Ruby's Building Blocks

Module 3

Arrays

- Ruby's arrays and hashes are indexed collections. Both store collections of objects, accessible using a key. With arrays, the key is an integer, whereas hashes support any object as a key. Both arrays and hashes grow as needed to hold new elements.
- A new array can be created by using the literal constructor [ ].
- Arrays can contain different types of objects.
- a = [1, "two", 3.0]
- An array can also be created by explicitly calling "Array.new".
- a = Array.new #=> **[ ]**
- Array.new(3) #=> **[nil, nil, nil]**
- Array.new(3, true) #=> **[true, true, true]**
- To build up multi-dimensional arrays a block can be passed.
- a = Array.new(3) {Array.new(3)}
  - #=> **[[nil, nil, nil], [nil, nil, nil], [nil, nil, nil]]**

Accessing Elements

- Elements in an array can be retrieved using the Array[] method.

- It can take a single integer argument (a numeric index), a pair of arguments (start and length) or a range.

- Negative indices start counting from the end, with -1 being the last element.

Accessing Elements Ex:

- arr = [1, 2, 3, 4, 5, 6]
- arr[2] #=> **3**
- arr[100] #=> **nil**
- arr[-3] #=> **4**
- arr[2, 3] #=> **[3, 4, 5]**
- arr[1..4] #=> **[2, 3, 4, 5]**
- arr[1..-3] #=> **[2, 3, 4]**
- Another way to access a particular array element is by using the at method
- arr.at(0) #=> **1**

- a = Array.new a[4] = "4“;
- a[0, 3] = ['a', 'b', 'c']
- a[1..2] = [1, 2]
- a[0, 2] = "?"
- a[0..2] = "A"
- a[-1] = "Z"
- a[1..-1] = nil
- a[1..-1] = []
- a[0, 0] = [1, 2]
- a[3, 0] = "B"

- a = Array.new a[4] = "4"; #=> **[nil, nil, nil, nil, "4"]**
- a[0, 3] = ['a', 'b', 'c'] #=> **["a", "b", "c", nil, "4"]**
- a[1..2] = [1, 2] #=> **["a", 1, 2, nil, "4"]**
- a[0, 2] = "?" #=> **["?", 2, nil, "4"]**
- a[0..2] = "A" #=> **["A", "4"]**
- a[-1] = "Z" #=> **["A", "Z"]**
- a[1..-1] = nil #=> **["A", nil]**
- a[1..-1] = [] #=> **["A"]**
- a[0, 0] = [1, 2] #=> **[[1, 2], "A"]**
- a[3, 0] = "B" #=> **[[1, 2],"A", "B"]**

- arr.fetch(100) #=> **IndexError: index 100 outside of array bounds:**
- arr.first #=> **1**
- arr.last #=> **6**
- To return the first n elements of an array, use take
- arr.take(3) #=> **[1, 2, 3]**
- The drop does the opposite of take, by returning the elements after n elements have been dropped
- arr.drop(3) #=> **[4, 5, 6]**

- To query an array about the number of elements it contains, use length, count or size.
- browsers = ['Chrome', 'Firefox', 'Safari', 'Opera', 'IE']
- browsers.length #=> **5**
- browsers.count #=> **5**
- browsers.empty? #=> **false**
- browsers.include?('Konqueror') #=> **false**

Adding Items to Arrays

- Items can be added to the end of an array by using either push or <<
- arr = [1, 2, 3, 4]
- arr.push(5) #=> **[1, 2, 3, 4, 5]**
- arr << 6 #=> **[1, 2, 3, 4, 5, 6]**
- unshift will add a new item to the beginning of an array.
- arr.unshift(0) #=> **[0, 1, 2, 3, 4, 5, 6]**

- a = ['ant', 'bee', 'cat', 'dog', 'elk']
- a[0] # => "ant"
- a[3] # => "dog"
- # this is the same if you don’t wish to add quotes and commas:
- a = %w{ ant bee cat dog elk }
- a[0] # => "ant"
- a[3] # => "dog“
- Ruby hashes are similar to arrays. A hash literal uses braces rather than square brackets. The literal must supply two objects for every entry: one for the key, the other for the value. The key and value are normally separated by =>.
- inst_section = { ‘cello' => 'string‘, 'clarinet' => 'woodwind', 'drum' => 'percussion‘} #Key is String ‘cello’
- p inst_section['cello'] # "string"

