

4. Introduction au langage Ruby par des exemples

4.1 Introduction : Pourquoi Ruby?

Ruby, comme Perl et Python, est un langage de «script» à typage dynamique :

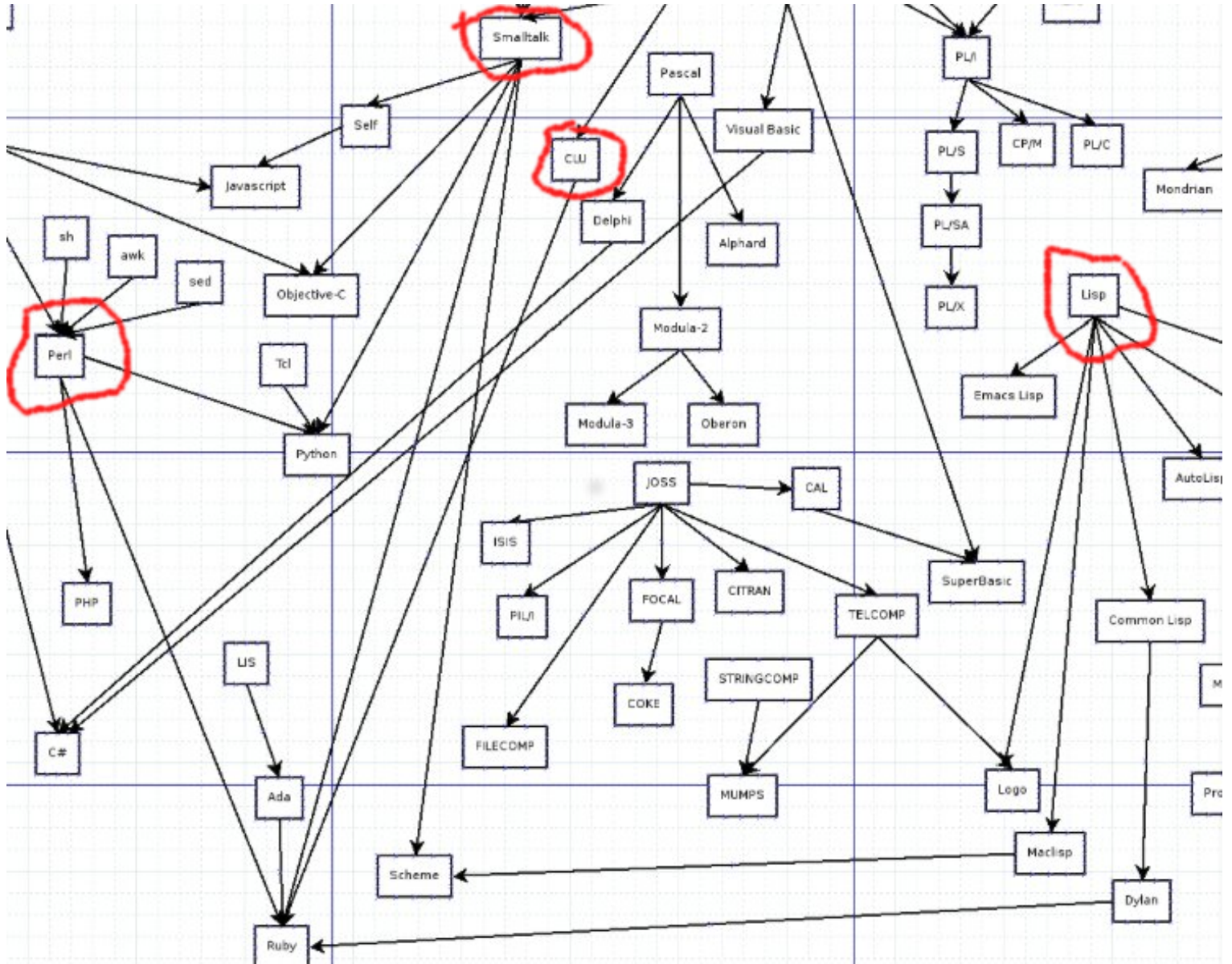
Ruby... est un langage open-source dynamique qui met l'accent sur la simplicité et la productivité. Sa syntaxe élégante en facilite la lecture et l'écriture.

<https://www.ruby-lang.org/fr/>

Conçu, milieu 90, par Yukihiro Matsumoto:

Ruby is “made for developer happiness”!

Y. Matsumoto



Langage	Année	Caractéristiques
Lisp	1958	approche fonctionnelle métaprogrammation
CLU	1974	itérateurs
Smalltalk	1980	langage objet pur, blocs de code GUI, sUnit
Eiffel	1986	<i>Uniform Access Principle</i>
Perl	1987	expressions régulières et <i>pattern-matching</i>
Ruby	1993	

Tableau 4.1: Les ancêtres de Ruby.

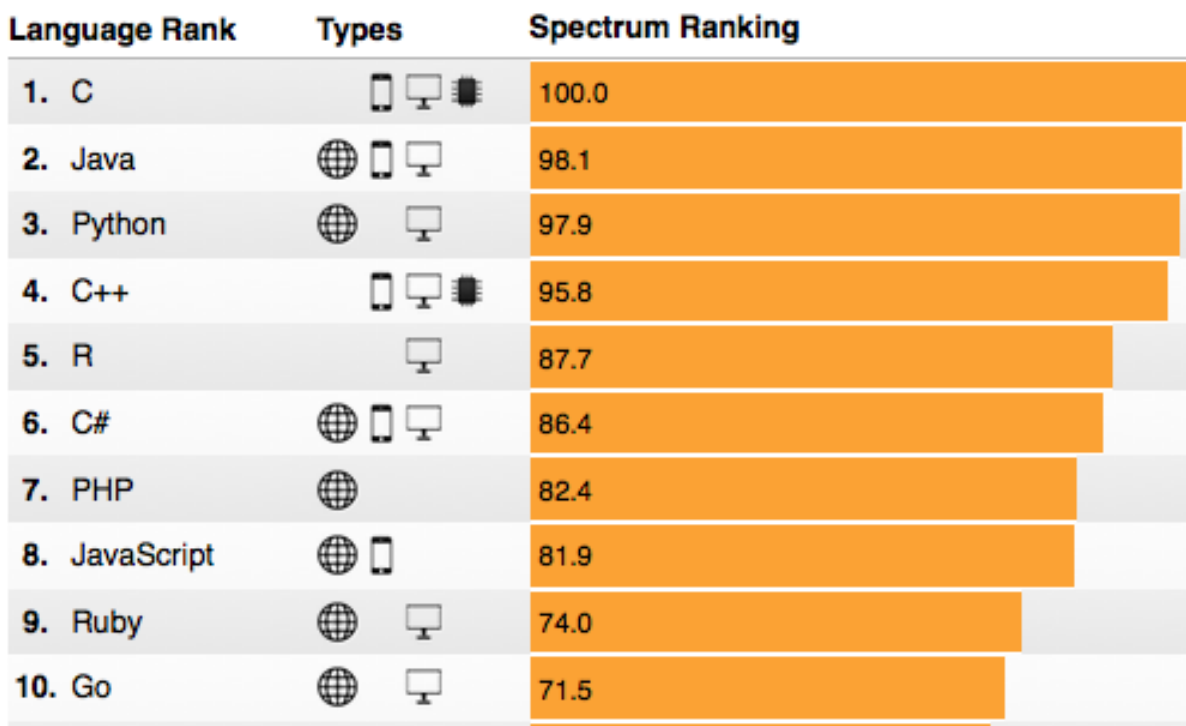


Figure 4.2: Les 10 premières positions du palmarès «*The 2016 Top Programming Languages*» (IEEE Spectrum).

Philosophie de Ruby

*I didn't work hard to make Ruby perfect for everyone, because you feel differently from me. No language can be perfect for everyone. I tried to make Ruby perfect for me, but **maybe it's not perfect for you**. The perfect language for Guido van Rossum is probably Python [peut-être aussi pour vous... ou pas?].*

Yukihiro Matsumoto

Source : <http://www.artima.com/intv/ruby.html>

*Sometimes people jot down pseudo-code on paper. If that pseudo-code runs directly on their computers, it's best, isn't it? **Ruby tries to be like that, like pseudo-code that runs.** Python people say that too.*

Yukihiro Matsumoto

Source : <http://www.artima.com/intv/ruby.html>

Un petit exemple : Somme de deux tableaux

Version Ruby/PRuby

```
def somme_tableaux( a, b )
  c = Array.new(a.size)

  PRuby.pcall( 0...c.size,
               -> (i) { c[i] = a[i] + b[i] }
             )

  c
end
```

Version Java

```
public static int[] sommeTableaux( int a[], int b[] ) {
  int n = a.length;
  int c[] = new int[n];

  Thread threads[] = new Thread[n];
  for( int i = 0; i < n; i++ ) {
    int fi = i;
    threads[i] = new Thread(
      () -> c[fi] = a[fi] + b[fi]
    );
    threads[i].start();
  }

  for( int i = 0; i < n; i++ ) {
    try { threads[i].join(); } catch( Exception e ){};
  }

  return c;
}
```


*Ruby inherited the Perl philosophy of having
more than one way to do the same thing.*

Yukihiro Matsumoto

Source : <http://www.artima.com/intv/rubyP.html>

Mises en oeuvre de Ruby

Dernière version = Ruby 2.4.0 (25 décembre 2016)

Source : <https://www.ruby-lang.org/en/news/2016/12/25/ruby-2-4-0-released/>

Plusieurs mises en oeuvre de Ruby sont disponibles

```
$ rvm list known
# MRI Rubies
[ruby-]1.8.6[-p420]
[ruby-]1.8.7[-head] # security released on head
[ruby-]1.9.1[-p431]
[ruby-]1.9.2[-p330]
[ruby-]1.9.3[-p551]
[ruby-]2.0.0[-p643]
[ruby-]2.1.4
[ruby-]2.1[.5]                # GoRuby
[ruby-]2.2[.1]                goruby
[ruby-]2.2-head              # Topaz
ruby-head                   topaz

# JRuby                      # MagLev
jruby-1.6.8                 maglev[-head]
jruby[-1.7.19]              maglev-1.0.0
jruby-head
jruby-9.0.0.0.pre1          # Mac OS X Snow Leopard Or Newer
                             macruby-0.10
                             macruby-0.11
                             macruby[-0.12]
                             macruby-nightly
                             macruby-head

# Rubinius
rbx-1.4.3
rbx-2.4.1
rbx[-2.5.2]                 # IronRuby
rbx-head                    ironruby[-1.1.3]
                             ironruby-head

# Opal
opal

# Minimalistic ruby implementation - ISO 30170:2012
mruby[-head]

# Ruby Enterprise Edition
ree-1.8.6
ree[-1.8.7] [-2012.02]
```

Figure 4.3: Les mises en oeuvre de Ruby disponibles par l'intermédiaire de `rvm` (février 2016) .

Plusieurs mises en oeuvre sont disponibles :

- MRI (CRuby)
- **JRuby**
- Rubinius
- ...



Figure 4.4: Quelques organisations qui utilisent JRuby.

Source : «*JRuby 9000 Is Out; Now What?*», T. Enebo and C. Nutter, RubyConf 2015, <https://www.youtube.com/watch?v=KifjmbSHHs0>

4.2 Compilation et exécution de programmes Ruby

Exemple Ruby 4.1 Deux versions d'un programme «Hello world!».

```
$ cat hello0.rb  
puts 'Bonjour le monde!'
```

```
$ ruby hello0.rb  
Bonjour le monde!
```

```
$ cat hello1.rb  
#!/usr/bin/env ruby  
  
puts 'Bonjour le monde!'
```

```
$ ls -l hello1.rb  
-rwxr-xr-x. 1 tremblay tremblay 46 26 jun 09:52 hello1.rb*
```

```
$ ./hello1.rb  
Bonjour le monde!
```

4.3 irb : Le *shell* interactif Ruby

- Première façon **facile** pour interagir avec Ruby
— et comprendre comment Ruby fonctionne

- Met en oeuvre un **REPL**
= *Read-Eval-Print Loop*

```
TANTQUE session pas terminée FAIRE
  Lire une expression
  Évaluer l'expression
  Imprimer la valeur de l'expression
FIN
```

Exemple Ruby 4.2 irb, le shell interactif de Ruby.

```
$ irb --prompt=simple
```

```
>> 10
```

```
=> 10
```

```
>> 2 + 4
```

```
=> 6
```

```
>> puts 'Bonjour le monde!'
```

```
Bonjour le monde!
```

```
=> nil
```

```
>> r = puts 'Bonjour le monde!'
```

```
Bonjour le monde!
```

```
=> nil
```

```
>> r
```

```
=> nil
```

```
>> puts( 'Bonjour le monde!' )
```

```
Bonjour le monde!
```

```
=> nil
```

```
>> STDOUT.puts( 'Bonjour le monde!' )
```

```
Bonjour le monde!
```

```
=> nil
```

```
>> STDERR.puts( 'Bonjour le monde!' )
```

```
Bonjour le monde!
```

```
=> nil
```

```
>> STDIN.puts( 'Bonjour le monde!' )
```

```
IOError: not opened for writing
```

```
    from org/jruby/RubyIO.java:1407:in 'write'
```

```
    [...]
```

```
    from /home/tremblay/.rvm/rubies/jruby-1.7.16.1/bin/irb
```

```
    '(root)'
```

```
>> # _ denote la valeur de la derniere expression evaluee.
```

```
>> 8 * 100 / 2
=> 400
```

```
>> _ + _
=> 800
```

```
>> _ / 3
=> 266
```

```
>> _ / 3.0
=> 88.66666666666667
```

```
# On peut creer une nouvelle "session" (interne) qui modifie
# l'objet courant.
```

```
>> irb [10, 20]
```

```
>> self.class
```

```
=> Array
```

```
>> self
```

```
=> [10, 20]
```

```
>> size
```

```
=> 2
```

```
>> irb "abcde"
```

```
>> self
```

```
=> "abcde"
```

```
>> ^D
```

```
=> #<IRB::Irb: @context=#<IRB::Context:0x0000000170a660>,
    @signal_status=:IN_EVAL, @scanner=#<RubyLex:0x0000000191a
```

```
>> self
```

```
=> [10, 20]
```

4.4 Tableaux

Exemple Ruby 4.3 Les tableaux et leurs opérations de base.

```
>> # Valeurs littérales , indexation et taille .
?> a = [10, 20, 30]
=> [10, 20, 30]

>> a[0]
=> 10

>> a[2]
=> 30

>> a[2] = 55
=> 55

>> a
=> [10, 20, 55]

>> a.size
=> 3
```

```
?> # Valeur nil par défaut et extension de la taille.
```

```
?> a[6]
```

```
=> nil
```

```
>> a.size
```

```
=> 3
```

```
>> a[5] = 88
```

```
=> 88
```

```
>> a.size
```

```
=> ??
```

```
>> a
```

```
=> ??
```

```
?> # Valeur nil par défaut et extension de la taille.  
?> a[6]  
=> nil  
  
>> a.size  
=> 3  
  
>> a[5] = 88  
=> 88  
  
>> a.size  
=> 6  
  
>> a  
=> [10, 20, 55, nil, nil, 88]
```

```
?> # Acces au 'dernier' element.  
?> a[a.size-1]  
=> 88
```

```
>> a[-1]  
=> 88
```

