Ruby Tutorial

...o como pasar un buen rato programando

Lección 1

- Introducción
- Instalación
- El Primer Programa
- Números en Ruby
- Strings y diversión
- Variables
- Alcance de las variables

Lección 2

- Introduciendo Datos
- Normas en los nombres
- Los métodos
- Los métodos: argumentos
- 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

Expresiones Regulares

Las expresiones regulares, aunque críticas, son una poderosa herramienta para trabajar con texto. Son usadas para reconocer patrones y procesar texto. Una **expresión regular** es una forma de especificar un patrón de caracteres, que será buscado en un string. En Ruby, se crean las expresiones regulares entre //: son objetos del tipo `Regexp` y pueden ser manipuladas como tales.

```
//.class #  Regexp
```

La forma más simple de encontrar si una expresión (también funciona con strings) está dentro de un string, es usar el método **`match`** o el operador **`=~`**:

```
m1 = /Ruby/.match("El futuro es Ruby")
puts m1         # "Ruby", puesto que encontró la palabra
puts m1.class   # devuelve MatchData; devuelve "nil" si no se encuentra

# operador {{**=~**}}:
m2 = "El futuro es Ruby" =~ /Ruby/
puts m2 # 13 -> posición donde empieza la palabra "Ruby"
```

Construyendo expresiones regulares

Cualquier caracter que vaya entre barras, se busca exactamente:

```
/a/ # se busca la letra a, y cualquier palabra que la contenga
```

Algunos caracteres tienen un significado espcial en las expresiones regulares. Para evitar que se procesen, y poder buscarlos, se usa la **secuencia de escape** \.

```
/\?/
```

La \ significa "no trates el siguiente carácter como especial". Los caracteres espciales incluyen: ^, $, ? , ., /, \, [,], {, }, (,), + y *.

El comodín . (punto)

Algunas veces, se busca cualquier caracter en una posición determinada. Esto se logra gracias al **.** (punto). Un punto, busca cualquier carácter, excepto el de retorno de carro.

```
/.azado/
```

Busca 'mazado' y 'cazado'. También encuentra '%azado' y '8azado'. Por eso hay que tener cuidado al usar el punto: puede dar más resultados que los deseados. Sin embargo, se pueden poner restricciones a los resultados, especificando las clases de caracteres buscadas.

Clases de caracteres

Una clase de carácter es una lista explícita de caracteres. Para ello se usan los corchetes:

```
/[mc]azado/
```

De esta forma, especificamos la búsqueda de 'azado' precedido por 'c' o 'm': solamente buscamos 'mazado' o 'cazado'.

Dentro de los corchetes, se puede especificar un **rango de búsqueda**.

```
/[a-z]/ # encuentra cualquier minúscula
/[A-Fa-f0-9]/ # encuentra cualquier número hexadecimal
```

Algunas veces se necesita encontrar cualquier carácter menos aquellos de una lista específica. Este tipo de búsqueda se realiza negando, usando ^ al principio de la clase.

```
/[^A-Fa-f0-9]/ # encuentra cualquier carácter, menos los hexadecimales
```

Algunos caracteres son tan válidos, que tienen su abreviación.

Abreviaciones para clases de caracteres

Para encontrar cualquier número, estas dos expresiones son equivalentes:

```
/[0-9]/
/\d/
```

Otras dos abreviaciones son:

- **\w** encuentra cualquier dígito, letra, o guión bajo (_).
- **\s** encuentra cualquier carácter espacio-en-blanco (character whitespace), como son un espacio, un tabulado y un retorno de carro.

Todas las abreviaciones precedentes, también tienen una forma negada. Para ello, se pone la misma letra en mayúsculas:

```
/\D/ # busca cualquier carácter que no sea un número
/\W/ # busca cualquier carácter que no sea una letra o guión bajo
/\S/ # busca un carácter que no sea un espacio en blanco.
```

Tabla resumen

expresión	**significado**
.	cualquier caracter
[]	especificación por rango. P.ej: [a-z], una letra de la a, a la z
\w	letra o número; es lo mismo que [0-9A-Za-z]
\W	cualquier carácter que no sea letra o número
\s	carácter de espacio; es lo mismo que [\t\n\r\f]
\S	cualquier carácter que no sea de espacio
\d	número; lo mismo que [0-9]
\D	cualquier carácter que no sea un número
\b	retroceso (0x08), si está dentro de un rango
\b	límite de palabra, si NO está dentro de un rango
\B	no límite de palabra
*	cero o más repeticiones de lo que le precede
+	una o más repeticiones de lo que le precede
$	fin de la línea
{m,n}	como menos m, y como mucho n repeticioes de lo que le precede
?	al menos una repetición de lo que le precede; lo mismo que {0,1}
()	agrupar expresiones
||	operador lógico O, busca lo de antes o lo después

Si no se entiende alguna de las expresiones anteriores, lo que hay que hacer es probar. Por ejemplo, veamos el último caso: la |. Supongamos que buscamos la palabra 'gato' o la palabra 'perro':

```
/gato|perro/
```

El | es un "O lógico": se busca la palabra de la izquierda o la palabra de la derecha.

Una búsqueda con éxito, devuelve un objeto MatchData

Cualquier búsqueda sucede con éxito o fracasa. Empecemos con el caso más simple: el fallo. Cuando intentas encontrar un string mediante un patrón, y el string no se encuentra, el resultado siempre es **`nil`** (nil = nada).

```
/a/.match("b") # nil
```

Sin embargo, si la búsqueda tiene éxito se devuelve un objeto **`MatchData`**. Este objeto tiene un valor 'true' desde el punto de vista booleano, y además almacena la información de lo encontrado: donde empieza (en qué carácter del string), qué porción del string ocupa,…Para poder usar esta información, hace falta almacenarla primero.

Veamos un ejemplo donde buscamos un número de teléfono dentro de un string:

```
string = "Mi número de teléfono es (123) 555-1234."
num_expr = /\((\d{3})\)\s+(\d{3})-(\d{4})/ # expresión regular
m = num_expr.match(string)                 # almacenamos búsqueda
unless m
  puts "No hubo concordancias."
  exit
end
print "El string de la búsqueda es: "
puts m.string    # string donde se efectúa la búsqueda
print "La parte del string que concuerda con la búsqueda es: "
puts m[0]        # parte del string que concuerda con nuestra búsqueda
puts "Las tres capturas:"
3.times do |index|
  # m.captures[index] - subcadenas encontradas (subcaden = () en la expresión)
  puts "Captura ##{index + 1}: #{m.captures[index]}"
end
puts "Otra forma para poner la primera captura: "
print "Captura #1: "
puts m[1] # cada número corresponde a una captura
```

la salida es:

```
El string de la búsqueda es: Mi número de teléfono es (123) 555-1234.
La parte del string que concuerda con la búsqueda es: (123) 555-1234
Las tres capturas:
Captura #1: 123
Captura #2: 555
Captura #3: 1234
Otra forma de poner la primera captura
Captura #1: 123
```

Para analizar la expresión regular, hay que prestar atención a cómo están agrupadas las búsquedas entre paréntesis:

```
num_expr = /\((\d{3})\)\s+(\d{3})-(\d{4})/
```

- **\((\d{3})\)** busca un grupo de tres números (\d{3}), entre dos paréntesis \(…\)
- **\s+** espacio en blanco una o varias veces
- **(\d{3})** tres números
- **-** signo menos
- **(\d{4})** cuatro números

Other interesting sites

Margopedia

Po••czenie wikipedii i nonsensopedii...

¥ The Unforgotten ¥

Frontman Wiki

Hulpbehoevende Ouderen