A Book for **Statistics**

I Joly

2019-12-13

Contents

Chapter 1

Prerequisites

1.1 Compile the book in local

```
bookdown::render_book("index.Rmd", "bookdown::gitbook")
```

1.2 Push on Github

```
git commit -m "Started book" git push -u origin master
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 2

Preface

```
library(ggplot2);library(knitr)
```

2.1 To Do List

- Voir multioutput : pdf + html
- Theme Tufte
- Insert de Shiny
- Faire un pdf: `Rscript -e "bookdown::render_book('index.Rmd', 'bookdown::pdf_book')"`

2.2 A mettre en place

- `fig_aption=TRUE` dans le Yalm, mais où ?
- `split_by` split en plusieurs fichiers HTML (par `rmd`, `chapter section none chapter+number section+number`)
- `split_bib= T or F` pour bib en fin de chaque page ou fin de doc

2.3 Rappels d'édition

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 2. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter **??**.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 2.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 2.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package [R-bookdown] in this sample book, which was built on top of R Markdown and **knitr** [xie2015].

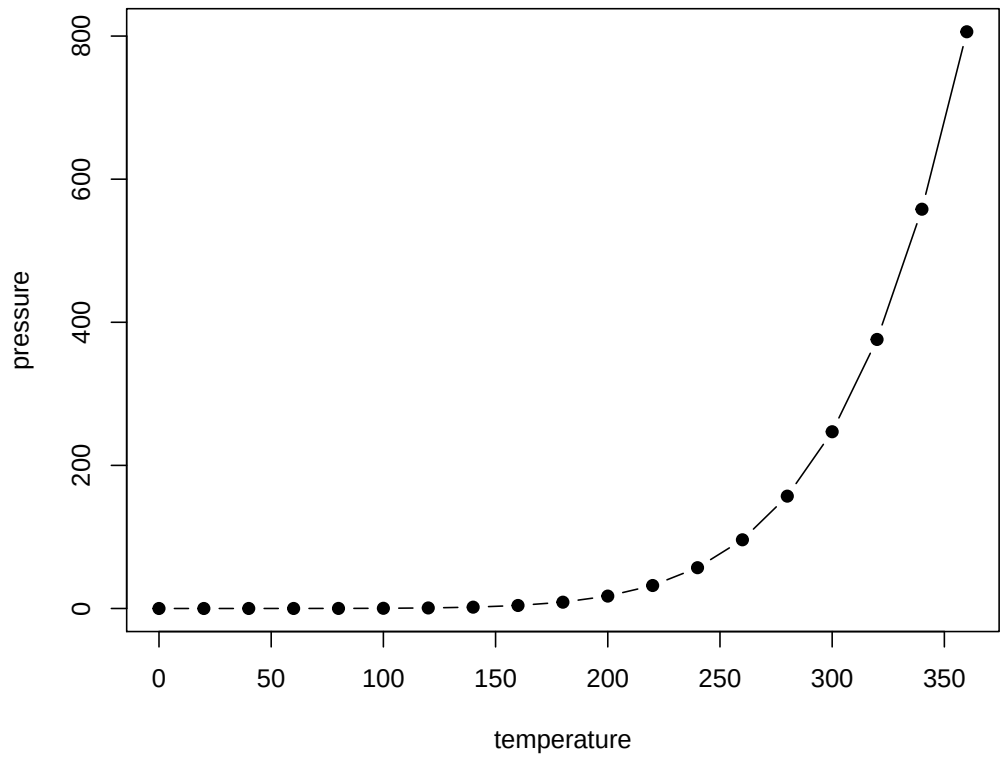

Figure 2.1: Here is a nice figure!

Table 2.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

Chapter 3

Premières opérations sous R

3.1 Les packages

Installation de packages

`R` a une configuration de base, qui peut être complétée par de nombreux packages ajoutant différentes fonctions.

Les packages doivent être installés une première fois sur le pc avant de pouvoir être utilisés. `install.packages("pack")` téléchargera est installera le package [1].

Chargement de packages

Les packages installés sur une machine ne sont pas chargés en mémoire au démarrage. On charge les packages requis dans l'environnement *R* par `library()` ou `require()`

Les fonctions et les données contenus dans le package sont alors accessibles.

Updating de R et ses packages

Le package `installr`propose la fonction `updateR()`, qui met automatiquement à jour la version de *R* et les packages.

3.2 Working directory: dossier de travail

Sans précision particulière, les fichiers seront chargés et sauvegardés dans le `working directory` (wd): le dossier dans lequel *R* va stocker et rechercher les fichiers. Les fonctions `getwd()` et `setwd()` permettent respectivement de voir le dossier de travail actuel et de le modifier.

Chemin d'accès sous *R*: *R* utilise un autre type d'écriture de chemin d'accès que celui de Windows (`"C:\Dossier\sous-dossier"`). Sous *R*, il faut utiliser `"C:/Dossier/sous-dossier"` ou ”`C:\\Dossier\\sous-dossier"`. Et sous Mac : `"/Users/Moi/Dossier/"`

3.3 Codage R:

Commandes et scripts Le code *R* peut être saisi directement dans la console ligne à ligne ou être sauvé comme un script: une succession de lignes de code. Un script peut être sauvegardé dans un fichier d'extension `.R`. Un script peut être exécuté de plusieurs façons:

[1] voir le fichier `install.packages.R` qui regroupe les packages utilisés pour ce cours et ceux conseillés.

- en gardant le curseur sur la ligne à exécuter et en cliquant sur le bouton `run` (ou avec le raccourcis clavier `CTRL+ENTREE`). Après éxecution, le curseur passe à la ligne suivante.
- en sélectionnant la ou les lignes à exécuter et en cliquant sur `run` (ou `CTRL+ENTREE`)
- par la fonction `source("monscript.R")`.

Les commandes dans le script sont séparées soit par `;` soit par un retour à la ligne

R is case sensitive. Majuscules et minuscules sont prises en compte dans les noms de fonctions, d'objets, etc.

Le caratère `#` précède des éléments de commentaires et ne sont pas exécutés.

Certaines commandes s'étendent sur plusieurs lignes. On utilise `+` à la fin d'une ligne pour une commande multilignes.

Programme R: Objets

R stocke les données et les résultats dans des objets. On assigne et on stocke dans des objets avec l'opérateur `<-`.

Pour afficher le contenu d'un objet, il suffit d'exécuter son nom.

Pour afficher tous les objets de l'environnement (ou la mémoire de R) : `ls()`

Par exemple:

```
# stocke 3 dans l'objet abc
abc <- 3
# affiche le contenu de abc
abc
```

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

```
# affiche le contenu de l'environnement
ls()
```

```
## [1] "abc"
```

Programme R: Fonctions

Les fonctions sont essentielles pour le travail des données sous *R*.

Comme pour les fonctions mathématiques, les fonction R réalisent des opérations sur un *input* et renvoient un *output*. Par exemple, la fonction `mean(x)` prend le vecteur de nombres `x` et renvoie leur moyenne. Les inputs sont souvent appelés des arguments de la fonction.

Programme R: Aide de R

L'aide et la documentation des fonctions et des objets de *R* sont accessibles par `?`

Par exemple `?mean`

Chapter 4

Manipulation des objets sous R

4.1 Structures de données

Plusieurs types d'objets de *R* peuvent contenir des données.

4.2 Les Vecteurs

Les vecteurs sont unidimentionnels et homogènes. Une variable seule peut être représentée par un vecteur de l'un des types suivants:
- `logical`: la variable prend les valeurs `TRUE` ou `FALSE` (1 ou 0)
- `integer`: un entier uniquement (représenté par un nombre suivi de `L`)
- `numeric` ou `double`: un nombre réel
- `character`: une chaine de caractères (du texte)

Une valeur seule est un vecteur de longueur 1.

`c()` est la fonction de concaténation. elle combine les valeurs de différents type en un vecteur.
`typeof()` identifie le type du vecteur. `length()` renvoie la longueur du vecteur.

Exemple:

```
mon_vec <- c(1, 3, 5)
mon_vec
```

```
## [1] 1 3 5
```

```
typeof(mon_vec)
```

```
## [1] "double"
```

```
char_vec <- c("these", "are", "some", "words")
length(char_vec)
```

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

Répétition et séquence
`rep()` créé un vecteur en répétant les élements:
`seq()` créé un vecteur en faisant une séquence d'éléments.
L'expression `m:n` génère une séquence d'entiers de `mà n`
Par exemple:

```r
# répéter 3 fois 0
rep(0, times=3)
```

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

```r
# répéter 4 fois "abc"
rep("abc", 4)
```

```
## [1] "abc" "abc" "abc" "abc"
```

```r
# Séquence de 1 à 5 avec un pas de 2
seq(from=1, to=5, by=2)
```

```
## [1] 1 3 5
```

```r
# Séquence de 10 à 0 avec un pas de -5
seq(10, 0, -5)
```

```
## [1] 10  5  0
```

```r
# Séquence d'entiers de 3 à 7
3:7
```

```
## [1] 3 4 5 6 7
```

```r
# Répétition de séquence:
rep(seq(1,3,1), times=2)
```

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

```r
# Séquence de répétition
rep(seq(1,3,1), each=2)
```

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

4.3 Recyclage vectoriel

Lorsque des opérations sont réalisées sur deux vecteurs ou plus, dont les dimensions sont inégales, les valeurs du vecteur le plus court seront *'recyclées'* pour compléter le vecteur jusqu'à obtenir la même dimension que le vecteur le plus long.

Dans l'exemple ci-dessous, le vecteur `c(1)` de taille 1 est recyclé pour devenir `c(1,1,1)`

```r
c(1,2,3) + 1
```

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

Ici, le vecteur `c(1,2)` est recyclé deux fois pour devenir `c(1,2,1,2,1,2)`

```r
c(1,2,3,4,5,6) + c(1,2)
```

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

`c(2)` devient `c(2,2,2)`

```r
c(1,2,3) < 2
```

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

Si le recyclage est partiel, on a le message

```r
c(2,3,4) + c(10, 20)
```

```
## Warning in c(2, 3, 4) + c(10, 20): la taille d'un objet plus long n'est pas
## multiple de la taille d'un objet plus court
```

```
## [1] 12 23 14
```

4.4 Sélectionner des sous-ensembles de vecteur avec []

Les éléments composant un vecteur peuvent être sélectionnés en spécifiant sa position dans le vecteur entre []

```
# créer un vecteur de 10 à 1
a <- seq(10,1,-1) ; a
```

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

Par exemple : Sélectionner le 2ème élément

```
a[2]
```

```
## [1] 9
```

Sélectionner les 5 premiers éléments

```
a[seq(1,5)]
```

```
## [1] 10  9  8  7  6
```

Sélection des premier, troisième et quatrième éléments:

```
a[c(1,3,4)]
```

```
## [1] 10  8  7
```

Les éléments peuvent être nommés, et appelés par leur nom.