Autre forme de commentaire :

```
=begin
```

```
Blah  blah
```

```
. . .
```

```
=end
```

Exemple Ruby 4.4 Les tableaux et leurs opérations de base (suite 1).

```
?> # Tableaux heterogenes.
```

```
?> a
```

```
=> [10, 20, 55, nil, nil, 88]
```

```
>> a[8] = 'abc'
```

```
=> "abc"
```

```
>> a
```

```
=> [10, 20, 55, nil, nil, 88, nil, nil, "abc"]
```

```
?> # Ajout d'elements.
```

```
?> a = []
```

```
=> []
```

```
>> a << 12
```

```
=> [12]
```

```
>> a << 'abc' << [2.7, 2.8]
```

```
=> [12, "abc", [2.7, 2.8]]
```

```
?> # Creation de tableaux avec valeurs initiales.  
?> b = Array.new(3) { 10 }  
=> [10, 10, 10]
```

```
>> d = Array.new(4)  
=> [nil, nil, nil, nil]
```

Exemple Ruby 4.5 Les tableaux et leurs opérations de base (suite 2).

```
?> # Tranches de tableaux.  
?> a = [10, 20, 30, 40, 50]  
=> [10, 20, 30, 40, 50]  
  
>> a[0..2]  
=> [10, 20, 30]  
  
>> a[3..3]  
=> ??  
  
>> a[1..-1]  
=> ??  
  
>> a[7..7]  
=> ??
```

```
?> # Tranches de tableaux.  
?> a = [10, 20, 30, 40, 50]  
=> [10, 20, 30, 40, 50]  
  
>> a[0..2]  
=> [10, 20, 30]  
  
>> a[3..3]  
=> [40]  
  
>> a[1..-1]  
=> [20, 30, 40, 50]  
  
>> a[7..7]  
=> nil
```

```
?> # Intervalles inclusifs vs. exclusifs
```

```
>> a
```

```
=> [10, 20, 30, 40, 50]
```

```
>> a[1..3]
```

```
=> [20, 30, 40]
```

```
>> a[1...3]
```

```
=> [20, 30]
```

```
>> a[1..a.size-1]
```

```
=> [20, 30, 40, 50]
```

```
>> a[1...a.size]
```

```
=> [20, 30, 40, 50]
```

4.5 Chaînes de caractères

Exemple Ruby 4.6 Les chaînes de caractères et leurs opérations de base.

```
>> # String semblable a Array.  
?> s1 = 'abc'  
=> "abc"  
  
>> s1.size  
=> 3  
  
>> s1[0..1] # Retourne String.  
=> "ab"  
  
>> s1[2]     # Retourne String aussi!  
=> "c"
```

```
?> # Concatenation vs. ajout.
```

```
?> s1 + 'def'
```

```
=> "abcdef"
```

```
>> s1
```

```
=> "abc"
```

```
>> s1 << 'def'
```

```
=> "abcdef"
```

```
>> s1
```

```
=> "abcdef"
```

Exemple Ruby 4.7 Les chaînes de caractères et leurs opérations de base (suite).

```
>> # Egalite de valeur *sans* partage de reference .
?> a, b = 'abc', 'abc'
=> ["abc", "abc"]

>> a == b
=> true

>> a.equal? b
=> false

>> a[0] = 'X'
=> "X"

>> a
=> "Xbc"

>> b
=> "abc"
```

```
?> # Egalite de valeur *avec* partage de reference.  
?> a = b = 'abc'  
=> "abc"  
  
>> a == b  
=> true  
  
>> a.equal? b  
=> true  
  
>> a[0] = 'X'  
=> "X"  
  
>> a  
=> "Xbc"  
  
>> b  
=> "Xbc"
```

Exemple Ruby 4.8 Interpolation d'une expression dans une chaîne.

```
>> # Interpolation d'une expression dans une chaîne.  
?> x = 123  
=> 123  
  
?> "abc \"#{x}\" def"  
=> "abc \"123\" def"  
  
?> "abc '#{10 * x + 1}' def"  
=> "abc '1231' def"  
  
?> "abc #{x > 0 ? '++' : 0} def"  
=> "abc ++ def"  
  
?> # String definie avec '...' => pas d'interpolation.  
?> 'abc "#{x}" def'  
=> "abc \"\#{x}\" def"
```

Exemple Ruby 4.9 Opérations `split` et `join`.

```
# Split decompose une chaine en sous-chainnes
# en fonction du <<motif>> specifie en argument.

>> s = "abc\ndef\nghi\n"
=> "abc\ndef\nghi\n"

>> s.split("\n")                # Un cas typique!
=> ["abc", "def", "ghi"]

>> s.split("def")
=> ["abc\n", "\nghi\n"]

>> s.split("a")
=> ["", "bc\ndef\nghi\n"]

>> s.split(/\w{3}/)             # \w = [a-zA-Z0-9_]
=> ["", "\n", "\n", "\n"]
```

```
# Join combine un tableau de sous-chaines  
# en une chaine unique.
```

```
>> s  
=> "abc\ndef\nghi\n"
```

```
>> r = s.split("\n")  
=> ["abc", "def", "ghi" ]
```

```
>> r.join("+")  
=> "abc+def+ghi"
```

```
>> r.join("\n") # Donc: s.split("\n").join("\n") != s  
=> "abc\ndef\nghi"
```

```
>> [].join(";")  
=> ""
```

```
>> ['abc'].join(";")  
=> "abc"
```

```
>> ['abc', 'def'].join(";")  
=> "abc;def"
```

4.6 Symboles

- Symbol

= objet associé à un **identificateur** — représentation unique dans la table des symboles

= possède une **représentation** sous forme

- d'un **entier** ($\approx$ adresse)

- d'une **chaine**

... mais n'est ni un entier, ni une chaine!

Exemple Ruby 4.10 Les symboles.

```
>> # Symbole = "sorte" de chaine *unique et immuable*.
>> :abc
=> :abc
```

```
>> :abc.class
=> Symbol
```

```
>> :abc.to_s
=> "abc"
```

```
>> puts :abc
abc
=> nil
```

```
>> :abc[2]
=> "c"
```

```
>> :abc[2] = "x"
NoMethodError: undefined method '[]=' for :abc:Symbol
    from (irb):4
    from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:11:in`
```

```
>> "abc".to_sym
=> :abc
```

```
>> "abc def .!#%".to_sym
=> : "abc def .!#%"
```

```
?> # Possede un numero d'identification unique.  
>> :a  
=> :a
```

```
>> :a.object_id  
=> 365128
```

```
>> "a".object_id  
=> 11000000
```

```
>> "a".object_id  
=> 10996280
```

```
>> :a.object_id  
=> 365128
```

```
>> "a".to_sym.object_id  
=> 365128
```

```
?> # Egalite de valeur vs. de reference.
```

```
?>
```

```
>> :abc == :abc
```

```
=> true
```

```
>> :abc.equal? :abc
```

```
=> true
```

```
>> "abc" == "abc"
```

```
=> true
```

```
>> "abc".equal? "abc"
```

```
=> false
```

```
>> "abc".to_sym == :abc
```

```
=> true
```

```
>> "abc".to_sym.equal? :abc
```

```
=> true
```

4.7 Hashes

Objet `Hash` $\approx$ forme généralisée de tableau

- l'index (la clé) est un objet arbitraire, pas juste un entier
- Autres noms : Dictionnaires, *maps*

Exemple Ruby 4.11 Les *hashes* et leurs opérations de base.

```
>> # Definition d'un hash.
?> hash = { :abc => 3, :de => 2, :ghijk => 5 }
=> {:abc=>3, :de=>2, :ghijk=>5}

?> # Principales proprietes.
?> hash.size
=> 3

>> hash.keys
=> [:abc, :de, :ghijk]

>> hash.values
=> [3, 2, 5]

>> # Indexation.
?> hash[:abc]
=> ??

>> hash[:de]
=> ??

>> hash["de"]
=> ??
```

```
>> # Definition d'un hash.
?> hash = { :abc => 3, :de => 2, :ghijk => 5 }
=> {:abc=>3, :de=>2, :ghijk=>5}

?> # Principales proprietes.
?> hash.size
=> 3

>> hash.keys
=> [:abc, :de, :ghijk]

>> hash.values
=> [3, 2, 5]

>> # Indexation.
?> hash[:abc]
=> 3

>> hash[:de]
=> 2

>> hash["de"]
=> nil
```

Exemple Ruby 4.12 Les *hashes* et leurs opérations de base (suite).

```
?> # Definition d'une nouvelle cle.
```

```
?> hash.include? "de"
```

```
=> false
```

```
>> hash["de"] = 55
```

```
=> 55
```

```
>> hash.include? "de"
```

```
=> true
```

```
?> # Redefinition d'une cle existante.
```

```
?> hash[:abc] = 2300
```

```
=> 2300
```

```
>> hash
```

```
=> {:abc=>2300, :de=>2, :ghijk=>5, "de"=>55}
```

Exemple Ruby 4.13 Les *hashes* et leurs opérations de base (suite) : Création et initialisation.

```
?> # Creation d'un Hash sans valeur par default.  
?> h1 = {} # Idem: h1 = Hash.new  
=> {}
```

```
>> h1[:xyz]  
=> nil
```

```
>> # Creation d'un Hash avec valeur par default.  
?> h2 = Hash.new( 0 )  
=> {}
```

```
>> h2[:xyz]  
=> 0
```

```
>> h2[:abc] += 1  
=> 1
```

```
>> # Creation d'un Hash avec valeur par default.  
    # Attention: La valeur est *partagee*  
    # par toutes les cles!  
?> h3 = Hash.new( [] )  
=> {}
```

```
>> p h3[:x], h3[:y]  
[]  
[]  
=> [ [], [] ]
```

```
>> h3[:x] << "abc"  
=> ["abc"]  
>> p h3[:x], h3[:y]  
["abc"]  
["abc"]  
=> [ ["abc"], ["abc"] ]
```

```
>> # Creation d'un Hash avec valeur par default ,  
    # definie via un bloc pour avoir  
    # une nouvelle valeur a chaque fois.  
>> h4 = Hash.new { |h, k| h[k] = [] }  
=> {}
```

```
>> p h4[:x], h4[:y]  
[]  
[]  
=> [ [], [] ]
```

```
>> h4[:x] << "abc"  
=> ["abc"]  
>> p h4[:x], h4[:y]  
["abc"]  
[]  
=> [ ["abc"], [] ]
```

4.8 Expressions booléennes

Le point important à retenir pour comprendre les expressions booléennes :

- `false` et `nil` sont des valeurs «fausses»
- Toute autre valeur est «vraie».

Quelques exemples avec l'opérateur ternaire `?:` :

```
>> false ? 'oui' : 'non'
=> "non"
```

```
>> nil ? 'oui' : 'non'
=> 'non'
```

```
>> 0 ? 'oui' : 'non'
=> "oui"
```

```
>> '' ? 'oui' : 'non'
(irb):5: warning: string literal in condition
=> "oui"
```

```
>> nil.nil? ? 'nil' : 'pas nil'
=> "nil"
```

Exemple Ruby 4.14 Les expressions booléennes.

```
>> # Toute valeur differente de false ou nil est vraie.
?> true ? 'oui' : 'non'
=> "oui"

>> 0 ? 'oui' : 'non'
=> "oui"

>> [] ? 'oui' : 'non'
=> "oui"

?> # Seuls false et nil ne sont pas vraies.
?> false ? 'oui' : 'non'
=> "non"

>> nil ? 'oui' : 'non'
=> "non"

>> !false ? 'oui' : 'non'
=> "oui"

>> !nil ? 'oui' : 'non'
=> "oui"

?> # Seul nil est nil
?> 2.nil? ? 'nil' : 'pas nil'
=> "pas nil"

>> [].nil? ? 'nil' : 'pas nil'
=> "pas nil"

>> nil.nil? ? 'nil' : 'pas nil'
=> "nil"
```

Exemple Ruby 4.15 Les expressions booléennes (suite 1).

```
?> # Les expressions && et || sont court-circuitées.  
?> true || (3 / 0) ? true : false  
=> true  
  
>> false && (3 / 0) ? true : false  
=> false  
  
?> false || (3 / 0) ? true : false  
ZeroDivisionError: divided by 0  
[...]  
  
>> true && (3 / 0) ? true : false  
ZeroDivisionError: divided by 0  
[...]
```

Exemple Ruby 4.16 Les expressions booléennes (suite 2).

```
>? # L'opérateur || retourne la première expression  
    # 'non fausse', sinon retourne la dernière expression.
```

```
?> 2 || 3
```

```
=> ??
```

```
>> nil || false || 2 || false
```

```
=> ??
```

```
>> nil || false
```

```
=> ??
```

```
>> false || nil
```

```
=> ??
```

```
>? # L'opérateur || retourne la première expression
    # 'non fausse', sinon retourne la dernière expression.
?> 2 || 3
=> 2

>> nil || false || 2 || false
=> 2

>> nil || false
=> false

>> false || nil
=> nil
```

Exemple Ruby 4.17 Les expressions booléennes (suite 3).

```
# On peut utiliser ||= pour initialiser une variable ,  
# sauf si elle est déjà initialisée.
```

```
>> x
```

```
NameError: undefined local variable or method 'x' for main:Object  
[...]
```

```
>> x ||= 3
```

```
=> 3
```

```
>> x
```

```
=> 3
```

```
>> x ||= 8
```

```
=> 3
```

```
>> x
```

```
=> 3
```

Abréviations :

```
x += 1      # x = x + 1
```

```
x /= 2      # x = x / 2
```

```
x ||= 1     # x || x = 1   ET   non pas x =
```

```
x &&= 1     # x && x = 1   Et   non pas x =
```

4.9 Définitions et appels de méthodes

Exemple Ruby 4.18 Définitions et appels de méthodes.

```
>> # Definition et appels de methode.
    def add( x, y )
      x + y
    end

>> add( 2, 3 )
=> 5

>> add 20, 30      # Les parentheses sont optionnelles.
=> 50

>> # Resultat = derniere expression evaluee.
    def abs( x )
      if x < 0 then -1 * x else x end
    end

>> abs( 3 )
=> 3

>> abs( -3 )
=> 3
```

```
>> # On utilise return pour sortir 'avant la fin'.
    def abs2( x )
        return x if x >= 0

        -x
    end

>> abs2( 23 )
=> 23
>> abs2( -23 )
=> 23
```

Remarque : «;» est un **séparateur**!

```
def add( x, y ); x + y; end
```

Remarque : «return» est **implicite**

```
def add( x, y )  
  x + y  
end
```

```
def add( x, y )  
  return x + y  
end
```

