Introduction à R

Importation et sauvegarde de fichiers

Pascal Bessonneau

06/2015

La spécification des chemins sous R

Tous les répertoires doivent être indiqués avec la syntaxe *NIXs*, c'est-à-dire avec des slashs (/) en lieu et place des backslashs (\) sous *Windows*.

La spécification des chemins sous R

Pour rappel, il y a trois répertoires spéciaux à se souvenir :

- `.` c'est le répertoire courant
- `..` c'est le répertoire parent du répertoire courant
- `~` c'est votre répertoire personnel

Le répertoire de travail de R

Le répertoire de travail de R est le point de référence pour accéder à vos fichiers.
Dans RStudio, le répertoire de travail ne change pas en manipulant l'explorateur de fichiers à droite. Il ne changera que si vous utilisez le bouton *More/Set as Working Directory*.

Le répertoire de travail de R

Le répertoire de travail de R dépend de la façon dont vous le lancez :

- avec la console graphique de R sous *Windows*, vous partez dans le répertoire X :\Program Files\R...
- avec RStudio, dans votre répertoire personnel
- depuis un terminal *Linux* dans le répertoire courant du shell
- ...

Les commandes utiles pour les répertoires

Les commandes à se souvenir sont les suivantes :

- setwd pour *set working directory* qui permet de déterminer le répertoire courant
- getwd pour *get working directory* qui renvoie dans un vecteur *character* le chemin courant
- dir pour récupérer dans un vecteur *character* les fichiers du répertoire courant. Il faut spécifier *all=T* pour avoir les fichiers commençant par un point (même sous *Windows*).

Sur la console R, il y a des raccourcis dans les menus. Dans RStudio, vous avez la fenêtre *Files*.

Les chemins sous R

```
> setwd("~/Documents/R/FormationR")
```

```
> getwd()

## [1] "/home/pascal/Dropbox/R/journee"
```

Les chemins sous R

```
> dir(pattern="tex")

##  [1] "00_Introduction-concordance.tex"
##  [2] "00_Introduction.synctex.gz"
##  [3] "00_Introduction.tex"
##  [4] "01_RStudio-concordance.tex"
##  [5] "01_RStudio.synctex.gz"
##  [6] "01_RStudio.tex"
##  [7] "03_Les_types_de_donnees-concordance.tex"
##  [8] "03_Les_types_de_donnees.synctex.gz"
##  [9] "03_Les_types_de_donnees.tex"
## [10] "04_Les_types_objets.synctex.gz"
## [11] "04_Les_types_objets.tex"
## [12] "05_Donnees-concordance.tex"
## [13] "05_Donnees.synctex.gz"
## [14] "05_Donnees.tex"
## [15] "10_Importation_et_exportation.tex"
## [16] "11_Statistiques_descriptives-concordance.tex"
```

Les chemins sous R

```
> dir(pattern="tex",all=T)

##  [1] "00_Introduction-concordance.tex"
##  [2] "00_Introduction.synctex.gz"
##  [3] "00_Introduction.tex"
##  [4] "01_RStudio-concordance.tex"
##  [5] "01_RStudio.synctex.gz"
##  [6] "01_RStudio.tex"
##  [7] "03_Les_types_de_donnees-concordance.tex"
##  [8] "03_Les_types_de_donnees.synctex.gz"
##  [9] "03_Les_types_de_donnees.tex"
## [10] "04_Les_types_objets.synctex.gz"
## [11] "04_Les_types_objets.tex"
## [12] "05_Donnees-concordance.tex"
## [13] "05_Donnees.synctex.gz"
## [14] "05_Donnees.tex"
## [15] "10_Importation_et_exportation.tex"
## [16] "11_Statistiques_descriptives-concordance.tex"
```

Sauvegarde de l'environnement

Il est appréciable de faire une sauvegarde complète de l'environnement avec lequel on travaille...
Dans ce cas, R permet de sauvegarder toutes les variables et fonctions en mémoire, seuls les paquets ne seront pas restaurés.
Il suffit d'utiliser la commande :

```
> save.image()
```

Sauvegarde de l'environnement

Dans le cas où il n'y a pas d'argument, le fichier s'appele automatiquement *.RData* et est sauvegardé dans le répertoire courant.
Lorsque R est lancé depuis un répertoire contenant un fichier dont le nom est *.RData*, il est automatiquement chargé.

