4.5546
4.5546
4.55
```

↳ Ruby makes this possible by allowing the last parameter in the parameter list to skip using curly braces if it's a hash, making for a much prettier invocation. That's why we default the 'option' to {}, because if it isn't passed, it should be an empty Hash.

A not-so-gentle workout

Q: Write 3 methods - `calculate`, `add` & `subtract`

Hint: Write `add` & `subtract` first, then have `calculate` call one or the other depending on the options passed

* The inject based approach for addition will require some modification for `subtraction`.
* For `calculate`, you can't use both splatted arguments & last-param-is-a-hash at the same time through Ruby, so you'll have to work on the arguments inside of calculate.
* There's no neat way to do this - you have to check if the last argument to calculate is a Hash, then remove it from the list before calling `add` or `subtract`

```
def add(*numbers)
  numbers.inject(0) { |sum, number| sum + number }
end

def subtract(*numbers)
  current_result = numbers.shift
  numbers.inject(current_result) { |current_result, number| current_result
  - number }
end

def calculate(*arguments)
  # if the last argument is a Hash, extract it. Otherwise create an empty Hash.
  options = arguments[-1].is_a?(Hash) ? arguments.pop : {}
  options[:add] = true if options.empty?
  return add(*arguments) if options[:add]
  return subtract(*arguments) if options[:subtract]
end
```

7.1 Lambdas in Ruby

- Lambdas : a function ... peculiarly ... without a name ; anonymous, little functional spies sneaking into the rest of your code ; objects ; last expression of lambda is its return value ≈ regular functions ; gives us a lot of power.
- As objects, lambda have methods & can be assigned to variables.
  - eg.
```
l = lambda { "Do or do not" }
puts l.call
```
    → Do or do not
    - Notice that our anticipatorily apologetic string is the return value of the lambda which we see by printing it using 'puts'.
- Returns sth more interesting ... Lambdas take parameters by surrounding them with pipes → | |

eg.
```
l = lambda do |string|
  if string == "try"
    return "There's no such thing"
  else
    return "Do or do not"
  end
end
puts l.call("try")
```
→ Lambdas do find with 'do...end' or '{}', but the convention followed in Ruby is to use '{}' for single line lambdas & 'do...end' for lambdas that are longer than a single line.

- There's no such thing

Ex : Add a lambda which increments any number passed to it by 1.
```
Increment = lambda { |n| n+1 }
```
Increment.call(1) → 2
Increment.call(-1) → 0
Increment.call(1000) → 1001.

7.2 Blocks in Ruby

Lambdas vs. Blocks

- Lambdas: A piece of code that you can store in a variable, and is an object
- Blocks: A piece of code that can't be stored in a variable & isn't an object, but faster than lambda, not as versatile & also one of the rare instances where Ruby's "everything is an object" rule is broken.
- eg. A block that increments any number passed to it by 1.

```
def demonstrate_block(number)
  yield(number)
end
puts demonstrate_block(1) { |number| number + 1 }
```

→ 2

  - * There's no 'lambda'
  - * sth new called 'yield'
  - * There's a method that has the body of lambda immediately after the parameter list, which is a little weird

Skipping the details

- One of the most common uses of lambda: passing exactly 1 block to a method.
- Ruby optimizes for this use case by offering the 'yield' keyword that can call a single lambda that has been implicitly passed to a method without using the parameter list.
  - * Similar pattern: 1 method, 1 block passed to it outside of the param list, and with the block called using 'yield'
- ex:

```
def calculate(a,b)
  yield(a,b)
end
```

```
calculate(2,3) { |a,b| a+b }   → 5
c      "       { |a,b| a-b }   → -1
calculate(2,3) { |a,b| a*b }   → 6
```

8.1 Getting Modular

- Mixing it up
  - Ruby modules allow you to create groups of methods that you can then include/mix into any number of classes.
  - Modules can only hold behavior, unlike classes, which hold both behavior and state.
  - Module can be instantiated, its method can't be called directly. Instead, it should be included in another class, which makes its methods available for use in instances of that class.
  - In order to include a module into a class, we use method `include` which takes one parameter - the name of a `Module`
  - eg.