Attention : Parenthèses et appels de méthodes :

add(2, 3) # OK 😊

add 2, 3 # OK 😊

add (2, 3) # P a s OK 😞

add (1+1), (2+1) # OK 😊

Exemple Ruby 4.19 Appels de méthodes et envois de messages.

```
>? # Un opérateur est une méthode.
```

```
?> 2 + 3
```

```
=> 5
```

```
>> 2.+( 3 )
```

```
=> 5
```

```
>> 2.+ 3
```

```
=> 5
```

```
>? # Un appel de méthode est un envoi de message.
```

```
>? 2.+( 3 )
```

```
=> 5
```

```
>> 2.send( :+, 3 )
```

```
=> 5
```

4.10 Structures de contrôle

Exemple Ruby 4.20 Structures de contrôles: if.

```
>> # Instruction conditionnelle classique.
def div( x, y )
  if y == 0
    fail "Oops! Division par zero :("
  else
    x / y
  end
end

>> div( 12, 3 )
=> 4

>> div( 12, 0 )
RuntimeError: Oops! Division par zero :(
  from (irb):4:in 'div'
  [...]
  from /home/tremblay/.rvm/rubies/jruby-1.7.16.1/bin/irb
```

```
>> # Garde (condition) if associee a une instruction.
def div( x, y )
    fail "Oops! Division par zero :(" if y == 0

    x / y
end

>> div( 12, 3 )
=> 4
```

Exemple Ruby 4.21 Structures de contrôles: `while`.

```
?> # Instruction while.
def pgcd( a, b )
  # On doit avoir a <= b.
  return pgcd( b, a ) if a > b

  while b > 0
    a, b = b, a % b
  end

  a
end

>> pgcd( 12, 8 )
=> 4
>> pgcd( 80, 120 )
=> 40
```

Affectations multiples (parallèles) :

```
x , y = y , x
```

```
x , y , z = [10 , 20 , 30]
```

```
# x == 10 && y == 20 && z == 30
```

```
x , y = [10 , 20 , 30]
```

```
# x == 10 && y == 20
```

```
x , *y = [10 , 20 , 30]
```

```
# x == 10 && y == [20 , 30]
```

Exemple Ruby 4.22 Structures de contrôles : Itération sur les index avec `for` et `each_index`.

```
?> # Instruction for
def somme( a )
  total = 0
  for i in 0...a.size
    total += a[i]
  end

  total
end

>> somme( [10, 20, 30] )
=> 60
```

```
?> # Iterateur each_index.
def somme( a )
  total = 0
  a.each_index do |i|
    total += a[i]
  end

  total
end

>> somme( [10, 20, 30] )
=> 60
```

Exemple Ruby 4.23 Structures de contrôles : Itération sur les éléments avec `for` et `each`.

```
?> # Instruction for (bis)
```

```
def somme( a )  
  total = 0  
  for x in a  
    total += x  
  end
```

```
  total  
end
```

```
>> somme( [10, 20, 30] )
```

```
=> 60
```

```
?> # Iterateur each.
```

```
def somme( a )  
  total = 0  
  a.each do |x|  
    total += x  
  end
```

```
  total  
end
```

```
>> somme( [10, 20, 30] )
```

```
=> 60
```

4.11 Paramètres des méthodes

Exemple Ruby 4.24 Paramètres des méthodes : valeur par défaut et nombre variable d'arguments.

```
?> # Argument optionnel et valeur par défaut.  
    def foo( x, y = 40 )  
      x + y  
    end
```

```
>> foo( 3, 8 )  
=> ??
```

```
>> foo( 3 )  
=> ??
```

```
?> # Argument optionnel et valeur par défaut.
```

```
def foo( x, y = 40 )  
  x + y  
end
```

```
>> foo( 3, 8 )
```

```
=> 11
```

```
>> foo( 3 )
```

```
=> 43
```

```
>> # Nombre variable d'arguments.  
def bar( x, *args, y )  
    "bar( #{x}, #{args}, #{y} )"  
end
```

```
>> bar( 1, 2, 3, 4, 5 )  
=> ??
```

```
>> bar( 1, 2 )  
=> ??
```

```
>> bar( 23 )  
??  
??  
??  
??
```

```
>> # Nombre variable d'arguments.  
def bar( x, *args, y )  
  "bar( #{x}, #{args}, #{y} )"  
end
```

```
>> bar( 1, 2, 3, 4, 5 )  
=> "bar( 1, [2, 3, 4], 5 )"
```

```
>> bar( 1, 2 )  
=> "bar( 1, [], 2 )"
```

```
>> bar( 23 )
```

```
ArgumentError: wrong number of arguments (1 for 2+)
```

```
from (irb):24:in 'bar'
```

```
from (irb):27
```

```
from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb
```

Exemple Ruby 4.25 Paramètres des méthodes : arguments par mots-clés (*keyword arguments*).

```
>> # Arguments par mot-clés (keyword arguments).  
    def diviser( numérateur:, denominateur: 1 )  
      numérateur / denominateur  
    end
```

```
>> diviser numérateur: 12, denominateur: 3  
=> 4
```

```
>> diviser denominateur: 3, numérateur: 12  
=> 4
```

```
>> diviser numérateur: 12  
=> 12
```

```
>> diviser 10
```

```
ArgumentError: missing keyword: numérateur  
    from (irb):31
```

```
    from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:1
```

```
?> # Argument par mot-cle.
def premier_index( a, x, res_si_absent: nil )
  a.each_index do |i|
    return i if a[i] == x
  end
  res_si_absent
end

>> premier_index( [10, 20, 10, 20], 10 )
=> 0
>> premier_index( [10, 20, 10, 20], 88 )
=> nil
>> premier_index( [10, 20, 10, 20], 88, res_si_absent: -1 )
=> -1
```

Autre façon, mais moins claire :

```
def premier_index( a, x, res_si_absent = n
    ...
end
```

```
premier_index( [10, 20, 10, 20], 88, -1 )
```

Soit la méthode suivante :

```
def foo( x, y, z = nil )  
  return x + y * z if z  
  
  x * y  
end
```

Indiquez ce qui sera affiché par chacun des appels suivants :

```
# a.  
puts foo( 2, 3 )  
  
# b.  
puts foo 2, 3, 5  
  
# c.  
puts foo( "ab", "cd", 3 )  
  
# d.  
puts foo( "ab", "cd" )
```

Exercice 4.1: Définition et utilisation de méthodes diverses avec plusieurs sortes d'arguments.

Soit la méthode suivante :

```
def bar( v = 0, *xs )  
  m = v  
  xs.each do |x|  
    m = [m, x].max  
  end  
  
  m  
end
```

Indiquez ce qui sera affiché par chacun des appels suivants :

```
# a.  
puts bar  
  
# b.  
puts bar( 123 )  
  
# c.  
puts bar( 0, 10, 20, 99, 12 )
```

Exercice 4.2: Définition et utilisation de méthodes diverses avec plusieurs sortes d'arguments.

Soit le segment de code suivant :

```
def foo( x, *y, z = 10 )  
    x + y.size + z  
end
```

```
puts foo( 10, 20, 30 )
```

Qu'est-ce qui sera affiché?

Exercice 4.3: Définition d'une méthode avec plusieurs sortes d'arguments.

4.12 Définitions de classes

Exemple Ruby 4.26 Un script avec une classe (simple) pour des cours.

```
$ cat cours.rb
# Definition d'une classe (simple!) pour des cours.
class Cours
  attr_reader :sigle

  def initialize( sigle, titre, *prealables )
    @sigle = sigle
    @titre = titre
    @prealables = prealables
  end

  def to_s
    sigles_prealables = " "
    @prealables.each do |c|
      sigles_prealables << "#{c.sigle} "
    end

    "< #{@sigle} '#{@titre}' (#{sigles_prealables}) >"
  end
end
```

... (suite du fichier page suivante)...

Exemple Ruby 4.26 Un script avec une classe (simple) pour des cours (suite).
Note : «\$0» = nom du programme Ruby en cours d'exécution.

```
if $0 == __FILE__
  # Definition de quelques cours.
  inf1120 = Cours.new( :INF1120, 'Programmation I' )
  inf1130 = Cours.new( :INF1130, 'Maths pour informaticien'
  inf2120 = Cours.new( :INF2120, 'Programmation II',
                        inf1120 )
  inf3105 = Cours.new( :INF3105, 'Str. de don.',
                        inf1130, inf2120 )

  puts inf1120
  puts inf3105
  puts inf1120.sigle
  puts inf1120.titre
end
```

Exemple Ruby 4.27 Appel du script avec une classe pour des cours.

```
$ ruby cours.rb
< INF1120 'Programmation I' ( ) >
< INF3105 'Str. de don.' ( INF1130 INF2120 ) >
INF1120
NoMethodError: undefined method 'titre' for #<Cours:0x13969f
  (root) at cours.rb:34
```

Une «*déclaration*» «`attr_reader :sigle`» définit un attribut accessible en lecture, équivalent à la méthode suivante :

```
def sigle
  @sigle
end
```

Note : En fait, «`attr_reader :sigle`» représente un appel à la méthode `attr_reader` avec l'argument `:sigle`. Voir plus loin.

Pour la classe `Cours`, définissez une méthode qui permet d'obtenir le titre d'un cours et une autre méthode qui permet de **modifier** le titre d'un cours.

Utilisez ensuite cette dernière méthode pour changer le titre du cours `inf1120` en "Programmation Java I".

Exercice 4.4: Méthodes pour lire et modifier le titre d'un cours.

4.13 Lambda-expressions

Les lambda-expressions — λ -expressions — sont le fondement de la programmation **fonctionnelle**.



Exemple Ruby 4.28 Les lambda-expressions : type et méthodes de base.

```
>> # Une lambda-expression représente un objet ,  
    # de classe Proc , qu'on peut 'appeler'.  
    # Un Proc est donc une "fonction anonyme".  
?> lambda { 0 }.call  
=> 0
```

```
>> zero = lambda { 0 }  
=> #<Proc:0x5c5eefef@(irb):2 (lambda)>
```

```
>> zero.class  
=> Proc
```

```
>> zero.arity    # Lambda avec 0 argument!  
=> 0  
>> zero.parameters  
=> []
```

```
>> zero.call  
=> 0
```

```
?> # Une lambda-expression peut avoir des arguments.
?> inc = lambda { |x| x + 1 }
=> #<Proc:0x16293aa2@(irb):8 (lambda)>

>> inc.arity
=> 1

>> inc.parameters
=> [[:req, :x]]

>> inc.call( 3 )
=> 4

# Et le nombre d'arguments est verifie!
>> inc.call
ArgumentError: wrong number of arguments (0 for 1)
      from [...]
>> inc.call( 10, 20 )
ArgumentError: wrong number of arguments (2 for 1)
      from [...]
```

```
?> double = lambda do |y|
?>   y + y
>> end
=> #<Proc:0x5158b42f@(irb):11 (lambda)>

>> double.arity
=> 1

>> double.call( 3 )
=> 6
```

Exemple Ruby 4.29 Les lambda-expressions, comme n'importe quel autre objet, peuvent être transmises en argument.

```
>> # Une methode pour executer deux fois du code (sans arg.)
    def deux_fois( f )
      f.call
      f.call
    end
```

```
>> deux_fois( lambda { print 'Bonne '; print 'journee!\n' }
Bonne journee!
Bonne journee!
=> nil
```

```
>> deux_fois lambda { print 'Bonne '; print 'journee!\n' }
Bonne journee!
Bonne journee!
=> nil
```

```
?> # Ici, les () sont obligatoires, sinon erreur de syntaxe
?>   deux_fois( lambda do
                    print 'Bonne '
                    print 'journee!\n'
                  end )

Bonne journee!
Bonne journee!
=> nil
```

Exemple Ruby 4.30 Les lambda-expressions, comme n'importe quel objet, peuvent être retournées comme résultat d'une fonction.

```
?> # Une lambda-expression peut etre retournée comme resultat
?> def plus_x( x )
      lambda { |y| x + y }
    end

>> plus_x(3).call(12)
=> 15

?> plus_bis = lambda { |a| lambda { |b| a + b } }
=> #<Proc:0x2d7275fc@(irb):44 (lambda)>

>> plus_bis.call(3).call(12)
=> 15
```

Exemple Ruby 4.31 Le bloc d'une lambda-expression capture les variables non-locales.

```
?> # Le bloc d'une lambda-expression 'capture'
    # les variables non-locales utilisees dans le bloc.
?> x = 23
=> 23

>> plus_x = lambda { |y| x + y }
=> #<Proc:0x72d1ad2e@(irb):31 (lambda)>

>> plus_x.call(7)
=> 30

>> x = 999
=> 999

>> plus_x.call 2
=> 1001
```

Exemple Ruby 4.32 Les appels à une lambda-expression peuvent aussi être faits avec «.()» plutôt qu’avec `call` — mais c’est rarement utilisé!

```
>> lambda { 0 }.()  
=> 0
```

```
>> zero = lambda { 0 }  
=> #<Proc:0x5c5eefef@(irb):2 (lambda)>
```

```
>> zero.()  
=> 0
```

```
>> inc = lambda { |x| x + 1 }  
=> #<Proc:0x16293aa2@(irb):8 (lambda)>
```

```
>> inc.( 3 )  
=> 4
```

Pour la classe `Cours` :

- a. Définissez une méthode `prealables` qui reçoit en argument un `predicat` — une lambda-expression — et qui retourne la liste des préalables du cours qui satisfont ce `predicat`.
- b. Utilisez la méthode `prealables` pour obtenir les préalables du cours `inf3105` dont le sigle contient la chaîne "INF".

Remarque : Pour ce dernier point, vous devez utiliser une expression de *pattern-matching*. En Ruby, l'expression suivante retourne un résultat *non nil* si `x`, une chaîne, matche le motif `INF` :

```
/INF/ =~ x
```

Plus précisément, l'expression retourne `nil` si le motif n'apparaît pas dans la chaîne, sinon elle retourne la **position** du premier *match*.

Exercice 4.5: Une méthode pour identifier un sous-ensemble de préalables d'un cours.

4.14 Blocs

Un **bloc** est un **segment de code** entre accolades `{...}` ou entre `do...end` :

```
a.each { |x| total += x }
```

```
a.each_index do |i|  
  total += a[i]  
end
```

```
inc = lambda { |x| x + 1 }
```

```
double = lambda do |y|  
  y + y  
end
```

Un **bloc** est un **segment de code** entre accolades `{...}` ou entre `do...end` :

```
a.each { |x| total += x }
```

```
a.each_index do |i|  
  total += a[i]  
end
```

```
inc = lambda { |x| x + 1 }
```

```
double = lambda do |y|  
  y + y  
end
```

Mais plus important :

A block is a chunk of code that can be passed to an object/method and [can be] executed under the context of that object.

<https://scotch.io/tutorials/understanding-ruby-closures>

Un bloc est **semblable** à une lambda-expression, mais pas tout à fait identique :

Lambda-expression	Bloc
Objet (Proc) créé de façon explicite	Utilisée comme argument implicite d'une méthode
Évalué avec call	Évalué avec yield
	Peut être transformé en Proc (argument explicite d'une méthode)

L'utilisation des blocs en Ruby est étroitement liée à l'instruction `yield`.