Sauvegarde de l'environnement

La sauvegarde de l'environnement dans un fichier *.RData* vous est proposé quand vous quittez R, RStudio ou la console R.
Pour éviter ce comportement, pour un script par exemple, vous pouvez taper :

```
> q("no")
```

Sauvegarde de l'environnement

Lorsqu'on spécifie un argument, cela doit être une chaine texte qui indique le nom (et le répertoire éventuellement) du fichier.
Le format est un format *.Rdata* qui est le format propre de R. Il a la particularité d'être compressé (GNU/ZIP, *gzip*). Les fichiers produits sont donc légers.

Sauvegarde d'objets

Les objets complexes, *list*, *data.frame*, *array*, ... peuvent être sauvegardés au format *RData*.
Pour sauvegarder un(des) objet(s), il suffit d'utiliser la commande :

Sauvegarde d'objets

```
> save(iris,file="data/iris.RData")
> save(iris,mtcars,file="data/misc.RData")
```

Avec, en premier, l'objet (ou les objets) puis *file* et le nom du fichier.

Restauration d'objets

Il suffit pour recharger un environnement ou un objet (c'est-à-dire un objet de type *.RData*) d'utiliser la commande *load* :

```
> load("data/iris.RData")
```

La commande restaure en mémoire l'objet sous le nom que vous avez utilisé pour le sauvegarder (dans l'exemple c'est *iris*).

Restauration d'objets

Le chargement en lui-même est « silencieux »... C'est-à-dire que R ne précise pas le(s) objet(s) qui est(sont) chargé(s) par la commande load.
Comme vu précédemment, l'objet sauvegardé prends en mémoire le nom qu'on lui a donné lors de la commande *save*.

Restauration d'objets

Il est évident que, 6 mois après, il peut être difficile de se rappeler du nom de l'objet sauvegardé...
Comme souvent dans R, il faut changer le contexte d'évaluation pour obtenir le nom de l'objet :

```
> (load("data/iris.RData"))

## [1] "iris"
```

Restauration d'objets

Pour des traitements automatisés, il est ainsi possible de récupérer le nom des objets sauvegardés.

```
> recharger <- load("data/iris.RData")
> recharger

## [1] "iris"
```

read.table

Sauf cas particulier, on peut charger (presque) n'importe quel type de fichier texte délimité avec la commande *read.table*.
La fonction *read.table* prend comme argument le nom du fichier texte.

read.table

Par défaut ce fichier texte doit avoir comme séparateurs des blancs entre les champs et pour séparateur entre la partie entière et la partie décimale un point.
La première ligne n'est pas considérée comme une entête mais comme des données.
Chaque colonne détectée correspondra à une variable qui sera restituée dans un *data.frame*.

read.table

La fonction *read.table* est une fonction de haut niveau. Par haut niveau, cela signifie que le plus gros du travail est épargné à l'utilisateur.
En effet la fonction va détecter elle-même le type de chaque variable/colonne.

les options de *read.table*

Qu'en-est-il des autres formats de fichier avec séparateurs ?
Il suffit de se rappeler les trois arguments suivants :

- **header** si *TRUE* alors la première ligne est utilisée pour définir les noms de variable
- **sep** c'est un vecteur caractère qui permet de définir quel(s) est(sont) le(s) séparateur(s) de colonnes
- **dec** c'est un vecteur caractère qui définit le caractère utilisé pour séparer la partie entière de la partie décimale

les options de *read.table*

A partir de ces réglages, on peut charger n'importe quel fichier délimité.
Par exemple, le format CSV français d'Excel s'écrit :

```
> iris <- read.table("data/iris.csv",
+ header=T,sep=";",dec=",")
```

les options de *read.table*

Pour les formats de fichiers courants, il existe des alias de la fonction *read.table* : elles sont équivalentes à *read.table* mais ont des valeurs par défaut différentes.
Les version numérotées *2* correspondent aux formats français.

TABLE – Alias de read.table

Arguments	**read.delim**	**read.delim2**	**read.csv**	**read.csv2**
header	T	T	T	T
sep	\t	\t	,	;
dec	.	,	.	,

les options de *read.table*

Parmi les autres options utiles de *read.table*, on pourra noter les options suivantes :

stringsAsFactors par défaut *TRUE*, les variables de type *character*, si *TRUE*, seront transformées en variable de type *factor* automatiquement

encoding l'encodage du fichier qui peut par exemple être « latin1 » ou « UTF-8 »

na.strings un vecteur *character* indiquant la(es) valeur(s) à considérer comme une(des) valeur(s) manquante(s)

row.names le nom de la colonne ou le numéro de la colonne qui sera utilisée pour le nom des observations

nrows indique le nombre de lignes à lire

skip nombre de lignes à ignorer en début de fichier

les options de *read.table*

Il est à noter deux options pour forcer certain aspect du chargement :

col.names vecteur *character* permettant de nommer les variables.

