Programming Paradigms

Unit 3 — Ruby Basics

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

Outline

Meet Ruby

- It's time to start programming: for the object-oriented part we are using Ruby

- Ruby was created by Yukihiro Matsumoto around 1993
- It is an interpreted, object-oriented, dynamically typed language
- Ruby optimizes the simplicity and the productivity of the programmers
- The efficiency of the lanugage is less important
- Official Web page: http://www.ruby-lang.org/en/

Ruby Interpreter

- In Linux, open a shell and type `irb` to start the interactive Ruby interpreter

  `rb(main):001:0>`
- The Ruby interpreter is now open and ready to execute interactively ruby expressions read from stdin
- An input expression is executed when it is syntactically completed (can be one line or multiple lines)

```
>> "Hello World!"
=> "Hello World!"
```

- Every evaluated expression returns a value
- IRB tells us the result of the last evaluated expression
  - In the above expression, it returns simply the string `"Hello World!"`
- Is this the world's shortest "Hello World" program?

Hello World Program

- Now lets look at the Hello World program in Ruby

```
>> puts "Hello World!"
Hello World!
=> nil
```

- `puts` is the basic command to print something
- The `puts` command always returns `nil`
- Alternatively, you could also write: `print "Hello Ruby!\n"`
- Compare this to Java:

```
import java.io.*;
public class Hello {
   public static void main ( String[] args) {
      System.out.println ("Hello Ruby!\n");
   }
}
```

Ruby Source Files

- Sometimes it is desirable to store Ruby programs in a (source) file
  - By convention, Ruby source files should have the extension `.rb`
- Consider the following program stored in `hello.rb`

```
puts "Hello World!"
```

- Can be executed in a (Linux) console using the Ruby interpreter: `ruby <filename>`
  - Gets a script of statements to execute
  - Begins executing at the first line and continues to the last line

```
gamper@carbon:  ruby hello.rb
Hello World!
gamper@carbon:
```

- Alternatively, you can also load the file into `irb`: `load <filename>`

```
>> load 'hello.rb'
Hello World!
=> true
```

Variables and Single-quoted Strings

- There are (at least) two different types of strings
- Simplest string literals are enclosed in single quotes
  - String is interpreted literally
  - The text within the quotes is the value of the string

```
>> language = 'Ruby'
=> "Ruby"
>> puts 'Hello #{language}!'
Hello #{language}!
=> nil
```

- `language` is a local variable
- Local variables start with a lowercase letter or `'_'`
- = is the assignment operator (which returns the variable value)
- Implicit declaration of local variable in first assignment

Double-quoted Strings

- Strings enclosed in double quotes
  - String is evaluated before it is returned
  - Supports expression substitution as a means of embedding the value of any Ruby expression into a string using #{...}

```
>> language = 'Ruby'
=> "Ruby"
>> puts "Hello #{language}!"
Hello Ruby!
=> nil
>> puts "The sum is #{2 + 3}!"
The sum is 5
=> nil
```

Strings with Other Delimiters

- Strings can also be enclosed in a pair of matching though arbitrary delimiter characters preceded by a %
  - e.g., !, (, {, etc.,
  - Supports expression substitution too

```
>> puts %{Ruby is fun.}
>> Ruby is fun.
=> nil

>> puts %(Ruby is fun.)
>> Ruby is fun.
=> nil

>> puts %<The sum is #{2 + 3}!>
The sum is 5
=> nil
```

- ... and there are even more ways to represent strings

Comparisons

- Ruby has the standard comparison (==, >=, !=, etc.) and Boolean (`and`, `or`, `not`) operators

```
>> x = 4
=> 4
>> y = 3
=> 3
>> x == 4
=> true
>> x >= 4
=> true
>> (x == 4) and (y > 5)
=> false
>> (x != 3) or (not (y < 4))
=> true
```

Object-Orientation

- Ruby is a "pure" object-oriented language
- Everything is an object, even numbers, strings and expressions are objects
  - class returns the class of an object
  - methods returns the methods of an object
  - object_id returns the identfier of object

```
>> 4.class
=> Fixnum

>> 4.methods
=> ["%", "odd?", "inspect", "prec_i", ...

>> 4.object_id
=> 9

>> 'Hello World!'.class
=> String

>> (2 + 3).class
=> Fixnum

>> (3 < 4).class
=> TrueClass
```

Outline

Conditionals: `if` and `unless`/1

- Ruby provides `if` and `unless`
- Both come in two flavors
  - block:

    ```
    if condition
       statements
    end
    ```
  - one-line: *statement* `if` *condition*

```
>> x = 4
=> 4

>> if x == 4
>>    puts 'x is equal to 4'
>> end
x is equal to 4
=> nil

>> puts 'x is equal to 4' if x == 4
x is equal to 4
=> nil
```

