

# Une introduction à R

Christophe Ambroise

25/05/2018



# Le langage R de base (R base)

# Qu'est-ce que R ?

## En bref

R est un logiciel de développement scientifique spécialisé dans le calcul et l'**analyse statistique**

## Mais aussi

- un langage interprété,
- un environnement de développement,
- un projet open source (projet **GNU**),
- un logiciel multi-plateforme (Linux, Mac, Windows),
- la 18<sup>ième</sup> lettre de l'alphabet

# Principales fonctionnalités

- 1 Gestionnaire de données
  - Lecture, manipulation, stockage.
- 2 Algèbre linéaire
  - Opérations classiques sur vecteurs, tableaux et matrices
- 3 Statistiques et analyse de données
  - Dispose d'un *grand* nombre de méthodes d'analyse de données (des plus anciennes et aux plus récentes)
- 4 Moteur de sorties graphiques
  - Sorties écran ou fichier
- 5 Système de modules
  - Alimenté par la communauté (+ de 2000 extensions!)
- 6 Interface facile avec C/C++, Fortran

## Chronologie

**1970s** développement de S au Bell labs.

**1980s** développement de  $S^+$  au AT&T. Lab

**1993** développement de R sur le modèle de par Robert Gentleman et Ross Ihaka au département de statistique de l'université d'Auckland.

**1995** dépôts des codes sources sous licence GNU/GPL

**1997** élargissement du groupe

**2002** la fondation dépose ses statuts sous la présidence de Gentleman et Ihaka

**2010** Les débuts de Rstudio

- 1 La page web de la **fondation**
  - les statuts, des liens, des références.
  - <http://www.r-project.org/>
- 2 La page web du **CRAN** (Comprehensive R Arxiv Network)
  - binaires d'installation, packages, documentations, ...
  - <http://cran.r-project.org/>
- 3 La **conférence** des utilisateurs de R:
  - annuelle, prochaine édition à Toulouse
  - <http://www.user2019.fr/>
- 4 **The R journal** propose des articles sur
  - de nouvelles extensions, des applications, des actualités.
  - <http://journal.r-project.org/>

# Qualités et défauts de R

## Plus

- Libre et gratuit,
- Richesse des modules (en statistique),
- Rapidité d'exécution,
- Développement rapide (langage de scripts),
- Syntaxe intuitive et compact,
- Nombreuses possibilités graphiques.

## Moins

- Aide intégrée succincte,
- Debugger un peu sec,
- Code parfois illisible (compacité),
- Personnalisation des graphiques un peu lourde.

# Les concurrents plus ou moins directs

Les logiciels de développement scientifique sont spécialisés en

## 1 algèbre linéaire

- Matlab de Mathworks, une référence,
- Scilab de l'INRIA, l'alternative libre de matlab,
- Octave de GNU, l'alternative open source ,

## 2 statistiques

- SAS (SAS Inc.), la référence,
- S-PLUS (TIBCO), le concurrent,

## 3 calcul symbolique

- Mathematica (Wolfram), la référence,
- Mapple (Maplesoft), la référence aussi,
- Maxima (GNU), l'alternative open source .

## 4 Autres

- Julia
- Python, le concurrent le plus sérieux ?,



- Anciens ouvrages de référence
- R base, un livre de base
- Nouvelle mode: Wickam et co

# R Studio comme interface

# Les fonctionnalités principales

- Il fournit un éditeur intégré,
- fonctionne sur toutes les plates-formes (y compris sur des serveurs) et
- offre de nombreux avantages tels que l'intégration avec la version contrôle et gestion de projet.

Lorsque vous ouvrez RStudio pour la première fois, vous serez accueilli par trois panneaux:

- La console interactive R (entier gauche)
- Environnement / Histoire (onglet en haut à droite)
- Files / Plots / Packages / Help / Viewer (onglet en bas à droite)

Une fois que vous ouvrez des fichiers, tels que des scripts R, un panneau d'éditeur s'ouvre également en haut à gauche.

Il existe deux manières principales d'interagir avec R:

- en utilisant la console
- ou en utilisant des fichiers de script (fichiers texte contenant votre code).

# La fenêtre de la console (dans RStudio, le panneau en bas à gauche)

- endroit où R vous attend pour lui dire quoi faire et où il montrera les résultats d'une commande
- Vous pouvez taper des commandes directement dans la console,
- mais elles seront oubliées lorsque vous fermerez la session.

- Il est préférable d'entrer les commandes dans l'éditeur de script
- et de sauvegarder le script
- envoyer la ligne actuelle ou le texte sélectionné à la console R à l'aide du raccourci Ctrl-Entrée

## Raccourcis console - éditeur

Ctrl-1 et Ctrl-2 qui vous permettent de sauter entre le script et les fenêtres de la console.

- Utilisez # signes pour commenter.
- Commentez libéralement dans vos scripts R.
- Tout ce qui se trouve à droite d'un # est ignoré par R.



# Astuce: Exécution de segments de votre code

- RStudio vous offre une grande souplesse dans l'exécution du code depuis l'éditeur fenêtre. Il existe des boutons, des choix de menus et des raccourcis clavier. Pour exécuter la ligne en cours, vous pouvez
  - ➊ cliquez sur le bouton `Run` situé au-dessus du panneau de l'éditeur,
  - ➋ sélectionnez "Run Lines" dans le menu "Code", ou
  - ➌ appuyez sur `Ctrl-Entrée` dans Windows ou Linux ou sur `Commande-Entrée` sur OS X. (Ce raccourci peut également être vu en survolant la souris sur le bouton).

- Pour lancer un bloc de code, sélectionnez-le, puis Exécuter.
- Si vous avez modifié une ligne de code dans un bloc de code que vous venez d'exécuter, il n'est pas nécessaire de resélectionner la section et Run, vous pouvez utiliser le bouton suivant,
- Ré-exécuter la région précédente Cela exécutera le bloc de code précédent inculquer les modifications que vous avez apportées.

# Utiliser R comme calculatrice

La chose la plus simple que vous puissiez faire avec R est l'arithmétique:

```
1 + 100
```

```
## [1] 101
```

Et R imprimera la réponse, avec un précédent “[1]”. Ne t'inquiète pas pour ça pour l'instant, nous l'expliquerons plus tard. Pour l'instant, considérez-le comme indiquant une sortie.



# Astuce: Annulation des commandes

- Si vous utilisez R depuis la ligne de commande plutôt que depuis RStudio,
- vous devez utiliser `Ctrl + C` au lieu de `Esc` pour annuler la commande. Ce s'applique également aux utilisateurs de Mac!
- Annuler une commande n'est pas seulement utile pour tuer des commandes incomplètes:
- vous pouvez aussi l'utiliser pour dire à R d'arrêter d'exécuter du code (par exemple si
- en prenant beaucoup plus de temps que prévu, ou pour se débarrasser du code que vous êtes en train d'écrire.