```
module WarmUp
  def push_ups
    "Phew, I need a break!"
  end
end
class Gym
  include WarmUp
  def preacher_curls
    "I'm building my biceps."
  end
end
class Dojo
  include WarmUp
  def tai_kyo_kyu
    "Look at my stance!"
  end
end
puts Gym.new.push_ups
puts Dojo.new.push_ups
```

    - "Phew, I need a break!" x 2
    - 'Gym' & 'Dojo' each have their own, distinct behavior, but both require 'push_ups', so this behavior has been separated into a module, which is then included into both classes.

Some hierarchy & a little exercise ..

- Just like all classes are instances of Ruby's 'Class', all modules in Ruby are instances of 'Module'
- Interestingly, 'Module' is the superclass of 'Class' → all classes are also modules. *(Inheritence)

- eg.
```
module WarmUp
end
puts WarmUp.class              → Module
puts Class.superclass          → Module
puts Module.superclass         → Object
```

Ex.
```
module Perimeter
  def perimeter
    sides.inspect(0){ |sum, side| sum + side }
  end
end
class Rectangle
  include Perimeter
  def initialize(length, breadth)
    @length = length
    @breadth = breadth
  end
  def sides
    [@length, @breadth, @length, @breadth]
  end
end
class Square
  include Perimeter
  def initialize(side)
    @side = side
  end
  def sides
    [@side, @side, @side, @side]
  end
end
```

eg. Rectangle.new(2,3).perimeter → 10
Square.new(5).perimeter → 20

8.2 Modules as Namespaces

- Collision course
  - Namespacing is a way of bundling logically related objects together & Modules serve as a convenient tool for this.
  - This allows classes or modules with conflicting names to co-exist while avoiding collisions.
  - Think of this as storing different files with the same names under separate directories in your filesystem.
- Modules can also hold classes.
  - eg.

```
module Perimeter
  class Array
    def initialize
      @size = 400
    end
  end
end

our_array = Perimeter::Array.new
ruby_array = Array.new
p our_array.class
p ruby_array.class
```

→ define 'Array' class under 'Perimeter' module

→ `::` : a constant lookup operator that looks up the 'Array' constant only in the 'Perimeter' module.

→ Perimeter::Array
Array

- What happens when we don't namespace our 'Array' class?

```
class Array
  def initialize
    @size = 400
  end
end

our_array = Array.new
p our_array.class
```

→ Array

* because Ruby has open classes, doing this simply extends the 'Array' class globally throughout the program, which is dangerous.

- The previous examples are a bit contrieved for the sake of simplicity; the real problem that namespacing solves is when you're loading libraries. If your program bundles libraries written by different authors, it is often the case that there might be classes or modules defined by the same time.

eg
```
#class Push
#  def up
#    40
#  end
# end
  require "gym" # up returns 40
  gym_push = Push.new
  p gym_push.up
# class Push
# def up
#   30
# end
# end
require "dojo" # up returns 30
dojo_push = Push.new
p dojo_push.up
```

$ 30
  40

'dojo' library is loaded before 'gym', it has overriden gym's class def. of 'Push' & therefore creates an instance of 'Push' defined in 'dojo'

⇝ The way to solve this problem is to wrap these classes in appropriate namespaces using modules.

```
require "gym"
require "dojo"
dojo_push = Dojo::Push.new
p dojo_push.up          ⇝ 30
gym_push = Gym::Push.new
p gym_push.up           ⇝ 40
```

* When you're creating libraries with Ruby, it is a good practice to namespace your code under the name of your library or project.

- Constant lookup (::)
  - To scope our class to the module
  - You can scope any constant using this operator & not just classes

```
module Dojo
  A = 4
  module Kata
    B = 8
    module Roulette
      class ScopeIn
        def push
          15
        end
      end
    end
  end
end
A = 16
B = 23
C = 42
puts "A - #{A}"
puts "Dojo::A - #{Dojo::A}"
puts
puts "B - #{B}"
puts "Dojo::Kata::B - #{Dojo::Kata::B}"
puts
puts "C - #{C}"
puts "Dojo::Kata::Roulette::ScopeIn.new.push - #{Dojo::Kata::Roulette::ScopeIn.new.push}"
```

- * Constant 'A' is scoped within 'Dojo' and accessing it via :: works as expected.
- * Same for constant 'B' which is nested further inside 'Kata'
- * Class 'ScopeIn' is nested even deeper inside 'Roulette' which has a method returning 15.

↓

- * 2 pts:
  - We can nest constant lookups as deep as we want.
  - We aren't restricted to just classes / modules

Output:

```
A - 16
Dojo::A - 4
B - 23
Dojo::Kata::B - 8
C - 42
Dojo::Kata::Roulette::ScopeIn.new.push - 15
```

Q: Given a library 'RubyMonk', containing a module 'Parser', which defines a class 'CodeParser'. Write another class 'TextParser' in the same namespace that parses a string and returns an array of capitalized alphabets.

Ex:
```
module RubyMonk
  module Parser
    class TextParser
      def self.parse(str)
        str.upcase.split("")
      end
    end
  end
end
```

↳ If you prepend a constant with :: without a parent, the scoping happens on the topmost level.

Q: In this Ex, change 'push' to return '10' as per 'A = 10' in the topmost level, outside the 'Kata' module

