INITIATION AU LANGAGE R

Table des matières :

INTRODUCTION

R est un langage de programmation créé pour traiter et organiser des jeux de données afin de pouvoir y appliquer des tests statistiques plus ou moins complexes et représenter ces données graphiquement à l'aide d'une grande variété de graphiques disponibles.
De nos jours, R est très utilisé par les professionnels de la data.

Développé initialement au début des années 1990 par Ross Ihaka et Robert Gentleman, ce langage (basé sur un autre langage statistique appelé S) est désormais maintenu et mis à jour par une équipe de développeurs au sein du R Project. Cette structure garantit des mises à jour fréquentes et une communauté importante d'utilisateurs apporte son aide pour notamment développer de nouvelles fonctionnalités (des librairies) au projet.

Le site internet de référence concernant le langage R est : http://cran.r-project.org/
Depuis ce site, on peut télécharger R, et trouver un grand nombre de documentation.
On trouve de plus une bonne base de documentation sur : https://www.rdocumentation.org/

Le langage R est très utilisé pour diverses raisons :

- il permet d'organiser et traiter des volumes importants de données de manière rapide et flexible.
- il permet assez facilement de créer des graphiques paramétrables afin de pouvoir mieux visualiser le résultat des analyses.
- il est entièrement gratuit et sous licence GPL, ce qui signifie que l'on peut en télécharger les sources et les modifier. C'est notamment grâce à cela qu'une communauté active peut améliorer le langage en permanence.
- il est multi-plateforme : on peut l'utiliser sous Windows, Mac OS ou Linux.

I. PRISE EN MAIN

1. Installer R

R est disponible sous différente plateformes : Windows, Mac OS et Linux.
On peut télécharger R depuis : http://cran.r-project.org/

Installation sous Windows :
Une fois le téléchargement fini, lancer le programme d'installation et suivre les instructions données par les différentes boites de dialogue, permettant par exemple de choisir la langue du protocole d'installation, ou le dossier dans lequel sera installé R.
L'installation complète ne demande que peu d'espace, et il est donc conseillé de la sélectionner.

Installation sous Ubuntu
Sous Ubuntu ou autre distribution dérivée de Debian on peut trouver R via apt-get ou synaptic. Le paquet s'appelle r-base et on peut donc l'installer en lançant la ligne de commande suivante :
sudo apt-get install r-base
Si on dispose d'un autre type de distribution ou que l'on souhaite compiler directement les sources, choisir le lien Linux sur la page de téléchargement de R et sélectionner le dossier correspondant à votre distribution pour télécharger les sources. Les instructions d'installation se trouvent alors dans un fichier d'aide présent dans le dossier téléchargé.

Installation sous Mac OS X
Sur la page de téléchargement, choisir le lien Mac OS X et dans la section Files cliquer sur le premier lien proposé (du type R-x.xx.x.pkg où x.xx.x représente le numéro de version).
Lancer alors le fichier téléchargé en prenant soin de vérifier que votre compte dispose des droits nécessaires. Suivre les instructions données par les boites de dialogue. On n'aura normalement pas à changer d'options et pouvoir donc valider chaque étape.
Un mot de passe sera demandé lors de l'installation, cela sert seulement à vérifier que vous avez les droits d'installer des logiciels sur la machine.

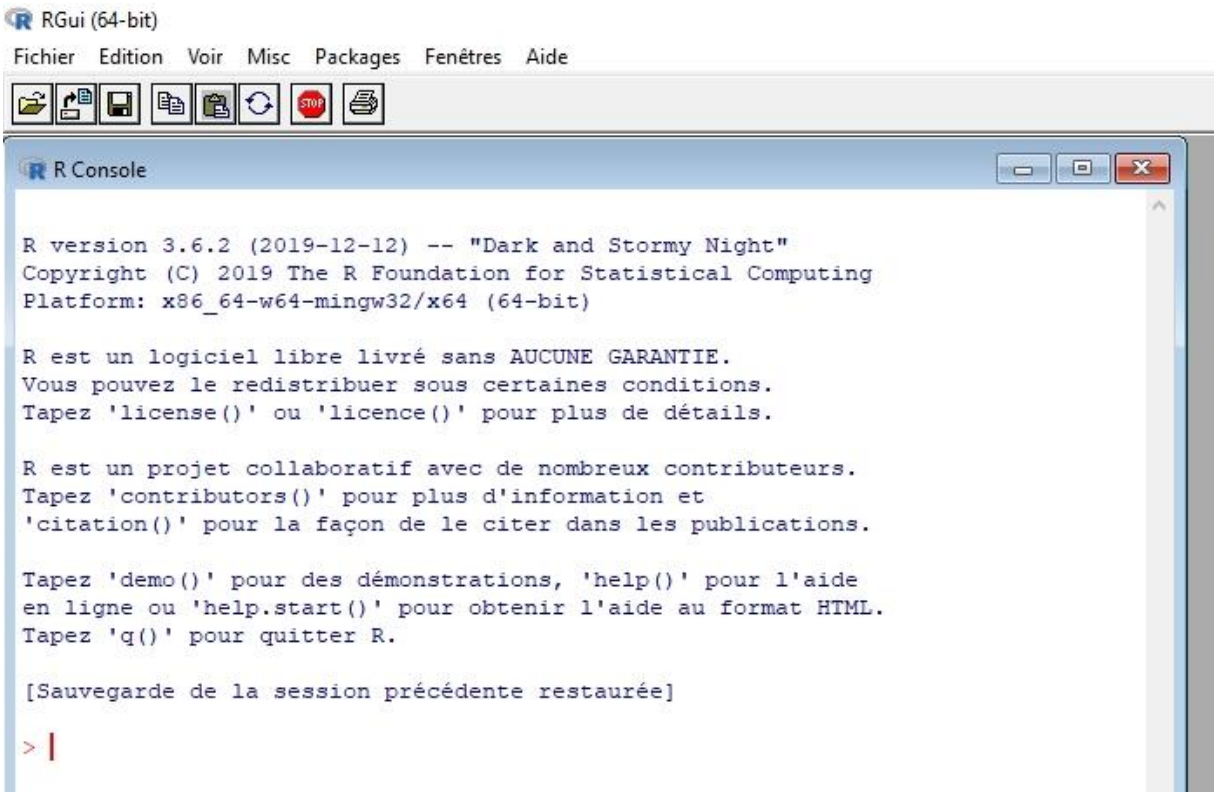
2. La console R

Le langage R est un **langage interprété**. Cela signifie que l'on peut directement écrire une ligne de code, la valider (Enter) et en voir le résultat. Il n'y a donc pas besoin d'une étape préalable de compilation du code, celui-ci est interprété à la volée.

Pour lancer R, le raccourci fourni lors de l'installation devrait suffire.
Sinon :

- sous Windows, lancer le programme Rxxx (xxx correspondant à la version installée).
- sous Mac OS X, lancer le programme R présent dans le dossier Applications.
- sous Linux (et plus généralement tout système Unix), ouvrir un terminal et lancer la commande : R

Devrait alors apparaitre une fenêtre de la sorte :



La ligne commençant par le caractère > et sur laquelle se trouve le curseur est appelée l'invite de commande (ou prompt en anglais). Elle signifie que R est disponible et en attente d'une commande.

Nous allons tout de suite lui fournir une première commande :

```
> 2 + 3
```

La réponse de R est immédiate :

```
[1] 5
```

Autres opérations arithmétiques de base :

```
> 8 - 12
[1] -4
> 14 * 25
[1] 350
> -3/10
[1] -0.3
> 5^3
[1] 125
```

Division entière (quotient euclidien) :

```
> 13 %/% 3
[1] 4
```

Reste de la division entière (modulo) :

```
> 13 %% 3
[1] 1
```

Le chiffre [1] devant chaque réponse signifie qu'il s'agit du premier élément de réponse renvoyé. Par la suite, lorsque nous manipulerons des données plus complexes, de nombreux résultats pourront être retournés, et dans ce cas cette numérotation sera utile.

Les règles de priorités mathématiques sont identiques sous R, et il est possible bien sûr d'utiliser des parenthèses.

Enfin, pour quitter la console R, on tapera :

```
> q()
```

3. Premiers objets

a) Objets simples

Comme tout langage de programmation, R permet d'utiliser des objets :

```
> x <- 2
```

L'opérateur `<-` est l'**opérateur d'assignation**.
Il prend une valeur quelconque à droite et la place dans l'objet indiqué à gauche.
La commande pourrait donc se lire : « mettre la valeur `2` dans l'objet nommé `x` ».