# Ordre des opérations I

Lorsque vous utilisez R comme calculatrice, l'ordre des opérations est le même que vous auriez appris à l'école.

De la plus haute à la plus basse préséance:

- Parenthèses: (, )
- Exposants: ^ ou \*\*
- Diviser: /
- Multiplier: \*
- Ajouter: +
- Soustraire: -

```
3 + 5 * 2
```

```
## [1] 13
```

# Ordre des opérations II

Utilisez des parenthèses pour regrouper les opérations afin de forcer l'ordre de évaluation si elle diffère du défaut, ou pour préciser ce que vous avoir l'intention

```
(3 + 5) * 2
```

```
## [1] 16
```

Cela peut devenir compliqué lorsque cela n'est pas nécessaire, mais clarifie vos intentions. Rappelez-vous que d'autres peuvent lire votre code ultérieurement.

```
(3 + (5 * (2 ^ 2))) # difficile à lire  
3 + 5 * 2 ^ 2 # si vous vous souvenez des règles  
3 + 5 * (2 ^ 2) # Si vous oubliez certaines règles, cela pourrait vous aid
```

## Ordre des opérations III

Le texte après chaque ligne de code s'appelle un "commentaire". Tout ce qui suit le symbole de hachage (ou octothorpe) "#" est ignoré par R lorsqu'il exécute du code.

Les nombres vraiment petits ou grands ont une notation scientifique:

```
2/10000
```

```
## [1] 2e-04
```

Ce qui est un raccourci pour "multiplié par  $10^X$ ". Donc  $2e-4$  est un raccourci pour  $2 * 10^{-4}$ .

Vous pouvez aussi écrire des nombres en notation scientifique:

```
5e3 # Notez l'absence de moins ici
```

```
## [1] 5000
```



# Fonctions mathématiques

R a beaucoup de fonctions mathématiques intégrées. Pour appeler une fonction, nous tapons simplement son nom, suivi par des parenthèses ouvertes et fermantes. Tout ce que nous tapons à l'intérieur des parenthèses s'appelle la fonction arguments:

```
sin (1) # fonctions trigonométriques
```

```
## [1] 0.841471
```

```
log (1) # logarithme naturel
```

```
## [1] 0
```

```
log10 (10) # base-10 logarithme
```

```
## [1] 1
```

```
exp (0.5) #  $e^{(1/2)}$ 
```

Ne vous souciez pas d'essayer de vous souvenir de toutes les fonctions de R:

- rechercher sur Google,
- ou si vous vous souvenez de la début du nom de la fonction, utilisez *complétion* dans RStudio.

C'est un avantage que RStudio sur R , il dispose de *capacités d'auto-complétion* qui vous permettent plus facilement rechercher des fonctions, leurs arguments et les valeurs qu'ils prendre.

Taper un ? Avant le nom d'une commande ouvrira la page d'aide pour cette commande. En plus de fournir

- une description la page d'aide affichera généralement
- une collection d'exemples

# Comparer les choses

Nous pouvons également faire la comparaison en R:

```
1 == 1 # égalité (noter deux signes égaux, lire comme "est égal à")
```

```
## [1] TRUE
```

```
1 != 2 # inégalité (lire comme "n'est pas égal à")
```

```
## [1] TRUE
```

```
1 < 2 # moins que
```

```
## [1] TRUE
```

```
1 <= 1 # inférieur ou égal à
```

```
## [1] TRUE
```

```
1 > 0 # plus grand que
```

# Astuce: Comparer les nombres

- Un mot d'avertissement sur la comparaison des chiffres: vous devriez ne jamais utiliser `==` pour comparer deux nombres à moins qu'ils ne soient entiers (un type de données pouvant représenter spécifiquement uniquement des nombres entiers).
- Les ordinateurs ne peuvent représenter que des nombres décimaux avec un certain degré de précision, donc deux nombres qui semblent le même lorsqu'il est imprimé par R, peut effectivement avoir différentes représentations sous-jacentes et donc être différent par une petite marge d'erreur (appelée Machine tolérance numérique).
- Au lieu de cela, vous devez utiliser la fonction `all.equal`.
- Lectures complémentaires: <http://floating-point-gui.de/>

# Variables et affectation I

Stocker des valeurs dans des variables en utilisant l'opérateur d'affectation  
<-

```
x <- 1/40
```

```
x
```

```
## [1] 0.025
```

Plus précisément, la valeur stockée est une approximation \* décimale \* de cette fraction appelée [nombre à virgule flottante] ([http://en.wikipedia.org/wiki/Floating\\_point](http://en.wikipedia.org/wiki/Floating_point)).

Recherchez l'onglet `Environment` dans l'un des volets de RStudio, et vous verrez que `x` et sa valeur est apparu. Notre variable `x` peut être utilisée à la place d'un nombre dans tout calcul qui attend un nombre:

# Variables et affectation II

```
log (x)
```

```
## [1] -3.688879
```

Notez également que les variables peuvent être réaffectées:

```
x <- 100
```

x contenait la valeur 0.025 et a maintenant la valeur 100.

Les valeurs d'affectation peuvent contenir la variable affectée à:

```
x <- x + 1 #notice comment RStudio met à jour sa description de x en haut
```

Le côté droit de l'affectation peut être toute expression R valide. Le côté droit est *entièrement évalué* avant que l'affectation ait lieu.

# Noms de variables

Les noms de variables peuvent contenir des lettres, des chiffres, des traits de soulignement et des points. Ils ne peuvent pas commencer avec un nombre ni contenir des espaces du tout. Différentes personnes utilisent différentes conventions pour les noms de variables longues, celles-ci comprennent

- `periods.between.words`
- `souligne_entre_mots`
- `camelCaseToSeparateWords`

Ce que vous utilisez dépend de vous, mais **soyez cohérent**.

# Affectation (suite)

Il est également possible d'utiliser l'opérateur = pour l'affectation:

```
x = 1/40
```

Mais c'est beaucoup moins courant parmi les utilisateurs de R.

Donc, la recommandation est d'utiliser <-.



Il existe quelques commandes utiles que vous pouvez utiliser pour interagir avec la session R.

`ls` listera toutes les variables et fonctions stockées dans l'environnement global (votre session de travail):

```
ls()
```

```
## [1] "def.chunk.hook" "x"
```

# Astuce: objets cachés

- Comme dans le shell, `ls` cachera toutes les variables ou fonctions commençant avec un `"."` par défaut.
- Pour lister tous les objets, tapez `ls (all.names = TRUE)`

## Remarque

Notez ici que nous n'avons donné aucun argument à `ls`, mais nous avons quand même nécessaire de donner aux parenthèses de dire à R d'appeler la fonction.