as.is vecteur *character* indiquant le type de chaque colonne

les options de *read.table*

L'argument *as.is* permet notamment de spécifier un type *character* pour des codes « 001 », « 010 » qui seraient traités comme des nombres par R.
Dans ce cas, il peut être intéressant d'importer le fichier (ou une partie avec l'option *nrows*), récupérer le type de chaque variable dans un vecteur, et modifier seulement le type de quelques variables.

les options de *read.table*

Sans utiliser les fonctions *apply*...

1. on récupère le nom des colonnes
2. on crée un vecteur avec NA pour l'auto-détection de type et *character* pour la variable *Species*
3. on charge le fichier avec ce vecteur comme argument de *ColClasses*

les options de *read.table*

```
> iris <- read.csv2( "data/iris.csv", nrow = 1)
> types <- rep( NA, ncol(iris) )
> names(types) <- colnames(iris)
> types["Species"] <- "character"
> iris <- read.csv2( "data/iris.csv", colClasses=types )
> (types <- sapply(as.list(iris),class))

## Sepal.Length  Sepal.Width Petal.Length
##    "numeric"    "numeric"    "numeric"
##  Petal.Width      Species
##    "numeric"  "character"
```

les options de *read.table*

Un exemple plus souple avec les fonctions *apply* :
Pour éviter de spécifier le type de chaque colonne :

1. on lit une première fois le fichier (en partie)
2. on récupère et on modifie le type de chaque colonne
3. on lit le fichier avec le type de colonne définitif

les options de *read.table*

```
> iris <- read.csv2( "data/iris.csv", nrow = 10 )
> (types <- sapply(as.list(iris),class))

## Sepal.Length  Sepal.Width Petal.Length
##    "numeric"    "numeric"    "numeric"
##  Petal.Width      Species
##    "numeric"     "factor"

> types["Species"] <- "character"
> iris <- read.csv2( "data/iris.csv", colClasses=types )
> (types <- sapply(as.list(iris),class))

## Sepal.Length  Sepal.Width Petal.Length
##    "numeric"    "numeric"    "numeric"
##  Petal.Width      Species
##    "numeric"  "character"
```

l'aternative *readr*

Le paquet *readr* est un paquet de Hadley Wichkam (très connu).
Il fonctionne sur le principe d'automate finis par conséquent il est plus robuste aux erreurs dans les fichiers.
Par exemple il permet d'importer des fichiers contenant des verbatim avec des sauts de lignes si le champ est encadré par des quotes.

l'aternative *readr*

Le paquet a aussi comme particularité de disposer d'une fonction qui essaie de "deviner" le format du CSV (français/anglophone) permettant ainsi de simplifier l'importation.
Parmi les inconvénients, il y a le fait qu'il essaie de deviner le format (c'est un défaut et une qualité) et qu'il n'y a pas d'option spécifiques pour indiquer l'*encoding* du fichier.

l'aternative *readr*

Le diable se cache dans les détails, les fonctions de *readr* utilise la même syntaxe excepté qu'il faut remplacer les points par des "_".

```
> require(readr)
> iris <- read_csv2("Support R/data/Iris.csv")
```

l'aternative *readr*

Le diable se cache dans les détails, les fonctions de *readr* utilise la même syntaxe excepté qu'il faut remplacer les points par des "_".

```
> require(readr)
> iris <- read_csv2("Support R/data/Iris.csv")
```

l'aternative *readr*

readr permet de lire également des lignes de textes avec read_lines, des fichiers log, format fixe, etc. et ce plus rapidement que les fonctions de base.

l'aternative *data.table*

Avec le paquet *data.table* qui permet de manipuler des *data.frame* plus rapidement si celle-ci sont manipulées avec des variables "index" est livré une version de *read.csv*.
Attention le type de retour est *data.table* et non *data.frame*.

Exporter avec *write.table*

L'export d'un fichier texte délimité se fait avec la fonction *write.table*.
Comme la fonction *read.table*, elle a le même type d'alias.
Elle prend pour premier argument la *data.frame* à exporter puis l'argument *file* qui est un vecteur *character* indiquant le nom du fichier.
Les arguments essentiels sont les mêmes que pour *read.table* : *sep*, *dec* et *header*.

