

Se former au logiciel R : initiation et perfectionnement

François Rebaudo

2018-07-13

Contents

1	Préambule	5
2	Remerciements	7
3	Licence	9
4	Introduction	11
4.1	Pourquoi se former à R	11
4.2	Ce livre	11
4.3	Lectures complémentaires en français	11
I	Concepts de base	13
5	Premiers pas	15
5.1	Installation de R	15
5.2	R comme calculatrice	15
5.3	La notion d'objet	22
5.4	Les scripts	24
6	Choisir un environnement de développement	27
6.1	Editeurs de texte et environnement de développement	27
6.2	RStudio	27
6.3	Notepad++ avec Npp2R	30
6.4	Geany (pour Linux, Mac OSX et Windows)	33
6.5	Autres solutions	33
6.6	Conclusion	35
7	Les types de données	37
7.1	Le type <code>numeric</code>	37
7.2	Le type <code>character</code>	39
7.3	Le type <code>factor</code>	40
7.4	Le type <code>logical</code>	41
7.5	A propos de <code>NA</code>	42
7.6	Conclusion	42
8	Les conteneurs de données	43
8.1	Le conteneur <code>vector</code>	43
8.2	Le conteneur <code>list</code>	51
8.3	Le conteneur <code>data.frame</code>	62
8.4	Le conteneur <code>matrix</code>	62
8.5	Le conteneur <code>array</code>	62
8.6	Plus d'information sur les objets	62

9 Les fonctions	63
9.1 Quest-ce qu'une fonction	63
9.2 Les fonctions les plus courantes	63
9.3 Autres fonctions utiles	63
9.4 Ecrire une fonction	63
10 Importer et exporter des données	65
10.1 Lire des données depuis un fichier tableur	65
10.2 Sauver des données pour R	65
10.3 Exporter des données	65
11 Les boucles	67
11.1 Pourquoi faire des boucles	67
11.2 La boucle if	67
11.3 La boucle switch	67
11.4 La boucle for	67
11.5 La boucle while	67
11.6 repeat, next, break, stop	67
11.7 Les boucles de la famille apply	68

Chapter 1

Préambule

Ce livre est incomplet pour le moment et vous visualisez sa version préliminaire. Si vous avez des commentaires, des suggestions ou si vous identifiez des erreurs, n'hésitez pas à m'envoyer un mail (francois.rebaudo@ird.fr¹), ou si vous connaissez GitHub sur le site du projet (https://github.com/frareb/myRBook_FR). Ce livre est également disponible en espagnol (<http://myrbooksp.netlify.com/>).

Dernières modifications:

13/07/2018

- mise en ligne du contenu en français sur la base du livre en espagnol

¹<mailto:francois.rebaudo@ird.fr>

Chapter 2

Remerciements

Je remercie tous les contributeurs qui ont participé à améliorer ce livre par leurs conseils, leurs suggestions de modifications et leurs corrections :

=> liste à mettre à jour (attente de confirmation écrite des contributeurs)

- Camila Benavides Frias (Bolivia)
- EQ ()

Les versions gitbook, html et epub de ce livre utilisent les icônes open source de Font Awesome (<https://fontawesome.com>). La version PDF utilise les icônes issues du projet Tango disponibles depuis opencart (<https://opencart.org/>). Ce livre a été écrit avec le package R bookdown (<https://bookdown.org/>). Le code source est disponible sur GitHub (https://github.com/frareb/myRBook_FR). La compilation utilise Travis CI (<https://travis-ci.org>). La version en ligne est hébergée et mise à jour grâce à Netlify (<http://myrbookfr.netlify.com/>).

Chapter 3

Licence

Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 France (CC BY-NC-ND 3.0 FR ; <https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>)

C'est un résumé (et non pas un substitut) de la licence.

Vous êtes autorisé à :

- Partager — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats.
- L'Offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.

Selon les conditions suivantes :

- Attribution — Vous devez créditer l'Œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'Œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son Œuvre.
- Pas d'Utilisation Commerciale — Vous n'êtes pas autorisé à faire un usage commercial de cette Œuvre, tout ou partie du matériel la composant.
- Pas de modifications — Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'Œuvre originale, vous n'êtes pas autorisé à distribuer ou mettre à disposition l'Œuvre modifiée.
- Pas de restrictions complémentaires — Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser l'Œuvre dans les conditions décrites par la licence.