# Contenu d'une fonction

Si vous tapez `ls` par lui-même, R imprimera le code source de cette fonction!

```
ls
```

```
## function (name, pos = -1L, envir = as.environment(pos), all.names = FALSE,
##   pattern, sorted = TRUE)
## {
##   if (!missing(name)) {
##     pos <- tryCatch(name, error = function(e) e)
##     if (inherits(pos, "error")) {
##       name <- substitute(name)
##       if (!is.character(name))
##         name <- deparse(name)
##       warning(gettextf("%s converted to character string",
##         sQuote(name)), domain = NA)
##       pos <- name
##     }
##   }
##   all.names <- .Internal(ls(envir, all.names, sorted))
##   if (!missing(pattern)) {
```

# Suppression

Vous pouvez utiliser `rm` pour supprimer des objets dont vous n'avez plus besoin:

```
rm (x)
```

Si vous avez beaucoup de choses dans votre environnement et souhaitez les supprimer toutes, vous pouvez transmettre les résultats de `ls` à la fonction `rm`:

```
rm(list=ls())
```

Dans ce cas, nous avons spécifié que les résultats de `ls` devraient être utilisés pour le L'argument `list` dans `rm`. Lorsque vous attribuez des valeurs aux arguments par nom, vous *devez* utiliser l'opérateur `=` !!

Si au lieu de cela nous utilisons `<-`, il y aura des effets secondaires inattendus, ou vous pourriez avoir un message d'erreur:

```
rm(list <- ls ())
```

# Astuce: Avertissements vs erreurs

- Quand R fait quelque chose d'inattendu! Les erreurs, comme ci-dessus, sont émises lorsque R ne peut pas procéder à un calcul.
- En revanche, les avertissements signifient généralement que la fonction a été exécutée, mais cela n'a probablement pas fonctionné comme prévu.
- Dans les deux cas, le message que R imprime vous donne généralement des indices sur la manière de résoudre un problème.

# R Packages

Il est possible d'ajouter des fonctions à R en écrivant un paquet, ou par obtenir un paquet écrit par quelqu'un d'autre. Au moment de l'écriture, il y a sont plus de 7 000 paquets disponibles sur CRAN (l'archive complète de R réseau). R et RStudio ont des fonctionnalités pour gérer les paquets:

- Vous pouvez voir quels paquets sont installés en tapant

```
installed.packages ()
```

- Vous pouvez installer des paquets en tapant

```
install.packages (" packagename "), où packagename est le nom  
du package, entre guillemets.
```

- Vous pouvez mettre à jour les paquets installés en tapant  

```
update.packages ()
```
- Vous pouvez supprimer un paquet avec 

```
remove.packages (" packagename ")
```
- Vous pouvez rendre un paquet disponible pour être utilisé avec

- 1 Nommer toutes les panels de R studio
- 2 Réaliser une addition et une multiplication dans la console
- 3 Sauver un script R avec votre code d'addition

**Chercher de l'aide**



R, et chaque paquet, fournissent des fichiers d'aide pour les fonctions. Pour chercher de l'aide sur une fonction spécifique qui est dans un package chargé

```
?nom_fonction  
help(nom_fonction)
```

Cela chargera une page d'aide dans RStudio (ou du texte brut dans R seul).

# Opérateurs spéciaux

Pour demander de l'aide sur des opérateurs spéciaux, utilisez des guillemets:

```
?"+"
```

# Obtenir de l'aide sur les packages

De nombreux paquets contiennent des “vignettes”: des tutoriels et des exemples de documentation. Sans aucun argument, `vignette()` listera toutes les vignettes pour tous les paquets installés;

```
vignette (package =" nom-du-paquet ")
```

listera toutes les vignettes disponibles pour

```
package-name et vignette ("vignette-name")
```

ouvriront la vignette spécifiée.

Si un paquet ne contient aucune vignette, vous pouvez généralement trouver de l'aide en tapant

```
help("nom-du-paquet").
```

# Quand vous vous souvenez de la fonction

Si vous ne savez pas exactement dans quel paquet est une fonction ou comment elle est spécifiquement orthographiée, vous pouvez effectuer une recherche floue:

```
??nom_fonction
```

# Mélanger code et texte avec knitr

## Principe de Claerbout (Géophysicien, Stanford)

*An article about computational science in a scientific publication is not the scholarship itself, it is merely > advertising of the scholarship. The actual scholarship is the complete >software development environment and the complete set of instructions which generated the figures.*

## `knitr`

pour générer des rapports qui combinent le texte, le code et les résultats.

## Markdown

pour mettre en forme le texte

*Markdown est un langage de balisage léger créé par John Gruber en 2004. Son but est d'offrir une syntaxe facile à lire et à écrire. Un document balisé par Markdown peut être lu en l'état sans donner l'impression d'avoir été balisé ou formaté par des instructions particulières. — Wikipedia*

Lien Wikipedia

## Code Chunks

code dans des blocs délimités par des guillemets triples suivis de `{r}`.

Rmarkdown est un univers extensible, si vous voulez continuer, lisez

<https://rmarkdown.rstudio.com/>



# Installer knitr

## Dans la console

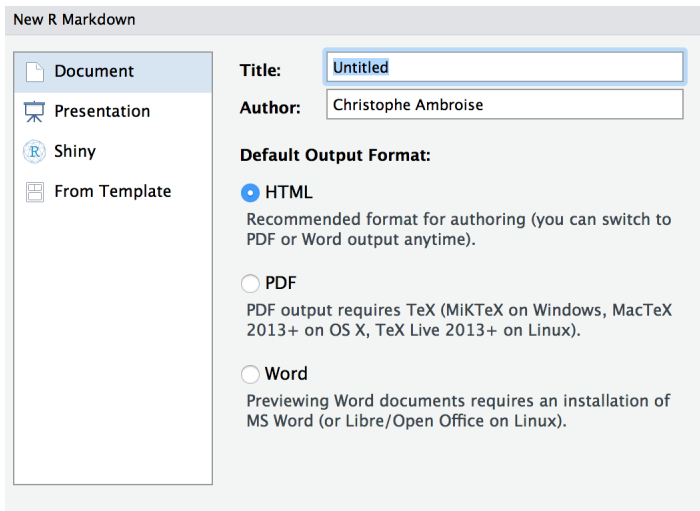
```
install.packages('knitr')
```

## Par le menu

Tools -> Install Packages

# Créer un de type fichier Rmarkdown (.Rmd)

Dans R Studio, cliquez sur Fichier → Nouveau fichier → R Markdown et vous obtiendrez une boîte de dialogue du type



New R Markdown

Document

Presentation

Shiny

From Template

**Title:**

**Author:**

**Default Output Format:**

**HTML**  
Recommended format for authoring (you can switch to PDF or Word output anytime).

**PDF**  
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

**Word**  
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

Vous créez un fichier avec une entête dite yaml du type

```
---  
title: "Initial R Markdown document"  
author: "Karl Broman"  
date: "April 23, 2015"  
output: html_document  
---
```

qui précise comment peut être transformé le fichier

# Markdown I

Markdown est un langage à balise

- écrivez **en gras** en utilisant deux astérisques, comme ceci: `**gras**`,
- écrivez en *italics* en utilisant des traits de soulignement, comme ceci: `_italics_`.

