



→ Digitalize society



Formation F# 5.0

*Les collections*



Décembre 2021

SOAT.FR

# About me



## Romain DENEAU

- SOAT depuis 2009
- Senior Developer C# F# TypeScript
- Passionné de Craft
- Auteur sur le blog de SOAT



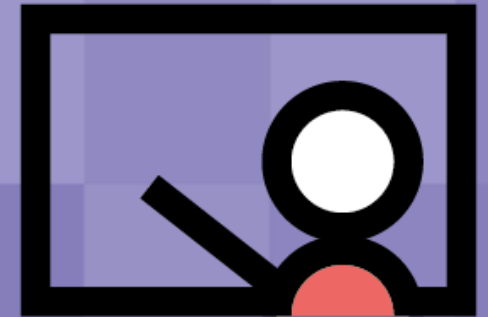
DeneauRomain



rdeneau

# Sommaire

- Vue d'ensemble
- Types
- Fonctions



# 1. Vue d'ensemble



# Types et Modules

5 collections usuelles en F# avec leur module associé

Module	Type	Alias BCL	Immutable	Trié par
Array	'T array	$\equiv$ Array<T>	✗	Ordre ajout
List	'T list	$\approx$ ImmutableList<T>	✓	Ordre ajout
Seq	seq<'T>	$\equiv$ IEnumerable<T>	✓	Ordre ajout
Set	Set<'T>	$\approx$ ImmutableHashSet<T>	✓	Valeur
Map	Map<'K, 'V>	$\approx$ ImmutableDictionary<K, V>	✓	Clé



# Homogénéité des fonctions

Communes aux 5 modules :

- `empty` / `isEmpty`, `exists` / `forall`
- `find` / `tryFind`, `pick` / `tryPick`, `contains` (`containsKey` pour `Map`)
- `map` / `iter`, `filter`, `fold`

Communes à `Array`, `List`, `Seq` :

- `append` / `concat`, `choose`, `collect`
- `item`, `head`, `last`
- `take`, `skip`
- ... *une centaine de fonctions en tout !*

# Homogénéité de la syntaxe

Type	Éléments	Range	Comprehension
Array	[  1; 2  ]	[  1 .. 5  ]	...
List	[ 1; 2 ]	[ 1 .. 5 ]	...
Seq	seq { 1; 2 }	seq { 1 .. 5 }	...
Set	set [ 1; 2 ]	set [ 1 .. 5 ]	...

# ⚠ Piège de la syntaxe

Les crochets `[]` sont utilisés pour :

- *Valeur* : instance d'une liste `[ 1; 2 ]` (de type `int list`)
- *Type* : tableau `int []`, par ex. de `[] 1; 2 []`

## 👉 Recommendations

- Bien distinguer type vs valeur !
- Préférer écrire `int array` plutôt que `int []`
  - *N.B.* En console FSI, le type affiché est encore `int []`



# Création par *Comprehension*

- Syntaxe similaire à boucle `for`
- Même principe que générateurs en C#, JS
  - Mot clé `yield` mais souvent optionnel (F# 4.7 / .NET Core 3)
  - Mot clé `yield!`  $\equiv$  `yield*` JS
  - Fonctionne pour toutes les collections 👍

# Comprehension : exemples

```
// Syntaxes équivalentes
seq { for i in 1 .. 10 → i * i }           // Plutôt obsolète
seq { for i in 1 .. 10 do yield i * i }    // 'yield' explicite
seq { for i in 1 .. 10 do i * i }          // 'yield' omis 👍

// Avec 'if'
let halfEvens =
    [ for i in [1..10] do
        if (i % 2) = 0 then i / 2 ] // [1; 2; 3; 4; 5]

// 'for' imbriqués
let pairs =
    [ for i in [1..3] do
        for j in [1..3] do
            (i, j) ] // [(1, 1); (1, 2); (1, 3); (2, 1); ... (3, 3)]
```

F#

## Comprehension : exemples (2)

```
// Même ici les 'yield' peuvent être omis 👍  
let twoToNine =  
    [ for i in [1; 4; 7] do  
        if i > 1 then i  
        i + 1  
        i + 2 ] // [2; 3; 4; 5; 6; 7; 8; 9]
```

F#

**yield!** permet d'aplatir des collections imbriquées :

```
let oneToSix =  
    [ for i in [1; 3; 5] do  
        yield! set [i; i+1] ]
```

F#

# 2. ■ Les Types



# Type **List**

Implémentée sous forme de **liste simplement chaînée** :

→ 1 liste = 1 élément (*Head*) + 1 sous-liste (*Tail*)

→ Construction nommée *Cons* et notée `::`

Pour éviter récursion infinie, besoin d'un cas de "sortie" :