Exporter avec *write.table*

```
> write.csv2(iris,"data/iris.csv")
```

les options de *write.table*

Contrairement aux alias de *read.table*, les options des alias ne sont pas modifiables. Pour modifier le comportement pour les champs *sep*, *dec* et *header*, il faut passer par la fonction d'origine *write.table*.

les options de *write.table*

Les options intéressantes sont notamment :

row.names	par défaut *T*, les identifiants de ligne sont exportés dans une première colonne
append	pour ajouter à un fichier existant si TRUE
na	la valeur à utiliser pour les valeurs manquantes
col.names	pour spécifier les noms des colonnes éventuellement

Chargement de données au format fixe

La fonction *read.fwf* est l'équivalent de read.table pour les anciens formats de fichiers texte sans séparateur de colonne mais à position de colonne fixe.
On indique la taille de chaque colonne séquentiellement.

Chargement de données au format fixe

Cette fonction a un comportement un peu surprenant. Quand *header*=*T*, les noms de colonnes sont récupérés, ils doivent être séparés par un séparateur de champs.
En fait cela est logique si on considère que les noms peuvent être plus longs que la largeur attribuée aux données.

Chargement de données au format fixe

```
AMC  Concord2229304099
AMC  Pacer  1733504749
AMC  Spirit 2226403799
BuickCentury2032504816
BuickElectra1540807827
```

Ce fichier est importé avec la commande :

```
> a=read.fwf("data/fixed.txt",width=c(5, 7, 2, 4, 4))
> colnames(a) <- c("model","make","mph","weight","price")
```

Chargement de données au format fixe

La fonction ci-dessous permet de récupérer automatiquement les noms s'ils utilisent la disposition des données.

```
> read.fwf2 <- function ( file, width, ... ) {

+     l = scan( file, what="character", nlines=1, sep="\n" )

+     col.names <- substr(
+          rep(l,length(width)) ,
+          cumsum(c(1,width[-length(width)])),
+          c(cumsum(width))
+     )
+     return(
+        read.fwf( file, width=width, col.names=col.names, skip = 1, ...
+     )
+ }
```

Chargement de données au format fixe

On peut également importer ce type de fichier avec la commande *read.fortran* qui reprend la syntaxe de ce langage de 1977.
La syntaxe n'est pas développée ici. SAS utilise dans certains cas une syntaxe proche (pour les fonctions *put* et *input*).

Chargement de données texte avec les fonctions de bas niveau

Comme indiqué précédemment, les fonctions *read.table* sont des fonctions de haut niveau et nécessite peu de sueur pour l'utilisateur.
Mais elles reposent sur l'existence d'une fonction de bas niveau, *scan*, qui permet de lire n'importe quel format de fichier au prix d'un peu d'efforts.

Chargement de données texte/binaire avec les fonctions de bas niveau

R fournit aussi des fonctions de type C pour la lecture :

- **readChar** pour la lecture caractère par caractère d'un buffer texte
- **readBin** pour la lecture d'un fichier binaire
- **readLines** pour la lecture ligne par ligne d'un fichier

Sauvegarde de données texte avec les fonctions de bas niveau

Comme indiqué précédemment, les fonctions *write.table* sont des fonctions de haut niveau et nécessite peu de travail pour l'utilisateur.
Mais elles reposent sur l'existence d'une fonction de bas niveau, *cat*, qui permet de créer n'importe quel type de fichier.

Sauvegarde de données texte/binaire avec les fonctions de bas niveau

Les fonctions de type C pour l'écriture sont :

- cat pour l'enregistrement d'un buffer texte
- writeLines pour l'enregistrement ligne par ligne d'un fichier

le paquet *foreign*

Le paquet *foreign* permet de charger de nombreux formats externes. Il fonctionne comme *read.table*, avec en lieu et place de *table*, le type de fichier.

le paquet *foreign*

Les formats supportés sont : Minitab, S, SAS, SPSS, Stata, Systat, dBase,...

