1 Prise en main de R

Le logiciel R est un logiciel d'étude statistique. Il est très utilisé dans le monde académique mais il connaît aussi un développement important dans l'industrie, la banque, etc. Il est à la fois un langage informatique et un environnement de travail. Ce logiciel sert à manipuler des données, à tracer des graphiques et à faire des analyses statistiques sur ces données. Il s'agit d'un logiciel *opensource* et multi-plates-formes qui fonctionne sous UNIX (et Linux), Windows et Macintosh. Il est distribué gratuitement et vous pouvez le télécharger et installer sur une machine personnelle.

Cette initiation à R s'appuie sur les livres *Le logiciel R* de Lafaye de Micheaux et al. et *Statistiques avec R* de Cornillon *et al.* qui fournissent des introductions plus complètes au logiciel R.

1.1 Les différents interfaces

Il y a plusieurs interfaces pour utiliser R. Tout d'abord, il y a l'interface graphique standard de R, qui est très simple et épuré. Ce minimalisme peut dérouter les novices, mais pour installer R il faut commencer par cette version de R. Elle est disponible sur la plateforme du CRAN (*Comprehensive R Archive Network*). Une interface graphique un peu plus commode et plus jolie avec exactement les mêmes fonctionnalités s'appelle RStudio.

1.1.1 Installation

Pour installer R sur votre ordinateur personnel voici les démarches. Notez que tout est gratuit.

1. **Installer** Rà partir de la plateforme du CRAN à l'adresse
   `http://cran.r-project.org`
   Installer la dernière version R-3.3.1.
2. **Installer** RStudio à partir du site
   `https://www.rstudio.com/products/rstudio/download2/`
3. **Installer Jupyter Notebook**. Quelques instructions sont disponibles à l'adresse
   `http://jupyter.readthedocs.io/en/latest/install.html`
   Le plus facile pour obtenir Jupyter passe à travers Anaconda que l'on peut télécharger sur le site
   `https://www.continuum.io/downloads`
   Le choix de la version de Python (Python 2.7 ou Python 3.5) n'a pas d'importance pour nous.
4. **Installer le kernel R pour Jupyter Notebook**. Pour des informations sur l'installation du kernel voir
   `https://github.com/IRkernel/IRkernel/blob/master/README.md`
   Sous Windows il faut essentiellement ouvrir R et exécuter les instructions suivantes : «eval=FALSE»= install.packages(c('pbdZMQ', 'repr', 'devtools')) devtools : :install_github('IRkernel/IRkernel') IRkernel : :installspec() @
   Sous Mac, il faut faire pareil, mais il faut lancer R à partir d'un terminal. Pour cela, on tape simplement

`R`

dans un terminal, puis on exécute les commandes ci-dessus et pour sortir de `R` on tape

`q()`

Ensuite, on lance Jupyter depuis le terminal par la commande

`jupyter notebook`

Sous Ubuntu, taper la commande suivante dans le terminal (avec votre mot de passe)

`sudo apt-get install libcurl4-openssl-dev libssl-dev libzmq3-dev`

ensuite lancer `R` depuis le terminal par l'instruction

`R`

puis tapez

```
install.packages(c(`crayon`, `pbdZMQ`, `devtools`))
devtools::install\_github(paste0(`IRkernel/`, c(`repr`, `IRdisplay`,
  `IRkernel`)))
IRkernel::installspec()
```

Enfin quitter `R` par la commande

`q()`

et taper

`jupyter notebook`

dans le terminal pour lancer Jupyter.

1.2 Ma première session

Lancer `RStudio` sur votre machine.

Vous voyez apparaître trois fenêtres. La plus importante est celle à gauche : la fenêtre de commandes (*Console*). La console permet d'exécuter des instructions ou commandes. À la fin de l'affichage qui se déroule dans la console, le caractère d'invite de commande `>` vous invitant à taper votre première instruction en langage `R`.

```
>
```

Il s'agit du symbole d'incitation à donner une instruction (*prompt symbol*).