```
module Kata
  A = 5
  module Dojo
    B = 9
    A = 7
    class ScopeIn
      def push
        ::A
      end
    end
  end
end

A = 10 *
```

Kata::Dojo::ScopeIn.new.push ⇒ 10

Using modules & namespacing is the standard way of organizing libraries with Ruby. It's a good practice to keep this in mind while writing one

9.1 Streams

I/O streams & the IO class

- An input/output stream is a sequence of data bytes that are accessed sequentially or randomly;
- I/O streams are used to work with almost everything about your computer that you can touch, see, or hear:
  - printing text to the screen; receiving key-press input from the keyboard; playing sound through speakers; sending & receiving data over a network; reading & writing files stored on disk.
    - considered "side-effects" in Computer Science. The touch/see/hear metric doesn't seem to work for network traffic & disk activity but side-effects are not necessarily obvious.
  - In I/O, sth in the world has physically changed even if you can't see it.
- Comparatively, "pure" code is code without side-effects: code which simply performs calculations, but a "pure" program isn't very useful if it can't even print its results to the screen. This is where I/O streams come in. Ruby `IO` class allows you to initialize these streams.

eg.
```
# open the file "new-fd" and create a file descriptor:
fd = IO.sysopen("new-fd", "w")
# create a new I/O stream using the file descriptor for "new-id":
p IO.new(fd)
```

- `fd`, the first argument to IO.new, is a file descriptor. This is a `fixnum` value we assign to an `IO` object.
- We're using a combination of the `sysopen` method with `IO.new` but we can also create IO objects using the `BasicSocket` and `File` classes that are subclasses of `IO`.
- The notion of creating a "file descriptor" is inherited from UNIX, where everything is a file.

- There are bunch of I/O stream that Ruby initializes when the interpreter gets loaded.

eg
```
io_streams = Array.new
ObjectSpace.each_object(IO) {|x| io_streams << x}

p io_streams
```

- Standard Output, Input, and Error.
  - Ruby defines constant `STDOUT`, `STDIN` and `STDERR` that are IO objects pointing to your program's input, output and error streams that you can use through your terminal, without opening any new files.

eg.
```
p STDOUT.class          => IO
p STDOUT.fileno         => 1
p STDIN.class           => IO
p STDIN.fileno          => 0
p STDERR.class          => IO
p STDERR.fileno         => 2
```

  - Whenever you call `puts`, the output is sent to the `IO` object that `STDOUT` points to. It is the same for `gets`, where the input is captured by the `IO` object for `STDIN` and the `warn` method which directs to `STDERR`

- There is more to this. The `Kernel` module provides us with global variables `$stdout`, `$stdin`, and `$stderr` as well, which point to the same `IO` objects that the constant `STDOUT`, `STDIN` and `STDERR` point to. We can see this by checking their `object_id`.

eg.
```
p $stdin.object_id          20152620
p STDIN.object_id           20152600

puts                        ⇒
p $stdout.object_id         20152560
p STDOUT.object_id          20152580
puts
p $stderr.object_id         20152500
p STDERR.object_id          20152500
```

↳ The 'object_ids' are consistent between the global variables & constraints.

↳ Whenever you call 'puts', you're actually calling 'Kernel.puts' (methods in 'Kernel' are accessible everywhere in Ruby), which in turn calls '$stdout.puts'.

↳ So why all the indirection? The purpose of these global variables is temporary redirection: you can assign these global variables to another 'IO' object and pick up an IO stream other than the one that it is linked to by default.

↳ We can use the 'StringIO' class to easily fake out the 'IO' objects. Try to capture STDERR so that calls to 'warn' are redirected to our custom 'StringIO' object.

Ex.
```
capture = StringIO.new
$stderr = capture
```

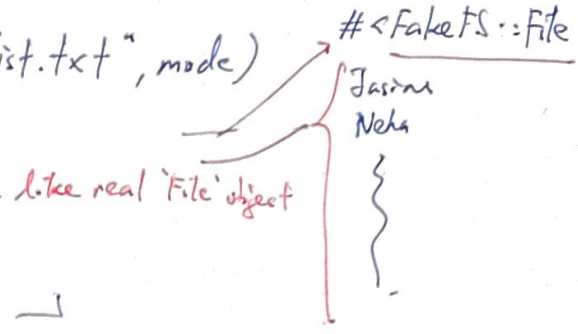
9.2 Using the `File` Class

Opening & Closing

- Where we used `IO.sysopen` and `IO.new` to create a new `IO` object in the last lesson, we'll use the `File` class here.

(Note that `file.inspect` will return a `FakeFS::File` -- this isn't a real `File` object because otherwise your "friend-list.txt" would conflict with other rubymonk users "friends-list.txt") ~~Don't worry~~

eg.
```
mode = "r+"
file = File.open("Friend-list.txt", mode)
puts file.inspect
puts file.read
file.close
```



- `mode` is a string that specifies the way you would like your file to be opened.
- `'r+'` : which opens the file in read-write mode, starting from the beginning.
- `'w'` : write-only mode, truncating the existing file.
- `File.open` takes an optional block which will auto-close the file you opened & once you are done with it.

eg.
```
what_am_i = File.open("clean-slate.txt", "w") do |file|
  file.puts "Call me Ishmael."
end
p what_am_i
File.open("clean-slate.txt", "r") {|file| puts file.read}
```

