Lecture 10
CSE/ISE 337 Scripting Languages
Ruby 2 : Lecture 10

Instr: Prof Abid M. Malik
2/21/2023

Last Time

- Ruby Basics
- Input/output
- Data Types and Objects
- Variables
- Ruby Syntax
- Ruby Keywords
- Strings the main data type
- Expression and Operators
- Statement Control statements

Equal? - used for checking shallow or deep copying

```
str1 = "hello"
str2 = "hello"
str3 = str1

puts str1.equal?(str2)  # Output: false
puts str1.equal?(str3)  # Output: true
```


Eql? To check if two objects are same type

```
puts 5.eql?(5)   # Output: true
puts 5.eql?(5.0) # Output: false

str1 = "hello"
str2 = "hello"
puts str1.eql?(str2)  # Output: true
```

Practice Homework

Write a function to Reverse a string without using String#reverse() method

- Do it with all loops available in Ruby (do, until, for … loops)
- Do it with the -ve indexing as well!
- Do the slicing of string as well
- Apply different string methods/ include in the function to make it more useful!

Solution HW1

```ruby
def reverse_string(str)
  reversed = ''
  i = str.length - 1
  while i >= 0
    reversed += str[i]
    i -= 1
  end
  return reversed
end
```

```ruby
original_str = "Hello, world!"
reversed_str = reverse_string(original_str)
puts reversed_str  # Output: "!dlrow ,olleH"
```

Do it with the -ve indexing as well!

Today's Agenda

- Array
- Hash
- Methods
  - Proc, Block, and Lambda
- Iterators
- More on Range Operators
- I/O File operations
- Exceptions

Function convention ? or !

- In Ruby convention, a method name ending with a question mark"?" often implies that the method returns a boolean value. It's commonly used to indicate that the method is a predicate method, meaning it tests some condition and returns true or false.

```
def is_even?(number)
  number % 2 == 0
end
puts is_even?(5)  # Output: false
puts is_even?(6)  # Output: true
```

Function convention ? or !

- **Adding an exclamation mark "!" to a method name usually indicates that the method may modify the object it's called on or performs a potentially dangerous or irreversible operation. It's often used to denote a "bang" or dangerous version of a method. For example, `sort!` would modify the array in place instead of returning a new sorted array.**

```
array = [3, 1, 2]

array.sort!

puts array  # Output: [1, 2, 3]
```


Blocks

A ***block*** is the same thing as a **method**, but it does not belong to an object. There are some important points about Blocks in Ruby:

- Block can accept arguments and returns a value.
- Block does not have their own name.
- Block consist of chunks of code.
- A block is always invoked with a function or can say passed to a method call.
- To call a block within a method with a value, **yield** statement is used.
- Blocks can be called just like methods from inside the method that it is passed to.

Blocks

- Syntax 1 :

```
block_name {

   #statement-1
   #statement-2
   .
   .
}
```

- Syntax 2:

```
block_name do

   #statement-1
   #statement-2
   .
   .
end
```


Examples on Jubyter Notebook

LINE BREAKS ARE SIGNIFICANT
Syntax issue!

```
3.times do |i|
   puts( i )
end

3.times { |i|
   puts( i )
}
```

```
3.times
do |i|
   puts( i )
end

3.times
{ |i|
   puts( i )
}
```


Concept of Yield

- Similar to Generators
- When a function uses a block as an argument, it calls block through **yield** command
- Check the jupyter notebook Example 4-A

Proc

- ***Procs*** that is very much similar to block but with a few differences like a *procs* is assigned or store in a variable and it is executed by **calling .call method.**
- You can pass one or more proc to a method. As we know that block is not an object but Proc is an object.
- It is a block that turned into an object of the **Proc class,** that why the Proc looks similar to the instantiation of a class

Proc

Syntax:

```
variableName = Proc.new {# Statement}
```

```
# Executing Procs
 variableName.call
```

Passing Block as Argument to a function

- **See Example on Jupyter notebook**

Blocks and Procs are closure

- What are closures?
- You find it in all functional languages including Python
- **A closure is a block of code that can be passed around as a value, stored in variables, and executed later. Closures capture the surrounding state (variables, constants, and methods) and carry it along with them for later execution.**

Blocks within Blocks

Code on the Jupyter notebook

```
["hello","good day","how do you do"].each{
  |s|
  caps( s ){ |x| x.capitalize!
    puts( x )
  }
}
```

What is the output?

