Venir à Ruby après un autre langage

Au premier coup d'œil, Ruby vous rappellera certainement les autres langages que vous avez déjà pu utiliser. Rien de plus normal : une bonne partie de la syntaxe se rapproche de Perl, de Python et de Java (entre autres), de sorte que si vous venez d'un de ces langages, apprendre Ruby devrait être plutôt facile.

Le présent document se partage en deux grandes sections. En premier lieu, une revue rapide de ce à quoi vous pouvez vous attendre en passant d'un langage donné à Ruby. Dans un deuxième temps, un examen plus approfondi des fonctionnalités propres de Ruby, illustré de quelques comparaisons avec d'autres langages.

À quoi s'attendre : passer d'un langage X à Ruby

- De C/C++ à Ruby
- De Java à Ruby
- De Perl à Ruby
- De PHP à Ruby
- De Python à Ruby

Les fonctionnalités importantes et autres

Lancez-vous, c'est facile !

Essayez Ruby !

Apprenez Ruby en vingt minutes

Vous venez d'un autre langage ?

Explorez un nouvel univers...

Documentation

Livres

Bibliothèques

Témoignages

Participez à la communauté

Listes de diffusion: discutez de Ruby avec des programmeurs du monde entier.

Groupes d'utilisateur: entrez en contact avec d'autres rubyistes de votre région.

Blogs: suivez toute

astuces

Voici un aperçu et des conseils concernant les fonctionnalités majeures de Ruby que vous allez rencontrer au cours de votre apprentissage.

Itération

Représentant en général une nouveauté technique pour celui qui découvre Ruby, les blocs et les itérateurs demandent en général un petit temps d'adaptation. Au lieu de construire une boucle sur un index (comme en C, C++ ou java < 1.5) ou sur une liste (comme en Perl, avec `for (@a) {...}`, ou en Python, avec `for i in Liste: ...`), vous écrirez souvent en Ruby quelque chose comme :

```
une_liste.each do |item_courant|
  # Nous sommes dans le bloc.
  # Travaillons avec l'item_courant de la liste...
end
```

Pour plus d'informations sur `each` et ses méthodes apparentées (`collect`, `find@n @inject`, `sort`, etc.), voyez `ri Enumerable` dans un terminal, par exemple (affinez ensuite avec `ri Enumerable#méthode`).

Des valeurs, partout

Ruby ne fait pas de différence entre une expression et une déclaration. Tout ce qui existe possède intrinsèquement une valeur, même si cette valeur est **nil** (qui *modélise* l'absence de valeur). De ce fait, ce qui suit est possible :

```
x = 10
y = 11
z = if x < y
      true
    else
      false
    end
z # => true
```

Les symboles ne sont pas des chaînes allégées

Beaucoup de débutants en Ruby se débattent longtemps avec la nature *exacte* des symboles, et la pertinence de leur utilisation.

l'actualité de la communauté Ruby.

Ruby Core: apportez votre aide à l'élaboration du prochain Ruby.

Rapports de bugs: aidez à corriger et améliorer Ruby.

Syndication

Actualité récente (RSS)

La meilleure façon de décrire les symboles seraient de dire qu'ils sont des identités. Un symbole porte essentiellement sur le **qui**, et non sur le *ce que*—il ne s'agit pas de déterminer la nature de ce qui est, mais bien de savoir ce qui est. Le test suivant dans IRB illustre la notion :

```
irb(main):001:0> :george.object_id ==
:george.object_id
=> true
irb(main):002:0> "george".object_id ==
"george".object_id
=> false
irb(main):003:0>
```

La méthode `object_id` retourne l'identité d'un objet donné. Si deux objets ont la même identité, alors il s'agit d'un seul et même objet (même allocation mémoire).

Comme vous pouvez le voir, une fois un symbole défini, tout symbole de mêmes caractères (même nom) fait référence au même objet en mémoire. Pour deux symboles donnés représentant les mêmes caractères, l' `object_id` est unique.

