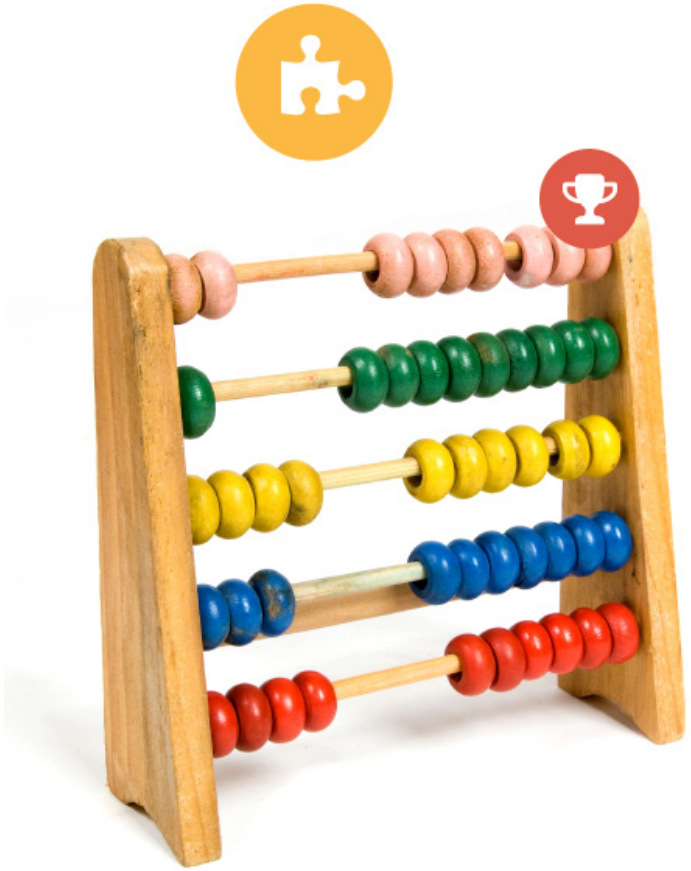
Programación en Lenguaje Ruby

Arrays

Delio Tolivia

Gijón crece 2016 / 2019
AYTO. DE GIJÓN / CCOO / FADE / UGT
Ayuntamiento de Gijón
Gijón impulsa Empleo
dicampus TALENTO FORMACIÓN



Índice

Creación de arrays(I)

- Un array es una lista de datos. Podemos crear arrays vacíos con ***[]*** o con ***Array.new***. Esta ultima opción es más flexible y admite distintos tipos de argumentos.

```
Array.new # => []
Array.new(2) # => [nil,nil]
Array.new(5,"A") # =>["A","A","A","A","A"]
```

- Para incluir valores en el array al crearlo los colocamos dentro de los ***[]*** separados por comas.

```
[1, 2, 3]
```

Creación de arrays (II)

- Los elementos de un array pueden ser de diferentes tipos. En un mismo array podemos tener numeros y cadenas de texto.

[1, 'one', 2, 'two']

- Podemos definir un array de cadenas con la sintaxis %w[……..] de tal manera que el contenido incluido nos será separado por palabras en un array

```
def array_of_words_literal
  %w[With this double-u shorthand it wasn't very hard at all to type out this list
  of words. Heck, I was even able to use double-quotes like "these"!]
end
```

```
["With", "this", "double-u", "shorthand", "it", "wasn't", "very", "hard", "at",
 "all", "to", "type", "out", "this", "list", "of", "words.", "Heck,", "I", "was",
 "even", "able", "to", "use", "double-quotes", "like", "\"these\"!"]
```

Acceso a elementos

- Para acceder a un elemento podemos usar el operador ***[]*** indicando el índice de la posición del elemento (recordar que se comienza contado desde 0):

```
2.0.0-p648 > [1, 2, 3, 4, 5][2]
 => 3
```

- Podemos acceder a los elementos contando también desde el final. Esto se consigue utilizando como índice un número negativo. En este caso el primer elemento es el -1

```
2.0.0-p648 :096 > [1, 2, 3, 4, 5][-1]
 => 5
```

Ejercicio

- Hacer el ejercicio 5 del pdf de ejercicios
- Hacer el ejercicio 11 del pdf de ejercicios

Añadir elementos

- Podemos añadir elementos a un array con el operador **<<**

```
2.0.0-p648 > [1, 2, 3, 4, 5]<<"woot"
 => [1, 2, 3, 4, 5, "woot"]
```

- Otra forma es con el método ***push***

```
2.0.0-p648 > [1, 2, 3, 4, 5].push "woot"
 => [1, 2, 3, 4, 5, "woot"]
```

Transformando arrays

- Podemos modificar todos los elementos de un array utilizando el método ***map*** (o ***collect***) pasando un bloque de código a realizar sobre cada uno de los elementos:

```
2.0.0-p648 > [1, 2, 3, 4, 5].map { |i| i + 1 }
 => [2, 3, 4, 5, 6]

2.0.0-p648 > [1, 2, 3, 4, 5].collect { |i| i + 1 }
 => [2, 3, 4, 5, 6]
```

Ejercicio

- Hacer el ejercicio 2 del pdf de ejercicios

Filtrando arrays

- Otra cosa que nos permite Ruby realizar con los arrays es filtrarlos según una expresión booleana con el método ***select***

2.0.0-p648 > [1,2,3,4,5,6].select {\|number\| number % 2 == 0}
=> [2, 4, 6]

Borrando elementos

- Para eliminar elementos de un array tenemos el método ***delete***

```
2.0.0-p648> a=[1,3,5,4,6,7]
 => [1, 3, 5, 4, 6, 7]
2.0.0-p648> a.delete 5
 => 5
2.0.0-p648> a
 => [1, 3, 4, 6, 7]
```