Quelques définitions du verbe anglais «*to yield*» :

- *to produce (something) as a result of time, effort, or work*
- *to surrender or relinquish to the physical control of another : hand over possession of*

Quelques traductions françaises possibles du verbe «*to yield*» sont «céder» ou «**produire**».

yield, **exécutée dans une méthode**, a l'effet suivant :

- elle évalue **le bloc** passé en argument
- mais...
- ce bloc peut **ne pas apparaître** dans la liste des arguments

Exemple Ruby 4.33 Une méthode pour exécuter deux fois un bout de code — avec un *bloc*.

```
>> # Une autre methode pour executer deux_fois du code , avec
    def deux_fois
      yield
      yield
    end
```

```
>> deux_fois { print 'Bonne '; print 'journee!\n' }
Bonne journee!
Bonne journee!
=> nil
```

```
>> deux_fois do
      print 'Bonne '
      print 'journee!\n'
    end
Bonne journee!
Bonne journee!
=> nil
```

```
>> deux_fois
LocalJumpError: no block given (yield)
      from (irb):1:in 'deux_fois'
      from (irb):3
      from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb
```

```
>> # Methode pour executer k fois du code.  
def k_fois( k )  
    k.times do  
        yield  
    end  
end
```

```
>> k_fois( 3 ) do  
    print 'Bonne '  
    print 'journee!\n'  
end
```

```
Bonne journee!  
Bonne journee!  
Bonne journee!
```

Exemple Ruby 4.34 Une méthode pour évaluer une expression — avec `lambda`, avec *bloc implicite* et avec *bloc explicite*.

```
>> # Methode pour evaluer une expression: avec lambda.  
def evaluer( x, y, expr )  
  expr.call( x, y )  
end
```

```
>> evaluer( 10, 20, lambda { |v1, v2| v1 + v2 } )  
=> 30
```

```
>> # Methode pour evaluer une expression: avec bloc implicite  
def evaluer( x, y )  
  yield( x, y )  
end
```

```
>> evaluer( 10, 20 ) { |a, b| a * b }  
=> 200
```

```
>> # Methode pour evaluer une expression: avec bloc explicite
    def evaluer( x, y, &expr )
      expr.call( x, y )
    end

>> evaluer( 10, 20 ) { |a, b| b / a }
=> 2
```

```
>> # On peut verifier si un bloc a ete passe ou non.
def evaluer( x, y )
  return 0 unless block_given?
  yield( x, y )
end

>> evaluer( 10, 20 ) { |a, b| b / a }
=> 2
>> evaluer( 10, 20 )
=> 0
```

```
>> def foo( &b )  
      [b.class, b.arity, b.parameters] if block_given?  
    end  
=> :foo  
  
>> foo  
=> nil  
  
>> foo { 2 }  
=> [Proc, 0, []]  
  
>> foo { |x| x + 1 }  
=> [Proc, 1, [[:opt, :x]]]
```

Remarque concernant les deux formes de bloc :

Au niveau **sémantique**, les deux formes de blocs — avec accolades `{... }` et avec `do... end` — sont équivalentes.

Il existe toutefois une différence au niveau de la **priorité** lors de l'analyse syntaxique :

Avec accolades => priorité + forte

```
foo bar { ... }
```

Équivalent

```
foo( bar() { ... } )
```

Avec accolades => priorité - forte

```
foo bar do ... end
```

Équivalent

```
foo( bar() ) { ... }
```

4.15 Portée des variables

sigil (Ésotérisme) Symbole graphique ou sceau représentant une intention ou un être magique.

Source : <https://fr.wiktionary.org/wiki/sigil>

Ruby utilise un certain nombre de *sigils* pour indiquer la portée des variables :

<code>foo</code>	variable locale
<code>@foo</code>	variable d'instance
<code>@@foo</code>	variable de classe
<code>\$foo</code>	variable globale

(Même question que la précédente, mais en utilisant un bloc.)

Pour la classe `Cours` :

- a. Définissez une méthode `prealables` qui reçoit en argument un `predicat` — représenté par un **bloc** — et qui retourne la liste des prélabes du cours qui satisfont ce `predicat`.
- b. Utilisez la méthode `prealables` pour obtenir les prélabes du cours `inf3105` dont le sigle contient la chaîne "INF".

Remarque : Pour ce dernier point, vous devez utiliser une expression de *pattern-matching*. En Ruby, l'expression suivante retourne un résultat *non nil* si `x`, une chaîne, matche le motif `INF` :

```
/INF/ =~ x
```

Plus précisément, l'expression retourne `nil` si le motif n'apparaît pas dans la chaîne, sinon elle retourne la **position** du premier *match*.

Exercice 4.6: Une méthode pour identifier un sous-ensemble de prélabes d'un cours.

Exemple Ruby 4.35 Illustration de la vie et portée des variables.

```
>> # Une definition de methode ne voit pas
    # les variables non-locales.
?> x = 22
=> 22

>> def set_x
    x = 88
end
=> :set_x

>> set_x
=> 88

>> x      # Inchangee !
=> 22
```

```
?> # Un bloc capture les variables non-locales
    # si elles existent.
?> def executer_bloc
    yield
end
=> :executer_bloc

>> x = 44
=> 44

>> executer_bloc { x = 55 }
=> 55

>> x      # Modifiée!
=> 55
```

```
?> # Si la variable n'existe pas deja ,  
    # alors est strictement locale au bloc.  
?  
?> z  
NameError: undefined local variable or method 'z' for main:Object  
[...]  
  
?> executer_bloc { z = 88 }  
=> 88  
  
>> z  
NameError: undefined local variable or method 'z' for main:Object  
[...]
```

```
>> # Une variable globale est accessible partout!  
?> $x_glob = 99  
=> 99  
  
>> def set_x_glob  
      $x_glob = "abc"  
    end  
=> :set_x_glob  
  
>> set_x_glob  
=> "abc"  
  
>> $x_glob  
=> "abc"  
  
>> lambda { $x_glob = [10, 20] }.call  
=> [10, 20]  
  
>> $x_glob  
=> [10, 20]
```

```
>> # Une variable locale est accessible dans l'ensemble
    # de la methode.
?> def foo( x )
      if x <= 0 then a = 1 else b = "BAR" end
      [a, b]
    end
=> :foo

>> foo( 0 )
=> [1, nil]

>> foo( 99 )
=> [nil, "BAR"]
```

```
>> # Mais un bloc definit une nouvelle portee , avec des var  
    # strictement locales!  
?> def bar( *args )  
    args.each do |x|  
      r = 10  
      puts x * r  
    end  
    r  
  end  
=> :bar  
  
>> bar( 10, 20 )  
100  
200  
NameError: undefined local variable or method 'r' for main:Object  
[...]
```

4.16 Modules

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits:

- 1. Modules provide a **namespace** and prevent name clashes.*
- 2. Modules implement the **mixin** facility.*

Source : http://ruby-doc.com/docs/ProgrammingRuby/html/tut_modules.html

Exemple Ruby 4.36 Les modules comme *espaces de noms*.

```
module M1
  C1 = 0
end

module M2
  C1 = 'abc'
end

module M3
  module M4
    C1 = :c1
  end
end

M1::C1 == 0           # => true
M2::C1 == 'abc'       # => true
M3::M4::C1 == :c1     # => true

M1::C1 != M2::C1      # => true
M1::C1 != M3::M4::C1  # => true
...
```

```
module Module1
  def self.zero
    0
  end

  def un
    1
  end

  def val_x
    @x
  end

  def inc_inc( y )
    inc( y )
    inc( y )
  end
end

class C1
  include Module1

  def initialize( x )
    @x = x
  end

  def inc( y )
    @x += y
  end
end

class C2
  include Module1
end
```

Exemple Ruby 4.37 Un module *mixin* Module1 et son utilisation.

```
>> # Appel sur le module de la methode de classe.
```

```
?> Module1.zero
```

```
=> 0
```

```
>> # Appel sur le module de la methode d'instance.
```

```
?> Module1.un
```

```
NoMethodError: undefined method 'un' for Module1:Module
```

```
...
```

```
>> # Appel sur un objet C1 des methodes
```

```
?> # de classe et d'instance du module.
```

```
?> c1 = C1.new( 99 )
```

```
=> #<C1:0x12cf7ab @x=99>
```

```
>> c1.zero
```

```
NoMethodError: undefined method 'zero' for #<C1:0x12cf7ab @x
```

```
...
```

```
>> c1.un
```

```
=> 1
```

```
>> c1.val_x
```

```
=> 99
```

```
>> c1.inc_inc( 100 )
```

```
=> 299
```

```
>> # Appel sur un objet C2 des methodes
```

```
?> # de classe et d'instance du module.
```

```
?> c2 = C2.new
```

```
=> #<C2:0x1a8622>
```

```
>> c2.un
```

```
=> 1
```

```
>> c2.val_x
```

```
=> nil
```

```
>> c2.inc_inc( 100 )
```

```
NoMethodError: undefined method 'inc' for #<C2:0x1a8622>
```

```
...
```

NomDuModule.nom_methode

NomDuModule::nom_methode

```
module Module1
  def self.zero
    0
  end

  def un
    1
  end

  def val_x
    @x
  end

  def inc_inc( y )
    inc( y ); inc( y )
  end
end
```

```
?> Module1.zero
```

```
=> ??
```

```
?> Module1.un
```

```
??
```

```
...
```

```
?> Module1.val_x
```

```
??
```

```
...
```

```

module Module1
  def self.zero
    0
  end

  def un
    1
  end

  def val_x
    @x
  end

  def inc_inc( y )
    inc( y ); inc( y )
  end
end

```

```

class C1
  include Module1

  def initialize( x )
    @x = x
  end

  def inc( y )
    @x += y
  end
end

```

#

```

?> c1 = C1.new( 99 )
=> #<C1:0x12cf7ab @x=99>

```

```

>> c1.zero

```

```

??

```

```

...

```

```

>> c1.un

```

```

=> ??

```

```

>> c1.val_x

```

```

=> ??

```

```

>> c1.inc_inc( 100 )

```

```

=> ??

```

```

module Module1
  def self.zero
    0
  end

  def un
    1
  end

  def val_x
    @x
  end

  def inc_inc( y )
    inc( y ); inc( y )
  end
end

```

```

class C2
  include Module1
end

```

#

```

?> c2 = C2.new
=> #<C2:0x1a8622>

```

```

>> c2.un
=> ??

```

```

>> c2.val_x
=> ??

```

```

>> c2.inc_inc( 100 )
??

```

...

```
module Module1
  def self.zero
    0
  end

  def un
    1
  end

  def val_x
    @x
  end

  def inc_inc( y )
    inc( y ); inc( y )
  end
end
```

```
?> Module1.zero
=> 0
```

```
?> Module1.un
NoMethodError: undefined method 'un' for Module1:Module
...
```

```
?> Module1.val_x
NoMethodError: undefined method 'val_x' for Module1:Module
...
```

```

module Module1
  def self.zero
    0
  end

  def un
    1
  end

  def val_x
    @x
  end

  def inc_inc( y )
    inc( y ); inc( y )
  end
end

```

```

class C1
  include Module1

  def initialize( x )
    @x = x
  end

  def inc( y )
    @x += y
  end
end

```

#

```

?> c1 = C1.new( 99 )
=> #<C1:0x12cf7ab @x=99>

```

```

>> c1.zero
NoMethodError: undefined method 'zero' for #<C1:0x12cf7ab @x=99>
...

```

```

>> c1.un
=> 1

```

```

>> c1.val_x
=> 99

```

```

>> c1.inc_inc( 100 )
=> 299

```

```

module Module1
  def self.zero
    0
  end

  def un
    1
  end

  def val_x
    @x
  end

  def inc_inc( y )
    inc( y ); inc( y )
  end
end

```

```

class C2
  include Module1
end

```

```
#
```

```

?> c2 = C2.new
=> #<C2:0x1a8622>

```

```

>> c2.un
=> 1

```

```

>> c2.val_x
=> nil

```

```

>> c2.inc_inc( 100 )
NoMethodError: undefined method 'inc' for #<C2:0x1a8622>
...

```

4.17 Modules Enumerable et Comparable

4.17.1 Module Enumerable

Learn to use Enumerable. You will not be a rubyist until you do.

«Ruby QuickRef», R. Davis

(<http://www.zenspider.com/Languages/Ruby/QuickRef.html>)

La figure à la page ?? présente la liste des méthodes du module `Enumerable` — donc les diverses méthodes disponibles lorsque la méthode `each` est définie par une classe et que le module `Enumerable` est inclus (avec `include`)!

Methods

```
::[]  
::new  
:try_convert  
#&  
#*  
#+  
#-  
#<<  
#<=>  
#==  
#[]  
#[]=  
#any?  
#assoc  
#at  
#bsearch  
#clear  
#collect  
#collect!  
#combination  
#compact  
#compact!  
#concat  
#count  
#cycle  
#delete  
#delete_at  
#delete_if  
#drop  
#drop_while  
#each  
#each_index  
#empty?  
#eq?
```

```
#fetch  
#fill  
#find_index  
#first  
#flatten  
#flatten!  
#frozen?  
#hash  
#include?  
#index  
#initialize_copy  
#insert  
#inspect  
#join  
#keep_if  
#last  
#length  
#map  
#map!  
#pack  
#permutation  
#pop  
#product  
#push  
#rassoc  
#reject  
#reject!
```

```
#repeated_combination  
#repeated_permutation  
#replace  
#reverse  
#reverse!  
#reverse_each  
#rindex  
#rotate  
#rotate!  
#sample  
#select  
#select!  
#shift  
#shuffle  
#shuffle!  
#size  
#slice  
#slice!  
#sort  
#sort!  
#sort_by!  
#take  
#take_while  
#to_a  
#to_ary  
#to_h  
#to_s  
#transpose  
#uniq  
#uniq!  
#unshift  
#values_at  
#zip  
#|
```

