Lalla

Zeste de savoir

Les blocs en Ruby

12 août 2019

Table des matières

1. Les fonctions d'ordre supérieur 1
2. Les Proc de Ruby 2
3. Les blocs 3

Vous apprenez le Ruby, parce que c'est un très beau langage et partout vous entendez parler des *blocs*. On vous vend ce concept comme la *killer-feature* de Ruby et en même temps comme le concept le plus compliqué que l'informatique ait jamais connu. Peut-être avez-vous lu un cours sur les blocs et n'y avez-vous pas compris ce qu'ils représentent.

? Et si maîtriser les blocs était facile ?

C'est ce que je vais tenter de vous prouver en 3 étapes ! 3 cours sur le fondement même des blocs de Ruby et vous serez en mesure de créer vos propres pseudo-structures de contrôles pour Ruby !

! Ce cours exige un minimum de connaissance en Ruby et en programmation en général. Savoir utiliser les types de données de base et les fonctions, connaître les bases de la POO (ce que sont les objets, comment s'en servir) devrait être un minimum

1. Les fonctions d'ordre supérieur

Petite mise en situation : imaginons une quelconque fonction `filter` dont le but est de filtrer les éléments d'un tableau selon une condition bien précise et de ne garder que ceux qui la respectent. En voici une implémentation assez naïve :

```ruby
def filter(ary)
  max = ary.length - 1
  new_ary = Array.new

  # Tous les éléments strictement négatifs sont supprimés
  for i in (0..max)
    new_ary << ary[i] if ary[i] >= 0
  end

  return new_ary
```

```
11 end
```

Nous créons un nouveau tableau qui contiendra les éléments respectant notre condition et le renvoyons à la fin de la fonction. Nous avons cependant un petit problème : notre fonction ne prend en compte qu'une unique condition : `ary[i] >= 0`. Il serait agréable de pouvoir écrire quelque chose comme :

```
1 filter(ary, condition) # où ary est un tableau préexistant et
      condition la condition à respecter
```

Il faudrait donc envoyer une fonction à filter en guise de condition. On appelle **fonction d'ordre supérieur** (ou simplement **une fonctionnelle**) toute fonction qui prend en paramètre une fonction ou en renvoie une comme valeur de retour. On aurait alors un code ressemblant au suivant :

```
1  def filter(ary, cond)
2    max = ary.length - 1
3    new_ary = Array.new
4
5    for i in (0..max)
6      new_ary << ary[i] if cond(ary[i]) # Petit hic ici
7    end
8
9    new_ary
10 end
```

Malheureusement, en Ruby, il n'est pas possible de passer une fonction toute nue en paramètre à une autre. Comme vous le savez, les parenthèses sont optionnelles en Ruby et n'écrire que le nom d'une fonction cause systématiquement son appel. Il nous faut donc ruser un peu.

2. Les Proc de Ruby

Il se trouve que notre fonction `cond` est appelée avant d'entrer dans `filter`, nous exécutons donc `filter` avec la sortie de `cond` et non `cond` elle-même. Ce n'est pas ce qu'on veut ! On veut traiter `cond` comme un objet à part entière ! En Ruby, il existe une façon très simple d'encapsuler une fonction dans un objet, on utilise alors la classe Proc. Petite présentation :

```
1 def hello(name)
2   puts "Hello #{name}"
3 end
4
```

```ruby
proc = method(:hello).to_proc # On récupère la méthode et on la convertit en Proc
```

Grâce à ce petit objet, le code suivant fonctionnera :

```ruby
def apply(p)
  p.call "Lalla" # Nécessité de passer par la méthode call
end

apply(proc) # "Hello Lalla"
```

Nous avons ici montré que les fonctions d'ordre supérieur sont une réalité en Ruby. Il suffit pour l'instant d'avoir une fonction pré-existante et de créer une Proc correspondante. Vous noterez que cette construction est lourde et nécessite d'avoir une fonction écrite en dur à côté pour fonctionner.

Il existe cependant un moyen de créer ses fonctions à la volée : `Proc.new`.

```ruby
# On admet pour l'instant que la construction { ... } est l'écriture d'une nouvelle fonction
# Les paramètres sont notés entre | |
hello = Proc.new {|name| puts "Hello #{name}" }

apply(hello)
```

Nous pouvons maintenant créer nos fonctions à la volée afin de les donner à manger à nos fonctionnelles. Remarquez cependant que dans le corps de ces dernières, on est toujours obligés d'écrire `p.call` (où p est une Proc).