Vous pouvez créer une liste à puces en écrivant une liste avec des tirets ou astérisques, comme ceci:

- \* gras avec double astérisque
- \* italiques avec des soulignés
- \* police de type code avec backticks

ou comme ça:

# Markdown II

- gras avec double astérisque
- italiques avec des soulignés
- police de type code avec backticks

Vous pouvez créer une liste numérotée en utilisant simplement des chiffres. Vous pouvez utiliser le même nombre encore et encore si vous voulez:

1. gras avec double astérisque
1. italiques avec des soulignés
1. police de type code avec backticks

Cela apparaîtra comme:

- ① gras avec double astérisque
- ② italiques avec des soulignés
- ③ police de type code avec backticks

Vous pouvez créer des en-têtes de section de différentes tailles en initiant une ligne avec un certain nombre de symboles #:

```
# Titre
```

```
## Section principale
```

```
### Sous-section
```

```
#### Sous-sous-section
```

Vous *compilez* le document R Markdown en cliquant sur le “Knit HTML” en haut à gauche.

# Un peu plus de Markdown

- Vous pouvez créer un hyperlien comme celui-ci:

[texte à afficher] (<http://the-web-page.com>).

- Vous pouvez inclure un fichier image comme ceci:

![Caption] (<http://url/for/file>)

Vous pouvez faire des indices (par exemple,  $F_2$ ) avec `F~2~` et des exposants (par exemple,  $F^2$ ) avec `F^2^`.

Si vous savez écrire des équations dans [LaTeX] (<http://www.latex-project.org/>), vous serez heureux de savoir que vous pouvez utiliser `$ $` et `$$ $$` pour insérer des équations mathématiques, comme `$E = mc^2$` et

```
$$ y = \mu + \sum_{i = 1}^p \beta_i x_i + \epsilon $$
```



# Morceaux de code

Markdown est intéressant et utile, mais le plus grand intérêt vient du mélange du texte balisé avec des morceaux de code R.

- Quand traité, le code R sera exécuté; s'ils produisent des valeurs, figures, ceux-ci seront insérés dans le document final.

Les morceaux de code ressemblent à ceci:

```
```{r load_data}  
gapminder <- read.csv("~/Desktop/gapminder.csv")  
```
```

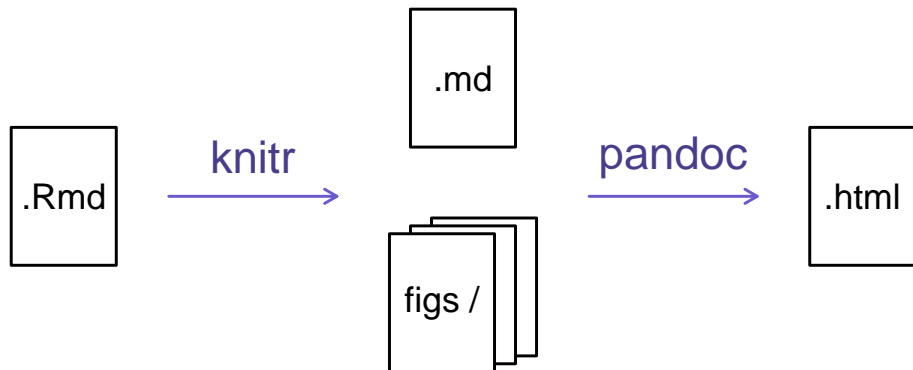
C'est une bonne idée de donner chaque morceau un nom, car ils vous aideront à corriger les erreurs et, si des graphiques sont produit, les noms de fichiers sont basés sur le nom du bloc de code les a produites.

# Comment les choses sont compilées I

Lorsque vous appuyez sur le bouton “Knit HTML”, le document R Markdown est traité par [knitr] (<http://yihui.name/knitr>) et un simple Markdown document est produit (ainsi que, potentiellement, un ensemble de fichiers de figure): le code R est exécuté et remplacé à la fois par l’entrée et la sortie; si les chiffres sont produits, des liens vers ces chiffres sont inclus.

Les documents Markdown et figure sont ensuite traités par l’outil [pandoc] (<http://pandoc.org/>), qui convertit le fichier Markdown en fichier html, avec les chiffres incorporés.

# Comment les choses sont compilées II



Il y a une variété d'options pour affecter la façon dont les morceaux de code sont traités.

- Utilisez `echo = FALSE` pour éviter que le code lui-même ne soit affiché.
- Utilisez `results = "hide"` pour éviter d'imprimer des résultats.
- Utilisez `eval = FALSE` pour que le code soit affiché mais pas évalué.
- Utilisez `warning = FALSE` et `message = FALSE` pour masquer les avertissements ou messages produits.
- Utilisez `fig.height` et `fig.width` pour contrôler la taille des figures produit (en pouces).

- Knitr in a knutshell tutorial
- Dynamic Documents with R and knitr (book)
- R Markdown documentation
- R Markdown cheat sheet

- 1 Créer un fichier Rmarkdown qui produira une sortie html avec un code chunk de votre choix

# Structures de données

## Propriétés

- objet le plus *élémentaire* sous  $\mathbb{R}$ ,
- collection d'entités *de même nature*,
- (ou type) défini par la nature des entités qui le composent.

## Les modes possibles

- 1 numérique
- 2 caractère
- 3 logique



# Quelques vecteurs typés I

## 1 Numérique

```
x0 <- 0
x1 <- c(-1,23,98.7)
mode(x0)
```

```
## [1] "numeric"
```

## 2 Caractère

```
y0 <- "bonjour"
y1 <- c("Pomme", "Flore", "Alexandre")
mode(y1)
```

# Quelques vecteurs typés II

```
## [1] "character"
```

## 3 Logique

```
z0 <- TRUE  
z1 <- c(FALSE, TRUE, FALSE, TRUE, TRUE)  
z2 <- c(T, F, F)  
mode(z2)
```

```
## [1] "logical"
```

# Remarques sur l'affectation I

## Affectation

C'est l'opération qui consiste à *\*attribuer une valeur\** à une

En R, plusieurs choix sont possibles:

**l'opérateur usuel est `<-` (signe inférieur suivi du signe moins)**

```
jo <- "l'indien"  
jo
```

```
## [1] "l'indien"
```

# Remarques sur l'affectation II

**l'opérateur = peut être utilisé la plupart du temps**

```
nb.max.d.annees.pour.faire.une.these = 3  
nb.max.d.annees.pour.faire.une.these
```

```
## [1] 3
```

