



Arreglos y archivos

Lo que debes saber antes de comenzar esta unidad

Ciclos

Los ciclos son instrucciones que nos permiten repetir la ejecución de una o más instrucciones.

Existen diversas formas de implementar estos ciclos en ruby, particularmente revisamos:

- `while` y `until`
- `for`
- `times`

Los problemas de iteración suelen utilizar contadores y acumuladores en las soluciones.

- Los contadores son una variable que va sumando de uno en uno, muy útiles cuando una operación tiene que ocurrir n veces.
- Los acumuladores guardan el valor de la iteración anterior y lo juntan con el de la iteración actual, ejemplos de estos son las sumatorias y productorias.

Ciclos y Condiciones de borde

Las condiciones de borde son muy importante en los ciclos, debemos analizar el número de iteraciones para evitar contar (o acumular) de más o de menos.

Por ejemplo para que el siguiente código se repita 5 veces debemos tener cuidado de ocupar $>$ y no mayor o igual.

```
i = 5
while(i > 0)
  puts i
  i -= 1
end
```

Bloques

Los bloques nos permiten pasar un grupo de instrucciones para ejecutarlas dentro de un método. Un bloque es todo lo que está entre `do` y `end` o entre llaves `{}`.

Bloques y ciclos

Algunos métodos que realizan ciclos utilizan bloques, como por ejemplo `times`.

Creando bloques

Existen dos formas de declarar un bloque:

con `do` y `end`

```
10.times do |i|
  puts i
end
```

Y de forma inline

```
10.times { |i| puts i }
```

Métodos

Los métodos nos permiten:

- Reutilizar código.
- Ordenarlo.
- Evitar repeticiones innecesarias (DRY).
- Abstraernos del problema.

Para crear un método ocuparemos la instrucción `def`

```
def metodo
end
```

Los métodos pueden recibir parámetros obligatorios u opcionales.

```
def metodo(x, y z = 5)
  puts z # 5
end
```

Los métodos retornan la evaluación de la última línea.

```
def metodo(x, y z = 5)
  puts z # puts z devuelve nil
end
resultado = metodo(2,3,1) # Se muestra 5 en pantalla, pero resultado será nil
puts resultado # nil
```

El alcance o **scope** es la asociación entre el nombre de una variable y una zona del programa. Hemos estudiado dos tipos de alcances.

- locales (solo se pueden acceder desde el método).
- globales (se pueden acceder desde cualquier parte del programa).

El alcance de las variables locales tiene límite entre el `def` y `end` :

```
def foo
  bar = 5
end

foo
puts bar # error :)
```

Las variables globales son peligrosas, especialmente cuando trabajamos con más personas o código de otros, puesto que es muy probable que sobreescribamos variables.

Capítulo: Introducción a Arrays (o arreglos)

Objetivos

- Conocer el uso de arrays.
- Crear arrays.
- Mostrar un array.
- Acceder a un elemento de un array ocupando su índice.

Introducción a Arrays

Los arrays, también conocidos como arreglos en español, son contenedores que permiten agregar múltiples datos. En lugar de guardar un sólo número o un único string, nos permitirán guardar varios en una variable.

```
a = ['dato 1', 'dato 2', 'dato 3']
```

¿Para qué sirven los arrays?

Los arrays son muy utilizados dentro de la programación. Nos permiten resolver diversos tipos de problema.

Posibles usos:

- En una aplicación web podemos traer los datos de la base de datos en un array y luego mostrar los datos en la página.
- Al traer los datos desde una API podría venir una colección y podríamos guardarlo dentro de un arreglo.
- Traer información guardada en uno o más archivos.
- Crear gráficos.

Creando arrays

Para crear un array utilizaremos la siguiente sintaxis:

```
a = [1, 2, 3, 4]
```

El array comienza utilizando corchetes `[]`, los datos en su interior deben estar separados por coma.

Mostrando un array

Lo más sencillo que podemos hacer con un array es mostrar sus elementos. Para esto podemos ocupar `puts` o `print`.

Con puts:

```
puts [1, 2, 'hola', 4]
# 1
# 2
# hola
# 4
```

Con print:

```
print [1, 2, 'hola', 4] # [1, 2, 'hola', 4]
```

Si queremos ver el arreglo en el mismo formato que lo ingresamos, entonces será mejor ocupar `print`.

Índices

Los elementos en el array tienen una posición, a la que se le llama índice.

El índice nos permite acceder al elemento que está dentro del arreglo.

```
a = [1, 2, 'hola', 'a', 'todos']
a[0] # 1
```

Nótese que los índices empiezan desde cero, así que para extraer `'todos'` se debe usar `a[4]`

Recorrido

Los índices van de cero hasta `n - 1`, donde n es la cantidad de elementos del arreglo.

```
a = [1, 2, 'hola', 'a', 'todos']
a[0] # 1
a[1] # 2
a[4] # todos
```

Índices mas allá de los límites

En caso de que el índice sea mayor o igual a la cantidad de elementos obtendremos nil, sin ninguna otra consecuencia (En otros lenguajes esto resultaría en un error).

```
a = [1, 2, 'hola', 'a', 'todos']  
a[8] # => nil
```

Índices negativos

Los índices también se pueden utilizar con números negativos y de esta forma referirse a los elementos desde el último al primero.

```
a = [1, 2, 3, 4, 5]  
a[-1] # => 5
```

Juguemos un juego

Dado el array:.

```
a = [1, 2, 3, 4, 'hola', 8]
```

¿Qué valor se muestra en cada uno de los siguientes índices?

- a[0]
- a[7]
- a[a[0]]
- a[4]
- a[-3]

```
a = [1, 2, 3, 4, 'hola', 8]  
a[0] # => 1  
a[7] # => nil  
a[a[0]] # => a[1] => 2  
a[4] # => 'hola'  
a[-3] # => 4
```

El Array ARGV

En las unidades anteriores utilizamos ARGV, ahora descubriremos que ARGV es un array

```
ARGV.class # => Array
```

Es por esto mismo que accedíamos a los elementos de ARGV a través del índice como ARGV[0] y ARGV[1]

Resumen

- Los arrays nos permiten guardar varios datos en una sola variable.
- Podemos mostrar los datos dentro de un array con print y puts, print los muestra de la misma forma que los ingresamos.
- Los elementos dentro de un array tienen un índice que indica su posición.
- Podemos acceder al elemento ocupando el índice, ej: a[0] => primer elemento.
- Los índices negativos nos permiten acceder al array desde atrás, ej: a[-1] => Último elemento.
- Utilizar un índice más allá del largo de un array nos da como resultado nil.

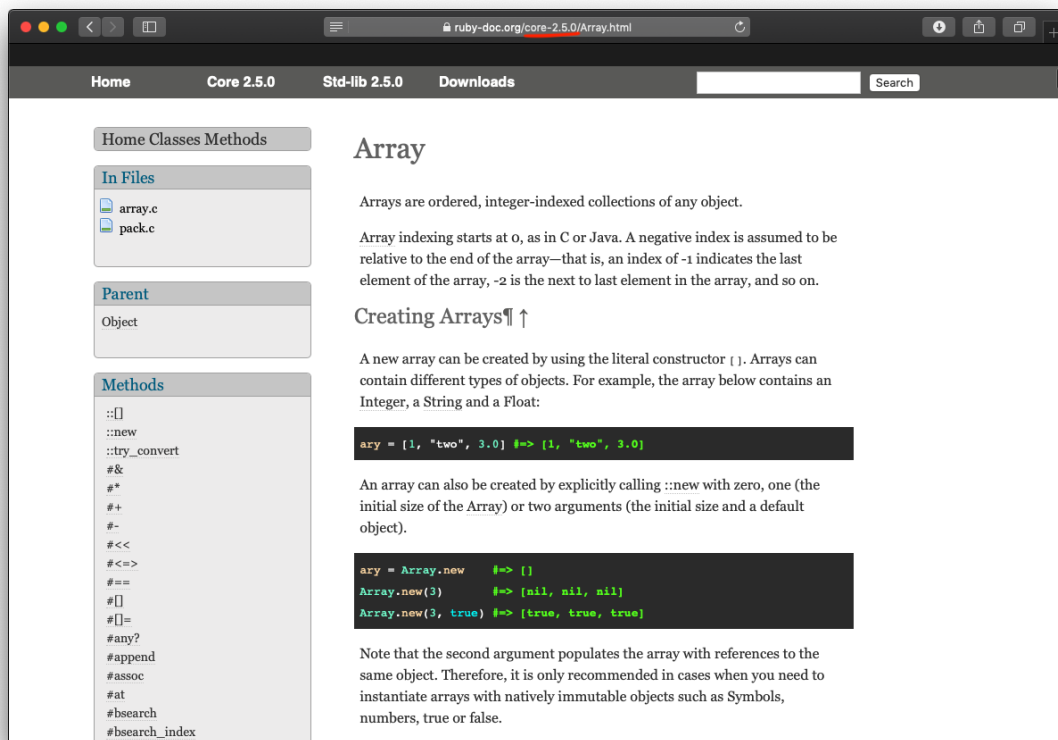
Ej: Si el arreglo a tiene 4 elementos: `a[5] => nil`

Capítulo: Operaciones básicas en un arreglo

Objetivos

- Leer documentación de la clase Array.
- Saber si un elemento se encuentra dentro de un arreglo.
- Agregar elementos a un arreglo.
- Borrar elementos a un arreglo.
- Contar elementos de un arreglo.