1.2.1 `R` comme calculatrice

Tout d'abord, on peut utiliser `R` comme une calculette classique. Essayons de calculer $2 + 3$:

```
> 2+3

[1] 5
```

R nous a compris et affiche le bon résultat. Essayons un exemple légèrement plus difficile :

```
> 3+8^2*(1-3)

[1] -125
```

Nous voyons que R connaît bien les règles de calcul et les priorités opératoires. Quand R ne comprend pas l'instruction, parce qu'elle est erronée, il affiche un message d'erreur :

```
2+*2

Error: <text>:1:3: '*' inattendu(e)
1: 2+*
     ^
```

Si l'instruction est incomplète, R retourne le signe + au lieu du prompteur >. Il faut alors compléter l'instruction ou sortir de cette situation et récupérer la main en tapant Ctrl + c ou Echap :

```
> 2*(3+1
+ )

[1] 8
```

Il existe de nombreuses fonctions mathématiques prédéfinies sous R. Exemples :

```
> exp(1)

[1] 2.718282

> sqrt(2)

[1] 1.414214

> abs(-5.7)

[1] 5.7

> round(-5.7)

[1] -6
```

Lorsqu'on effectue un calcul inadmissible, R renvoie la valeur NaN (pour *not a number*) et affiche un warning :

```
> sqrt(-2)

Warning in sqrt(-2): production de NaN

[1] NaN
```

La valeur `Inf` représente l'infini. Exemples :

```
> 1/0

[1] Inf

> -exp(exp(100))

[1] -Inf
```

Il faut jamais oublier que des calculs numériques ne sont pas forcément exacts à cause de la discrétisation inévitable des nombres sur une machine. Comparer les sorties suivantes :

```
> sin(0)

[1] 0

> sin(pi)

[1] 1.224647e-16

> sin(2*pi)

[1] -2.449294e-16
```

La précision de machine est de l'ordre de 10^{-16}. Cela semble petit, mais dans certains cas, les erreurs s'amplifient de façon étonnante. Exemple :

```
> sin(pi*10^16)

[1] -0.3752129
```

1.2.2 L'aide dans R

Le logiciel R possède un système d'aide efficace. Cette aide très complète, en anglais, est accessible au moyen de la fonction `help()` ou la commande `?` suivi du nom de la fonction pour laquelle vous cherchez de l'aide. Vous pouvez par exemple taper

```
> help(abs)
```

où de façon équivalente

```
> ?abs
```

pour obtenir de l'aide sur la fonction `abs()`. Vous constatez que chacune de ces commandes ouvre la page d'aide de la fonction `abs()` dans la fenêtre en bas à droite dans `RStudio`. On y trouve une description de la fonction, la liste des arguments à renseigner avec éventuellement leur valeur par défaut et la liste de résultats retournée par la fonction. En fin d'aide, un ou pluseirus exemples d'utilisation de la fonction sont présents et peuvent être copiés-collés dans `R` afin de comrpendre son utilisation.

Prenez le plus possible l'habitude d'utiliser le système d'aide de `R`.

1.2.3 Scripts

Au lieu de travailler directement dans la ligne de commandes, vous pouvez écrire des scripts. Un script est un fichier texte qui contient une succession de commandes. Vous pouvez le sauvegarder et le rouvrir lors d'une session ultérieure. Pour développer du code, l'utilisation de scripts est indispensable.

Pour créer un nouveau script et le sauvegarder, on utilise le menu déroulant. La terminaison de fichier d'un script `R` est `.R`.

Afin d'exécuter une partie des commandes d'un script, on sélectionne les lignes voulues et on appuie sur le bouton Run dans la barre à outils ou on appuie sur Ctrl + R (Cmd + Enter pour les Mac). Vous pouvez aussi exécuter une seule ligne d'instructions `R` du script en tapant Ctrl + R lorsque le curseur clignotant se trouve sur la ligne en question dans la fenêtre de script. Pour exécuter l'ensemble des instructions d'un script, on clique sur le bouton Source. Cela évitera de surcharger inutilement votre console.

Il est possible d'insérer des commentaires dans vos programmes en les faisant précéder du caractère # :

```
> 'ce qui suit' # ne s'affiche pas

[1] "ce qui suit"
```

Tout ce qui est écrit après le caractère # jusqu'à la fin de la ligne n'est pas interprété par `R`. Des commentaires permettent de fournir des informations utiles sur votre programme pour les futurs utilisateurs du script.

