

Контейнеры, блоки и итераторы

Андрей Васильев

2019

Массивы

Массивы создаются, обычно, с помощью литералов

```
array = [5.05, 'strawberry', 42]  
array[0] # => 5.05
```

Но можно и с помощью создания объекта

```
array = Array.new  
array[0] = 5.05  
array[0] # => 5.05
```

- Атрибут `#length` возвращает число элементов в массиве
- Атрибут `#size` тоже возвращает число элементов в массиве

Рекомендуется использовать `#size`.

Доступ к элементам

Для доступа к элементам массива используется метод #[]

- Элементы массива нумеруются с нуля
- Метод поддерживает отрицательные индексы, последний элемент имеет индекс -1
- Результат обращения к несуществующему элементу nil
- Данный метод можно переопределить в дочерних классах

```
a = [5, 6, 8, 11]
```

```
a[0] # => 5
```

```
a[-1] # => 11
```

```
a[-6] # => nil
```

Выборка части массива

Метод `#[]` может принимать два аргумента: начало и количество. Однако лучше использовать метод `slice`

- Начало выборки работает как выбор элемента
- Количество элементов может быть только положительным
- Результат такого метода - новый массив, содержащий элементы оригинального массива
- Если начало выборки вне массива, то результат работы `nil`

```
a = [3, 6, 9, 12, 15]
a.slice(1, 3) # => [6, 9, 12]
a.slice(3, 1) # => [12]
a.slice(-3, 2) # => [9, 12]
a.slice(-2, 3) # => [12, 15]
```

Использование диапазонов

В Ruby есть встроенный тип Range, диапазон. Для его создания существуют 2 литерала: `..`, включающий, и `...`, исключающий

```
(0..2).to_a # => [0, 1, 2]
(0...2).to_a # => [0, 1]
(2..0).to_a # => []
```

Их можно использовать для выборок в массивах

```
a = [3, 6, 9, 12, 15]
a.slice(0..2) # => [3, 6, 9]
a.slice(0...2) # => [3, 6]
a.slice(-3...-1) # => [9, 12, 15]
a.slice(-1..-3) # => []
```

Изменение значений массивов

Для записи значений в массив используется метод `#[]=`

```
a = [3, 6, 9, 12, 15] # => [3, 6, 9, 12, 15]
a[1] = 'lemon' # a => [3, "lemon", 9, 12, 15]
a[-2] = 'orange' # a => [3, "lemon", 9,
  # "orange", 15]
a[0] = [1, 2] # a => [[1, 2], "lemon", 9,
  # "orange", 15]
b = [3, 6] # => [3, 6]
b[5] = 15 # b => [3, 6, nil, nil, nil, 15]
```

При присвоении к несуществующему элементу пропуски заполняются `nil`

Изменение нескольких значений

Метод `[]` = также принимает набор

```
a = [3, 6, 9, 12, 1] #=> [3, 6, 9, 12, 1]
a[2, 2] = 'fly' # a => [3, 6, "fly", 1]
a[2, 0] = 'bot' # a => [3, 6, "bot", "fly", 1]
a[1, 1] = [2, 5] # a => [3, 2, 5, "bot", "fly"..
a[0..3] = [] # a => ["fly", 1]
a[5..6] = 98, 99 # a => ["fly", 1, nil, nil,
# nil, 98, 99]
```

Такой код невозможно воспринимать ни при каких обстоятельствах, поэтому использовать такую нотацию не рекомендуется

Краткий обзор методов массивов

Официальная документация по перечисляемым типам данным содержит множество методов для манипулирования списками

- `#push`, `#append` - добавить в конец массива
- `#pop` - извлечь элемент из конца массива
- `#shift` - извлечь элемент с начала массива
- `#unshift`, `#prepend` - добавить в начало массива
- `#first(n)` - получить `n` первых элементов
- `#last(n)` - получить `n` последних элементов
- `#drop(n)` - получить элементы массива с позиции `n`
- `#shuffle` - создать новый массив, перемешав элементы
- `#delete(obj)` - удалить все вхождения объекта `obj`

Хеши (ассоциативные массивы)

Хеши описывают соответствие между наборами из двух объектов. Первый объект называется ключом и должен быть уникальным среди всех ключей. Второй объект - значение

Хеши обычно создаются с помощью литералов

```
h = { 'dog' => 'canine', 'cat' => 'feline' }  
h.length # => 2  
h['dog'] # => "canine"  
h[12] = 'dodecine'  
h['cat'] = 99
```

Часто ключами хешей являются символы

```
h = { :dog => 'canine', :cat => 'feline' }  
h = { dog: 'canine', cat: 'feline' }
```

