Ruby v1.0 20/04/15

Le Ruby est un langage de programmation puissant et flexible, vous pouvez l'utiliser pour vos développements web, pour traiter du texte, créer des jeux ou au sein du célèbre framework web Ruby on Rails. Le Ruby est un langage :

Interprété, c'est à dire que vous n'avez pas besoin de compilateur pour écrire et exécuter du Ruby. Vous pouvez programmer sur Codecademy ou sur votre ordinateur (la plupart sont livrés avec un interpréteur Ruby, nous reviendrons plus en détail sur l'interpréteur un peu plus tard).

Orienté en fonction des objets, c'est à dire qu'il permet de manipuler des structures de données appelées objets pour construire des programmes. Nous verrons les objets plus en détail plus tard, mais pour le moment dites-vous simplement qu'en Ruby : tout est un objet.

Facile à utiliser. Le Ruby a été créé par Yukihiro Matsumoto (surnommé "Matz") en 1995. Matz voulait créer un langage qui favorise les besoins de l'homme par rapport à ceux de la machine, c'est pour cela que le Ruby est si facile à assimiler.

Syntaxe CONDITION IF ELSIF ELSE :
```
if variable
   puts "Texte"
elsif
   puts"Texte"
else
   puts "Texte"
end

lol = true
puts "Je lol" if lol == true
//IF simple
```

Syntaxe CONDITION unless:
```
unless faim
  puts "Je programme en Ruby !"
else
  puts "A table !"
end

puts "Je lol" unless lol == true
//Idem avec UNLESS
```

Syntaxe TERNAIRE:
```
puts true ? "Vrai !" : "Faux !"
```

Syntaxe BOUCLE WHILE :
```
while i < 5
  puts i
  i = i +1
end
#ou
until compteur > 10
  puts compteur
  compteur += 1
end
```

Syntaxe BOUCLE FOR :
```
for nombres in 1...10
  puts nombres
end
```

Syntaxe METHODE :
```
def bienvenue
  puts "Bienvenue à Rubyland !"
end
//Syntaxe de base
def par_cinq?(n)
  return n % 5 == 0
end
//Méthode qui return un booleen, mettre un ? à la fin du nom de la méthode
bienvenue
//Appel de la méthode sans paramètre
bienvenue()
//Appel de la méthode avec paramètre
```

Déclaration de variable:
```
mon_nombre = 25
mon_booleen = true
ma_string = "Ruby"
```

Opérateurs :
```
L'addition (+) La soustraction (-)
La multiplication (*) La division (/)
L'exponentiation (**) Le modulo (%)
Conditionnel (||=)
variable ||= "Coucou"
//Variable prend "coucou" si elle est vide
[1, 2, 3] << 4
# ==> [1, 2, 3, 4]
//Opérateur de contaténation
```

Afficher un élément :
```
print "Quel est votre prénom ?"
 //Ecrit sur 1 ligne
ou
puts "Quel est votre prénom ?"
//Ecrit sur plusieurs lignes
# commentaire //Mettre un commentaire
```

Demander la saisit d'un élément et le stocker:
```
nom_variable = gets.chomp
```

Afficher une variable :
"Vous êtes #{variable1} #{variable2}"

Modification d'un String :
variable.upcase! *//Mettre en Majucule*
variable.downcase! *//Mettre en minuscule*
variable.capitalize! *//Mettre la 1ere lettre en Maj*

Opérateur logiques ou booléens :
true && true => true
true && false => false
false && true => false
false && false => false
true || true => true
true || false => true
false || true => true
false || false => false
!false = true
!true = false

Vérification de la présence d'un élément dans un String :
nom_variable.include? "char ou texte à trouver"

Méthode de Substitution Globale (.gsub):
string_a_changer.gsub!(/element_a_remplacer/, "element_qui_remplace")

Intervalles EXCLUSIFS et INCLUSIFS :
```
for nombre in 1..10
  puts nombre
end
//2 points = de 1 à 10
for nombre in 1...10
  puts nombre
end
//3 points = de 1 à 9
```