```
scores <- c(John=25, Marge=34, Dan=24, Emily=29)
scores[c("John", "Emily")]
```

```
##  John Emily
##    25    29
```

4.5 Sélection conditionnelle - sous ensemble conditionné

Le *subsetting* logique peut être fait avec un vecteur (TRUE/FALSE)

```
scores[c(FALSE, TRUE, TRUE, FALSE)]
```

```
## Marge   Dan
##    34    24
```

Il peut aussi être fait avec une condition

```
scores[scores<30]
```

```
##  John   Dan Emily
##    25    24    29
```

4.6 Listes

Les listes sont des éléments uni-dimensionnels, mais dont les éléments peuvent être de types différents : des vecteurs (de toute dimension), des listes, des matrices, des dataframes.

Génération de liste par 'list()' Une liste composée d'un vecteur, d'un vecteur d'entiers et un vecteur de caractères.

```
mylist <- list(1.1, c(1L,3L,7L), c("abc", "def"))
mylist
```

```
## [[1]]
## [1] 1.1
##
## [[2]]
## [1] 1 3 7
##
## [[3]]
## [1] "abc" "def"
```

Les éléments peuvent être nommés

```
mary_info <- list(classes=c("Biology", "Math", "Music",
                            "Physics"),
                  friends=c("John", "Dan", "Emily"),
                  SAT=1450)
mary_info
```

```
## $classes
## [1] "Biology" "Math"    "Music"   "Physics"
##
## $friends
## [1] "John"  "Dan"   "Emily"
##
## $SAT
## [1] 1450
```

Accès aux éléments de listes

Comme précédemment, l'accès aux composants d'une liste se fait soit par `[[]]` pour accèder avec la position, soit avec `$` pour accèder avec le nom.

Par la position

```
mary_info[[2]]
```

```
## [1] "John"  "Dan"   "Emily"
```

Par le nom

```
mary_info$SAT
```

```
## [1] 1450
```

Le second élément du vecteur `friend`

```
mary_info$friends[2]
```

```
## [1] "Dan"
```

4.7 Matrices

Les matrices sont des structures de données à deux dimensions et homogènes.

Créer une matrice 'matrix()'

L'input de `matrix()` est un vecteur qui est transformé en une matrice en 2 dimensions, selon les spécifications `nrow` et `ncol`.

Par défaut, la matrice est remplie colonne par colonne, cela est modifié par `byrow=T`

Par exemple: une matrice 2lx3c, remplie par colonne:

```
# on remplit les colonnes avec le vecteur de 1 à 6 sur 2 lignes
a <- matrix(1:6, nrow=2)
a
```

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

Remplissage par ligne

```
b <- matrix(5:14, nrow=2, byrow=TRUE)
b
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    5    6    7    8    9
## [2,]   10   11   12   13   14
```

Accès aux éléments d'une matrice - '[row,column]'

L'omission de `row` correspond à la sélection de toutes les lignes. L'omission de `column` sélectionne toutes les colonnes.

Sélection de la ligne 2, colonne 3

```
a[2,3]
```

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

Toutes les lignes de la colonne 2

```
b[,2]
```

```
## [1]  6 11
```

Toutes les colonnes de la ligne 1

```
a[1,]
```

```
## [1] 1 3 5
```

4.8 Dataframe

Les données de l'analyse statistique sont typiquement stockées dans des `dataframe`.

Elles combinent les possibilités des matrices et des listes.

Comme pour les matrices, les `dataframe` sont rectangulaires, les colonnes sont les variables et les lignes sont les observations de ces variables. Comme les listes, les `dataframe` peuvent avoir des éléments (les colonnes) de différents types (numérique, caractère, facteurs, logique,etc.)

Mais les éléments d'un `dataframe` doivent être de longueur égale.

Créer un dataframe avec 'data.frame()'

La syntaxe est la même que pour `list()`, mais les éléments doivent être de même longueur. Les éléments peuvent être nommés.

Par exemple, un vecteur logique et un vecteur numérique

```
mydata <- data.frame(diabetic = c(TRUE, FALSE, TRUE, FALSE),
                     height = c(65, 69, 71, 73))
mydata
```

```
##   diabetic height
## 1     TRUE     65
## 2    FALSE     69
## 3     TRUE     71
## 4    FALSE     73
```

Sous ensemble de dataframes

Comme pour les matrices et les listes, les sélections sont faites avec`[rows, columns]` ou avec le nom de la variable.

Par exemple

```
mydata[3,2]
```

```
## [1] 71
```

Ou

```
mydata[1:2, "height"]
```

```
## [1] 65 69
```

Ou

```
mydata[,"diabetic"]
```

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

\textbf{Accéder aux variables ou colonnes par le nom ou $} Par exemple

```
mydata$height[2:3]
```

```
## [1] 69 71
```

```
mydata[["height"]]
```

```
## [1] 65 69 71 73
```

```
mydata[["height"]][2]
```

```
## [1] 69
```

Nommer les variables d'un dataframe

`colnames(data_frame)` renvoie les noms des colonnes d'une dataframe (ou d'une matrice).
`colnames(data_frame) <- c("some", "names")` assigne des noms aux colonnes.
Par exemple