Подсчёт частоты встречи слов

Подсчитаем насколько часто встречаются слова в тексте. Для решения этой задачи необходимо:

- Разбить строку на слова
- Подсчитать частоту встречи слов
- Отсортировать по частоте встречи

```
def words_from_string(string)
    string.downcase.scan(/[\w']+/)
end
```

- `#downcase` преобразует строку к нижнему регистру
- `#scan` возвращает массив строк, совпадающих с переданным регулярным выражением

Подсчёт частоты с помощью Хешей

```
counts = {}  
for word in word_list  
  if counts.has_key?(word)  
    counts[word] += 1  
  else  
    counts[word] = 1  
  end  
end
```

```
counts = Hash.new(0)  
for word in word_list  
  counts[word] += 1  
end  
counts
```

Сортировка результатов

Пары ключ-значение сохраняют свой порядок в Хешах, что позволяет их сортировать

Для сортировки можно использовать метод `#sort_by`, который принимает блок и использует его значения для сортировки

```
sorted = counts.sort_by {|word, count| count}
```

Вывод 5 наиболее часто встречающихся слов

```
top_five = sorted.last(5)
for i in 0...5
  word = top_five[i][0]
  count = top_five[i][1]
  puts "#{word}: #{count}"
end
```

Полезные методы Хешей

- `#has_key?` - проверка на наличие ключа
- `#has_value?` - проверка на наличие значения
- `#last` - получение последних элементов
- `#sort_by` - сортировка элементов
- `#length` - количество элементов
- `#delete` - удалить пару ключ-значение

Отладка исходного кода

Предположим, что мы написали некоторые методы в файле и хотим проверить их работу. Для их отладки есть следующие способы:

- Написать небольшую программу, выполняющие методы
- Использовать `irb` для запуска методов
- Написать модульные тесты для их запуска

`irb` - интерактивный интерпретатор Ruby, позволяющий выполнять код построчно (REPL, Read-Eval-Print Loop)

Для подключения собственных файлов в `irb`:

- Перейдите в каталог с вашим файлом
- Запустите `irb:irb -I .`
- Подключите файл в вашу сессию `require ...`

Использование pry

pry - альтернативная реализация интерактивного интерпретатора Ruby

- Домашний сайт: <http://pryrepl.org/>
- Исходный код: <https://github.com/pry/pry>

Установка

pry не входит в поставку Ruby, его необходимо поставить отдельно:

```
$ gem install pry
```

Желательно также установить пакет pry-byebug

```
$ gem install pry-byebug
```

Отладка кода с помощью pry и pry-byebug

Комбинация pry и pry-byebug позволяет реализовать интерактивную отладку приложения. Для включения отладки достаточно в нужное место добавить следующий код:

```
require 'pry'  
binding.pry
```

Для навигации следует использовать следующие команды:

- `break` - управлять списком точек останова
- `step` - сделать 1 или несколько шагов выполнения
- `next` - сделать 1 или несколько шагов в данном фрейме
- `finish` - выполнить код до завершения фрейма
- `continue` - продолжить выполнение и выйти из pry

Документация на byebug:

<https://github.com/deivid-rodriguez/pry-byebug>

Блоки и итераторы

Привычный императивный стиль

```
for i in 0...5
  word = top_five[i][0]
  count = top_five[i][1]
  puts "#{word} : #{count}"
end
```

С применением итераторов становится легче к восприятию, следовательно содержит меньше ошибок

```
top_five.each do |word, count|
  puts "#{word} : #{count}"
end
```

Метод `#each` - итератор, метод вызывающий блок

Блоки

- Блок - набор выражений, находящийся между ключевыми словами `begin` и `end` или фигурными скобками
- Блок можно назвать «анонимным» методом
- Блоки могут иметь аргументы, которые указываются между вертикальными линиями
- Блок не исполняется в том месте, где описан в коде
- Блок ассоциируется с методом в момент её вызова
- Блок описывается после параметров метода

Область видимости переменных

- Выражения блока имеют доступ к переменным, объявленным вне блока
- Переменные, объявленные внутри блока, доступны только лишь в выражениях внутри блока
- Аргументы блока маскируют внешние переменные

```
sum = 0
[1, 2, 3, 4].each do |value|
  square = value * value
  sum += square
end
puts sum
```

Область видимости переменных

Переменные блока маскируют внешние переменные

```
value = "some shape"  
[1, 2].each {|value| puts value}  
puts value
```

Можно определить список локальных переменных

```
square = "some shape"  
sum = 0  
[1, 2, 3, 4].each do |value; square|  
  square = value * value # локальная переменная  
  sum += square  
end  
puts sum  
puts square
```