Motivación



Los arreglos tienen muchos métodos que nos permiten realizar operaciones básicas, como por ejemplo saber si un elemento está dentro de un arreglo, agregar elementos, borrar elementos y contar elementos. Por lo mismo, es muy importante saber leer su documentación.

Leyendo la documentación de la clase Array

Para la documentación ocuparemos ruby-doc.org.

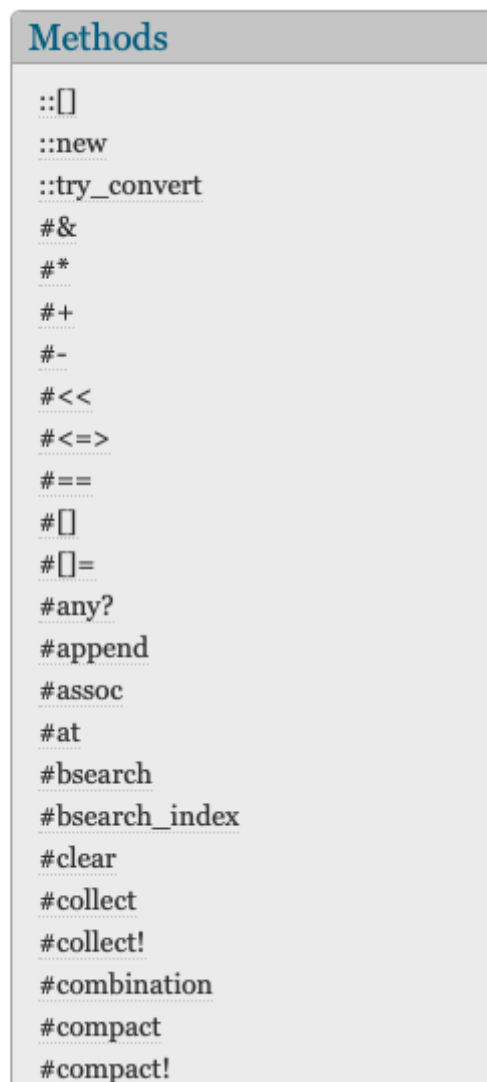
Si llegamos a la documentación a través del link podemos escoger la versión de Ruby, pero si llegamos a la página vía google es posible que entremos a la documentación de otra versión.

Como hay documentaciones para cada versión de Ruby, debemos preocuparnos de ver la documentación de la misma versión de Ruby que estemos ocupando.

Podemos cambiar la versión directamente desde la URL.

Leyendo la documentación.

En el panel izquierdo veremos los métodos, incluyendo todos los que estudiaremos en este capítulo, e incluso aparecen muchos más.



Un poco más abajo, en la sección de módulos incluidos, aparecen los enumerables, que estudiaremos en la segunda parte de esta unidad.



Membresía

Membresía es saber si un elemento es miembro o no de una entidad, o dicho dentro de este contexto, saber si un elemento pertenece a un arreglo.

Podemos saber si un arreglo incluye un elemento específico ocupando el método `.include?`

La documentación de `.include?`

`include?(object) → true or false`

Returns `true` if the given `object` is present in `self` (that is, if any `element == object`), otherwise returns `false`.

```
a = [ "a", "b", "c" ]  
a.include?("b")    #=> true  
a.include?("z")    #=> false
```

El lenguaje utilizado en la documentación puede ser un poco técnico, cuando dice `Returns true if the given object is present in self` lo que está diciendo es simplemente que devuelve `true` si el objeto está presente en ese objeto.

self

`self` se traduce al español como 'yo' o 'sí mismo'. Lo encontraremos frecuentemente cuando trabajemos con objetos.

`self` es el mismo objeto. O sea, si estamos trabajando con un arreglo en específico y el método dice `self`, entonces se refiere al arreglo con el que estamos trabajando.

Ejemplos de uso del método `.include?`

```
a = [1, 2, 3, 4, 5]  
a.include? 4 # => true  
a.include? 6 # => false
```

Ejemplo de membresía

Dado un arreglo llamado `ingredientes_pizza` se nos pide crear un programa donde el usuario ingrese un ingrediente y se muestre en pantalla si el ingrediente existe o no.

```
ingredientes_pizza = ['piña', 'jamón', 'salsa', 'queso']
```

Solución

```
ingrediente = ARGV[0]
if ingredientes_pizza.include? ingrediente
  puts "Tiene el ingrediente"
else
  puts "No lo tiene"
end
```

Agregar elementos

Podemos agregar un dato al final de un arreglo ocupando el método `.push`

```
a = [0, 1, 3, 4]
a.push(5) # => [0, 1, 3, 4, 5]
```

Ejercicio de agregar elementos

Dado un arreglo llamado `ingredientes_pizza` se nos pide crear un programa donde el usuario pueda consultar si un ingrediente existe, y si no existe debe ser añadido a la lista de ingredientes.

```
ingredientes_pizza = ['piña', 'jamón', 'salsa', 'queso']
```

Solución

```
ingrediente = ARGV[0]
if ingredientes_pizza.include? ingrediente
  puts "El ingrediente ya se encuentra dentro de la pizza"
else
  ingredientes_pizza.push(ingrediente)
  puts "El ingrediente #{ingrediente} fue agregado"
end
```

Otra forma de agregar elementos

Existe otra forma de agregar elementos que no es recomendada, ya que tiene una precedencia más alta, y por lo mismo puede darnos resultados que no deseamos.

```
a << 5
# => [0,1,3,4,5]
```

Utilizaremos siempre `.push`, pero ahora, si ves el operador en algún código, ya sabes lo que hace :).


Remover elementos


Hay varias formas de remover elementos de un arreglo. Una de la mas sencillas es con el método

`.delete`

```
a = [1, 2, 3, 4, 5, 6, 7]
a.delete(2)
print a # [1, 3, 4, 5, 6, 7] => nil
```

Revisemos la documentación del método `.delete`

 **delete(obj) → item or nil**

 **delete(obj) { block } → item or result of block**

Deletes all items from `self` that are equal to `obj`.

Returns the last deleted item, or `nil` if no matching item is found.

If the optional code block is given, the result of the block is returned if the item is not found. (To remove `nil` elements and get an informative return value, use `#compact!`)

```
a = [ "a", "b", "b", "b", "c" ]
a.delete("b")           #=> "b"
a                       #=> [ "a", "c" ]
a.delete("z")           #=> nil
a.delete("z") { "not found" } #=> "not found"
```

La documentación nos dice dos cosas adicionales del método `.delete`

- El método `delete` retorna el valor borrado
- Podemos realizar acciones en caso de no encontrar un valor.

El retorno del método delete

`.delete` devuelve el objeto borrado.

```
a = [1, 2, 3, 4, 5, 6, 7]
b = a.delete(2)
puts "Se ha borrado el elemento #{b}"
```

Probando el bloque

```
a = [1, 2, 3, 4, 5, 6, 7]
b = a.delete('hola') {a << 'error'; "Ups"}
puts "Se ha borrado a el elemento #{b}"
print a
```

```
Se ha borrado a el elemento Ups
[1, 2, 3, 4, 5, 6, 7, "error"]
```

```
b = a.delete('hola') {a << 'error'; "Ups"}
```

Recordando el concepto de bloque

Un bloque es todo lo que está entre `do` y `end`. Es una forma de agrupar código. Los bloques nos permiten enviar instrucciones a otros método.

El método `.delete` es un ejemplo de método que puede recibir un bloque.

Hay dos formas equivalentes de escribir un bloque

```
a = [1, 2, 6, 1, 7, 2, 3]

# Con do y end
a.delete(2) do
  'ups'
end

# con {}
a.delete(2) { 'ups' }
```

Múltiples instrucciones en un bloque inline

En los bloques delimitados por llaves se pueden poner varias líneas de código, y se puede usar más de una instrucción en la misma línea separando con `;`

```
# con {}
a.delete { a << 2; 'ups' }
```

En general esto último es poco utilizado.

Contar elementos

Podemos contar la cantidad de elementos de un array con los métodos `.count`, `.length` y `.size`

```
a = [1, 2, 3, 4, 5, 6, 7]
a.length # 7
a.count # 7
a.size # 7
```

El más utilizado para contar es `.count`

La documentación del método `.count`

 **count** → **int**

 **count(obj)** → **int**

 **count { |item| block }** → **int**

Returns the number of elements.

If an argument is given, counts the number of elements which equal `obj` using `==`.

If a block is given, counts the number of elements for which the block returns a true value.

```
ary = [1, 2, 4, 2]
ary.count           #=> 4
ary.count(2)        #=> 2
ary.count { |x| x%2 == 0 } #=> 3
```

Como dice la documentación, también podemos contar las apariciones de un elemento específico

```
a = [1, 2, 3, 4, 5, 6, 7]
a.count 2 #=> 1
a.count { |x| x.even? } #=> 3
```

Capítulo: Iterando un arreglo a partir del índice

Objetivos:

- Recorrer los elementos de un arreglo.
- Transformar los elementos dentro del array mientras lo recorremos.
- Filtrar elementos de un arreglo en base a algún criterio.

Motivación

En este capítulo estudiaremos cómo recorrer un arreglo y operar sobre sus datos para resolver diversos tipos de problema.

Este tipo de operaciones son parte del trabajo diario en el mundo de la programación.