1.2.4 Répertoire de travail

Le répertoire de travail (en anglais *working directory*) est le répertoire par défaut, ce qui veut dire c'est le répertoire qui s'ouvre quand vous cliquez sur le bouton pour enregistrer un script. La commande pour connaître le répertoire de travail actuel est

```
> getwd()

[1] "/Users/tabea/Enseignement/TP"
```

Pour le changer, on va dans *Session* → *Set Working Directory* → *Choose Directory* et on sélectionne le dossier souhaité. On peut également changer le répertoire de travail en utilisant la fonction `setwd()` (au risque de ne pas y arriver à cause des nombreuses fautes de frappes possibles...) :

```
setwd("/Users/tabea/Lectures/Stat_with_R")
```

Prenez l'habitude de changer le répertoire de travail à chaque ouverture de session avant tout autre chose. Par ailleurs, pour une bonne gestion des fichiers, il est fotement recommandé de créer un dossier pour chaque projet sur lequel vous travailler.

1.2.5 Créer des objets R

La force de R (comme de toute autre langage de programmation) consiste à aller au-delà de la simple utilisation de R comme calculatrice. Comme vous l'avez sans doute remarqué, R répond à vos requêtes en affichant le résultat obtenu après évaluation. Ce résultat est affiché puis perdu. Dans une première utilisation, cela peut paraître agréable, mais dans une utilisation plus poussée il apparaît plus intéressant de rediriger la sortie R de votre requête en la stockant dans une variable. En fait, pour développer des programmes, c'est-à-dire des ensembles de plusieurs d'instructions, il est très utile de pouvoir créer des objets qui stockent de l'information, qu'on peut modifier et avec lesquels on peut faire des calculs.

La création d'un objet R est tout simple. Dès qu'on assigne une valeur à un nom de variable, l'objet – s'il n'existe pas déjà – est créé et prend la valeur qu'on est en train de lui assigner. Cette opération s'appelle aussi affectation du résultat dans une variable. Une affectation évalue ainsi une expression, mais n'affiche pas le résultat qui est en fait stocké dans un objet. Exemple : On ceée une variables de nom `x` qui prend la valeur 74 :

```
> x <- 70 + 4
```

L'exécution de cette instruction ne produit pas de sortie, mais l'objet a bien été créé. Pour afficher la valeur d'un objet, on tape son nom dans la console :

```
> x

[1] 74
```

Maintenant on peut faire des calculs avec cette variable :

```
> 3*x

[1] 222
```

On change la valeur de `x`, en lui assignant une nouvelle valeur :

```
> x <- x+6
```

On vérifie que la valeur de `x` a bien changée :

```
> x

[1] 80
```

Le symbole pour assigner une valeur est soit `<-` soit `=` Exemple :

```
> y = 17
> y

[1] 17
```

Pour les noms d'objet vous avez le droit d'être créatif! En fait, les noms comme `x, y, z` sont de très mauvais exemples. Il vaut mieux d'utiliser des noms parlants. Si les noms de variables sont bien choisis, les commentaires deviennent presque superflus pour expliquer le code. Voyez l'exemple suivant :

```
> poids <- 52
> taille <- 1.65
> IMC <- poids/taille^2
> IMC

[1] 19.10009
```

Un nom de variable admissible sous `R` doit commencer par une lettre. Il peut aussi inclure des chiffres ou les caractères `.` et `_`

1.2.6 Supprimer des objets

L'**environnement de travail** (ou *work space* en anglais) est une sorte d'espace de stockage d'objets `R` créés lors d'une session, c'est-à-dire c'est de la liste d'objets actuellement connus par `R`. Pour connaître tous les objets créés depuis le début de la session, on utilise la commande

```
> ls()

[1] "IMC"    "poids"  "taille" "x"      "y"
```

Pour en supprimer par exemple l'objet `x`, on tape

```
> rm(x)
```

et on vérifie que `x` a bien disparu de l'environnement de travail :

```
> ls()

[1] "IMC"    "poids"  "taille" "y"
```

