Karnaj

Zeste de savoir

Méthodes et paramètres en Ruby

10 janvier 2023

Table des matières

Introduction

Lorsque nous programmons en Ruby, nous écrivons souvent des méthodes. Les méthodes sont pour ainsi dire au centre de la programmation. Ainsi, il est important de savoir écrire de bonnes méthodes. L'écriture d'une bonne méthode passe bien sûr par l'écriture d'un bon code, mais c'est un autre aspect que nous allons traiter dans ce tutoriel: les paramètres des méthodes. En Ruby, les paramètres sont assez versatiles et permettent de faire plusieurs choses assez utiles.

Par exemple, il peut nous arriver d'oublier l'ordre des paramètres d'une fonction (faut-il d'abord donner ce paramètre ou celui-là). Ruby nous permet de ne pas nous soucier de l'ordre des paramètres en utilisant des paramètres dont le nom doit être précisé à l'appel de la méthode. De même, Ruby nous permet d'avoir des méthodes avec un nombre non déterminé d'arguments (on les récupère alors dans une liste ou dans une table de hachage).

Dans ce tutoriel, nous verrons comment faire ces deux choses avec Ruby, et nous verrons comment Ruby a évolué dans sa gestion des paramètres.

Prérequis

Prérequis
Ce tutoriel nécessite bien sûr des bases en Ruby . Il faut notamment savoir ce qu'est une méthode et avoir déjà manipulé des tableaux et des tables de hachage.
Objectifs
Faire le point sur la création et l'appel de méthodes en Ruby et sur les subtilités qui y sont liées.

1. Différents types de paramètres

Introduction

Pour commencer, nous allons voir en profondeur les deux types de paramètres des méthodes de Ruby. Comment et quand les utiliser? Quels problèmes peuvent se poser durant leur utilisation? Nous allons répondre à ces questions ici.

1.1. Rappels

Pour démarrer sur de bonnes bases, faisons quelques petits rappels et posons les bases de notre travail.

1.1.1. Arguments ou paramètres ?

Pour poursuivre les rappels, il nous faut faire un point sur les notions d'arguments et de paramètres. Généralement, les deux mots sont employés de manière indifférenciée, mais nous allons cependant apporter une petite nuance à cela. Dans un tutoriel qui traite en profondeur des méthodes et des arguments, cette nuance semble indispensable et nous permettra de toujours bien savoir ce dont nous parlons, sans aucune ambiguïté.

Les paramètres sont les noms que nous utilisons lorsque nous définissons une méthode et les arguments sont les valeurs passées à la méthode lors de son appel.

```ruby
def introduce(first_name, last_name, age)
  "#{first_name} #{last_name} a #{age} ans."
end

introduce('Mickey', 'Mouse', 92)
introduce('Balthazar', 'Picsou', '73')
```

Dans cet exemple, la méthode `introduce` a trois paramètres, `first_name`, `last_name` et `age` et nous l'appelons deux fois avec des arguments différents. La première fois avec les arguments `'Mickey'`, `'Mouse'` et `92`, et la seconde fois avec les arguments `'Balthazar'`, `'Picsou'` et `'73'`.

Lorsque nous appelons une méthode, nous lui donnons des arguments, et nous récupérons les valeurs de ces arguments dans les paramètres de ladite méthode. Par exemple, avec notre premier

appel à `introduce`, nous récupérons la valeur `'Mickey'` dans la variable `first_name`, `Mouse` dans la variable `last_name` et `92` dans la variable `age`. Les paramètres sont donc les variables dans lesquelles on récupère les arguments donnés à la méthode lors d'un appel.

D'autres noms

Dans certains cas, nous entendrons également parler de «paramètre formel» pour désigner les paramètres et de «paramètre effectif» pour les arguments.
Les gens qui font un peu de théorie (par exemple de la logique, du lambda-calcul et pleins d'autres gros mots) ont la notion de variable liée qui correspond à la notion de paramètre d'une fonction. En effet, lorsque l'on appelle une méthode `f` de paramètre `x` avec un argument `v`, on peut se dire que l'on lie la valeur `v` à la variable `x` dans la méthode (la variable `x` a désormais la valeur `v`).

Finalement, parler des paramètres d'une méthode a du sens là où parler des arguments d'une méthode n'en a pas. En effet, une méthode n'a pas, à proprement parler, d'arguments. C'est seulement lorsque nous appelons la méthode que le mot argument a du sens: nous appelons la méthode avec des arguments.

1.1.2. Les paramètres positionnels

Les paramètres positionnels correspondent aux paramètres classiques que l'on connaît. Lorsqu'on appelle une méthode, Ruby sait dans quel paramètre il faut récupérer chaque argument grâce à leur position. Reprenons l'exemple précédent.

```ruby
def introduce(first_name, last_name, age)
  "#{first_name} #{last_name} a #{age} ans."
end

introduce('Mickey', 'Mouse', 92)
```

Ici,`introduce` a trois paramètres positionnels. Lorsque la méthode est appelée, Ruby sait alors que le premier argument doit être récupéré dans le paramètre `first_name`, le deuxième dans le paramètre `last_name`, et le dernier dans le paramètre `age`. C'est bien l'ordre dans lequel les arguments sont passés, leur *position* qui indique à Ruby dans quel paramètre les récupérer.