¿Qué tipos de problema podemos resolver recorriendo un array?

Recorrer los elementos de un array nos permite operar sobre sus valores, por ejemplo, para filtrar o transformar valores. Este tipo de problema es muy frecuente para un programador.

Por ejemplo: Dado un arreglo de precios, podemos encontrar el valor promedio, incrementar todos los valores o aplicar un descuento.

Formas de iterar

En Ruby existen diversas formas de iterar un arreglo, cada una tiene ventajas y desventajas.

Las formas de iterar un arreglo las clasificaremos en:

- Utilizando un índice.
- Elemento a Elemento.
- Elemento con índice.

En este capítulo nos enfocaremos en recorrer el array utilizando un índice, y las otras formas las estudiaremos más adelante.

Iterando un arreglo con el índice

Sabemos acceder a un elemento en una posición dada utilizando `a[posición]`. Por lo mismo podemos ocupar lo que hemos aprendido de ciclos para iterar el arreglo, o sea, podemos utilizar `while`, `for` o incluso `times`.

Veamos un ejemplo con `times`:

```
a = [1, 2, 3, 4]
n = a.count
n.times do |i|
  puts a[i]
end
```

Cuidado con las condiciones de borde

En el código hicimos lo siguiente:

```
n = a.count
```

Haber escrito el número de iteraciones manualmente habría sido un error potencial muy grande. Puesto que, en caso de que cambiemos el array, el código dejará de funcionar como lo esperamos.

Hardcoding

Ingresar los números manualmente es una práctica que se conoce como `Hardcode`. Es una muy mala práctica en la industria y distingue fácilmente a un novato de una persona más experimentada.

```
a = [1, 2, 3, 4]
4.times do |i|
  puts a[i]
end
```

Veamos el mismo ejemplo utilizando `while`

```
a = [5, 6, 7, 8, 9]
n = a.count
i = 0
while i < n
  puts a[i]
  i += 1 # Tenemos que recordar incrementar el índice en cada iteración
end
```

¿Qué tipo de problemas podemos resolver?

Ya sabemos suficiente para comenzar a resolver diversos tipos de problema. Aplicando lo aprendido podemos:

- Tomar todos los elementos y transformarlos, por ejemplo, de entero a string.
- Filtrar, o sea seleccionar o mostrar solo los elementos que cumplen algún criterio
- Reducir: Por ejemplo, sumar todos los elementos, concatenarlos o multiplicarlos.

Ejemplo de transformación

Sabemos que el usuario puede introducir múltiples datos via `ARGV`. ¿Que pasaría si quisiéramos transformarlos todos a enteros? Para esto tenemos que recorrer el arreglo `ARGV` elemento a elemento, transformándolos y guardando los cambios.

```
n = ARGV.count
n.times do |i|
  ARGV[i] = ARGV[i].to_i
end
```

Transformar y guardar los valores originales

En algunas situaciones podría ser mejor no tocar el arreglo original y guardar los datos en un arreglo nuevo, para eso solo tenemos que crear un arreglo vacío y guardar los datos ahí.

```
n = ARGV.count
array = []
n.times do |i|
  array.push ARGV[i].to_i
end
print array
```

Filtrar elementos de un arreglo

Filtrar un arreglo consiste en seleccionar los elementos que queremos en base a una condición, por ejemplo podríamos estar creando un programa que maneja datos astronómicos pero en algunos casos las mediciones entregadas están malas y necesitamos descartarlas.

Ejemplo de filtrado

Se pide crear un programa que filtre todos los números menores a 1000 de un arreglo, que es lo mismo que seleccionar todos los elementos mayores o iguales a mil.

```
a = [100, 200, 1000, 5000, 10000, 10, 5000]
```

Filtrando en un arreglo nuevo

```
a = [100, 200, 1000, 5000, 10000, 10, 5000]
n = a.count
filtered_array = []
n.times do |i|
  if a[i] >= 1000
    filtered_array.push a[i]
  end
end
print filtered_array
```

A lo largo de este capítulo y de los siguientes resolveremos muchos problemas de estos tipos.

Resumen

- Los problemas más frecuentes sobre arreglos consisten en iterarlos, aplicar una o más instrucciones a cada uno de los elementos y guardarlos en un elemento nuevo. A partir de ahora seguiremos trabajando con esta lógica, resolviendo diversos tipos de problemas.
- Para iterar sobre un arreglo utilizamos las instrucciones de `while` y el método `.times`.
- Ambos métodos tienen la misma función. Cuando el índice va aumentando de uno en uno requiere de menos código utilizar `.times`. En caso contrario utilizaremos `while`, porque nos da más flexibilidad.

Ejercicios de arreglos

Desafío: Adictos a redes

Se tiene un arreglo con la cantidad de minutos usados en redes sociales de distintos usuarios. Se pide crear el programa `adictos_a_redes.rb` con un método llamado `scan_addicts` que reciba un arreglo con los minutos de uso y como resultado entregue un nuevo arreglo cambiando todas las medidas inferiores a 90 minutos como 'bien' y todas las mayores o iguales a 90 como 'mal'.

```
scan_addicts([120, 50, 600, 30, 90, 10, 200, 0, 500])
# => ["mal", "bien", "mal", "bien", "bien", "bien", "mal", "bien", "mal"]
```

Solución al ejercicio de arreglo

Se tiene un arreglo con la cantidad de minutos usados en redes sociales de distintos usuarios. Se pide crear un método llamado `scan_addicts` que reciba un arreglo con los minutos de uso y como resultado entregue un nuevo arreglo cambiando todas las medidas inferiores a 90 minutos como 'bien' y todas las mayores o iguales a 90 como 'mal'.

```
scan_addicts([120, 50, 600, 30, 90, 10, 200, 0, 500])
# => ["mal", "bien", "mal", "bien", "bien", "bien", "mal", "bien", "mal"]
```

Al resolver problemas de este tipo hay dos acercamientos igualmente válidos, podemos partir por la iteración del arreglo o podemos partir por el método. Hacerlo desde el método es mejor porque una vez que tengamos listo el ejercicio no tenemos que reescribirlo.

Creando el método

Para crear el método tenemos que saber lo que recibe y lo que devuelve. En este ejercicio recibe un arreglo y devuelve un arreglo nuevo.

```
def scan_addicts(array)
  results = []
  results # Lo que devuelve.
end
```

Iterando sobre el arreglo

```
def scan_addicts(array)
  results = []
  n = array.count
  n.times do |i|
    # Aquí es donde va la lógica de la iteración
  end
  results
end
```

Agregamos la lógica de la transformación, si es mayor que 90 guardamos 'mal' en el arreglo nuevo y si es menor o igual que 90, guardamos 'bien'. Para guardar ocuparemos el método .push

```
def scan_addicts(array)
  results = []
  n = array.count
  n.times do |i|
    if array[i] > 90
      results.push 'mal'
    else
      results.push 'bien'
    end
  end
  results # cuidado con el retorno, times retorna la cuenta
end
```

Desafío 2: Adictos a redes v2

Se pide crear el programa adictos_a_redes2 con un método llamado scan_addicts2 que reciba un arreglo con los minutos de uso y como resultado entregue un nuevo arreglo cambiando todas las medidas inferiores a 90 minutos como 'bien', entre 90 y 180 como 'mejorable' y todas las mayores o iguales a 180 como 'mal'.

Tip: Cuidado con las condiciones de borde, analiza los casos de 90 y 180.

```
## Solución

def scan_addicts2(array)
  results = []
  n = array.count
  n.times do |i|
    if array[i] >= 180
      results.push 'mal'
    elsif array[i] >= 90
      results.push 'mejorable'
    end
  end
  results
end
```

```

    else
      results.push 'bien'
    end
  end
end
results # cuidado con el retorno, times retorna la cuenta
end

print scan_addicts2([120, 90, 600, 30, 90, 10, 200, 180, 500])

```

```
["mejorable", "mejorable", "mal", "bien", "mejorable", "bien", "mal", "mal", "mal"]
```

Desafío 3: Transformando segundos a minutos

Se tiene un arreglo con la cantidad de segundos que demoraron algunos procesos y se necesita un método para transformar todos los datos a minutos (las fracciones de minuto serán ignoradas). Para realizar este programa se debe crear el archivo `s_to_m.rb`

El método debe llamarse `to_minutes`. Debe recibir el arreglo con los tiempos en segundos y devolverlo con los tiempos en minutos.

```
seconds = [100, 50, 1000, 5000, 1000, 500]
```

Solución

```

def to_minutes(array)
  n = array.count
  result = []
  n.times do |i|
    result.push array[i] / 60
  end
  result
end

seconds = [100, 50, 1000, 5000, 1000, 500]
to_minutes(seconds)

```

```
[1, 0, 16, 83, 16, 8]
```

Capítulo: Archivos, Arreglos y Strings

Objetivos

- Conocer el concepto de persistencia.
- Leer un archivo con datos y guardarlos en una variable.
- Transformar un array en un string.
- Transformar un string en un arreglo.
- Guardar los resultados en un archivo.

Motivación

En este capítulo seguiremos trabajando con el mismo tipo de problemas de los capítulos anteriores pero, en lugar de agregar el array manualmente dentro del código, aprenderemos a cargar este array desde un archivo.

Esto nos permitirá trabajar con datos que existen mas allá de la ejecución del programa.

