Introduction à Ruby

Rémi "FiFouille" Laurent

FSUGAr &
#ubuntu-fr-classroom

14 mars 2007

Première partie I

Vue d'ensemble

Qui, quand, où ?

- Ruby est l'initiative du japonais Yukihiro "Matz" Matsumoto.
- Il a commencé à écrire ce langage en 1993, la première version publique date de 1995.
- Actuellement Ruby en est à la version 1.8. La version de développement 1.9 débouchera sur Ruby 2.0.
- Ruby est distribué sous les termes de la *GNU GPLv2* ainsi que la *Ruby License*.

Domaines d'utilisation de Ruby

Ruby peut avoir de nombreuses applications :

- Script : sysadmin, génération de code/documents
- Applications *classiques* se basant sur nombre de bibliothèques disponibles
- Web : Ruby on Rails, ERB, ...
- Apprentissage, test : irb, simplicité et lisibilité du code

Informations techniques diverses

Quelques caractéristiques techniques :

- langage de script
- dynamique
- ramasse miette / garbage collector
- procédural - orienté objet - fonctionnel
- **tout est objet**

Machines virtuelles

Actuellement Ruby est relativement plus lent que du Python par exemple
d'où l'idée d'utiliser des VM (Machines Virtuelles)

- Yarv - Yet Another Ruby VM - pour Ruby 2.0 - bientôt
- JRuby - utilisation de la JVM (Java) - opérationnel
- Rite - première VM prévue pour Ruby 2.0 - projet mort ?
- Parrot - la VM prévue pour Perl 6 - ? ? ?

Deuxième partie II

Un peu de pratique

Paquets utiles

Les applications à installer sont :

- Ruby - l'interpréteur
- irb - Interactive Ruby - interpréteur interactif

On peut encore installer :

- Ruby Gems - installeur de 'paquets' ruby
- Rake - Ruby make

Paquets utiles

Les applications à installer sont :

- Ruby - l'interpréteur
- irb - Interactive Ruby - interpréteur interactif

On peut encore installer :

- Ruby Gems - installeur de 'paquets' ruby
- Rake - Ruby make

Paquets utiles

Les applications à installer sont :

- Ruby - l'interpréteur
- irb - Interactive Ruby - interpréteur interactif

On peut encore installer :

- Ruby Gems - installeur de 'paquets' ruby
- Rake - Ruby make

Les commandes


```
sudo apt-get install ruby1.8 irb1.8
sudo apt-get install rubygems rake
```

Astuce :

```
echo "require 'irb/completion'" >> $HOME/.irbrc
```

Vous aurez alors une auto complétion dans *irb*

Les commandes


```
sudo apt-get install ruby1.8 irb1.8
sudo apt-get install rubygems rake
```

Astuce :

```
echo "require 'irb/completion'" >> $HOME/.irbrc
```

Vous aurez alors une auto complétion dans *irb*

Les commandes


```
sudo apt-get install ruby1.8 irb1.8
sudo apt-get install rubygems rake
```

Astuce :

```
echo "require 'irb/completion'" >> $HOME/.irbrc
```

Vous aurez alors une auto complétion dans *irb*

Hello World !

Dans irb :

```
puts "Hello World!"
```

Dans un fichier, hello.rb :

```
#!/usr/bin/ruby -w
puts "Hello World!"
```

- -w active les avertissements
- `ruby -c hello.rb` vérifie la syntaxe
- n'oubliez pas de rendre hello.rb exécutable ...

Hello World !

Dans irb :

```
puts "Hello World!"
```

Dans un fichier, hello.rb :

```
#!/usr/bin/ruby -w
puts "Hello World!"
```

- -w active les avertissements
- `ruby -c hello.rb` vérifie la syntaxe
- n'oubliez pas de rendre hello.rb exécutable ...

Hello World !

Dans irb :

```
puts "Hello World!"
```

Dans un fichier, hello.rb :

```
#!/usr/bin/ruby -w
puts "Hello World!"
```

- -w active les avertissements
- `ruby -c hello.rb` vérifie la syntaxe
- n'oubliez pas de rendre hello.rb exécutable ...

Hello #{you} !


```
you = gets
puts "Hello " + you + " !"
you = gets.chomp
puts "Hello #{you} !"
```

On obtient :

```
fifi <ENTER>
Hello fifi
 !
fifi <ENTER>
Hello fifi !
```

Hello #{you} !


```
you = gets
puts "Hello " + you + " !"
you = gets.chomp
puts "Hello #{you} !"
```

On obtient :

```
fifi <ENTER>
Hello fifi
 !
fifi <ENTER>
Hello fifi !
```

4 Installation
- Indispensables
- Ubuntu/Debian

5 Premiers pas
- Hello World !
- Hello #{you} !

6 "Types" de base
- Numériques
- Chaînes de caractères
- Tableaux
- Tables de hachage
  - Exemples
  - Application
- Conversion

Numeric, Integer, Fixnum, Bignum, Float

Hiérarchie :

...