Но не надо так делать

Создание итераторов

- Итератор - метод, вызывающий ассоциированный блок
- Для вызова блока используется ключевое слово `yield`

```
def two_times
  yield
  yield
end
two_times { puts "Hello" }
```

Метод вызывает ассоциированный блок два раза

- Вызываемому блоку можно передавать параметры
- У блока можно получать возвращаемое значение

Вычисление последовательности Фибоначи

```
def fib_up_to(max)
  i1, i2 = 1, 1 # Паралельное присваивание
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end
fib_up_to(1000) {|f| print f, " "}
```

Блоку передаются параметр - следующее число последовательности Фибоначи. Блок будет вызываться столько раз, сколько необходимо для выполнения условия цикла

Возвращение значение из блока

```
class Array
  def find # Вариант реализации find
    each do |value|
      return value if yield(value)
    end
    nil
  end
end

[1, 3, 5, 7, 9].find {|v| v*v > 30} # => 7
```

- Если блок возвращает правдивое значение, тогда `#find` возвращает значение элемента
- Метод `#find` ничего не знает об условии, но эффективно обходит все элементы массива и предоставляет общую структуру для решения задачи

Работа ключевых слов в блоках

```
def example
  puts yield
  puts 'done'
  return 'example'
end
```

- `return` не имеет специального смысла в блоках, связан только с методом, где он вызывается.
- `break` - закончить выполнение блока, вернуть значение, переданное `break`, либо `nil`. *Ломает работу метода после вызова блока. Обычно не стоит использовать.*
- `next` - завершить текущую итерацию, значение итератора - это аргументы слову `next`.

Итератор #map

- Итератор #map (или #collect) позволяет создать новый массив на основе значений текущего массива
- Создание каждого элемента нового массива описывается в блоке, ассоциированном с данным методом

```
["H", "A", "L"].map {|x| x.succ}
```

Метод `String#succ` возвращает «преемника» для данной строки, начиная с правого символа строки

Потоки ввода-вывода

Классы ввода-вывода предоставляют итераторы для чтения по линиям или байтам

```
f = File.open('testfile')
f.each do |line|
  puts "The line is:  #{line}"
end
f.close
```

Итераторы могут быть использованы для решения множества задач

Учёт позиции в итераторе

- Итератор скрывает позицию элемента в массиве
- Для учёта позиции используйте метод `with_index`
- Блок будет получать ещё и порядковое значение

```
f = File.open("testfile")
f.each.with_index do |line, index|
  puts "Line #{index} is: #{line}"
end
f.close
```

Данная функциональность основана на работе нумераторов, которые разберём в следующих лекциях

Итераторы, использующие логические значения

Данные итераторы предполагают, что блок будет возвращать логические значения

- Метод `#any?` - есть ли хотя бы один элемент
- Метод `#all?` - все ли элементы
- Метод `#one?` - только один элемент
- Метод `#none?` - ни один элемент
- Метод `#find` - найти первый элемент
- Метод `#find_all` - найти все элементы
- Метод `#find_index` - найти номер элемента
- Метод `#delete_if` - удалить элемент из массива

Вычисление агрегированных значений

Часто необходимо вычислить значение, основываясь на всех элементах массива. Итератор `#reduce` (или `#inject`) позволяет решить данную задачу

```
[1, 3, 5, 7].reduce(0) {|sum, element| sum+element}
[1, 3, 5, 7].reduce(1) {|product, el| product*el}
```

Можно не указывать начальное значение, тогда первый элемент массива - начальное значение

```
[1, 3, 5, 7].reduce {|sum, element| sum+element}
[1, 3, 5, 7].reduce {|product, el| product*el}
```

Можно просто указать метод, который необходимо вызывать у элементов массива

```
[1, 3, 5, 7].reduce(:+) # => 16
[1, 3, 5, 7].reduce(:*) # => 105
```

Итераторы #tap и #yield_self

Итератор #tap

Данный итератор позволяет встроиться в цепочку по обработке данных, не изменяя её содержимое. Он передаёт `self` как параметр итератора, и также как его значение

```
(1..10).tap { |x| puts "original: #{x}" }  
      .to_a.tap { |x| puts "array: #{x}" }
```

Итератор #then, #yield_self

Данный итератор был добавлен в Ruby 2.5 с целью структурировать обработку данных. В версии 2.6 данный метод был переименован в `#then`

```
"my string".then { |s| s.upcase } #=> "MY STRING"  
3.next.then { |x| x**x }.to_s  #=> "256"
```