Afin de supprimer tous les objets d'un seul coup, on écrit

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

character(0)
```

1.2.7 Quitter RStudio

À la fermeture de RStudio, le logiciel vous propose de sauver une image de la session, ce qui permet d'enregistrer l'ensemble d'objets créés lors de cette session, ou plus précisément d'objets actuellement dans l'environnement de travail. On peut les charger dans une session ultérieure pour reprendre son travail au même endroit.

2 Les vecteurs

Nous avons déjà vu comment définir une variable qui stocke une valeur numérique. En statistique, on aura plutôt besoin de *vecteurs* de valeurs numériques. On peut en créer par la fonction c( ) (c comme *concaténer*) :

```
vec1 <- c(5,8,3)
vec1

[1] 5 8 3
```

On peut également concaténer des vecteurs :

```
vec2 <- c(vec1,0,vec1)
vec2

[1] 5 8 3 0 5 8 3
```

Voici quelques exemples pour des calculs avec des vecteurs :

```
vec1+2

[1]  7 10  5
```

```
5*vec1

[1] 25 40 15
```

```
vec1+vec1
```

```
[1] 10 16  6
```

```
vec1*vec1
```

```
[1] 25 64  9
```

```
vec1^2
```

```
[1] 25 64  9
```

Vous constatez que c'est très simple. Mais attention aux dimensions des vecteurs :

```
vec1+vec2
```

```
Warning in vec1 + vec2: la taille d'un objet plus long n'est pas multiple de
la taille d'un objet plus court
```

```
[1] 10 16  6  5 13 11  8
```

Vous constatez que malgré le message d'erreur (prenez l'habitude de les lire !), `R` produit un résultat. Ici, `R` répète le vecteur `vec1` deux fois afin de créer un vecteur de la taille du vecteur `vec2`. C'est une technique courante de R lorsque les dimensions ne correspondent pas : répéter l'objet le plus petit jusqu'à ce que le calcul souhaité fait sens. (Mais assurez vous que ça correspond bien au calcul que vous souhaitez faire !).

2.1 Fonctions mathématiques pour les vecteurs

En `R` il existe un grand nombre de fonctions mathématiques prédéfinies et très utiles en pratique. Voici une liste de fonctions pour vecteurs :

- `sum()`
- `prod()`
- `length()`
- `min()`
- `max()`
- `which.min()`
- `which.max()`
- `sort()`
- `order()`
- `mean()`
- `var()`
- `sd()`
- `median()`

Exercice 1. Essayez de comprendre chacune des fonctions ci-dessus. Pour cela, lancez `RStudio`, définissez quelques vecteurs jouets, appliquez les fonctions ci-dessus et essayez de comprendre la sortie. Si vous n'arrivez pas à comprendre une fonction, consultez l'aide en utilisant la fonction `help()` ou la commande `?` suivi du nom de la fonction qui vous intéresse.

2.2 Générer des pseudo-variables aléatoires

En `R` il existe des fonctions qui génèrent des réalisations de variables aléatoires d'une loi donnée. Par exemple, la commande suivante génère 10 réalisations de la loi uniforme $U[0,1]$:

```
runif(10)

 [1] 0.28882472 0.06704419 0.72498083 0.74772676 0.93207037 0.84667471
 [7] 0.36957618 0.26681129 0.94650848 0.05227033
```

Exécutons la commande de nouveau pour constater que les valeurs renvoyées changent :

```
runif(10)

 [1] 0.40777739 0.44542213 0.02337936 0.81132470 0.70175204 0.02875725
 [7] 0.97466179 0.20489910 0.12958125 0.94587459
```

En fait, la fonction `runif( )` est un générateur de *pseudo-variables aléatoires*. On dit *pseudo*, parce qu'on a l'impression que ce sont des réalisations d'une vraie variable aléatoire. (C'est tout un champs de recherche de fabriquer des générateurs de pseudo-variables aléatoires (c'est-à-dire des algorithmes) qui imitent au mieux le vrai aléa.)

De même, il existe un générateur des (pseudo-)réalisations de la loi normale standard :

```
rnorm(5)