- inst_section = {:cello => 'string', :clarinet => 'woodwind', :drum => 'percussion‘} #Key is symbol because we have ‘:’ before the object
- inst_section[:oboe] # => "woodwind"
- inst_section['cello'] # => nil
- inst_section = {cello: 'string', clarinet: 'woodwind‘}
- inst_section[:cello] #another way
- h = { 'dog' => 'canine', 'cat' => 'feline', 'donkey' => 'asinine' }
- h.length # => 3
- h['dog'] # => "canine"
- h['cow'] = 'bovine'
- h[12] = 'dodecine'
- h['cat'] = 99
- h # => {"dog"=>"canine", "cat"=>99, "donkey"=>"asinine", "cow"=>"bovine", 12=>"dodecine"}

Word Frequency: Using Hashes and Arrays

- Calculates the number of times each word occurs in some text.
- Let's start with the method that splits a string into words:

```
def words_from_string(string)
string.downcase.scan(/[\w']+/)
end
```

- This method uses two very useful String methods: downcase returns a lowercase version of a string, and scan returns an array of substrings that match a given pattern. In this case, the pattern is [\w']+, which matches sequences containing "word characters" and single quotes.
- p words_from_string("But I didn't inhale, he said (emphatically)")
- produces:
- ["but", "i", "didn't", "inhale", "he", "said", "emphatically"]

Create a hash object using Hash.new(0), the parameter (0 in this case) will be used as the hash's default value—it will be the value returned if you look up a key that isn't yet in the hash. Using that, we can write our count_frequency method:

```
def count_frequency(word_list)
    counts = Hash.new(0)
for word in word_list
counts[word] += 1
end
    counts
end
```

- p count_frequency(["sparky", "the", "cat", "sat", "on", "the", "mat"])

produces:

- {"sparky"=>1, "the"=>2, "cat"=>1, "sat"=>1, "on"=>1, "mat"=>1}

Each new word will be added into has as key elements and value of 1 will be assigned unless untill if same word is found back it increments the value of that particular key.

Ex: { ..., "the" => 1, ... } counts[next_word] += 1 { ..., "the" => 2, ... }

Your Task read from text file and do the same above task in Lab

- With insert you can add a new element to an array at any position.
- arr.insert(3, 'apple') #=> **[0, 1, 2, 'apple', 3, 4, 5, 6]**

Removing Items from an Array:

- arr = [1, 2, 3, 4, 5, 6]
- arr.pop #=> **6** arr #=> **[1, 2, 3, 4, 5]**
- To retrieve and at the same time remove the first item, use shift :
- arr.shift #=> **1** arr #=> **[2, 3, 4, 5]**
- arr.delete_at(2) #=> **4** arr #=> **[2, 3, 5]**
- To delete a particular element anywhere in an array, use delete:
- arr = [1, 2, 2, 3]
- arr.delete(2) #=> **2** arr #=> **[1,3]**

- A useful method if you need to remove nil values from an array is compact:
- arr = ['foo', 0, nil, 'bar', 7, 'baz', nil]
- arr.compact! **#=> ['foo', 0, 'bar', 7, 'baz']**
- arr **#=> ['foo', 0, 'bar', 7, 'baz']**
- To remove duplicate elements from an array:
- arr = [2, 5, 6, 556, 6, 6, 8, 9, 0, 123, 556]
- arr.uniq **#=> [2, 5, 6, 556, 8, 9, 0, 123]**
- Set Intersection :
- [1, 1, 3, 5] & [3, 2, 1] **#=> [1, 3]**
- Concatenation: [1, 2, 3] + [4, 5] **#=> [1, 2, 3, 4, 5]**
- Difference: [1, 1, 2, 2, 3, 3, 4, 5] - [1, 2, 4]
- **#=> [3, 3, 5]**

- a = ["a", "b", "c", "d", "e"]
- a.clear #=> **[ ]**
- **bsearch {|x| block } → elem**
- By using binary search, finds a value from this array which meets the given condition in block. #O(log n).
- ary = [0, 4, 7, 10, 12]
- ary.bsearch {|x| x >= 4 } **#=> 4**
- ary.bsearch {|x| x >= 6 } **#=> 7**
- **Returns the first element which matches the criteria.**

- Arrays with blocks: Array.new(4) {|i| i.to_s } **#=> ["0", "1", "2", "3"]**
- Array({:a => "a", :b => "b"}) **#=> [[:a, "a"], [:b, "b"]]**
- arr = [1, 2, 3, 4, 5]
- arr.each {|a| print a -= 10, " "}
- **# prints: -9 -8 -7 -6 -5**
- **#=> [1, 2, 3, 4, 5] #original array remain unchanged**
- reverse_each:
- words = %w[first second third fourth fifth sixth]
- str = ""
- words.reverse_each {|word| str += "#{word} "}
- p str **#=> "sixth fifth fourth third second first "**

