CMSC330 NOTES

I. Intro

- Semantics: the meaning of sentences/languages
- Syntax: the structures of the language
- **Why so many languages?**

=> There are many programming languages because different languages have been designed to solve specific problems and cater to various requirements

II. Ruby

1. Comments: Use #

2. Print: Use puts

```
Example:    puts "hello world!"     #output: hello world!
            puts "abc" "ABC"        #output: abcABC
            puts 3 + 4              #output: 7
            puts 3 + "abc"          #output: TypeError
            puts "abc"*3            #output: abcabcabc
            puts 3*"abc"            #output: TypeError
```

3. Typing
   - Type Checking: The process of determining a variable's type
     + Dynamic typing: Type checking is performed at runtime
     + Static typing: Type checking is performed at compile time
   - Explicit/Implicit Typing:
     + Manifest (explicit): explicitly telling the compiler the type of new variables
       => Types are associated with variables
     + Latent (implicit): not needing to give a type to a variable
       => Types are associated with values

   >>> Ruby uses **dynamic** and **latent** typing

4. "Primitive" Data Types

1. Integer

- Arithmetic Operations: +, -, *, /, % (modulus), ** (exponentiation)
- Convert to other data types:

+ Float:	to_f	3.to_f #3.0
+ String:	to_s	3.to_s #"3"
+ Binary String:	to_s(2)	3.to_s(2) #"11"

- Bitwise Operations: AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>)
- Hexadecimal and binary representations: 0x and 0b
- Notes: 1_000_000 is also Integer

2. Float (Similar to Integer)

Notes:
2. and .0 are not valid for floats
2.0/2 = 1.0 where 2/2 = 1
Instead of doing Math.sqrt(3), we can do 3 ** 0.5

3. String

- Create strings: Use either single quotes or double quotes
- Concatenation and repetition: str + str and str * int (int * str doesn't work)
- String indexing and slicing: str = "Hello world"

```
+ Access individual character:      str[4]            # "o"
                                    str[20]           # nil
+ Extract substrings:               str[6..]          # "world"
                                    str[0,5]          # "Hello"
                                    str[3..7]         # "lo wo"
```

- Substitution: str = "Hello world"

```
                                    str["Hello"]      # "Hello"
                                    str["hello"]      # nil
                                    str["world"] = "" # str = "Hello "
```

- Escaping characters: quotes(\") and newline(\n) and others
- String methods: chomp, chomp!, **length or size**, reverse, upcase, downcase, capitalize, **include?, index, sub, sub!, gsub, gsub!**, start_with?, end_with?, sprintf, split, strip,...
- Interpolation: using #{expression} "I'm #{2023-2001} years old"
  Note: Strings created by single quotes doesn't allow interpolation
- Convert to numbers: "101".to_i=>101 "101".to_i(2)=>5
- Compare strings: == (https://medium.com/@khalidh64/difference-between-eql-equal-in-ruby-2ffa7f073532)

4. Symbol

- Creation: Using a colon followed by the identifier, such as :hello
- Immutable: Their value cannot be changed
- Unique: Two symbols with the same name refer to the same object

5. Array

- Creation: arr = [] or arr = [1,2,3] or arr = Array.new or arr = Array.new(10,1)
- Indexing: using arr[index], starting at 0
- Slicing: Subarrays can be extracted (similar to String)
- Iteration: By using for loops or using each (https://mixandgo.com/learn/ruby/each)

```
for i in arr          | for i in 0..arr.length-1     | arr.each{|x|
    puts i            |    puts arr[i]               |     puts x
end                   | end                          | }
```