[1]  0.83728215 -0.21270838  0.47703402  0.08815319  0.48344911
```

Vous avez constaté que l'argument des fonctions `runif` et `rnorm` est le nombre de réalisations souhaité.

Pour générer des réalisations de la loi normale $\mathcal{N}(\mu, \sigma^2)$ de moyenne μ et de variance σ^2, on utilise toujours la fonction `rnorm( )` avec deux arguments supplémentaires pour indiquer les valeurs souhaitées des paramètres de la loi normale. Notez qu'en `R` ce sont les valeurs de la moyenne μ et de l'*écart-type* σ (et non de la variance σ^2) qu'il faut passer à la fonction `rnorm( )`. Ainsi, on obtient 20 réalisations de la loi normale $\mathcal{N}(-5, 2)$ par les instructions :

```
n <- 20
mu <- -5
sig <- sqrt(2)
rnorm(n,mu,sig)
```

```
 [1] -6.318289 -4.270383 -7.167644 -6.776132 -4.970636 -5.121694 -5.722180
 [8] -6.357610 -7.339206 -4.405400 -5.091744 -4.468014 -5.383254 -6.367177
[15] -3.285406 -5.705408 -5.304806 -5.521791 -4.868823 -6.676552
```

Exercice 2. A partir de l'exercice suivant, nous voulons écrire toutes nos commandes dans un script `R` . Pour le faire proprement, il y a quelques petites démarches préparatives à faire.

- Tout d'abord, créez un nouveau dossier (nommé, par exemple, *TPstatistique*) dans lequel vous allez enregistrer tous les fichiers associés à ce cours.
- Ensuite, changez le répertoire de travail pour ce nouveau dossier. Pour cela, cliquez dans `RStudio` sur *Session* → *Set Working Directory* → *Choose Directory* et puis sélectionner votre dossier *TPstatistique*.
- Puis, ouvrez un nouveau script : *File* → *New File* → *R Script*
- Ecrivez en première ligne du script un commentaire du genre
  `# Exercice 3 du TP 1`
  Rappel : tout commentaire commence par le symbole `#`, et c'est du texte qui ne sera pas interpréter par `R` lors de l'exécution de cette ligne.
- Enfin, cliquez sur le bouton dans la barre à outils pour enregistrer le script dans votre dossier *TPstatistique*. Choissisez un nom de fichier à peu près intelligent, comme p. ex. *exo_ TP_ 1*.

Exercice 3. La moyenne empirique est un estimateur de la moyenne. Vérifions empiriquement la qualité de cet estimateur par des simulations.

- Considérons la loi normale $\mathcal{N}(10, 1)$. Générez un échantillon de taille 20 et calculez la moyenne empirique associée. Comparez le résultat à la moyenne théorique. Répétez plusieurs fois les mêmes instructions afin de vérifier si la valeur de la moyenne empirique est à peu près stable sur des échantillons différents.
- Faites la même chose pour la loi normale $\mathcal{N}(10, 10)$ et des échantillons de taille 20. Qu'observez vous ?
- La loi normale est une loi symétrique et donc son coefficient d'asymétrie vaut 0. Vérifiez que le coefficient d'asymétrie empirique $\alpha_{\mathbf{x}}$ est près de 0 pour des échantillons d'une loi normale. Testez des lois normales avec de différents paramètres et des tailles d'échantillon variées. Rappelons la définition du coefficient d'asymétrie empirique $\alpha_{\mathbf{x}}$ associé à l'échantillon $(X_1, \ldots, X_n)$:

$$\alpha_{\mathbf{x}} = \frac{\frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X}_n)^3}{s_{\mathbf{x}}^3}, \qquad \text{où} \qquad s_{\mathbf{x}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2}.$$

- Même question pour le coefficient d'aplatissement empirique $\beta_{\mathbf{x}}$ d'un échantillon de la loi normale qui est censé d'être près de 0. Rappelons sa définition :

$$\beta_{\mathbf{x}} = \frac{\frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X}_n)^4}{s_{\mathbf{x}}^4} - 3.$$

- Ajoutez un commentaire dans votre script avec un résumé de vos conclusions.

Exercice 4. Soient $X_1, X_2, \ldots$ des variables aléatoires indépendantes de loi normale standard. Posons $Y_k = \sum_{i=1}^{k} X_i^2$. Souvenez-vous que la loi de Y_k est la loi du khi-deux χ^2_k à k degrés de liberté.
D'après le théorème central limite, on a

$$\sqrt{\frac{k}{2}}\left(\frac{1}{k}\sum_{i=1}^{k} X_i^2 - 1\right) = \frac{1}{\sqrt{2k}}(Y_k - k) \xrightarrow{\mathcal{L}} \mathcal{N}(0,1), \qquad k \to \infty.$$

On en déduit que la loi de Y_k est approximativement la loi normale $\mathcal{N}(k, 2k)$ pour k suffisamment grand :

$$\chi^2_k \approx \mathcal{N}(k, 2k).$$

Le but de cet exercice consiste à vérifier empiriquement à partir de quelle valeur de k cette approximation est bonne. Plus précisément, nous allons comparer des indicateurs statistiques de la loi de Y_k aux valeurs théoriques correspondantes de la loi normale $\mathcal{N}(k, 2k)$.
En `R` on génère n réalisations de la loi du khi-deux χ^2_d à d degrés de liberté par la commande `rchisq(n,d)`.

- Pour tout k dans $\{1, 10, 100, 1000\}$ générez un échantillon de taille 1000 de la loi de χ^2_k, et comparez la moyenne empirique associée à la moyenne de la loi normale $\mathcal{N}(k, 2k)$. Commentez.
- Même question pour la variance empirique $s^2_{\mathbf{x}}$: Pour tout k dans $\{1, 10, 100, 1000\}$ générez un échantillon de taille 1000 de la loi de χ^2_k, et comparez la variance empirique associée à la variance de la loi normale $\mathcal{N}(k, 2k)$. Commentez.
- Même question pour le coefficient d'asymétrie.
- Même question pour le coefficient d'appaltissement.
- Ajoutez un commentaire dans votre script avec un résumé de vos conclusions.

3 Le type des données

`R` n'est pas limité à des variables de type numérique. L'instruction `typeof(x)` permet de connaître le type ou mode de l'objet `x`. Énumérons maintenant les principaux types de données.

3.1 Type numérique (*numeric*)

Parmi les objets de type numérique, on distingue deux types : les entiers (*integer*) et les réels (*real* ou *double*). Définissons deux variables et vérifions leur type :