→ Liste vide nommée *Empty* et notée `[]`

👉 **Type union générique et récursif :**

```
type List<'T> =  
    | ( [] )  
    | ( :: ) of head: 'T * tail: List<'T>
```

F#

# List : littéraux

Nb	Notation	Notation explicite	Signification
0	<code>[]</code>	<code>[]</code>	Empty
1	<code>[1]</code>	<code>1 :: []</code>	Cons (1, Empty)
2	<code>[2; 1]</code>	<code>2 :: 1 :: []</code>	Cons (2, Cons (1, Empty))
3	<code>[3; 2; 1]</code>	<code>3 :: 2 :: 1 :: []</code>	Cons (3, Cons (2, Cons (1, Empty)))

Vérification par décompilation avec [SharpLab.io](https://sharp-lab.io) :

```
// ...  
v1@2 = FSharpList<int>.Cons(1, FSharpList<int>.Empty);  
v2@3 = FSharpList<int>.Cons(2, FSharpList<int>.Cons(1, FSharpList<int>.Empty));  
// ...
```

C#

# List : immuable

Il n'est pas possible de modifier une liste existante.

→ C'est cela qui permet de l'implémenter en liste chaînée.

💡 L'idée est de créer une nouvelle liste pour signifier un changement.

→ Utiliser les opérateurs *Cons* ( `::` ) et *Append* ( `@` ) 📌

# List : initialisation

```
// Range: Start..End (Step=1)
let numFromOneToFive = [1..5] // [1; 2; 3; 4; 5]

// Range: Start..Step..End
let oddFromOneToNine = [1..2..9] // [1; 3; 5; 7; 9]

// Comprehension
let pairs =
    [ for i in [1..3] do
      for j in [1..3] do
        (i, j) ] // [(1, 1); (1, 2); (1, 3); (2, 1); ... (3, 3)]
```

F#



# List - Exercices

## 1. Implémenter la fonction `rev`

Inverse une liste : `rev [1; 2; 3] ≡ [3; 2; 1]`

## 2. Implémenter la fonction `map`

Transforme chaque élément : `[1; 2; 3] ▷ map ((+) 1) ≡ [2; 3; 4]`

### Astuces

- Pattern matching liste vide `[]` ou *Cons* `head :: tail`
- Sous-fonction (*tail-*) *recursive*

 5'



# List - Exercices - Solution


```
let rev list =  
    let rec loop acc rest =  
        match rest with  
        | [] → acc  
        | x :: xs → loop (x :: acc) xs  
    loop [] list  
  
let map f list =  
    let rec loop acc rest =  
        match rest with  
        | [] → acc  
        | x :: xs → loop (f x :: acc) xs  
    list ▷ loop [] ▷ rev
```


F#

💡 Vérification avec [sharplab.io](https://sharplab.io) de la *tail recursion* compilée en boucle `while`

# List - Exercices - Tests

```
// Tests en console FSI F#  
let (==) actual expected =  
    if actual = expected  
    then printfn $" {actual}"  
    else printfn $" {actual} ≠ {expected}"
```

```
[1..3] ▷ rev == [3; 2; 1];;  
//  [3; 2; 1]
```

```
[1..3] ▷ map ((+) 1) == [2; 3; 4];;  
//  [2; 3; 4]
```



# Type Array

- Différences / `List` : mutable, taille fixe, accès indexé en  $O(1)$
- Signature générique : `'T array` (*récemment recommandée*) ou `'T []`
- Littéral et *comprehension* : similaires à `List`

```
// Littéral
[1; 2; 3; 4; 5] // val it : int [] = [1; 2; 3; 4; 5]

// Comprehension using range
[1 .. 5] = [1; 2; 3; 4; 5] // true
[1 .. 3 .. 10] = [1; 4; 7; 10] // true

// Comprehension using generator
[for a in 1 .. 5 do (a, a * 2)]
// [(1, 2); (2, 4); (3, 6); (4, 8); (5, 10)]
```

F#

# Array : accès indexé & mutation

Accès par index : `my-array.[my-index]`

⚠ **Piège** : ne pas oublier le `.` avant les crochets `[]` !

🎁 **F# 6.0** supporte sans le `.` : `my-array[my-index]`

```
let names = [| "Juliet"; "Monique"; "Rachelle"; "Tara"; "Sophia" |]
names.[4] ← "Kristen" // "Rachelle"
names      // [| "Juliet"; "Monique"; "Rachelle"; "Tara"; "Kristen" |]
           //                ^^^^^^^
```

F#

## Array : *slicing*

Renvoie un sous-tableau entre les indices `start..end` optionnels

```
let names = ["0: Juliet"; "1: Monique"; "2: Rachelle"; "3: Tara"; "4: Sophia"]  
  
names.[1..3] // ["1: Monique"; "2: Rachelle"; "3: Tara"]  
names.[2..]  // ["2: Rachelle"; "3: Tara"; "4: Sophia"]  
names[..3]   // ["0: Juliet"; "1: Monique"; "2: Rachelle"; "3: Tara"]
```

F#

💡 Marche aussi avec une `string` : `"012345".[1..3] ≡ "123"`



## Type Seq

```
type Seq<'T> = IEnumerable<'T>
```

→ Série d'éléments de même type

*Lazy* : séquence construite au fur et à mesure lors de son itération

≠ **List** construite dès la déclaration

→ Peut offrir de meilleures performances qu'un **List** pour une collection avec beaucoup d'éléments et qu'on ne souhaite pas parcourir entièrement.