SPSS avec *foreign*

Par exemple pour un fichier SPSS :

```
> iris.spss <- read.spss("data/iris.sav")

## re-encoding from CP1252
```

SPSS avec *foreign*

```
> class(iris.spss)

## [1] "list"
```

SPSS avec *foreign*

```
> str(iris.spss)

## List of 5
##  $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 .
##  $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5
##  $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1
##  $ Species     : chr [1:150] "setosa    " "setosa    " "setosa    "
##  - attr(*, "label.table")=List of 5
##   ..$ Sepal.Length: NULL
##   ..$ Sepal.Width : NULL
##   ..$ Petal.Length: NULL
##   ..$ Petal.Width : NULL
##   ..$ Species     : NULL
##  - attr(*, "codepage")= int 1252
##  - attr(*, "variable.labels")= Named chr(0)
##   ..- attr(*, "names")= chr(0)
```

SPSS avec *foreign*

Les fonctions de ce paquet sont relativement avancées et permettent, pour la plupart des formats de fichiers, de récupérer les attributs des variables et autres données additionnelles.

SPSS avec *foreign*

En conséquence, l'objet retourné n'est pas toujours une *data.frame*. Dans le cas de SPSS, pour revenir à une *data.frame*, la commande est simple :

```
> str(as.data.frame(iris.spss))

## 'data.frame': 150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa    ","versicolor",..: 1
```

SPSS avec *foreign*

Mais on peut perdre éventuellement les attributs.
Les attributs sont accessibles par la fonction *attributes*.

SAS avec *foreign*

Le code SAS pour exporter une table sous ce format est le suivant :

```
libname xportout xport 'iris.xpt';
data xportout.iris;
   set iris;
run;
```

Mais les limitations sont nombreuses et cela reste moins pratique que d'utiliser un format texte.

SAS avec *haven*

Contrairement au paquet *foreign*, le paquet *haven*, permet d'importer des fichiers sas au format *sas7bdat*, c'est-à-dire le format "normal" de SAS.
Il est assez efficace mais il ne supporte pas toutes les fonctionnalités de SAS. Il est possible que l'importation échoue car des fonctionnalités avancées sont utilisés dans la table SAS.
Il n'existe pas de documentation claire sur ce qui est supporté et ce qui ne l'est pas ce qui rend les choses problématiques.

SAS avec *haven*

```
> require(haven)
> eleves <- read_sas("eleves.sas7bdat")
```

Une documentation spécifique existe quand il y a des dates dans les fichiers.

Stata 13

La nouvelle version de Stata (supérieure ou égale à la version 13) inaugure un nouveau format de fichiers.
Il n'est pas supporté par *foreign* par contre le paquet *readstata13* permet l'importation de ces fichiers.

JSON et XML

R permet l'importation des fichiers XML et JSON. Le premier est une syntaxe commune sur les applications orientées web old school tandis que le deuxième est typique des nouvelles applications web utilisant javascript lourdement comme Node.js, JQuery, D3.js, ... Par exemple sur de nombreux sites d'open data on trouve maintenant des exports en JSON pour faciliter la vie de ceux qui font des visualisations de ces données.

jsonlite

Le paquet le plus simple pour importer des données JSON est *jsonlite*.
On utilise l'argument simplifyDataFrame pour essayer de réduire les données à une *data.frame* dans la mesure du possible. Sinon les sorties sont du type *list* qui est la structure de données la plus proche du type JSON sous R.
Par exemple si on l'utilise sur les données des arbres remarquables collectées sur le site open data de la ville de Paris. . .

jsonlite

```
> require(jsonlite)

## Loading required package: jsonlite

> arbres <- fromJSON("data/arbresremarquablesparis2011.json",
+                    simplifyDataFrame = T)
```

jsonlite

Si on regarde de plus près, le retour est relativement complexe car les coordonnées géographiques notamment, font que l'objet est une liste. Les identifiants des arbres sont des vecteurs et les descriptifs des arbres sont dans une *data.frame* nommée*fields*. Les coordonnées sont dans une *list*.
En général, sur les formats un peu complexes, les traitements demandent un peu de programmation.

XML

Le paquet le plus utile et le plus rapide pour l'XML est le paquet *XML*. Il a fait l'objet d'un livre XML and Web Technologies for Data Sciences with R. Il n'est pas nécessaire d'acheter le livre pour se servir du paquet mais ce livre mais il est peu être utile si on travaille beaucoup avec les fichiers XML.
Le paquet contient une fonction *xmlToDataFrame* qui est généralement ce que l'on veut. Après il parfoit nécessaire de préciser sur quels noeuds on travaille avec la fonction *xmlNodes*.
Le paquet permet également de réaliser des requêtes *XSLT* éventuellement pour formater et/ou localiser les noeuds.

Les possibilités de R avec les bases de données

R est capable de transférer des données depuis une base de données vers R et inversement.
Il est en outre capable de lancer des commandes sur la base de données (suppression de tables, consultation des tables et des schémas).
Ces fonctionnalités sont basés, comme la plupart de langages modernes, sur le modèle DBI (Database Interface).