```
=> nil
Call me Ishmael
```

Reading & Writing

- some methods to read from an I/O screen. In below examples, I/o stream is a file; files behave just like any other I/o stream.
  - File.read method accepts 2 optional arguments:
    - 'length': the number of bytes upto which the stream will be read
    - 'buffer': to provide a 'String' buffer which will be filled with the file data, sometimes useful for performance when iterating over a file, as it re-uses an already initialized string

eg.
```
file = File.open("master", "r+")
p file.read
file.rewind  # if commented
buffer = ""
p file.read(23, buffer)
p buffer

file.close
```

→ "Master loves you as he loves Jacob"
nil
""

→ "Master loves you as he loves Jacob"
"Master loves you as "

- Why using 'File.rewind' before 2nd 'file.read'?
  Because Ruby's internal handling of files. When reading from a 'File' object, Ruby keeps track of your position. In doing so, you can read a file one line (or page / arbitrary chunk) at a time without recalculating where you left off after the last read.
- File.seek solidifies this idea even further. You can "seek" to a particular byte in the file to tell Ruby where you want to start reading from. If you want a particular set of bytes from the file, you can then pass the 'length' parameter to 'File.read' to select a number of bytes from your new starting pt.

eg.
```
p File.read("monk")
File.open("monk") do |f|
  f.seek(20, IO::SEEK_SET)
  p f.read(10)
end
```

→ "Master loves you as he loves Jacob"
"he loves J"

- `readlines` returns an array of all the lines of the opened IO stream. You can optionally limit the number of lines and/or insert a custom separator between each of these lines.

eg.
```
lines = File.readlines("monk")
p lines
p lines[0]
```
↳ ["Master loves", "you as he", "loves Jacob."]
"Master loves"

- To write to an I/O stream, we can use `IO.write` (or `File.write`) and pass in a string. It returns the number of bytes that were written. Try calling the method that writes "Bar" to a file named `disguise`.

Ex:
```
File.open("disguise", "w") do |f|
  f.write "Bar"
end
```

done Ruby Beginner.

Problems:

1) Calculator:
```
class Calculator
  def add(a, b)
    a+b
  end
  def subtract(a,b)
    a-b
  end
end
```

2) Find length of strings in an array
```
def length_finder(input_array)
  input_array.map { |element| element.length }
end
```
array
eg ['Kyle', 'is', 'handsome']
↳ [4, 2, 8]

3) Find the frequency of a string in a sentence
```
def find_frequency(sentence, word)
  sentence.downcase.split.count(word.downcase)
end
```
eg. [1,2,1,1,1].count(1)
↳ 4
↳ turn sentence into array

4) Sort the words in a given sentence

eg. "~~Sorts~~ Given a input "Sort words in a sentence" → "a in Sort words sentence"

```
def sort_string(string)
  string.split.sort{|x,y| x.length <=> y.length}.join(' ')
end
```

split(' '): turn string into an array of words.

<=> : combined comparison operator (-1 when a<b, 0 when a=b, 1 when a>b); a <=> b → -1

turn array into a string

5) Select random elements from an array

- Input: array, number of items
- Output: an array of randomly selected items.