Voyez maintenant le résultat obtenu sur la chaîne de caractères "george". L' `object_id` ne correspond pas. Cela signifie qu'il y a bien deux objets différents, qui se ressemblent, mais qui n'ont pas la même identité. À chaque fois qu'une chaîne de caractère est utilisée, Ruby réalise une nouvelle allocation en mémoire.

Si vous doutez quant à l'utilisation d'un symbole ou d'une chaîne, estimez ce qui est dans votre cas le plus important : l'identité de l'objet (par exemple, la clé d'un hash) ou son contenu (par exemple, « george »).

Tout est un objet

En Ruby, l'expression « orienté objet » n'est pas une hyperbole. Jusqu'aux classes et aux entiers, tout ce qui est manipulable est un objet à part entière—objet acceptant les manipulations usuelles du type :

```
# Voici un bout de code équivalent à :
# class MaClass
#   attr_accessor :var_instance
# end
MaClass = Class.new do
  attr_accessor :var_instance
end
```

Des constantes variables

Ce qu'on appele habituellement constante ne l'est pas vraiment, en Ruby. La modification d'une constante est possible : elle produit un avertissement, mais ne stopppe pas l'exécution du programme. Ce qui n'es pas un encouragement en soi à redéfinir les constantes, cela dit.

Conventions de nommages

Ruby intègre (et impose) des conventions de nommage sémantiques. Si un identifiant commence par une majuscule, il s'agit d'une constante. S'il débute par un signe dollar (`$`), c'est une variable globale. S'il débute par un `@` , il s'agit d'une variable d'instance. S'il commence par `@@` , c'est une variable de classe.

Les noms de méthodes peuvent débuter par une majuscule—mais cela peut conduire à des confusions, comme dans l'exemple qui suit :

```ruby
Constante = 10
def Constante
  11
end
```

`Constante` vaut 10, mais `Constante()` vaut 11.

Paramètres « mot-clés »

Depuis Ruby 2.0, à l'instar de Python, il est possible de définir une méthode avec des paramètres « mot-clés » :

```ruby
def deliver(from: "A", to: nil, via: "mail")
  "Sending from #{from} to #{to} via #{via}."
end

deliver(to: "B")
# => "Sending from A to B via mail."
deliver(via: "Pony Express", from: "B", to: "A")
# => "Sending from B to A via Pony Express."
```

La vérité, toujours la vérité

En Ruby, tout ce qui n'est pas **nil** ou **false** est considéré comme vrai (**true**). En C, Python et dans bien d'autres langages, le 0 et d'autres valeurs (telles les listes vides) sont considérées fausses. Voyez par exemple le bout de code suivant, écrit en Python :

```
# en Python
if 0:
  print "0 est vrai/true"
else:
  print "0 est faux/false"
```

Ce qui affichera « 0 est faux/false. » L'équivalent en Ruby, maintenant :

```
# en Ruby
if 0
  puts "0 est vrai/true"
else
  puts "0 est faux/false"
end
```

Cette fois, vous lirez « 0 est vrai/true. »

Les modifications d'accès sont actives jusqu'à preuve du contraire

Considérez le bout de code suivant :

```
class MyClass
  private
  def a_method; true; end
  def another_method; false; end
end
```

Vous pourriez vous attendre à ce que `another_method` soit publique. Ce n'est pas le cas. Le mot-clé `private` est effectif jusqu'à la fin de la portée actuelle (ici, la classe `MyClass`), ou jusqu'à ce qu'un autre mot-clé change la donne. Par défaut, les méthodes sont publiques :

```
class MyClass
  # Méthode implicitement publique
  def a_method; true; end
```

```
  private

  # Cette méthode est privée
  def another_method; false; end
end
```

`public`, `private` et `protected` sont des méthodes à part entière, elles peuvent prendre des paramètres. Si vous passez un symbole à l'une d'elle, la visibilité de cette méthode est modifiée.