# Seq - Syntaxe

```
seq { comprehension }
```

```
seq { yield 1; yield 2 } // 'yield' explicites 😞  
seq { 1; 2; 3; 5; 8; 13 } // 'yield' implicites 👍
```

F#

```
// Range  
seq { 1 .. 10 } // seq [1; 2; 3; 4; ... ]  
seq { 1 .. 2 .. 10 } // seq [1; 3; 5; 7; ... ]  
  
// Générateur  
seq { for a in 1 .. 5 do (a, a * 2) }  
// seq [(1, 2); (2, 4); (3, 6); (4, 8); ... ]
```



# Seq - Séquence infinie

**Option 1** : appeler la fonction `Seq.initInfinite` :

→ `Seq.initInfinite : (initializer: (index: int) → 'T) → seq<'T>`

→ Paramètre `initializer` sert à créer l'élément d'index ( $\geq 0$ ) spécifié

**Option 2** : écrire une fonction récursive générant la séquence

```
// Option 1
let seqOfSquares = Seq.initInfinite (fun i → i * i)

// Option 2
let seqOfSquares' =
    let rec loop n = seq { yield n * n; yield! loop (n+1) }
    loop 0

// Test
let firstTenSquares = seqOfSquares ▷ Seq.take 5 ▷ List.ofSeq // [0; 1; 4; 9; 16]
```

F#



# Type Set

Collection auto-ordonnée d'éléments uniques (*sans doublon*)  
→ Implémentée sous forme d'arbre binaire

```
// Création
set [ 2; 9; 4; 2 ]           // set [2; 4; 9]  // 🖐️ Élément 2 dédoublonné
Set.ofArray [ 1; 3 ]         // set [1; 3]
Set.ofList [ 1; 3 ]          // set [1; 3]
seq { 1; 3 } ▷ Set.ofSeq     // set [1; 3]

// Ajout/retrait d'élément
Set.empty                   // set []
▷ Set.add 2                  // set [2]
▷ Set.remove 9               // set [2]      // 🖐️ Pas d'exception
▷ Set.add 9                  // set [2; 9]
▷ Set.remove 9               // set [2]
```

F#

# Set : informations

→ count, minElement, maxElement

```
let oneToFive = set [1..5]           // set [1; 2; 3; 4; 5]

// Nombre d'éléments : propriété `Count` ou fonction `Set.count` - ⚠ O(N)
// 🙅 Ne pas confondre avec `Xxx.length` pour Array, List, Seq
let nb = Set.count oneToFive // 5

// Élément min, max
let min = oneToFive ▷ Set.minElement // 1
let max = oneToFive ▷ Set.maxElement // 5
```

F#

# Set : opérations

→ **Union, Différence, Intersection** (*idem ensembles en Math*)

Opération	?	Opérateur	Fonction 2 sets	Fonction N sets
Union	$\cup$	+	<code>Set.union</code>	<code>Set.unionMany</code>
Différence	$\ominus$	-	<code>Set.difference</code>	<code>Set.differenceMany</code>
Intersection	$\cap$	$\times$	<code>Set.intersect</code>	<code>Set.intersectMany</code>

# Set : opérations - exemples

```
let oneToFive = set [1..5]           // A - set [1; 2; 3; 4; 5]
let evenToSix = set [2; 4; 6]        // B - set [2; 4; 6]

let union = oneToFive + evenToSix    // set [1; 2; 3; 4; 5; 6]
let diff = oneToFive - evenToSix     // set [1; 3; 5]
let inter = Set.intersect oneToFive evenToSix // set [2; 4]
```

F#

Valeur	Union	Différence	Intersection
A	[ 1 2 3 4 5 ]	[ 1 2 3 4 5 ]	[ 1 2 3 4 5 ]
B	[ 2 4 6 ]	[ 2 4 6 ]	[ 2 4 6 ]
A ? B	[ 1 2 3 4 5 6 ]	[ 1 3 5 ]	[ 2 4 ]