**la commande `assign` permet cette opération (d'où l'anglicisme *assignment*)**

```
assign("x", c(8,9,-pi,sqrt(2)))
```

# Valeurs spéciales

## Variables réservées par R

- NA est le code R pour les valeurs manquantes (absentes des données),
- NaN est le code de R pour signifier un résultat numérique aberrant ,
- Inf et -Inf sont les valeurs réservées pour plus et moins  $\infty$ ,
- NULL est l'objet nul.

```
c(4,2,NA,5)
```

```
## [1] 4 2 NA 5
```

```
0/0
```

```
## [1] NaN
```

```
1/0
```

```
## [1] Inf
```

# Opérations arithmétiques I

Les opérations sur les vecteurs s'effectuent terme-à-terme

Soient  $x, y$  tels que

```
x<-c(1,2,-3,-4)
```

```
y<-c(-5,-6,9,0)
```

+

addition des éléments de deux vecteurs

```
x+y
```

```
## [1] -4 -4 6 -4
```

# Opérations arithmétiques II

-

soustraction des éléments de deux vecteurs

$x-y$

```
## [1] 6 8 -12 -4
```

\*

multiplication des éléments de deux vecteurs

$x*y$

```
## [1] -5 -12 -27 0
```

# Opérations arithmétiques III

```
\
```

```
division des éléments de deux vecteurs
```

```
x/y
```

```
## [1] -0.2000000 -0.3333333 -0.3333333 -Inf
```



# Recyclage des éléments du vecteur I

Lors d'une opération entre vecteurs, les vecteurs trop courts sont ajustés pour atteindre la taille du plus grand vecteur en recyclant les données.

*Exemple*

```
x <- c(10,100,1000)
y <- c(1,2)
2*x + y - 1
```

```
## [1] 20 201 2000
```

↪ souvent pratique mais **attention aux effets de bords!**

# Opérateurs mathématiques I

## Fonctions numériques élémentaires

floor, ceiling, round

```
floor(2/3)
```

```
## [1] 0
```

```
ceiling(2/3)
```

```
## [1] 1
```

```
round(2/3,3)
```

```
## [1] 0.667
```

# Fonctions arithmétiques élémentaires I

```
` ^{ }, \% \%, \% / \%, abs, log, exp, log10, sqrt, cos, tan, sin...`
```

s'appliquent toutes **terme-à-terme**.

```
log10(c(10, 100, 1000))
```

```
## [1] 1 2 3
```

```
cos(c(pi/2, pi))^2 + sin(c(pi/2, pi))^2
```

```
## [1] 1 1
```

# Fonctions spécifiques à un vecteur I

prod, sum, max, min, range, which.min, which.max, length

```
x <- c(-8,1.5,3)
prod(x)
```

```
## [1] -36
```

```
sum(x)
```

```
## [1] -3.5
```

```
length(x)
```

```
## [1] 3
```

# Fonctions spécifiques à un vecteur II

```
max(x)
```

```
## [1] 3
```

```
min(x)
```

```
## [1] -8
```

```
range(x)
```

```
## [1] -8 3
```

```
which.max(x)
```

```
## [1] 3
```

# Fonctions spécifiques à un vecteur III

```
which.min(x)
```

```
## [1] 1
```

Pour le minimum / maximum terme-à-terme: `pmin`, `pmax`.

# Fonctions appliquées le long du vecteur I

```
`cumsum, cumprod, cummin, cummax`
```

```
x <- c(-2,1,-3,2)  
cumprod(x)
```

```
## [1] -2 -2 6 12
```

```
cumsum(x)
```

```
## [1] -2 -1 -4 -2
```

```
cummax(x)
```

```
## [1] -2 1 1 2
```

# Fonctions appliquées le long du vecteur II

```
cummin(x)
```

```
## [1] -2 -2 -3 -3
```



# Opérateurs ensemblistes

## Fonctionnent pour tous les modes

unique, intersect, union, setdiff, setequal, is.element

```
unique(c("banane", "citron", "banane"))
```

```
## [1] "banane" "citron"
```

```
intersect(c("banane", "citron"), c("orange", "banane"))
```

```
## [1] "banane"
```

```
union(c("banane", "citron"), c("orange", "banane"))
```

```
## [1] "banane" "citron" "orange"
```

```
setequal(c("banane", "citron"), c("orange", "banane"))
```

```
## [1] FALSE
```

# Génération de vecteurs I

Il existe de nombreuses manières de générer des vecteurs de manière plus ou moins automatique

R étant un langage vectoriel savoir générer le vecteur que l'on veut représente un atout important pour programmer en R

# L'opérateur :

```
from:to
```

Génère une séquence par pas de un depuis le nombre `from` jusqu'à `to` (si possible).

```
-5:5
```

```
## [1] -5 -4 -3 -2 -1 0 1 2 3 4 5
```

```
5:-5
```

```
## [1] 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

```
pi:6
```

```
## [1] 3.141593 4.141593 5.141593
```

```
1:6/2
```

```
## [1] 0.5 1.0 1.5 2.0 2.5 3.0
```

## Plusieurs schémas possibles

- `seq(from,to)`
- `seq(from,to,by=)`
- `seq(from,to,length.out=)`

# La commande seq II

## Exemple

```
seq(1,10)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
```

```
seq(2,10,by=2)
```

```
## [1] 2 4 6 8 10
```

```
seq(2,10,length.out=6)
```

```
## [1] 2.0 3.6 5.2 6.8 8.4 10.0
```

# La commande rep

## Fonctionne pour tous les modes

- `rep(x,times)`, où `times` peut être un vecteur,
- `rep(x,each)`.

## Exemple

```
rep(1,3)
```

```
## [1] 1 1 1
```

```
rep("Mercy",3)
```

```
## [1] "Mercy" "Mercy" "Mercy"
```

```
rep(c("A", "B", "C"),c(3,2,4))
```

```
## [1] "A" "A" "A" "B" "B" "C" "C" "C" "C"
```

# Génération de vecteurs logiques

## Obtenus par conditions avec

- les opérateurs logiques '<', '<=', '>', '>=', '==' '!='
- le ET, le OU, NON, OU exclusif: '\&' (intersection), '|' (union), '!' (négation), xor.

```
note1 <- c(8,9,14,3,17.5,11)
note2 <- c("C","B","A","B","E","B")
admis <- (note1 >= 10) & (note2 == "A" | note2 == "B")
mention <- (note1 >= 15) & (note2 == "A")
admis
```

```
## [1] FALSE FALSE TRUE FALSE FALSE TRUE
```

```
sum(admis)
```

```
## [1] 2
```

```
sum(mention)
```

# Par concaténation I

## Avec 'c()'

L'opérateur `'c()'` peut s'appliquer à n'importe quoi pourvu que l'on concatène des vecteurs de même type.