Si nous appelons la méthode avec les mêmes arguments, mais dans un autre ordre, le résultat sera donc différent.

```ruby
introduce('Mouse', 92, 'Mickey')
# => 'Mouse 92 a Mickey ans.
```

1.1.2.1. Valeurs par défaut

Ruby nous permet de donner une valeur par défaut à certains paramètres. Lorsque nous appelons la méthode, nous pouvons alors ne pas donner d'argument pour ce paramètre. Ici, nous allons avoir un paramètre pour choisir si le nom doit être affiché avant ou après le prénom.

```ruby
def introduce(first_name, last_name, age, reverse = false)
  return "#{last_name} #{first_name} a #{age} ans." if reverse
  "#{first_name} #{last_name} a #{age} ans."
end

introduce('Mickey', 'Mouse', 92)
introduce('Mickey', 'Mouse', 92, true)
```

Dans le premier appel, nous n'avons pas fourni de valeur pour le dernier paramètre `reverse`, sa valeur était donc celle par défaut, à savoir `false`. Dans le second appel, nous avons fourni la valeur `true`, ce qui nous a permis d'obtenir une chaîne formatée différemment.

Dans notre exemple, nous avons une valeur par défaut, mais nous pourrions en mettre plusieurs. Rajoutons par exemple un argument `location` avec comme valeur par défaut `'Mickeyville'` (si nous supposons écrire un programme sur l'univers de Mickey, il est raisonnable d'avoir Mickeyville comme ville par défaut.

```ruby
def introduce(first_name, last_name, age, location =
    'Mickeyville', reverse = false)
  return "#{last_name} #{first_name} a #{age} ans et habite à #{lo
      cation}." if
      reverse
  "#{first_name} #{last_name} a #{age} ans et habite à #{location}
      ."
end

introduce('Mickey', 'Mouse', 92)
introduce('Donald', 'Duck', 86, 'Donaldville')
introduce('Donald', 'Duck', 86, 'Donaldville', true)
```

Cette fois-ci, nous avons trois exemples à examiner. Dans le premier cas, nous ne fournissons de valeur ni pour `location`, ni pour `reverse`, les valeurs par défaut sont prises. Dans le deuxième cas, nous fournissons une valeur, `'Donaldville'` en quatrième argument. Cette valeur est récupérée dans le paramètre `location` et `reverse` garde la valeur par défaut. Et finalement, dans le troisième cas, nous fournissons `'Donaldville'` et `true` en quatrième et cinquième arguments, qu'on récupérera respectivement dans les quatrième et cinquième paramètre, `location` et `reverse`.

Nombre d'arguments

Bien sûr, le nombre d'arguments donnés à la méthode doit être supérieur ou égal au nombre de paramètres sans valeur par défaut de cette méthode, et doit être inférieur ou égal au nombre de paramètres total de cette méthode. Ainsi, `introduce` doit être appelée avec un nombre d'arguments compris entre `3` et `5`.

Nous pouvons faire un peu plus de choses avec les valeurs par défaut des paramètres positionnels, ce que nous verrons plus loin.

1.2. Un problème à régler

1.2.1. À chaque problème...

Lorsque nous avons plusieurs valeurs par défaut comme dans le dernier code que nous avons considéré, nous ne pouvons pas faire tous les appels que nous voudrions possiblement faire.

```
1 def introduce(first_name, last_name, age, location =
      'Mickeyville', reverse = false)
2   return "#{last_name} #{first_name} a #{age} ans et habite à #{lo
        cation}." if
        reverse
3   "#{first_name} #{last_name} a #{age} ans et habite à #{location}
        ."
4 end
```

Nous pouvons faire trois types d'appels avec cette méthode.

1. Ne pas donner de valeur pour `location` et `reverse`.
2. Donner de valeur pour `location` et ne pas en donner pour `reverse`.
3. Donner une valeur pour `location` et donner une valeur pour `reverse`.

Mais il y a un type d'appel que nous ne pouvons pas faire: nous ne pouvons pas donner de valeur pour `reverse` sans donner de valeur pour `location`! En effet, ce sont les positions des arguments qui indiquent dans quel paramètre ils sont récupérés et appeler `introduce` avec quatre arguments nous ferait donc récupérer le quatrième argument dans `location`.

Dans certains langages, on contourne ce problème en indiquant lors de l'appel de la méthode le nom du paramètre que l'on veut associer à la valeur, de la manière suivante.

```
1 # On précise que l'argument `true` est pour le paramètre `reverse`.
2 introduce('Mickey', 'Mouse', 92, reverse = true)
```

En Ruby, cela ne fonctionne pas. En fait, lorsque Ruby lit ce code, il comprend que nous voulons faire l'affectation `reverse = true` et donner le résultat de cette affectation à la méthode. Et il se trouve que le résultat d'une affectation est le résultat de l'expression que l'on affecte (donc dans notre cas, `true`). Finalement, ce code donne à `reverse` la valeur `true` et appelle `introduce` avec `true` comme quatrième argument.

```ruby
# reverse n'est pas encore définie
introduce('Mickey', 'Mouse', 92, reverse = true)
# => 'Mickey Mouse a 92 ans et habite à true.'
reverse
# => true
```

C'est une mécanique de Ruby intéressante qui peut être utilisée pour définir ou modifier des variables lors de l'appel d'une méthode.

Pas d'abus

Il ne faut cependant pas en abuser. Notre but est d'avoir un code clair et lisible.

```ruby
def f(x, y, z, t)
  0
end

x = 0
f(x = 3, x += 2, x = 4, x -= 1)
# Que vaut x ?
```

Le code précédent n'est certainement pas le plus lisible qui soit. Nous pouvons trouver la valeur de `x` car les arguments sont évalués dans l'ordre. Ainsi, on fait d'abord `x = 3`, puis `x += 2`, `x = 4`, et enfin `x -= 1`. Donc `x` vaut `3`.