# Type Map

Tableau associatif { *Clé* → *Valeur* }  $\simeq$  **Dictionary** immutable en C#

```
// Création : depuis collection de tuples (key, val)
// → Fonction `Map.ofXxx` (Array, List, Seq)
let map1 = seq { (2, "A"); (1, "B") } ▷ Map.ofSeq
// → Constructeur `Map(tuples)`
let map2 = Map [ (2, "A"); (1, "B"); (3, "C"); (3, "D") ]
// map [(1, "B"); (2, "A"); (3, "D")]
// 🖐️ Ordonnés par clés (1, 2, 3) et dédoublonnés en last win - cf. { 3 → "D" }

// Ajout/retrait d'élément
Map.empty // map []
▷ Map.add 2 "A" // map [(2, "A")]
▷ Map.remove 5 // map [(2, "A")] // 🖐️ Pas d'exception si clé absente
▷ Map.add 9 "B" // map [(2, "A"); (9, "B")]
▷ Map.remove 2 // map [(9, "B")]
```

F#

# Map : accès par clé

```
let table = Map [ (2, "A"); (1, "B"); (3, "D") ]  
  
// Syntaxe `[key]`  
table.[1] // "B" // ⚠️ `1` est bien une clé et pas un indice  
table.[0] // ⚡ KeyNotFoundException  
  
// Fonction `Map.find` : renvoie valeur ou ⚡ si clé absente  
table ▷ Map.find 3 // "D"  
  
// Fonction `Map.tryFind` : renvoie `V option`  
table ▷ Map.tryFind 3 // Some "D"  
table ▷ Map.tryFind 9 // None
```

F#

# Map : performance des lookups ( find )

🔗 *High Performance Collections in F#* <https://kutt.it/dxDOi7> (Jan 2021)

## Map VS Dictionary

Fonction `readOnlyDict` permet de créer rapidement un `ReadOnlyDictionary`

→ à partir d'une séquence de tuples `key, item`

→ très performant : 10x plus rapide que `Map` pour le *lookup*

## Dictionary VS Array

→ `Array` suffit si peu de lookups (< 100) et peu d'éléments (< 100)

→ `Dictionary` sinon



## Types `Set` et `Map` vs `IComparable`

Ne marchent que si éléments (d'un `Set`) / clés (d'une `Map`) sont **comparables** !


 Compatibles avec tous les types F# (cf. *égalité structurelle*)


 Pour les classes : implémenter `IComparable`

# 3 ■ Les Fonctions



# Accès à un élément

↓ Accès \ Renvoie →	'T ou 	'T option
Par index	<code>list.[index]</code>	
	<code>item index</code>	<code>tryItem index</code>
Premier élément	<code>head</code>	<code>tryHead</code>
Dernier élément	<code>last</code>	<code>tryLast</code>

- Fonctions à préfixer par le module associé : `Array`, `List` ou `Seq`
- Dernier paramètre, la "collection", omis par concision
-  `ArgumentException` ou `IndexOutOfRangeException`

```
[1; 2] ▷ List.tryHead    // Some 1  
[1; 2] ▷ List.tryItem 2  // None
```

F#

# Accès à un élément : coût ⚠

Fonction \ Module	Array	List	Seq
head	$O(1)$	$O(1)$	$O(1)$
item	$O(1)$	$O(n)$ !	$O(n)$ !
last	$O(1)$	$O(n)$ !	$O(n)$ !
length	$O(1)$	$O(n)$ !	$O(n)$ !

# Combiner des collections

Fonction	Paramètre(s)	Taille finale
<code>append</code> / <code>@</code>	2 collections de tailles $N_1$ et $N_2$	$N_1 + N_2$
<code>concat</code>	K collections de tailles $N_1..N_k$	$N_1 + N_2 + \dots + N_k$
<code>zip</code>	2 collections de même taille $N$ !	$N$ tuples $(x_1, x_2)$

💡 `@` = opérateur infixé alias de `List.append` uniquement (~~Array, Seq~~)

```
List.append [1;2;3] [4;5;6] // [1; 2; 3; 4; 5; 6]
[1;2;3] @ [4;5;6]          // idem

List.concat [ [1]; [2; 3] ] // [1; 2; 3]

List.zip [1; 2] ['a'; 'b'] // [(1, 'a'); (2, 'b')]
```

F#

## List : :: VS @

*Cons* `1 :: [2; 3]`


- Élément ajouté en tête de liste → liste paraît en ordre inverse 😞
- Mais opération en  **$O(1)$**  👍 -- (*Tail conservée*)

*Append* `[1] @ [2; 3]`

- Liste en ordre normal
- Mais opération en  **$O(n)$**  ! -- (*Nouvelle Tail à chaque niveau*)

# Recherche d'un élément

Via un prédicat `f : 'T → bool` :