```
c( c(1,2), c(3,4))
```

```
## [1] 1 2 3 4
```

```
round(c(seq(-pi,pi,len=4),rep(c(1:3),each=2),0),2)
```

```
## [1] -3.14 -1.05 1.05 3.14 1.00 1.00 2.00 2.00 3.00 3.00 0.00
```

## remarque

Dans le second exemple, les entiers composants `c(1:3)` ont été forcés au typage flottant.



# Par concaténation II

## Avec paste

Concaténation de chaînes de caractères. Convertit en caractères les éléments passés en argument avant toute opération.

```
paste("R", "c'est", "bien")
```

```
## [1] "R c'est bien"
```

```
paste(2:4, "ieme")
```

```
## [1] "2 ieme" "3 ieme" "4 ieme"
```

```
paste("A", 1:5, sep="")
```

```
## [1] "A1" "A2" "A3" "A4" "A5"
```

```
paste("A", 1:5, sep="", collapse="")
```

## Principe

- Permet la **sélection d'un sous-ensemble** du vecteur  $x$ .
- Le sous-ensemble est spécifié **entre crochets**  $x[\text{subset}]$ .

## L'objet `subset` peut prendre 4 types différents

- 1 **un vecteur logique**, qui doit être de la même taille que le vecteur  $x$ ;
- 2 **un vecteur numérique aux composantes positives**, qui spécifie les valeurs à inclure;
- 3 **un vecteur numérique aux composantes négatives**, qui spécifie les valeurs à exclure;
- 4 **un vecteur de chaînes de caractères**, qui spécifie les noms des éléments de  $x$  à conserver.

# Indexation des vecteurs: exemples I

## Vecteurs logiques

```
x <- c(3,6,-2,9,NA,sin(-pi/6))  
x[x > 0]
```

```
## [1] 3 6 9 NA
```

```
x[!is.na(x)]
```

```
## [1] 3.0 6.0 -2.0 9.0 -0.5
```

```
x[!is.na(x) & x>0]
```

```
## [1] 3 6 9
```

# Indexation des vecteurs: exemples II

```
mean(x,na.rm=TRUE)
```

```
## [1] 3.1
```

```
x[x <= mean(x,na.rm=TRUE)]
```

```
## [1] 3.0 -2.0 NA -0.5
```

## Vecteurs aux composantes positives ou négatives

```
x
```

```
## [1] 3.0 6.0 -2.0 9.0 NA -0.5
```

```
x[2]
```

# Indexation des vecteurs: exemples III

```
## [1] 6
```

```
x[1:5]
```

```
## [1] 3 6 -2 9 NA
```

```
x[-c(1,5)]
```

```
## [1] 6.0 -2.0 9.0 -0.5
```

```
x[-(1:5)]
```

```
## [1] -0.5
```

# Indexation des vecteurs: exemples IV

## Vecteurs de chaînes de caractères

```
names(x) <- c("var1", "var2", "var3", "var4", "var5", "var6")
```

```
x  
  
## var1 var2 var3 var4 var5 var6  
## 3.0 6.0 -2.0 9.0 NA -0.5
```

```
x[c("var1", "var3")]
```

```
## var1 var3  
## 3 -2
```

## 1 Classer

- `sort` renvoie le vecteur classé par ordre croissant ou décroissant,
- `order` renvoie les indices d'ordre des éléments par ordre croissant ou décroissant,

## 2 Extraire

- `which` renvoie les indices de `x` vérifiant une condition;

## 3 Échantillonner

- `sample` échantillonne aléatoirement dans un vecteur `x`, avec ou sans remise.

# Exemples I

```
x <- -5:5  
y <- sample(x)  
sort(y)
```

```
## [1] -5 -4 -3 -2 -1 0 1 2 3 4 5
```

```
order(y)
```

```
## [1] 4 11 9 2 8 5 7 3 10 1 6
```

```
y[order(y)]
```

```
## [1] -5 -4 -3 -2 -1 0 1 2 3 4 5
```

```
y[order(y,decreasing=TRUE)]
```



# Exemples II

```
## [1] 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

```
which(sample(x,4) > 0)
```

```
## [1] 2 4
```

## Facteur

Un **facteur** est un vecteur de **variables catégorielles** .  
Les **niveaux** du facteur peuvent être ordonnés ou pas.

## Utilisation les facteurs s'utilisent pour

**catégoriser les données** d'un vecteur (ce qui s'avère très utile pour la gestion des variables qualitatives).

↪ un facteur est souvent associé à d'autres vecteurs pour en définir une **partition**.

# Création, manipulation

## Création: la fonction factor

```
factor(sample(1:3,10,replace=TRUE))
```

```
## [1] 2 1 1 1 1 3 1 2 2 1  
## Levels: 1 2 3
```

```
factor(sample(1:3,10,replace=TRUE),levels=1:5)
```

```
## [1] 3 2 1 1 1 1 2 2 1 1  
## Levels: 1 2 3 4 5
```

## Gestion: nlevels,levels,table

```
x <- factor(sample(c("thésard", "CR", "MdC"),15,replace=TRUE))  
cat(nlevels(x), "niveaux:", levels(x))
```

```
## 3 niveaux: CR MdC thésard
```

# Un exemple de facteur associé à un vecteur

Un exemple de facteur associé à un vecteur

## Données

Chacun me donne son âge et son grade<sup>^</sup>[sauf un qui refuse : '( ]

```
age <- c(25,35,32,27,32,40,26,25,26,28,30,NA,36,30,30)
```

```
grd <- c("thd", "CR", "MdC", "thd", "thd", "MdC", "MdC", "thd", "thd", "MdC", "CR", "M
```

## Question : nombre d'individus par catégorie ?

```
table(grd)
```

```
## grd
##  CR MdC thd
##   3  5  7
```

# La fonction `tapply`

Un autre point fort de R

## Utilisation

Applique une fonction sur un vecteur partitionné en groupes.