Le fonctionnement de la DBI

Le principe du module *DBI* est de rationnaliser les échanges avec les bases de données.
Ainsi on accède, sauf commande particulière, de la même façon depuis R à une base de données qu'il s'agisse de MySQL, PostgreSQL, Oracle ou SQLite.
La seule limite est de disposer d'un *driver* correspondant à la base de données auquelle on veut accéder.

Le fonctionnement de la DBI

1. Le module de R correspondant à la base de données est appelé par l'utilisateur
2. Le module *DBI* est chargé implicitement
3. L'utilisateur se connecte à la base de données
4. l'utilisateur effectue les requêtes qu'il souhaite
5. L'utilisateur, à la fin de l'utilisation, clôt la connexion

Le fonctionnement de la DBI

Il est à noter que l'utilisateur à la fin de la connexion doit récupérer l'objet créé. Cet objet complexe représente une connexion entre R et la base de données. Par la suite, cet objet est utilisé pour toutes les opérations sur la base.
Rien n'interdit (sauf l'administrateur de la bd) d'avoir plusieurs connexions actives à la base de données avec des paramètres utilisateur identiques ou différents.
Le danger est de ne pas penser à détruire l'objet *connexion* et de laisser une connexion ouverte sur la base de données.

Le fonctionnement de la DBI

Sauf exception, la création d'une connexion nécessite des droits utilisateur adéquats et une déclaration d'usage auprès de l'administrateur de la base de données.
Cela permet notamment de choisir le pilote le plus adapté pour l'attaque de la base de données par R.
Et à l'administrateur d'être vigilant pendant la période d'apprentissage.

Le fonctionnement de la DBI

La différence avec certains logiciels statistiques est qu'il est nécessaire de connaître un peu le langage SQL pour accéder aux données.
En effet les requêtes sont formulées en SQL.

Un exemple avec SQLite

SQLite est une base de données de « poche ». En effet ce n'est pas un serveur et donc être installé sur son poste.
Son usage ici est purement illustratif et pédagogique.
Comme dit précédemment, la *DBI* fait que les commandes ci-dessous sont (presque) les mêmes que pour un serveur MySQL ou Oracle.

Connexion à la base de données

Dans un premier temps, il faut se connecter à la base de données.

```
> require(RSQLite)
> driver <- dbDriver("SQLite")
> con <- dbConnect( driver, dbname = "data/eslc_stu.sqlite")
```

Le driver correspond au nom de la base de données. Il est passé en argument à la *DBI*. A noter que SQLite ne nécessite pas d'utilisateur et de mots de passe.

Connexion à la base de données

Dans le cas d'un serveur de base de données, la syntaxe est légèrement différente. La commande de l'utilisateur, l'adresse du serveur ainsi que les identifiants.

```
> require(RMySQL)
> driver <- dbDriver("MySQL")
> conn <- dbConnect(
+     driver,
+     host="127.0.0.1",
+     username="user",
+     password = "passwd",
+     dbname = "eslc"
+ )
```

Requêtes sur une table

Dans ce cas, on veut rapatrier des informations de la base vers une data.frame.
Le pilote se charge automatiquement des conversions nécessaires entre les noms de variables (caractères interdits pour les noms de variables pour R ou pour la base de données) et le type des variables (par exemple les dates sont converties).

Requête SELECT sur une table

Le principe est toujours le même :

1. on « forme » la requête
2. la requête est exécutée

Requête SELECT sur une table

Une première requête très simple...

```
> requete <- dbSendQuery( con, "SELECT * from stu")
> stu <- fetch(requete, n = 5 )
> dbClearResult(requete)
```

L'argument $n = 5$ de *fetch* indique que l'on ne veut rapatrier que les 5 premières lignes de la requête.
La commande *dbClearResult(requete)* est optionnelle pour certaines base de données.

Requête SELECT sur une table

Une seconde requête très simple avec une sélection... La requête peut être très complexe. Elle doit respecter la syntaxe SQL supportée par le serveur.

```
> requete <- dbSendQuery(
+     con,
+     "SELECT * from stu WHERE country_id='FR'"
+ )
> stu <- fetch(requete, n = 5 )
> dbClearResult(requete)

## [1] TRUE
```

Au lieu d'utiliser les guillemets simples, on peux utiliser \".

Bonnes pratiques avec les serveurs...