Méthode LOOP :
```
loop do
  i -= 1
  print "#{i}"
  break if i <= 0
end
// Continuer de faire …
```

Méthode .times :
```
10.times { print "Texte" }
// Répète un nombre de fois donné une instruction
```

Méthode de STRING :
```
texte.split(",")
//commande à Ruby de séparer le string texte à chaque fois qu'il croisera une virgule.
```

TABLEAUX:
```
mon_tableau = [1,2,3,4,5]
// Déclarer un tableau
object.each { |item| # Faire quelque chose }
// Afficher tous les éléments
object.each do |item| # Faire quelque chose end
// Afficher tous les éléments
tableau.each do |x|
  print "#{x}"
end
// Exemple
tableau[2]
//Accéder à un élément du tableau
tableau_2d = [[0,0,0,0],[0,0,0,0]]
//Déclarer un tableau multidimensionnel

tableau.each { |element| puts element }
//itération de toutes les valeurs d'un tableau
nom_tableau.each do | x |
  x.each do | y |
    puts y
  end
end
//itération de toutes les valeurs d'un tableau multidimensionnel
```

HASH :
```
hash = {
  cle1 => valeur1,
  cle2 => valeur2,
  cle3 => valeur3
}
EX:
mon_hash = { "nom" => "Eric",
  "age" => 26,
  "faim?" => true
}
puts mon_hash["nom"]
//Affiche la valeur correspondant à la clé "nom"=>eric
nom_hash = Hash.new
#ou
nom_hash = {}
//Déclarer un hash
animaux["chat"] = "lol"
//Ajouter un élément
puts nom_tableau["nom_cle"]
//Afficher la valeur d'une clé
tableau1.each { |x| puts "#{x}" }
tableau2.each { |x, y| puts "#{x}: #{y}" }
//Afficher toutes les valeurs d'un hash
tableau.each do |cle,val|
   puts cle, matz[cle]
end //itération de toutes les clés et valeurs d'un hash multidimensionnel
.sort_by
//crée une copie qui sera trié par ordre croissant
.reverse!
//inverse la liste d'éléments
films.delete(titre)
//Supprimer un élément
```

Arguments ETOILES :
```
def salut(bonjour, *potes)
  potes.each {|pote| puts "#{bonjour}, #{pote}!" }
end
salut("Salut", "Justin", "Ben", "Kevin Sorbo")
//Dit bonjour à autant de potes passés en paramètre
```

RAPPEL Les blocs sont des méthodes sans noms :
```
1.times do
  puts "Je suis un bloc de code !"
end
//Est égale à :
1.times { puts " Je suis un bloc de code !" }
```

Opérateur de comparaison combinée:
```
val1 <=> val2
//Il retourne :
 0 si le premier opérande est égal au second
1 si le premier opérande est plus grand que le second
-1 si le premier opérande est inférieur au second.
```

Il est important de comprendre que false et nil ne sont pas la même chose : false signifie "faux ou pas-vrai", alors que nil est une façon de dire "rien du tout" en Ruby.

```
no_nil_hash = Hash.new("Il n'y a pas de valeur correspondante à cette clé")
//Créer un nil personnalisé (Valeur donnée à un Hash sans clé)
```

Les SYMBOLES :
La méthode .object_id retourne l'id d'un objet. C'est comme cela que que Ruby sait si deux objets sont exactement les mêmes objets. Lancez le code dans l'éditeur pour voir que les deux "string" sont en fait des objets différents, alors que le .symbol est le même objet.
```
puts "string".object_id //7760200
puts "string".object_id //7759600
puts :symbol.object_id //317928
puts :symbol.object_id //317928
```

Les symboles sont de très bonnes clés de hash, et ce pour plusieurs raisons :