Vous savez donc utiliser des Proc pour profiter pleinement des fonctionnelles, deuxième étape terminée ! On peut à présent parler de notre amour de toujours : les blocs !

3. Les blocs

J'ai une terrible nouvelle à vous annoncer : on vient d'en utiliser un ! En fait, la contruction `{ ... }` est un bloc. Les blocs de Ruby sont ce qu'on appelle une fermeture . Il va s'agit de créer une fonction là même où vous écriviez votre algorithme. Cette fonction aura la particularité de capturer les variables existantes autour d'elle, exemple :

```ruby
i = 5
# i n'est pas déclaré dans p, pourtant il existe quand même,
# car existant dans l'environnement dans lequel p est déclaré.
```

```ruby
# p a capturé i
p = Proc.new { puts i.to_s }
p.call # affiche 5
```

Ainsi, vous l'aurez compris, lorsque vous écrivez un bloc derrière `Array#each` ou `Fixnum#times` vous créez en fait une fermeture. Les deux codes suivants sont quasi-synonymes :

```ruby
5.times {|i| puts i.to_s }

p = Proc.new {|i| puts i.to_s }
5.times(&p)
```

Vous avez certainement noté la présence d'un &. En fait, donner une Proc à une fonction et lui donner un bloc est différent. Ce n'est pas le même objet. La différence est faite essentiellement lors de la création d'une fonctionnelle. C'est lors de l'écriture de la fonctionnelle que le choix entre Proc et bloc est fait. Notez que si votre fonction a vraiment vocation à manipuler une Proc, demandez simplement une Proc, alors que si vous voulez une fonction afin d'appliquer un traitement sur des données (comme vu au début de ce cours avec map et filter), on préférera un bloc (c'est d'ailleurs toujours le cas dans la bibliothèque standard de Ruby).

Il existe deux façons de demander un bloc : explicitement et implicitement, exemple :

```ruby
def implicite(value)
  # block_given? sert à savoir si un bloc a été donné (bonne pratique)
  yield value if block_given? # appelle le bloc avec value en paramètre
end

def explicite(value, &block)
  block.call value # comme ça
  yield value # ou comme ça, mais pas les deux en même temps, ça sert à rien
end
```

Pour appeler le bloc à partir de sa fonctionnelle, on utilise le mot-clef `yield`. Il est tout à fait possible de l'appeler par son nom, dans le cas d'un bloc "explicite", même s'il est bien plus courant d'employer le mot-clef `yield` qui est créé spécialement pour ça.

Il est possible de rendre le bloc **facultatif** avec une bonne utilisation de `block_given?` alors qu'autrement, le bloc est **obligatoire**. Afin de demander explicitement un bloc, il faut précéder son nom de &. Ruby sait alors que vous attendez un bloc et non une Proc. Le & dans `5.times &p` sert à dire "envoie cette Proc sous forme de bloc".

L'utilisation de l'opérateur préfixe `&` sert en fait d'opérateur de conversion entre Proc et blocs. Cette conversion fonctionne dans les deux sens :

```ruby
def f(&block)
  # Le fait d'écrire explicitement &block nous crée un objet Proc à partir du bloc obtenu
  block.class
end

f do
  puts "Bonjour"
end

# Soit p une Proc quelconque
# On convertit la Proc p en bloc grâce à l'opérateur &
f &p

# Chacun de ces exemples renverra "Proc", puisque le bloc est converti en Proc par la fonction f.
```

Vous pouvez alors appeler vos jolies fonctionnelles ainsi :

```ruby
implicite 5 # bloc facultatif
implicite 5 do |i| # syntaxe avec do
  puts i.to_s
end

p = Proc.new {|i| puts i.to_s }
implicite 5, &p

explicite 5 {|i| puts i.to_s }
```

Il est à noter qu'il est d'usage d'utiliser les accolades quand votre bloc fait une ligne (comme ici) ou la notation `do ... end` quand le bloc est plus long.

Vous avez à présent maîtrisé les blocs en 3 étapes comme promis. Facile, n'est-ce pas ?

Comme vous l'avez vu, les blocs ne sortent pas de nulle part ! Ce ne sont finalement que du sucre syntaxique pour créer des fonctions à la volée et les donner à manger à d'autres fonctions. Je vous encourage désormais à pratiquer et continuer à vous documenter sur ce magnifique langage.

Je vous dis à bientôt pour de nouvelles aventures...