```
colnames(mydata)
```

```
## [1] "diabetic" "height"
```

```
colnames(mydata) <- c("Diabetic", "Height")
colnames(mydata)
```

```
## [1] "Diabetic" "Height"
```

Pour changer le nom d'une variable:

```r
colnames(mydata)[1] <- "Diabetes"
colnames(mydata)
```

```
## [1] "Diabetes" "Height"
```

Enfin on peut *"copier/coller"* une variable sous condition :

```r
dat[dat$Subject=='s2',]$RT <- dat[dat$Subject=='s2',]$Trial
```

Examiner la structure d'un objet

Pour les objets à 2 dimensions `dim()` renvoie le nombre de lignes et de colonnes.

On utilise `str()` pour connaître la structure d'un objet et le type de ses éléments.

```r
dim(mydata)
```

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

```r
str(mydata)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ Diabetes: logi  TRUE FALSE TRUE FALSE
##  $ Height  : num  65 69 71 73
```

Chapter 5

Chargement de données sous R

5.1 Manipulation de données

Chargement de données sous R

Débutant avec *R*, il peut sembler efficace de manipuler les données avec un logiciel familier tel que *MS Excel*, pour corriger les erreurs, ajouter quelques informations ou faire quelques calculs simples.

Pourtant, le travail sous *R* est encouragé ici, pour 100% des opérations sur les données !

Quelques arguments:

- *MS Excel* charge les données réelles, ce qui signifie que les modifications sont sauvegardées dans le fichier de données même. De plus, aucun historique des manipulations n'est sauvegardé, il est donc difficile, voire impossible, de conserver une trace des changements. *Au contraire, R*, charge une copie des données dans sa mémoire et n'altère pas directement les données. Il permet de fonctionner avec des scripts, qui listent les manipulations appliquées aux données, les rendant ainsi **reproductibles** et **transparentes**.

- Travailler directement sur les données peut produire des erreurs qui sont alors difficiles à identifier et à corriger. Avec *R* il est facile de détecter les erreurs et problèmes systématiques à partir de petits programmes.

- Différents formats de données peuvent être chargés avec *R*, notamment les formats de logiciels spécialisés tels que SAS, Spss, Stata, etc.

Formats de données - exemples

Dataframe Une base de données sous *R* est appelée une `dataframe`, équivalent à une matrice de données où les lignes sont indéxées par leur numéro et les colonnes par des noms de variables. Les bases de données d'autres formats (par ex. ceux d'autres logiciels) peuvent être importées (ou chargées) sous *R*.

Formats de données R
Les fichiers `.Rdata` ou `.rda` contiennent un ou plusieurs objets `R` (dataframes, fonctions, shapefiles, etc.)

```
# Chargement d'un fichier (fictif) de données dans R:
load('DATA\\Movie.Data.Clean.RData')
# utiliser ls() pour lister les objets de l'environnement
ls()
```

```
##  [1] "a"          "abc"        "b"          "char_vec"   "mary_info"
##  [6] "mon_vec"    "Movie.Data" "mydata"     "mylist"     "scores"
```

Les fichiers `.rds` contiennent un objet *R* sans nom. Cet objet est donc stocké dans une seule variable. `readRDS('file.rds')` permet de charger un tel fichier et `str(newdat)`de savoir quels objets sont dans l'objet chargé.

Les fichers texte

Les fichiers `.txt`ou `.csv` sont des fichiers de données brutes, sans mise en forme des données (par exemple format de date).

Dans un fichier texte, une ligne correspond à une observation (une ligne du dataframe). Sur la ligne, les valeurs des différentes colonnes sont séparées par un caractère séparateur (souvent le `;` ou la tabulation).

`read.table('file.txt', header=TRUE)` permet de charger un fichier texte où la 1ère ligne contient les noms de variables.

Problèmes classiques de lecture des fichiers textes

1. *R* ne trouve pas le fichier que vous voulez charger:

- Vérifiez votre working directory (dossier de travail) `getwd()`. Ce dossier contient-il le fichier que vous voulez charger ? Modifiez ce dossier avec `setwd()`.

- Parfois il est utile de préciser tout le chemin d'accès dans le nom du fichier à charger. Par exemple: `"C://Documents//data.txt"` sous Windows ou sous Mac `"/Users/Moi/Documents/data.txt"`.

2. `Incorrect delimiter`. Aucun message d'erreur n'est affiché, mais les données paraissent étranges. Le séparateur de variable est mal spécifié. Excel utilise souvent la `,` alors que d'autres logiciels utilisent l'espace ou la tabulation. `dat <- read.table("file.csv", header=TRUE, sep=',')`permet de préciser le séparateur.

3. *Décimale*. Les données issues de locigiels anglosaxons utilisent le `.` pour indiquer les décimales, plutôt que la `,`. Cela peut être précisé avec: `dat <- read.table("file.csv", header=TRUE, sep=',', dec='.')`

Cependant, le `.` est aussi parfois utilisé pour séparer les milliers. C'est alors à l'exportation des données que l'on doit s'assurer du caractère des décimales et qu'il n'y a pas de caractère pour les milliers.

4. *Erreurs sur les variables*. Dans certains cas, le problème d'importation affecte une ou plusieurs variables.

- Une chaine de caractère est reconnue comme un facteur : `as.character(data$V1)` pour convertir une variable en caractère.
- Dans un champs de caractère, le `.` est utilisé pour les milliers, alors que la `,`marque les décimales. `gsub("\\.", "", data$V1)`supprime les `.` dans le champs de valeurs.
- La `,`marque les décimales au lieu du `.`: `gsub("\\,", "\\.", data$V1)` : remplace les `,` par un `.`
- pour convertir une chaine de caractères en numérique: `as.numeric(data$V1)`

Par exemple:

```
data <- read.table("file.txt", header=TRUE, stringsAsFactors=FALSE)
data$V1 <- as.numeric(gsub("\\,", "\\.", gsub("\\.", "", data$V1)))
```

5. les valeurs manquantes sont indiquées par `.`au lieu d'un vide: `NA`:
`read.table("file.txt", header=TRUE, na.strings='.')` précise le codage des vides. Si deux codages existent dans le fichier des `.` et les caractères *NA* : `read.table("file.txt", header=TRUE, na.strings=c('.', NA))`

6. *Nom de colonne manquant*:
R considère qu'une colonne sans nom contient les noms de lignes.`read.table("file.csv", header=TRUE, row.names=NULL)` force la colonne sans nom à être prise comme une variable.

7. *Lignes de longueur différente*:
   Si des lignes ont des longueurs différentes, *R* ne retrouve pas toutes les variables. Il faut le forcer par :`read.table("file.csv", header=TRUE, fill=TRUE)`

8. *drop des premières lignes*:
   On peut commencer la lecture du fichier au numéro de ligne souhaité: `read.table("file.csv", header=TRUE, skip=4)`: pour commencer à la ligne 5.

Recommendations de lecture:
- https://stats.idre.ucla.edu/stat/data/intro_r/intro_r_flat.html

5.2 Les packages

Installation de packages

`R` a une configuration de base, qui peut être complétée par de nombreux packages ajoutant différentes fonctions.

Les packages doivent être installés une première fois sur le pc avant de pouvoir être utilisés. `install.packages("pack")` téléchargera est installera le package [1].

Chargement de packages

Les packages installés sur une machine ne sont pas chargés en mémoire au démarrage. On charge les packages requis dans l'environnement *R* par `library()` ou `require()`

Les fonctions et les données contenus dans le package sont alors accessibles.

Updating de R et ses packages

Le package `installr`propose la fonction `updateR()`, qui met automatiquement à jour la version de *R* et les packages.

5.3 Working directory: dossier de travail

Sans précision particulière, les fichiers seront chargés et sauvegardés dans le `working directory` (wd): le dossier dans lequel *R* va stocker et rechercher les fichiers. Les fonctions `getwd()` et `setwd()` permettent respectivement de voir le dossier de travail actuel et de le modifier.

Chemin d'accès sous *R*: *R* utilise un autre type d'écriture de chemin d'accès que celui de Windows (`"C:\Dossier\sous-dossier"`). Sous *R*, il faut utiliser `"C:/Dossier/sous-dossier"` ou ”`C:\\Dossier\\sous-dossier"`. Et sous Mac : `"/Users/Moi/Dossier/"`

5.4 Codage R:

Commandes et scripts Le code *R* peut être saisi directement dans la console ligne à ligne ou être sauvé comme un script: une succession de lignes de code. Un script peut être sauvegardé dans un fichier d'extension `.R`. Un script peut être exécuté de plusieurs façons:

- en gardant le curseur sur la ligne à exécuter et en cliquant sur le bouton `run` (ou avec le raccourcis clavier `CTRL+ENTREE`). Après éxecution, le curseur passe à la ligne suivante.
- en sélectionnant la ou les lignes à exécuter et en cliquant sur `run` (ou `CTRL+ENTREE`)
- par la fonction `source("monscript.R")`.

[1] voir le fichier `install.packages.R` qui regroupe les packages utilisés pour ce cours et ceux conseillés.

Les commandes dans le script sont séparées soit par `;` soit par un retour à la ligne

R is case sensitive. Majuscules et minuscules sont prises en compte dans les noms de fonctions, d'objets, etc.

Le caratère `#` précède des éléments de commentaires et ne sont pas exécutés.

Certaines commandes s'étendent sur plusieurs lignes. On utilise `+` à la fin d'une ligne pour une commande multilignes.

Programme R: Objets

R stocke les données et les résultats dans des objets. On assigne et on stocke dans des objets avec l'opérateur `<-`.

Pour afficher le contenu d'un objet, il suffit d'exécuter son nom.

Pour afficher tous les objets de l'environnement (ou la mémoire de R) : `ls()`

Par exemple:

```
# stocke 3 dans l'objet abc
abc <- 3
# affiche le contenu de abc
abc
```

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