Archivos y strings

Para poder leer y guardar información en archivos necesitamos aprender dos métodos que nos harán la vida muy fácil. `.split` y `.join`

- `.split` nos permite generar un array de strings a partir de un string más largo.
- `.join` nos permite convertir un array en un string.

`.split`

Split nos permite separar un string por algún criterio.

```
palabras = 'palabra1, palabra2, palabra3, palabra4'.split(',')  
# => ["palabra1", "palabra2", "palabra3", "palabra4"]
```

`.join`

`.join` nos permite unir un array con un criterio separador.

```
['palabra1', 'palabra2', 'palabra3', 'palabra4'].join(',')  
# => "palabra1, palabra2, palabra3, palabra4"
```

Creando un archivo para leer.

Necesitamos un archivo para poder leer desde ruby. Crearemos uno con el editor de texto y lo guardaremos en la misma carpeta de nuestro programa.

Crearemos el archivo data en la misma carpeta en que está nuestro código y lo llamaremos data (sin extensión)

Dentro del archivo guardaremos la siguiente información.

```
1,2,3,4,5,6,7,8,9,10
```

Observación: No dejar espacios entre las comas

Abrir y leer el archivo

Para abrir un archivo ocuparemos el método open, pero open solo nos devuelve un archivo abierto, pero no el texto que contiene. Para leerlo ocuparemos el método .read

```
data = open('data').read  
# => "1,2,3,4,5,6,7,8,9,10\n"
```

Si el archivo no está en la carpeta o nos equivocamos de nombre del archivo obtendremos este error:

```
Errno::ENOENT: No such file or directory @ rb_sysopen - data
```

Eliminamos el salto de línea y lo transformamos en array

Para limpiar el salto de línea ocuparemos `.chomp`

```
data = open('data').read.chomp  
# => "1,2,3,4,5,6,7,8,9,10"
```

Transformando los datos del archivo a un array

Podemos convertir todos los datos en un array utilizando split.

```
data = open('data').read.chomp.split(',')
```

Transformando los datos a enteros

```
data = open('data').read.chomp.split(',')
array = []
data.each do |d|
  array.push d.to_i
end
print array # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Si transformamos los datos a enteros no es necesario limpiar el salto de línea
```

Leyendo un archivo con información en múltiples líneas.

Es posible que los datos vengan en distintas líneas del archivo. Para esto vamos a utilizar un archivo con datos en distintas líneas y lo llamaremos `data2`. Dentro de el guardaremos la siguiente información.

```
21
10
6
9
11
0
2
3
50
```

Leyendo el archivo con `.readlines`

Para hacer sencilla la lectura del archivo y traer todos los datos como un arreglo ocuparemos `.readlines`.

```
data = open('archivo2').readlines ##=> ["21\n", "10\n", "6\n", "9\n", "11\n", "0\n", "2\n", "3\n", "50\n"]
```

Removiendo los saltos de línea

Veremos aquí que tenemos múltiples saltos de línea, uno por cada dato. Para limpiarlo podemos aplicar el método `.chomp` a cada uno de los elementos, pero si los transformamos a enteros se eliminará el `\n` automáticamente.


```
original_data = open('archivo2.txt').readlines
lines = original_data.count
array = []
lines.times do |i|
  array << original_data[i].to_i
end
```

Reutilizando la lógica con un método

El código para leer archivos lo reutilizaremos en algunos ejercicios, así que lo dejaremos dentro de un método para reutilizarlo de forma sencilla.

Dejaremos como parámetro el nombre del archivo y retornaremos los datos como un array.

```
def read_file(filename)
  original_data = open(filename).readlines
  lines = original_data.count
  array = []
  lines.times do |i|
    array << original_data[i].to_i
  end
  return array
end

read_file("archivo2.txt")
# => [21, 10, 6, 9, 11, 0, 2, 3, 50]
```

Ejercicio resuelto

Dado el archivo2 que tiene los siguientes datos

```
21
10
6
9
11
0
2
3
50
```

Se pide un crear un programa que tome los datos de ese archivo y construya un arreglo con los mismos pero transformando todos los valores mayores de 20 a un máximo de 20.

Solución

```
def read_file(filename)
  original_data = open(filename).readlines
  lines = original_data.count
```

```

array = []
lines.times do |i|
  array << original_data[i].to_i
end
return array
end

data = read_file("archivo2.txt")
n = data.count
n.times do |i|
  data[i] = 20 if data[i] > 20
end

```

Guardando los resultados

Existen muchas formas de guardar datos en un archivo, una de las más sencillas es:

```
File.write('/path/to/file', 'datos')
```

Si no especificamos estaremos utilizando una relativa al archivo que estamos ejecutando, y los datos tienen que ser un string.

```

## Guardando los resultados

def read_file(filename)
  original_data = open(filename).readlines
  lines = original_data.count
  array = []
  lines.times do |i|
    array << original_data[i].to_i
  end
  return array
end

data = read_file("archivo2")
n = data.count
n.times do |i|
  data[i] = 20 if data[i] > 20
end

File.write('output', data.join("\n"))

```

!!! Cuidado !!!

Al abrir un archivo para guardar datos sobreescribiremos el contenido. Esto es peligroso porque, si no somos cuidadosos, podríamos destruir un archivo que nos sirve.

Persistencia

Se llama **persistencia** cuando los datos viven más allá del programa que los usa, por ejemplo guardando los resultados en archivos que luego otro programa (o el mismo) utiliza.

Los archivos son una forma de hacer persistente los datos pero existen otras, como las bases de datos.

Resumen

En este capítulo aprendimos a abrir archivos y leer datos de ellos, así como transformar los datos leídos y luego volver a guardarlos.

Para leer archivos vimos los métodos:

- `open(filename).read`
- `open(filename).readlines`
- `.read` es útil cuando queremos traer todo el archivo como un solo string a memoria, luego lo separamos ocupando el método `split` y algún criterio de separación
- `.readlines` es útil cuando queremos leer todo el archivo como un arreglo donde cada línea es un elemento del arreglo.

Para guardar datos en archivo ocuparemos:

- `File.write('output', data)`

Donde `data` es un string, pero si tenemos un array, podemos ocupar el método `.join` para transformarlo en un string.

Capítulo: Iterando elemento a elemento con .each

Objetivos:

- Recorrer un arreglo elemento a elemento.
- Transformar elementos de un arreglo.

Motivación

Existen varios tipos de problema donde el índice del elemento no es relevante.

En este capítulo estudiaremos cómo identificar estos casos y cómo resolverlos sin necesidad de ocupar el índice, lo que nos permitirá escribir nuestro código de forma más sencilla.

Iterando con .each

Podemos iterar un arreglo elemento a elemento ocupando el método `.each` y un bloque

```
a = [1, 2, 6, 1, 7, 2, 3]
a.each do |ele|
  puts ele
end
# 1
# 2
# 6
# 1
# 7
# 2
# 3
```

También podemos utilizar un bloque inline

```
a = [1, 2, 6, 1, 7, 2, 3]
a.each {|ele| puts ele}
```

Trabajando elemento a elemento podemos resolver **casi** los mismos problemas que resolvimos ocupando índices y con menos código.

Transformando con .each

El método `.each` nos permite recorrer un arreglo y transformar todos sus elementos, por ejemplo, si tenemos un listado de precios y queremos aumentar cada uno de los precios un 20% (o sea multiplicar por 1.2 cada uno de los precios):

```
prices = [120, 210, 309, 104, 192]
new_prices = []

prices.each do |price|
  new_prices.push(price * 1.2)
end

print new_prices
```

Limitación importante

Al acceder solo a los elementos y no a la posición de estos, no podemos modificarlos directamente dentro del arreglo, por lo que necesitamos un arreglo nuevo para guardar los resultados.

Ejercicio

Crear el programa `aumento_precios.rb` dentro del programar crear un método llamado `augment` que recibe un arreglo y un multiplicador y que como resultado de un arreglo con todos los valores aumentados.

Solución:

```
def augment(array, factor)
  new_array = []
  array.each do |price|
    new_array.push(price * factor)
  end
  new_array
end

print augment([10,20,10], 1.2) # [12.0, 24.0, 12.0]
```

```
[12.0, 24.0, 12.0]
```

Desafío:

Supongamos que tenemos un caso donde tenemos un arreglo de notas y queremos calcular el promedio, pero dentro de este arreglo tenemos alumnos que no dieron la prueba y están marcados en el arreglo como 'N.A', Como regla adicional cada N.A tendrá nota base 2.0.

```
notas = [5, 7, 1, 3, 5, 8, 9, 'N.A', 'N.A', 3]
```

Se pide crear el programa `arreglo_notas.rb` con el método promedio que devuelva el promedio de un arreglo de notas con las características entregadas.

Pistas: Para resolver este problema tenemos dos grandes pasos

1. Transformar todos los 'N.A' a nota 2
2. Sumar y dividir por la cuenta de elementos

Solución

```
notas = [5, 7, 1, 3, 5, 8, 9, 'N.A', 'N.A', 3]
#transformamos
notas_transformadas = []
notas.each do |nota|
  if nota == 'N.A'
    notas_transformadas.push 2
  else
    notas_transformadas.push nota
  end
end

# En este punto notas_transformadas tiene los datos que necesitamos

# Calculamos el promedio
puts notas_transformadas.sum / notas_transformadas.count.to_f
```

Capítulo: Filtrando con .each

Objetivo:

- Filtrar elementos de un arreglo.