Quel élément \ Renvoie	'T ou 	'T option
Premier trouvé	<code>find</code>	<code>tryFind</code>
Dernier trouvé	<code>findBack</code>	<code>tryFindBack</code>
Index du 1er trouvé	<code>findIndex</code>	<code>tryFindIndex</code>
Index du der trouvé	<code>findIndexBack</code>	<code>tryFindIndexBack</code>

```
[1; 2] ▷ List.find (fun x → x < 2) // 1
[1; 2] ▷ List.tryFind (fun x → x ≥ 2) // Some 2
[1; 2] ▷ List.tryFind (fun x → x > 2) // None
```

F#

# Recherche d'éléments

Recherche	Combien d'éléments	Méthode
Par valeur	Au moins un	<code>contains value</code>
Par prédicat <code>f</code>	Au moins un	<code>exists f</code>
"	Tous	<code>forall f</code>

```
[1; 2] ▷ List.contains 0 // false
[1; 2] ▷ List.contains 1 // true
[1; 2] ▷ List.exists (fun x → x ≥ 2) // true
[1; 2] ▷ List.forall (fun x → x ≥ 2) // false
```

F#



# Sélection d'éléments

Quels éléments	Par nombre	Par prédicat <code>f</code>
Tous ceux trouvés		<code>filter f</code>
Premiers ignorés	<code>skip n</code>	<code>skipWhile f</code>
Premiers trouvés	<code>take n</code>	<code>takeWhile f</code>
	<code>truncate n</code>	

## 👉 Notes :

- Avec `skip` et `take`, ⚡ exception si `n > list.Length` ; pas avec `truncate`
- Alternative pour `Array` : sélection par *Range* `arr.[2..5]`

# Mapping d'éléments

Fonction prenant en entrée :

- Une fonction de mapping `f`
- Une collection d'éléments de type `'T`

Fonction	Mapping <code>f</code>	Retour	Quel(s) élément(s)
<code>map</code>	<code>'T → 'U</code>	<code>'U list</code>	Autant d'éléments
<code>mapI</code>	<code>int → 'T → 'U</code>	<code>'U list</code>	idem
<code>collect</code>	<code>'T → 'U list</code>	<code>'U list</code>	<i>flatMap</i>
<code>choose</code>	<code>'T → 'U option</code>	<code>'U list</code>	Moins d'éléments
<code>pick</code>	<code>'T → 'U option</code>	<code>'U</code>	1er élément ou 💣
<code>tryPick</code>	<code>'T → 'U option</code>	<code>'U option</code>	1er élément

## map VS mapi

`mapi`  $\equiv$  `map` *with index*

`map` : mapping `'T → 'U`

→ Opère sur valeur de chaque élément

`mapi` : mapping `int → 'T → 'U`

→ Opère sur index et valeur de chaque élément

```
["A"; "B"]  
▷ List.mapi (fun i x → $"{i+1}. {x}")  
// ["1. A"; "2. B"]
```

F#

## Alternative à `mapi`

Hormis `map` et `iter`, aucune fonction `xxx` n'a de variante en `xxxi`.

💡 Utiliser `indexed` pour obtenir les éléments avec leur index

```
let isOk (i, x) = i ≥ 1 && x ≤ "C"
```

F#

```
["A"; "B"; "C"; "D"]
```

```
▷ List.indexed // [ (0, "A"); (1, "B"); (2, "C"); (3, "D") ]
```

```
▷ List.filter isOk // [ (1, "B"); (2, "C") ]
```

```
▷ List.map snd // [ "B" ; "C" ]
```

## map vs iter

`iter`  $\equiv$  `map` sans *mapping*: `f: 'T  $\rightarrow$  unit` (= *Action* en C#)

👉 Même si `map` marche, utiliser `iter` pour la compréhension du code  
→ Révèle intention d'itérer/parcourir la liste plutôt que de mapper ses éléments

```
// ❌ À éviter
["A"; "B"; "C"] ▷ List.mapi (fun i x → printfn $"Item #{i}: {x}")

// ✅ Recommandé
["A"; "B"; "C"] ▷ List.iteri (fun i x → printfn $"Item #{i}: {x}")
// Item #0: A
// Item #1: B
// Item #2: C
```

F#

## `choose`, `pick`, `tryPick`

Mapping `'T → 'U option`

- Peut échouer en fonction des éléments
- Renvoie `Some value` pour indiquer le succès du mapping
- Exemple : `tryParseInt: string → int option`

`choose` et `pick` *unwrap* la/les valeurs dans les `Some`

`pick` émet une exception 🌟 si aucun `Some` (= que des `None`)  
`tryPick` renvoie tel quel le 1er `Some`

## choose, pick, tryPick - Examples

```
let tryParseInt (s: string) =  
    match System.Int32.TryParse(s) with  
    | true, i → Some i  
    | false, _ → None
```

F#

```
["1"; "2"; "?"] ▷ List.choose tryParseInt // [1; 2]  
["1"; "2"; "?"] ▷ List.pick tryParseInt   // 1  
["1"; "2"; "?"] ▷ List.tryPick tryParseInt // Some 1
```

# Sélection vs mapping

→ `filter` ou `choose` ?