Exemple Ruby 4.38 Exemples d'utilisation du module Enumerable.

```
?>> # La classe Array definit la methode each et  
      # inclut le module Enumerable.
```

```
>> a = [10, 20, 30, 40]  
=> [10, 20, 30, 40]
```

```
?> # Appartenance d'un element.  
?> a.include? 20  
=> true
```

```
>> a.include? 999  
=> false
```

```
>> a
=> [10, 20, 30, 40]
```

```
?> # Application fonctionnelle.
?> a.map { |x| x + 2 }      # Synonyme = collect.
=> [12, 22, 32, 42]
```

```
>> a
=> ??
```

```
?> # Application imperative (mutable)!
>> a.map! { |x| 10 * x }
=> [100, 200, 300, 400]
```

```
>> a
=> ??
```

```
>> a
=> [10, 20, 30, 40]

?> # Application fonctionnelle.
?> a.map { |x| x + 2 }    # Synonyme = collect.
=> [12, 22, 32, 42]

>> a                      # a n'est pas modifie.
=> [10, 20, 30, 40]

?> # Application imperative (mutable)!
>> a.map! { |x| 10 * x }
=> [100, 200, 300, 400]

>> a                      # a est modifie!
=> [100, 200, 300, 400]
```

```
?> # Selection/rejet d'elements selon un critere.  
>> a.select { |x| x >= 300 }  
=> [300, 400]
```

```
>> a.reject { |x| x >= 300 }  
=> [100, 200]
```

```
>> a  
=> [100, 200, 300, 400]
```

```
# Il existe aussi des variantes imperatives/mutables:  
#   select!  
#   reject!
```

```
?> # Obtention du premier element qui satisfait un critere.  
>> a  
=> [100, 200, 300, 400]  
  
>> a.find { |x| x > 200 } # Synonyme = detect.  
=> 300  
  
>> a.find { |x| x < 0 }  
=> nil
```

```
?> # Quantificateurs .
?> a.all? { |x| x > 0 }
=> true

>> a.any? { |x| x > 500 }
=> false
```

```
?> # Reduction avec un operateur binaire.  
>> a  
=> [100, 200, 300, 400]  
  
?> a.reduce { |x, y| x + y } # Synonyme = inject.  
=> 1000  
  
>> a.reduce( :+ )  
=> 1000  
  
>> a.reduce( &:+ )  
=> 1000  
  
>> a.reduce( :* )  
=> 24000000000  
  
>> a.reduce( 999, :+ )  
=> 1999
```

```
?> # Autres exemples de reduction, avec operateurs divers.
>> a.reduce(0) { |max, x| x > max ? x : max }
=> ??

>> a.map { |x| x / 10 }
=> ??

>> a.reduce([]) { |a, x| a << x / 10 }
=> ??

>> a.reduce([]) { |ar, x| [x] + ar + [x] }
=> ??
```

```
?> # Autres exemples de reduction, avec operateurs divers.
>> a.reduce(0) { |max, x| x > max ? x : max }
=> 400

>> a.map { |x| x / 10 }
=> [10, 20, 30, 40]

>> a.reduce([]) { |a, x| a << x / 10 }
=> [10, 20, 30, 40]

>> a.reduce([]) { |ar, x| [x] + ar + [x] }
=> [400, 300, 200, 100, 100, 200, 300, 400]
```

Note : Le `a` de `a.reduce` (défini au niveau global) est [distinct](#) du `a` dans `a << x` (paramètre, donc identificateur strictement local au bloc).

```
?> # Regroupement , dans un Hash , des elements
    # avec une meme valeur specifiee par le bloc .
>> a.group_by { |x| x }
=> {100=>[100] , 200=>[200] , 300=>[300] , 400=>[400]}
```

```
>> a.group_by { |x| x >= 222 }
=> {false=>[100, 200] , true=>[300, 400]}
```

```
>> a.group_by { |x| x / 100 }
=> {1=>[100] , 2=>[200] , 3=>[300] , 4=>[400]}
```

```
>> a.group_by { |x| x % 2 }
=> {0=>[100, 200, 300, 400]}
```

```
>> a.group_by { |x| (x / 100) % 2 }
=> {1=>[100, 300] , 0=>[200, 400]}
```

```
?> # <<Aplatissement>> des elements d'un tableau.  
>> [10, 20, 30].flatten  
=> [10, 20, 30]
```

```
>> [10, [20, 30], [40], [], [50], 60].flatten  
=> [10, 20, 30, 40, 50, 60]
```

```
>> [10, [20, 30], [[40], []], [[[50]]], 60].flatten  
=> [10, 20, 30, 40, 50, 60]
```

```
# f.flat_map { |x| ... } = f.map { |x| ... }.flatten  
>> [1, 2, 3].map { |n| [*1..n] }  
=> [[1], [1, 2], [1, 2, 3]]
```

```
>> [1, 2, 3].map { |n| [*1..n] }.flatten  
=> [1, 1, 2, 1, 2, 3]
```

```
>> [1, 2, 3].flat_map { |n| [*1..n] }  
=> [1, 1, 2, 1, 2, 3]
```

Exemple Ruby 4.39 Une mise en oeuvre, en Ruby, de quelques méthodes du module `Enumerable`, méthodes qui utilisent la méthode `each` de la classe ayant exécuté l'appel «include Enumerable».

```
# Mise en oeuvre possible , en Ruby , de quelques methodes
# du module Enumerable: on utilise *uniquement* each!

module Enumerable
  def include?( elem )
    each do |x|
      return true if x == elem
    end

    false
  end

  def find
    each do |x|
      return x if yield(x)
    end

    nil
  end

  def reduce( val_initiale )
    # Autre argument implicite = bloc recevant deux arguments
    accum = val_initiale
    each do |x|
      accum = yield( accum, x )
    end

    accum
  end
end
```

Donnez une mise en oeuvre, dans un style fonctionnel, de la méthode `to_s` de la classe `Cours` vue précédemment.

Exercice 4.7: Mise en oeuvre fonctionnelle de `Cours#to_s`.

4.17.2 Module Comparable

La figure ci-bas présente la liste des méthodes du module `Comparable`, c'est-à-dire, les diverses méthodes disponibles lorsque la méthode `<=>` est définie par une classe et que le module `Comparable` est inclus (avec `include`)!

Methods
<code>#<</code>
<code>#<=</code>
<code>#==</code>
<code>#></code>
<code>#>=</code>
<code>#between?</code>

Exemple Ruby 4.40 Tris avec Enumerable et <=>.

```
>> # Comparaison avec l'opérateur 'spaceship'.  
?> 29 <=> 33  
=> -1  
>> 29 <=> 29  
=> 0  
>> 29 <=> 10  
=> 1
```

```
>> # Tris.
>> a = [29, 10, 44, 33]
=> [29, 10, 44, 33]

>> a.sort
=> [10, 29, 33, 44]

>> a.sort { |x, y| x <=> y }
=> [10, 29, 33, 44]

>> a.sort { |x, y| -1 * (x <=> y) }
=> [44, 33, 29, 10]

>> a.sort { |x, y| (x % 10) <=> (y % 10) }
=> [10, 33, 44, 29]
```

Exemple Ruby 4.41 Comparaison et tri de Cours via les sigles.

```
$ cat cours-bis.rb
require_relative 'cours'

class Cours
  include Comparable

  def <=>( autre )
    sigle <=> autre.sigle
  end
end

if $0 == __FILE__
  # Definition de quelques cours.
  inf1120 = Cours.new( :INF1120, 'Programmation I' )
  inf1130 = Cours.new( :INF1130, 'Maths pour informaticien' )
  inf2120 = Cours.new( :INF2120, 'Programmation II', inf1120 )
  inf3105 = Cours.new( :INF3105, 'Str. de don.', inf1130, inf2120 )

  cours = [ inf3105, inf1120, inf2120, inf1130 ]

  # Quelques expressions
  puts inf3105 < inf1120
  puts inf2120 >= inf1130
  cours.sort.each { |c| puts c }
end

-----

$ ruby cours-bis.rb
false
true
< INF1120 'Programmation I' ( ) >
< INF1130 'Maths pour informaticien' ( ) >
< INF2120 'Programmation II' ( INF1120 ) >
< INF3105 'Str. de don.' ( INF1130 INF2120 ) >
```

Que fait la méthode suivante? Quel nom plus significatif pourrait-on lui donner?

```
class Array
  def mystere( p )
    reduce( [], [], [] ) do |res, x|
      res[1 + (x <=> p)] << x

      res
    end
  end
end
```

Exercice 4.8: Méthode `mystere` sur un `Array`.

4.18 Itérateurs définis par le programmeur

Exemple Ruby 4.42 Une classe (simplifiée) pour des Ensembles.

```
class Ensemble
  include Enumerable

  # Ensemble initialement vide (sans element).
  def initialize
    @elements = []
  end

  # Ajout d'un element, sauf si deja present!
  def <<( x )
    @elements << x unless contient? x

    self
  end

  def each
    @elements.each do |x|
      yield( x )
    end
  end
end
```

```
def cardinalite
  count
end

def contient?( x )
  include? x
end

def somme( val_initiale = 0 )
  reduce(val_initiale) { |s, x| s + x }
end

def produit( val_initiale = 1 )
  reduce(val_initiale) { |s, x| s * x }
end

def to_s
  "{ " << map { |x| x.to_s }.join(", ") << " }"
end
end
```

Pourquoi la méthode << retourne-t-elle `self`?

Que se passe-t-il si on omet `self`?

Exercice 4.9: Pourquoi la méthode << retourne-t-elle `self`?

alias :cardinalite :count

alias :contient? :include?

Exemple Ruby 4.43 Quelques expressions utilisant un objet `Ensemble`.

```
?> # Cree un ensemble avec divers elements.
```

```
?> ens = Ensemble.new << 1 << 5 << 3
```

```
=> #<Ensemble:0x000000023c9298 @elements=[1, 5, 3]>
```

```
>> ens.to_s
```

```
=> "{ 1, 5, 3 }"
```

```
?> # L'operation << modifie l'objet.
```

```
?> ens << 2
```

```
=> #<Ensemble:0x000000023c9298 @elements=[1, 5, 3, 2]>
```

```
>> ens.to_s
```

```
=> "{ 1, 5, 3, 2 }"
```

```
?> # Appels a diverses methodes directement definies par Ensemble
```

```
?> ens.contient? 10
```

```
=> false
```

```
>> ens.contient? 2
```

```
=> true
```

```
>> ens.somme
```

```
=> 11
```

```
>> ens.somme(33)
```

```
=> 44
```

```
>> ens.produit
```

```
=> 30
```

```
>?> # Appels a des methodes definies par Enumerable.  
>> ens.to_s  
=> "{ 1, 5, 3, 2 }"  
  
?> ens.map { |x| x * 10 }  
=> [10, 50, 30, 20]  
  
>> ens.reject { |x| x.even? }  
=> [1, 5, 3]  
  
>> ens.find { |x| x >= 2 }  
=> 5
```

Remarque : Suppose que la méthode suivante est définie dans **Ensemble** :

Supposons que dans la classe `Array`, on veuille définir les méthodes `map` et `select`, et ce utilisant `each` ou `each_index`. Quel code faudrait-il écrire?

```
class Array
  def map
    ...
  end

  def select
    ...
  end
end
```

Remarque : *Conceptuellement*, dans la vraie classe `Array`, ces méthodes sont disponibles simplement parce que la classe `Array` **inclut** le module `Enumerable`. **En pratique**, la mise en oeuvre de ces méthodes pour la classe `Array` est faite de façon spécifique à cette classe, pour des raisons d'efficacité — notamment, méthodes écrites en C dans Ruby/MRI.

Exercice 4.10: Mises en oeuvre de `map` et `select`.

4.19 Expressions régulières et *pattern-matching*

4.19.1 Les caractères spéciaux

\	Supprime la signification spéciale du caractère qui suit
.	Un caractère arbitraire
Répétitions	
*	0, 1 ou plusieurs occurrences du motif qui précède
?	0 ou 1 occurrence du motif qui précède
+	1 ou plusieurs occurrences du motif qui précède
{n}	Exactement n occurrences du motif qui précède
{n,}	Au moins n occurrences du motif qui précède
{,n}	Au plus n occurrences du motif qui précède
{n,m}	De n à m occurrences du motif qui précède
Ancrages	
^	Début de la ligne
\$	Fin de la ligne
Classes de caractères	
[...]	Un caractère qui fait partie de la classe
[^...]	Un caractère qui ne fait pas partie de la classe
\d	Un nombre décimal
\D	Tout sauf un nombre décimal
\s	Un espace blanc (espace, tabulation, saut de ligne, etc.)
\S	Tout sauf un espace blanc
\w	Un caractère alphanumérique = a-zA-Z0-9_
\W	Tout sauf un caractère alphanumérique
Autres caractères spéciaux	
$m_1 m_2$	Choix entre motif m_1 ou motif m_2
(...)	Création d'un groupe et d'une référence au groupe matché
\b	Une <i>frontière</i> de mot
\A	Le début de la chaîne
\Z	La toute fin de la chaîne
\Z	La fin de la chaîne (ignore le saut de ligne qui suit)

Tableau 4.2: Les principaux caractères spéciaux utilisés dans les expressions régulières.

4.19.2 Les expressions régulières et la méthode «=~»

Exemple Ruby 4.44 Une expression régulière est un objet de classe Regexp.

```
>> # Exemples de base .

>> /ab.*zz$/ .class
=> Regexp

>> re = /ab.*zz$/
=> /ab.*zz$/
>> re.class
=> Regexp

>> re = Regexp.new( "ab.*zz$" )
=> /ab.*zz$/
>> re.class
=> Regexp

# Autre façon .
>> re = %r{ab.*zz$}
=> /ab.*zz$/
```

Exemple Ruby 4.45 Une expression régulière peut être utilisée dans une opération de *pattern-matching* avec «`=~`».

```
>> # Exemples de base (suite).
```

```
>> re = Regexp.new( "ab.*zz$" )  
=> /ab.*zz$/
```

```
>> re =~ "abcdzz00"  
=> nil
```

```
>> re =~ "abcdzz"  
=> 0
```

```
>> re =~ ("abcdzz")  
=> 0
```

```
>> re =~ "....abcdzz"  
=> 4
```

```
>> "....abcdzz" =~ re  
=> 4
```

```
>> puts "Ca matche" if re =~ "....abcdzz"
Ca matche
=> nil
```

```
>> re !~ "abcdzz00"
=> true
```

```
>> re !~ "abcdzz"
=> false
```

L'opérateur `=~` est une méthode de la classe `Regexp`, mais les appels suivants sont équivalents :

- `re =~ ch`
- `re =~( ch )`
- `re =~ ch`
- `ch =~ re`

4.19.3 Quelques caractères spéciaux additionnels et quelques options

Exemple Ruby 4.46 Autres caractères spéciaux des motifs et options.

```
>> # L'option i permet d'ignorer la casse.
```

```
>> /bc/ =~ "ABCD"
```

```
=> nil
```

```
>> /bc/i =~ "ABCD"
```

```
=> 1
```

```
>> # Un "." *ne* matche pas* un saut de ligne...
    #      sauf avec l'option m.
    # Un \s matche un saut de ligne.
```

```
>> /z.abc/ =~ "xyz\nabc"
=> nil
```

```
>> /z.abc/m =~ "xyz\nabc"
=> 2
```

```
>> /z\sabc/ =~ "xyz\nabc"
=> 2
```

Exemple Ruby 4.47 L'option «x» permet de mieux formater des expressions régulières complexes.

```
>> motif = /({CODE_REG}) # Le code regional
              -           # Un tiret
              ({TEL})     # Le numero de tel.
            /x
=> /((?-mix:\d{3})) # Le code regional
      -           # Un tiret
      ((?-mix:\d{3}-\d{4})) # Le numero de tel.
    /x

>> motif.match "Tel.: 514-987-3000 ext. 8213"
=> #<MatchData "514-987-3000" 1:"514" 2:"987-3000">
```

Exemple Ruby 4.48 Début/fin de chaine vs. début/fin de ligne.

```
>> # Debut de ligne vs. debut de chaine.
```

```
>> /^abc/ =~ "xxx\nabc\n"  
=> 4
```

```
>> /\Aabc/ =~ "xxx\nabc\n"  
=> nil
```

```
>> # Fin de ligne vs. fin de chaine.
```

```
>> /abc$/ =~ "xxx\nabc\n"  
=> 4
```

```
>> /abc\z/ =~ "xxx\nabc\n"  
=> nil
```

```
>> /abc\n\z/ =~ "xxx\nabc\n"  
=> 4
```

```
>> /abc\Z/ =~ "xxx\nabc\n"  
=> 4
```