On va ensuite pouvoir réutiliser cet objet dans d'autres calculs, ou simplement afficher son contenu :

```
> x + 3
[1] 5
> x
[1] 2
```

On peut utiliser autant d'objets que l'on veut.
Ceux-ci peuvent contenir des nombres, des chaînes de caractères (indiquées par des guillemets doubles ") et bien d'autres choses encore :

```
> x <- 27
> y <- 10
> foo <- x + y
> foo
[1] 37
> x <- "Hello"
foo <- x
> foo
[1] "Hello"
```

Les noms d'objets doivent commencer par une lettre, et peuvent ensuite contenir des lettres, des chiffres, les symboles . et _ .
R fait la différence entre les majuscules et les minuscules, ce qui signifie que x et X sont deux objets différents. On évitera d'utiliser des caractères accentués dans les noms d'objets, et comme les espaces ne sont pas autorisés on pourra les remplacer par un point ou un tiret bas.

Enfin, signalons que certains noms courts sont utilisés par R pour son usage interne et doivent être évités. On citera notamment : c, q, t, C, D, I, diff, length, mean, pi, range, var.

Certains mots sont réservés et il est interdit de les utiliser comme nom d'objet.
Les mots réservés pour le système sont :

break, else, for, function, if, in, next, repeat, return, while,
TRUE, FALSE, Inf, NA, NaN, NULL,
NA_integer_, NA_real_, NA_complex_, NA_character_,
..., ..1, ..2, etc.

b) Vecteurs

Dans R, un vecteur est un **ensemble de données de même nature**, et peut se construire à l'aide de la fonction `c` (abréviation de « combine »).

On l'utilise en lui donnant une liste de données entre parenthèses, séparées par des virgules :

```
> tailles <- c(168, 192, 173, 174, 172, 167, 171, 185, 163, 170)
```

On a ainsi été créé un objet nommé `tailles` comprenant l'ensemble des données, que l'on peut afficher :

```
> tailles
[1] 168 192 173 174 172 167 171 185 163 170
```

Dans le cas où le vecteur serait beaucoup plus grand et comporterait par exemple 45 tailles, on aurait par exemple le résultat suivant :

```
> tailles
 [1] 144 168 179 175 182 188 167 152 163 145 176 155 156 164 167 155 157
[18] 185 155 169 124 178 182 195 151 185 159 156 184 172 156 160 183 148
[35] 182 126 177 159 143 161 180 169 159 185 160
```

Il y a bien une suite de 45 tailles, et on peut remarquer la présence de nombres entre crochets au début de chaque ligne (`[1]` , `[18]` et `[35]`).
Ces nombres entre crochets indiquent la position du premier élément de la ligne dans le vecteur. Ainsi, le `185` en début de deuxième ligne est le 18ème élément du vecteur, tandis que le `182` de la troisième ligne est à la 35ème position.

On peut appliquer des **opérations** arithmétiques simples entre **un vecteur et un nombre** :

```
> u <- c(-1, 0, 1, 2)
> u + 10
[1] 9 10 11 12
> > u^2
[1] 1 0 1 4
```

On peut aussi effectuer des **opérations entre plusieurs vecteurs** :

```
> v <- c(10, 20, 30, 40)
> u + v
[1] 9 20 31 42
> v - 2 * u
[1] 12 20 28 36
> u * v
[1] -10 0 30 80
```

On constate que toutes les opérations ont donc lieu « **terme à terme** ».

Attention : quand R effectue une opération avec **deux vecteurs de longueurs différentes**, il recopie le vecteur le plus court de manière à lui donner la même taille que le plus long, ce qui s'appelle la **règle de recyclage** (recycling rule).

```
> u <- c(1, 2)
> v <- c(4, 5, 6, 7, 8)
> u + v
[1] 5 7 7 9 9
```

Ainsi, `c(1,2) + c(4,5,6,7,8)` est équivalent à `c(1,2,1,2,1) + c(4,5,6,7,8)`

C'est d'ailleurs ce qui se passe dans le cas d'une opération entre un vecteur et un nombre :
`c(-1, 0, 1, 2) + 10` est équivalent à `c(-1, 0, 1, 2) + c(10,10,10,10)`

On a vu jusque-là des vecteurs composés de nombres, mais on peut tout à fait créer des vecteurs composés de chaînes de caractères :

```
> reponse <- c("Bac+2", "Bac+1", "Bac", "BEP", "CAP")
```

Enfin, notons que l'on peut accéder à un élément particulier du vecteur en faisant suivre le nom du vecteur de crochets contenant le numéro de l'élément désiré. Par exemple :

```
> reponse[2]
[1] "Bac+1"
```

Cette opération s'appelle l'**indexation** d'un vecteur.
Il s'agit ici de sa forme la plus simple, mais il en existe d'autres beaucoup plus complexes. L'indexation des vecteurs et des tableaux dans R est l'un des éléments particulièrement souples et puissants du langage.

En voici quelques exemples :

```
> reponse[1:3]
[1] "Bac+2" "Bac+1" "Bac"
> reponse[c(2, 3, 5)]
[1] "Bac+1" "Bac" "CAP"
> reponse[-1]
[1] "Bac+1" "Bac" "BEP" "CAP"
> reponse
"Bac+2" "Bac+1" "Bac" "BEP" "CAP"
```

La première instruction a généré le vecteur constitué des éléments d'index 1 à 3 de `reponse`.
La deuxième instruction a généré le vecteur constitué des éléments d'index 2, 3 et 5 de `reponse`.
La troisième instruction a généré le vecteur constitué des éléments de `reponse` privé de l'élément d'index 1.
La dernière instruction montre que toutes ces indexations n'ont pas modifié le vecteur `reponse`.

Bien sûr, tous ces critères se combinent et on peut stocker le résultat dans un nouvel objet :

```
> rep <- reponse[-c(2, 3, 5)]
> rep
[1] "Bac+2" "BEP"
```

On peut d'autre part utiliser l'**indexation combinée avec l'assignation pour modifier un vecteur** :

```
> reponse[6] <- "Brevet"
> reponse
"Bac+2" "Bac+1" "Bac" "BEP" "CAP" "Brevet"
> reponse[4] <- "BacPro"
> reponse
"Bac+2" "Bac+1" "Bac" "BacPro" "CAP" "Brevet"
> reponse[1:3] <- c("BAC+2", "BAC+1", "BAC")
> reponse
"BAC+2" "BAC+1" "BAC" "BacPro" "CAP" "Brevet"
```

La première instruction a ajouté un nouvel élément d'index 6 à `reponse`.
La troisième instruction a modifié l'élément d'index 4 de `reponse`.
La cinquième instruction a modifié les éléments d'index 1 à 3 de `reponse`.

On peut enfin effectuer une **indexation par condition**.

Une condition est une expression logique dont le résultat est soit `TRUE` (vrai),
soit `FALSE` (faux).
Une condition comprend la plupart du temps un opérateur de comparaison, dont les plus courants sont : `==` , `!=` , `>` , `<` , `>=` , `<=` ,

Application d'une condition sur un vecteur :

```
> reponse <- c("Bac+2", "Bac+1", "Bac", "BEP", "CAP")
> reponse == "Bac"
[1] FALSE FALSE TRUE FALSE FALSE
```

La condition `reponse == "Bac"` a donc renvoyé un **vecteur booléen** de même taille que `reponse`.

On peut combiner ou modifier des conditions avec les **opérateurs logiques** habituels
(& : et logique ; | : ou logique (inclusif) ; ! : négation logique) :

```
> tailles <- c(168, 192, 173, 174, 172, 167, 171, 185, 163, 170)
> !(tailles > 180 | tailles %% 2 == 0) & tailles < 170
[1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE
```

De telles conditions peuvent être utilisées pour l'indexation, selon le principe suivant :
si on indexe un vecteur avec un vecteur booléen, seuls les éléments correspondant à `TRUE` seront conservés :

```
> tailles[tailles >180]
[1] 192 185
> tailles[!(tailles > 180 | tailles %% 2 == 0) & tailles < 170]
[1] 167 163
```

4. Fonctions

Une fonction se caractérise de la manière suivante :
- elle a un nom, suivi de parenthèses ;
- elle peut accepter des arguments entre ces parenthèses, séparés par des virgules (ces arguments pouvant être nommés ou pas) ;
- elle retourne un résultat, ou peut effectuer une action comme dessiner un graphique, lire un fichier, etc.

Nous avons déjà utilisé quelques fonctions jusqu'ici, dont la fonction c.
Dans l'instruction suivante :

```
> reponse <- c("Bac+2", "Bac", "CAP", "Bac", "Bac", "CAP", "BEP")
```

on fait appel à la fonction `c`, on lui passe en arguments une série de chaînes de caractères, et elle retourne un vecteur de chaînes de caractères, que nous stockons dans l'objet `reponse`.