## Question : âge moyen / écart-type par catégorie ?

```
tapply(age,grd,mean,na.rm=TRUE)
```

```
##          CR          MdC          thd  
## 33.66667 31.50000 27.85714
```

```
tapply(age,grd,sd,na.rm=TRUE)
```

```
##          CR          MdC          thd  
## 3.214550 6.191392 2.794553
```

# Matrices (et tableaux)

Les matrices et tableaux sont la des structure de stockage très courantes

# Tableau: définition I

## objet array

Un tableau est un vecteur muni d'un attribut dimension (``dim``) lui même défini par un vecteur. Il est défini par la commande

```
`array(data,dim,dimnames=)`
```

# Tableau: définition II

## Exemple

```
array(1:8,c(2,2,2))
```

```
## , , 1
```

```
##
```

```
##      [,1] [,2]
```

```
## [1,]    1    3
```

```
## [2,]    2    4
```

```
##
```

```
## , , 2
```

```
##
```

```
##      [,1] [,2]
```

```
## [1,]    5    7
```

```
## [2,]    6    8
```



# Matrice: définition I

## objet `matrix`

Une matrice est un tableau à deux dimensions. Elle est définie par la commande

```
matrix(data, nrow=, ncol=, byrow)`
```

En conséquence

- Un objet `array` à deux dimensions est automatiquement converti en `matrix`
- Un vecteur auquel on ajoute un attribut `dimension` est automatiquement converti en `matrix`

# Matrice: définition II

## Exemple

```
class(array(1:4,c(2,2)))
```

```
## [1] "matrix"
```

```
x <- c(1,2,3,4)  
dim(x) <- c(2,2)  
class(x)
```

```
## [1] "matrix"
```

# Remarques importantes I

- R range les éléments d'une matrice par défaut par **colonne**.

```
matrix(1:6,nrow=2)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
matrix(1:6,nrow=2,byrow=TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

# Remarques importantes II

- Lors de la création d'une matrice, R **recycle** les éléments jusqu'à ce que les contraintes de dimension soient vérifiées.

```
matrix(1:3,nrow=2,ncol=2)
```

```
##      [,1] [,2]  
## [1,]    1    3  
## [2,]    2    1
```

# Matrices: opérateurs élémentaires I

Étant donné qu'une matrice est un vecteur pourvu d'une dimension, on a la proposition suivante:

**\*\*La plupart des opérateurs vectorielles s'appliquent\*\***  
(arithmétiques/mathématiques, ensemblistes, d'indexation).

```
a <- matrix(sample(-4:4,9),3,3)
cat(max(a),sum(a),prod(a))
```

```
## 4 0 0
```

```
which(a > 0)
```

```
## [1] 1 5 7 9
```

# Matrices: opérateurs élémentaires II

```
cumsum(a[a > 0])
```

```
## [1] 1 4 8 10
```

```
order(a)
```

```
## [1] 8 6 3 4 2 1 9 5 7
```

```
round(exp(a),4)
```

```
##      [,1]    [,2]    [,3]  
## [1,] 2.7183 0.3679 54.5982  
## [2,] 1.0000 20.0855 0.0183  
## [3,] 0.1353 0.0498 7.3891
```

## Opérateurs matriciels usuels

- $+$ ,  $/$ ,  $*$ ,  $\wedge\{\}$  sont les opérateurs usuels terme-à-terme,
- $\%*\%$  est le produit matriciel,
- `crossprod()` est le produit scalaire,
- `t()` transpose une matrice,
- `diag()` extrait / spécifie la diagonale.

# Manipulation de matrices II

## Exemples

```
a <- matrix(sample(-4:4,9),3,3)
b <- matrix(sample(a),3,3)
diag(a)
```

```
## [1] -3 -2  4
```

```
diag(a) <- diag(b) <- 1
diag(a)
```

```
## [1] 1 1 1
```

```
a + t(b) %*% b
```

```
##      [,1] [,2] [,3]
## [1,]  15   0   1
## [2,]   1  22 -13
## [3,]  -3 -19  12
```



## Trois fonctions selon l'effet voulu:

- `c()` concatène les éléments de plusieurs matrices en un vecteur,
- `cbind()` empile **horizontalement** plusieurs matrices,
- `rbind()` empile **verticalement** plusieurs matrices.

# Concaténation de matrices II

## Exemples

```
a <- matrix(1,2,3)
b <- matrix(2,2,3)
c(a,b)
```

```
## [1] 1 1 1 1 1 1 2 2 2 2 2 2
```

## `cbind(a,b)`

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    1    1    2    2    2
## [2,]    1    1    1    2    2    2
```

## `rbind(a,b)`

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    2    2    2
```

## Résolution de systèmes linéaires, inversion matricielle

La commande ``solve`` résout

$$Ax = b,$$

```
A <- matrix(c(4,2,8,-3),2,2)
b <- c(2,3)
solve(A,b)
```

```
## [1] 1.0714286 -0.2857143
```

ou inverse une matrice:

```
round(solve(A) %*% A,8)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

# Commandes avancées d'algèbre linéaire

Utile pour l'analyse numérique

R dispose des outils classiques d'algèbre linéaire

- `det`: calcule le **déterminant** d'une matrice;
- `chol`: factorisation de **Cholesky** ( $A = C^t C$ , avec  $A$  symétrique,  $C$  triangulaire supérieure);
- `qr`: factorisation **QR** ( $A = QR$  avec  $Q$  orthogonale,  $R$  triangulaire supérieure);
- `eigen`: calcule valeurs propres et **vecteurs propres** d'une matrice;
- `svd`: calcule la décomposition en **valeurs singulières**.
- ...

# Liste: définition

## objet list

Une liste est une **collection d'objets hétérogènes**. Elle est définie par la commande ``list(el1=, el2=, ...)``. Les éléments d'une liste peuvent posséder un nom.

```
list(c(1,2,3),c("robert","johnson"),matrix(rnorm(4),2,2))
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "robert" "johnson"
##
## [[3]]
##           [,1]      [,2]
## [1,]  0.5599110 0.9033587
## [2,] -0.1613184 1.7392404
```

```
list(numero = c(1,2,3), noms = c("robert","johnson"), mat = matrix(rnorm(4),2,2))
```

# Accéder aux éléments

## Deux situations

- Les éléments de la liste **ne sont pas nommés**: on accède au ieme élément par indexation `nom_liste[[i]]` uniquement.
- Les éléments de la liste **sont nommés**: on peut y accéder comme ci-dessus ou en utilisant le nom de l'élément `nom_liste$nom_elt`.

```
maliste <- list(numero = c(1,2,3), noms = c("robert","johnson"), mat = matrix(1:6,2,3))
maliste$nom
```

```
## [1] "robert" "johnson"
```

```
maliste$nom[2]
```

```
## [1] "johnson"
```

```
maliste[[2]]
```

```
## [1] "robert" "johnson"
```

# Tableau de données: définition

Un autre point fort de R

## objet `data.frame`

C'est une liste à laquelle on impose certaines contraintes [que je vous épargne], afin de rassembler vecteurs et facteurs sous la forme d'un tableau de données.

- Pratiquement, **un tableau de données est une matrice dont les colonnes sont de mode différent**,
- C'est l'objet idéal pour la **manipulation de données** (forcez-vous à l'utiliser).