Lambda

***lambda* and *Procs* are kind of same** there isn't much difference.

Lambda is like a regular method because they impose the number of parameters passed when they're called and also they return like normal methods(It treat return keyword just like methods)

variableName = `->` **{# Statement}**

You can use the "lambda" keyword as well

BEGIN and END are also blocks

```
# Ruby Program of BEGIN and END Block
BEGIN {
# BEGIN block code
puts "BEGIN code block"
}
END {
# END block code
puts "END code block"
}
# MAIN block code
puts "CSE/ISE337"
```

See the code on the notebook as well!

Arrays

- Ruby arrays are **ordered, integer-indexed** collections of any object.
- Each element in an array is associated with and referred to by an index.
- Array indexing starts at 0, as in C or Java. A negative index is assumed relative to the end of the array --- that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.
- Ruby arrays can hold objects such as String, Integer, Fixnum, Hash, Symbol, even other Array objects. *Ruby arrays are not as rigid as arrays in other languages. Ruby arrays grow automatically while adding elements to them.*

Creating an Array

There are many ways to create or initialize an array. One way is with the new class method –

```
names = Array.new
```

You can set the size of an array at the time of creating array –

```
names = Array.new(20)
```

The array names now has a size or length of 20 elements. You can return the size of an array with either the size or length methods –

When you create a new array, what elements are in the array

You can check it using

```
>>> p names
```


Creating Arrays with Blocks

Array initialization

- With blocks
- With []
- With range operator
- With Array.new() function

Using [] Operators for initializing an array

```
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]
```

Initializing an Array

- **Using range Operator**
  - **(..)**
  - **(...)**
- **Using %w operator**
  - **This converts the array elements to string**

Accessing Elements

- Through element number
- Through **.at** function
- Through **-ve index**
- Through **.first**
- Through **.last**
- **.index** is used to check the index of an item in the array
- **.slice** is used to partition a given array
- **.include?** check if an element exists

Concatenation

- **+**
  - **Combined two arrays to one**
- **<<**
  - **Used to add an element at the end of a given array**

Set Operations

- **Intersection with &**
- **Difference with -**
- **Union with |**

Array as Stack

- **Push and pop**

```
fruit = %w[apple orange banana]

fruit.pop

print fruit

fruit.push "mango"

print fruit

number = [1,2,3,4,7]

number.pop

print number
```


Arrays as Queue

- `.shift()`
- `.unshift()`
  - Homework: try these function on an array

Comparing Arrays

- ==
- ⇔
- Eql?
  - Homework : try it on array

Array build in methods

- .unique() # unique items in the array
- .insert() # takes index number
- .to_s # convert to string
- .delete ()# clear the array of all items
- .sort!
- .sort_by # map function, takes a function to sort the elements

Arrays and Blocks

- Arrays and blocks are fundamental constructs in Ruby that often work together to perform various operations on collections of data.
- Arrays are ordered, indexed collections of objects. Blocks are anonymous functions that can be passed as arguments to methods in Ruby.

Multidimensional Arrays

```ruby
output = [[1,1,1,1,1,1,1,1,1],
          [2,2,2,2,2,2,2,2,2],
          [3,3,3,3,3,3,3,3,3],
          [4,4,4,4,4,4,4,4,4],
          [5,5,5,5,5,5,5,5,5],
          [6,6,6,6,6,6,6,6,6],
          [7,7,7,7,7,7,7,7,7],
          [8,8,8,8,8,8,8,8,8],
          [9,9,9,9,9,9,9,9,9]]
print output[0][1]
arr2 = output.transpose
print arr2
print arr2.flatten
```

Hashes

- **Hash** is a collection of unique keys and their values.
- Hash is like an Array, except the indexing is done with the help of arbitrary keys of any object type.
- In Hash, the order of returning keys and their value by various iterators is arbitrary and will generally not be in the insertion order.
- The default value of Hashes is ***nil***.
- When a user tries to access the keys which do not exist in the hash, then the ***nil*** value is returned.