```
var.a <- 1.7
typeof(var.a)

[1] "double"
```

```
var.b <- -3
typeof(var.b)

[1] "double"
```

Les variables `var.a` et `var.b` sont du type *double*. Afin d'obtenir une variable de type *integer*, il faut forcer le type en utilisant la fonction `as.integer`. L'intérêt du type *integer* est que son stockage prend moins de place en mémoire.

```
var.c <- as.integer(var.b)
typeof(var.c)

[1] "integer"
```

3.2 Type booléen ou logique (*logical*)

`R` sait répondre à certaines questions simples, pas avec "oui" ou "non", mais avec `TRUE` ou `FALSE`. Voici quelques exemples :

```
# égalité des valeurs de var.a et var.b ?
var.a==var.b

[1] FALSE
```

```
# var.a plus grand que var.b ?
var.a>var.b

[1] TRUE
```

Ces réponses sont des objets de type logique. Le type logique ne peut prendre que les valeurs `TRUE` et `FALSE` (il n'y a pas de valeur du genre "je n'ai sais pas" ou "je m'en fous"). On appelle les questions des exemples ci-dessus des *conditions logiques*, car le résultat est de type logique.

De même, le résultat renvoyé par des fonctions qui testent le type d'un objet `x` (commençant par `is.`) est de type logique :

```
is.numeric(var.b)

[1] TRUE
```

```
is.integer(var.b)

[1] FALSE
```

```
var.x <- FALSE
is.logical(var.x)

[1] TRUE
```

`TRUE` et `FALSE` peuvent être saisis de manière plus succincte en tapant respectivement `T` et `F`.

Lorsque cela se révèle nécessaire, ce type de données est naturellement converti en type numérique sans qu'il y ait à le spécifier : `TRUE` vaut 1 et `FALSE` vaut 0. Cela est illustré par les exemples suivant :