Conditionals: `if` and `unless`/2

- Even if not really needed, `unless` allows sometimes to express conditions much better than with negation

```
>> y = 3
=> 3

>> unless y == 3
>>    puts 'y is different from 3'
>> end
=> nil

>> puts 'y is different from 3' unless y == 3
=> nil
```

Conditionals: `if` and `unless`/3

- The `if` and `unless` statements also support `else` branches

```
...
>> if x > 4
>>    puts 'x is greater than 4'
>> else
>>    puts 'x is less than 4'
>> end
x is less than 4
=> nil
```

```
...
>> unless y > 3
>>    puts 'y is less than 3'
>> else
>>    puts 'y is greater than 3'
>> end
y is greater than 3
=> nil
```

Conditionals: `case`

- Similar to other languages, Ruby provides a `case`-statement

```
case expr0
when expr1 [, expr2, ...]
   stmt1
when expr3 [, expr4, ...]
   ...
[ else
   stmt3 ]
end
```

  - `10..20` represents a range of numbers

```
>> age = 5
=> 5
>> case age
>> when 0..2, 90..100
>>    puts "baby or old man"
>> when 3..12
>>    puts "child"
>> when 13..18
>>    puts "youth"
>> else
>>    puts "adult"
>> end
child
=> nil
```

Loops

- Ruby has two constructs for loops: `while` and `until`

```
>> z = 0
=> 0

>> while z < 10
>>     z = z + 1
>> end
=> nil

>> z
=> 10
```

```
>> z = 0
=> 0

>> until z > 9
>>     z = z + 1
>> end
=> nil

>> z
=> 10
```

- One-line versions are also supported

```
z = z + 1 while z < 10
z = z + 1 until z > 9
```

Outline

A Strongly Typed Language

- For the most part, Ruby is a strongly typed language

```
>> 4 + 4
=> 8
>> 4 + '4'
TypeError:  String can't be coerced into Fixnum ...
>> 4 + 4.0
=> 8.0
```

Dynamic Typing

- Ruby definitely uses dynamic typing, i.e., type checking takes place when the code is actually executed (not when it is defined)
- Defining a function `add_four_and_four` that adds a number and a string is OK

```
>> def add_four_and_four
>>     4 + 'four'
>> end
=> nil
```

- Calling the function yields a runtime error

```
>> add_four_and_four
TypeError:  String can't be coerced into Fixnum ...
```

Substitutability

- Ruby is very flexible when it comes to substitutability

```
>> a = ['100', 100.0, 100, 'a']
=> ["100", 100.0, 100, "a"]
```

- Variable a is an array that stores a string, a float, and an integer

```
>> i = 0
=> 0

>> while i < 3
>>     puts a[i].to_i
>>     i = i + 1
>> end
100
100
100
0
```

- The method `to_i` is applied to each element and performs a conversion-to-integer function (gives 0 for "a")

Duck Typing/1

- What we have seen on the previous slide is called duck typing
- Duck typing refers to the tendency of Ruby
  - to be less concerned with the class of an object
  - and more concerned with what methods can be called on it and what operations can be performed on it

"If it walks like a duck and swims like a duck and quacks like a duck, it is a duck"

Duck Typing/2

- Duck typing allows a programmer to code to interfaces without a lot of overhead, e.g.,
  - If an object has `push()` and `pop()` methods, you can treat it like a stack
  - If it does not, you cannot
- Nevertheless, duck typing comes at a price
  - All the standard tools and techniques for statically typed languages won't work
  - You cannot catch as many errors automatically as with static typing, making debugging more difficult

Outline

Arrays/1

- Ruby arrays are ordered, integer-indexed collections of any object
- Each element in an array is referred to by an index, starting with 0 for the first element

```
>> animals = ['lions', 'tigers', 'bears']
=> ["lions", "tigers", "bears"]

>> animals[0]
=> "lions"

>> animals[2]
=> "bears"
```

Arrays/2

- There are certain peculiarities about arrays in Ruby
  - You have already seen an array containing objects of many different types
- Accessing elements beyond an array returns nil (not an error!)

```
>> animals[10]
=> nil
```

- Elements can be referenced from the end of the array
  - Index -1 gives the last element

```
>> animals[-1]
=> "bears"
```

```
>> animals[-2]
=> "tigers"
```

- Or a range of items can be selected

```
>> animals[0..1]
=> ["lions", "tigers"]
```

Arrays/3

- Before using a variable to hold an array, it has to be declared as one

```
>> a[0] = 0
NameError:  undefined local variable ...
```

```
>> a = []
=> []
```

```
>> a[0] = 0
=> 0
```

- There are also other ways to declare and create arrays

```
b = Array.new
=> []
```

```
c = Array.new(4)
=> [nil, nil, nil, nil]
```

```
c = Array.new(4,'Ruby')
=> ["Ruby", "Ruby", "Ruby", "Ruby"]
```

