Capítulo 9

INTRODUCCIÓN A RUBY

> To the optimist, the glass is half full. To the pessimist, the glass is half empty. To the engineer, the glass is twice as big as it needs to be,
>
> Anónimo.
>
> Si pones mil millones de micos tecleando en computadores, eventualmente uno escribirá un programa en Ruby. Todos los demás lo harán en Perl,
>
> Anónimo.
>
> Llevo usando vim desde hace dos años, principalmente porque no logro dar con el comando para salir,
>
> Anónimo.

No voy a comenzar aquí una soflama a favor o en contra de lenguajes de programación. Cada lenguaje es el resultado de un proceso histórico, unas ventajas en ciertos contextos, y unas facilidades que ofrece y promesas que hace (sí, también interviene aquí la publicidad y las modas). Hay lenguajes que son muy rápidos en ejecución, hay lenguajes que están pensados para que los programas puedan ser verificados con facilidad, hay lenguajes para la web, para ambientes empresariales, para juegos, fáciles de escribir, fáciles de leer, entre otros muchos. No suele existir un lenguaje bueno en todo a la vez.

Quiero presentar a Ruby como un lenguaje extremadamente lento en ejecución, que soporta los paradigmas procedural, orientado a objetos y funcional, y donde es muy fácil y rápido expresar ideas complejas. Es un lenguaje bueno para aprender a programar (si nos limitamos al paradigma

procedural[54]) y es un lenguaje muy bueno para llevar a cabo pruebas con nuestras ideas complejas acerca de la complejidad de la vida artificial.

La migración desde cualquier lenguaje de programación orientado a objetos (C++, Java, Python, Perl...) es inmediata, aunque como todo en la vida, para adquirir mucha destreza se requiere tiempo. Pero comprender los conceptos no requiere esfuerzo. Y una vez que te hayas cambiado a este lenguaje, no querrás regresar.

La instalación de Ruby sobre Linux es como cualquier otro paquete:

sudo apt-get install ruby

Pero si vas a trabajar profesionalmente con este lenguaje, es recomendable hacer la instalación desde un manejador de versiones como rvm. De este modo puedes tener simultáneamente varios proyectos, cada uno con su propia versión del intérprete de Ruby y de sus gemas (las librerías), sin que interfieran unos con otros.

Para ejecutar un programa que hayas escrito en Ruby debes guardarlo en un archivo con extensión rb y luego dar el comando:

ruby archivo.rb

Similitudes entre Ruby y Python

Ruby tiene muchos aspectos sofisticados, pero cuando uno quiere aprender a programar (bajo el paradigma procedural) es casi idéntico a Python, solo que más sencillo.

- Las variables tienen tipado automático, como en Python.
- Hay un intérprete interactivo *irb* pero no lo recomiendo usar (por las mismas razones que en Python: al estudiante le confunde que las variables de pruebas anteriores siguen estando ahí).
- Las palabras clave para definiciones de funciones *(def)*, estructuras condicionales *(if-else-elsif)* estructuras de repetición *(for in, while)* son las mismas.

Diferencias entre Ruby y Python, a este nivel

- No hay que poner dos puntos en ningún sitio.
- La indentación es recomendable, por estética y estilo, pero no tiene consecuencias semánticas. Es incómodo y hasta peligroso en *Python*

54 Ruby es orientado a objetos (OO) puro, pero es muy fácil "engañar" al estudiante que se inicia en la programación y no mostrarle nada OO, sino fingir que sus construcciones son procedurales (funciones, *if-else for, while*). La sintaxis de Ruby es tan limpia que el estudiante no nota que está programando dentro de la clase *Object*.

que haya símbolos invisibles (el espacio y el tabulador) con consecuencias semánticas.