Accès aux méthodes

En Java, `public` signifie qu'une méthode est accessible par tout un chacun. `protected` signifie que les instances de la classe, les instances de classes filles et les instances de classes du même paquet peuvent y accéder, à l'exclusion de tout autre domaine. `private` signifie que personne à part les instances de la classe ne peut accéder à la méthode.

Ruby se démarque un peu ici. `public` signifie la même chose, mais `private` signifie que les méthodes sont accessibles uniquement si elles peuvent être appelées sans destinataire explicite. En fait, seul `self` est autorisé. `protected` est à part : une méthode protégée peut être appelée depuis une instance de classe ou de classe fille, mais également avec une autre instance comme destinataire.

Un exemple, repris de la FAQ Ruby:

```
$ irb
irb(main):001:0> class Test
irb(main):002:1>   # publique par défaut
irb(main):003:1*   def func
irb(main):004:2>     99
irb(main):005:2>   end
irb(main):006:1>
irb(main):007:1*   def ==(other)
irb(main):008:2>     func == other.func
irb(main):009:2>   end
irb(main):010:1> end
=> nil
irb(main):011:0>
irb(main):012:0* t1 = Test.new
=> #<Test:0x34ab50>
irb(main):013:0> t2 = Test.new
=> #<Test:0x342784>
irb(main):014:0> t1 == t2
=> true
```

```
irb(main):015:0> # passer `func` en protected
fonctionne toujours :
irb(main):016:0* # la référence à other est
autorisée
irb(main):017:0* class Test
irb(main):018:1>   protected :func
irb(main):019:1> end
=> Test
irb(main):020:0> t1 == t2
=> true
irb(main):021:0> # par contre, si `func` est
private...
irb(main):022:0* class Test
irb(main):023:1>   private :func
irb(main):024:1> end
=> Test
irb(main):025:0> t1 == t2
NoMethodError: private method `func' called for #
<Test:0x342784>
        from (irb):8:in `=='
        from (irb):25
        from :0
irb(main):026:0>
```

Les classes restent ouvertes

Ouvertes à la modification, à tout moment. Vous pouvez y faire des ajouts, les modifier durant l'exécution. Y compris les classes standards, telles que `Fixnum`, voire `Object`, la classe parente de toute autre. Par exemple, l'application Ruby on Rails défini nombre de méthodes pour traiter le temps, au sein de `Fixnum`. Voyez ceci :

```ruby
class Fixnum
  def hours
    self * 3600 # nombre de secondes dans une heure
  end
  alias hour hours
end

# 14 heures après le 1er janvier à 00h00
Time.mktime(2006, 01, 01) + 14.hours # => Sun Jan
01 14:00:00
```

Indices sémantiques sur les méthodes

Un nom de méthode en Ruby peut se terminer par un point d'interrogation ou d'exclamation. Le premier signe de ponctuation

sera utilisé pour les méthodes qui donnent une réponse à une question (par exemple, `Array#empty?`, qui retourne `true` si le destinataire est vide). Les méthodes potentiellement « dangereuses » (parce qu'elles modifient `self` ou les paramètres, comme `exit!`) sont signalées par le second signe.

Mais ce n'est pas obligatoire, et parfois le choix a été de ne rien indiquer du tout. Ainsi, `Array#replace` modifie sur place le contenu d'un tableau avec le contenu d'un autre tableau.