```
def random_select(array, n)
  result = []
  n.times do
    result << array[rand(array.length)]
  end
  result
end
```

loops. ← n.times do

eg. rand(5) randomly return 0-4; takes a parameter & randomly returns a number that lesser than the param.

<< : append

[ ] : indexing

6) Hiring Programmers - Boolean Expressions in Ruby

```
is_an_experienced_ruby_programmer =
  (candidate.languages_worked_with.include? 'Ruby') &&
  (candidate.years_of_experience >= 2 || candidate.github_points>=500)
  &&
  !(candidate.age < 15 || candidate.applied_recently?)
```

7) Palindromes · read the same backward

```
def palindrome?(sentence)
  downcase_stripped_sentence = sentence.downcase.gsub(" ", "")
  downcase_stripped_sentence == downcase_stripped_sentence.reverse
end
```

downcase & remove all spaces between words.

8) Compute sum of cubes for given range

```
def sum_of_cubes(a,b)
  sum=0
  for i in a..b
    sum += i**3
  end
  sum
end
```

```
def sum_of_cubes(a,b)
  (a..b).inject(0){|sum,x| sum += (x*x*x)}
end
```

9) Find non-duplicate values in an Array

eg. [1,2,2,3,3,4,5] → [1,4,5]

```
def non_duplicated_values(values)
  values.find_all{|x| values.count(x) == 1}
end
```

10) Check if all elements in an array are Fixnum

```
def array_of_fixnum?(array)
  array.all?{|x| x is_a? Fixnum}
end
```

11) Kaprekar's Number

eg. 9 ($9^2=81$, $8+1=9$); 297 ($297^2=88209$, $88+209=297$)

for a Kaprekar number k with n-digits, if you square it & add the right n digits to the left n or n-1 digits, the resultant sum is k.

⇓

```
def kaprekar?(k)
    no_of_digits = k.to_s.size
    square = (k**2).to_s

    second_half = square[-no_of_digits..-1]
    first_half = square.size_even? ? square[0..no_of_digits-1] :
                 square[0..no_of_digits-2]

    k == first_half.to_i + second_half.to_i
end
```

4 A Kaprekar number k with n-digits, if you square it & add the right n digits to the left n or n-1 digits, the resultant sum is k.
- eg. 9 ($9^2=81$, $8+1=9$); 297 ($297^2=88209$, $88+209=297$)

(2) Enough Contrast?
- For 2 colors in RGB:
  (R1, G1, B1) & (R2, G2, B2)

Brightness index is:
(299*R1 + 587*G1 + 114*B1) / 1000

Brightness difference is:
Absolute diff. in brightness indices

Hue difference is
|R1-R2| + |G1-G2| + |B1-B2|

Q: If Brightness difference > 125 & Hue diff. > 500, then the colors have sufficient contrast.

```
class Color
  attr_reader :r, :g, :b
  def initialize(r,g,b)
    @r = r
    @g = g
    @b = b
  end
  def brightness_index
    (r*299 + g*587 + b*114)/1000
  end
  def brightness_difference(another_color)
    (brightness_index - another_color.brightness_index).abs
  end
  def hue_difference(another_color)
    (r-another_color.r).abs +
    (g-another_color.g).abs +
    (b-another_color.b).abs
  end
  def enough_contrast?(another_color)
    brightness_difference(another_color) > 125 && hue_difference(another_color) > 500
  end
end
```

B) Time to run code.

- Given some code in the form of lambdas, measure & return the time taken to execute that code (may use Time.now to get the current time)

```
def exec_time(proc)
  begin_time = Time.now
  proc.call
  Time.now - begin_time
end
```

14) Number shuffle

└ Given a 3 to 4 digit number with distinct digits, return a sorted array of all the unique numbers that can be formed with those digits.

```
def number_shuffle(number)
  no_of_combinations = number.to_s.size == 3 ? 6 : 24
  digits = number.to_s.split(//)
  combinations = []
  combinations << digits.shuffle.join.to_i while
    combinations.uniq.size != no_of_combinations
    combinations.uniq.sort
end
```

15) Orders and costs

└ A restaurant has incoming orders for which you need to compute the costs based on the menu.

```
class Restaurant
  def initialize(menu)
    @menu = menu
  end
  def cost(*orders)
    orders.inject(0) do |total_cost, order|
      total_cost + order.keys.inject(0) {|cost, key| cost + @menu[key] * order[key]}
    end
  end
end
```

16) Your Sum - Given a custom class MyArray, write a 'sum' method that takes a block of parameter

```
class MyArray
  attr_reader :array
  def initialize(array)
    @array = array
  end

  def sum(initial_value = 0)
    return array.inject(:+) + initial_value unless block_given?
    sum = initial_value
    array.each {|n| sum += yield(n)}
    sum
  end
end
```