Introducción

En muchos tipos de problemas necesitamos eliminar elementos de un arreglo en base a una condición. En este capítulo aprenderemos a hacerlo con el método `.each`

Filtrando con .each

Supongamos que tenemos un arreglo de notas y queremos mostrar todas las notas superiores a 5

```
array = [8, 2, 5.3, 7, 2, 9, 9, 6]
array.each do |ele|
  if ele > 5
    puts ele
  end
end
```

También sería posible generar un nuevo arreglo solo con esos elementos

```
array = [8, 2, 5.3, 7, 2, 9, 9, 6]
new_array = []
array.each do |ele|
  if ele > 5
    new_array.push(ele)
  end
end
```

Para ordenar mejor nuestro código podemos crear un método que filtre y devuelva un arreglo nuevo.

```
def filter(array, value)
  new_array = []
  array.each do |ele|
    if ele > value
      new_array.push(ele)
    end
  end
  new_array #devolvemos el arreglo nuevo
end

# Lo probamos
a = [8, 2, 5.3, 7, 2, 9, 9, 6]
filter(a, 5)
```

El retorno de .each

`.each` cuando termina de iterar devuelve el arreglo original, y Ruby devuelve implícitamente la última línea. Es por eso que tenemos el nombre del arreglo nuevo al final.

Más adelante aprenderemos que el método `.select` resuelve esto con mucho menos trabajos, pero es importante para nuestra formación profesional que desarrollemos las capacidades lógicas para resolver problemas con arrays.

¿Cómo saber cuándo ocupar .each o .times?

Si el problema no requiere los índices de los elementos del arreglo entonces podemos utilizar `.each` y resolver el problema con menos líneas de código, pero no hay problemas de ocupar `.times` si así lo prefieres.

Capítulo: Otras formas de iterar sobre arreglos

Objetivos

En este capítulo aprenderemos que existen métodos más específicos que el `.each` para iterar, transformar, filtrar y reducir valores. Utilizar estos métodos nos ayudará a expresar nuestras ideas con menos código.

En particular aprenderemos a:

- Aplicar el método `.map` para transformar datos en un arreglo
- Diferenciar los métodos `.each` y `.map`
- Aplicar los métodos `.select` y `.reject` para seleccionar y remover datos de un arreglo
- Aplicar el método `.inject` para transformar un arreglo en un único dato.

Introducción

Además de la iteración con `.each` hay tres formas muy interesante de operar sobre los datos de un array.

- `map` o `collect`
- `select` o `reject`
- `inject`

Empecemos estudiando el método `.map`

Map

Map sirve para aplicar una operación a cada elemento del array, y devuelve un array con los resultados de las operaciones aplicadas.

```
# con map
a = [1, 2, 3, 4, 5, 6, 7]
b = a.map do |e|
  e * 2
end
# => [2, 4, 6, 8, 10, 12, 14]
```

```
# con .each
a = [1, 2, 3, 4, 5, 6, 7]
b = []

a.each do |e|
  b.push(a * 2)
end
```

O con bloque inline

```
a = [1, 2, 3, 4, 5, 6, 7]
b = a.map { |e| e * 2 }
```

Comprendiendo el .map

`.map` parece mágico, pero lo que hace es aplicar una o más instrucciones a cada uno de los elementos y devolver el resultado de la aplicación de la última instrucción como el nuevo elemento.

```
a = [1, 2, 3, 4, 5, 6, 7]
b = a.map { |e| 1 }
```

```
[1, 1, 1, 1, 1, 1, 1]
```

El arreglo nuevo solo tiene el resultado de la última operación en cada elemento.

```
b = a.map do |e|
  e += 100
  1
end
```

Analicemos esta situación

```
b = a.map do |e|
  e += 100 << # Qué hace esto?, ¿suma o no suma?
  1
end
```

Todas las instrucciones se realizan, pero el resultado es descartado.

```

b = a.map do |e|
  puts "antes: #{e}"
  e += 100 # << Qué hace esto?, ¿suma o no suma?
  puts "después: #{e}"
  1 ## << Esto es lo importante
end

```

```

antes: 1
después: 101
antes: 2
después: 102
antes: 3
después: 103
antes: 4
después: 104
antes: 5
después: 105
antes: 6
después: 106
antes: 7
después: 107

```

```
[1, 1, 1, 1, 1, 1, 1]
```

Podemos aprovechar este comportamiento para transformar un valor si cumple una condición.

```

a = [1, 2, 3, 4, 5]
b = a.map do |e|
  if e > 4
    4
  else
    e
  end
end
end

```

```
[1, 2, 3, 4, 4]
```

Podemos hacer más sencillo nuestro código utilizando el operador condicional `?`, el cual es una forma de escribir un if en una sola línea.

Usos de map

Pensaremos en ocupar `.map` cuando tengamos que hacer transformaciones u operaciones a todos los elementos.

Transformar datos de un tipo a otro

`.map` es muy útil para transformar un arreglo con datos de un tipo a otro tipo

```
arr = ['1', '2', '3', '4']  
result = arr.map { |x| x.to_i }  
# => [1, 2, 3, 4]
```

Operar sobre todos los datos

`.map` también sirve para aplicar una operación a cada uno de los elementos de un arreglo. Por ejemplo podríamos tener un arreglo de datos que guarde tiempo de procesos en milisegundos y los queremos transformar a segundos

```
tiempos = [10000, 50000, 3000, 21000]  
tiempos.map { |x| x / 1000 }
```

```
[10, 50, 3, 21]
```

Otro ejemplo sería transformar todos los elementos de un arreglo de nombres a minúsculas.

```
nombres = ['Violeta', 'Andino', 'Clemente', 'Pia', 'Ray', 'Camilo']  
nombres.map { |x| x.downcase }
```

```
["violeta", "andino", "clemente", "pia", "ray", "camilo"]
```

O podríamos contar la cantidad de letras que tiene cada nombre.

```
largos = nombres.map { |x| x.length }
```

```
[7, 6, 8, 3, 3, 6]
```

Remplazar datos

`.map` también nos permite remplazar un dato fuera de un rango. Por ejemplo si queremos que cualquier valor mayor que dos sea remplazado por dos.

```
arr = [1, 2, 3, 4]
result = arr.map { |x| x < 2 ? x : 2 }
```

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

.map y .collect

El método `.map` tiene un alias llamado `.collect`. Esto quiere decir que `.map` y `.collect` son exactamente lo mismo, no hay ninguna diferencia entre ocupar uno u otro.

El método `.map` se llama así porque se le indica el camino entre los valores originales y los valores finales del arreglo (indica el camino, genera un mapa). Por otro lado `.collect` recolecta los elementos del arreglo y trabaja sobre cada uno de ellos.Cuál depende completamente del gusto personal.

```
a = [1, 2, 3, 4, 5, 6, 7]
b = a.collect { |e| e * 2 }
```

```
[2, 4, 6, 8, 10, 12, 14]
```

.map vs .each

`.map` es mas directo que `.each` cuando queremos transformar todos los datos de un arreglo porque automáticamente devuelve un arreglo nuevo con los datos que necesitamos en lugar de tener que nosotros crear un arreglo vacío.

.map!

`.map!` es una variante de `.map` que modifica el arreglo original y de esta forma no tenemos que guardar el resultado en una variable nueva.

```
arr = ['1', '2', '3', '4']  
arr.map! { |x| x.to_i }
```

```
[1, 2, 3, 4]
```

`.map!` sólo debemos ocuparlo cuando no nos interesa volver a utilizar los datos originales.

Desventajas de map

El método `.map` no es igualmente útil para todos los casos, por ejemplo en casos donde hay que filtrar elementos no es la mejor opción.

Veamos un ejemplo donde de un arreglo queremos seleccionar los valores mayores o iguales a 3

```
a = [1, 2, 3, 4]  
a.map { |x| x >= 3 ? x : nil } # => [nil, nil, 3, 4]
```

Si bien podemos remover los valores `nil` del arreglo utilizando `.compact` hay una forma más eficiente y mucho más expresiva para hacerlo. Se llama `.select`

Select

`.select` permite filtrar los elementos de un arreglo bajo una condición. `.select` al igual que `.map` devuelve un array nuevo sin modificar el original.

```
a = [1, 2, 3, 4, 5, 6, 7]  
b = a.select { |x| x % 2 == 0 } # seleccionamos todos los pares  
  
# => [2,4,6]
```

Aunque a diferencia de `.map`, el arreglo puede tener menos elementos.

Reject

Hay un método que hace lo contrario a `.select`, se llama `.reject` y lo que hace es eliminar a todos los elementos del arreglo que no cumplan con el criterio.

```
a = [1, 2, 3, 4, 5, 6, 7]  
b = a.reject { |x| x.even? } # Eliminamos todos los números pares
```

```
[1, 3, 5, 7]
```

`.select` y `.reject` tienen usos más allá de los números. Supongamos que tenemos un arreglo que tiene números y palabras y queremos seleccionar solo las palabras.

```
a = [1, 'hola', 2, 'aprendiendo', 3, 'ruby'].select{ |x| x.class == String }
```

```
["hola", "aprendiendo", "ruby"]
```

O por ejemplo si queremos descartar palabras muy largas de un array.

```
a = ['Supercalifragilisticexpialidocious', 'hola'].reject{ |x| x.length > 10 }
```

```
["hola"]
```

Inject

Inject permite operar sobre todos los elementos, pero en lugar de devolver un arreglo, devuelve un único valor con el resultado de las operaciones, por lo que en el bloque hay que pasar dos variables: La que itera y la que guarda el resultado.

```
a = [1, 2, 3, 4]
b = a.inject(0){ |sum, x| sum + x }
```

```
10
```

`.inject` tiene un valor inicial, el que va dentro de los paréntesis, un variable que guarda el último valor y luego la variable que itera.