Autres exemples de fonctions courantes :

```
> tailles <- c(168, 192, 173, 174, 172, 167, 171, 185, 163, 170)
> length(tailles)
[1] 10
> mean(tailles)
[1] 173.5
```

La fonction `length` renvoie le nombre d'éléments d'un vecteur,
et la fonction `mean` renvoie la moyenne de ses éléments.

a) Arguments

En général les premiers arguments passés à la fonction sont des données servant au calcul, et les suivants des paramètres influant sur ce calcul. Ceux-ci sont en général transmis sous la forme d'**argument nommés**.

Reprenons l'exemple des tailles précédent, et imaginons que le deuxième enquêté n'ait pas voulu nous répondre. Nous avons alors dans notre vecteur une valeur manquante.
Celle-ci est symbolisée dans R par le code `NA` :

```
> tailles <- c(168, NA, 173, 174, 172, 167, 171, 185, 163, 170)
```

Recalculons notre taille moyenne :

```
> mean(tailles)
[1] NA
```

Par défaut, R renvoie `NA` pour un grand nombre de calculs lorsque les données comportent une valeur manquante. On peut cependant modifier ce comportement en fournissant un argument supplémentaire à la fonction `mean`, nommé `na.rm` :

```
> mean(tailles, na.rm = TRUE)
[1] 171.444
```

Positionner le paramètre `na.rm` à `TRUE` indique à la fonction `mean` de ne pas tenir compte des valeurs manquantes dans le calcul.
Lorsque l'on passe un argument à une fonction de cette manière,
c'est-à-dire sous la forme `nom = valeur`, on parle d'**argument nommé**.

b) Fonctions numériques

R fournit un grand nombre de fonctions numériques, notamment :

Fonction	Description
`abs`	retourne la valeur absolue du nombre en argument
`log`	retourne le logarithme népérien du nombre en argument
`exp`	retourne l'exponentielle (de base e) du nombre en argument
`sqrt`	retourne la racine carrée du nombre en argument
`round`	retourne une valeur approchée

Ces fonctions peuvent accepter des arguments nommés.
Par exemple :

```
> round(5.816)
[1] 6
> round(5.816, digits = 2)
[1] 5.82
```

c) Fonctions concernant les vecteurs

Les fonctions numériques précédentes peuvent être appliquées à des vecteurs numériques :

```
> abs(c(-2.34, 5.6, -8))
[1] 2.34 5.60 8.00
```

Et il existe bon nombre de fonctions spécifiques aux vecteurs :

Fonction	Description
c	retourne un vecteur à partir d'une série de valeurs en arguments
length	retourne le nombre d'éléments d'un vecteur en argument
mean	retourne la moyenne des éléments d'un vecteur numérique en argument
var	retourne la variance des éléments d'un vecteur numérique en argument
sd	retourne l'écart-type des éléments d'un vecteur numérique en argument
min	retourne la valeur minimale d'un vecteur en argument
max	retourne la valeur maximale d'un vecteur en argument
head	retourne un vecteur constitué des premiers éléments d'un vecteur en argument
tail	retourne un vecteur constitué des derniers éléments d'un vecteur en argument
sort	retourne un vecteur constitué des éléments triés d'un vecteur en argument
order	retourne un vecteur constitué des indices des éléments triés d'un vecteur en argument
%in%	opérateur permettant de savoir si une valeur fait partie des éléments d'un vecteur (retourne un booléen)
unique	retourne un vecteur constitué des éléments sans répétition d'un vecteur en argument
rep	retourne un vecteur constitué de répétitions de son argument
:	retourne un vecteur d'entiers consécutifs
summary	fournit des informations sur un vecteur en argument, selon sa nature

Fonctions `min`, `max`, `head`, `tail`, `sort`, `order` :

```
> u <- c ('a', 'z', 'e', 'r', 't', 'y', 'u', 'o')
> min(u)
[1] "a"
> max(u)
[1] "z"
> head(u)
[1] "a" "z" "e" "r" "t" "y"
> head(u, 3)
[1] "a" "z" "e"
> tail(u)
[1] "e" "r" "t" "y" "u" "o"
> tail(u, 3)
[1] "y" "u" "o"
> sort(u)
[1] "a" "e" "o" "r" "t" "u" "y" "z"
> order(u)
[1] 1 3 8 4 5 7 6 2
> sort(u, decreasing = TRUE)
[1] "z" "y" "u" "t" "r" "o" "e" "a"
> order(u, decreasing = TRUE)
[1] 2 6 7 5 4 8 3 1
```

Opérateur `%in%` :

```
> "cat" %in% c("cat", "dog", "horse")
[1] TRUE
```

Fonction `unique` :

```
> unique(c(3, 2, 3, 1, 4, 4, 2))
[1] 3 2 1 4
```

Fonction `rep` :

Si le seul argument est un vecteur, le vecteur retourné est identique au vecteur en argument :

```
> v <- c("cat", "dog", "horse")
> rep(v)
[1] "cat" "dog" "horse"
```

L'argument nommé `times` (qui vaut 1 par défaut) permet d'indiquer le nombre de répétitions du vecteur en argument :

```
> rep(v, times = 3)
[1] "cat" "dog" "horse" "cat" "dog" "horse" "cat" "dog" "horse"
```

L'argument nommé `each` (qui vaut 1 par défaut) permet de générer un vecteur dans lequel chaque élément du vecteur en argument est répété l'un après l'autre :

```
> rep(v, each = 2)
[1] "cat" "cat" "dog" "dog" "horse" "horse"
```

Tout ceci peut bien sûr être combiné :

```
> rep(v, times = 3, each = 2)
[1] "cat" "cat" "dog" "dog" "horse" "horse" "cat" "cat" "dog" "dog" "horse"
[12] "horse" "cat" "cat" "dog" "dog" "horse" "horse"
```

De plus, lorsque son argument est un objet simple, la fonction `rep` permet de créer un vecteur constitué d'éléments identiques :

```
> rep("toc", times = 3)
[1] "toc" "toc" "toc"
```

Fonction `:`

Cette fonction à la syntaxe particulière retourne un vecteur constitué d'entiers consécutifs :

```
> -2:3
[1] -2 -1 0 1 2 3
```

Et on peut appliquer toutes sortes d'opérations à ce vecteur :

```
> (-2:3)^2
[1] 4 1 0 1 4 9
```

Fonction `summary` :

```
summary(c("cat", "dog", "horse"))
   Length     Class      Mode
        3 character character
```

```
summary(0:15)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    3.75    7.50    7.50   11.25   15.00
```

d) Fonctions concernant les chaines de caractères (et les vecteurs de chaines) :

Fonction `nchar` :

La fonction `nchar` retourne le **nombre de caractères** d'une chaine en argument.

```
> sentence <- "the computer"
> nchar(sentence)
[1] 12
```

Lorsque son argument est un vecteur de chaines :

```
> nchar(c("cat", "dog", "horse"))
[1] 3 3 5
```

Fonction `grepl` :

La fonction `grepl` retourne un **booléen** indiquant **si une chaine est incluse** dans une chaine en argument.

```
> grepl('com', sentence)
[1] TRUE
> grepl('a', sentence)
[1] FALSE
```

Lorsque son deuxième argument est un vecteur de chaines :

```
> grepl("o", c("cat", "dog", "horse"))
[1] FALSE TRUE TRUE
```

Fonction `substr` :

La fonction `substr` retourne une **chaine extraite** d'une chaine en argument.

```
> substr(sentence, start = 5, stop = 9)
[1] "compu"
```

Lorsque son argument est un vecteur de chaines :

```
> substr(c("cat", "dog", "horse"), start = 1, stop = 3)
[1] "cat" "dog" "hor"
```

Fonction `paste` :

La fonction `paste` convertit ses arguments en chaines de caractères, et les **concatène** :

```
> paste(5, "cats")
[1] "5 cats"
```

Les chaines concaténées sont séparées par un espace, ce qui peut être modifié par l'argument nommé `sep` (égal à `" "` par défaut) :

```
> paste(5, "cats", sep = "-")
[1] "5-cats"
```

La fonction `paste0` est équivalente à la fonction `paste` avec `sep = ""` (concaténation directe) :

```
> paste0("ca", "ts")
[1] "cats"
```

Lorsque les arguments sont des vecteurs, leurs éléments sont concaténés terme à terme :

```
> paste0(1:6, c("st", "nd", "rd", rep("th", times = 3)))
[1] "1st" "2nd" "3rd" "4th" "5th" "6th"
```

```
> paste("a", c("cat", "dog", "horse"))
[1] "a cat" "a dog" "a horse"
```

(Dans ce dernier exemple, on a utilisé la règle de recyclage).

e) Fonction `print` :

La fonction `print` affiche une chaine en argument, avec notamment la possibilité de supprimer les quotes :

```
> print("Nice !", quote = FALSE)
[1] Nice !
```

La fonction `print` accepte aussi un objet numérique en argument, ainsi qu'un vecteur :

```
> print(1)
[1] 1
> print(1:3)
[1] 1 2 3
```

f) Fonctions concernant le type d'un objet.