Notes :

Vous n'êtes pas dans l'obligation de respecter la licence pour les éléments ou matériel appartenant au domaine public ou dans le cas où l'utilisation que vous souhaitez faire est couverte par une exception. Aucune garantie n'est donnée. Il se peut que la licence ne vous donne pas toutes les permissions nécessaires pour votre utilisation. Par exemple, certains droits comme les droits moraux, le droit des données personnelles et le droit à l'image sont susceptibles de limiter votre utilisation.

Chapter 4

Introduction

4.1 Pourquoi se former à R

manipuler ses données statistiques et nombreux packages disponibles communauté d'utilisateurs graphiques de qualité transparence scientifique et reproductibilité des résultats

4.2 Ce livre

L'objectif de ce livre est de fournir aux étudiants et aux personnes souhaitant s'initier à R une base solide pour ensuite mettre en oeuvre leurs propres projets scientifiques et la valorisation de leurs résultats. Il existe de nombreux livres dédiés à R, mais aucun ne couvre les éléments de base de ce langage dans un objectif de rendre les résultats scientifiques publiables et reproductibles. De manière générale ce livre s'adresse à toute la communauté scientifique et en particulier à celle intéressée par les sciences du vivant, et les nombreux exemples de ce livre s'appuieront sur des études en agronomie et en écologie.

initiation aux bases de R : pour tous ! perfectionnement en français contrairement à d'autres livres, importance de la lisibilité du code, des standards, orientation recherche "outil pour le scientifique"

4.3 Lectures complémentaires en français

- R pour les débutants, Emmanuel Paradis (https://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf)
- Introduction à la programmation avec R, Vincent Goulet (https://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf)

Part I

Concepts de base

Chapter 5

Premiers pas

5.1 Installation de R

Le programme permettant l'installation du logiciel R peut être téléchargé depuis le site web de R : <https://www.r-project.org/>. Sur le site de R il faut au préalable choisir un miroir CRAN (serveur depuis lequel télécharger R ; sauf cas particulier le plus proche de sa localisation géographique), puis télécharger le fichier *base*. Les utilisateurs de Linux pourront préférer un `sudo apt-get install r-base`.



Le logiciel R peut être téléchargé depuis de nombreux serveurs du CRAN (Comprehensive R Archive Network) à travers le monde. Ces serveurs s'appellent des miroirs. Le choix du miroir est manuel. Les informations complémentaires comme cette note seront toujours représentées avec ce pictogramme *information*.

5.2 R comme calculatrice

Une fois le programme lancé, une fenêtre apparaît dont l'aspect peut varier en fonction de votre système d'exploitation (Figure 5.1). Cette fenêtre est dénommée la *console*.

La console correspond à l'interface où va être interprété le code, c'est à dire à l'endroit où le code va être transformé en langage machine, exécuter par l'ordinateur, puis retransmis sous une forme lisible par des humains. Cela correspond à l'écran d'affichage d'une calculatrice (Figure ??). C'est de cette manière que R va être utilisé dans la suite de cette section.



Tout au long de ce livre, les exemples de code R apparaîtront sur fond en gris. Ils peuvent être copiés et collés directement dans la console, bien qu'il soit préférable de reproduire soit même les exemples dans la console (ou plus tard dans les scripts). Le résultat de ce qui est envoyé dans la console apparaîtra également sur fond en gris avec `##` devant le code afin de bien faire la distinction entre le code et le résultat du code.