Usos de .inject

`.inject` puede llegar a tener usos muy creativos, por ejemplo seleccionar la palabra más larga de un arreglo.

```
palabras = ['Supercalifragilisticexpialidocious', 'Supercalifrag', 'Super', 'Su']  
resultado = palabras.inject("") { |longest, word| longest.length >= word.length ? longest : word }
```

```
"Supercalifragilisticexpialidocious"
```

¿Cómo es que esto funciona, nunca se hace una asignación a `longest` ?

Esto es porque en realidad la variable que guarda el resultado lo va haciendo cada vez que finaliza una iteración del bloque. Dominar el método `.inject` requiere una gran capacidad de abstracción y no hay problema en ocupar `.each` para resolver inicialmente un problema y luego mejorar el código.

Desafíos

Desafío 1:

Dado el array:

```
a = [1, 9, 2, 10, 3, 7, 4, 6]
```

1. Utilizando map sumar uno a cada uno de los valores del array.

- Utilizando map convertir todos los valores a float.
- Utilizando select descartar todos los elementos menores a 5 en el array.
- Utilizando inject sumar todos los valores del array.
- Utilizando .count contar todos los elementos menores que 5.

solución

Ejercicio 1

```
a = [1, 9, 2, 10, 3, 7, 4, 6]
```

Utilizando map sumar 1 a cada uno de los valores del array.

```
a.map { |x| x + 1 }
```

Utilizando map convertir todos los valores a float.

```
a.map { |x| x.to_f }
```

Utilizando select descartar todos los elementos menores a 5 en el array.

```
a.select { |x| x > 5 }
```

Utilizando inject sumar todos los valores del array.

```
a.inject { |sum, x| x + sum }
```

```
a.count { |x| x < 5 }
```

4

Desafío 2

```
nombres = ['Violeta', 'Andino', 'Clemente',  
           'Javiera', 'Paula', 'Pía', 'Ray']
```

- Obtener todos los elementos que excedan los 5 caracteres, utilizando `.select`.
- Utilizar `.map` para crear un arreglo con todos los nombres en minúscula.

- Utilizar `.select` para crear un arreglo con todos los nombres que empiecen con P.
- Utilizando `.count`, contar los elementos que empiecen con 'A', 'B' o 'C'.
- Utilizando `.map`, crear un arreglo único con la cantidad de letras que tiene cada nombre.

Solución al ejercicio 2

```
nombres = ['Violeta', 'Andino', 'Clemente', 'Javiera', 'Paula', 'Pia', 'Ray', 'Camilo']  
nombres.select { |x| x.length > 5 } # Los elementos que excedan mas de 5 caracteres utilizando select.  
nombres.map { |x| x.downcase }  
nombres.select { |x| x if x[0] == 'P' }  
nombres.count { |x| x[0] == 'A' || x[0] == 'B' || x[0] == 'C' }  
nombres.map { |x| x.length }
```

```
[7, 6, 8, 7, 5, 3, 3, 6]
```

Capítulo: Problemas con dos arrays

Objetivos

- Trabajar con datos repartidos a lo largo de dos arrays

En algunas ocasiones tendremos nuestros datos guardados en dos arreglos y los relacionaremos a través del índice.

```
notas = [5, 9, 6, 8, 9]
alumnos = ['Juliana', 'Francisca', 'Pedro', 'Diego', 'Macarena']
```

En el ejemplo Juliana tiene nota 5, Francisca tiene nota 9, y así sucesivamente. Se nos pide construir un programa donde introduzcamos el nombre de un alumno y, si este existe, se nos devuelve la nota.

En ambos arreglos los elementos se corresponden por índice: Juliana tiene el índice 0 y `notas[0]` es 5, Francisca tiene el índice 1 en el arreglo de alumnos y `notas[1]` es 9.

Algoritmo

- El usuario introduce un string
- Buscamos el índice en el arreglo de alumnos de ese string
- Si existe el índice:
 - buscamos la nota dentro del arreglo de notas
- Si no:
 - Mostramos que no pudimos encontrar la nota.

```
nombre_a_buscar = gets.chomp
indice = alumnos.index(nombre_a_buscar)
if indice
  puts "la nota es: #{notas[indice]}"
else
  puts "No se encontró a #{nombre_a_buscar}"
end
```

Versión con métodos

Se nos pide construir un método que reciba los arrays, el nombre de la persona y de como resultado su nota.

```
def search(name, names, scores)
  i = names.index(name)
  if i
    scores[i]
  else
    nil
  end
end
```

Versión 2.0

```
def search(name, names, scores)
  i = names.index(name)
  scores[i] if i
end
```

Operando sobre valores en dos arrays

En algunos casos tendremos los datos a lo largo de dos arrays y necesitaremos operar sobre ellos, Un ejemplo de esto podría ser que tengamos un array con los datos de ventas en una ciudad y un segundo array con los datos de ventas de otra ciudad

Ejemplo de ventas en una empresa

Para simplificar el ejercicio diremos que vienen en un arreglo (en lugar de obtenerlos de un archivo)

```
v1 = [100, 20, 50, 70, 90]
v2 = [150, 30, 50, 20, 30]
```

Diremos que los números representan la cantidad de ventas cada día, de esta forma `v1[0]` representan las ventas del primer día en la tienda 1, `v1[1]` representa las ventas del segundo día en la tienda 1 y `v2[0]` representa las ventas día 1 de la tienda 1.

¿Cómo podemos obtener las ventas diarias de las 2 tiendas en conjunto? Sumando los elementos.

Por elemento o por índice

Nuevamente, antes de resolver un ejercicio de este tipo, debemos preguntarnos qué tiene mas sentido: Movernos por elemento o por el índice.

Dado que tenemos que iterar elementos de dos arreglos simultáneamente tiene sentido hacerlo simplemente por el índice.

Iteraremos sobre los índices y guardaremos los elementos en un nuevo arreglo.

```
v1 = [100, 20, 50, 70, 90]
v2 = [150, 30, 50, 20, 30]
vt = [] # Ventas totales
n = v1.count
```

```
n.times do |i|
  vt.push v1[i] + v2[i]
end
```

```
print vt
```

```
[250, 50, 100, 90, 120]
```

Ejemplo de torneo

Un ejemplo parecido es construir un listado de todos contra todos. Por ejemplo, si hay 3 equipos, e1, e2 y e3, sería:

```
e1 V.S e2
e1 V.S e3
e2 V.S e1
e2 V.S e3
e3 V.S e1
e3 V.S e2
```

Solución 1

```
a = ['Equipo 1','Equipo 2','Equipo 3','Equipo 4','Equipo 5']
b = ['Equipo 1','Equipo 2','Equipo 3','Equipo 4','Equipo 5']
t = []
```

```
a.each do |e1|
  b.each do |e2|
    t.push "#{e1} V.S #{e2}"
  end
end
```

```
puts t.join("\n")
```

```
Equipo 1 V.S Equipo 1
Equipo 1 V.S Equipo 2
Equipo 1 V.S Equipo 3
Equipo 1 V.S Equipo 4
Equipo 1 V.S Equipo 5
Equipo 2 V.S Equipo 1
Equipo 2 V.S Equipo 2
Equipo 2 V.S Equipo 3
```

Equipo 2 V.S Equipo 4
Equipo 2 V.S Equipo 5
Equipo 3 V.S Equipo 1
Equipo 3 V.S Equipo 2
Equipo 3 V.S Equipo 3
Equipo 3 V.S Equipo 4
Equipo 3 V.S Equipo 5
Equipo 4 V.S Equipo 1
Equipo 4 V.S Equipo 2
Equipo 4 V.S Equipo 3
Equipo 4 V.S Equipo 4
Equipo 4 V.S Equipo 5
Equipo 5 V.S Equipo 1
Equipo 5 V.S Equipo 2
Equipo 5 V.S Equipo 3
Equipo 5 V.S Equipo 4
Equipo 5 V.S Equipo 5

Solución 2: Eliminando los repetidos

```
a = ['Equipo 1','Equipo 2','Equipo 3','Equipo 4','Equipo 5']  
b = ['Equipo 1','Equipo 2','Equipo 3','Equipo 4','Equipo 5']  
t = []  
  
a.each do |e1|  
  b.each do |e2|  
    if e1 != e2  
      t.push "#{e1} V.S #{e2}"  
    end  
  end  
end  
  
puts t.join("\n")
```

Equipo 1 V.S Equipo 2
Equipo 1 V.S Equipo 3
Equipo 1 V.S Equipo 4
Equipo 1 V.S Equipo 5
Equipo 2 V.S Equipo 1
Equipo 2 V.S Equipo 3
Equipo 2 V.S Equipo 4
Equipo 2 V.S Equipo 5
Equipo 3 V.S Equipo 1
Equipo 3 V.S Equipo 2
Equipo 3 V.S Equipo 4
Equipo 3 V.S Equipo 5
Equipo 4 V.S Equipo 1
Equipo 4 V.S Equipo 2
Equipo 4 V.S Equipo 3
Equipo 4 V.S Equipo 5
Equipo 5 V.S Equipo 1
Equipo 5 V.S Equipo 2