Arrays/4

- Though we can specify the size of arrays when we create an array, there is no need to do so
- Ruby arrays grow automatically while adding elements to them

```
>> b[0] = 'a'
=> "a"

>> b[1] = 'b'
=> "b"

>> b[2] = 'c'
=> "c"

>> b
=> ["a", "b", "c"]
```

Arrays/5

- Arrays don't have to be homogeneous, rather they can hold elements of any type, even nested arrays

```
>> a = ['zero', 1]
=> [ "zero", 1]
>> a[2] = ['two', 'things']
=> ["two", "things"]
>> a
=> [ "zero", 1, ["two", "things"]]
```

- Multidimensional arrays are just arrays of arrays

```
>> a = [[1,2,3],[4,5,6],[7,8,9]]
=> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>> a[1][2]
=> 6
```

Ducktyping and Arrays

- Arrays implement a very rich interface

```
>> [].methods
=> [:to_s, :to_a, :first, :last, :concat, :push, :pop
    :shift, :unshift, :insert, :each, :length, :size, ...]
```

- Arrays can be used as queues, linked lists, stacks, sets, etc.
- Example: array as stack

```
>> a = [1]
=> [1]

>> a.push(3)
=> [1, 3]

>> a.pop
=> 3

>> a.pop
=> 1

>> a
=> []
```

Iterating through Arrays

- Any loop can be used to iterate over an array
- But so-called iterator methods make it more elegant, and reduce complexity and possible errors
  - `each` enumerates over each element in the array
  - An iteration variable between two vertical bars is used to access elements

```
>> primes = [2, 3, 5, 7]
=> [2, 3, 5, 7]
>> primes.each do |number|
     puts number
   end
2
3
5
7
```

- In a similar way, the `each_index` iterator provides access to all indexes of an array

Deleting Array Elements

- Deleting array elements

```
>> a = Array["orange", "lemon", "apple"]
=> ["orange", "lemon", "apple"]
```

- Delete element at index 1

```
>> a.delete_at(1)
=> "lemon"

>> puts a
orange
apple
=> nil
```

- Delete element with value "apple"

```
a.delete("apple")
=> "apple"

>> puts a
orange
=> nil
```

Hashes

- A hash is a collection of key-value pairs like this:
  `"employee" => "salary"`
  - The keys are labels, the values are objects
  - We can look up an object in a hash using its label
- A hash is similar to an array, except that indexing is done via arbitrary keys of any object type, not an integer index

```
>> numbers = {'obj1' => 'one', 'obj2' => 'two'}
=> "obj1"=>"one", "obj2"=>"two"

>> numbers['obj1']
=> "one"

>> numbers['obj2']
=> "two"

>> numbers['obj1'] = 'three'
=> "three"

>> numbers['obj3']
=> nil
```

Creating Hashes

- There are many different ways to create hashes
- The following creates an empty hash

```
>> months = Hash.new
=> {}
```

```
>> days = {}
=> {}
```

- A hash with default value `'month'` (which otherwise is just `nil`)
  - Default value is returned if the access key doesn't exist

```
>> months = Hash.new('month')
=> {}
```

```
>> months['jan'] = 'January'
=> "January"
```

```
>> months['jan']
=> "January"
```

```
>> months['feb']
=> "month"
```

Hashes and Labels

- While the previous code works, there is a problem in using strings as labels
  - Every time we change one of the entries, a new string will be created for the label, wasting memory
- Two strings with the same value are stored at different memory locations, hence are different string objects

```
>> 'string' == 'string'
=> true
```

```
>> 'string'.object_id == 'string'.object_id
=> false
```

```
>> 'string'.object_id
=> 19371260
```

```
>> 'string'.object_id
=> 19358460
```

- In contrast to strings, numbers are unique and are fine as labels

```
> 4.object_id == 4.object_id
=> true
```

Symbols/1

- A symbol in Ruby is an identifier preceeded by a colon, e.g., `:symbol`
- A symbol is a unique, immutable string, i.e., it never changes

```
>> :highlander
=> :highlander
```

```
>> :highlander.class
=> Symbol
```

- Every time we use `:highlander`, it will refer to the same object

```
>> :highlander.object_id == :highlander.object_id
=> true
```

- Since a symbol is a special string, we can access the associated string

```
>> :highlander.to_s
=> "highlander"
```

Symbols/2

- Once created, the values of a symbol cannot be changed

```
>> :highlander = "Sean Connery"
syntax error, unexpected '=', expecting $end
```

- Symbols are perfect identifiers to be used for hash labels (instead of strings)

```
>> numbers = {:obj1 => 'one', :obj2 => 'two'}
=> {:obj2=>"two", :obj1=>"one"}
>> numbers[:obj1]
=> "one"
```