Avant d'aller plus loin, il est important de rappeler certains points :

- la méthode d'authentification, les identifiants et l'usage doit être approuvé par l'administrateur de la base de données.
- le paquet *DBI* camoufle une bonne partie de la complexité des échanges. La commande *dbClearResults* illustre par exemple le fait que la requête est mise en cache par le serveur. Il est primordial de suivre quelques règles simples lors des requêtes...

Bonnes pratiques avec les serveurs...

- Même si c'est optionnel, vider dès que possible le cache coté serveur avec *dbClearResults*
- Ecrire tous les scripts avec *fetch(con, n=limits)*. La variable *limits* sera fixée au début pendant le déboguage à quelques lignes. Puis pour la valeur sera mise -1 quand le script sera stable. Mieux, limiter dans le code SQL le nombre de lignes récupérées (option *limit*, *fetch*, *rownum*,... selon le serveur).
- Sauf cas d'espèce, vous ne devez avoir qu' **une seule déclaration de connexion** : un seul objet *dbConnect*. Si vous en avez plusieurs, vous êtes autant d'utilisateurs sur le serveur que de connexions...
- Respecter toutes les précautions d'usage que vous employez habituellement en travaillant en SQL

Bonnes pratiques avec les serveurs...

Dans tous les cas, il est important de lire les recommandations indiquées dans la notice du pilote et de travailler de concert avec les administrateurs de la base de données.

Rapatrier une table...

Plutôt qu'une requête, vous pouvez rapatrier toute la table...

```
> stu <- dbReadTable( con, "NomDeLaTable" )
> stu <- dbReadTable( con, "NomDeLaTable", row.names=student_id )
```

Créer une table...

Si vous avez les droits, vous pouvez créer une table :

```
> dbWriteTable( con, "stu2", stu )
```

Les commandes non standardisées

Les commandes permettant de faire des requêtes de type *UPDATE*, *INSERT* utilise le *mapping*. La requête est écrite en SQL avec une syntaxe particulière pour les champs dont les données proviendront de R.
Lors de l'exécution, la commande met en relation chaque identificateur dans la requête avec une variable d'une *data.frame*.

Les commandes non standardisées

Les commandes permettant de visualiser les tables de la base de données, la structure des tables, ... sont des commandes propres à chaque base de données.
De ce fait, ils ne sont pas standards.
Pour *RSQLite*, les seules commandes (dont le noms sont assez transparents) sont *dbListTables*, *sqliteCopyDatabase*.

Le paquet *XLConnect*

Le paquet *XLConnect* est relativement simple d'utilisation et permet d'importer et d'exporter des fichiers Excel au format 2003 ou 2007/2010/2013.
Le paquet est basé sur une bibliothèque Java de la fondation Apache. Office n'est pas nécessaire mais « Java » l'est : soit openjdk, soit le Java d'Oracle. Le fait de pouvoir produire des fichiers Office sous NIXs par exemple est très appréciable.
Outre l'import et export simple, le paquet permet de modifier la présentation du tableur (couleurs,styles,...).
La documentation de *XLConnect* est particulièrement complète.

Le paquet *XLConnect*

Le système ressemble un peu à l'accès à une base de données. Il y a deux étapes :

- on ouvre une connection sur un fichier Excel (existant ou nouveau)
- on manipule le fichier en utilisant la connexion créée

Lecture de fichier Excel

```
> wb = loadWorkbook("eslc.xlsx", create = T)
> data = readWorksheet(wb, sheet = "Eleves")
```

Selon le suffixe *xls* ou emphxslx, il s'adapte automatiquement à la version d'Excel 2003 ou 2010/../2013.

Sauvegarde dans un fichier Excel

```
> classeur = loadWorkbook("res_eslc.xlsx", create = T)
> createSheet(classeur, name = "eleves")
> writeWorksheet(classeur, eslc_eleves, sheet = "eleves")
> saveWorkbook(wb)
```

En plus de la base...

Il faut savoir que contrairement aux fonctions de R, les *rownames* ne sont pas exportés. Un argument permet d'utiliser une colonne existante pour les *rownames* mais les *rownames* eux-mêmes.
L'écriture est définitive quand la fonction *saveWorkbook* est appelée et l'écriture se fait en ajoutant au fichier existant.
Ainsi si un *data.frame* plus petit est exporté dans une feuille déjà pleine, il restera les données pré-existantes.
Lors de l'écriture pour éviter ces problèmes, le moyen le plus direct est de supprimer la feuille existante.