Equipo 5 V.S Equipo 3
Equipo 5 V.S Equipo 4

Solución 3: Bonus

Si siempre son los mismos equipos se puede trabajar con uno solo de los arreglos

```
a = ['Equipo 1','Equipo 2','Equipo 3','Equipo 4','Equipo 5']  
t = []  
  
a.each do |e1|  
  a.each do |e2|  
    if e1 != e2  
      t.push "#{e1} V.S #{e2}"  
    end  
  end  
end  
  
puts t.join("\n")
```

Equipo 1 V.S Equipo 2
Equipo 1 V.S Equipo 3
Equipo 1 V.S Equipo 4
Equipo 1 V.S Equipo 5
Equipo 2 V.S Equipo 1
Equipo 2 V.S Equipo 3
Equipo 2 V.S Equipo 4
Equipo 2 V.S Equipo 5
Equipo 3 V.S Equipo 1
Equipo 3 V.S Equipo 2
Equipo 3 V.S Equipo 4
Equipo 3 V.S Equipo 5
Equipo 4 V.S Equipo 1
Equipo 4 V.S Equipo 2
Equipo 4 V.S Equipo 3
Equipo 4 V.S Equipo 5
Equipo 5 V.S Equipo 1
Equipo 5 V.S Equipo 2
Equipo 5 V.S Equipo 3
Equipo 5 V.S Equipo 4

Capítulo: Los arrays como conjuntos

Objetivos

- Realizar operaciones típicas de conjunto sobre arrays

Es posible hacer operaciones propias de conjuntos sobre los arreglos. Para esta sección tendremos dos arreglos, a y b

```
a = [1, 2, 3]
b = [3, 4, 5]
```

Concatenación

¿Recuerdan la concatenación de strings? el operador suma (+) actúa de la misma forma en los arreglos: Genera uno con todos los elementos de los originales. Se puede ver como una suma de conjuntos: Al conjunto a se le suman los elementos del conjunto b

```
a + b
# => [1, 2, 3, 3, 4, 5]
```

Por ejemplo si tenemos un ranking dividido en dos arreglos, top_10 y el_resto se necesita mostrar todos los resultados, podemos unir las listas.

```
top_10 + el_resto
```

Diferencia de elementos

Al conjunto a se le quitan los elementos del conjunto b. Dicho de otra forma: ¿Qué tiene a que no tenga b? Con esta operación podemos mostrar lo que hace único a nuestro arreglo:

```
a - b
# => [1, 2]
```

Un ejemplo podría ser que tenemos una lista de puntajes, y otra lista con los puntajes de los participantes que hicieron trampa y queremos mostrar los resultados descartando a los tramosos.

Unión

Esta operación entrega los elementos que tienen en común ambos conjuntos. Visto de otra forma: Son los elementos que están en `a` o en `b`

```
a | b  
# => [1, 2, 3, 4, 5]
```

Intersección

La intersección indica los elementos que están en ambos conjuntos al mismo tiempo: Los elementos que pertenecen al conjunto `a` y al conjunto `b`

```
a & b  
# => [3]
```

Capítulo: Mutabilidad

Objetivos:

- Conocer el concepto de mutabilidad.
- Duplicar objetos antes de operar con ellos.

Motivación

En este capítulo estudiaremos como trabajar con un array cuando necesitamos transformar sus datos pero además necesitamos guardar los originales.

Este tipo de situaciones se dan en el trabajo diario, por ejemplo si tenemos un arreglo de datos con precios y queremos hacer distintas simulaciones, una aumentando los precios de los primeros productos, luego otra aumentando los últimos y finalmente una aumentando todos.

Para resolver esto tendremos que conocer el concepto **mutabilidad**.

Introducción a mutabilidad

En lugar de partir explicando el problema con el caso del arreglo de precios y tener que resolver con ciclos reduciremos el problema a la situación más sencilla.

```
a = ["hola", "yo", "soy", "un", "arreglo"]
b = a
b[0] = "nos vemos"
puts a[0] #nos vemos
```

Cambiamos parte del arreglo guardado en la variable `b` pero sin intención también cambiamos el arreglo dentro de la variable `a`, y esto sucede porque ambos son el mismo arreglo.

En este capítulo estudiaremos este fenomeno y como prevenirlo.

Mutabilidad

La mutabilidad quiere decir que un objeto puede cambiar después de que fue creado.

```
a = ["hola", "yo", "soy", "un", "arreglo"] # aquí lo creamos
b = a
b[0] = "nos vemos" # aquí lo modificamos
puts a[0] #nos vemos
```

Asignación vs modificación

Asignar un nuevo objeto es distinto a crear un objeto

```
a = 8
a = 10 # Aquí estamos asignando el objeto 10

a = [0,1,2,3,4,5]
a[0] = 100 # Aquí estamos modificando el array
```

Mutabilidad y string

Los strings en ruby también son mutables.

```
a = "hola"
a[0] = "H"
puts a => "Hola"
```

En cambio los Integers y los Floats no son mutables, o sea no hay forma de cambiar el valor al número 2 sin que sea otro número.

En resumen hasta ahora

- Existen objetos que pueden cambiar se llaman mutables.
- Existen objetos que no pueden cambiar se llaman inmutables.
- La mutabilidad es peligrosa si no tenemos cuidado porque podemos cambiar una variable por error.

¿Cómo podemos saber si son el mismo objeto o no?

Copiando un arreglo para evitar cambios indeseados

Para esto ocuparemos el método `.dup`

```
a = [1, 2, 3, 4]
b = a.dup
a[0] = 100
print a #[100, 2, 3, 4]
print b [1, 2, 3, 4]
```

Capítulo: Arrays dentro de arrays

Objetivos

- Iterar arreglos bidimensionales

Introducción

Los arrays pueden contener otros arrays en su interior:

```
a = [[1, 2, 3], 4, 5]
```

Esto nos entrega nuevas formas para guardar información que tiene diversas ventajas pero presenta nuevos desafíos. Por ejemplo gracias a esto podremos representar matrices o poder leer un archivo con múltiples líneas y múltiples datos por línea.

Iterando un array con arrays

```
array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
array.each do |array_int|
  array_int.each do |ele|
    puts ele
  end
end
```

Entendiendo el proceso de iteración

```
array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
array.each do |array_int|
  array_int.each do |ele|
    puts ele
  end
end
```

El proceso de iteración es igual al de un array normal, solo que por cada elemento también obtendremos un array que tenemos que volver a iterar, por eso necesitamos dos ciclos.

Iterando un array con arrays a partir de los índices

```
array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
n = array.count
n.times do |i|
  n.times do |j|
    print "\t#{array[i][j]}"
  end
  puts
end
```

```
1  2  3
4  5  6
7  8  9
```

3

Debemos tener mucho cuidado con los índices, especialmente cuando la cantidad de elementos son distintos

```
array = [[1, 2, 3], [4, 5, 6, 91], [7, 8, 9, 10]]
n = array.count
n.times do |i|
  n.times do |j|
    print "\t#{array[j][i]}"
  end
  puts
end
```

```
1  4  7
2  5  8
3  6  9
```

3

Contando los elementos internos en cada iteración

Debemos tener mucho cuidado con los índices, especialmente cuando la cantidad de elementos es distintos, porque podríamos estar apuntando fuera de los bordes de un arreglo.

```

array = [[1, 2, 3], [4, 5, 6, 91], [7, 8, 9, 10]]
n_exterior = array.count
n_exterior.times do |i|
  n_interior = array[i].count
  n_interior.times do |j|
    print "\t#{array[i][j]}"
  end
  puts
end

```

```

1  2  3
4  5  6  91
7  8  9  10

```

3

¿Qué sucede si cambiamos el índice i por j?

```

array = [[1, 2, 3], [4, 5, 6, 91], [7, 8, 9, 10]]
n_exterior = array.count
n_exterior.times do |i|
  n_interior = array[i].count
  n_interior.times do |j|
    print "\t#{array[i][j]}"
  end
  puts
end

```

```

1  2  3
4  5  6  91
7  8  9  10

```

3

Capítulo: Matrices

Introducción

Una matriz son simplemente datos agrupados de forma rectangular.

$$\begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$$

En ruby las podemos escribir como un arreglo de arreglos. A esto también se le denomina arreglo de dos dimensiones.

Motivación

No es importante que nos convirtamos en expertos en el trabajo de matrices si es importante que desarrollamos las abstracciones necesarias para trabajar con arreglos que contengan otros arreglos.

Creando una matriz con arreglos

Podemos implementar una matriz con arreglos de forma sencilla, simplemente agruparemos los datos por fila donde cada fila es un arreglo.

```
m = [[3, 2],[1, 4]]
```

Mostrando la matriz

Mostrar una matriz es exactamente el mismo ejercicio que mostrar un arreglo de arreglos.

```
m = [[3, 2],[1, 4]]

m.each do |row|
  row.each do |ele|
    print "\t#{ele}"
  end
  print "\n"
end
```

Podemos crear un método para mostrar una matriz. Para crear este método recordemos que necesitamos saber que parámetros necesita y que resultado entregará.