Ils sont immuables, c'est à dire qu'ils ne peuvent pas être modifiés une fois qu'ils ont été créés.
Une seule et unique copie d'un symbole existe à un moment donné, donc ils prennent peu d'espace.
En tant que clé de hash, les symboles sont plus rapides que les strings pour les raisons énoncées ci-dessus.

```
symbol_hash = {
  :un => 1,
  :deux => 2,
  :trois => 3,
}
```

CONVERSION symbole ⇔ string :
```
:yeti.to_s
# ==> "yeti"
"yeti".to_sym
ou
"yeti".intern
# ==> :yeti
```

CONVERSION en INT:
```
"34".to_i
# ==> 34
```

Nouvelle SYNTAXE:
```
films = {
  Dracula: "Pas mal",
  Independance: "Cool"
}
```

SELECTION d'éléments:
```
tab2  = tab2.select {|val1, val2| val2 > 3}
```

EACH_KEY … EACH_VALUE:
```
mon_hash = { un: 1, deux: 2, trois: 3 }
mon_hash.each_key { |c| print c, " " }
# ==> one two three
mon_hash.each_value { |v| print v, " " }
# ==> 1 2 3
```

Syntaxe CASE (SWITCH):
```
case langage
when "JS"
  puts "Sites Web !"
when "Python"
  puts "Science !"
when "Ruby"
  puts "Appli Web !"
else
  puts "Je ne sais pas !"
end
```

UPTO & DOWNTO:
```
"A".upto("Z") { |lettre| puts lettre }
//# Affiche tous l'alphabet
95.upto(100) { |nombre| print nombre, " " }
//# Affiche 95 96 97 98 99 100
100.downto(95) { |nombre| print nombre, " " } //Idem au dessus mais à l'envers
```

APPEL & REPONSE:
```
["texte"].respond_to?(:push)
 // Permet de tester si l'on peut appeler push sur un tableau (exemple)
```

METHODE .collect ou .map:
```
mon_tab = [1, 2, 3]
mon_tab.collect { |num| num ** 2 }
# ==> [1, 4, 9]
```

YIELD :

```
def bloc_test
  puts "Appel de yield"
  yield
  puts "Nous sommes de retour dans la méthode !"
end

bloc_test { puts ">>> Nous sommes dans le bloc !" }
```

// Résultat :
Appel de yield
>>> Nous sommes dans le bloc !
Nous sommes de retour dans la méthode !

```
def yield_nom(nom)
  yield("Kim")
  yield(nom)
  puts "Bloc terminé ! De retour dans la méthode."
end

yield_nom("Eric") { |n| puts "Mon nom est #{n}." }
```

// Résultats:
Mon nom est Kim.
Mon nom est Eric.
Bloc terminé ! De retour dans la méthode.

Code DRY (Don't Repeat Yoursel) PROCS:

- Les procs sont des objets, donc elles ont les mêmes capacités que tous les objets. (Ce que les blocs n'ont pas.)
- Contrairement aux blocs, les procs peuvent être appelées autant de fois que vous en avez besoin sans avoir à les réécrire. Cela vous évite de taper à chaque fois le code pour l'exécuter.

```
multiples_de_3 = Proc.new do |n|
  n % 3 == 0
end
(1..12).to_a.select(&multiples_de_3)
// Résultats: [3, 6, 9, 12]

floats = [1.2, 3.45, 0.91, 7.727, 11.42, 482.911]
arrondi = Proc.new { |x| x.floor }
ints = floats.collect(&arrondi)
// Affiche: [1, 3, 0, 7, 11, 482]

def bonjour
    yield
end
phrase = Proc.new { puts "Bonjour !" }
bonjour(&phrase)
// Affiche: Bonjour !
```

APPEL d'un PROC :

```
salut = Proc.new { puts "Salut !" }
salut.call
```

Conversion SYMBOLE en PROC :

```
tableau_nombres = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
tableau_strings = tableau_nombres.collect(&:to_s)
```

LAMBDA et PROC :

```
LAMBDA { PUTS "BONJOUR !" }
```

Est égale à:

```
Proc.new { puts "Bonjour!" }
```

RAPPEL> Bloc,proc,lambda… :

