Ruby: Episode II

Хешове, функции, блокове, ламбди

Хешове

Още: речници, хеш таблици, map

Класът е `Hash`, но има и литерал

Дефинират се така: `{1 => :one, 2 => :two}`

Ключовете трябва да могат да се хешират: числа, низове, символи, списъци

По възможност, ключовете трябва да са immutable

От Ruby 1.9 насам редът на ключовете се запазва

Хешове и индексиране

```
numbers = {:one => :eins, :two => :zwei}

numbers[:one]      # :eins
numbers[:three]    # nil

numbers[:three] = :drei

numbers[:three]                          # :drei
numbers.fetch(:four, :keine_ahnung) # :keine_ahnung
numbers.fetch(:four)                     # KeyError
```

Итериране върху хешове

```
numbers = {:one => :eins, :two => :zwei}
numbers.keys    # [:one, :two]
numbers.values  # [:eins, :zwei]

numbers.each { |key| puts key }
numbers.each { |key, value| puts key, value }
```

Методи на хеша

```
numbers = {1 => 2, 3 => 4}

numbers.has_key?(:three) # false
numbers.size             # 2
numbers.invert           # {1 => 2, 3 => 4}
numbers.merge({5 => 6})  # {1 => 2, 3 => 4, 5 => 6}
numbers.to_a             # [[1,2], [3,4]]
Hash[1, 2, 3, 4]         # {1 => 2, 3 => 4}
```

Още може видите в документацията.

Алтернативен синтаксис на хешове

Долните два реда произвеждат еднакви хешове. Второто е 1.9 синтаксис:

```
{:one => 1, :two => 2}

{one: 1, two: 2}
```

Методи

Дефинирането става с ключовата дума `def`. Резултатът от функцията е последният оценен израз, ако няма `return` някъде.

```
def factorial(n)
  unless n == 1
    factorial(n - 1) * n
  else
    1
  end
End
```

В Ruby няма tail recursion оптимизация. Този код яде стек.

За методите и хората

В ruby няма такова нещо като "глобална функция"

`def` винаги дефинира метод в някакъв клас

Ако `def` не е в дефиниция на клас, отива като `private` метод на `Object`

`puts` е пример за нещо такова, както и методите, които дефинирате в irb

Дефинирането на методи в `Object` е удачно само за кратки скриптове

Отвъд тях, дефинирането на методи в `Object` е ужасно лош стил

Методи в съществуващи класове

За да добавите метод в съществуващ клас, например Array, просто "отваряте" класа и дефинирате метода:

```
class Array
  def fourty_second
    self[41]
  end
end

list      = []
list[41] = 'The Universe'

list.fourty_second # “The Universe”
```

Методи: `return`

Можете да излезете от функция с `return`:

```
def includes?(array, element)
  array.each do |item|
    return true if item == element
  end
  false
end
```

Разбира се, такава функция е излишна.
Може да ползвате `array.include?(element)`.

Методи: стойности по подразбиране

Параметрите в Ruby могат да имат стойности по подразбиране:

```
def order(drink, size = 'large')
  puts "A #{size} #{drink}, please!"
end

order 'tea'
order 'coffee', 'small'
```

Стойностите по подразбиране могат да бъдат всякакъв израз
"Всякакъв израз" включва и извикване на друга функция
Оценяват се на всяко извикване
Сложни изрази за аргументи по подразбиране са лош стил

Методи с променлив брой аргументи

Методите в ruby могат да вземат променлив брой аргументи.

Параметърът се означава със * и при извикване на функцията съдържа списък от аргументите.

```
def say_hi(name, *drinks)
  puts "Hi, I am #{name} and I enjoy: #{drinks.join(',')}"
end

say_hi 'Stefan', 'coffee', 'tea', 'water'
```

Методи с променлив брой аргументи

Параметърът за променлив брой аргументи може да е на всяка позиция в дефиницията:

```
def something(*a, b, c)
end

def something(a, *b, c)
end
```

Очевидно, може да има само един такъв параметър във функция.

Когато последният параметър е хеш...

...може да изтървете фигурните скоби около него.
Долните редове правят едно и също:

```
def order(drink, preferences)
end

order 'Latte', {:size => 'grande', :syrup => 'hazelnut'}
order 'Latte', :size => 'grande', :syrup => 'hazelnut'
order 'Latte', size: 'grande', syrup: 'hazelnut'
```

Така Ruby симулира извикане на функция с наименовани аргументи.

Методи и хешове (отново)

Често се среща следният код:

```
def order(drink, preferences = {})
end

order 'Latte'
order 'Latte', size: 'grande', syrup: 'hazelnut'
```

Така preferences е незадължителен и няма нужда да го подавате, ако нямате предпочитания.

Методи предикати

Името на метод може да завършва на ?. Това се ползва, за методи, които връщат лъжа или истина (предикати):

```
def even?(n)
  n % 2 == 0
end

even? 2
even? 3
```

Това е, разбира се, само конвенция.