1.2.2. Sa solution !

La solution à ce problème est un deuxième type de paramètres de Ruby: les paramètres nommés. Il s'agit de paramètres dont nous devons préciser le nom à l'appel de la méthode. Ruby utilisera alors ce nom, plutôt que la position de l'argument, pour déterminer dans quel paramètre cet argument doit être récupéré. Les paramètres nommés s'écrivent avec une syntaxe qui n'est pas sans rappeler celle des symboles.

```ruby
def introduce(first_name, last_name, age, location: 'Mickeyville',
    reverse: false)
  return "#{last_name} #{first_name} a #{age} ans et habite à #{lo
      cation}." if
      reverse
  "#{first_name} #{last_name} a #{age} ans et habite à #{location}
      ."
end

introduce('Mickey', 'Mouse', 92, reverse: true)
introduce('Donald', 'Duck', 86, reverse: true, location:
    'Donaldville')
```

L'ordre des paramètres nommés n'a pas d'importance. Ainsi, nous pouvons donner l'argument pour `reverse` avant celui pour `location` (comme nous l'avons fait dans notre dernier exemple). Notons de plus que nous ne sommes pas obligés de donner une valeur par défaut à notre argument. Nous pouvons transformer notre méthode pour n'utiliser que des paramètres nommés.

```ruby
def introduce(first_name:, last_name:, age:, location:
    'Mickeyville', reverse: true)
  return "#{last_name} #{first_name} a #{age} ans et habite à #{lo
      cation}." if
      reverse
  "#{first_name} #{last_name} a #{age} ans et habite à #{location}
      ."
end

introduce(first_name: 'Donald', last_name: 'Duck', location:
    'Donaldville', age: 86)
```

Nous pouvons alors appeler `introduce` sans nous soucier de l'ordre des arguments que l'on donne. Cependant, puisque cet ordre n'a aucune importance, il est obligatoire de préciser le nom du paramètre lors de l'appel de la méthode.

```ruby
introduce('Donald', 'Duck', 86, true, 'Donaldville')
# => ArgumentError
```

Nous pourrions nous attendre à ce que Ruby décide d'utiliser l'ordre des arguments pour décider des paramètres dans lesquels ils sont récupérés, mais ce n'est pas le cas, nous obtenons une erreur `ArgumentError`. Les paramètres nommés sont donc un peu plus lourd à utiliser et nous verrons comment les utiliser à bon escient.

1.2.2.1. Mélanger les deux types de paramètres

Notre premier exemple de méthode avec des paramètres nommés avait également des paramètres positionnels. Si c'est autorisé par le langage, il y a une contrainte à respecter: les paramètres positionnels doivent apparaître avant les paramètres nommés. La même contrainte est à respecter pour les arguments lors de l'appel de la méthode.

Si une définition de méthode ne respecte pas cela, nous obtiendrons une erreur de syntaxe `SyntaxError`, et si c'est lors d'un appel de méthode que nous ne respectons pas la contrainte, nous aurons une erreur `ArgumentError`.

```
1  def f(x: 2, y)
2    x + y
3  end
4  # => SyntaxError
5
6  def g(y, x: 2)
7    x + y
8  end
9
10 g(x: 3, 2) # => ArgumentError
```

Cette contrainte n'est pas compliquée à respecter et se comprend plutôt bien: d'abord on regarde les choses qui sont liées à une position, et ensuite on regarde les choses qui peuvent apparaître dans n'importe quel ordre.

1.3. Les paramètres nommés

Les paramètres nommés nous ont permis de résoudre notre problème précédent. Mais quand faut-il les utiliser? Était-ce la bonne solution pour notre problème? Pour répondre à cette question, il faut se rendre compte que notre but, en écrivant du code, est d'obtenir un code clair et lisible. Ce constat nous donne plusieurs cas d'usage des paramètres nommés.

1.3.1. Cas d'usage

1.3.1.1. Parfait pour des options

Le premier d'entre eux correspond à la manière dont nous l'utilisons pour `reverse`. Dans notre cas, notre méthode a deux modes de fonctionnement en fonction de la valeur de `reverse`. C'est un usage très répandu, car, sans le nom du paramètre, il peut être compliqué de savoir à quoi correspond l'argument.

```ruby
introduce('Mickey', 'Mouse', 92, true)
introduce('Mickey', 'Mouse', 92, reverse: true)
```

Dans le premier cas, l'appel de méthode ne permet pas vraiment de savoir à quoi correspond le dernier argument. Dans le second cas, c'est déjà beaucoup plus clair.

Bonnes pratiques

En fait, ce guide de bonnes pratiques recommande d'utiliser des paramètres nommés lorsqu'un booléen est attendu!

Dans la librairie standard, nous avons quelques exemples de méthode avec de tels paramètres. La méthode `round`, par exemple, permet d'arrondir les flottants. Elle a un paramètre nommé, `half`, qui indique comment arrondir si la partie décimale du flottant est `0.5`. Avec `:up` on arrondit au supérieur (c'est la valeur par défaut), avec `:down` à l'inférieur, et avec `:even` on arrondit à l'entier pair.

```ruby
2.5.round               # => 3
2.5.round(half: :up)    # => 3
2.5.round(half: :down) # => 2
2.5.round(half: :even) # => 2
3.5.round(half: :even) # => 4
```