- Un bloc est juste un morceau de code entre do .. end ou des {}. Ce n'est pas un objet en lui-même, mais on peut le passer à des méthodes telles que .each ou .select.
- Une proc est un bloc sauvegardé que l'on peut utiliser encore et encore...
- Un lambda est une proc, seulement il fait attention au nombre d'arguments qu'on lui passe. De plus il rend le contrôle à la méthode lorsqu'il a terminé de s'exécuter.
- Les blocs, les procs et les lambdas peuvent avoir la même utilité dans de nombreux cas, mais le contexte exact de votre programme vous aidera à choisir le plus adapté à votre situation.

Syntaxe CLASSE :

```
class  nom_classe
  def initialize(param1, param2 …)
    @elem1 = elem1
    @ elem2= elem2
  end
end
```

Instancier un OBJET :

```
mon_objet = nom_classe.new(param1, param2, param3…)
```

Les VARIABLES :

Les variables de classe sont comme des variables d'instance, mais au lieu d'appartenir à une instance d'une classe, elles appartiennent à la classe elle-même. Les variables de classe commencent toujours par deux @, comme ceci : @@fichiers.

Les variables globales peuvent être déclarées de deux façons. La première vous est déjà familière : on définit simplement la variable en dehors d'une méthode ou d'une classe, et that's it ! Elle est globale. Si vous voulez créer une variable globale depuis l'intérieur d'une méthode ou d'une classe, commencez-la simplement par un $, comme ceci : $matz.

Syntaxe HERITAGE :

```
class ClasseDerivee < ClasseDeBase
# Quelque chose !
end
```

Accéder au methodes de la classe MERE :

```
class ClasseDerivee < Base
        def une_methode
                super(arguments optionnels)
                # quelque chose
                end
        end
end
```

Syntaxe CLASSE courte :

```
class Dragon < Creature; end
// Est égale à :
class Dragon < Creature
end
```

Syntaxe METHODE de CLASSE :

```
class Machine
        def Machine.salut
                puts "Salut de la machine !
        end
end
// ou comme cela
```

METHODE public ou privé :

```
class NomDeClasse
  # Des trucs de classe ..
  public
  # Les méthodes publiques vont ici
  def methode_publique; end

  private
  # Les méthodes privées vont ici
  def methode_privee; end
end
```

ACCEDER et MODIFIER une variable :

Nous pouvons utiliser attr_reader pour accéder à une variable et attr_writer pour la modifier

```
class Personne
  attr_reader :nom
  attr_writer :nom
  def initialize(nom)
    @nom= nom
  end
end
```

Ruby s'occupe de rajouter quelque chose comme ça, automatiquement, à notre place :

```
def nom
  @nom
end

def nom=(valeur)
  @nom= valeur
end
```

ATTR (reader + writer) :

```
attr_reader :nom
+
attr_writer :nom
=
attr_accessor :nom
```

Syntaxe des MODULES :

```
module NomDuModule
  # Des choses
end

require 'date'
puts Date.today
//Certains modules ont besoin d'être ajouté avec le require

include Math (ou un autre module)
//EX : Pour pouvoir utiliser directement
PI au lieu de Math::PI
ou
cos au lieu de Math::cos
```

L'un des intérêts majeurs des modules est de séparer des méthodes et des constantes dans des "namespaces" (espaces de nommage). On appelle cela le namespacing),

Opérateur de résolution de portée " :: " :

C'est grâce à cela que le Ruby ne confond pas
Math::PI
Cercle::PI.

INCLUDE et EXTEND :

Le mot-clé include "mixe" les méthodes d'un module au niveau de l'instance (c'est à dire que les instances d'une certaine classe ont le droit d'utiliser les méthodes du module).

Le mot-clé extend permet de "mixer" les méthodes d'un module au niveau de la *classe. C'est à dire que la classe en elle-même peut utiliser les méthodes et pas seulement les instances de cette classe.