Ruby Tutorial

...o como pasar un buen rato programando

Lección 1

Lección 2

- Rangos
- Arrays

Lección 3

- Bloques
- Más malabares con strings
- Expresiones Regulares
- Condicionales
- Bucles
- Números Aleatorios

Lección 4

- Clases y Objetos
- Accesores
- Ficheros: lectura/escritura
- Cargando librerías
- Herencia de clases
- Modificando clases
- Congelando objetos
- Serializando objetos

Lección 5

- Control de acceso
- Excepciones
- Módulos
- Constantes
- Hashes y Símbolos
- La clase Time

Lección 6

- self
- Duck Typing
- Azúcar Sintáctico
- Test de unidades

contacto

e-mail

Los métodos

En Ruby, todo lo que se manipula es un objeto, y el resultado de esas operaciones también son objetos. La única forma que tenemos de manipular los objetos, son los **métodos**:

```
5.times { puts "Ratón!\n" } #se hablará más tarde de bloques
"A los elefantes le gustan los cacahuetes".length
```

Si los objetos (como los strings, números,…) son los nombres, entonces los métodos son los verbos. Todo método necesita un objeto. Es fácil decir qué objeto recibe el método: el que está a la izquierda del punto. Algunas veces, puede que no sea obvio. Por ejemplo, cuando se usa `puts`y `gets`, ¿dónde están sus objetos? Nada más iniciarse el intérprete, estamos dentro de un objeto: el objeto **`main`**. Por tanto, al usar `puts` y `gets`, estamos mandando el mensaje al objeto `main`.

¿Cómo podemos saber dentro de qué objeto estamos? Usando la variable self.

```
puts self
```

Para más detalles sobre self, mirar aquí.

Escribiendo métodos

Un bloque de instrucciones que define un método, empieza por la palabra **`def`** y acaba por la **`end`**. Los

parámetros son la lista de variables que van entre paréntesis. Aunque en Ruby, dichos paréntesis son opcionales: `puts`, `p` y `gets` son muy usados, y por ello que el uso de paréntesis sea opcional. En Rails, se llama a los métodos sin paréntesis.

Un método devuelve el valor de su última línea. Por norma, es recomendable dejar una línea en blanco entre las definiciones de métodos:

```
#metodos.rb

# Definición de un método
def hello
  puts 'Hola'
 end
 #uso del método
 hello

 # Método con un argumento
def hello1(nombre)
  puts 'Hola ' + nombre
  return 'correcto'
end
puts(hello1('Pedro'))

# Método con un argumento (sin paréntesis, no funciona en versiones
nuevas)
def hello2 nombre2
  puts 'Hola ' + nombre2
  return 'correcto'
end
puts(hello2 'Juan')
```

Esto es lo que obtenemos

```
>ruby metodos.rb
Hola
Hola Pedro
correcto
Hola Juan
correcto
metodos.rb:18 warning: parenthesize argument(s) for future version
>Exit code: 0
```

Los métodos bang (!)

Los métodos que acaban con una ! son métodos que modifican al objeto. Por lo tanto, estos métodos son considerados como peligrosos, y existen métodos iguales, pero sin el !. Por su peligrosidad, el nombre "bang". Ejemplo:

```
a = "En una lugar de la mancha"

#método sin bang: el objeto no se modifica
b = a.upcase
puts b
puts a

#método con bang: el objeto se modifica
c = a.upcase!
puts c
puts a
```

Normalmente, por cada método con !, existe el mismo método sin !. Aquellos sin bang, nos dan el mismo resultado, pero sin modificar el objeto (en este caso el string). Las versiones con !, como se dijo, hacen la misma acción, pero en lugar de crear un nuevo objeto, transforman el objeto original.

Ejemplos de esto son: upcase/upcase!, chomp/chomp!,…En cada caso, si haces uso de la versión sin !, tienes un nuevo objeto. Si llamas el método con !, haces los cambios en el mismo objeto al que mandaste el mensaje.

Alias

```
alias nuevo_nombre nombre_original
```

`alias` crea un nuevo nombre que se refiere a un método existente. Cuando a un método se le pone un alias, el nuevo nombre se refiere al método original: si el método se cambia, el nuevo nombre seguirá invocando el original.

```
def viejo_metodo
   "viejo metodo"
end
alias nuevo_metodo viejo_metodo
def viejo_metodo
  "viejo metodo mejorado"
end
puts viejo_metodo
puts nuevo_metodo
```

En el resultado, vemos como `nuevo_metodo` hace referencia al `viejo_metodo` sin modficar:

```
viejo metodo mejorado
```

```
viejo metodo
```

NOTA: a día de hoy Ruby no se lleva muy bien con las tildes. Esto es por que no tiene soporte para strings Unicode. Se planea implementarlo en la versión 1.9 que saldrá en Diciembre del 2007.

Métodos perdidos

Cuando mandas un mensaje a un objeto, el objeto busca en su lista de métodos, y ejecuta el primer método con el mismo nombre del mensaje que encuentre. Si no encuetra dicho método, lanza una error **`NoMethodError`**.

Una forma de solucionar esto, es mediante el método **`method_missing`**: si definimos dicho método dentro de una clase, se ejecuta este método por defecto:

```
class Dummy
  def method_missing(m, *args)
    puts "No existe un metodo llamado #{m}"
  end
end

Dummy.new.cualquier_cosa
```

obtenemos:

```
No existe un metodo llamado cualquier_cosa
```

Por lo tanto, `method_missing` es como una red de seguridad: te da una forma de manejar aquellos métodos que de otra forma darían un error en tu programa

page_revision: 19, last_edited: 6 Aug 2010, 01:31 GMT-05 (116 days ago)

Other interesting sites

Therafim RPG

Epic Destiny Awaits

Ethisch Forum

The Collaboratory

at Colby Community College

BGC Digital Media Lab

A collaborative space for all things digital at the BGC