Exemple Ruby 4.49 Autres exemples de groupes : avec vs. sans capture.

```
>> /(ab)(cd)(ef)/ =~ "abcdef"
=> 0
```

```
>> puts $1, $2, $3
ab
cd
ef
=> nil
```

```
>> /(ab)(?:cd)(ef)/ =~ "abcdef"
=> 0
```

```
>> puts $1, $2, $3
ab
ef

=> nil
```

```
>> /(ab)(?#cd)(ef)/ =~ "abcdef"
=> nil
```

```
>> /(ab)(?#cd)(ef)/ =~ "abef"
=> 0
```

Exemple Ruby 4.50 Autres exemples de *pattern-matching* : matche vorace vs. paresseux.

```
>> /ab(c.*)d/ =~ "abccddccddccd"
=> 0
```

```
>> $1
=> "ccddccddcc"
```

```
>> /ab(c.*?)d/ =~ "abccddccddccd"
=> 0
```

```
>> $1
=> "ccc"
```

Exemple Ruby 4.51 Autre caractère spécial : frontière de mot.

```
>> /abc/ =~ "xabc"  
=> 1
```

```
>> /\babc/ =~ "xabc"  
=> nil
```

```
>> /\babc/ =~ "x abc"  
=> 2
```

4.19.4 La classe MatchData

Exemple Ruby 4.52 Les méthodes d'un objet MatchData, objet retourné par l'opération Regexp#match.

```
>> # Les objets MatchData.

>> CODE_REG = /\d{3}/
=> /\d{3}/
>> TEL = /\d{3}-\d{4}/
=> /\d{3}-\d{4}/

>> m = /(#{CODE_REG})-(#{TEL})/
      .match "F00"
=> nil

>> m = /(#{CODE_REG})-(#{TEL})/
      .match "Tel.: 514-987-3000 ext. 8213"
=> #<MatchData "514-987-3000" 1:"514" 2:"987-3000">

>> m[0..-1]
=> ["514-987-3000", "514", "987-3000"]

>> m.begin(0)..m.end(0)
=> 6..18
>> m.begin(1)..m.end(1)
=> 6..9
>> m.begin(2)..m.end(2)
=> 10..18

>> m.pre_match
=> "Tel.: "

>> m.post_match
=> " ext. 8213"
```

Exemple Ruby 4.53 Les groupes **avec noms** et les variables spéciales «**\$i**» définies par la méthode «**=~**».

```
# Des groupes avec noms explicites .
```

```
>> m = /( ?<code_reg>#{CODE_REG}) - ( ?<tel>#{TEL}) / .  
      match "Tel.: 514-987-3000 ext. 8213"  
=> #<MatchData "514-987-3000" code_reg:"514"  
    tel:"987-3000">
```

```
>> m[:code_reg]  
=> "514"
```

```
>> m.begin(:code_reg)  
=> 6
```

```
>> m[:tel]  
=> "987-3000"
```

```
>> m.end(:tel)  
=> 18
```

```
# Les variables speciales $1 , $2 , etc .
# indiquent les groupes captures .

>> if /({CODE_REG})-({TEL})/ =~
      "Tel.: 514-987-3000 ext. 8213"
    puts "code reg. = #{ $1 }; tel. = #{ $2 }"
  end
code reg. = 514; tel. = 987-3000
=> nil
```

Qu'est-ce qui sera imprimé par les instructions p suivantes :

```
code_permanent = /(\w{4})      # N O M P
                  (\d{2})      # A n n e e
                  (\d{2})      # M o i s
                  (\d{2})      # J o u r
                  ([^\D]{2})
                  /x

m = code_permanent
    .match "CP: DEFG11229988."

p m[1]
p m[5]
p m.pre_match
p m.post_match
```

Exercice 4.11: Objet MatchData.

4.20 Interactions avec l'environnement

4.20.1 Arguments du programme

Exemple Ruby 4.54 Les arguments d'un programme Ruby et les variables d'environnement.

```
$ cat argv.rb
#!/usr/bin/env ruby

i = 0
while arg = ARGV.shift do
  puts "ARGV[#{i}] = '#{arg}' (#{arg.class})"
  i += 1
end

puts "ENV['FOO'] = '#{ENV['FOO']}'"
ENV['FOO'] = 'FOO argv.rb'
puts "-----"
```

```
$ echo $FOO
```

```
$ ./argv.rb  
ENV['FOO'] = ''  
-----
```

```
$ ./argv.rb 1234 'abc " def' abc def "'"  
ARGV[0] = '1234' (String)  
ARGV[1] = 'abc " def' (String)  
ARGV[2] = 'abc' (String)  
ARGV[3] = 'def' (String)  
ARGV[4] = '' (String)  
ENV['FOO'] = ''  
-----
```

```
$ export FOO=xyz; ./argv.rb def; echo $FOO  
ARGV[0] = 'def' (String)  
ENV['FOO'] = 'xyz'  
-----  
xyz
```

```
$ FOO=123 ./argv.rb def; echo $FOO  
ARGV[0] = 'def' (String)  
ENV['FOO'] = '123'  
-----  
xyz
```

Soit le script suivant :

```
$ cat argv2.rb
#!/usr/bin/env ruby

ENV['NB'].to_i.times do
  puts ARGV[0] + ARGV[1]
end
```

Qu'est-ce qui sera imprimé par les appels suivants :

```
# a .
NB=3 ./argv2.rb 3 8

# b .
NB=2 ./argv2.rb [1, 2] [3]

# c .
unset NB; ./argv2.rb [1009, 229342] [334]
```

Exercice 4.12: Utilisation de ARGV et ENV.

4.20.2 Écriture sur le flux de sortie standard : printf, puts, print et p

Exemple Ruby 4.55 Exemples d'utilisation de printf, sprintf et print.

```
>> printf "%d\n", "123"  
123  
=> nil
```

```
>> STDOUT.printf "%s\n", "123"  
123  
=> nil
```

```
>> printf "%d\n", "abc"  
ArgumentError: invalid value for Integer(): "abc"  
[...]
```

```
>> printf "%s\n", "abc"  
abc  
=> nil
```

```
>> printf "%d\n", [10, 20]  
TypeError: can't convert Array into Integer  
[...]
```

```
>> printf "%s\n", [10, 20]  
[10, 20]  
=> nil
```

```
# On peut aussi utiliser un format pour  
# generer une chaine , sans effet  
# sur le flux de sortie .
```

```
>> res = sprintf "%d\n", 123  
=> "123\n"
```

```
>> res  
=> "123\n"
```

```
>> print 123  
123=> nil  
>> print "123"  
123=> nil  
>> print "123\n"  
123  
=> nil
```

Exemple Ruby 4.56 Écriture d'un entier ou d'une chaîne simple.

```
$ cat print-et-al.rb
#!/usr/bin/env ruby

def imprimer( methode, *valeurs )
  puts "*** Avec #{methode}:"
  valeurs.each do |x|
    send methode, x
    puts "..."
  end
end

imprimer( :puts, 123, "123" )
puts
imprimer( :p, 123, "123" )
```

```
$ ./print-et-al.rb
*** Avec puts:
123
...
123
...

*** Avec p:
123
...
"123"
...
```

Exemple Ruby 4.57 Écriture d'un tableau d'entiers ou un tableau de chaines.

```
$ cat print-et-al.rb
```

```
#!/usr/bin/env ruby
```

```
def imprimer( methode, *valeurs )
  puts "*** Avec #{methode}:"
  valeurs.each do |x|
    send methode, x
    puts "..."
  end
end
```

```
imprimer( :puts, [123, 456], ["123", "456"] )
puts
imprimer( :p, [123, 456], ["123", "456"] )
```

```
$ ./print-et-al.rb
```

```
*** Avec puts:
```

```
123
```

```
456
```

```
...
```

```
123
```

```
456
```

```
...
```

```
*** Avec p:
```

```
[123, 456]
```

```
...
```

```
["123", "456"]
```

```
...
```

Exemple Ruby 4.58 Écriture d'un objet qui n'a pas de méthodes `to_s` et `inspect`.

```
$ cat print-et-al.rb
#!/usr/bin/env ruby

def imprimer( methode, *valeurs )
  puts "*** Avec #{methode}:"
  valeurs.each do |x|
    send methode, x
    puts "..."
  end
end

class Bar
  def initialize( val ); @val = val; end
end
```

```
imprimer( :puts, Bar.new(10) )
puts
imprimer( :p, Bar.new(10) )
```

```
$ ./print-et-al.rb
*** Avec puts:
#<Bar:0x000000015022a0>
...

*** Avec p:
#<Bar:0x00000001501f80 @val=10>
...
```

Exemple Ruby 4.59 Écriture d'un objet **qui a** des méthodes `to_s` et `inspect`.

```
$ cat print-et-al.rb
#!/usr/bin/env ruby

def imprimer( methode, *valeurs )
  puts "*** Avec #{methode}:"
  valeurs.each do |x|
    send methode, x
    puts "..."
  end
end

class Foo
  def initialize( val ); @val = val; end

  def to_s; "#{@val}"; end

  def inspect; "#<Foo: val=#{@val}>"; end
end

imprimer( :puts, Foo.new(10) )
puts
imprimer( :p, Foo.new(10) )

$ ./print-et-al.rb
*** Avec puts:
10
...

*** Avec p:
#<Foo: val=10>
...
```

4.20.3 Manipulation de fichiers

Exemple Ruby 4.60 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

File.open( nom_fichier, "r" ) do |fich|
  fich.each_line do |ligne|
    puts ligne
  end
end
```

```
$ cat foo.txt
abc def
123 456

xxx
...
```

```
$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

Exemple Ruby 4.60 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

fich = File.open( nom_fichier, "r" )

fich.each_line do |ligne|
  puts ligne
end

fich.close
```

```
$ cat foo.txt
abc def
123 456

xxx
...
```

```
$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

Exemple Ruby 4.60 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

IO.readlines( nom_fichier ).each do |ligne|
  puts ligne
end
```

```
$ cat foo.txt
abc def
123 456

xxx
...
```

```
$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

Exemple Ruby 4.60 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

puts IO.readlines( nom_fichier )
```

```
$ cat foo.txt
abc def
123 456

xxx
...

$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

Mode	Meaning
"r"	Read-only, starts at beginning of file (default mode).
"r+"	Read-write, starts at beginning of file.
"w"	Write-only, truncates existing file to zero length or creates a new file for writing.
"w+"	Read-write, truncates existing file to zero length or creates a new file for reading and writing.
"a"	Write-only, starts at end of file if file exists, otherwise creates a new file for writing.
"a+"	Read-write, starts at end of file if file exists, otherwise creates a new file for reading and writing.
"b"	Binary file mode (may appear with any of the key letters listed above). Suppresses EOL <-> CRLF conversion on Windows. And sets external encoding to ASCII-8BIT unless explicitly specified.
"t"	Text file mode (may appear with any of the key letters listed above except "b").

Figure 4.5: Modes d'ouverture des fichiers (source : <http://ruby-doc.org/core-2.0.0/IO.html>).

Exemple Ruby 4.61 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte, dont une façon qui permet de recevoir les données **par l'intermédiaire du flux standard d'entrée**.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

puts (nom_fichier ? IO : STDIN).readlines nom_fichier
```

```
$ cat foo.txt
abc def
123 456

xxx
...
```

```
$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

```
$ cat foo.txt | ./cat.rb
abc def
123 456

xxx
...
```

4.20.4 Exécution de commandes

Exemple Ruby 4.62 Exécution de commandes externes avec *backticks* ou `%x{...}`

```
>> # Execution avec backticks.  
>> ext = 'rb'  
=> "rb"
```

```
>> puts `ls [e]*.#{ext}`  
ensemble.rb  
ensemble_spec.rb  
entrelacement.rb  
=> nil
```

```
>> "#{$?}"  
=> "pid 29829 exit 0"
```

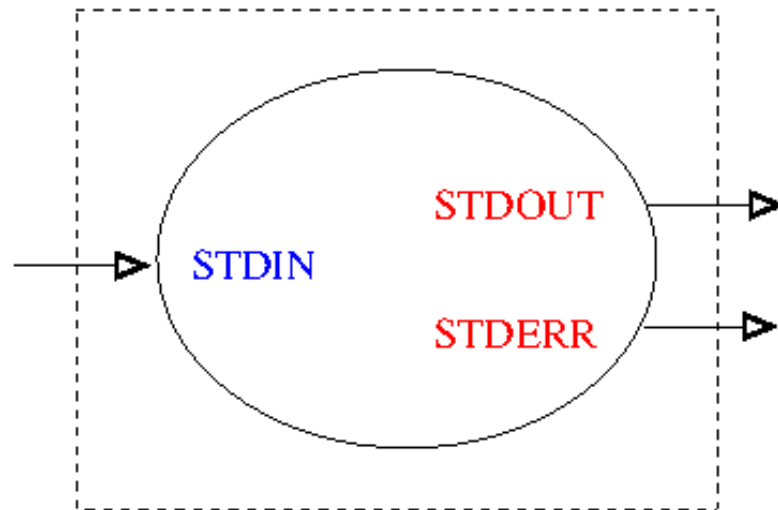
```
>> # Execution avec %x{...}.
>> puts %x{ ls [e]*.#{ext} }
ensemble.rb
ensemble_spec.rb
entrelacement.rb
=> nil

>> $?
=> #<Process::Status: pid 30019 exit 0>
```

```
>> # Emission sur stderr vs. stdout
>> %x{ ls www_xx_z }
ls: impossible d'accéder à www_xx_z:
      Aucun fichier ou dossier de ce type
=> ""

>> "$$?"
=> "pid 29831 exit 2"
```

Vue de l'intérieur



Vue de l'extérieur

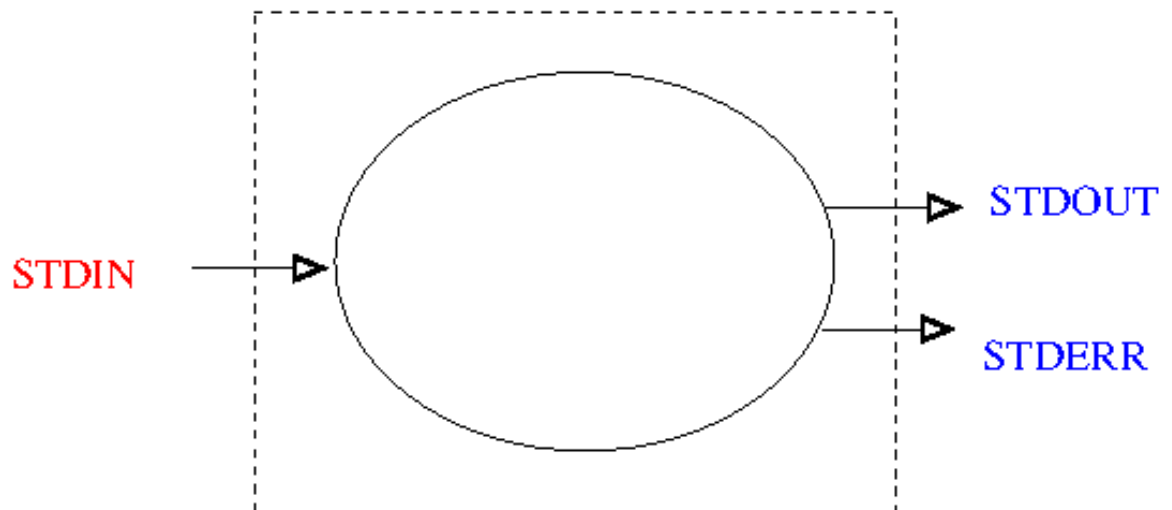


Figure 4.6: Deux points de vue sur les flux associés à un processus.