- The map method can be used to create a new array based on the original array, but with the values modified by the supplied block.
- arr.map {|a| 2*a} #=> **[2, 4, 6, 8, 10]**
- arr #=> **[1, 2, 3, 4, 5]**
- arr.map! {|a| a**2} #=> **[1, 4, 9, 16, 25]**
- arr #=> **[1, 4, 9, 16, 25]**
- Non-destructive Selection
- arr = [1, 2, 3, 4, 5, 6]
- arr.select {|a| a > 3} #=> **[4, 5, 6]**
- arr.reject {|a| a < 3} #=> **[3, 4, 5, 6]**
- arr.drop_while {|a| a < 4} #=> **[4, 5, 6]**
- arr #=> **[1, 2, 3, 4, 5, 6]**
- Destructive Selection
- Original array elements will be changed **select! and reject!** are the corresponding destructive methods

- [1, 2, 3] * 3 #=> **[1, 2, 3, 1, 2, 3, 1, 2, 3]**
- [1, 2, 3] * "," #=> **“1,2,3”**

- **Collect:**
- a = ["a", "b", "c", "d"]
- a.collect {|x| x + "!"}
- |x| for each element in block, the value gets returned from index 0 until end and that will be added with “!”.
- **#=> ["a!", "b!", "c!", "d!"]**
- **count {|item| block} → int:**
- ary = [1, 2, 4, 2]
- ary.count {|x| x%2 == 0} **#=> 3**
- **Each and Collect:**
- **a=[2,3,4,5]**
- **a.collect{|x|} # It needs a return variable**
- **=> [nil, nil, nil, nil]**
- **a.each{|x|}** **#each method** doesn't modify the array itself, no return value.
- **=>[2,3,4,5]**

- Fill :

- a = ["a", "b", "c", "d"]
- a.fill("x") #=> **["x", "x", "x", "x"]**
- a.fill("z", 2, 2) #=> **["x", "x", "z", "z"]**
- a.fill("y", 0..1) #=> **["y", "y", "z", "z"]**
- a.fill {|i| i*i} #=> **[0, 1, 4, 9]**

- **flatten → new_ary:**
- Returns a new array that is a one-dimensional flattening of self (recursively).
- s = [1, 2, 3] **#=> [1, 2, 3]**
- t = [4, 5, 6, [7, 8]] **#=> [4, 5, 6, [7, 8]]**
- a = [s, t, 9, 10]

  #=> [[1, 2, 3], [4, 5, 6, [7, 8]], 9, 10]
- a.flatten **#=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]**

- **hash → integer**
- A hash function is a function that takes an input (in this case, an array) and returns a fixed-size output, usually a numeric value, called a hash code or hash value.
- arr = [1, 2, 3]
- arr.hash #=> 1831741227259963022
- The hash method generates a unique hash code based on the contents of the array.
- If two arrays have the same contents, they will have the same hash code.
- However, if the contents of the array change, the hash code will also change.

- **slice(index) → obj**
- **slice(start, length) → new_ary or nil**
- **slice(range) → new_ary or nil**
- **a.slice(1) => "b“**
- **a.slice(1..3) => ["b", "c", "d"]**
- **a.slice(1,3)=> ["b", "c", "d"]**
- **a.slice(1,2)=> ["b", "c"]**
- **a.slice(-3,3) => ["c", "d", "e"]**

sort → new_ary

- ary = ["d", "a", "e", "c", "b"]
- ary.sort **#=> ["a", "b", "c", "d", "e"]**

- **take(n) → new_ary:**
- a = [1, 2, 3, 4, 5, 0]
- a.take(3) #=> **[1, 2, 3]**
- **take_while {|obj| block} → new_ary**
- a = [1, 2, 3, 4, 5, 0]
- a.take_while {|i| i < 3} #=> **[1, 2]**
- **transpose → new_ary**
- a = [[1,2], [3,4], [5,6]]
- a.transpose #=> **[[1, 3, 5], [2, 4, 6]]**

Ranges

- (1...5).to_a # => [1, 2, 3, 4]
- (1..10).step(2).to_a # => [1, 3, 5, 7, 9]

- (Date.new(2024, 1, 1)..Date.new(2024, 1, 5)).to_a
- # [2024-01-01, 2024-01-02, 2024-01-03, 2024-01-04, 2024-01-05]