```
# affiche le contenu de l'environnement
ls()
```

```
##  [1] "a"          "abc"        "b"          "char_vec"   "mary_info"
##  [6] "mon_vec"    "Movie.Data" "mydata"     "mylist"     "scores"
```

Programme R: Fonctions

Les fonctions sont essentielles pour le travail des données sous *R*.

Comme pour les fonctions mathématiques, les fonction R réalisent des opérations sur un *input* et renvoient un *output*. Par exemple, la fonction `mean(x)` prend le vecteur de nombres `x` et renvoie leur moyenne. Les inputs sont souvent appelés des arguments de la fonction.

Programme R: Aide de R

L'aide et la documentation des fonctions et des objets de *R* sont accessibles par `?`

Par exemple `?mean`

5.5 Structures de données

Plusieurs types d'objets de *R* peuvent contenir des données.

5.6 Les Vecteurs

Les vecteurs sont unidimentionnels et homogènes. Une variable seule peut être représentée par un vecteur de l'un des types suivants:
- `logical`: la variable prend les valeurs `TRUE` ou `FALSE` (1 ou 0)
- `integer`: un entier uniquement (représenté par un nombre suivi de `L`)
- `numeric` ou `double`: un nombre réel
- `character`: une chaine de caractères (du texte)

Une valeur seule est un vecteur de longueur 1.

`c()` est la fonction de concaténation. elle combine les valeurs de différents type en un vecteur.
`typeof()` identifie le type du vecteur. `length()` renvoie la longueur du vecteur.

Exemple:

```
mon_vec <- c(1, 3, 5)
mon_vec
```

```
## [1] 1 3 5
```

```
typeof(mon_vec)
```

```
## [1] "double"
```

```
char_vec <- c("these", "are", "some", "words")
length(char_vec)
```

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

Répétition et séquence
`rep()` créé un vecteur en répétant les élements:
`seq()` créé un vecteur en faisant une séquence d'éléments.
L'expression `m:n` génère une séquence d'entiers de `m` à `n`
Par exemple:

```
# répéter 3 fois 0
rep(0, times=3)
```

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

```
# répéter 4 fois "abc"
rep("abc", 4)
```

```
## [1] "abc" "abc" "abc" "abc"
```

```
# Séquence de 1 à 5 avec un pas de 2
seq(from=1, to=5, by=2)
```

```
## [1] 1 3 5
```

```
# Séquence de 10 à 0 avec un pas de -5
seq(10, 0, -5)
```

```
## [1] 10  5  0
```

```
# Séquence d'entiers de 3 à 7
3:7
```

```
## [1] 3 4 5 6 7
```

```
# Répétition de séquence:
rep(seq(1,3,1), times=2)
```

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

```
# Séquence de répétition
rep(seq(1,3,1), each=2)
```

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

5.7 Recyclage vectoriel

Lorsque des opérations sont réalisées sur deux vecteurs ou plus, dont les dimensions sont inégales, les valeurs du vecteur le plus court seront *'recyclées'* pour compléter le vecteur jusqu'à obtenir la même dimension que le vecteur le plus long.

Dans l'exemple ci-dessous, le vecteur `c(1)` de taille 1 est recyclé pour devenir `c(1,1,1)`

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

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

Ici, le vecteur `c(1,2)` est recyclé deux fois pour devenir `c(1,2,1,2,1,2)`

```
c(1,2,3,4,5,6) + c(1,2)
```

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

`c(2)` devient `c(2,2,2)`

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

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

Si le recyclage est partiel, on a le message

```
c(2,3,4) + c(10, 20)
```

```
## Warning in c(2, 3, 4) + c(10, 20): la taille d'un objet plus long n'est pas
## multiple de la taille d'un objet plus court
```

```
## [1] 12 23 14
```

5.8 Sélectionner des sous-ensembles de vecteur avec []

Les éléments composant un vecteur peuvent être sélectionnés en spécifiant sa position dans le vecteur entre []

```
# créer un vecteur de 10 à 1
a <- seq(10,1,-1) ; a
```

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

Par exemple : Sélectionner le 2ème élément

```
a[2]
```

```
## [1] 9
```

Sélectionner les 5 premiers éléments

```
a[seq(1,5)]
```

```
## [1] 10  9  8  7  6
```

Sélection des premier, troisième et quatrième éléments:

```
a[c(1,3,4)]
```

```
## [1] 10  8  7
```

Les éléments peuvent être nommés, et appelés par leur nom.

```
scores <- c(John=25, Marge=34, Dan=24, Emily=29)
scores[c("John", "Emily")]
```

```
##  John Emily
##    25    29
```

5.9 Sélection conditionnelle - sous ensemble conditionné

Le *subsetting* logique peut être fait avec un vecteur `(TRUE/FALSE)`

```
scores[c(FALSE, TRUE, TRUE, FALSE)]
```

```
## Marge   Dan
##    34    24
```

Il peut aussi être fait avec une condition

```
scores[scores<30]
```

```
##  John   Dan Emily
##    25    24    29
```

5.10 Listes

Les listes sont des éléments uni-dimensionnels, mais dont les éléments peuvent être de types différents : des vecteurs (de toute dimension), des listes, des matrices, des dataframes.

Génération de liste par 'list()' Une liste composée d'un vecteur, d'un vecteur d'entiers et un vecteur de caractères.