1.3.1.2. Et pour ne pas se soucier de l'ordre

L'utilisation des paramètres nommés pour des options est assez fréquent. Tellement qu'il est le plus souvent croisé avec des valeurs par défaut. Pourtant, comme nous l'avons vu, les valeurs par défaut ne sont pas obligatoires avec des paramètres nommés. Nous pouvons juste les utiliser pour ne pas nous soucier de l'ordre des paramètres. Ainsi, nous avons précédemment écrit ce code.

```ruby
def introduce(first_name:, last_name:, age:, location:
    'Mickeyville', reverse: true)
  return "#{last_name} #{first_name} a #{age} ans et habite à #{lo
      cation}." if
      reverse
  "#{first_name} #{last_name} a #{age} ans et habite à #{location}
      ."
end

introduce(first_name: 'Donald', last_name: 'Duck', location:
    'Donaldville', age: 86)
```

Ici, pas besoin de connaître l'ordre des arguments dans la définition de la méthode. Bien sûr,

1.3.2. De la clarté !

Comme nous ne cessons de le répéter, nous devons viser un code clair. Nous voulons un code facile à lire et facile à écrire et les paramètres nommés sont un outil supplémentaire pour cela. En particulier, cela signifie qu'ils ne sont pas utiles tout le temps. Par exemple, personne n'aurait l'idée d'avoir un paramètre nommé pour une méthode comme `puts`! Si on se base toujours sur l'exemple de notre méthode `introduce`, un paramètre nommé pour `reverse` est raisonnable et rend effectivement notre code plus clair. Cependant, des paramètres positionnels semblent corrects pour les paramètres restants. Nous pouvons donc écrire cette méthode `introduce`.

```
1 def introduce(first_name, last_name, age, location =
      'Mickeyville', reverse: false)
2   return "#{last_name} #{first_name} a #{age} ans et habite à #{lo
        cation}." if
        reverse
3   "#{first_name} #{last_name} a #{age} ans et habite à #{location}
        ."
4 end
```

Ici, nous utilisons les valeurs par défaut des paramètres positionnels pour `location` afin de le traiter comme les autres paramètres qui ne se rapprochent pas d'une option. Ainsi, seul le paramètre `reverse` est nommé et nous pouvons faire ces différents appels.

```
1 introduce('Mickey', 'Mouse', 92, reverse: true)
2 introduce('Donald', 'Duck', 86, 'Donaldville')
3 introduce('Donald', 'Duck', 86, 'Donaldville', reverse: true)
```

Certains pourraient trouver que l'ordre n'est pas clair pour les paramètres positionnels ici. Nous entrons là dans le domaine du subjectif et chacun décidera alors de la nature des paramètres. Cependant, il semble raisonnable de ne pas avoir `first_name` en paramètre positionnel et `last_name` en paramètre nommé!

La méthode `lines` illustre bien ce point. Elle permet de découper une chaîne de caractères en un tableau de sous-chaînes. Les découpes se font à chaque fois qu'un séparateur apparaît dans la chaîne, ce séparateur étant un paramètre positionnel qui vaut `\n` par défaut. La méthode a également un paramètre nommé, `chomp` qui vaut `false` par défaut et qui indique s'il faut supprimer les séparateurs. Nous pouvons alors faire les appels suivants, tirés de la documentation de la méthode.

```
1 "hello\nworld\n".lines                #=> ["hello\n", "world\n"]
2 "hello  world".lines(' ')             #=> ["hello ", " ", "world"]
```

```ruby
"hello\nworld\n".lines(chomp: true)  #=> ["hello", "world"]
"hello word".lines(' ', chomp: true) #=> ["hello", "world"]
```

Le premier argument est plutôt clair puisque l'on sait ce que la méthode fait. Le second argument, quant à lui, mériterait quelques éclaircissements s'il n'était pas nommé. C'était donc une bonne idée d'avoir `chomp` en paramètre nommé. Nous remarquons d'ailleurs qu'il s'agit là du premier cas d'usage dont nous avons parlé; `chomp` est une option de la méthode. Mais même là, nous pourrions trouver que `str.lines(' ')` n'est pas clair et vouloir faire du séparateur un paramètre nommé… Et ce ne serait pas forcément absurde!

1.4. Retour sur les valeurs par défaut

Pour finir ce chapitre, nous allons voir quelques subtilités des valeurs par défaut

1.4.1. Valeurs par défaut

Les valeurs par défaut sont assez simples lorsque nous utilisons des paramètres nommés. En effet, vu que ces paramètres n'ont pas d'ordre, on peut donner une valeur par défaut à n'importe quel paramètre nommé.

```ruby
def introduce(first_name, last_name, age:, reverse: false,
    location:, universe: 'Disney')
  if reverse
    "#{last_name} #{first_name} [#{universe}] a #{age}
         ans et habite à #{location}."
  else
    "#{first_name} #{last_name} [#{universe}] a #{age}
         ans et habite à #{location}."
  end
end

introduce('Mickey', 'Mouse', age: 92, location: 'Mickeyville',
    reverse: false)
introduce('Bruce', 'Wayne', age: 43, location: 'Gotham City',
    universe: 'DC')
```

Ici, nous pouvons avoir deux paramètres nommés avec un argument par défaut, `reverse` et `universe`. Puisque les paramètres sont nommés, Ruby n'a aucun mal à savoir quel argument correspond à quel paramètre lorsque l'on appelle la méthode. Avec des paramètres positionnels, ce n'est pas le cas.

```ruby
def introduce(first_name, last_name, age, universe = 'Disney',
    location, reverse = false)
  if reverse
    "#{last_name} #{first_name} [#{universe}] a #{age}
         ans et habite à #{location}."
  else
    "#{first_name} #{last_name} [#{universe}] a #{age}
         ans et habite à #{location}."
  end
end
```

Ici, Ruby nous donne une erreur `SyntaxError` dès la définition de la méthode. En fait, les paramètres positionnels avec une valeur par défaut doivent se suivre et former un seul bloc. La définition suivante, elle, est correcte.

```ruby
def introduce(first_name, last_name, age, universe = 'Disney',
    reverse = false, location)
  if reverse
    "#{last_name} #{first_name} [#{universe}] a #{age}
         ans et habite à #{location}."
  else
    "#{first_name} #{last_name} [#{universe}] a #{age}
         ans et habite à #{location}."
  end
end
```

La méthode attend un nombre `i` de paramètres positionnels obligatoires, puis `j` paramètres positionnels facultatifs, et ensuite un nombre `j` de paramètres positionnels obligatoires (dans notre exemple, `i` vaut `3`, `j` vaut `2` et `k` vaut `1`). L'appel de la méthode se passe alors de la façon suivante.