- Numeric
  - Integer
    - Fixnum
    - Bignum
  - Float

```
puts 5                  => 5
puts 5*2                => 10
a = 5**2
puts a                  => 25
puts a + 10             => 35
puts "5" * 4            => 5555  ?!&?
```

Numeric, Integer, Fixnum, Bignum, Float

Hiérarchie :

...

- Numeric
  - Integer
    - Fixnum
    - Bignum
  - Float

```
puts 5                  => 5
puts 5*2                => 10
a = 5**2
puts a                  => 25
puts a + 10             => 35
puts "5" * 4            => 5555   ?!&?
```

Chaînes de caractères - String


```
puts "foobar"                  => foobar
str = "foo" + "bar"
puts str + " hello"            => foobar hello
puts str * 2                   => foobarfoobar
puts "_#{str}_"                => _foobar_
puts "foobar".upcase           => FOOBAR
```

puts ajoute le retour à la ligne, alors que *print* non.

Les tableaux - Array


```
arr1 = [1,2,3]
arr1[0]                 => 1
arr1[-1]                => -3
arr1[1..2]              => [2, 3]
arr2 = []
arr2 << 2 << 4 << 3
arr1 + arr2             => [1, 2, 3, 2, 4, 3]
arr1 & arr2             => [2, 3]
arr1 - arr2             => [1]
(arr1+arr2).sort        => [1, 2, 2, 3, 3, 4]
```

Définition - Hash

Il s'agit d'une table où des *clés* correspondent à des *valeurs*. Une fonction de hachage calcule une clé unique pour une clé donnée et lui fait correspondre une valeur.

- clé 'pomme' - valeur 'rouge'
- collision : 'pomme' ne peut être 'rouge' et 'verte' à la fois
- fonction de hachage : répartition uniforme des index
- rapide d'accès par rapport aux tableaux (surtout pour les grandes tailles)

Exemples - Hash


```
h = { 'pomme' => 'rouge' }
h[42] = "la réponse à ..."
puts h['pomme'] + h[42]
puts h.has_key?'pomme'        => true
puts h.has_key?('poire')      => false
h = Hash.new(5)
puts h[123]                   => 5
h[123] += 2
puts h[123]                   => 7

h = { 'pomme' => [ 'rouge', 'verte']\
    'banane' => 'jaune' }
```

Application pratique

Nous allons compter le nombre de mots dans une phrase :

```
str = "Les pommes sont rouges ou vertes,\
  les bananes jaunes et\
  les poires sont vertes ou jaunes"
h = Hash.new(0)
str.split(/ /).each { |mot| h[mot] += 1 }
puts "jaunes : #{h['jaunes']}"  => 2
puts "rouges : #{h['rouges']}"  => 1
puts "orange : #{h['orange']}"  => 0
```

PS : si vous n'avez pas tout compris, ça va venir ;)

Application pratique

Nous allons compter le nombre de mots dans une phrase :

```
str = "Les pommes sont rouges ou vertes,\
  les bananes jaunes et\
  les poires sont vertes ou jaunes"
h = Hash.new(0)
str.split(/ /).each { |mot| h[mot] += 1 }
puts "jaunes : #{h['jaunes']}"  => 2
puts "rouges : #{h['rouges']}"  => 1
puts "orange : #{h['orange']}"  => 0
```

PS : si vous n'avez pas tout compris, ça va venir ;)

Conversions entre types (classes)

Les conversions sont possibles :

- entier → chaîne de caractère — `5.to_s`
- entier → réel — `6.to_f`
- chaîne → réel — `"3.14".to_f`
- tableau → chaîne de caractère — `[2,3].to_s`
- range → tableau — `(1..5).to_a`
- ...

Attention aux arrondis, mise à plat, ...