```
mylist <- list(1.1, c(1L,3L,7L), c("abc", "def"))
mylist
```

```
## [[1]]
## [1] 1.1
##
## [[2]]
## [1] 1 3 7
##
## [[3]]
## [1] "abc" "def"
```

Les éléments peuvent être nommés

```
mary_info <- list(classes=c("Biology", "Math", "Music",
                            "Physics"),
                  friends=c("John", "Dan", "Emily"),
                  SAT=1450)
mary_info
```

```
## $classes
## [1] "Biology" "Math"    "Music"   "Physics"
##
## $friends
## [1] "John"  "Dan"   "Emily"
##
## $SAT
## [1] 1450
```

Accès aux éléments de listes

Comme précédemment, l'accès aux composants d'une liste se fait soit par `[[]]` pour accèder avec la position, soit avec `$` pour accèder avec le nom.

Par la position

```
mary_info[[2]]
```

```
## [1] "John"  "Dan"   "Emily"
```

Par le nom

```
mary_info$SAT
```

```
## [1] 1450
```

Le second élément du vecteur `friend`

```
mary_info$friends[2]
```

```
## [1] "Dan"
```

5.11 Matrices

Les matrices sont des structures de données à deux dimensions et homogènes.

Créer une matrice 'matrix()'

L'input de `matrix()` est un vecteur qui est transformé en une matrice en 2 dimensions, selon les spécifications `nrow` et `ncol`.

Par défaut, la matrice est remplie colonne par colonne, cela est modifié par `byrow=T`

Par exemple: une matrice 2lx3c, remplie par colonne:

```
# on remplit les colonnes avec le vecteur de 1 à 6 sur 2 lignes
a <- matrix(1:6, nrow=2)
a
```

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

Remplissage par ligne

```
b <- matrix(5:14, nrow=2, byrow=TRUE)
b
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    5    6    7    8    9
## [2,]   10   11   12   13   14
```

Accès aux éléments d'une matrice - '[row,column]'

L'omission de `row` correspond à la sélection de toutes les lignes. L'omission de `column` sélectionne toutes les colonnes.

Sélection de la ligne 2, colonne 3

```
a[2,3]
```

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

Toutes les lignes de la colonne 2

```
b[,2]
```

```
## [1]  6 11
```

Toutes les colonnes de la ligne 1

```
a[1,]
```

```
## [1] 1 3 5
```

5.12 Dataframe

Les données de l'analyse statistique sont typiquement stockées dans des `dataframe`.

Elles combinent les possibilités des matrices et des listes.

Comme pour les matrices, les `dataframe` sont rectangulaires, les colonnes sont les variables et les lignes sont les observations de ces variables. Comme les listes, les `dataframe` peuvent avoir des éléments (les colonnes) de différents types (numérique, caractère, facteurs, logique,etc.)

Mais les éléments d'un `dataframe` doivent être de longueur égale.

Créer un dataframe avec 'data.frame()'

La syntaxe est la même que pour `list()`, mais les éléments doivent être de même longueur. Les éléments peuvent être nommés.

Par exemple, un vecteur logique et un vecteur numérique

```
mydata <- data.frame(diabetic = c(TRUE, FALSE, TRUE, FALSE),
                     height = c(65, 69, 71, 73))
mydata
```

```
##   diabetic height
## 1     TRUE     65
## 2    FALSE     69
## 3     TRUE     71
## 4    FALSE     73
```

Sous ensemble de dataframes

Comme pour les matrices et les listes, les sélections sont faites avec`[rows, columns]` ou avec le nom de la variable.

Par exemple

```
mydata[3,2]
```

```
## [1] 71
```

Ou

```
mydata[1:2, "height"]
```

```
## [1] 65 69
```

Ou

```
mydata[,"diabetic"]
```

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

\textbf{Accéder aux variables ou colonnes par le nom ou $} Par exemple

```
mydata$height[2:3]
```

```
## [1] 69 71
```

```
mydata[["height"]]
```

```
## [1] 65 69 71 73
```

```
mydata[["height"]][2]
```

```
## [1] 69
```

Nommer les variables d'un dataframe

`colnames(data_frame)` renvoie les noms des colonnes d'une dataframe (ou d'une matrice).
`colnames(data_frame) <- c("some", "names")` assigne des noms aux colonnes.
Par exemple

```
colnames(mydata)
```

```
## [1] "diabetic" "height"
```

```
colnames(mydata) <- c("Diabetic", "Height")
colnames(mydata)
```

```
## [1] "Diabetic" "Height"
```

Pour changer le nom d'une variable:

```
colnames(mydata)[1] <- "Diabetes"
colnames(mydata)
```

```
## [1] "Diabetes" "Height"
```

Enfin on peut *"copier/coller"* une variable sous condition :

```
dat[dat$Subject=='s2',]$RT <- dat[dat$Subject=='s2',]$Trial
```

Examiner la structure d'un objet

Pour les objets à 2 dimensions `dim()` renvoie le nombre de lignes et de colonnes.

On utilise `str()` pour connaître la structure d'un objet et le type de ses éléments.

```
dim(mydata)
```

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

```
str(mydata)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ Diabetes: logi  TRUE FALSE TRUE FALSE
##  $ Height  : num  65 69 71 73
```

5.13 Manipulation de données

Chargement de données sous R

Débutant avec *R*, il peut sembler efficace de manipuler les données avec un logiciel familier tel que *MS Excel*, pour corriger les erreurs, ajouter quelques informations ou faire quelques calculs simples.

Pourtant, le travail sous *R* est encouragé ici, pour 100% des opérations sur les données !

Quelques arguments:

- *MS Excel* charge les données réelles, ce qui signifie que les modifications sont sauvegardées dans le fichier de données même. De plus, aucun historique des manipulations n'est sauvegardé, il est donc difficile, voire impossible, de conserver une trace des changements. *Au contraire, R*, charge une copie des données dans sa mémoire et n'altère pas directement les données. Il permet de fonctionner avec des scripts, qui listent les manipulations appliquées aux données, les rendant ainsi **reproductibles** et **transparentes**.

- Travailler directement sur les données peut produire des erreurs qui sont alors difficiles à identifier et à corriger. Avec *R* il est facile de détecter les erreurs et problèmes systématiques à partir de petits programmes.

- Différents formats de données peuvent être chargés avec *R*, notamment les formats de logiciels spécialisés tels que SAS, Spss, Stata, etc.

Formats de données - exemples

Dataframe Une base de données sous *R* est appelée une `dataframe`, équivalent à une matrice de données où les lignes sont indéxées par leur numéro et les colonnes par des noms de variables. Les bases de données d'autres formats (par ex. ceux d'autres logiciels) peuvent être importées (ou chargées) sous *R*.

Formats de données R
Les fichiers `.Rdata` ou `.rda` contiennent un ou plusieurs objets `R` (dataframes, fonctions, shapefiles, etc.)