- Los bloques de código, como funciones, clases, bucles y condicionales terminan en end y pueden ir en una sola línea o en varias líneas. Si se desea poner todo en la misma línea se suele delimitar el bloque de código entre llaves { }

```
5.times do
    p "Hola"
end
```

O bien:

```
5.times { p "Hola" }
```

Comentarios

```
# Comentarios en una línea
=begin
 Comentarios
 que ocupan
 varias lineas
=end
```

Funciones

Declaración de funciones:

```
def mayor(a, b)
  if a > b
    return a
  else
    return b
  end
end
```

Uso de funciones:

```
p mayor(3, 5)
```

Las funciones pueden retornar varios valores separados por comas:

```
def ordenar(a, b)
  if a > b
    return b, a
  else
    return a, b
  end
end
```

Y los resultados de estas funciones se recogen así:

```
elMenor, elMayor = ordenar(8, 5)
p elMenor
p elMayor
```

Paso de argumentos a funciones

Como en muchos lenguajes modernos es:

- Argumentos de entrada: paso por valor cuando los argumentos son simples (números enteros y flotantes).
- Argumentos de entrada-salida: paso por referencia cuando los argumentos son complejos (strings, arrays, estructuras de datos...).

La explicación técnica es más complicada: siempre se pasan "referencias por valor", y los objetos enteros y flotantes son inmutables.

Ejemplos con argumentos simples:

```
def duplicar(numero)
   return numero*2
end

def triplicar(valor)
  valor = valor*3  # Se cambia el valor, pero eso no afecta a dato
  return  valor
end

dato = 5
resultado = duplicar(dato)
otroResultado = triplicar(dato)
p otroResultado    # vale 15
p resultado    # vale 10
p dato    # sigue valiendo 5
```

Ejemplos con argumentos complejos

```
def modificar(array)
      array[2] = "Hasta la vista"
end

saludos = ["Hola", "Hello", "Hi", "Olá"]
p saludos
modificar(saludos)
p saludos
```

Bucles

Las construcciones de control son las mismas, pero los rangos se definen con el operador dos puntos (valor final incluido) o tres puntos (valor final excluido). Ejemplos:

```
p "valor final incluido"
for indice in 0..10
   p indice
end

p "valor final excluido"
for indice in 0...10
   p  indice
end

contador=5
while contador< 10
   p contador
   contador += 1
end
```

No hay bucle do-while. En su lugar se hace lo siguiente:

```
loop
   # sentencias internas del bucle
   break if condicion
end
```

Verdadero y falso

- Las palabras clave son *true y false*.
- nil también vale false (nil es el valor que toma un elemento inexistente dentro de una estructura de datos).
- Y cualquier cosa distinta de *false es true*.

Ejemplos:

```
verdad = 5>2
p verdad

mentira = 4==3
p mentira

datos = []
```

```
datos[3] = 100
p datos

datos = [9, 25, 14, 6]
contador = 0
while item=datos[contador]   # Ojo: asignación
  p item
  contador += 1
end
```

No existe el operador incremento (contador++) o decremento (contador--) como en C, *C++* o *Java*.

Condicionales

Las palabras clave son: *if elsif else end* como en casi todos los lenguajes imperativos. Ejemplo:

```
ahorros=20_000_001     # El _ es solo una cosa estética
if ahorros > 20_000_000
   p "Puedo comprar un carro nuevo"
elsif ahorros > 10_000_000
   p "Puedo comprar un carro usado"
else
   p "Me toca ir en el Masivo"
end
```

Otra forma del condicional

Si la sentencia a ejecutar dentro de un *if* es solo una, se puede poner de esta otra manera más compacta y elegante: *sentencia if condición*

Por ejemplo:

```
if a < 0
  a = -a
end
```

Se puede poner de esta forma:

```
a = -a   if a < 0
```

Existe el *switch-case* de C, *C++* y *Java*, pero se llama *case-when*, y es mucho más flexible y sofisticado, puesto que retorna un valor:

```
color="amarillo"
codigo =
  case color
  when "azul"
```

```
    1
when "rojo"
    2
when "amarillo"
    3
else
    0
end
```