— Les `i` premiers arguments seront récupérés dans les `i` premiers paramètres positionnels (qui sont obligatoires).
— Les `j` derniers arguments seront récupérés dans les `j` derniers paramètres positionnels (qui sont obligatoires).
— Les arguments restants, s'il y en a, seront récupérés par les paramètres facultatifs, dans l'ordre.

Ainsi, Ruby peut dans tous les cas déterminer à quel argument correspond un paramètre. Si les paramètres avec une valeur par défaut ne se suivaient pas, ce serait possible, mais un peu plus embêtant. Regardons les résultats des différents appels suivants.

```ruby
introduce('Mickey', 'Mouse', 92, 'Mickeyville')
introduce('Bruce', 'Wayne', 43, 'DC', 'Gotham City')
introduce('Bruce', 'Wayne', 43, 'DC', true, 'Gotham City')
```

Nous remarquons que nous ne pouvons pas donner de valeur pour le paramètre `reverse` sans en donner pour le paramètre `universe`. C'est une des raisons qui explique que l'on préfère utiliser des paramètres nommés pour des paramètres correspondant à des options: pour modifier juste `option2`, nous ne voulons pas avoir à donner de valeur pour `option1`.

Et nous connaissons maintenant les règles d'utilisation des valeurs par défaut. Bien sûr, ce n'est pas parce qu'il est possible de faire tout ceci qu'il faut à tout prix en abuser. Nous pourrions écrire une méthode avec des paramètres positionnels obligatoires, des paramètres positionnels facultatifs, des paramètres nommés obligatoires et des paramètres nommés facultatifs, mais il faudrait une très bonne excuse pour cela!

Finalement, nous écrirons généralement des méthodes avec un nombre raisonnable d'arguments (nous devrions éviter d'avoir plus de trois ou quatre paramètres) et en utilisant les paramètres comme nous l'avons vu précédemment.

Bonnes pratiques

La plupart des guides de bonnes pratiques conseillent également de placer les paramètres optionnels à la fin de la liste des paramètres et certains conseillent même de préférer les paramètres nommés aux paramètres positionnels pour les paramètres optionnels.

1.4.2. Référence à d'autres paramètres

Parmi les autres choses possibles avec les paramètres de Ruby, il y a la possibilité d'utiliser un paramètre pour définir une valeur par défaut d'un autre paramètre. En voici un exemple.

```ruby
def f(x, y = x + 1)
  "x : #{x} et y : #{y}"
end

f(2, 3) # => x : 2 et y : 3
f(2)    # => x : 2 et y : 4
```

Ici, la valeur par défaut du paramètre `y` dépend de la valeur donnée au paramètre `x`. Cela semble inutile, mais voici un exemple d'utilisation. Écrivons une méthode qui doit créer un compte avec un nom, un prénom et un pseudo. Par défaut, le pseudo est `nom.prénom`.

```ruby
def create_account(first_name, last_name, username:
    "#{first_name}.#{last_name}")
  puts "Le compte #{username} de #{first_name} #{last_name}
      a été créé."
end
```

```
4
5 create_account('Mickey', 'Mouse')
6 create_account('Donald', 'Duck', username: "onc' Donald")
7 create_account('Antonhy', 'Stark', username: 'Tony@Stark')
```

Dans ce dernier exemple, nous utilisons les paramètres `first_name` et `last_name` pour définir la valeur par défaut de `username`. Remarquons de plus que nous l'utilisons cette fois avec un paramètre nommé.

Ce procédé a une seule petite contrainte; lorsque nous l'utilisons pour définir un paramètre `param`, nous ne pouvons utiliser que les paramètres qui apparaissent avant `param`.

Les codes suivants nous donneront alors une erreur.

```ruby
def f(x = y + 1, y)
  [x, y]
end

f(1, 2) # => [1, 2]
f(1)    # => Ne donne pas y + 1 mais une NameError.
```

Ici, nous voyons de plus que l'appel de méthode est correct lorsque nous fournissons des arguments pour tous les paramètres. Ce n'est que dans le deuxième appel que nous obtenons une erreur. Essayons de comprendre ce qu'il peut se passer.

1. On voit `f(1)`, on comprend qu'il faut appeler la méthode.
2. La méthode a deux paramètres et l'appel est fait avec un argument, la valeur de cet argument est donc pour le deuxième paramètre (celui qui n'a pas de valeur par défaut).
3. On essaye de définir le premier paramètre. Pour cela, on crée une variable `x` et on lui donne la valeur `y`. Mais `y` n'existe pas, on obtient l'erreur *undefined local variable or method 'y' for main:Object*.

En fait, lorsqu'on rentre dans une méthode, de nouvelles variables sont définies pour chaque paramètre, et elles sont définies dans l'ordre des paramètres, donc pour définir un paramètre `param`, on ne peut utiliser que des paramètres présents avant `param`.