- Adding and Deleting methods: arr = [1,2,3,4] /Examples below are separately/
    - **push(element1, element2,...):** add elements to the end of an array
      ```
      arr.push(5,6)          # => arr = [1,2,3,4,5,6]
      ```
    - **pop** or **pop(n):** remove the (n) last element of an array and return it
      ```
      arr.pop                # => 4
      arr.pop(2)             # => [3,4]
      ```
    - **unshift(element1, element2,...):** add elements to the beginning of an array
      ```
      arr.unshift(0,1,2)     # => arr = [0,1,2,1,2,3,4]
      ```
    - **shift** or **shift**(n): remove the (n) first element of an array and return it
      ```
      arr.shift              # => 1      (arr = [2,3,4])
      arr.shift(3)           # => [1,2,3](arr = [4])
      ```
    - **delete(value, n):** remove n elements from an array based on its value
      ```
      arr.delete(3)          # => 3      (arr = [1,2,4])
      arr.delete(5)          # => nil    (arr = [1,2,3,4])
      ```
    - **delete_at(index):** remove an element from an array based on its index
      ```
      arr.delete_at(1)       # => 2      (arr = [1,3,4])
      ```
    - **detele_if{|x| condition}:** remove elements when condition is true
      ```
      arr.delete_if{|x| x < 3} # => arr = [3,4]
      ```
- Dynamic Sizing: arr = []
  ```
  arr[5] = 5 # => arr = [nil,nil,nil,nil,nil,5]
  ```
- Array Operations: A = [1,2,3,4,5] B = [4,5,6,7,8]
    - add: A + B # => [1,2,3,4,5,4,5,6,7,8]
    - difference: A - B # => [1,2,3] B - A # =>[6,7,8]
    - union: A | B # => [1,2,3,4,5,6,7,8] B | A # =>[4,5,6,7,8,1,2,3]
    - intersect: A & B # => [4,5]
- Some other helpful methods in Array: arr = [1,2,3,4,5]
    - **first:** return the first element of an array (arr[0])
      ```
      arr.first   => 1
      ```
    - **last:** return the last element of an array (arr[-1])
      ```
      arr.last    => 5
      ```
    - **length** or **size:** return the number of elements in an array
      ```
      arr.length  => 5          arr.size => 5
      ```
    - **empty?:** return true if the array is empty, false otherwise
      ```
      arr.empty?  => false
      ```
    - **include?(element):** return true if the array has element, false otherwise
      ```
      arr.include?(3) => true   arr.include?(10) => false
      ```
    - **index(element):** return the index of the first occurrence, nil if not found
      ```
      arr.index(4) => 3         arr.index(10) => nil
      ```
    - **sort** or **sort!:** sort the array
    - **reverse** or **reverse!:** reverse the array
    - **each{}:** iterate every element without changing the array
      ```
      sum = 0; arr.each{|x| sum += x};puts sum # output: 15
      ```
    - **find{}:** return the first element for which code block returns true
      ```
      arr.find{|x| x % 2 == 0} # => 2
      ```
    - **select{}** or **select!{}:** return a new array containing all elements of the original array for which the block returns
      ```
      arr.select!{|x| x % 2 == 1}   # arr = [1,3,5]
      ```
    - **map{}** or **map!{}:** returns a new array containing the results of running a block on each element of the original array
      ```
      arr.map!{|x| x**2} # arr = [1,4,9,16,25]
      ```
    - **join**(Array=>String): convert the array into a string
      ```
      str = arr.join(",") # => str = "1,2,3,4,5"
      ```

Note: Methods with ! will change the original array instead of creating new one

6. Hash

- Definition: Hash is an unordered collection of key-value pairs
- Creation: Using {} or Hash class constructor
  h = {} or h = {"a"=>1} or h = Hash.new or h = Hash.new(default)
- Indexing: Access value of a hash by using hash[key]
  Example:
  ```
  h1 = Hash.new
  h2 = Hash.new 0
  puts h1["hello"] #output: nil
  puts h2["hello"] #output: 0
  ```
- Adding: h[key] = value
  Note: if key already exists, then the new value will be assigned to old key
- Deleting: h = {"a"=>1, "b"=>2, "c"=>2}
  + delete(key): h.delete("b") # => h = {"a"=>1, "c"=>2}
  + delete_if{|key,value| condition}: delete when condition is true
    h.delete_if{|k,v| v == 2} # => h = {"a"=>1}
- Some helpful methods in Hash: h = {"a"=>1, "b"=>2, "c"=>3}
  + **each{}:** iterate over the key-value pairs of a hash
    sum = 0; h.each{|k,v| sum += v}; puts sum #output: 6
    Note: h.each{|x| ...} will iterate the hash like an array
  + **keys:** return an array of all keys in hash
  + **values:** return an array of all values in hash
  + **length:** return the number of key-value pairs in a hash
    puts h.length #output: 3
  + **has_key?(key):** return true if a hash has a given key, false otherwise
  + **has_value?(value):** return true if a hash has a given value
  + **merge(Hash):** merge two hash into a single hash, giving priority to the values of the second hash in case of overlapping keys.
    k = {"a"=>10, "d"=>4}
    puts h.merge(k) #output: {"a"=>10, "b"=>2, "c"=>3, "d"=>4}
    puts k.merge(h) #output: {"a"=>1, "d"=>4, "b"=>2, "c"=>3}
  + **select{|key,value| condition}:** returns a new hash that includes only the key-value pairs that meet a certain condition
  + **reject{|key,value} condition}:** similar to select, pairs fails condition
  + **sort, to_a(Hash => Array), invert, fetch,...**

7. Boolean

- A boolean value is either true or false
- ALL values in Ruby are evaluated to true, EXCEPT for false and nil
  Note: FalseClass and NilClass do not evaluate to false
  ```
  if 0; puts "hello"; end       #output: "hello"
  if nil; put "hello"; end      #output: (nothing)
  if NilClass; put "hello"; end #output: "hello"
  ```

5. Control Flow

1. **if-else statement**
   ```
   if condition
     statements
   elsif condition
     statements
   else
   ```
   Note: **unless condition** is equivalent to **if !condition**
   We can do something like:
   ```
   a = 10 if false     #a = nil (or previous value)
   a = 10 unless false #a = 10
   ```
2. **for loop:** executes a block of code for each element in a collection
   for i in collection; statements; end
3. **while loop:** executes a block of code while a certain condition is true
   while condition; statements; end
4. **case-when**(switch-case):
   ```
   case var
   when value_1; statement #Note: can enter a new line instead of using ;
   when value_2; statement
   ...
   else; statement #Default statement (optional)
   end
   ```
5. **others:**
   until: similar to while but stop when condition is true
   each: (0..5).each do |i|; puts i; end #Notes: (0..5) is Range in Ruby