La fonction `class` permet de déterminer le type d'un objet :

```
> class(-5)
[1] "numeric"
> class(3.2)
[1] "numeric"
> class("cat")
[1] "character"
> class(5 == 2)
[1] "logical"
```

Appliquée sur un vecteur, la fonction `class` indiquera le type de ses éléments :

```
> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185, 163, 170)
> class (tailles)
[1] "numeric"
```

Des résultats similaires peuvent être obtenus pas des fonctions retournant un booléen :

```
> is.numeric(-5)
[1] TRUE
> is.numeric(-2:3)
[1] TRUE
> is.character(-5)
[1] FALSE
> is.character("cat")
[1] TRUE
> is.character(c("cat", "dog", "horse"))
[1] TRUE
> is.logical(5 == 2)
[1] TRUE
```

Ce genre de fonctions existe pour tous les types de variable.

On peut de plus **imposer un type** à une variable :

```
> a1 <- as.numeric("42")
> a1
[1] 42
> a2 <- as.numeric(c("42", "-5", "0"))
> a2
[1] 42 -5 0
> b <- as.numeric("cat")
Warning message:
NAs introduits lors de la conversion automatique
> c1 <- as.character(7)
> c1
[1] "7"
> c2 <- as.character(-2:3)
> c2
[1] "-2" "-1" "0" "1" "2" "3"
> d <- as.integer(c(3.14, -5.6))
> d
[1] 3 -5
> e <- as.integer("3.14")
> e
[1] 3
> f <- as.logical(0)
> f
[1] FALSE
> g <- as.logical(7)
> g
[1] TRUE
> h <- as.logical("cat")
> h
[1] NA
```

L'instruction `as.numeric("42")` a permis de « convertir » une chaine de caractères en nombre.
L'instruction `as.numeric(c("42", "-5", "0"))` a permis de « convertir » un vecteur de chaines de caractères en un vecteur numérique
L'instruction `as.numeric("cat")` a tenté de « convertir » la chaine de caractères `"cat"` en nombre, mais cela a logiquement déclenché une message d'avertissement (voir ci-après).
L'instruction `as.character(7)` a permis de « convertir » un nombre en chaine de caractères.
L'instruction `as.character(-2:3)` a permis de « convertir » un vecteur numérique en un vecteur de chaines de caractères.
L'instruction `as.integer(c(3.14, -5.6))` a permis de « convertir » un vecteur contenant des nombres décimaux en un vecteur contenant la « partie entière » de ces décimaux.
(On remarquera qu'ici la notion de « partie entière » est différente de la notion mathématique de la partie entière).
L'instruction `as.integer("3.14")` a permis de prendre la « partie entière » d'un nombre décimal désigné par une chaine de caractères.

La fonction `as.logical` a un comportement spécial :

- si l'argument est un nombre, elle retournera `TRUE` sauf pour le nombre `0` ;
- si l'argument est la chaine de caractères `"F"` ou `"FALSE"`, elle retournera `FALSE` ;
- si l'argument est la chaine de caractères `"T"` ou `"TRUE"`, elle retournera `TRUE` ;
- si l'argument est toute autre chaine de caractères, elle retournera `NA`.

Le message d'avertissement apparu avec l'instruction `as.numeric("cat")` indique que la variable qu'on a tenté de définir n'a pas pu l'être comme on tentait de le faire, et que R lui a du coup assigné la valeur `NA` :

```
> b
[1] NA
> is.na(b)
[1] TRUE
```

Le type **`NULL`** est un autre type particulier, qui permet de créer une variable qui n'a pas de valeur :

```
> i <- as.null()
> i
[1] Null
> class(i)
[1] "NULL"
> is.null(i)
[1] TRUE
```

g) Aide sur une fonction

Il est très fréquent de ne plus se rappeler quels sont les paramètres d'une fonction ou le type de résultat qu'elle retourne.
Dans ce cas on peut très facilement accéder à l'aide décrivant une fonction particulière en tapant :

```
> help("fonction")
```

(en remplaçant `fonction` par le nom de la fonction)

Ou, de manière équivalente :

```
> ?fonction
```

Chacune de ces deux commandes affiche une page (en anglais) décrivant la fonction, ses paramètres, son résultat, le tout accompagné de diverses notes, références et exemples. Ces pages d'aide contiennent à peu près tout ce qu'on peut chercher à savoir, mais elles ne sont pas toujours d'une lecture aisée.

5. Instructions conditionnelles, boucles

Instructions if et else :

```
> i <- 1
> if (i == 1) print(i) else print("a")
[1] 1
> i <- 2
> if (i == 1) print(i) else print("a")
[1] "a"
```

On peut faire exécuter plusieurs instructions si celles-ci sont encadrées par des accolades. Exemple (en mode script) :

```
if (i == 1) {
      print(i)
      i <- i + 1
      print(i)} else {
            print("Non !")
            print("i != 1")}
```

Boucle while :

Exemple (en mode script) :

```
i <- 0
while (i < 10) {
      print(i)
      i <- i + 1
}
```

Boucle for :

```
> for (x in 1:3) print(1:x)
[1] 1
[1] 1 2
[1] 1 2 3
```

Autre exemple :

```
> x <- c("a", "b", "ab")
> y <- 1:length(x)
> for (i in y) if (x[i] == "b") y[i] <- 0 else y[i] <- 1
> y
[1] 1 0 1
```

Et là encore, on peut faire exécuter plusieurs instructions si celles-ci sont encadrées par des accolades :

```
for (i in y) {
    ...
}
```

II. IMPORTATION ET EXPORTATION DE DONNEES

La principale vocation de R étant de traiter des jeux de données pour y appliquer des tests statistiques, l'importation et l'exportation de fichiers sont des sujets essentiels.
L'import et l'export de données depuis ou vers d'autres applications, est couvert en détail dans l'un des manuels officiels (en anglais) nommé R Data Import/Export et accessible, comme les autres manuels, à l'adresse suivante : http://cran.r-project.org/manuals.html
Ne seront exposés ici que quelques cas précis.

1. Accès aux fichiers et répertoire de travail

La fonction `read.table`, très utilisée pour l'import de fichiers texte, prend comme premier argument le nom du fichier à importer.
Exemple d'instruction :

```
> donnees <- read.table("fichier.txt")
```

Cependant, cette instruction ne fonctionnera que si le fichier `fichier.txt` se trouve dans le **répertoire de travail** de R, à savoir le répertoire dans lequel R est actuellement en train de s'exécuter.

Pour savoir quel est le répertoire de travail actuel de R, on peut utiliser la fonction `getwd` :

```
> getwd()
[1] "C:/Users/jcduc/OneDrive/Documents/DatasetsR"
```
(exemple sous Windows)

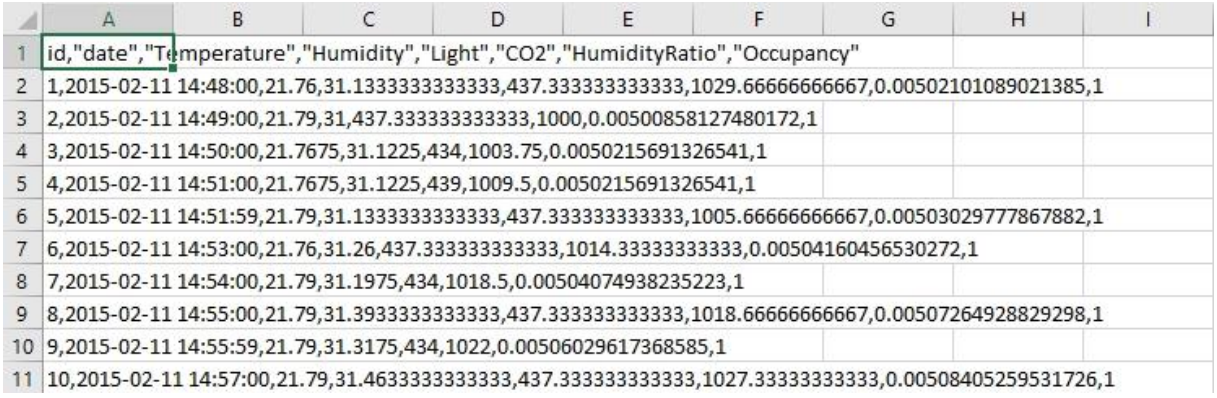
Si on veut modifier le répertoire de travail, on utilise la fonction `setwd`, en lui donnant comme argument la chaine de caractère exprimant le chemin complet vers le répertoire dans lequel on veut que R travaille.