Là encore, ce n'est pas quelque chose qui est très souvent utilisé, même s'il peut être utilisé dans certaines situations comme l'a montré l'exemple `create_account`.

Conclusion

Finalement, même si certaines subtilités sont présentes, notamment au niveau des valeurs par défaut, les deux types de paramètres ne sont pas vraiment compliqués à comprendre, ni à utiliser.

Retenons également que le plus important est d'avoir un **code clair et lisible** et que l'abus de syntaxes compliquées est à l'opposé de ce but.

2. Listes d'arguments

Introduction

Après avoir vu les deux types de paramètres des méthodes de Ruby, nous allons nous intéresser à une mécanique permettant de récupérer plusieurs arguments dans un seul paramètre qui correspondra alors soit à une liste, soit à une table de hachage.

2.1. L'opérateur de splat

Pour commencer, nous allons parler d'un opérateur appelé opérateur de *splat*. Cet opérateur est principalement utilisé pour construire et déconstruire des tableaux en récupérant plusieurs éléments de la manière suivante.

```ruby
# On déconstruit ary en récupérant les deux premiers
# éléments, le dernier, et ce qui reste au milieu.
ary = [1, 2, 3, 4, 5]
a, b, *middle, c = ary
print middle # => [3, 4]

# On construit un tableau en utilisant les
# éléments de deux tableaux.
ary2 = [1, 2, *middle, 5, *[6, 7]]
print ary2 # => [1, 2, 3, 4, 5, 6, 7]
```

2.1.1. Avec les paramètres positionnels

L'opérateur de splat `*` est utilisé de manière similaire à ce que nous venons de faire avec les tableaux. D'un côté, il permet de transformer un tableau en une liste d'arguments (ça correspond à déconstruire le tableau).

```ruby
def f(x, y, z, t, u, v)
  [x, y, z, t, u, v]
end

ary = [1, 2, 3, 4, 5, 6]
```

```
6  f(*ary) # Est équivalent à f(1, 2, 3, 4, 5, 6)
7  f(ary) # Donne une erreur, un seul argument est donné à f.
8
9  ary = [3, 4]
10 f(1, 2, *ary, 5, 6) # => [1, 2, 3, 4, 5, 6]
11
12 ary1 = [2, 3]
13 ary2 = [5]
14 f(1, *ary1, 4, *ary2, 6) # => [1, 2, 3, 4, 5, 6]
```

Finalement, ici, il sert à «aplatir» le tableau dans les arguments. Nous pouvons d'ailleurs en avoir plusieurs comme nous l'avons fait dans le dernier exemple.

Et de l'autre côté, l'opérateur de *splat* permet de transformer une liste d'arguments en tableau. C'est ce qu'il fait lorsqu'il est dans une définition de méthode.

```
1  def f(*ary)
2    ary
3  end
4
5  f              # => []
6  f(1, 2)        # => [1, 2]
7  f(1, 2, 3, 4) # => [1, 2, 3, 4]
8  f([1, 2])      # => [[1, 2]]
```

On obtient ainsi une méthode qui prend un nombre quelconque d'arguments qu'elle récupère dans un tableau. C'est par exemple le cas de la méthode `concat` des chaînes de caractères. Elle permet de concaténer toutes les chaînes passées en paramètre à la chaîne de départ.

```
1  'ab'.concat('cd', 'ef', 'gh') # => 'abcdefgh'
```

L'opérateur de *splat* est très puissant. Il faut cependant parfois se demander si la méthode attend un tableau ou une liste d'arguments (on obtient une erreur en donnant un tableau à `concat` et dans notre dernier exemple avec `f`, nous obtenons `[[1, 2]]` en donnant à `f` un tableau), mais ce n'est pas très gênant. Au contraire, cela nous donne plutôt de la flexibilité dans l'écriture du code.

Notons de plus que nous pouvons accepter d'autres arguments positionnels.

```
1  def f(x, y, *ary, z, t)
2    ary
3  end
4
5  f(1, 2, 3, 4, 5, 6, 7) # => [3, 4, 5]
```

```ruby
f(1, 2, 3, 4)           # => []
```

2.1.2. Les limites

En revanche, nous ne pouvons avoir qu'un seul paramètre avec un *splat*. Et c'est plutôt logique, dans le code qui suit, nous n'avons aucune manière de savoir à quels arguments doivent correspondre `ary1` et `ary2` (et nous avons ce problème dès qu'il y a plus d'un paramètre avec *splat*).

```ruby
def f(x, *ary1, y, *ary2, z)
  ary1
end

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

Finalement, voyons comment se comporte l'opérateur de *splat* en présence d'un paramètre avec une valeur par défaut. En fait, les paramètres ayant une valeur par défaut doivent être placés **avant** l'éventuel paramètre avec le *splat*.

```ruby
# Donne une erreur.
def f(x, y, *ary, z = 'z')
  print "x : #{x}, y : #{y}, ary : #{ary} et z : #{z}."
end

# Est OK.
def f(x, y = 'y', **ary, z)
  print "x : #{x}, y : #{y}, ary : #{ary} et z : #{z}."
end
```

À part ça, le comportement est normal. On remplit d'abord les paramètres obligatoires, puis les paramètres facultatifs, et s'il reste des paramètres, ils iront dans le paramètre *splat*.