#Note: (0..5) has integer values from 0 to 5 (inclusive), while (0...5) is from 0 to 4

6. Functions
- Method definition: using **def** keyword, followed by the method name and parameters
  Example:
```
def greet(name)
    puts "Hello, #{name}!"
end
```
- Method call: like other languages. Example: greet("Henry") #Note:() are optional
- Return value: by default, **the last expression** in a method is the return value
  You can also explicitly return a value using the return keyword
  Example: puts greet("Henry").class #output: NilClass (since puts return nil)
- Default parameters: You can provide default values for parameters in case they are not passed in when the method is called
```
def greet(name = "World")
    puts "Hello, #{name}!"
end
greet"Henry"      #output: Hello, Henry!
greet             #output: Hello, World!
```
- Variable number of arguments: by using *
```
def sum(*num); puts num.class; end
sum(1,2,3,4,5)    #output: Array
```
- Block: call block using **yield** keyword

7. Object Oriented Programming
- Definition: In Ruby, everything is an object, and every object is an instance of a class. Classes define the behavior and attributes of objects, and objects inherit the behavior and attributes from their class
  Example:
```
a = "Hello"
a.class          #String     true.class        #TrueClass
3.class          #Integer    nil.class         #NilClass
3.14.class       #Float      1.class.class     #Class
```
- Class: A blueprint for creating objects. Classes are defined using **class** keyword
- Object: An instance of a class. Objects are created using **new** method
- Variables:
  + Instance variables: variables that store the state of an object (denoted by **@**)
  + Class variables: variables that are shared across all objects (denoted by **@@**)
- Methods: Methods are behaviors that objects created from a class can perform
  + initialize: is called when a new object is created (similar to constructor)
  + Accessor methods: methods that provide access to instance variables
    - **attr_reader:** getter (read only)
    - **attr_writer:** setter (write only)
    - **attr_accessor:** both (read and write)
  + Class methods: can be called only on the class, not on objects
    Class methods are defined using the **def self.method_name** syntax
  + Normal methods: similar to defining functions, but inside class
- Inheritance: The child class is defined using **<** operator
- Polymorphism: Ruby allows us to **override** methods but **not overload** (2 methods with the same name but different arguments, the latter method will be used for the class)
- Example of a class:
```
class Car
    attr_accessor :make, :model   #Accessor methods
    @@total = 0                   #Class variable
    def initialize(make, model)   #Constructor
        @make = make              #Instance variable
        @model = model            #Instance variable
        @@total += 1
    end

    def run                       #Normal method
        puts "#{@make} #{@model} is running"
    end

    def self.print_total          #Class method
        puts "The number of cars is #{@@total}!"
    end
end
car_1 = Car.new("Honda", "Civic"); car_1.run; Car.print_total; car_1.make = "BMW"
```

8. Code Blocks
- Definition: Codeblock is a section of code that can do some specific tasks. Codeblocks don't have a name, and they **are not objects**
- Creation: A codeblock is defined between **{}** or **do...end**, it can take arguments

```
Example: Array.new(3){Array.new(3){|x| x + 1}}  # => [[1,2,3],[1,2,3],[1,2,3]]
         def fun3; yield 3,4; end;  func3{|a,b| puts a+b} # => 7
```

- Associate with Functions/Methods:
  + **block_given?** : return true if there is a given block, false otherwise
  + **yield** : call the given codeblock to function
- Proc: Make codeblocks into objects, and have to be called, not yielded to.
  + Example:

```
def fun(p)
      p2 = Proc.new { p.call }
      puts "Inside fun"
      return p2
end
pr = Proc.new{puts "Outside fun"}
p3 = fun(pr)      #output: "Inside fun"
puts p3.class     #output: Proc
p3.call           #output: "Outside fun"
```

  + Procs can be strung together:

```
def say(y)
      t = Proc.new{|x| Proc.new{|z| x + y + z}}
      return t
end
s = say(2).call(3)
puts s.class      #output: Proc
puts s.call(4)    #output: 9
```

9. Files
- Open: f = File.open(file_name); puts f.class #output: File (default is open to read)
- Read:
  + gets: read a single line — puts f.gets.class #output: String or NilClass
  + readlines: read whole file — puts f.readlines.class #output: Array (only)
- Write: f = File.open(file_name, "w"); f.puts("Hi") #file now has "Hi"
- Append: f = File.open(file_name, "a"); f.puts("Hi") #append "Hi" to end of file
- Close: f.close #return nil
- What if we want to add data in between 2 lines? Solution: read, insert, then write

```
=>    f = File.open(file_name, "r")
      lines = f.readlines
      # insert data into lines
      lines.insert(index, line)
      f = File.open(file_name, "w")
      f.puts lines
      f.close
```

10. Regular Expressions
- Definition: A pattern that describes a set of strings
  + An Alphabet: symbols in the string
  + Concatenation
  + Branching (or)
  + Grouping
  + Repetition
- Syntax: /pattern/
- Details (SKIP FOR NOW)