Troisième partie III

Commençons à programmer

Opérateurs arithmétiques de base


```
puts "hello" * 2                    => "hellohello"
[1,2,3,4] - [2,3]                   => [1,4]
str = "hello"
str << " world!"
puts str + "%0.2f" % 4.425          => hello world!4.42
[1,2,3] * 2                         => [1,2,3,1,2,3]
puts 5 % 3                          => 2
```

Opérateurs de comparaison

`==`, `!=` (in)égalité au sens strict

```
5 == 5 , "hello" != "Hello"
```

`<=>` comparaison, renvoie -1, 0 ou +1 selon la comparaison, utilisé pour spécifier un ordre de tri (Array#sort par exemple)

`===` comparaison utilisée dans le cadre de *case ... when*

```
(5..10)    ===  6     => true
('a'..'z') === 'b'    => true
('a'..'z') === 'Z'    => false
```

`<`,`>`,`<=`,`>=` classiques dirons-nous

Opérateurs logiques

Opérateurs logiques classiques qui ont, entre autres, des propriétés intéressantes sur les tableaux :

```
[1,2,3] & [1,2]          => [1, 2]
[1,2,3] | [1,2,4]        => [1, 2, 3, 4]
7 ^ 2   (111^010=110)    => 5
6 & 2   (110&010=010)    => 2
```

7 Opérateurs
- Arithmétique
- Comparaison
- Logique

8 Blocs conditionnels
- if elsif else end
- for x in y
- while do ...
- case where else

9 I/O
- Lecture
- Écriture

if *expr* then ... elsif *expr* ... else ... end


```
val = gets
if ( val == 6 )
   puts "c'est 6"
elsif (1..5) === val
   puts "entre 1 et 5 compris"
else
   puts "autre chose"
end

puts "c'est 6" if val == 6
puts "c'est pas 6" unless val == 6
```

for x in y ... f(x) end


```
for i in [1,3,2,4].sort.reverse
   puts "valeur = #{i}"
end

for i in (2..10)
   puts "2 exposant #{i} = #{2**i}"
end
```

while *expr* do ... end


```
i = 2
while i <= 256 do
   puts i
   i = i*i
end
# avec une expression rationnelle
#  pour nos plus fidèles auditeurs
while s = gets
   next if ~ /^#/
   break if ~ /^quitter$/
   puts s
end
```

case *expr* when *expr* then ... else ... end


```
var = gets
case var
   when /quitter/
      puts "quitter"
      exit
   when /heure/
      puts Time.now
   else
      puts "non reconnu"
end
```

7 Opérateurs
- Arithmétique
- Comparaison
- Logique

8 Blocs conditionnels
- if elsif else end
- for x in y
- while do ...
- case where else

9 I/O
- Lecture
- Écriture

Lecture dans un fichier


```
fd = open("monfichier", "r")
fd.each do |ligne|
   puts ligne
end
fd.close
```

On ouvre donc le fichier, en lecture *"r"* et ensuite on utilise un itérateur sur le 'descripteur' de fichier obtenu.

```
fd.each do |ligne|
```

projette ligne par ligne dans la variable *ligne*, variable *ligne* qu'on affiche avec *puts*.

Écriture dans un fichier

L'opération d'écriture n'est pas fort différente :

```
fd = open("monfichier", "w+")

10.times { |i| fd.puts "#{i} : #{i*i}" }

fd.close
```

Il s'agit du même principe que pour la lecture, sauf que nous avons spécifié le *bloc* entre *parenthèses* et pas entre *do ... end*. Voyons maintenant ce qu'est un itérateur et un bloc.

PS : les modes sont les mêmes qu'en C avec *fopen(3)* : *man fopen*

Écriture dans un fichier

L'opération d'écriture n'est pas fort différente :

```
fd = open("monfichier", "w+")

10.times { |i| fd.puts "#{i} : #{i*i}" }

fd.close
```

Il s'agit du même principe que pour la lecture, sauf que nous avons spécifié le *bloc* entre *parenthèses* et pas entre *do ... end*.
Voyons maintenant ce qu'est un itérateur et un bloc.
PS : les modes sont les mêmes qu'en C avec *fopen(3)* : *man fopen*

Quatrième partie IV

Principes avancés

Itérateurs - Définition

Un itérateur permet de parcourir une collection de données, et ce, d'une manière unifiée, quelle que soit la représentation interne de cette collection de donnée.

Dès lors il permet, à l'aide de mécanismes sensiblement ressemblants, de parcourir : un tableau, une liste, un arbre, une table de hachage, une base de donnée, un fichier, ...

Itérateurs - Définition

Un itérateur permet de parcourir une collection de données, et ce, d'une manière unifiée, quelle que soit la représentation interne de cette collection de donnée.

Dès lors il permet, à l'aide de mécanismes sensiblement ressemblants, de parcourir : un tableau, une liste, un arbre, une table de hachage, une base de donnée, un fichier, ...