- Podemos también realizar eliminaciones basadas en una expresión booleana con ***delete_if***

```
2.0.0-p648 > [1,2,3,4,5,6,7].delete_if{|i| i < 4 }
 => [4, 5, 6, 7]
```

Iterando (recorriendo) arrays

- Una primera forma de iterar sobre un array es mediante el uso de los bucles ***for*** aunque no es la forma más utilizada

```
array = [1, 2, 3, 4, 5]
for i in array
    puts I
end
```

- Una forma más común es utilizar el método ***each***

```
array = [1, 2, 3, 4, 5]
array.each do |i|
    puts i
end
```

Destructuring (I)

- Mediante esta acción podemos asignar a varias variables el contenido de un array de una forma directa

zen, life = [42, 43]

- Que es lo mismo que hacer

array = [42, 43]

zen = array[0]
life = array.at(1)

- Si intentamos asignar más variables que elementos del array nos las rellenará con ***nil***

Destructuring (II)

- Esto se puede utilizar también dentro de los bloques (los veremos con más detalle en un tema posterior)

```
[[1, 2, 3, 4], [42, 43]].each { |a, b| puts "#{a} #{b}" }

# 1 2
# 42 43
```

Operador splat * (I)

- Con el operador * podemos hacer el destructuring de una forma más adecuada. Por ejemplo

car, *cdr = [42, 43, 44]

car: 42
cdr: [43, 44]

****initial, second_last, last = [42, 43, 44]***

initial: [42]
second_last: 43
last: 44

Operador splat *(II)

- Se puede utilizar para pasar un número variable de argumentos a una función (solamente se puede hacer en el último parámetro)

```
def zen(*args)
    [args.first, args.last]
end

p zen(42, 43, 44, 45, 46)

# [42, 46]
```

Operador splat *(III)

- También sirve para crear arrays a partir de Strings o de rangos

```
zen = *(1..5)
str = *"Zen”

# [1,2,3,4,5]
#[“zen”]
```

- Otro uso es romper un array en argumentos para un método

```
def zen(a, b)
      a + b
end

puts zen(*[41, 1])
```

Operador splat *(IV)

- Para crear Hash

```
ary = [[4, 8], [15, 16], [23, 42]]

bry = ary.flatten #[4,8,15,16,23,42]

puts Hash[*bry] # Igual que hacer Hash[ary]

# {4=>8, 15=>16, 23=>42}
```

API de Arrays (I)

- Métodos para contar número de elementos

```
puts [4, 8, 15, 16, 23, 42].count
puts [4, 8, 15, 16, 23, 42].size
puts [4, 8, 15, 16, 23, 42].length
```

- ***count*** además acepta un parámetro y nos devolverá el número de elementos iguales a ese parámetro en el array

```
puts [42, 8, 15, 16, 23, 42].count(42) # 2
puts ["Jacob", "Alexandra", "Mikhail", "Karl", "Dogen", "Jacob"].count("Jacob") # 2
```

- También puede recibir un bloque y devuelve los elementos para los que ese bloque retorne true

```
[4, 8, 15, 16, 23, 42].count { |e| e % 2 == 0 }  # 4
```

API de Arrays (II)

- El método ***index*** devuelve el índice en el array del objeto especificado como parámetro

puts [4, 8, 15, 16, 23, 42].index(15) # 2

- Así mismo puede recibir un bloque y devuelve el índice del primer elemento para el que el bloque devuelva true

puts [4, 8, 15, 16, 23, 42].index { |e| e % 2 == 0 } # 0

- El método ***flatten*** devuelve una versión unidimensional del array que le pasemos

p [4, [8], [15], [16, [23, 42]]].flatten # [4, 8, 15, 16, 23, 42]

- Podemos indicar hasta cuantos niveles entra ***flatten*** a transformarlo

[4, [8], [15], [16, [23, 42]]].flatten(1) # [4, 8, 15 ,16, [23, 42]]

API de Arrays (III)

- El método ***compact*** devuelve un array con los elementos nil quitados

```
p [nil, 4, nil, 8, 15, 16, nil, 23, 42, nil].compact # [4, 8, 15, 16, 23, 42]
```

- El método ***zip*** recibe un numero variable de argumentos y devuelve un array que contiene los elementos de cada array unidos de forma correlativa

```
p [4, 8, 15, 16, 23, 42].zip([42, 23, 16, 15, 8])

# [[4, 42], [8, 23], [15, 16], [16, 15], [23, 8], [42, nil]]
```

- El método ***slice*** es como el operador ***[ ]***. Admite también rangos

```
p [4, 8, 15, 16, 23, 42].slice(2) # 15
p [4, 8, 15, 16, 23, 42].slice(2..5) #15, 16, 23, 42
```

API de Arrays (IV)

- El método ***join*** une todos los elementos de un array en un string. Podemos pasarle un separador que utilizará entre todos los elementos salvo en el último

[4, 8, 15, 16, 23, 42].join(", ") # "4, 8, 15, 16, 23, 42"

- ***shift*** nos devuelve el primer elemento de un array (o si le pasamos un argumento el numero de elementos del array desde la izquierda) y modifica el array original quitándolos. El método contrario en ***unshift***

```
a = [4, 8, 15, 16, 23, 42]
p a.shift # 4
p a # [8, 15, 16, 23, 42]
p a.shift(2) # [8, 15]
p a # [16, 23, 42]
a.unshift (4, 8, 15)
p a # [4, 8, 15, 16, 23, 42]
```

Ejercicio

- Hacer ejercicio 14 del pdf de ejercicios