Une fois le répertoire de travail fixé, on pourra accéder directement aux fichiers qui s'y trouvent, en spécifiant seulement leur nom (avec l'extension .r).

2. Import de données depuis un tableur

Il est assez courant de vouloir importer des données saisies ou traitées avec un tableur du type OpenOffice ou Excel.

Une fois convertis en fichiers .csv, ceux-ci peuvent se présenter ainsi :



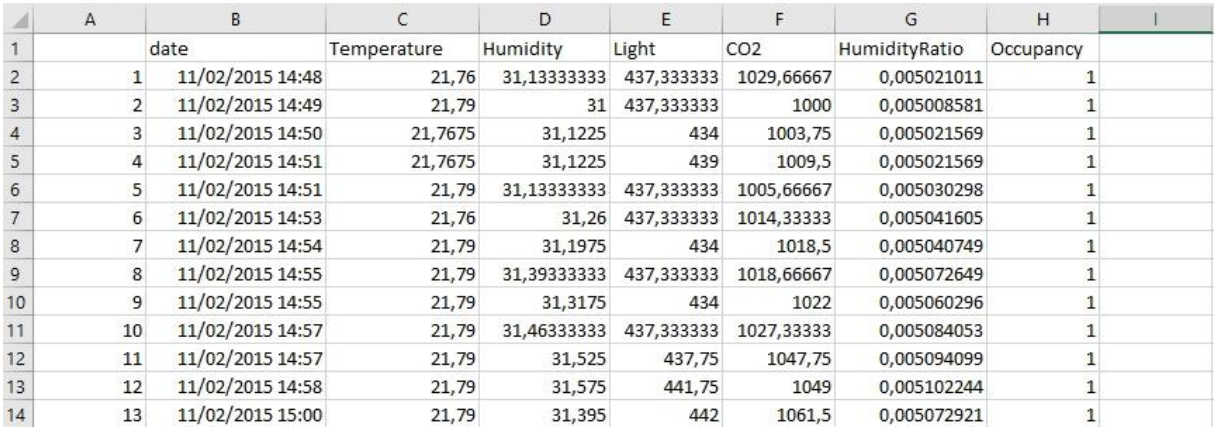
	A	B	C	D	E	F	G	H	I
1	id,"date","Temperature","Humidity","Light","CO2","HumidityRatio","Occupancy"								
2	1,2015-02-11 14:48:00,21.76,31.1333333333333,437.333333333333,1029.66666666667,0.00502101089021385,1								
3	2,2015-02-11 14:49:00,21.79,31,437.333333333333,1000,0.00500858127480172,1								
4	3,2015-02-11 14:50:00,21.7675,31.1225,434,1003.75,0.0050215691326541,1								
5	4,2015-02-11 14:51:00,21.7675,31.1225,439,1009.5,0.0050215691326541,1								
6	5,2015-02-11 14:51:59,21.79,31.1333333333333,437.333333333333,1005.66666666667,0.00503029777867882,1								
7	6,2015-02-11 14:53:00,21.76,31.26,437.333333333333,1014.33333333333,0.00504160456530272,1								
8	7,2015-02-11 14:54:00,21.79,31.1975,434,1018.5,0.00504074938235223,1								
9	8,2015-02-11 14:55:00,21.79,31.3933333333333,437.333333333333,1018.66666666667,0.00507264928829298,1								
10	9,2015-02-11 14:55:59,21.79,31.3175,434,1022,0.00506029617368585,1								
11	10,2015-02-11 14:57:00,21.79,31.4633333333333,437.333333333333,1027.33333333333,0.00508405259531726,1								

auquel cas on pourra les charger avec une instruction du type :

```
donnees <- read.csv("fichier.csv")
```

Cette instruction a permis de créer un objet `donnees`, qui est un tableau qui contient les données du fichier.

S'ils se présentent sous la forme suivante :



	A	B	C	D	E	F	G	H	I
1		date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy	
2	1	11/02/2015 14:48	21,76	31,13333333	437,333333	1029,66667	0,005021011	1	
3	2	11/02/2015 14:49	21,79	31	437,333333	1000	0,005008581	1	
4	3	11/02/2015 14:50	21,7675	31,1225	434	1003,75	0,005021569	1	
5	4	11/02/2015 14:51	21,7675	31,1225	439	1009,5	0,005021569	1	
6	5	11/02/2015 14:51	21,79	31,13333333	437,333333	1005,66667	0,005030298	1	
7	6	11/02/2015 14:53	21,76	31,26	437,333333	1014,33333	0,005041605	1	
8	7	11/02/2015 14:54	21,79	31,1975	434	1018,5	0,005040749	1	
9	8	11/02/2015 14:55	21,79	31,39333333	437,333333	1018,66667	0,005072649	1	
10	9	11/02/2015 14:55	21,79	31,3175	434	1022	0,005060296	1	
11	10	11/02/2015 14:57	21,79	31,46333333	437,333333	1027,33333	0,005084053	1	
12	11	11/02/2015 14:57	21,79	31,525	437,75	1047,75	0,005094099	1	
13	12	11/02/2015 14:58	21,79	31,575	441,75	1049	0,005102244	1	
14	13	11/02/2015 15:00	21,79	31,395	442	1061,5	0,005072921	1	

on pourra les charger avec une instruction du type :

```
donnees <- read.csv2("fichier.csv")
```

Ces fonctions sont en fait des dérivées de la fonction générique `read.table`.
Celle-ci contient de nombreuses options permettant d'adapter l'import au format du fichier.
On pourra se reporter à la page d'aide de `read.table` si on rencontre des problèmes ou si on souhaite importer des fichiers d'autres sources.

3. Export de données

Export de tableaux

Les données importées depuis des tableurs génèrent des tableaux sous R, qui vont être manipulés et transformés comme on le souhaite.
Pour exporter un tableau, `write.table` est l'équivalent de `read.table` et permet d'enregistrer des tableaux de données au format texte, avec de nombreuses options.

Pour exporter au format csv, il y a

```
write.csv(tableau, file = "fichier.csv")
```

ou

```
write.csv2(tableau, file = "fichier.csv")
```

où `tableau` désigne le nom sous R du tableau que l'on veut exporter, et `fichier` le nom du fichier qui va le stocker dans notre PC (dans le répertoire de travail de R).

Export de graphiques

Exporter les graphiques que nous créons sous R est très simple : le menu Fichier de la console R propose en effet des options de sauvegardes simples et multiples.

III. MANIPULATION DE DONNEES

Prenons l'exemple du chargement d'un fichier `bank2.csv` qui contient des informations sur les clients d'une banque :

```
data <- read.csv2("bank2.csv")
```

1. Inspection des données

Pour vérifier que l'importation s'est bien déroulée, on peut utiliser les fonctions `nrow`, `ncol` et `dim`, qui donnent respectivement le nombre de lignes, le nombre de colonnes et les dimensions du tableau créé :

```
> nrow(data)
[1] 45211
> ncol(data)
[1] 17
> dim(data)
[1] 45211    17
```

La fonction `names` fournit un vecteur contenant les noms des colonnes du tableau, c'est-à-dire les noms des variables :

```
> names(data)
 [1] "age"       "job"       "marital"   "education" "default"   "balance"
 [7] "housing"   "loan"      "contact"   "day"       "month"     "duration"
[13] "campaign"  "pdays"     "previous"  "poutcome"  "y"
```

La fonction `str` est plus complète : elle liste les différentes variables, indique leur type, et donne le cas échéant des informations supplémentaires ainsi qu'un échantillon des premières valeurs prises par cette variable :

```
> str(data)
'data.frame':   45211 obs. of  17 variables:
 $ age       : int  58 44 33 47 33 35 28 42 58 43 ...
 $ job       : Factor w/ 12 levels "admin.","blue-collar",..: 5 10 3 2 12 5 5 3 6 ...
 $ marital   : Factor w/ 3 levels "divorced","married",..: 2 3 2 2 3 2 3 1 2 3 ...
 $ education: Factor w/ 4 levels "primary","secondary",..: 3 2 2 4 4 3 3 3 1 2 ...
 $ default   : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 2 1 1 ...
 $ balance   : int  2143 29 2 1506 1 231 447 2 121 593 ...
 $ housing   : Factor w/ 2 levels "no","yes": 2 2 2 2 1 2 2 2 2 2 ...
 $ loan      : Factor w/ 2 levels "no","yes": 1 1 2 1 1 1 2 1 1 1 ...
 $ contact   : Factor w/ 3 levels "cellular","telephone",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ day       : int  5 5 5 5 5 5 5 5 5 5 ...
 $ month     : Factor w/ 12 levels "apr","aug","dec",..: 9 9 9 9 9 9 9 9 9 9 ...
 $ duration  : int  261 151 76 92 198 139 217 380 50 55 ...
 $ campaign  : int  1 1 1 1 1 1 1 1 1 1 ...
 $ pdays     : int  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ...
 $ previous  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ poutcome  : Factor w/ 4 levels "failure","other",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ y         : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
```

La première ligne nous informe qu'il s'agit bien d'un tableau de données, avec 45211 observations et 17 variables. Vient ensuite la liste des variables.
La première se nomme `age` et est de type nombre entier (`int`).
La cinquième se nomme `default` et il s'agit d'un **facteur** (`Factor w/`).
Un facteur et une variable pouvant prendre un nombre limité de modalités (`levels`).
Ici notre variable a deux modalités possibles : `no` et `yes`.
Ce type de variable, très fréquent, correspond à la plupart des variables issues d'une question fermée dans un questionnaire.