```ruby
f(0, 4)          # x : 0, y : y, ary : [] et z : 4.
f(0, 1, 4)       # x : 0, y : 1, ary : [] et z : 4.
f(0, 1, 2, 3, 4) # => x : 0, y : 1, ary : [2, 3] et z : 4.
```

Bien sûr, ce n'est pas quelque chose que nous croiserons tous les jours. De manière générale, il vaut mieux avoir une méthode qui ne prend pas trop de paramètres. Nous voulons des méthodes qui font une action et la font bien (conformément à tout ce qui est principe de responsabilité unique, KISS et autres). Ainsi, une méthode qui prend trop d'arguments n'est pas une bonne idée!

2.2. L'opérateur de double splat

2.2.1. Avec les paramètres nommés

Il y a également un opérateur de *splat* utilisé avec les paramètres nommés. Il s'agit cette fois du double *splat* `**` et lui permet de faire le passage entre table de hachage à paramètres nommés. En dehors de ceci, il fonctionne comme l'opérateur de *splat*.

```ruby
def compute_price(price, tva: 0.2, offer: 0)
  price * (1 + tva) - offer
end

compute_price(100, offer: 10) # => 110.0

hash = {tva: 0.18, offer: 6}
compute_price(100, **hash)  # => 112.0
# Équivalent à compute_price(100, tva: 0.18, offer: 6)

compute_price(100, **{tva: 0.3}) # => 130.0
```

Dans «l'autre sens», nous pouvons récupérer une table de hachage dans une méthode en l'appelant avec des arguments nommés.

```ruby
def build_computer(**components)
  puts 'On construit un ordinateur avec les composants suivants :'
  components.each do |c, v|
    puts "- #{c} : #{v}."
  end
end

build_computer(
  "carte mère": 'celle-là',
  écran: 'lui',
  processeur: 'un bon processeur',
  "carte graphique": 'une géniale'
)
```

Point de syntaxe

La syntaxe `:"str"` permet de déclarer un symbole équivalent à `:str`. Ainsi, `:"écran` déclare le symbole `:écran`. Cette syntaxe permet notamment de déclarer des symboles contenant des espaces. Ici, avec `"carte mère": value`, nous avons la clé `:carte mère` et la valeur `value`.

Ici, nous sommes intéressés par une table de hachage (que nous voulons parcourir), nous préférons donc une table de hachage à des paramètres nommés. Cela nous permet de plus de faire facilement des ordinateurs plus ou moins complets; nous pouvons facilement rajouter une carte réseau ou un disque SSD, là où avec des paramètres nommés il nous faudrait rajouter un paramètre pour la carte réseau, un autre pour le SSD (et avec des valeurs par défaut pour beaucoup d'entre eux).

Nous ne pouvons avoir qu'un seul paramètre avec un double *splat*, mais contrairement au *splat*, il est obligatoirement à la fin des paramètres nommés.

```ruby
def f(arg1: 1, arg2: 2, **hash)
  hash
end

f(other: 3, key: 4) # => {other: 3, key: 4}

# Le code qui suit donne une erreur
def g(**hash, arg1: 1, arg2: 2)
  hash
end
```

2.2.1.1. Paramètres nommés et table de hachage, la même chose ?

Nous constatons facilement le lien fort entre les paramètres nommés et les tables de hachage. En fait, avant la version 2.7 de Ruby, il y avait des conversions implicites dans tous les sens entre les tables de hachage et les paramètres nommés. Ce code était par exemple valide.

```ruby
def f(arg:)
  arg
end

hash = {arg: 2}
f(hash)
```

Depuis la version 2.7, ce code nous donne un avertissement transformé en erreur depuis la version 3 de Ruby. La méthode attend un argument nommé et nous lui donnons un argument positionnel (qui se trouve être une table de hachage), nous obtenons une erreur *ArgumentError (wrong number of arguments (given 1, expected 0 ; required keyword : arg))*. Pour le corriger, nous devons explicitement donner à `f` des paramètres nommés.

```ruby
f(**hash) # => 2
```

En fait, **une** conversion implicite a été conservée, parce qu'elle était totalement claire, sans ambiguïté. Il s'agit du cas où une méthode a un paramètre positionnel et que nous lui donnons des arguments nommés. Ruby va alors automatiquement convertir ces arguments nommés en table de hachage.

```ruby
def f(hash)
  hash
end

f(arg1: 1, arg2: 2) # => {arg1: 1, arg2: 2}

def g(x, hash)
  hash
end

g(1, arg1: 1, arg2: 2) # => {arg1: 1, arg2: 2}

# L'appel qui suit donne une erreur.
g(arg1: 1, 1)

def h(x, hash, key: 1)
  hash
end

# Ne fonctionne pas si la méthode a des paramètres nommés
h(1, arg1: 1, key: 1)
```

Cette conversion ne fonctionne que s'il ne peut vraiment pas y avoir d'ambiguïté.

— Comme `h` a des paramètres nommés, on obtient une erreur.
— Le second appel de `g` donne une erreur, car les arguments nommés doivent être donnés après les arguments positionnels.

En particulier, ces deux informations signifient que pour que cette conversion fonctionne, les arguments nommés que l'on veut «convertir» doivent être en dernier et seront donc forcément récupérés dans le dernier argument de la méthode (argument qui est forcément positionnel).

Finalement, une méthode a d'abord ses paramètres positionnels, parmi lesquels il peut y avoir un paramètre avec un *splat* (généralement le dernier, même si ce n'est pas obligatoire), puis ses paramètres nommés et ensuite potentiellement un paramètre avec un double *splat*.