Operadores lógicos

Existen las operaciones lógicas igual que en otros lenguajes populares: &&, || *y !*

```
a,b=3,4
p "Error" if (a> 0 || b+a < 5) && a*b==12
p "Error" if !c
```

También existen las palabras clave *and or not*, pero tienen la más baja prelación (menor que la del operador asignación), por lo que siempre deben encerrarse entre paréntesis externos que agrupen todo lo que se va a evaluar con estos operadores:

```
p "Error" if ((a> 0 or b+a < 5) and a*b==12)
p "Error" if (not c)
```

Arrays

Los arrays son dinámicos y con sintaxis muy sencilla y natural (al contrario que en *Python*):

```
letras = ["A", "B", "C"]
letras = letras << "D"    # Añadir al final
p letras
p letras[2]
letras[3] = "Hola"
p letras
letras[10] = "Chao"
p letras
p letras[8]
p letras[333]
p letras[-1]
p letras[-9]
```

Índices de los arrays

Los *arrays* no disparan excepciones con facilidad. Acabamos de ver que podemos salirnos fuera del *array* y los items no existentes los toma como *nil*. Se admiten también índices negativos, que significan

- -1 el último item
- -2 el penúltimo item
- y así sucesivamente.

Arrays de dos dimensiones

Los *arrays* de 2 o más dimensiones son triviales de hacer sabiendo que un elemento de un *array* puede ser otro *array*:

```
tablaMultiplicar = []              # Array vacío
for fila in 0..10
   tablaMultiplicar[fila] = []     # Cada elemento es otro array
   for columna in 0..10
      tablaMultiplicar[fila][columna] = fila * columna
   end
end
p tablaMultiplicar
```

Biblioteca estándar

Ruby cuenta con una biblioteca estándar muy rica y con un *API* muy uniforme (sin sorpresas), y es mejor consultar un libro de referencia como (Thomas, 2005), para conocer cada detalle. Por ejemplo, para averiguar el tamaño de un *Array* (o de cualquier otro contenedor) se hace así:

```
a = [ [6, 5], [2, -5, 9, 8], [9] ]
p a.length
p a[0].length
p a[1].length
p a[2].length
```

Cómo imprimir en la pantalla

```
a = "Hola"

print a       # No salta de línea al finalizar
print(a)      # Es lo mismo que lo anterior

puts a        # Sí salta de línea al finalizar
puts(a)       # Es lo mismo que lo anterior
p a           # Es lo mismo que lo anterior
```

Interpolación
Se pueden insertar variables dentro de un *string* con el operador *#{}* que lo que hace es ejecutar la expresión encerrada dentro de las llaves.

```
a = 5.5
b = 7
c = "Si multiplico #{a} por #{b} me da #{a*b}"
p c
```

Lo habitual es interpolar al imprimir:

```
p "Si multiplico #{a} por #{b} me da #{a*b}"
```

Cómo leer el teclado
Siempre lee *strings* que luego se pueden convertir a números enteros o flotantes.

```
a = gets
p a
```

Cómo convertir un *string* a entero, a flotante o a *string*

```
p "Introduce un número"
a = gets
a = a.to_i
p a
p "Introduce otro número"
b = gets
b = b.to_f
p b
c = b.to_s
p c
```

Ruby es OO puro

Todo, absolutamente todo en Ruby, es un objeto. Eso significa que todo es manipulable. Por ejemplo, los números son objetos y tienen un *API*:

```
-1.abs
```

Las clases también son objetos y tienen un *API* que las permite ser interrogadas y modificadas en tiempo de ejecución.

Ruby es un lenguaje orientado a objetos a la vez que funcional. Las funciones se pueden pasar como argumentos a otras funciones. Existen las funciones anónimas y la función *lambda*.