## Syntaxe

On peut spécifier le nom des colonnes par le vecteur ``row.names`` ou directement comme pour une liste:

```
`data.frame(e1=,e2=, ...,row.names)`
```



# Création de tableau de données II

## Exemple

```
age <- c(25,35,32,27,32,40,26,25,26,28,30,NA,36,30,30)
grd <- c("thd","CR","MdC","thd","thd","MdC","MdC","thd","thd","MdC","CR","M
sex <- factor(sample(c(rep("M",3),rep("F",12))))
donnees <- data.frame(age=age,grade=grd,sexe=sex)
head(donnees)
```

```
##   age grade sexe
## 1  25   thd   F
## 2  35    CR   F
## 3  32   MdC   F
## 4  27   thd   F
## 5  32   thd   F
## 6  40   MdC   F
```

# Manipulation des éléments du tableau de données

- Comme une liste !
- les commandes `attach()` `detach` placent ôtent les éléments du tableaux de données dans l'itinéraire de recherche.

```
donnees$age
```

```
## [1] 25 35 32 27 32 40 26 25 26 28 30 NA 36 30 30
```

```
attach(donnees, warn.conflicts=FALSE)  
grade
```

```
## [1] thd CR MdC thd thd MdC MdC thd thd MdC CR MdC CR thd thd  
## Levels: CR MdC thd
```

```
detach(donnees)
```

# Travailler avec les tableaux de données I

- beaucoup de fonctions prédéfinies
- penser aux fonctions `tapply` (ou `by`)

```
summary(donnees)
```

```
##           age           grade  sexe
## Min.      :25.00    CR :3    F:12
## 1st Qu.:26.25    MdC:5    M: 3
## Median :30.00    thd:7
## Mean     :30.14
## 3rd Qu.:32.00
## Max.     :40.00
## NA's     :1
```

```
attach(donnees, warn.conflicts=FALSE)
by(age, sexe, mean, na.rm=TRUE)
```

# Travailler avec les tableaux de données II

```
## sexe: F
## [1] 30.18182
## -----
## sexe: M
## [1] 30
```

```
by(age, grade, mean, na.rm=TRUE)
```

```
## grade: CR
## [1] 33.66667
## -----
## grade: MdC
## [1] 31.5
## -----
## grade: thd
## [1] 27.85714
```

```
detach(donnees)
```

Les structures de contrôle sont les blocs d'un programme

# Regrouper les expressions

## Syntaxe

```
{expr_1; expr_2; ...; expr_n }  
ou  
{  
  expr_1 ...  
  expr_n  
}
```

## Remarques sur les groupes

- La dernière valeur du groupe est retournée;
- un groupe d'expressions peut être passé à une fonction, réutilisé dans une expression plus grande, etc.

# Exécution conditionnelle: if,if/else,ifelse

## Syntaxe

```
if (condition) {  
  expr_1  
else {  
  expr_2
```

ou {

```
ifelse(condition, a, b)
```

## Remarques

- condition est une valeur logique: penser à `\&`, `|`, `!`, ...
- le else est optionnel,
- elseif permet d'imbriquer les conditionnements.

# Exécution répétée: boucle for

## Syntaxe

```
for (var in set) { expr(var) }
```

ou

```
for (var in set) expr(var)
```

à fuir pour éviter les effets de bords sournois!

## Remarques sur la boucle for

- `var` est la variable incrémentée,
- `set` est un vecteur définissant les valeurs successives,
- **lente** comparée aux opérateurs matriciels.



# Exécution répétée: boucles `while` et `repeat`

## Syntaxe

```
while (condition) { expr  
}
```

ou

```
repeat { expr }
```

## Remarque

- Comme pour `for`, éviter les imbrications sources de lenteur.

## Exemples d'utilisation

```
repeat { expr if (condition) {break} }
```

ou

```
while (condition1){ expr_1 if (condition2) {next} expr_2  
}
```

## Remarque

- `break` est la seule manière d'interrompre une boucle `repeat`.

Permettent de factoriser des lignes de codes

# Définir une fonction

## Syntaxe

```
nom_de_la_fonction <- function(arg1,arg2, ...) { expression  
return(var) }
```

## Remarques

- `return` peut être omis (à éviter): dans ce cas la dernière valeur calculée est renvoyée.
- peut être tapée directement dans l'interpréteur ou dans un fichier externe `functions.R`, chargé par source.

# Un exemple simple I

## Moyenne empirique d'un vecteur

Avec suppression des valeurs manquantes !

```
moyenne <- function(x){  
  ## suppression des valeurs manquantes  
  x.not.na <- x[!is.na(x)]  
  ## moyenne empirique  
  resultat <- sum(x.not.na) / length(x.not.na)  
  return(resultat)  
}
```

# Un exemple simple II

## Tests

```
moyenne(rnorm(100))
```

```
## [1] 0.08686897
```

```
moyenne(c(1,-5,3,NA,8.7))
```

```
## [1] 1.925
```

# Les arguments, leurs valeurs par défaut

Encore un point fort de R

## Propriétés

- les arguments peuvent être passés dans le **\*\*désordre\*\*** s'ils sont **\*\*nommés\*\***: ``var=object``,
- on peut définir une valeur par défaut pour n'importe quel argument lors de la définition de la fonction: ``var=10``
- en cas de **\*\*sorties multiples\*\***, les sorties doivent être renvoyées sous forme de liste.

## Remarques

- Les valeurs par défaut rendent la lecture des fonctions beaucoup plus aisée pour l'utilisateur: **imposer peu d'arguments obligatoires** }.
- Les noms des éléments de la liste définie dans la fonction sont conservés à l'extérieur de la fonction.

# Un exemple (un tout petit peu) plus avancé I



# Un exemple (un tout petit peu) plus avancé II

## Résumé numérique d'un vecteur

```
resume <- function(x,na.rm=TRUE,affiche=FALSE) {  
  mu <- mean(x,na.rm=na.rm)  
  s2 <- var(x,na.rm=na.rm)  
  if (affiche) {  
    cat("\nMoyenne:",mu,"et variance:",s2)  
  }  
  return(list(moyenne = mu, variance = s2))  
}
```

```
out <- resume(rnorm(100))  
out$variance
```

```
## [1] 1.074956
```

```
out <- resume(affiche=TRUE,x=rexp(100,0.5))
```

```
##  
## Moyenne: 2.264771 et variance: 4.767308
```

Les packages sont des codes qui peuvent être inclus pour étendre les fonctionnalités de R

## Principe de Claerbout (Géophysicien, Stanford)

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

## Démarche

- Proposer une méthode et exposer dans un article ses propriétés,
- Écrire et déposer un package sur CRAN,
- Publier un article dans « journal of statistical software » ou une note dans « Bioinformatics ».

# Simplicité de la création d'un package

## Définir un objectif

Par exemple, ``SIMoNe`` : construire un graphe des interactions significatives entre gènes à partir de données du transcriptome

## Organisation type

- Fichier DESCRIPTION
- Répertoire R : fonctions R (fonction `inferGraph(data)`)
- Répertoire man : documentation des fonctions
- Répertoire data : données
- (Répertoire src : pour les fichiers à compiler, header etc.)