Attention : les lignes affichées par `str` sont les colonnes du tableau !

Pour accéder aux variables, c'est-à-dire aux colonnes du tableau, on peut utiliser :
le nom du tableau, suivi de l'opérateur `$`, suivi du nom de la variable.

Par exemple :

```
> data$age
[1] 58 44 33 47 33 35 28 42 58 43 41 29 53 58 57 51 45 57 60 33 28 56 32 25 40
[26] 44 39 52 46 36 57 49 60 59 51 57 25 53 36 37 44 NA 60 54 58 36 58 44 55 29
[51] 54 48 32 42 24 NA 38 47 40 46 32 53 57 33 49 51 60 59 55 35 57 31 54 55 NA
[76] 53 44 55 49 55 45 47 42 59 46 51 56 41 46 57 42 30 60 60 57 36 55 60 39 46
[101] 44 53 52 59 27 44 47 34 59 45 29 46 56 36 59 44 41 33 59 57 56 51 34 43 52
```

etc (la totalité de la colonne `age` s'est affichée).

Les fonctions `head` et `tail` permettent d'afficher seulement les premières (respectivement les dernières) valeurs prises par la variable :

```
> head(data$marital)
 [1] married single  married married single  married
Levels: divorced married single
```

Nous avons les premières valeurs prises par la variable `marital`, ainsi que les `Levels` puisque `marital` est un facteur.

`data$age` et `data$marital` sont des **vecteurs**, de nature différente :

```
> class(data$age)
[1] "integer"
> class(data$marital)
[1] "factor"
```

On peut en obtenir un aperçu grâce à la fonction `summary` :

```
> summary(data$marital)
divorced  married   single
    5207    27214    12790
```

On a ici les effectifs des différents `Levels`.

```
> summary(data$age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  18.00   33.00   39.00   40.94   48.00   95.00       4
```

`data$age` étant un vecteur numérique, `summary` nous a en fourni les valeurs min et max, les quartiles et la moyenne.
On remarque de plus la présence de 4 `NA`, ce qui signifie 4 données manquantes.

On peut récupérer chacune de ces valeurs, comme par exemple la moyenne
de la variable `data$age` :

```
> summary(data$age)[[4]]
[1] 40.93638
```

On remarquera les doubles crochets dans cette instruction.

Enfin, on peut tracer des histogrammes (fonction `hist`),
des boîtes à moustaches (fonction `boxplot`)…

2. Indexation

L'accès aux variables tel qu'on l'a fait précédemment correspond à l'**indexation par nom**. C'est un mode d'indexation très simple, mais limité, notamment dans le cas où on veut concevoir un script (voir paragraphe V) agissant sur des colonnes dont on ne connait pas le nom à priori.

On peut agir sur les variables et leur contenu en utilisant l'**indexation directe**.
On a déjà vu l'indexation de vecteurs.
Dans le cas d'un tableau, qui est un objet à 2 dimensions, l'indexation prend deux arguments séparés par une virgule : le premier concerne les lignes et le second les colonnes.
Ainsi, si on veut l'élément correspondant à la troisième ligne et à la douzième colonne du tableau de données `data` :

```
> data[3,12]
[1] 76
```

On peut également indiquer des vecteurs :

```
> data[1:3,1:2]
  age          job
1  58   management
2  44   technician
3  33 entrepreneur
```

Si on laisse l'un des deux critères vide, on sélectionne l'intégralité des lignes ou des colonnes.

Ainsi si l'on veut sélectionner la deuxième colonne du tableau `data` :
`data[,2]` (identique à `data$job`)

Si on veut sélectionner les deux premières lignes du tableau `data` :
`data[1:2,]`

Indexation et assignation :

L'indexation (qui permet de sélectionner un ou plusieurs éléments, ou bien tout ou partie d'une ligne ou d'une colonne d'un tableau) peut être combinée avec l'assignation pour modifier le tableau :

- L'instruction : `data[1:3,1] <- 50` ou son équivalent : `data[1:3,"age"] <- 50` modifie l'âge des trois premiers individus du tableau `data`.
- L'instruction : `data[,1] <- 40` ou son équivalent : `data$age <- 40` modifie l'âge de tous les individus du tableau `data`.
- L'instruction : `data[,18] <- data[,1]` créé une 18ème colonne dans `data`, constituée des valeurs de sa 1ère colonne.

L'utilisation de l'indexation et de l'assignation peut aussi permettre de créer un nouveau tableau.
Par exemple, l'instruction : `data2 <- data[1:3, -c(2, 6, 8)]`
créé un nouveau tableau `data2` constitué des trois premières lignes de `data`
sans les colonnes 2, 6 et 8 .

Indexation par condition :

Conformément à ce qu'on a vu au paragraphe I.3.b), l'application d'une condition sur un vecteur renvoie un vecteur booléen :

```
> head(data$age)
[1] 58 44 33 47 33 35
```

```
>head(data$age > 40)
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

Ce qui est identique à :

```
>head(data[,1] > 40)
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

On a aussi vu que l'indexation d'un vecteur par condition permet de sélectionner des éléments du vecteur.
Rappel de son principe de fonctionnement : si on indexe un vecteur avec un vecteur booléen, seuls les éléments correspondant à `TRUE` seront conservés.

Ainsi, je peux par exemple annuler toutes les valeurs de la variable `duration` qui sont inférieures à 100 par l'instruction :

```
data$duration[data$duration < 100] <- 0
```

Analysons à présent l'instruction :

```
> data2 <- data[data$age > 40 & data$job == "management", ]
```

La virgule avant le dernier crochet nous montre que la condition posée concerne les lignes du tableau `data`, et il va s'agir de ne conserver que celles qui contiennent des individus qui sont des manageurs de plus de 40 ans.
L'instruction a donc créé donc un nouveau tableau `data2` extrait de `data`, ne contenant que les manageurs de plus de 40 ans.

On observe ci-dessous que ce sont bien des lignes qui ont été sélectionnées, et les 17 colonnes de `data` sont bien présentes dans `data2`, avec beaucoup moins de lignes (`obs.`) :

```
> str(data2)
'data.frame':   4089 obs. of  17 variables:
 $ age      : int  58 56 46 49 51 NA 48 51 59 54 ...
 $ job      : Factor w/ 12 levels "admin.","blue-collar",..: 5 5 5 5 5 NA 5 5 5 ...
 $ marital  : Factor w/ 3 levels "divorced","married",..: 2 2 3 2 2 NA 1 2 1 2 ...
 $ education: Factor w/ 4 levels "primary","secondary",..: 3 3 2 3 3 NA 3 3 3 2 ...
 $ default  : Factor w/ 2 levels "no","yes": 1 1 1 1 1 NA 1 1 1 1 ...
 $ balance  : int  2143 779 -246 378 10635 NA -244 6530 59 282 ...
 $ housing  : Factor w/ 2 levels "no","yes": 2 2 2 2 2 NA 2 2 2 2 ...
 $ loan     : Factor w/ 2 levels "no","yes": 1 1 1 1 1 NA 1 1 1 2 ...
 $ contact  : Factor w/ 3 levels "cellular","telephone",..: 3 3 3 3 3 NA 3 3 3 ...
 $ day      : int  5 5 5 5 5 NA 5 5 5 5 ...
 $ month    : Factor w/ 12 levels "apr","aug","dec",..: 9 9 9 9 9 NA 9 9 9 9 ...
 $ duration : int  261 164 255 230 336 NA 253 91 273 154 ...
 $ campaign : int  1 1 2 1 1 NA 1 1 1 1 ...
 $ pdays    : int  -1 -1 -1 -1 -1 NA -1 -1 -1 -1 ...
 $ previous : int  0 0 0 0 0 NA 0 0 0 0 ...
 $ poutcome : Factor w/ 4 levels "failure","other",..: 4 4 4 4 4 NA 4 4 4 4 ...
 $ y        : Factor w/ 2 levels "no","yes": 1 1 1 1 1 NA 1 1 1 1 ...
```

3. Création de sous-populations

a) Par indexation

On a vu précédemment qu'on pouvait sélectionner une partie des données d'un tableau par l'indexation, et les assigner à un nouveau tableau qui constitue donc une sous-population du tableau original.