→ `find` / `tryFind` ou `pick` / `tryPick` ?

`filter`, `find` / `tryFind` opèrent avec un **prédicat** `'T → bool`, sans mapping

`choose`, `pick` / `tryPick` opèrent avec un **mapping** `'T → 'U option`



# Sélection vs mapping (2)

→ `filter` ou `find` / `tryFind` ?

→ `choose` ou `pick` / `tryPick` ?

`filter`, `choose` renvoient **tous** les éléments trouvés/mappés

`find`, `pick` ne renvoient que le **1er** élément trouvé/mappé

# Agrégation : fonctions spécialisées

Opération	Sur élément	Sur projection <code>'T → 'U</code>
Maximum	<code>max</code>	<code>maxBy projection</code>
Minimum	<code>min</code>	<code>minBy projection</code>
Somme	<code>sum</code>	<code>sumBy projection</code>
Moyenne	<code>average</code>	<code>averageBy projection</code>
Décompte	<code>length</code>	<code>countBy projection</code>

```
[1; 2; 3] ▷ List.max // 3
[ (1,"a"); (2,"b"); (3,"c") ] ▷ List.sumBy fst // 6
[ (1,"a"); (2,"b"); (3,"c") ] ▷ List.map fst ▷ List.sum // Equivalent explicite
```

F#

# Agrégation : fonctions génériques

- `fold` (f: 'U → 'T → 'U) (seed: 'U) list
- `foldBack` (f: 'T → 'U → 'U) list (seed: 'U)
- `reduce` (f: 'T → 'T → 'T) list
- `reduceBack` (f: 'T → 'T → 'T) list

👉 `f` prend 2 paramètres : un "accumulateur" `acc` et l'élément courant `x`

⚠ Fonctions `xxxBack` : tout est inversé / fonctions `xxx` !

- Parcours des éléments en sens inverse : dernier → 1er élément
- Paramètres `seed` et `list` inversés (pour `foldBack` vs `fold`)
- Paramètres `acc` et `x` de `f` inversés

🌟 `reduceXxx` plante si liste vide car 1er élément utilisé en tant que `seed`

# Agrégation : fonctions génériques (2)

Exemples :

```
[ "a"; "b"; "c" ] ▷ List.reduce (+) // "abc"
[ 1; 2; 3 ] ▷ List.reduce ( * ) // 6

[1;2;3;4] ▷ List.reduce (fun acc x → 10 * acc + x) // 1234
[1;2;3;4] ▷ List.reduceBack (fun x acc → 10 * acc + x) // 4321

( "", [1;2;3;4] ) ▷ List.fold (fun acc x → $"{acc}{x}") // "1234"
([1;2;3;4], "" ) ▷ List.foldBack (fun x acc → $"{acc}{x}") // "4321"
```

F#

# Changer l'ordre des éléments

Opération	Sur élément	Sur projection <code>'T → 'U</code>
Inversion	<code>rev list</code>	
Tri ascendant	<code>sort list</code>	<code>sortBy f list</code>
Tri descendant	<code>sortDescending list</code>	<code>sortDescendingBy f list</code>
Tri personnalisé	<code>sortWith comparer list</code>	

```
[1..5] ▷ List.rev // [5; 4; 3; 2; 1]
[2; 4; 1; 3; 5] ▷ List.sort // [1..5]
["b1"; "c3"; "a2"] ▷ List.sortBy (fun x → x.[0]) // ["a2"; "b1"; "c3"] cf. a < b < c
["b1"; "c3"; "a2"] ▷ List.sortBy (fun x → x.[1]) // ["b1"; "a2"; "c3"] cf. 1 < 2 < 3
```

F#

# Séparer

💡 Les éléments sont répartis en groupes.

Opération	Résultat ( <i>; omis</i> )	Remarque
<code>[1..10]</code>	<code>[ 1 2 3 4 5 6 7 8 9 10 ]</code>	<code>length = 10</code>
<code>chunkBySize 3</code>	<code>[[1 2 3] [4 5 6] [7 8 9] [10]]</code>	<code>forall: length ≤ 3</code>
<code>splitInto 3</code>	<code>[[1 2 3 4] [5 6 7] [8 9 10]]</code>	<code>length ≤ 3</code>
<code>splitAt 3</code>	<code>([1 2 3],[4 5 6 7 8 9 10])</code>	Tuple <b>!</b>

# Grouper les éléments - Par taille

💡 Les éléments peuvent être **dupliqués** dans différents groupes.