```ruby
def f(x = 'x', *ary, y, i: 'i', j: 'j', **hash)
  puts "ary : #{ary}."
  puts "x vaut #{x} et y vaut #{y}."
  puts "i: #{i} et j: #{j}."
  puts "hash : #{hash}."
```

```ruby
end

f(1)
f(0, 1)
f(1, a: 2, b: 3)
f(0, 1, 2, k: -1, i: 0, l: -2)
```

2.3. Histoire des paramètres nommés

2.3.1. L'avant Ruby 2

Avant la version 2 de Ruby, il n'y avait pas de paramètre nommé. Mais l'envie de tels paramètres existait déjà, et ils étaient souvent simulés à l'aide de table de hachage. L'idée est simple: la méthode prend en paramètre une table de hachage dont les clés sont les noms que nous aurions voulu pour les paramètres. Une version simple de `introduce` pourrait alors s'écrire de la manière suivante.

```ruby
def introduce(kwargs)
  first_name = kwargs[:first_name]
  last_name = kwargs[:last_name]
  age = kwargs[:age]
  "#{first_name} #{last_name} a #{age} ans."
end

introduce({first_name: 'Mickey', age: 92, last_name: 'Mouse'})
```

Notons que nous pouvons utiliser la méthode `fetch_values` pour récupérer plusieurs clés d'une table de hachage d'un coup. De plus, il est également possible de simuler des valeurs par défaut à l'aide de la méthode `merge`. Elle s'applique à une table de hachage `h` et renvoie la fusion de cette table avec les arguments passés à la méthode (et si une clé est présente dans `h` et dans une table en argument, c'est la valeur de cette dernière table qui est conservée).

```ruby
def introduce(kwargs)
  hash = {reverse: false, location: 'Mickeyville'}.merge(kwargs)
  first_name, last_name, age, location, reverse =
      hash.fetch_values(
    :first_name,
    :last_name,
    :age,
    :location,
    :reverse
  )
```

```
10    return "#{last_name} #{first_name} a #{age} ans et habite à #{lo
          cation}." if
          reverse
11    "#{first_name} #{last_name} a #{age} ans et habite à #{location}
          ."
12  end
13
14  introduce({first_name: 'Mickey', age: 92, last_name: 'Mouse',
          reverse: true})
15  introduce({first_name: 'Donald', last_name: 'Duck', location:
          'Donaldville', age: 86})
```

2.3.2. Introduction des paramètres nommés

Le code est correct, mais la syntaxe est plus lourde que celle que nous obtenons avec des paramètres nommés. Les paramètres nommés arrivent donc… À point nommé. De plus, ils donnent des erreurs beaucoup plus claires. Lorsqu'il manque un paramètre, nous obtenons une `ArgumentError` nous indiquant le paramètre manquant dans le cas des paramètres nommés. Avec une table de hachage, ce serait une `KeyError` dans le cas où nous utiliserions `fetch_values` et avec la syntaxe `hash[:key]` nous n'obtenons même pas d'erreur, mais `nil` si `:key` n'est pas une clé de `hash`!

```
1  def introduce(kwargs)
2    first_name = kwargs[:first_name]
3    last_name = kwargs[:last_name]
4    age = kwargs[:age]
5    "#{first_name} #{last_name} a #{age} ans."
6  end
7
8  introduce({}) #=> '  a  ans.'
```

De plus, donner une table de hachage avec des clés supplémentaires ne cause aucune erreur. Ce n'est généralement pas très gênant, mais ce serait mieux! Imaginons par exemple que je crois avoir programmé `introduce` avec `reverse` mais que ce n'est pas le cas. Mes appels avec la clé `reverse` dans la table de hachage ne me permettent pas de savoir que je ne l'ai pas fait.

Toujours est-il que les paramètres nommés arrivent avec Ruby 2 et permettent d'écrire ce genre de code plus facilement.

2.3.3. Le passage de témoin

Avec ces paramètres nommés, Ruby a l'idée de permettre de donner une table de hachage en argument là où des paramètres nommés sont attendus (les valeurs sont alors automatiquement récupérées dans la table de hachage). Ça permettait notamment aux codes comme celui qui

suit de fonctionner et de ne pas avoir à déconstruire la table de hachage pour passer chaque argument à la méthode.

```ruby
def introduce(first_name:, last_name:, age:)
  "#{first_name} #{last_name} a #{age} ans."
end

hash = {first_name: 'Mickey', age: 92, last_name: 'Mouse'}
introduce(hash)
```

Ainsi, une méthode pouvait être réécrite pour utiliser les paramètres nommés à la place d'une table de hachage sans pour autant changer tous les appels à la méthode, nous n'avions pas besoin de *double splat* pour transformer la table de hachage passée en argument.

Cela a permis d'opérer une migration en douceur. Comme nous l'avons vu précédemment , ce code ne fonctionne plus depuis la version 3 de Ruby (et donne un avertissement en Ruby 2.7).

Connaître l'histoire des paramètres nommés permet de savoir pourquoi il peut exister une certaine confusion avec les tables de hachage, et pourquoi certaines conversions étaient autorisées. En clair, cela permet de comprendre certains choix de l'équipe de développement de Ruby.

Conclusion

Les opérateurs de *splat* et de double *splat* sont très utiles et très puissants… Mais pas forcément adaptés à toutes les situations. Comme nous l'avons vu, les paramètres nommés offrent cependant de meilleurs messages d'erreurs que ceux obtenus avec un *double splat*; ce dernier est néanmoins utile dans certains cas (dans l'exemple de construction d'un ordinateur, n'importe quelle clé est acceptée).

Conclusion

Ça y est, nous savons maintenant ce qu'il y a à savoir pour comprendre comment se passe la gestion des paramètres en Ruby.

i

À venir

Ce tutoriel n'est pas terminé. D'autres chapitres sont à venir, notamment pour traiter des blocs et de leurs arguments ou encore de quelques astuces de Ruby.