Методи с две версии

Името на метод може да завършва на !.
Това се ползва, когато методът има две версии с различно поведение:

```
numbers = [4, 1, 3, 2, 5, 0]

numbers.sort   # връща нов списък
numbers.sort!  # променя списъка на място
```

В случая, "по-опасният" метод завършва на удивителна.

Анонимни функции a.k.a. ламбди

Анонимни функции в Ruby се дефинират с lambda. Имат три начина на извикване:

```
pow = lambda { |a, b| a ** b }

pow.call 2, 3
pow[2, 3]
pow.(2, 3)
```

За нещастие, не може да извиквате така: `double(2)`. Това е несъвместимо с изтърваването на скобите при извикването на метод.

Ламбди (2)

Може и така:

```
double = lambda do |x|
  x * 2
end
```

Важи стандартната конвенция за { } и do/end.

Ламбди (3)

От 1.9 има по-симпатичен синтаксис за ламбди:

```
say_hi = lambda { puts 'Hi there!' }
double = lambda { |x| x * 2 }
divide = lambda { |a, b| a / b }

say_hi = -> { puts 'Hi there' }
double = -> (x) { x * 2 }
divide = -> (a, b) { a / b }
```

Блокове

Всеки метод може да приеме допълнителен аргумент, който е "анонимна функция". Може да го извикате от метода с `yield`:

```
def sequence(first, last, step)
  current = first
  while current < last
    yield current
    current += step
  end
end

sequence(1, 10, 2) { |n| puts n }
# Извежда 1, 3, 5, 7, 9
```

Блокове (1)

`yield` се оценява до стойността на блока:

```
def calculate
  result = yield(2)
  puts "The result for 2 is #{result}"
end

calculate { |x| x ** 2 }
# The result for 2 is 4
```

Блокове - пример

Реализация на `filter`:

```
def filter(array)
  result = []
  array.each do |item|
    result << item if yield item
  end
  result
end

filter([1, 2, 3, 4, 5]) { |n| n.odd? }
```

Блокове: #block_given?

`block_given?` ще ви каже дали методът е извикан с блок:

```
def i_can_haz_block
  if block_given?
    puts 'yes'
  else
    puts 'no'
  end
end

i_can_haz_block                    # no
i_can_haz_block { 'something' }    # yes
```

Блокове и ламбди

Ако имате ламбда, която искате да подадете като блок, може да ползвате &:

```
is_odd = lambda { |n| n.odd? }

filter([1, 2, 3, 4, 5], &is_odd)
filter([1, 2, 3, 4, 5]) { |n| n.odd? }
```

Горните са (почти) еквиваленти. Има малка разлика в някои други случаи.

Блокове в сигнатурата на метода

Ако искате да вземете блока като обект, има начин:

```
def invoke_with(*args, &block)
  block.(*args)
end
invoke_with(1, 2) { |a, b| puts a + b }
```

Може и така:

```
def make_block(&block)
  block
end
doubler = make_block { |n| n * 2 }
doubler.(2) # =>
```

Proc.new

В Ruby има два вида анонимни функции. Другият е Proc.

```
double = Proc.new { |x| x * 2 }

double.call(2)
double[2]
double.(2)
```

Дотук е същото, но има разлики при извикване.

Разлики между `Proc.new` и `lambda`

f =	Proc.new { |x,y| p x, y}	lambda { |x, y| p x, y }
f.call(1)	1 nil	ArgumentError
f.call(1, 2)	1 2	1 2
f.call(1, 2, 3)	1 2	ArgumentError
f.call([1, 2])	1 2	ArgumentError
f.call(*[1, 2])	1 2	1 2

`yield` ползва семантиката на `Proc.new` (т.е. `yield`-вате `Proc`-ове)
Извикването на метод ползва семантиката на `lambda`
Има още няколко малки разлики. Например `return` при ламбдите излиза само от ламбдата.

Няколко примера за блокове

```
numbers   = [-9, -4, -1, 0, 1, 4, 9]

positive = numbers.select { |n| n >= 0 }
even     = numbers.reject { |n| n.odd? }
squares  = numbers.collect { |n| n ** 2 }

roots = numbers.select { |n| n > 0 }.collect do |n|
  n ** 0.5
end
```

Още няколко примера

`#select` и `#collect` имат синоними `#find_all` и `#map`:

```
numbers   = [-9, -4, -1, 0, 1, 4, 9]

squares  = numbers.map { |n| n ** 2 }
positive = numbers.find_all { |n| n >= 0 }
```

#inject

`#inject` свежда списък до единична стойност с някаква операция:

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

numbers.inject(0) { |a, b| a + b }
numbers.inject(1) { |a, b| a * b }

numbers.inject { |a, b| a + b }
numbers.inject { |a, b| "#{a}, #{b}" }
```

`#inject`: примерна имплементация

```
def inject(array, initial = nil)
  remaining = array.dup
  buffer    = initial || remaining.shift

  until remaining.empty?
    buffer = yield buffer, remaining.shift
  end

  buffer
end

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

Въпроси?

Където вие питате, а аз пиша кода в `pry` / `irb`...