Les méthodes singletons

Une méthode singleton est une méthode liée à *un* objet. Elle n'est disponible que pour l'objet défini.

```
class Car
  def inspect
    "Cheap car"
  end
end

porsche = Car.new
porsche.inspect # => Cheap car
def porsche.inspect
  "Expensive car"
end

porsche.inspect # => Expensive car

# Les autres objets ne sont pas affectés
other_car = Car.new
other_car.inspect # => Cheap car
```

Gestion des méthodes manquantes

Si un message ne correspond pas à une méthode explicitement définie, Ruby n'abandonne pas la partie et passe la main à la méthode nommée `method_missing`. Elle l'informe du nom de la méthode introuvable et des éventuels paramètres joints au message. Par défaut, `method_missing` lève une exception du type NameError, mais vous pouvez la redéfinir pour mieux l'intégrer au contexte de votre application—de nombreuses bibliothèques exploitent cette possibilité. Voici un exemple :

```
# id est le nom de la méthode appelée, la syntaxe *
renvoie
```

```
# tous les paramètres dans un tableau nommé «
arguments »
def method_missing(id, *arguments)
  puts "La méthode #{id} a été appelée, mais elle
n'existe pas. " +
       "Voici les paramètres de l'appel : #
{arguments.join(", ")}"
end

__ :a, :b, 10
# => La méthode __ a été appelée, mais elle
n'existe pas. Voici les paramètres de l'appel :
# arguments: a, b, 10
```

Le code ci-dessus ne fait qu'afficher les détails de l'appel, mais vous êtes libres de manipuler tout ça à votre guise.

Envoi de message et non appel de fonction

Ce qui est communément appelé « appel de méthode » est bel et bien *un message envoyé* à un autre objet—voyez plutôt :

```
# Ceci...
1 + 2
# est équivalent à...
1.+(2)
# qui est la même chose que :
1.send "+", 2
```

Les blocs aussi sont des objets—bien qu'ils ne le sachent pas (encore)

Les blocs (*closures*) sont massivement utilisés dans la bibliothèque standard. Pour appeler un bloc, vous pouvez utiliser `yield`, ou bien le transformer en `Proc` en rajoutant un paramètre spécial à la liste d'arguments, comme ceci :

```
def bloc(&le_bloc)
  # Ici, dedans, le_bloc est le bloc passé à la
méthode
  le_bloc # retourne le bloc
end
addition = bloc { |a, b| a + b }
# addition est maintenant un objet du genre Proc
addition.class # => Proc
```

Cela signifie que vous pouvez créer des blocs en-dehors du contexte des appels de méthode, en utilisant `Proc.new` avec un bloc ou en appelant une méthode `lambda`.

De la même façon, les méthodes sont également des objets bien réels :

```
method(:puts).call "puts est un objet !"
# => puts est un objet !
```

Opérer sur les opérateurs

La plupart des opérateurs ne sont là que pour faciliter la vie du programmeur (et gèrent aussi les règles de priorité mathématique). Vous pouvez, par exemple, redéfinir la méthode `+` de la classe `Fixnum` :

```
class Fixnum
  # Possible, mais pas recommandé...
  def +(other)
    self - other
  end
end
```

Pas besoin des `operator+` comme en C++, etc.

Vous pouvez aussi créer des accès du type tableau en définissant les méthodes `[]` et `[]=`. Pour définir les signes + et -, comme pour +1 et -2, vous définirez les méthodes `+`@ et `-`@, respectivement.

Les opérateurs ci-dessous ne sont pas des méthodes, et ne peuvent pas être modifiés :

```
=, .., ..., !, not, &&, and, ||, or, !=, !~, ::
```

Par ailleurs, `+=`, `*=`, etc. ne sont que des raccourcis pour `var = var + autre_var`, `var = var * autre_var`, etc. et ne peuvent être redéfinis.

En savoir (encore) plus

Enthousiaste ? Direction notre section Documentation.

Autres langues disponibles : Български, Deutsch, English, Español, Français, Bahasa Indonesia, Italiano, 日本語, 한국어, polski, Português, Русский, Türkçe, Tiếng Việt, 简体中文, 繁體中文.

Ce site est propulsé par Ruby et Jekyll. Il est fièrement maintenu par des membres de la communauté Ruby. Contribuez en vous rendant sur GitHub ou contactez le webmaster pour toute question ou tout commentaire.