Itérateurs - Exemples


```
[1,2,3].each { |i| puts i }

h = { 'banane' => 'jaune', 'poire' => 'verte',\
   'pomme' => 'rouge' }
h.each { |k,v| puts "#{k} est #{v}" }

#exemple en Ruby on Rails
list = Clients.find(:all,\
    :conditions => [ 'nom ilike ?', 'jacque' ])
list.each { |c| c.email = 'foo@bar.com'; c.save }
```

Blocs & Procs - Définition

Un peu comme les pointeurs de fonction en C, Ruby permet d'associer une fonction à une variable.
Cependant Ruby va plus loin, non seulement avec les *closures* qui permettent d'accéder à une partie de l'environnement manipulé, mais encore en permettant la création de bloc/proc dynamiques.
Un bloc ou proc peut être vu comme un bout de code associé à un symbole et réutilisable facilement dans différents contextes.

Blocs & Procs - Exemples 1


```
heure = lambda { Time.now.to_s }
def do_n_times(n, &block)
  lambda { n.times { puts yield } }
end

proc1 = do_n_times(3) { "il est " + heure.call }
proc2 = do_n_times(3) { sleep 1;\
   "il est " + heure.call }

proc1.call
2.times { proc2.call }
```

Blocs & Procs - Exemples 2


```
arr = ['11','2','33','4','55']
arr.sort { rand }
arr.sort { |x,y| x.length <=> y.length }

class Hash
  def each_times(&block)
    each { |k,v| k.times { yield v }  }
  end
end
h = { 2 => "hello", 1 => "coin", 4 => "pouet" }

h.each_times { |s| puts "valeur : " + s }
```

Expressions rationnelles - Définition

Une expression rationnelle c'est :
une notation permettant de décrire des chaînes de caractères, en gros :

- de repérer des schémas donnés de chaîne de caractères
- de les manipuler
- (d'impressionner—de traumatiser) (sa copine—ses copains—son patron)

Pour plus d'informations, je vous renvoie à une autre présentation ;)

Expressions rationnelles - Exemples


```
#bon courage ;)
str = "bonjour, comment va ?"
str.match(/^bon(.*),.+(m{2,}).+(..\?)$/)
foo, bar, baz = $1, $2, $3.gsub(/ /, 'h')

puts "#{bar}h, quelle #{foo}née ! #{baz}"

puts "bonjour" if str =~ /^bonjour/
puts str
puts "bien" if str =~ /va\s*\?$/
```

Exceptions - Définition

Avec de simples codes de retour il est parfois difficile de remonter une erreur et de la définir précisément, il est parfois encore plus difficile de se remettre proprement d'une erreur rencontrée.

Les exceptions sont là pour palier à ce problème, en définissant précisément une erreur, et en permettant de terminer proprement chaque partie ou étape du programme.

Les cas les plus classiques sont la fermeture de fichiers ou de connexions réseaux, mais d'autres sont envisageables bien entendu.

Exceptions - Exemples


```
filename = "fichier.txt"
tmpfilename = "/tmp/" + filename
begin
  fd = open(filename, 'w+')
  fd.puts Time.now.to_s
rescue
  raise "erreur" if filename == tmpfilename
  $stderr.puts "essai avec un fichier temporaire"
  filename = tmpfilename
  retry
ensure
  fd.close if fd
end
```

Orienté Objet

Vous voulez qu'on en parle ?

Parce qu'il paraît que ça ferait de trop pour une seule fois ;)

Orienté Objet

Vous voulez qu'on en parle ?

Parce qu'il paraît que ça ferait de trop pour une seule fois ;)

Cinquième partie V

Conclusion

Conclusion

Ruby semble être un langage fort agréable,
facile à apprendre et utiliser,
qui dispose de nombre de bibliothèques intéressantes,
qui propose des extensions simples à mettre en oeuvre,
une approche orientée objet omniprésente,
et de nombreux domaines d'application.

Pour ceux qui ne l'auraient pas compris, l'auteur de ces slides *apprécie* Ruby

Conclusion

Ruby semble être un langage fort agréable,
facile à apprendre et utiliser,
qui dispose de nombre de bibliothèques intéressantes,
qui propose des extensions simples à mettre en oeuvre,
une approche orientée objet omniprésente,
et de nombreux domaines d'application.

Pour ceux qui ne l'auraient pas compris, l'auteur de ces slides *apprécie* Ruby

Documentation disponible

Il existe beaucoup de documentation accessible gratuitement, la plupart du temps :

- `http://www.rubycentral.com/book/` - EN
- `http://pine.fm/LearnToProgram/` - EN/FR
- `http://www.ruby-lang.org/en/documentation/` - EN
- The Pragmatic Programmer Guide - 2nd Edition par Dave Thomas
- $ gem_server et puis http ://localhost :8808
  si vous avez installé des modules avec *rubygems*
- Agile Web Development with Rails
  pour ceux qui voudraient essayer Ruby On Rails