Cependant, un problème jusque-là ignoré peut provenir de la présence de valeurs manquantes (`NA`), car celles-ci seront toujours sélectionnées lors d'une indexation par condition :

```
> v <- c(1:5, NA)
> v
[1] 1  2  3  4  5 NA
> v2 <- v[v <= 2]
> v2
[1] 1  2 NA
```

Cela peut être évité ainsi :

```
> v3 <- v[v <= 2 & !is.na(v)]
> v3
[1] 1 2
```

Le problème se posera donc de même lors de la création de tableaux extraits via l'indexation par condition.

Reprenons l'instruction :

```
> data2 <- data[data$age > 40 & data$job == "management", ]
```

qui sélectionne les manageurs de plus de 40 ans, et inspectons le contenu de `data2` :

```
> summary(data2$age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   41.0    44.0    49.0    49.6    54.0    81.0       1
```

Il y a donc encore au moins 1 `NA` dans `data2`, qui ne correspond donc pas à la sous-population que je voulais créer.

On peut bien sûr l'éviter comme on l'a vu précédemment :

```
> data3 <- data[data$age > 40 & data$job == "management"
+ & !is.na(data$age > 40 & data$job == "management"), ]
> summary(data3$age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   41.0    44.0    49.0    49.6    54.0    81.0
```

Mais cela donne une instruction bien lourde, et c'est notamment pour cette raison que l'on préférera le plus souvent utiliser la fonction `subset`.

b) Fonction subset

La fonction `subset` permet d'extraire des sous-populations de manière plus simple et un peu plus intuitive que par l'indexation.

Celle-ci prend deux arguments principaux :
- le nom de l'objet de départ ;
- une condition sur les observations (lignes).

Pour sélectionner les manageurs de plus de 40 ans :

```
> data4 <- subset(data, age > 40 & job == "management")
```

Cette instruction créé le même tableau que la lourde instruction précédente, en écartant les valeurs manquantes (`NA`) : `data4` est identique à `data3`.
Dans cette instruction, on remarquera de plus la façon simplifiée de nommer les lignes sur lesquelles on impose une condition.

De plus, l'utilisation de l'argument `select` permet d'exprimer de manière simple une condition sur les colonnes.

Par exemple :

```
> data5 <- subset(data, age > 40, select = c(age, job))
```

créé un tableau à deux variables (`age` et `job`) constitué uniquement de personnes de plus de 40 ans (sans `NA`).

```
> data6 <- subset(data, age > 40, select = -c(4:17))
```

créé un tableau ne gardant que les 3 premières colonnes de `data`, constitué uniquement de personnes de plus de 40 ans (sans `NA`).

4. Tri

Pour trier les données d'un tableau selon l'ordre croissant ou décroissant d'une variable (variable numérique ou chaines de caractères), on utilise la fonction `order`, déjà évoquée à propos des vecteurs :

```
> order(data$age)
[1]  40737 40745 40888 41223 41253 41274 41488 42147 42275 42955 43638
[12] 44645 30792 31042 31264 31305 31433 31493 32170 33775 33790 34282
[23] 34289 34676 40377 40565 40928 41058 41403 41447 ...
```

etc.
Nous avons ici les indices des éléments de `data$age` par ordre croissant.
Le plus jeune individu se trouve donc sur la ligne `40737`.

Ainsi, pour obtenir un tableau trié par ordre d'âge croissant :

```
> datatri <- data[order(data$age), ]
```

On peut bien sûr trier par ordre décroissant en utilisant l'option `decreasing = TRUE`.

IV. STATISTIQUE BIVARIEE

La statistique bivariée décrit l'étude des relations entre deux variables.
Nous nous limiterons ici à l'étude de l'ajustement de deux variables quantitatives, au travers d'un exemple fourni par le fichier : `budget_pub.csv`
Celui-ci contient un tableau exprimant, pour chaque mois d'une année, un budget publicitaire et un chiffre d'affaire, et nous allons étudier la corrélation entre ces deux variables.

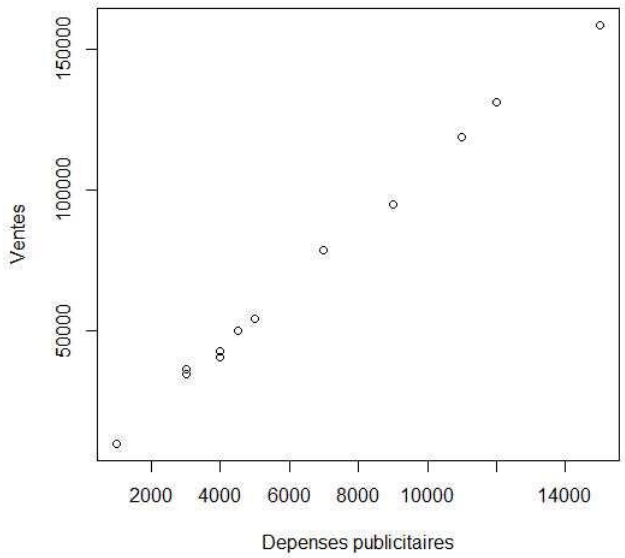
On ouvre le fichier dans R et on l'assigne à l'objet `budg` :

```
> budg <- read.csv2("budget_pub.csv")
> str(budg)
'data.frame':   12 obs. of  3 variables:
 $ Mois        : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Depenses.pub: int  1000 4000 5000 4500 3000 4000 9000 11000 15000 ...
 $ Ventes      : int  9914 40487 54324 50044 34719 42551 94871 118914 ...
```

L'éventuelle corrélation entre les ventes et les dépenses publicitaires peut être visualisée par un aperçu graphique, qui peut être obtenu par l'instruction suivante :

```
> plot(budg$Depenses.pub, budg$Ventes, xlab = "Depenses publicitaires", ylab = "Ventes")
```

qui affiche le nuage de points :



On constate ici que les points sont « très alignés », ce qui suggère une forte corrélation linéaire, ce que va confirmer le calcul du **coefficient de corrélation linéaire** entre ces deux variables :

```
> cor(budg$Depenses.pub, budg$Ventes, use = "complete.obs")
[1] 0.9988322
```

Plus la valeur absolue de ce coefficient est proche de 1 (ce qui est nettement le cas dans notre exemple), meilleure est la corrélation linéaire entre les deux variables, et d'autant plus significatif sera un ajustement affine entre elles.
Le signe de ce coefficient dépend du sens de variation des variables : il est positif lorsque les deux variables varient dans le même sens (ce qui est le cas dans notre exemple), et négatif dans le cas contraire.
Dans notre exemple, l'excellente valeur du coefficient de corrélation linéaire nous amène à effectuer un ajustement affine entre elles.