```
TRUE + FALSE

[1] 1
```

```
TRUE/2 - var.b

[1] 3.5
```

```
(TRUE + T)^2 + FALSE*F + T*FALSE + F

[1] 4
```

3.3 Type chaînes de caractères (*character*)

Toute information mise entre guillemets (simple ' ou double ") correspond à une chaîne de caractères :

```
string1 <- "I love maths"
string1

[1] "I love maths"
```

```
typeof(string1)

[1] "character"
```

```
string2 <- 'Moi aussi !'
string2

[1] "Moi aussi !"
```

```
is.character(string2)

[1] TRUE
```

3.4 Conversion de type

Les conversions de type sont possibles grâce aux fonctions commençant par `as.`

Il faut cependant être prudent quant à la signification de ces conversions. `R` retourne toujours un résultat à une instruction de conversion même si cette dernière n'a pas de sens. Voici quelques exemples :

```
# Conversion vers une chaîne de caractères
as.character(2.3)

[1] "2.3"
```

```
# Conversion depuis une chaîne de caractères
string3 <- "2.3"
as.numeric(string3)

[1] 2.3
```

```
# Conversion depuis une chaîne de caractères
as.integer(string3)

[1] 2
```

```
# Conversion impossible
as.integer(string1)

Warning: NAs introduits lors de la conversion automatique

[1] NA
```

4 Structure des données

`R` offre la possibilité d'organiser les différents types de données définies précédemment. En statistique nous travaillons beaucoup avec des vecteurs et des tableaux (*dataframes*) présentés ici. Les listes seront traitées ultérieurement.

4.1 Les vecteurs

Cette structure de données est la plus simple. Elle représente une *suite de données de même type*. Nous avons déjà vu les vecteurs de type numérique, mais il y a également des vecteurs de type logique ou de type chaîne de caractère :

```
vec.num <- c(1,4,9,2,0)
vec.num

[1] 1 4 9 2 0
```

```
vec.char <- c("R","c'est","trop","facile","!")
vec.char

[1] "R"      "c'est"  "trop"   "facile" "!"
```

Un vecteur de type logique peut être obtenu par une condition logique appliquée à un ou plusieurs vecteurs (de type numérique ou chaîne de caractère). Exemple :

```
vec.logique <- (vec.num>=4)
vec.logique

[1] FALSE  TRUE  TRUE FALSE FALSE
```

Dans cet exemple les éléments du vecteur `vec.num` sont comparé un à un à la valeur 4 et le résultat est reporté dans le vecteur `vec.logique`.

Dans l'exemple suivant, une comparaison élément par élément de deux vecteurs est effectuée :

```
vec.char2 <- c("R","c'est pas","mon","truc","!")
vec.char2

[1] "R"         "c'est pas" "mon"       "truc"      "!"

vec.logique2 <- (vec.char == vec.char2)
vec.logique2

[1]  TRUE FALSE FALSE FALSE  TRUE
```

4.1.1 Conversion automatique

Par définition, tous les éléments d'un vecteur sont de même type. Lorsqu'on mélange des données de types différents lors de la création d'un vecteur, `R` se charge alors d'opérer une conversion implicite vers le type de donnée le plus général comme vous pouvez le constater dans les exemples ci-dessous.

```
c(3,TRUE,7)
```

```
[1] 3 1 7
```

```
c(3,T,"7")
```

```
[1] "3"    "TRUE" "7"
```

4.1.2 Suites régulières

On peut créer des vecteurs particuliers grâce à des suites régulières.

Exercice 5. Exécutez les commandes suivantes et utilisez l'aide afin de comprendre
- l'utilisation des deux-points `:`,
- la fonction `seq()` et ses arguments `length` et `by`,
- la fonction `rep()` et ses arguments `times` et `each`.

```
2:5
```

```
.5:4
```

```
12:-4
```

```
seq(2,6)
```

```
seq(2,6,by=.2)
```

```
seq(2,6,length=4)
```

```
rep(1,5)
```

```
rep(1:3,times=4)
```

```
rep(1:3,each=4)
```