Los paréntesis no son necesarios, salvo para forzar prelaciones:

```
a.multiplicar_por 5
```

Toda construcción retorna un valor, que es el último valor calculado. Ejemplo:

```
a = if 5 < 7 then 3 else 4 end  # a valdrá 3
```

Por ello el return se usa pocas veces. Ejemplo:

```
def duplicar(a)
  2*a
end
```

Iteradores

Ruby incluye iteradores en el lenguaje. Esta es la construcción que más confunde a las personas que ven *Ruby* por primera vez, pero es muy sencilla y potente. En *Ruby* no se suelen usar los bucles for para explorar arrays, sino iteradores:

Ejemplo sin iteradores:

```
a=[4, 2, -5, 3, 7]
for i in 0...a.length
   item = a[i]
   p item
end
```

Ejemplo con iteradores:

```
a.each do |item|
   p item
end
```

O más abreviado:

```
a.each { |item|  p item }
```

Hay iteradores que sirven para acumular:

```
a.inject(0) { |acumulado, item|  acumulado + item }
```

each es un iterador que existe en el *API* de cualquier contenedor, y que explora uno a uno cada item y me lo entrega en el argumento que yo le ponga encerrado entre llaves verticales. El iterador each recibe como argumento una función que yo le doy encerrada entre llaves {} (recordemos que *Ruby* es también un lenguaje funcional). En esa función, recibo el (o los) argumentos encerrados entre las barras verticales y hago con ellos lo que desee, en este caso, imprimirlos.

Hay muchos iteradores estándar *(each, collect, inject, delete_if...)* y yo puedo crear otros si lo necesito.

Los iteradores hacen la programación muy elegante:

- Eliminan índices absurdos *i,j,k* que no tienen nada que ver con el algoritmo si lo piensan bien, aunque nos hemos acostumbrado tanto a ellos que creemos que sí.
- Eliminan la necesidad de averiguar el valor inicial y el final de esos índices (con los consecuentes errores si me equivoco al hacerlo).
- El código resultante es mucho más corto y limpio.

Ruby incorpora automáticamente el polimorfismo, sin tener que hacer nada especial. No hace falta definir árboles de herencia, ni clases abstractas, ni *templates*, ni nada de nada. Eso es gracias a que las colecciones de objetos pueden contener objetos de cualquier clase.

Por ejemplo, suponiendo que las clases *Integer*, *Float*, *Array* y *Complex* tuvieran una función llamada *duplicar()* que multiplique por 2 el respectivo dato, entonces esto funciona sin problemas:

```
i = 5
f = 4.8
a = [3, 2, 10.1, 6.6]
c = Complex.new(2, -7)
[i, f, a, c].each { |item| item.duplicar }
```

Clases y objetos

Las clases se definen con la palabra *class* y terminan en *end*. Dentro están sus funciones, que se definen con *def* y terminan en *end*. El constructor siempre se llama *initialize*. Por convenio, el nombre de las clases comienza con letra mayúscula, mientras que el de los objetos comienza con letra minúscula.

```
class CuentaBancaria
   def initialize(valorInicial)
      @ahorros =  valorInicial
   end
   def ingresar(valor)
      @ahorros = @ahorros + valor
   end
   def retirar(valor)
      @ahorros = @ahorros - valor
   end
   def saldo
      @ahorros
   end
end
```

Y los objetos se crean con el operador new, que llama al constructor de la clase.

```
miCuenta = CuentaBancaria.new(200)
miCuenta.ingresar(500)
miCuenta.retirar(100)
p miCuenta.saldo     # imprimirá 600
```

Las variables que comiencen por @ son atributos de objeto y las que comienzan por @@ son atributos de clase. Se crean en el momento en que se usan por primera vez, es decir, no hay que declararlas previamente. Las variables que no llevan ningún símbolo delante son variables locales. Las variables que comienzan por $ son variables globales, aunque, como todo el mundo sabe, no conviene usarlas pues producen código no reentrante (que no funciona cuando hay varios hilos).