Creating Hashes

```
months = Hash.new
months.empty?
months.length
months.size
```

[] : This method creates a new hash that is populated with the given objects. It is equivalent to creating a hash using literal {Key=>value….}.

Keys and values are present in the pair so there is even number of arguments present.

Hash[(key=>value)*]

try_convert : This method is used to convert *obj* into hash and returns hash or nil. It return nil when the *obj* does not convert into hash.

Hash.try_convert(obj)

Accessing Hashes

- Hash.keys # will give array of all keys
- Hash.index "x" # will give the key of value "x" if exists
- Hash.values
- Hash.values_at
- Hash.default(nil=key)
- Hash.to_a # convert to array
- Hash.delete(key)
- Hash.clear
- Hash.merge
- Hash.sort

Try these as homework

Iterating over Hashes

Assignment Expression

- An assignment sets the left variable to the value from right expression, then returns that value as the result
  - E.g., a = 2 >> a is 2; it returns 2
- Since assignments return values, they can be chained or used in other expressions
  - a = b = c = 2 >> c is 2 and returns 2; so b is 2 and returns 2; so a is 2
  - c = a + b = 1 >> c is 3
- Values can be assigned in parallel; right values are evaluated in order of appearance
  - a,b = b,a # swap a and b
  - a,b,c = 0,1,2 >> [0,1,2]
  - a,b,c,d = 0,1,2 >> [0,1,2]; d is nil

Assignment Expression

- You can add asterisk to the variables of an assignment
  - b, *c = 1,2,3,4,5,6 >> b == 1, c == [2,3,4,5,6]
  - *b, c = 1,2,3,4,5,6 >> b == [1,2,3,4,5], c == 6
  - b, *c, d = 1,2,3,4,5,6 >> b == 1, c == [2,3,4,5], d == 6
- The left hand side may have parenthesized variables
  - b, (c,d), e = 1,2,3,4 >> b == 1, c == 2, d == nil, e == 3
  - b, (c,d), e = 1,[2,3],4 >> b == 1, c == 2, d == 3, e == 4
  - b, (c,*d), e = 1,[2,3],4 >> b == 1, c == 2, d == [3], e == 4

Iterators in Ruby

The word ***iterate*** means doing one thing multiple times and that is what ***iterators*** do. Sometimes iterators are termed as the custom loops.

- "Iterators" is the object-oriented concept in Ruby.
- In more simple words, iterators are the **methods which are supported by *collections(Arrays, Hashes etc.)***. Collections are the objects which store a group of data members.
- Ruby iterators return all the elements of a collection one after another.
- Ruby iterators are "chainable" i.e adding functionality on top of each other.

Each Iterator

This iterator returns all the elements of an array or a hash. Each iterator returns each value one by one.

```
collection.each do |variable_name|
   # code to be iterate
end
```

Collect Iterator

This iterator returns all the elements of a collection. The collect iterator returns an entire collection, regardless of whether it is an array or hash.

Syntax:

```
Collection = collection.collect
```


Other iterators (Code in Jupyter)

1. **Times Iterator**
2. **Upto Iterator**
3. **Downto Iterator**
4. **Step Iterator**
5. **Each_Line Iterator**

Range Operator in Ruby

- Ranges as Sequences
- Ranges as Conditions
- Ranges as Intervals

Range for Pattern Matching

Homework practice!

```
while gets
   print if /start/../end/
end
```

Ranges as Intervals

- A final use of the versatile range is as an interval test: seeing if some value falls within the interval represented by the range. This is done using ===, the case equality operator.
- See the code on the Jupyter Notebook in "Range Operator in Ruby" Section