Exemple Ruby 4.63 Exécution de commandes externes avec `Open3.popen3`.

```
$ cat commandes2.rb
require 'open3'
Open3.popen3( "wc -lw" ) do |stdin, stdout, stderr|
  stdin.puts ["abc def", "", "1 2 3"]
  stdin.close

  puts "--stdout--"
  puts stdout.readlines
  puts "--stderr--"
  puts stderr.readlines
  puts
end

$ ./commandes2.rb
--stdout--
      3      5
--stderr--
```

Exemple Ruby 4.64 Exécution de commandes externes avec `Open3.popen3`.

```
$ cat commandes3.rb
require 'open3'
Open3.popen3( "wc -lw xsfdf.txt" ) do |_, out, err|
  puts "--out--"
  puts out.readlines
  puts "--err--"
  puts err.readlines
  puts
end
```

```
$ ./commandes3.rb
--out--
--err--
wc: xsfdf.txt: Aucun fichier ou dossier de ce type
```

4.21 Traitement des exceptions

4.21.1 Classe Exception et sous-classes standards

```
NoMemoryError
ScriptError
  LoadError
  NotImplementedError
  SyntaxError
SignalException
  Interrupt
StandardError -- default for rescue
  ArgumentError
  IndexError
    StopIteration
  IOError
    EOFError
  LocalJumpError
  NameError
    NoMethodError
  RangeError
    FloatDomainError
  RegexpError
  RuntimeError -- default for raise
  SecurityError
  SystemCallError
    Errno::*
  SystemStackError
  ThreadError
  TypeError
  ZeroDivisionError
SystemExit
fatal -- impossible to rescue
```

Figure 4.7: Hiérarchie des classes/sous-classes standards pour les exceptions (source : <http://ruby-doc.org/core-2.1.1/Exception.html>).

4.21.2 Attraper et traiter une exception

Exemple Ruby 4.65 Une méthode `div` qui attrape et traite diverses exceptions.

```
>> def div( x, y )
      begin
        z = x / y
      rescue ZeroDivisionError => e
        puts "*** Division par 0 ({e})"
        p e.backtrace
        nil
      rescue Exception => e
        puts "*** Erreur = '#{e.inspect}'"
      end
    end
  => :div

>> div 3, 0
*** Division par 0 (divided by 0)
["(irb):4:in '/'",
 "(irb):4:in 'div'", "(irb):14:in 'irb_binding'",
 "/home/tremblay/.rvm/rubies/ruby-2.1.4/lib/ruby/2.1.0/irb/workspace.rb",
 ...,
 "/home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:11:in '<main>'" ]
=> nil

>> div 3, nil
*** Erreur = '#<TypeError: nil can't be coerced into Fixnum>'
=> nil

>> div nil, 3
*** Erreur = '#<NoMethodError: undefined method '/' for nil:NilClass>'
=> nil
```

Exemple Ruby 4.66 Une méthode `traiter_fichier` qui attrape et traite des exceptions et qui s'assure de restaurer le système dans un bon état, qu'une exception soit signalée ou non — dans ce cas-ci, en s'assurant de fermer le descripteur du fichier ayant été ouvert.

```
>> def traiter_fichier( fich )
      f = File.open( fich )
      begin
        traiter_contenu_fichier( f.readlines )
        puts "+++ Traitement termine"
      rescue Exception => e
        puts "*** Erreur = '#{e.inspect}'"
      ensure
        f.close
      end

      f.inspect # Pour voir l'état final de f.
    end
=> :traiter_fichier

>> traiter_fichier( "foo.txt" )
+++ Traitement termine
=> "#<File:foo.txt (closed)>"

>> traiter_fichier( "bar.txt" )
*** Erreur = '#<RuntimeError: Erreur dans traiter_contenu_fi
=> "#<File:bar.txt (closed)>"
```

Exemple Ruby 4.67 La méthode `File.open`, lorsqu'appelée avec un bloc, assure que le fichier sera fermé, qu'une exception survienne ou pas.

```
>> def traiter_fichier( fich )
    le_f = nil
    File.open( fich ) do |f|
        le_f = f
        begin
            traiter_contenu_fichier( f.readlines )
            puts "+++ Traitement termine"
        rescue Exception => e
            puts "*** Erreur = '#{e.inspect}'"
        end
    end
    le_f.inspect
end
=> :traiter_fichier

>> traiter_fichier( "bar.txt" )
*** Erreur = '#<RuntimeError: Erreur dans traiter_contenu_fi
=> "#<File:bar.txt (closed)>"
```

4.21.3 Signaler une exception

Exemple Ruby 4.68 Exemples illustrant l'instruction `fail`, appelée avec 0, 1 ou 2 arguments.

```
>> class MonException < RuntimeError
      def initialize( msg = nil )
        super
      end
    end
=> :initialize

>> def executer
      begin
        yield
      rescue Exception => e
        "classe = #{e.class}; message = '#{e.message}'"
      end
    end
=> :executer

>> executer { fail }
=> "classe = RuntimeError; message = ''"

>> executer { fail "Une erreur!" }
=> "classe = RuntimeError; message = 'Une erreur!'"

>> executer { fail MonException }
=> "classe = MonException; message = 'MonException'"

>> executer { fail MonException, "Probleme!!" }
=> "classe = MonException; message = 'Probleme!!'"
```

Exemple Ruby 4.69 Exemples illustrant l'instruction `raise` utilisée pour **resigner** une exception.

```
>> def executer
      begin
        yield
      rescue Exception => e
        puts "classe = #{e.class}; message = '#{e.message}'"
        raise
      end
    end
  => :executer

>> executer { fail MonException, "Probleme!!" }
classe = MonException; message = 'Probleme!!'
MonException: Probleme!!
      from (irb):16:in 'block in irb_binding'
      from (irb):9:in 'executer'
      from (irb):16
      from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin
```

4.22 Autres éléments de Ruby

4.22.1 L'opérateur préfixe «*»

Exemple Ruby 4.70 Utilisation de l'opérateur «*» (*splat*) **devant** un objet — Range, scalaire ou Range — dans une expression.

```
>> # L'opérateur "splat" (*) devant un tableau "enleve" un r
    # tableau, i.e., integre directement les elements du tabl
    # que le tableau lui-meme.
```

```
>> a = [98, 99]
=> [98, 99]
```

```
>> [1, [10, 20], a, 1000]
=> [1, [10, 20], [98, 99], 1000]
```

```
>> [1, *[10, 20], *a, 1000]
=> [1, 10, 20, 98, 99, 1000]
```

```
>> # L'opérateur splat (*) devant un scalaire ou un Range géométrique
    # tableau avec l'élément ou les éléments indiqués... mais
    # n'importe où.
```

```
>> a = *10
=> [10]
```

```
>> a = *(1..10)
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>> (1..10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
# Mais...
```

```
>> *(1..10)
SyntaxError: (irb):41: syntax error, unexpected '\n',
              expecting :: or '[' or '.'
...

```

```
>> *10
SyntaxError: (irb):33: syntax error, unexpected '\n',
              expecting:: or '[' or '.'
...

```

Exemple Ruby 4.71 Utilisation de l'opérateur «*» du côté gauche d'une affectation parallèle (multiple).

```
>> # Dans la partie gauche d'une affectation parallele , un >
    # <<deconstruire>> un tableau. Dans ce cas , la variable
    # prefixee avec * doit etre unique et va denoter un sous-
    # d'elements .
```

```
>> a, b, c = [10, 20, 30, 40]
=> [10, 20, 30, 40]
```

```
>> puts "a = #{a}; b = #{b}; c = #{c}"
a = 10; b = 20; c = 30
=> nil
```

```
>> a, *b, c = [10, 20, 30, 40]
=> [10, 20, 30, 40]
```

```
>> puts "a = #{a}; b = #{b}; c = #{c}"
a = 10; b = [20, 30]; c = 40
=> nil
```

```
>> premier, *derniers = [10, 20, 30]
=> [10, 20, 30]
```

```
>> puts "premier = #{premier}; derniers = #{derniers}"
premier = 10; derniers = [20, 30]
=> nil
```

```
>> *premiers, dernier = [10, 20, 30]
=> [10, 20, 30]
```

```
>> puts "premiers = #{premiers}; dernier = #{dernier}"
premiers = [10, 20]; dernier = 30
=> nil
```

Exemple Ruby 4.72 Utilisation de «*» dans la spécification de paramètres de méthodes : l'effet est semblable à des affectations parallèles.

```
>> # L'utilisation de * s'applique aussi aux parametres
    # formels d'une methode, ainsi qu'aux arguments effectifs
    # (expressions passees en argument).
>> def foo( x, *args )
    puts "x = #{x}"
    args.each_index { |k| puts "args[#{k}] = #{args[k]}" }
end
=> :foo

>> foo( 10 )
x = 10
=> []

>> foo( 10, 20 )
x = 10
args[0] = 20
=> [20]

>> foo( 10, 20, 30 )
x = 10
args[0] = 20
args[1] = 30
=> [20, 30]

>> foo( [10, 20, 30] )
x = [10, 20, 30]
=> []

>> foo( *[10, 20, 30] )
x = 10
args[0] = 20
args[1] = 30
=> [20, 30]
```

4.22.2 L'opérateur préfixe «&» pour la manipulation de blocs

Exemple Ruby 4.73 Utilisation de l'opérateur «&» pour rendre explicite un bloc comme paramètre d'une méthode.

```
>> # L'opérateur préfixe & utilise devant le dernier paramètre
    # rend explicite le bloc transmis à l'appel de la méthode
    # Ce paramètre est alors un objet Proc pouvant
    # être exécuté avec call.

>> def call_yield( x, &bloc )
    return x unless block_given?

    [ bloc.class, bloc.call(x), yield(x) ]
end
=> :call_yield

>> call_yield( 99 )
=> 99

>> call_yield( 99 ) { |x| x + 10 }
=> [Proc, 109, 109]
```

Exemple Ruby 4.74 Utilisation de l'opérateur «&» pour transformer un objet lambda ou Symbole en bloc.

```
>> # L'opérateur prefixe & devant une lambda expression
    # transforme l'objet Proc en un bloc.
    # Ce bloc peut alors transmis explicitement comme
    # dernier argument (argument additionnel en plus
    # des arguments non blocs explicites).
```

```
>> double = lambda { |x| 2 * x }
=> #<Proc:0x000000028b0950@(irb):24 (lambda)>
```

```
>> call_yield( 2 ) { |x| 2 * x }
=> [Proc, 4, 4]
```

```
>> call_yield( 2 ) double
SyntaxError: (irb):26: syntax error, unexpected tIDENTIFIER,
    ...
```

```
>> call_yield( 2 ) &double
TypeError: Proc can't be coerced into Fixnum
    ...
```

```
>> call_yield( 2, &double )
=> [Proc, 4, 4]
```

```
>> # Cette transformation s'applique meme lorsque le bloc
    # est implicite.
    # Et elle s'applique aussi aux symboles,
    # via un appel implicite a to_proc.

>> def yield_un_arg( x )
    yield( x )
end
=> :yield_un_arg

>> yield_un_arg( 24, &double )
=> 48

>> yield_un_arg( 24, &:even? )
=> true

>> # :s.to_proc == Proc.new { |o| o.s } (...ou presque)
>> yield_un_arg( 24, &:even?.to_proc )
=> true

>> yield_un_arg( 24, &:- )
ArgumentError: wrong number of arguments (0 for 1)
...

>> yield_un_arg( 24, &:-@ ) # Voir section suivante.
=> -24
```

4.22.3 Les opérateurs (préfixes) unaires

Exemple Ruby 4.75 Opérateurs (préfixes) unaires définis par le programmeur.

```
>> class Foo
      def +( autre )
        puts "self = #{self}; autre = #{autre}"
      end

      def +@
        puts "self = #{self}"
      end
    end

=> :+@

>> foo = Foo.new
=> #<Foo:0x000000019910c8>

>> foo + 10
self = #<Foo:0x000000019910c8>; autre = 10
=> nil

>> + foo
self = #<Foo:0x000000019910c8>
=> nil
```

4.22.4 Un mini irb en une seule ligne

```
$ ruby -n -e 'p eval($_)'  
10 + 30  
40  
:a.class  
Symbol  
puts "10"  
10  
nil  
^D
```

4.22.5 La méthode tap

```
tap {|x| ...} -> obj
```

Yields `self` to the block, and the return `self` of this method is to "`tap into`" a method operations on intermediate results within

```
class Object
  def tap
    yield self
    self
  end
end
```

```
$ cat tap.rb
```

```
p (1..10)
  .tap { |x| puts "Original: #{x}" }
  .to_a
  .tap { |x| puts "Array: #{x}" }
  .select { |x| x.even? }
  .tap { |x| puts "Paires: #{x}" }
  .map { |x| x * x }
  .tap { |x| puts "Carres: #{x}" }
```

```
$ ruby tap.rb
```

```
Original: 1..10
```

```
Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Paires: [2, 4, 6, 8, 10]
```

```
Carres: [4, 16, 36, 64, 100]
```

```
[4, 16, 36, 64, 100]
```


4.A Installation de Ruby sur votre machine

1. Obtenir la clé pour `rvm` et obtenir `rvm` :

```
$ gpg --keyserver hkp://keys.gnupg.net\  
      --recv-keys 409B6B1796C275462A1703113804BB82D39DC  
$ curl -sSL https://get.rvm.io | bash -s stable
```

2. Activer les fonctions associées à `rvm` :

```
$ source ~/.rvm/scripts/rvm
```

3. Pour la programmation parallèle avec la bibliothèque `PRuby` (cours INF5171/INF7235), il faut installer `jruby` — `rvm list` permet de vérifier qu'il est bien installé, puis il faut ensuite installer le *gem* `pruby` :

```
$ rvm install jruby-1.7
```

```
$ rvm list
```

```
$ wget http://www.labunix.uqam.ca/~tremblay/INF7235/pru
```

```
$ gem install pruby-0.3.0.gem
```

Important : Il ne faut pas installer `jruby-9`
— `pruby` n'a pas encore été adapté pour cette version.

4. Installer le *gem* `bundler` :

```
$ gem install bundler
```


4.B Le cadre de tests unitaires MiniTest

4.B.1 Tests unitaires et cadres de tests

Niveaux de tests

Différents niveaux de tests :

- Tests unitaires
- Tests d'intégration
- Tests de système
- Tests d'acceptation

Dans ce qui suit : *tests unitaires*.