Se pueden crear automáticamente las funciones getter y setter para los atributos usando las palabras clave: *attr_reader, attr_writer, attr_accesor*

Por ejemplo, en vez de la función saldo del ejemplo anterior podríamos poner:

```
class CuentaBancaria
   attr_reader :ahorros
   def initialize(valorInicial)
      @ahorros =  valorInicial
   end
   def ingresar(valor)
      @ahorros = @ahorros + valor
   end
   def retirar(valor)
      @ahorros = @ahorros - valor
   end
end
```

Que se puede usar así:

```
miCuenta = CuentaBancaria.new(200)
miCuenta.ingresar(500)
miCuenta.retirar(100)
p miCuenta.ahorros        # Función getter. Imprimirá 600
```

Si se hubiera definido el atributo ahorros de tipo lectura-escritura:

```
   attr_accesor :ahorros
```

Entonces también se podría escribir directamente así:

```
miCuenta.ahorros = 900    # Función setter
```

Las clases son también objetos y tienen un *API* por medio del cual pueden ser interrogadas y modificadas, permitiendo la metaprogramación. Estos *getter* y *setter* son realmente ejemplos de metaprogramación.

Composición de objetos

- La herencia existe pero no es muy importante.
- Son más importantes los mixins, una especie de "herencia horizontal". Responde a lo que hace años se llamaba "programación orientada a aspectos" y permite incluir en una clase los aspectos más relevantes que ya existen en otras clases. Por ejemplo, todas las colecciones de objetos (*Arrays, Hashes, Sets...*) incluyen el *mixin Enumerable*, que a su vez arrastra muchos iteradores como *each* e *inject*.

Metaprogramación

Ruby permite la metaprogramación muy fácil y ordenadamente, es decir, escribir programas que escriban otros programas. O que se automodifiquen. Eso sirve para:

- Crear nuevos idiomas dentro del lenguaje (nuevas sentencias como *attr_reader, attr_writer, attr_accesor*).
- Evitar la escritura de código repetitivo.
- Conectarse y manipular automáticamente bases de datos, como se hace al heredar de *ActiveRecord*, y entonces todos los atributos de la clase se mapean en una tabla, los objetos son filas, los atributos son columnas, y se logra automáticamente la persistencia.
- Mapearse automáticamente a interfaces web (como se hace en *Ruby* on *Rails*).
- Usar patrones de forma sencillísima (*singleton, delegación, decorador, función default...*).

Para saber más

Con estos dos libros se puede comenzar a programar *Ruby*, y seguir programando y aprendiendo por varios años. Son los mejores para empezar:

- Dave Thomas, Chad Fowler y Andy Hunt (2005). *Programming Ruby. The Pragmatic Programmer's Guide (Second Edition)*. The Pragmatic Bookself.

La primera parte te enseña poco a poco a programar en *Ruby*. Si ya sabes otro lenguaje orientado a objetos, mucho mejor, pero en caso contrario también te sirve. La segunda parte explica los accesorios de *Ruby*, su entorno,

la línea de comandos, cómo conectarlo con otros lenguajes, etc. La tercera parte explica lo interno de *Ruby*, cuál es su filosofía y cómo se representan los objetos en memoria. Gracias a ello podrás entender la parte más difícil: la reflectividad, que permite a un programa interrogarse y modificarse a sí mismo. Ello lo convierte en un excelente libro incluso para programadores expertos. La cuarta parte es la librería estándar de *Ruby*, donde se explica cada clase y cada método, con varios ejemplos.

- Lucas Carlson y Leonard Richardson (2006). *Ruby Cookbook*. USA: O'Reilly.

Ya sabes algo de *Ruby*, pero probablemente todavía no dominas todo su potencial. Todavía no piensas de la forma "*the Ruby way*". Quieres hacer algo sencillo, pero te das cuenta de que estás empleando modelos mentales que arrastras de otros lenguajes como *Java*, *Python* o *C++*. ¿Será que hay una forma más simple de hacerlo en *Ruby*? Casi seguro que la respuesta es afirmativa, y en este libro te encontrarás multitud de ejemplos de trozos de código (unas pocas líneas) con las que realizar las tareas más habituales. Basta copiar, pegar y adaptarlo a tu problema específico. Cada trozo de código viene extensamente explicado y se dan ejemplos alternativos, para que no emplees solamente el *Copy&Paste* sino que sigas aprendiendo.