- ('a'..'f').include?('c') # => true
- ('a'..'f').cover?('z') # => false

Numbers

- require 'bigdecimal'
- a = BigDecimal("0.1") + BigDecimal("0.2")
- b = BigDecimal("0.3")
- a == b # => true

- c1 = Complex(2, 3) # 2 + 3i
- c2 = Complex(1, -1) # 1 - i
- c1 + c2 # => Complex(3, 2)

- Math.sqrt(16) # => 4.0
- Math.log(10) # => 2.302585092994046
- Math.sin(Math::PI / 2) # => 1.0

- rand # => random float between 0.0 and 1.0
- rand(100) # => random integer between 0 and 99

String

- A String object holds and manipulates an arbitrary sequence of bytes, typically representing characters.
- "Ho! " * 3 **#=> "Ho! Ho! Ho! “**
- **str + other_str → new_str:**
- str1 = "Hello, "
- str2 = "world!"
- result = str1 + str2
- puts result #=> "Hello, world!“
- **Interpolation:** #{}
- name = "Alice" puts "Hello, #{name}!" #=> "Hello, Alice!"

- num = 3.14159
- str = sprintf("The value of pi is %.2f", num)
- puts str
- **Substring manipulation:**
- str = "Hello, world!“
- puts str[0]
- puts str[7, 5]
- puts str[7..11]
- **String case manipulation**
- str = "Hello, World!“
- puts str.upcase
- puts str.downcase
- puts str.capitalize
- puts str.swapcase

- **String formatting**
- num = 3.14159
- str = sprintf("The value of pi is %.2f", num)
- puts str #=> "The value of pi is 3.14“
- **Substring manipulation:**
- str = "Hello, world!“
- puts str[0] #=> "H"
- puts str[7, 5] #=> "world"
- puts str[7..11] #=> "world“
- **String case manipulation**
- str = "Hello, World!“
- puts str.upcase #=> "HELLO, WORLD!"
- puts str.downcase #=> "hello, world!"
- puts str.capitalize #=> "Hello, world!"
- puts str.swapcase #=> "hELLO, wORLD!"

- **String trimming:**
- str = " hello, world "
- trimmed_str = str.strip
- puts trimmed_str #=> "hello, world”

Containers

- Container is a general term used to refer to data structures that hold and organize multiple values. There are several built-in container types in Ruby, such as arrays and hashes, which allow you to store and manipulate collections of data.
- **Arrays:** An array is an ordered collection of objects, which can be of any data type. Arrays are represented by square brackets ([]) and can be created by listing the elements inside the brackets, separated by commas.
- **Hashes:** unordered collection of key-value pairs, where each key is unique. Hashes are represented by curly braces ({})
- hash2 = { 1 => "one", 2 => "two", 3 => "three" }
- puts hash2[2] #two
- **Sets:** A set is an unordered collection of unique elements.
- set1 = Set.new([1, 2, 3, 4, 5])
- set2 = Set.new([3, 4, 5, 6, 7])
- puts set1 | set2 # {1, 2, 3, 4, 5, 6, 7}

- require 'set'
- set1 = Set.new([1, 2, 3])
- set2 = Set.new([3, 4, 5])
- set1.union(set2) # => #{1, 2, 3, 4, 5}
- set1.intersection(set2) #{3}

- set1.difference(set2) # {1, 2}
- set1.subset?(set2) # => false

Stack and Queue

- stack = []
- stack.push(1)
- stack.push(2)
- stack.pop # => 2
- stack # => [1]

```
require 'thread'
queue = Queue.new
queue.push(1)
queue.push(2)
queue.pop # => 1
```

Implementing Stack with Class:

```
class Stack
  def initialize
    @elements = []
  end

  def push(element)
    @elements.push(element)
  end

  def pop
    @elements.pop
  end

  def top
    @elements.last
  end

  def empty?
     @elements.empty?
   end
end

stack = Stack.new
stack.push(1)
stack.push(2)
stack.pop # => 2
```

Balanced Parentheses Check Using Stack

```
def balanced_parentheses?(string)
  stack = [ ]
  pairs = { '(' => ')', '{' => '}', '[' => ']' } #hash key and value

  string.each_char do |char|
    if pairs.keys.include?(char)
      stack.push(char)
    elsif pairs.values.include?(char)
      return false if stack.empty? || pairs[stack.pop] != char
    end
  end

  stack.empty?
end

puts balanced_parentheses?("({[]})") # => true
puts balanced_parentheses?("({[})")  # => false
```