Pratique professionnelle et tests

«code source = programme + tests»

Approches agiles et «développement dirigé par les tests» :

- Les tests doivent être écrits **avant** le code!
- *Never write a line of functional code without a broken test case. (K. Beck)*

Cadres de tests

Il faut que les tests puissent être exécutés **fréquemment** et de façon **automatique**

Outils qui permettent d'automatiser l'exécution des tests unitaires = **cadres de tests**

Caractéristique= on utilise des **assertions** :

```
assertEquals( expectedResult , value )  
assertEquals( expectedResult , value , precision )  
assertTrue( booleanExpression )  
assertNotNull( reference )  
etc .
```

Donc : aucun résultat n'est produit si le test ne détecte pas d'erreur.

4.B.2 Le cadre de tests MiniTest

Test dans le style «JUnit» :

```
class TestFoo < MiniTest::Unit::TestCase
  def setup
    @foo = Foo.new
  end

  def test_bar_est_initialement_0
    assert_equal 0, @foo.bar
  end

  ...
end
```

Test dans le style «RSpec» :

```
describe Foo do
  describe "#bar" do
    before do
      @foo = Foo.new
    end

    it "retourne une taille nulle lorsque
      @foo.bar.must_equal 0
    end
    ...
  end
  ...
end
```

4.B.3 Des spécifications MiniTest pour la classe Ensemble

Exemple Ruby 4.76 Une suite de tests pour la classe `Ensemble` (partie 1)

```
require 'minitest/autorun'
require 'minitest/spec'

require_relative 'ensemble'

describe Ensemble do
  before do
    @ens = Ensemble.new
  end

  describe '#contient?' do
    it "retourne faux quand un element n'est pas present" do
      refute @ens.contient? 10
    end

    it "retourne vrai apres qu'un element ait ete ajoute" do
      refute @ens.contient? 10
      @ens << 10
      assert @ens.contient? 10
    end
  end
end

# ...
```

Exemple Ruby 4.77 Une suite de tests pour la classe Ensemble (partie 2)

```
# ...

describe '#<<' do
  it "ajoute un element lorsque pas deja present" do
    @ens << 10
    assert @ens.contient? 10
  end

  it "laisse l'element ajoute lorsque deja present" do
    @ens << 10
    assert @ens.contient? 10

    @ens << 10
    assert @ens.contient? 10
  end

  it "retourne self ce qui permet de chainer des operation" do
    res = @ens << 10
    res.must_be_same_as @ens
  end
end

# ...
```

Exemple Ruby 4.78 Une suite de tests pour la classe Ensemble (partie 3)

```
# ...

describe '#cardinalite' do
  it "retourne 0 lorsque vide" do
    @ens.cardinalite.must_equal 0
  end

  it "retourne 1 lorsqu'un seul et meme element est ajoute" do
    @ens << 1
    @ens.cardinalite.must_equal 1

    @ens << 1 << 1 << 1
    @ens.cardinalite.must_equal 1
  end

  it "retourne le nombre d'elements distincts peu importe" do
    @ens << 1 << 1 << 1 << 2 << 2 << 1 << 2
    @ens.cardinalite.must_equal 2
  end
end
end
end
```

```
res.must_be_same_as @ens =  
  assert res.equal? @ens
```

```
@ens.cardinalite.must_equal 0 =  
  assert @ens.cardinalite == 0  
  assert_equal 0, @ens.cardinalite
```

Exemple Ruby 4.79 Des exemples d'exécution de la suite de tests pour la classe Ensemble.

=====

Execution ordinaire

=====

\$ ruby ensemble_spec.rb

Run options: --seed 43434

Running:

.....

Finished in 0.001556s, 5140.4367 runs/s, 7068.1005 assertions/s.

8 runs, 11 assertions, 0 failures, 0 errors, 0 skips

=====

Execution 'verbeuse'

=====

\$ ruby ensemble_spec.rb -v

Run options: -v --seed 18033

Running:

Ensemble::#<<#test_0003_retourne self ce qui permet de chainer des operations = 0.00

Ensemble::#<<#test_0001_ajoute un element lorsque pas deja present = 0.00 s = .

Ensemble::#<<#test_0002_laisse l'element ajoute lorsque deja present = 0.00 s = .

Ensemble::#contient?#test_0001_retourne faux quand un element n'est pas present = 0.0

Ensemble::#contient?#test_0002_retourne vrai apres qu'un element ait ete ajoute = 0.0

Ensemble::#cardinalite#test_0001_retourne 0 lorsque vide = 0.00 s = .

Ensemble::#cardinalite#test_0002_retourne 1 lorsqu'un seul et meme element est ajoute
1 ou plusieurs fois = 0.00 s = .

Ensemble::#cardinalite#test_0003_retourne le nombre d'elements distincts peu importe
le nombre de fois ajoutees = 0.00 s = .

Finished in 0.001686s, 4745.7382 runs/s, 6525.3900 assertions/s.

8 runs, 11 assertions, 0 failures, 0 errors, 0 skips

Exemple Ruby 4.80 Un exemple d'exécution de la suite de tests pour la classe `Ensemble` avec des échecs — la méthode `cardinalite` retourne toujours 0.

=====

Execution avec echecs

=====

\$ ruby ensemble_spec.rb

Run options: --seed 7910

Running:

...FF...

Finished in 0.001950s, 4101.7438 runs/s, 5127.1797 assertions/s.

1) Failure:

Ensemble::#cardinalite#test_0002_retourne 1 lorsqu'un seul et meme element est ajoute
1 ou plusieurs fois [ensemble_spec.rb:54]:

Expected: 1

Actual: 0

2) Failure:

Ensemble::#cardinalite#test_0003_retourne le nombre d'elements distincts peu importe
le nombre de fois ajoutees [ensemble_spec.rb:62]:

Expected: 2

Actual: 0

8 runs, 10 assertions, 2 failures, 0 errors, 0 skips

Methods

```
#must_be
#must_be_close_to
#must_be_empty
#must_be_instance_of
#must_be_kind_of
#must_be_nil
#must_be_same_as
#must_be_silent
#must_be_within_delta
#must_be_within_epsilon
#must_equal
#must_include
#must_match
#must_output
#must_raise
#must_respond_to
#must_send
#must_throw
#wont_be
#wont_be_close_to
#wont_be_empty
#wont_be_instance_of
#wont_be_kind_of
#wont_be_nil
#wont_be_same_as
#wont_be_within_delta
#wont_be_within_epsilon
#wont_equal
#wont_include
#wont_match
#wont_respond_to
```

Figure 4.8: La liste des *expectations* disponibles dans MiniTest. Source : <http://ruby-doc.org/stdlib-2.1.0/libdoc/minitest/rdoc/MiniTest/Expectations.html>.

Exemple Ruby 4.81 Quelques autres méthodes de MiniTest — dans le style avec *expectations*.

```
gem 'minitest'
require 'minitest/autorun'
require 'minitest/spec'

describe Array do
  let (:vide) { Array.new }

  before do
    @singleton_10 = Array.new << 10
  end

  describe ".new" do
    it "cree un tableau vide lorsque sans argument" do
      vide.must_be :empty?
    end
  end

  describe "#push" do
    it "ajoute un element, lequel devient inclu" do
      @singleton_10.must_include 10
    end
  end
end
```

```
describe "#size" do
  it "retourne 0 lorsque vide" do
    vide.size.must_equal 0
  end

  it "retourne 0 lorsque vide (bis)" do
    vide.size.must_be :==, 0
  end

  it "retourne > 0 lorsque non vide" do
    @singleton_10.size.must_be :>, 0
  end
end
```

```
describe "#to_s" do
  it "retourne '[]' lorsque vide" do
    vide.to_s
      .must_equal "[]"
  end

  it "retourne les elements separes par des virgules" do
    (vide << 10 << 20 << 30).to_s
      .must_equal "[10, 20, 30]"
  end

  it "retourne les elements separes par des virgules (bis)" do
    a = vide << 10 << 20 << 30
    virgule = /\s*,\s*/

    a.to_s
      .must_match
        /\^[\\s*10#{virgule}20#{virgule}30\\s*\]$//
  end
end
end
```

4.C Règles de style Ruby

Pourquoi des **conventions** sur le style de programmation sont importantes :

- *80% of the lifetime cost of a piece of software goes to maintenance.*
- *Hardly any software is maintained for its whole life by the original author.*
- *Code conventions improve the readability of the software, **allowing engineers to understand new code more quickly and thoroughly.***

<http://www.oracle.com/technetwork/java/index-135089.html>

Une présentation assez complète des règles spécifiques à Ruby :

<https://github.com/styleguide/ruby>

Principales règles **que vous devriez respecter** :

- Utilisation du *snake_case* vs. *CamelCase* :
 - NomDeClasse
 - NOM_DE_CONSTANTE
 - nom_de_methode
 - nom_de_parametre_ou_variable

- **Indentation** avec **des (2) espaces blancs seulement**, pas de caractères de tabulation
- Jamais de blancs à la fin d'une ligne.
- Des blancs autour des opérateurs binaires (y compris $=$), après les virgules, les deux points et les points-virgules, autour des $\{$ et avant les $\}$.
- Pas de blanc avant ou après $[$ et $]$, ou après $!$.

- **Jamais de** `then` pour une instruction `if/unless` et **jamais de parenthèses autour des conditions** :

<code># NON</code>	<code># OK</code>
<code>if (condition) then</code>	<code>if condition</code>
<code>...</code>	<code>...</code>
<code>end</code>	<code>end</code>

- **Pas de parenthèses** si aucun argument :

```
def une_methode_sans_arg
  ...
end

def une_methode_avec_args( arg1, ..., argk )
  ...
end

# NON
une_methode_sans_arg()

# OK
une_methode_sans_arg
```

- On utilise une **garde** if/unless quand il y a une seule instruction simple/courte :

```
# NON      # OK
if condition    une_instr if condition
    une_instr
end
```


- On utilise `unless` si la condition est négative :

```
# NON
if !expr
  ...
  res
end
```

```
# OK
unless expr
  ...
  res
end
```

```
# NON
unless expr
  ... si faux ...
else
  ... si vrai ...
end
```

```
# OK
if expr
  ... si vrai ...
else
  ... si faux ...
end
```

- **Pour les blocs**, on utilise `{...}` lorsque le corps peut s'écrire sur une seule ligne.

Autrement, on utilise `do ... end`.

<pre># NON col.map do x ... end col.map { x ... }</pre>	<pre># OK col.map { x ... } col.map do x ... end</pre>
--	---

- **On utilise return** seulement pour retourner au milieu d'une méthode :

```
# NON
if expr
  ...
  return res
else
  ...
  return autre_res
end
```

```
# OK
if expr
  ...
  res
else
  ...
  autre_res
end
```

```
# NON
def m_rec( ... )
  if expr
    return res_base
  else
    ...
    return res_rec
  end
end
```

```
# OK
def m_rec( ... )
  return res_base if expr
  ...
  res_rec
end
```

- Dans une classe `C`, on utilise `def self.m` pour définir une méthode de classe `m`.

- Pour les objets de classe `Hash`, on utilise des `Symbols` comme clés :

```
hash = {  
  :cle1 => defn1,  
  :cle2 => defn2,  
  ...  
  :clek => defnk  
}
```

Quelques remarques additionnelles concernant les exemples :

- Des espaces sont mis autour des parenthèses des définitions de méthodes :

```
# Style suggere dans le guide.  
def methode(a, b, c)  
    ...  
end
```

```
# Style dans le materiel de cours  
def methode( a, b, c )  
    ...  
end
```

Quelques règles additionnelles

Les règles qui suivent sont basées sur des erreurs typiques rencontrées dans les devoirs.

- **Les méthodes `map`, `select`, `reject` doivent être utilisées pour produire une nouvelle collection**, et non pour des effets de bord.

NON

```
res = []  
a.map { |x| res << foo(x) }
```

OK

```
a.map { |x| foo(x) }
```

- On utilise une instruction avec garde seulement si l'instruction s'écrit sur une seule ligne :

```
instr if condition      # OK si instr cour
```

Si l'instruction est trop longue, alors on utilise une **instruction** if :

```
if condition
    instruction
end
```


- Il faut éviter les effets de bord dans les gardes :

```
puts x if x = ARGV.shift      # NON !
```

Dans certains cas simples, on peut accepter une affectation en début d'une instruction :

```
if x = ARGV.shift
  puts x
end
```

- On utilise une instruction avec garde **seulement si le cas complémentaire n'a pas besoin d'être traité** :

```
instr1 if condition
instr2 unless condition    # NON !
```

Autrement, on utilise plutôt une instruction `if` :

```
if condition
  instr1
else
  instr2
end
```

- Il est correct d'**enchaîner** plusieurs appels de méthodes. :

```
# OK seulement si *très* court
res = a.select { |x| ... }.map { |x| ... }.sort.join

# Preferable lorsque plusieurs appels: plus facile
a lire, a modifier,
# pour ajouter un autre appel, etc.
# OK
res = a.select { |x| ... }
      .map { |x| ... }
      .sort
      .join
```

- Dans le bloc transmis à `reduce`, **la mise à jour de l'accumulateur se fait implicitement** :

NON

```
(1..n).reduce(1.0) { |res, x| x == 0 ? res : res /= x }
```

OK

```
(1..n).reduce(1.0) { |res, x| x == 0 ? res : res / x }
```

Note : La 1^{ère} expression fonctionne parce que :

```
res /= x # est la meme chose que  
res = res / x
```

```
# et parce que  
(res = v) == v
```

4.D Méthodes attr_reader et attr_writer

Exemple Ruby 4.82 Une définition des méthodes attr_reader et attr_writer.

```
class Class
  def attr_reader( attr )
    self.class_eval "
      def #{attr}
        @#{attr}
      end
    "
  end

  def attr_writer( attr )
    self.class_eval "
      def #{attr}=( v )
        @#{attr} = v
      end
    "
  end
end

class Foo
  attr_reader :bar
  attr_writer :bar

  def initialize
    self.bar = 0
  end
end

foo = Foo.new
foo.bar += 3
```

Exemple Ruby 4.83 Une autre définition des méthodes `attr_reader` et `attr_writer`.

```
class Class
  def attr_reader_( attr )
    self.class_eval do
      define_method attr do
        instance_variable_get "@#{attr}"
      end
    end
  end

  def attr_writer_( attr )
    self.class_eval do
      define_method "#{attr}=" do |v|
        instance_variable_set( "@#{attr}", v )
      end
    end
  end
end

class Foo
  attr_reader :bar
  attr_writer :bar

  def initialize
    self.bar = 0
  end
end
```

4.E Interprétation vs. compilation

Soit l'affirmation suivante : «Ruby est un langage interprété».

Cette affirmation est-elle vraie ou fausse?

Exercice 4.13: Ruby, un langage interprété?

Pourquoi les performances d'un programme Ruby sont-elles généralement moins bonnes (programme plus lent ☹) que celles d'un programme Java?

Exercice 4.14: Performances de Ruby.