

# Concepts principaux

## Valeurs Booléennes

Les valeurs Booléennes représentent les états “vrai” et “faux”, et sont particulièrement importantes pour nous: elles vont intervenir dans la majorité des sences. De manière générale, beaucoup de décisions que nous allons prendre seront *in fine* des questions dont la réponse est “oui” ou “non”, et les valeurs Booléennes sont appropriés dans ce contexte.

La première valeur est “vrai”:

true

true

et la seconde est “faux”:

false

false

Nous pouvons combiner ces valeurs via différentes opérations.

## Opérations sur les valeurs Booléennes

Les valeurs Booléennes ont leurs propres opérations. Ces opérations ont la propriété de prendre comme entrée une où plusieurs valeurs Booléennes, et de retourner une réponse Booléenne.

La première est le or, qui renvoie vrai ssi au moins une de ses entrées est vrai. Elle est représentée par la barre verticale:

true | true

true

true | false

true

false | false

false

La seconde opération importante est and, qui renvoie vrai ssi ses deux entrées sont vraies. Elle est représentée par le signe &:

true & true

true

true & false

false

false & false

false

On peut aussi prendre la *négation* d’une valeur Booléenne avec l’opérateur not, qui est en général représenté par !, mais parfois aussi par ~:

!true

false

!false

true

Le dernier opérateur Booléen est le xor (“ou exclusif”), qui renvoie vrai uniquement ssi l’opération or appliquée à des deux entrées renvoie vrai *et* que l’opération and renvoie faux. Il est représenté par le signe `^`, qui s’écrit `\xor<Tab>`

```
true ^ false
true
false ^ false
false
true ^ true
false
```

Ces opérateurs peuvent être utilisés pour prendre des décisions complexes. Par exemple, `^` est défini, pour deux entrées  $x_1$  et  $x_2$ , comme  $(x_1 \mid x_2) \ \& \ (! (x_1 \ \& \ x_2))$ .

## Vecteurs et matrices

Une des tâches les plus courantes que nous devons réaliser est de stocker de l’information dans des structures avec plusieurs dimensions. Autant que possible, nous essaierons de connaître les dimensions de ces objets avant de les créer.

Un objet à une seule dimension est un vecteur, et on peut en créer un avec la commande

```
zeros(5)
5-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
```

qui se lit “un vecteur de cinq positions initialement rempli de zéros”. Par défaut, ce vecteur pourra stocker des *nombres* (nous reviendrons sur la définition d’un nombre plus tard), mais on peut créer un vecteur qui contient des valeurs Booléennes:

```
zeros(Bool, 3)
3-element Vector{Bool}:
 0
 0
 0
```

**NB:** Regardez la documentation des fonctions `rand` et `ones`.

On peut aussi créer des objets avec plus d’une dimension, comme des matrices (deux dimensions), des tenseurs (trois dimensions), etc.. Par exemple, cette commande crée une matrice initialement rempli de valeurs Booléennes aléatoires, avec 3 lignes et 2 colonnes:

```
rand(Bool, 3, 2)
3×2 Matrix{Bool}:
 1  0
 1  0
 0  0
```

Au cours de la session, nous allons identifier des façons différentes de naviguer dans ces objets. Pour le moment, nous allons nous contenter de trouver et de modifier le contenu de ces objets en utilisant les coordonnées.

Pour un vecteur, on peut extraire l'information en utilisant le numéro de la position. Notez que les vecteurs dans Julia sont des colonnes (pour faciliter les opérations d'algèbre linéaire):

```
V = zeros{Bool, 3}
V[1]
false
```

On peut modifier la deuxième position de ce vecteur:

```
V[2] = true
V
3-element Vector{Bool}:
 0
 1
 0
```

Il existe aussi les raccourcis `begin` (premier élément) et `end` (dernier élément):

```
V[begin] = true
V
3-element Vector{Bool}:
 1
 1
 0
```

Pour une matrice, l'indexation se fait exactement de la même manière, mais on utilise les coordonnées sous la forme ligne, colonne:

```
M = zeros{Bool, 2, 3}
M[1, 2] = true
M[2, 3] = true
M
2×3 Matrix{Bool}:
 0  1  0
 0  0  1
```

## Installer des packages

### Les projets

```
import Pkg
Pkg.activate(".")
```

### Installer un package

```
Pkg.add("CairoMakie")
```

### Charger un package

```
using CairoMakie
```

## Simulation: *Conus textile*

### Définir les paramètres de la simulation

```
n_cellules = 191
191
```

Définir le nombre de générations comme  $(n\_cellules - 1) / 2$  et convertir en entier

```
n_generations = Int((n_cellules - 1) / 2)
```

```
95
```

```
TODO
```

```
shell = zeros(Bool, n_cellules, n_generations);
```

Trouver l'index de la cellule au milieu et mettre sa valeur à true

```
milieu_index = div(n_cellules, 2) + 1
```

```
shell[milieu_index, 1] = true
```

```
true
```

## Effectuer la simulation

```
for generation in 2:n_generations
```

```
    for cellule in 2:n_cellules-1
```

```
        p = shell[cellule-1, generation-1]
```

```
        q = shell[cellule, generation-1]
```

```
        r = shell[cellule+1, generation-1]
```

```
        # Règle de transition pour la Rule 30 des automates cellulaires : p xor (q or  
r)
```

```
        shell[cellule, generation] = p ⊕ (q || r)
```

```
    end
```

```
end
```

## Afficher l'état final de la simulation

```
heatmap(
```

```
    shell,
```

```
    colormap=[:white, :black],
```

```
    axis=(; aspect=DataAspect()),
```

```
    figure=(; size=(3n_cellules, 3n_generations), figure_padding=0)
```

```
)
```

```
hidespines!(current_axis())
```

```
hidedecorations!(current_axis())
```

```
current_figure()
```