Try

- expression = ['2', '3', '+', '4', '*']
- # Equivalent to (2 + 3) * 4
- puts evaluate_postfix(expression) # => 20

```ruby
# Creating a hash
student = {
  "name" => "John Doe",
  "age" => 20,
  "grade" => "A"
}

# Accessing hash values
puts "Name: #{student["name"]}"
puts "Age: #{student["age"]}"
puts "Grade: #{student["grade"]}"

# Modifying hash values
student["age"] = 21
student["grade"] = "B"

puts "Modified Age: #{student["age"]}"
puts "Modified Grade:
    #{student["grade"]}"

# Adding new key-value pairs
student["city"] = "New York"
student["country"] = "USA"

puts "City: #{student["city"]}"
puts "Country: #{student["country"]}"

# Iterating over hash
student.each do |key, value|
  puts "#{key}: #{value}"
end

# Removing a key-value pair
student.delete("grade")

puts "After deleting 'grade':"
student.each do |key, value|
  puts "#{key}: #{value}"
end
```

O/P


Name: John Doe
Age: 20
Grade: A
Modified Age: 21
Modified Grade: B
City: New York
Country: USA
name: John Doe
age: 21
grade: B
city: New York
country: USA
After deleting 'grade':
name: John Doe
age: 21
city: New York
country: USA

Min Stack

```
class MinStack
  def initialize
    @stack = []
    @min_stack = []
  end

  def push(x)
    @stack.push(x)
    if @min_stack.empty? || x <= @min_stack.last
      @min_stack.push(x)
    end
  end

  def pop
    return if @stack.empty?
    popped = @stack.pop
    @min_stack.pop if popped == @min_stack.last
  end

  def get_min
     @min_stack.last
   end
end

# Usage
min_stack = MinStack.new
min_stack.push(3)
min_stack.push(5)
puts min_stack.get_min # Output: 3
min_stack.push(2)
min_stack.push(1)
puts min_stack.get_min # Output: 1
min_stack.pop
puts min_stack.get_min # Output: 2
```

```
# Inventory Management System # Create an empty inventory hash

inventory = {}
# Function to add an item to the inventory
def add_item(inventory)
  puts "Enter the item name:"
  name = gets.chomp
#Chomp Used to remove the trailing newline character (\n) from a string.
  puts "Enter the quantity:"
  quantity = gets.chomp.to_i

  puts "Enter the price per unit:"
  price = gets.chomp.to_f

  inventory[name] = { quantity: quantity, price: price }

  puts "#{name} has been added to the inventory."
end
```

```
# Function to remove an item from the inventory

def remove_item(inventory)
  puts "Enter the item name to remove:"
  name = gets.chomp

  if inventory.key?(name)
    inventory.delete(name)
    puts "#{name} has been removed from the inventory."
  else
    puts "#{name} is not found in the inventory."
  end
end
```

```
# Function to display the current inventory

def display_inventory(inventory)
  if inventory.empty?
    puts "The inventory is empty."
  else
    puts "Current Inventory:“
# Iterate over each key-value pair in the inventory hash.
    inventory.each do |name, data|
# accesses the value of the :quantity key within the data
      quantity = data[:quantity]
      price = data[:price]
      total_value = quantity * price
      puts "Item: #{name}, Quantity: #{quantity}, Price per unit: $#{price}, Total
     Value: $#{total_value}"
    end
  end
end
```

```
# Main program loop
loop do
  puts "Select an option:"
  puts "1. Add an item"
  puts "2. Remove an item"
  puts "3. Display inventory"
  puts "4. Exit"

  option = gets.chomp.to_i

  case option
  when 1
    add_item(inventory)
  when 2
    remove_item(inventory)
  when 3
    display_inventory(inventory)
  when 4
    break
  else
    puts "Invalid option. Please try
    again."
  end

  puts "\n"
end

puts "Exiting the program.
    Goodbye!"
```

O/P

```
C:\Users\HOME\Desktop\Ruby\Ruby Programs>ruby hash2.rb
Select an option:
1. Add an item
2. Remove an item
3. Display inventory
4. Exit
1
Enter the item name:
Pen
Enter the quantity:
5
Enter the price per unit:
100
Pen has been added to the inventory.

Select an option:
1. Add an item
2. Remove an item
3. Display inventory
4. Exit
3
Current Inventory:
Item: Pen, Quantity: 5, Price per unit: $100.0, Total Value: $500.0
```