En plus de la base...

```
> classeur = loadWorkbook("res_eslc.xlsx", create = T)
> try({removeSheet(wb, sheet = "eleves")},silent=T)
> createSheet(classeur, name = "eleves")
> writeWorksheet(classeur, eslc_eleves, sheet = "eleves")
> saveWorkbook(wb)
```

En plus de la base...

Il y a une autre possibilité en vidant la feuille existante. Plus doux... Avec la fonction *clearSheet*.
Mais là également pour des traitements automatisés il faudra souvent utiliser des *try* pour éviter les erreurs lors de l'exécution.

Les arguments supplémentaires

Les arguments *startRow*, *startCol* permettent à l'importation et à l'exportation de se concentrer sur une zone de la feuille. Votre collègue pourra laisser son titre en première ligne avec les données en troisième ligne.
Vous pouvez également personnaliser le style des cellules : cela passe par la création d'un style de cellules, d'ajouter des paramètres de style et enfin l'appliquer à des cellules définies.

La mise en forme

```
> csHeader = createCellStyle(wb, name = "header")
> setFillPattern(csHeader,
+                fill = XLC$FILL.SOLID_FOREGROUND)
> setFillForegroundColor(csHeader,
+                        color = XLC$COLOR.GREY_25_PERCENT)
> setCellStyle(wb, sheet = sheet, row = 1,
+              col = seq(length.out = ncol(curr)),
+              cellstyle = csHeader)
```

La mise en forme

Ce code est pompé sur la vignette car l'auteur n'a pas beaucoup pratiqué ces mises en forme. Remarquer qu'on retrouve un peu l'esprit Microsoft avec des constantes ...pas pratiques... pour appliquer un style (ex : *XLC$FILL.SOLID_FOREGROUND*). Elles sont contenues dans une grande liste *XLC*.
L'autre point non abordé est le fait que le paquet repose (en fait) beaucoup sur la définition de régions de cellules (ou noms sous Excel). Elle permet de contrôler l'importation, l'exportation, la mise en forme sur des portions de la feuille identifiées par des noms. Pour cela voir les fonctions : *createName*, *writeNamedRegion*, ...
Et pour info vous pouvez insérer des graphiques (statiques !) au format png sur une feuille.

readxl, l'alternative à XLConnect

Le paquet *readxl* est un paquet qui permet la lecture de fichiers Excel sans la présence d'un moteur Java.
Il lit les fichiers des versions 2003 à 2013.
Il est très rapide mais a moins d'options que le paquet XLConnect et est donc très simple à utiliser.

readxl, l'alternative à XLConnect

```
> require(readxl)
> classeur = read_excel("res_eslc.xlsx")
```

readxl, l'alternative à XLConnect

Avec l'argument *row*, il est possible de définir à partir de quelle ligne, la lecture se fait. On peut également définir le type de chaque colonne : character, date, numeric, ...
Ainsi l'utilisation est très proche de read_csv du même auteur.

Les sorties pour le reporting

Pour rester dans les paquets pour Microsoft Office, si vous en avez besoin il existe le paquet *ReporteRs* qui permet l'exportation de résultats et/ou tableaux pour Word et Powerpoint. Le site est très bien fait et a de nombreux tutoriaux.
ReporteRs site

Les sorties pour le reporting

Mais ces paquets restent en retrait car ils sont destinés à la production de rapports plutôt qu'à l'esprit *litterate programming* de Sweave qui est remplacé maintenant par *knitr* ou *RMarkdown*. Ils permettent de réaliser des "cahiers d'analyse" et/ou des rapports en permettant des documents dynamiques tout au long d'une analyse.

Shiny, D3.js, ...

Des paquets et le serveur Shiny permettent enfin d'exporter des documents totalement dynamiques pour le web.
Shiny est produit par la société qui produit RStudio. Le principe est de créer des documents web interactifs.
Pour avoir une idée : gallerie Shiny, paquet pour D3.js.
C'est pour créer des "datavisualistion" comme on peux voir dans le NYT par exemple.

Shiny, D3.js, ...

Ces infographies sont plus simples à réaliser qu'il n'y paraît.
Il y a un MOOC sur Coursera encore gratuit pour l'instant qui permet de s'y mettre.
Et c'est très pédagogiques : Developping Data Products
et un sur le *Reproductible*
*Research*https ://www.coursera.org/course/repdata

Base NoSQL

Des paquets pour les bases NoSQL et pour le big data sont disponibles sur le CRAN :

- HadoopStreaming
- hive
- RcppRedis
- RCassandra
- ...