```
# Chargement d'un fichier (fictif) de données dans R:
load('DATA\\Movie.Data.Clean.RData')
# utiliser ls() pour lister les objets de l'environnement
ls()
```

```
##  [1] "a"          "abc"        "b"          "char_vec"   "mary_info"
##  [6] "mon_vec"    "Movie.Data" "mydata"     "mylist"     "scores"
```

Les fichiers `.rds` contiennent un objet *R* sans nom. Cet objet est donc stocké dans une seule variable. `readRDS('file.rds')` permet de charger un tel fichier et `str(newdat)`de savoir quels objets sont dans l'objet chargé.

Les fichers texte

Les fichiers `.txt`ou `.csv` sont des fichiers de données brutes, sans mise en forme des données (par exemple format de date).

Dans un fichier texte, une ligne correspond à une observation (une ligne du dataframe). Sur la ligne, les valeurs des différentes colonnes sont séparées par un caractère séparateur (souvent le `;` ou la tabulation).

`read.table('file.txt', header=TRUE)` permet de charger un fichier texte où la 1ère ligne contient les noms de variables.

Problèmes classiques de lecture des fichiers textes

1. *R* ne trouve pas le fichier que vous voulez charger:

- Vérifiez votre working directory (dossier de travail) `getwd()`. Ce dossier contient-il le fichier que vous voulez charger ? Modifiez ce dossier avec `setwd()`.

- Parfois il est utile de préciser tout le chemin d'accès dans le nom du fichier à charger. Par exemple: `"C://Documents//data.txt"` sous Windows ou sous Mac `"/Users/Moi/Documents/data.txt"`.

2. `Incorrect delimiter`. Aucun message d'erreur n'est affiché, mais les données paraissent étranges. Le séparateur de variable est mal spécifié. Excel utilise souvent la `,` alors que d'autres logiciels utilisent l'espace ou la tabulation. `dat <- read.table("file.csv", header=TRUE, sep=',')`permet de préciser le séparateur.

3. *Décimale*. Les données issues de locigiels anglosaxons utilisent le `.` pour indiquer les décimales, plutôt que la `,`. Cela peut être précisé avec: `dat <- read.table("file.csv", header=TRUE, sep=',', dec='.')`

Cependant, le `.` est aussi parfois utilisé pour séparer les milliers. C'est alors à l'exportation des données que l'on doit s'assurer du caractère des décimales et qu'il n'y a pas de caractère pour les milliers.

4. *Erreurs sur les variables*. Dans certains cas, le problème d'importation affecte une ou plusieurs variables.

- Une chaine de caractère est reconnue comme un facteur : `as.character(data$V1)` pour convertir une variable en caractère.
- Dans un champs de caractère, le `.` est utilisé pour les milliers, alors que la `,`marque les décimales. `gsub("\\.", "", data$V1)`supprime les `.` dans le champs de valeurs.
- La `,`marque les décimales au lieu du `.`: `gsub("\\,", "\\.", data$V1)` : remplace les `,` par un `.`
- pour convertir une chaine de caractères en numérique: `as.numeric(data$V1)`

Par exemple:

```
data <- read.table("file.txt", header=TRUE, stringsAsFactors=FALSE)
data$V1 <- as.numeric(gsub("\\,", "\\.", gsub("\\.", "", data$V1)))
```

5. les valeurs manquantes sont indiquées par `.`au lieu d'un vide: `NA`:
`read.table("file.txt", header=TRUE, na.strings='.')` précise le codage des vides. Si deux codages existent dans le fichier des `.` et les caractères *NA* : `read.table("file.txt", header=TRUE, na.strings=c('.', NA))`

6. *Nom de colonne manquant*:
R considère qu'une colonne sans nom contient les noms de lignes.`read.table("file.csv", header=TRUE, row.names=NULL)` force la colonne sans nom à être prise comme une variable.

7. *Lignes de longueur différente*:
Si des lignes ont des longueurs différentes, *R* ne retrouve pas toutes les variables. Il faut le forcer par :`read.table("file.csv", header=TRUE, fill=TRUE)`

8. *drop des premières lignes*:
On peut commencer la lecture du fichier au numéro de ligne souhaité: `read.table("file.csv", header=TRUE, skip=4)`: pour commencer à la ligne 5.

Chapter 6

Methods

We describe our methods in this chapter.

Chapter 7

Applications

Some *significant* applications are demonstrated in this chapter.

7.1 Example one

7.2 Example two

Chapter 8

Final Words

We have finished a nice book.