Pour chercher la fonction affine qui exprime au mieux le comportement des ventes en fonction des dépenses publicitaires (fonction obtenue par la méthode « des moindres carrés »), on effectue une régression linéaire à l'aide de la fonction `lm` :

```
> reg <- lm(Ventes ~ Depenses.pub, data = budg)
```

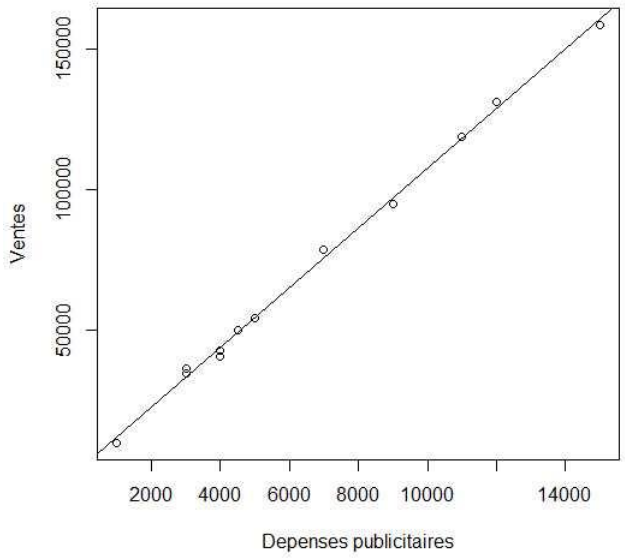
On affiche `reg` :

```
> reg

Call:
lm(formula = Ventes ~ Depenses.pub, data = budg)

Coefficients:
 (Intercept)  Depenses.pub
     1383.47         10.62
```

Les deux derniers nombres affichés sont les coefficients de la fonction affine qui « ajuste » au mieux les deux variables.
On a donc ici : Ventes = 1383,47 + 10,62 × Dépenses.pub

Un tracé de la droite de régression peut être obtenu par l'instruction :

```
> abline(reg)
>
```



La relation affine exprimant les ventes en fonction des dépenses publicitaires peut être utilisée pour faire des **extrapolations**, et prévoir par exemple les ventes pour un budget publicitaire de 17000€ :
$1383{,}47 + 10{,}62 \times 17000 = 181923{,}47$
On peut donc s'attendre à des ventes de l'ordre de 182000€.
La **fiabilité** de cette prévision est conditionnée par la fiabilité de la régression linéaire effectuée, celle-ci étant exprimée par la valeur du coefficient de corrélation linéaire (qui est excellente dans notre exemple).
De plus, la fiabilité d'une telle extrapolation diminuera fortement au fur et à mesure que l'on s'éloigne l'intervalle correspondant à l'étendue de la variable Dépenses.pub.

Il est possible d'effectuer de tels calculs dans R, en récupérant indépendamment les deux coefficients (de manière plus précise) :

```
> reg$coefficients[[1]]
[1] 1383.471
> reg$coefficients[[2]]
[1] 10.6222
```

Remarque : si on ne dispose pas du nom des variables, on peut utiliser l'indexation directe. Les instructions :

```
> plot(budg[,2], budg[,3], xlab = names(budg[2]), ylab = names(budg[2]))
> cor(budg[,2], budg[,3], use = "complete.obs")
> reg <- lm(budg[,3] ~ budg[,2], data = budg)
```

pour l'affichage du nuage de points, le calcul du coefficient de corrélation linéaire et de la régression sont identiques aux instructions précédentes.

Lorsqu'on dispose du nom des variables, et que la régression a été calculée à partir de ces noms, on peut utiliser la fonction `predict` :

```
> predict(reg, data.frame(Depenses.pub = 17000))
       1
181960.8
```

qui nous indique donc une prévision de ventes (plus précise) de 181960.80€

Cette valeur est bien sûr récupérable :

```
> prev <- predict(reg, data.frame(Depenses.pub = 17000))
> prev[[1]]
[1] 181960.8
```

On peut de plus obtenir un intervalle dans lequel la prévision se trouve :

```
> predict(reg, data.frame(Depenses.pub = 17000), interval = "prediction")
       fit      lwr      upr
1 181960.8 175394.4 188527.2
```

La valeur `fit` est la moyenne entre les valeurs `lwr` et `upr`, et la largeur de l'intervalle entre `lwr` et `upr` caractérise la précision de la prévision.

Enfin, des ajustements non linéaires peuvent être effectués par l'intermédiaire de changements de variables (voir exercice 4).

V. UTILISATION D'UN EDITEUR DE TEXTE

Jusqu'à maintenant nous avons utilisé uniquement la console pour communiquer avec R via l'invite de commandes. Le principal problème de ce mode d'interaction est qu'une fois qu'une commande est tapée, elle est pour ainsi dire « perdue », c'est-à-dire que l'on doit la saisir à nouveau si on veut l'exécuter une seconde fois. L'utilisation de la console est donc restreinte aux petites commandes « jetables », le plus souvent utilisées comme test.

Pour écrire des programmes plus complexes, les instructions seront écrites dans un éditeur de texte et seront stockées dans un fichier d'extension .r
On appelle en général ce type de fichier un script.

De plus, l'écriture d'un programme R dans un éditeur de texte permet d'inclure des commentaires dans le script (chaque ligne de commentaire commençant par le symbole #).

L'ouverture d'un script dans la console R s'obtient via le menu Fichier, et son exécution via le menu Edition.

EXERCICES

Exercice 1 (Manipulation de vecteurs numériques) :

Proposer du code R pour chacune des demandes suivantes :

a) Initialiser un vecteur v de booléens contenant `TRUE` ou `FALSE` selon que la valeur correspondante d'un vecteur numérique u quelconque est strictement positive ou non.
Par exemple, si u est : `-54  12  0  35` , alors v sera : `FALSE  TRUE  FALSE  TRUE`

b) Initialiser un vecteur v contenant les valeurs strictement positives d'un vecteur numérique u quelconque.
Par exemple, si u est : `-54  12  0  35` , alors v sera : `12  35`

c) Initialiser un vecteur v contenant 3 répétitions d'un vecteur numérique u quelconque sans la dernière valeur.
Par exemple, si u est : `54  12  35` , alors v sera : `54  12  35  54  12  35  54  12`

d) Initialiser un vecteur v contenant les indices des valeurs strictement positives d'un vecteur numérique u quelconque.
Par exemple, si u est : `-54  12  0  35` , alors v sera : `2  4`

e) A l'aide d'une indexation par condition, initialiser un vecteur v contenant les 10 premiers entiers naturels impairs.

f) Initialiser un vecteur v contenant les prix TTC correspondant aux prix HT contenus dans un vecteur numérique u quelconque (avec un taux de TVA de 20%).
Par exemple, si u est : `54.99  12.41  35.14` , alors v sera : `65.99  14.89  42.17`

g) Initialiser un vecteur v contenant un vecteur numérique u quelconque incrémenté à 100, 200, 300.
Par exemple, si u est : `54  12  35`
alors v sera : `54  12  35  154  112  135  254  212  235  354  312  335`

h) Initialiser un vecteur v qui va remplacer le plus grand élément d'un vecteur numérique u quelconque par son double.
Par exemple, si u est : `-54  100  35` , alors v sera : `-54  200  35`

Exercice 2 (Manipulation de vecteurs de chaines) :

Proposer du code R pour chacune des demandes suivantes :

a) Initialiser un vecteur v contenant les adresses mail d'un vecteur u quelconque de chaines (une adresse mail étant détectée par la présence de @).
Par exemple, si u est :
`"Hello" "untel@gmail.com" "next" "comfortable" "me@mycompany.com"`
alors v sera : `"untel@gmail.com" "me@mycompany.com"`

b) Initialiser un vecteur v contenant les chaines de plus de 10 caractères d'un vecteur u quelconque de chaines.
Par exemple, si u est :
`"Hello" "untel@gmail.com" "next" "comfortable" "me@mycompany.com"`
alors v sera : `"untel@gmail.com" "comfortable" "me@mycompany.com"`

c) Initialiser un vecteur v contenant les versions tronquées des chaines de plus de 10 caractères d'un vecteur u quelconque de chaines : on ne gardera que les 10 premiers caractères de ces chaines et on leur ajoutera "...".
Par exemple, si u est :
`"Hello" "untel@gmail.com" "next" "comfortable" "me@mycompany.com"`
alors v sera : `"untel@gmai..." "comfortabl…" "me@mycompa..."`

Exercice 3 :

Proposer du code R initialisant un objet s égal à la somme des éléments d'un vecteur u quelconque de « chaines numériques ».
Par exemple, si u est : `"3.1" "-5" "6"` , alors s sera : `4.1`

Exercice 4 :

Le fichier moteurs.csv exprime des valeurs pour deux variables : des nombres de moteurs fabriqués et des temps moyens de fabrication de ces moteurs. On y voit que le temps moyen de fabrication diminue au fur et à mesure que le nombre de moteurs fabriqués augmente, et on va chercher à ajuster ces deux variables.

1. Charger ce fichier dans R, renommer les variables (on les nommera `motfab` et `temps` afin d'alléger leur manipulation), vérifier le bon chargement du fichier et le renommage avec la fonction str, calculer le coefficient de corrélation linéaire des deux variables et afficher le nuage de points (`motfab` en abscisses et `temps` en ordonnées).

2. Créer deux nouvelles variables dans le tableau (nommées `lnmotfab` et `lntemps`) qui expriment les logarithmes népériens des valeurs des variables `motfab` et `temps`, calculer tous les coefficients de corrélation linéaire reliant les deux variables avec ou sans logarithme (il y en 4 en tout), afficher le nuage de points (avec la droite de régression) correspondant au couple de variables ayant le meilleur coefficients de corrélation linéaire, et effectuer la meilleure prévision de temps moyen de fabrication des moteurs lorsque 200 d'entre eux seront fabriqués (afficher ce résultat par une phrase).