Opération	Résultat ( ' et ; omis)	Remarque
<code>[1..5]</code>	<code>[ 1 2 3 4 5 ]</code>	
<code>pairwise</code>	<code>[(1,2) (2,3) (3,4) (4,5)]</code>	Tuple !
<code>windowed 2</code>	<code>[[1 2] [2 3] [3 4] [4 5]]</code>	Tableau de tableaux de 2
<code>windowed 3</code>	<code>[[1 2 3] [2 3 4] [3 4 5]]</code>	Tableau de tableaux de 3

# Grouper les éléments - Par critère

Opération	Critère	Retour
<code>partition</code>	<code>predicate: 'T → bool</code>	<code>('T list * 'T list)</code> → 1 tuple <code>([OKs], [KOs])</code>
<code>groupBy</code>	<code>projection: 'T → 'K</code>	<code>('K * 'T list) list</code> → N tuples <code>[(clé, [éléments associés])]</code>

```
let isOdd i = (i % 2 = 1)
[1..10] ▷ List.partition isOdd // ( [1; 3; 5; 7; 9] , [2; 4; 6; 8; 10] )
[1..10] ▷ List.groupBy isOdd // [ (true, [1; 3; 5; 7; 9]); (false, [2; 4; 6; 8; 10]) ]

let firstLetter (s: string) = s.[0]
["apple"; "alice"; "bob"; "carrot"] ▷ List.groupBy firstLetter
// [('a', ["apple"; "alice"]); ('b', ["bob"]); ('c', ["carrot"])]
```

F#



# Changer de type de collection

Au choix : `Dest.ofSource` ou `Source.toDest`

De / vers	Array	List	Seq
Array	×	List.ofArray	Seq.ofArray
	×	Array.toList	Array.toSeq
List	Array.ofList	×	Seq.ofList
	List.toArray	×	List.toSeq
Seq	Array.ofSeq	List.ofSeq	×
	Seq.toArray	Seq.toList	×

# Fonctions vs compréhension

Les fonctions de `List` / `Array` / `Seq` peuvent souvent être remplacées par une compréhension :

```
let list = [ 0..99 ]
```

F#

<code>list ▷ List.map f</code>	<code>↔</code>	<code>[ for x in list do f x ]</code>
<code>list ▷ List.filter p</code>	<code>↔</code>	<code>[ for x in list do if p x then x ]</code>
<code>list ▷ List.filter p ▷ List.map f</code>	<code>↔</code>	<code>[ for x in list do if p x then f x ]</code>
<code>list ▷ List.collect g</code>	<code>↔</code>	<code>[ for x in list do yield! g x ]</code>

# Module **Map** : fonctions spécifiques

## **Map.change** : modification intelligente

Signature : `Map.change key (f: 'T option → 'T option) table`

Selon la fonction **f** passée en argument, on peut :

→ Ajouter, modifier ou supprimer l'élément d'une clé donnée

Entrée	Renvoie <b>None</b>	Renvoie <b>Some newVal</b>
<b>None</b> (Élément absent)	Ignore cette clé	Ajoute l'élément <i>(key, newVal)</i> ≡ <code>Map.add key newVal table</code>
<b>Some value</b> (Élément existant)	Supprime la clé ≡ <code>Map.remove key table</code>	Passe la valeur à <i>newVal</i> ≡ <code>Map.add key newVal table</code>

# Map : containsKey VS exists VS filter

Fonction	Signature	Commentaire
containsKey	'K → Map<'K,'V> → bool	Indique si la clé est présente
exists	f → Map<'K,'V> → bool	Indique si un couple clé/valeur satisfait le prédicat
filter	f → Map<'K,'V> → Map<'K,'V>	Conserve les couples clé/valeur satisfaisant le prédicat

Avec prédicat f: 'K → 'V → bool

```
let table = Map [ (2, "A"); (1, "B"); (3, "D") ]
```

F#

```
table ▷ Map.containsKey 0 // false
```

```
table ▷ Map.containsKey 2 // true
```

```
let isEven i = i % 2 = 0
```

```
let isFigure (s: string) = "AEIOUY".Contains(s)
```

```
table ▷ Map.exists (fun k v → (isEven k) && (isFigure v)) // true
```

```
table ▷ Map.filter (fun k v → (isEven k) && (isFigure v)) // map [(2, "A")]
```

# Module **String**

`string`  $\equiv$  `Seq<char>`  $\rightarrow$  Module `(FSharp.Core.)String` ( $\neq$  `System.String`)

$\rightarrow$  Propose quelques fonctions similaires à celles de `Seq` en + performantes :

```
String.concat (separator: string) (strings: seq<string>) : string
String.init (count: int) (f: (index: int)  $\rightarrow$  string) : string
String.replicate (count: int) (s: string) : string

String.exists (predicate: char  $\rightarrow$  bool) (s: string) : bool
String.forall (predicate: char  $\rightarrow$  bool) (s: string) : bool
String.filter (predicate: char  $\rightarrow$  bool) (s: string) : string

String.collect (mapping: char  $\rightarrow$  string) (s: string) : string
String.map (mapping: char  $\rightarrow$  char) (s: string) : string
String.mapi (mapping: int  $\rightarrow$  char  $\rightarrow$  char) (s: string) : string
// Idem iter/iteri qui renvoie unit
```