Referencias

Ruby tiene aplicaciones especializadas en muchas áreas, desde la programación web a la concurrencia, paralelismo, ambientes distribuidos y *testing*, por nombrar las más conocidas. Estos libros te ayudarán a seguir aprendiendo en cada tema.

Libros, artículos y enlaces web

Baird, K. C. (2007). *Ruby by example*. San Francisco: No Starch Press.

Berube, D. (2007). *Practical Ruby Gems*. Berkeley: Apress.

Black, D. A. (2006). *Ruby for Rails*. Greenwich: Manning Publications.

Brown, G. T. (2009). *Ruby best practices*. Sebastopol: O'Reilly.

Burd, B. (2007). *Ruby on Rails for dummies*. New Jersey: Wiley.

Dees, I. (2008). *Scripted GUI testing with Ruby*. The Pragmatic Bookself.

Feldt, R., Johnson, L. y Neumann, M. (2002). *Ruby developer's guide*. USA: Syngress Publishing.

Fitzgerald, M. (2007). *Learning Ruby*. Sebastopol: O'Reilly.

Fisher, T. (2008). *Ruby on Rails bible*. Indianapolis: Wiley.

Fulton, H. (2007). *The Ruby Way: Solutions and Techniques in Ruby Programming, Second Edition*. Addison Wesley Professional. Boston: Pearson Education.

Gray II, J. E. (2006). *Best of Ruby quiz*. USA: The Pragmatic Bookself.

Grimm, A. (2013). *Confident Ruby*. USA: The Pragmatic Bookself.

___. (2013). *Exceptional Ruby*. USA: The Pragmatic Bookself.

Gutschmidt, T. (2003). Game programming with Python, Luan and Ruby. Premier Press.

Hal, F. (2002). *The Ruby Way*. USA: SAMS Publishing.

Harris, J. (2006). *Rubyisms in Rails*. Addison Wesley Professional.

Hartl, M. y Prochazka, A. (2008). *Rails space. Building a social networking website with Ruby on Rails*. Boston: Pearson Education.

Hellsten, C. y Laine, J. (2006). *Beginning Ruby on Rails E-Commerce. From Novice to Professional*. Berkeley: APress.

Lenz, P. (2007). *Build Your Own Ruby On Rails Web Applications*. Australia: SitePoint Pty.

Mahadevan, S. (2002). *Making use of Ruby*. Indianapolis: Wiley.

Marick, B. (2006). *Everyday scripting with Ruby for teams, testers and you*. USA: The Pragmatic Bookself.

Matsumoto, Y. (2001). *Ruby in a nutshell*. Sebastopol: O'Reilly.

Perrota, P. (2014). *Metaprogramming Ruby. Program like the Ruby Pros*. The Pragmatic Bookself.

Rappin, N. (2008). *Professional Ruby on Rails*. Indianapolis: Wiley.

Schmidt, M. (2006). *Enterprise Integration with Ruby*. USA: The Pragmatic Bookself.

Seki, M. (2012). *The DRuby book. Distributed and Parallel Computing with Ruby*. USA: The Pragmatic Bookself.

Tate, B. A. (2006). *From Java to Ruby. Things every manager should know*. USA: The Pragmatic Bookself.

___. y Hibbs, C. (2006). *Ruby on Rails up and running*. Sebastopol: O'Reilly.

Tennis, C. (2006). *Rapid GUI development with QtRuby*. USA: The Pragmatic Bookself.

Vohra, D. (2007). *Ruby on Rails for PHP and Java developers*. Berlin. Springer.

Williams, J. (2007). *Rails solutions. Ruby on Rails made easy*. Berkeley: Apress.