Si solo queremos que la muestra en pantalla no es necesario que el método retorne valor alguno. El método si necesita recibir un arreglo de arreglos que es lo que mostrará.

```
def show(matrix)
  matrix.each do |row|
    row.each do |ele|
      print "\t#{ele}"
    end
    print "\n"
  end
end
```

Sumando matrices

Sumas matrices es un ejercicio muy sencillo, consiste en sumar cada uno de los elementos de cada una de las matrices en la posición correspondiente.

$$\begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 6 & 4 \\ 2 & 8 \end{bmatrix}$$

```
m1 = [[3, 2],[1, 4]]
m2 = [[3, 2],[1, 4]]
result = []

cols = m1.length
rows = m1[0].length

cols.times do |i|
  rows.times do |j|
    print (m1[i][j] + m2[i][j]).to_s + " "
  end
  print "\n"
end
```

```
6 4
2 8
```

```
2
```

En ruby existe un tipo de dato específico para matrices, llamado matrix que está fuera del alcance de este estudio pero puedes complementar con la documentación oficial.

<https://ruby-doc.org/stdlib-2.5.1/libdoc/matrix/rdoc/Matrix.html>

Capítulo: Archivos y múltiples arrays

Objetivo

- Leer y procesar archivos de múltiples líneas y varios registros por línea.

Introducción

El escenario mas frecuente a la hora de trabajar con archivos es que una archivo tenga varios registros por línea y múltiples línaes, por ejemplo un archivo podría tener un set de datos de empresas y ventas por periodo.

```
Empresa1, 100, 20, 30, 50
Empresa2, 200, 10, 20, 30
```

Otros ejemplos

O ser un registro de alumnos y notas

```
Camilo, 90, 50 80, 90
Francisca 100, 40, 100, 100
...
Verónica 60, 70, 80, 100
```

En este capítulo aprenderemos a leer estos archivos, guardar la información en arreglos e iterar los resultados.

Archivos CSV

Los archivos como los que vimos en la introducción tienen un formato llamado CSV, es un formato similar al de un excel pero en formato plano, esto quiere decir que lo podemos leer sin necesidad de protocolos complejos desde cualquier programa. coma.

Leyendo un CSV

Para leer un CSV lo primero que haremos será incorporar la clase CSV. Esto lo podemos hacer con la instrucción `require`

```
require 'csv'
```

Luego podemos leer el archivo utilizando:

```
CSV.open('data.csv').readlines # => [{"Camilo", " 90", " 50 80", " 90"}, {"Francisca 100", " 40", " 100", " 100"}, {"Verónica 60", " 70", " 80", " 100"}, []]
```

Obviamente para seguir trabajando con los datos lo guardaremos en una variable.

```
data = CSV.open('data.csv').readlines
```

Removiendo las líneas vacías

En este caso nos quedó un arreglo vacío al final porque hay un salto de línea al final del archivo, pueden haber casos donde haya una o incluso mas de una.

Para removerla ocuparemos `.reject`

```
data.reject! {|x| x.empty? }
```

Seleccionando una fila

- Si queremos obtener la primera fila lo podemos hacer con `a[0]`.
- Podemos seleccionar la primera fila con `a[1]`.
- Podemos seleccionar la última fila con `a[-1]`.
- Otra forma de seleccionar la última fila es con `a[a.length - 1]`

Seleccionando un elemento de una fila

Para seleccionar solo un elemento, ya sea un nombre o una nota necesitamos dos índices, uno para la fila y el otro para el elemento dentro de la fila. Ejemplo:

Si queremos obtener el nombre de la primera fila utilizaremos `a[0][0]`.

Si queremos obtener el último nombre lo podemos hacer con `a[-1][0]` o podríamos hacer `a[a.length - 1][0]`

Limpiando y transformar datos

Por defect `CSV.open('data.csv').read` devuelve todos los datos como string, si bien podemos pedirle de forma automática que intente transformar automáticamente los números a tipo Integer con `CSV.open('data.csv', converters: :numeric).read` Esto nos quita una oportunidad de practicar lo aprendido con `.map`

Nota: Algunos métodos como por ejemplo `.open` pueden reciben como argumento un hash, esto lo estudiaremos en la próxima unidad.

Transformando datos

La primera pregunta es si al iterar ocuparemos `.times` o `.each`. Podemos ocupar cualquiera de las dos pero si queremos cambiar los datos dentro del mismo arreglo directamente utilizaremos `.times`, porque sin el índice tendríamos que crear un arreglo nuevo con todos los datos.

Modificando una columna

Por ejemplo supongamos que queremos aumentar 15 puntos extra a todos los alumnos en la segunda nota.

Algoritmo

- Cargamos la biblioteca de CSV.
- Leemos el archivo transformando los datos automáticamente a número.
- Contamos la cantidad de líneas del archivo.
- Por cada línea:
 - Sumamos 10 puntos a la segunda nota (tercer elemento)

```
require 'CSV'

data = CSV
  .open('examples/data.csv',
    converters: :numeric)
  .readlines
  .reject! {|x| x.empty? }

lines = data.length

lines.times do |i|
  data[i][2] += 15
end

print data
```

```
[["Camilo", 90, 65, 80, 90], ["Francisca", 100, 55, 100, 50], ["Verónica", 60, 85, 80, 100]]
```

Agregando una columna

Supongamos que los alumnos son tan simpáticos que queremos agregar una columna con nota

100

Algoritmo

El proceso es muy similar

- Cargamos la biblioteca de CSV.
- Leemos el archivo transformando los datos automáticamente a número.
- Contamos la cantidad de líneas del archivo.
- Por cada línea:
 - Agregamos una nota 100

```
require 'CSV'

data = CSV
  .open('examples/data.csv',
    converters: :numeric)
  .readlines
  .reject! {|x| x.empty? }
lines = data.length

lines.times do |i|
  data[i] << 100
end

print data
```

```
[["Camilo", 90, 50, 80, 90, 100], ["Francisca", 100, 40, 100, 50, 100], ["Verónica", 60, 70, 80, 100, 100]]
```

Guardando los resultados en un archivo

Es bastante probable que después de haber modificado el archivo queramos guardar los datos, para eso aprenderemos a abrir un archivo nuevo y guardar una línea

```
#require 'csv'
require 'CSV'
CSV.open("archivo.csv", "w") do |csv|
  csv << ["Juan", 80, 21, 55]
end
```

```
<#CSV io_type:File io_path:"myfile.csv" encoding:UTF-8 lineno:1 col_sep:"," row_sep:"\n" quote_char:"\"">
```

Para guardar todos los resultados simplemente tenemos que iterar los elementos y guardarlos por línea.

```
require 'CSV'

data = CSV
  .open('examples/data.csv',
    converters: :numeric)
  .readlines
  .reject! {|x| x.empty? }

lines = data.length

CSV.open("archivo.csv", "w") do |csv|
  lines.times do |i|
    csv << data[i]
  end
end
```

3

Comentarios importantes respecto a arreglos y archivos

Es posible que rara vez nos enfrentemos a una situación como la de este capítulo, hoy en día existen múltiples motores de bases de datos están fuertemente difundidas y son mucho mas flexibles y rápidas a la hora de guardar, actualizar y recuperar datos, sin embargo este tipo de ejercicios y la forma de enfrentarlos nos ayudan a resolver diversos tipos de situaciones y la lógica a la hora de trabajar con Bases de datos o APIs es muy similar, tendremos grandes cantidades de datos que tendremos que iterar para poder obtener los resultados que deseamos.

normalización.rb

La normalización vectorial, aunque tiene un nombre que puede asustar bastante, es algo sencillo. Consiste en que dado un arreglo con valores, se genera un nuevo arreglo donde todos los valores están entre cero y uno.

Para lograrlo cada valor se divide por el módulo, y el módulo se calcula como la raíz de la suma de cada uno de los elementos al cuadrado.

o sea: $\text{modulo}([1, 2, 3]) = \text{raiz}(1^2 + 2^2 + 3^2)$

Necesitamos:

- Un método para calcular el módulo de un arreglo
- Dividir cada uno de los elementos del arreglo por el módulo y guardarlos en un nuevo arreglo.

Podemos verificar el resultado dado que la suma de todos los valores al cuadrado debe ser uno.

Creando el método módulo, recibe un array y devuelve el módulo del arreglo

```
def modulo(array)
  n = array.count
  suma = 0
  n.times do |i|
    suma = suma + array[i]**2
  end
  Math.sqrt(suma)
end

modulo([1,2,3])
```

3.7416573867739413

Recorremos el arreglo y a cada elemento lo dividimos por el módulo

```
def modulo(array)
  n = array.count
  suma = 0
  n.times do |i|
    suma = suma + array[i]**2
  end
  Math.sqrt(suma)
end

def normalizar(array)
  n = array.count
  m = modulo(array)
  array_normalizado = []
  n.times do |i|
    array_normalizado.push array[i] / m
  end
  array_normalizado
end

normalizar([1,2,3])
```

[0.2672612419124244, 0.5345224838248488, 0.8017837257372732]

Verificamos el resultado

```
# Dijimos que la suma de cada uno de los números al
# cuadrado debe ser uno, esto lo podemos probar
# iterando el arreglo.

array = [0.2672612419124244, 0.5345224838248488, 0.8017837257372732]
suma = 0
n = array.count
n.times do |i|
  suma += array[i] ** 2
end

puts suma
```

1.0