Working with Files

- **You can manipulate file directories (folder) and files from within Ruby programs using method from Dir and File classes**
  - **Directories**
    - **Dir.chdir()**
    - **Dir.ped()**
    - **Dir.mkdir()**
  - **Looking inside Directories**
    - **Dir.entries()**
    - **You can attach block with it!**
    - **Question! I would like to see all files in the directory!**
  - **The Directory Stream**
    - **You can Open a directory stream and have a look around in it**
    - **Dir.open() # This will return a pointer/object to play with**

Creating a New File

- `file = File.new ("file.rb", "w")`

File Opening Modes

"r"	*Read-only*, starts at beginning of file (default)
"r+"	*Read-write*, starts at beginning of file
"w"	*Write-only*, creates a new file for writing
"w+"	*Read-write*, creates a new file for reading and writing
"a"	*Write-only*, appends data to end of file, creates new file if it doesn't exist
"a+"	*Read-write*, creates new file for reading and writing if file does not exist

Use the following in conjunction with the above modes:

"b"	open file in binary mode
"t"	open file in text mode

Opening an Existing File

```
file = file.open("something.txt") # default mode is r

file.each{ |line| print "#{file.lineno}.", line}

file.close
```

ARGV and ARGF

- **ARGV (ARGument Vector) (or $*)** is an array, and each of its elements is a filename submitted on the command line
- **ARGF (ARGument File) ($<)** is a special construct that represents the concatenation of all files mentioned in command-line arguments (ARGV).

Opening a URI

In Ruby, the URI module is used for parsing and manipulating **URIs (Uniform Resource Identifiers),** which include URLs (Uniform Resource Locators) and URNs (Uniform Resource Names). It provides classes and methods for working with URIs in a structured and convenient way.

```
Require 'open-uri'
Url = "http://www.google.com/searh?q=ruby"
open(url) { |page| page_content = page.read()}
```

Deleting and Renaming Files

- **File.new()**
- **File.rename()**
- **File.delete()**

File is Ruby Class for manipulating files

File Inquires

- There are many useful methods that can be used to get information about the file
  - `File::exist?()`
  - `File::directory()?`
  - `File::size()`
  - `File::type() # file is a directory or file`
  - `File.chmod() # to change the mode of a given file`

Exception in Ruby

- The execution and the exception always go together. If you are opening a file, which does not exist, then if you did not handle this situation properly, then your program is considered to be of bad quality.
- The program stops if an exception occurs. So exceptions are used to handle various type of errors, which may occur during a program execution and take appropriate action instead of halting program completely.
- Ruby provide a nice mechanism to handle exceptions. We enclose the code that could raise an exception in a begin/end block and use rescue clauses to tell Ruby the types of exceptions we want to handle.

Exceptions in Ruby

```
begin
# -
rescue OneTypeOfException
# -
rescue AnotherTypeOfException
# -
else
# Other exceptions
ensure
# Always will be executed
end
```

- Everything from begin to rescue is protected. If an exception occurs during the execution of this block of code, control is passed to the block between rescue and end.
- For each rescue clause in the begin block, Ruby compares the raised Exception against each of the parameters in turn. The match will succeed if the exception named in the rescue clause is the same as the type of the currently thrown exception, or is a superclass of that exception.
- In an event that an exception does not match any of the error types specified, we are allowed to use an else clause after all the rescue clauses.

Catch and Throw

- The raise-clause is good for abandoning execution; but catch-throw is used to jump out of deeply nested constructs

```
r = catch (:done)
do  i = 0
while i < 10
   throw :done, "Quitting now ..." if i == 5
   puts i
  i = i + 1
end
puts "still in catch"
end
puts r
```

When Ruby encounters a **throw**, it zips back up the call stack looking for a **catch** block with a matching symbol.

When it finds it, Ruby unwinds the stack to that point and terminates the block.

If the **throw** is called with the optional second parameter, that value is returned as the value of the **catch**

Exceptions in Ruby

- **See code on Jupyter notebook**

Next Time

- **Symbols**
- **OO Programming in Ruby**
- **Classes**
- **Modules**
- **Mixin**
- **Regular Expressions**