F#

# Module **String** - Examples

```
let a = String.concat "-" ["a"; "b"; "c"] // "a-b-c"
let b = String.init 3 (fun i → $"#{i}") // "#0#1#2"
let c = String.replicate 3 "0" // "000"

let d = "abcd" ▷ String.exists (fun c → c ≥ 'b') // true
let e = "abcd" ▷ String.forall (fun c → c ≥ 'b') // false
let f = "abcd" ▷ String.filter (fun c → c ≥ 'b') // "bcd"

let g = "abcd" ▷ String.collect (fun c → $"#{c}{c}") // "aabbccdd"

let h = "abcd" ▷ String.map (fun c → (int c) + 1 ▷ char) // "bcde"
```

F#

# 4 ■ Quiz



# Question 1

```
type Address = { City: string; Country: string }  
  
let format address = $"{address.City}, {address.Country}"  
  
let addresses: Address list = ...
```

F#

Quelle fonction de **List** utiliser sur **addresses** pour appliquer **format** aux éléments ?

- A. **List.iter()**
- B. **List.map()**
- C. **List.sum()**

🕒 10"





# Réponse 1

A. `List.iter()` ❌

B. `List.map()` ✅

C. `List.sum()` ❌



## Question 2

Que vaut `[1..4] ▷ List.head` ?

A. `[2; 3; 4]`

B. `1`

C. `4`

 10''



# Réponse 2

`[1..4] ▷ List.head =`

A. `[2; 3; 4]` ✗

(Ne pas confondre avec `List.tail`)

B. `1` ✓

C. `4` ✗

(Ne pas confondre avec `List.last`)



## Question 3

Quelle est la bonne manière d'obtenir la moyenne d'une liste ?

- A. `[2; 4] ▷ List.average`
- B. `[2; 4] ▷ List.avg`
- C. `[2.0; 4.0] ▷ List.average`

 10''



# Réponse 3

**Bonne manière d'obtenir la moyenne d'une liste :**

A. `[2; 4] ▷ List.average` ❌

💣 Error FS0001: Le type `int` ne prend pas en charge l'opérateur `DivideByInt`

B. `[2; 4] ▷ List.avg`

💣 Error FS0039: La valeur [...] `avg` n'est pas définie.

C. `[2.0; 4.0] ▷ List.average` ✅

```
val it : float = 3.0
```



# 5. ■ Le Récap'



# Types

5 collections dont 4 fonctionnelles/immuables

- **List** : choix par défaut
  - *Passe-partout*
  - *Pratique* : pattern matching, opérateurs *Cons* **::** et *Append* **@**...
- **Array** : mutabilité / performance
- **Seq** : évaluation différée (*Lazy*), séquence infinie
- **Set** : unicité des éléments
- **Map** : classement des éléments par clé

# API

**Riche** → Centaine de fonctions >> Cinquantaine pour LINQ

**Homogène** → Syntaxe et fonctions communes

**Sémantique** → Nom des fonctions proche du JS





# API - Comparatif C# / F# / JS

C# LINQ	F#	JS Array
Select(), SelectMany()	map, collect	map(), flatMap()
Any(predicate), All()	exists, forall	some(), every()
Where(), ×	filter, choose	filter(), ×
First(), FirstOrDefault()	find, tryFind	×, find()
×	pick, tryPick	×
Aggregate([seed])	fold, reduce	reduce()
Average(), Sum()	average, sum	×
ToList(), AsEnumerable()	List.ofSeq, toSeq	×
Zip()	zip	×

# Exercices

Sur [exercism.io](https://exercism.io) (se créer un compte)

Exercice	Niveau	Sujets
High Scores	Facile	List
Protein Translation	Moyen+	Seq / List 💡
ETL	Moyen	Map de List, Tuple
Grade School	Moyen+	Map de List

## 💡 Astuces :

- string est une Seq<char>
- Quid de Seq.chunkBySize ?

# Ressources complémentaires

*Toutes les fonctions, avec leur coût en  $O(?)$*

<https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/fsharp-collection-types#table-of-functions>

*Choosing between collection functions (2015)*

<https://fsharpforfunandprofit.com/posts/list-module-functions/>

*An F# Primer for curious C# developers - Work with collections (2020)*

<https://laenas.github.io/posts/01-fs-primer.html#work-with-collections>

*Formatage des collections*

<https://docs.microsoft.com/en-us/dotnet/fsharp/style-guide/formatting#formatting-lists-and-arrays>

Merci 🙏

**SOAT**

→ Digitalize society



**SOAT.FR**