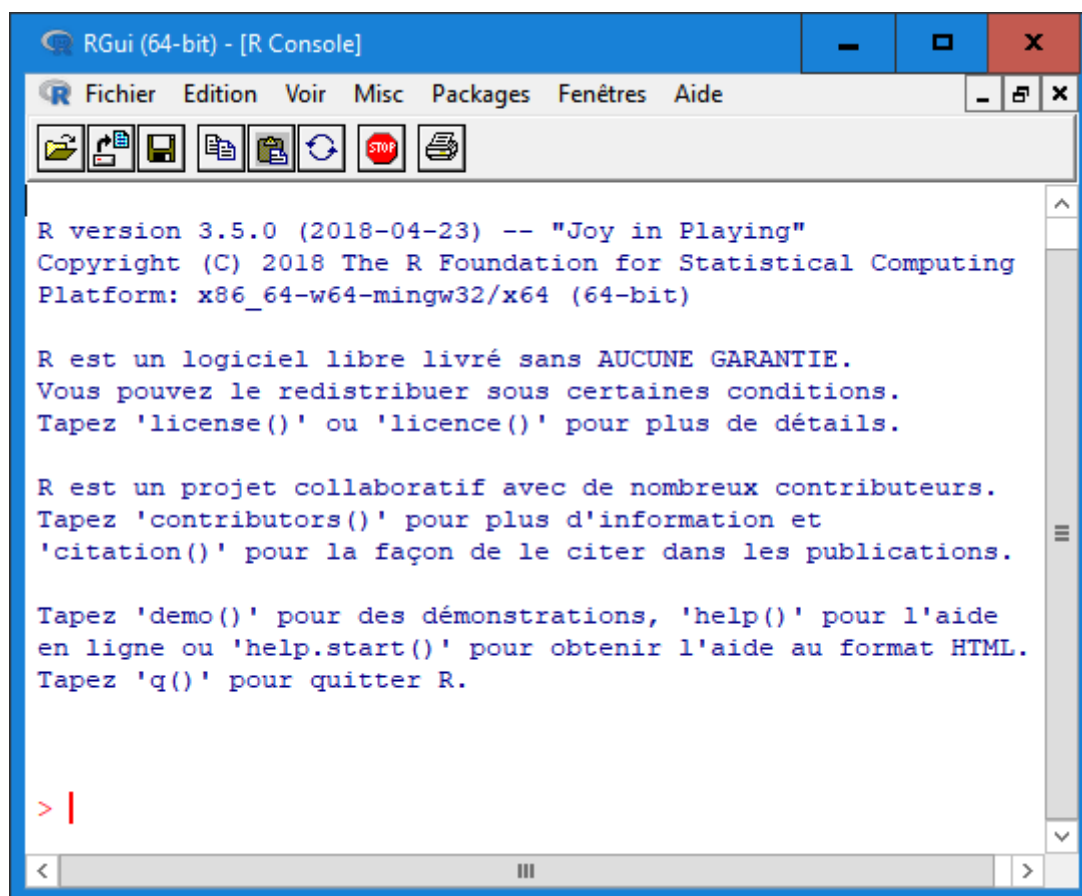
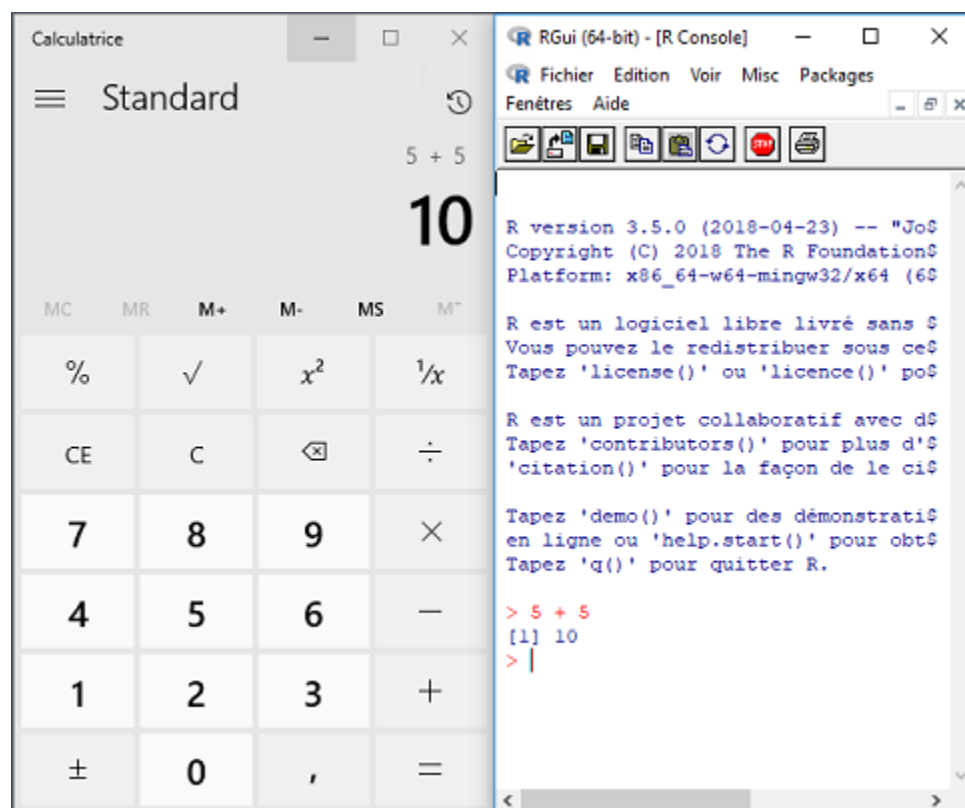


Figure 5.1: Capture d'écran de la console R sous Windows.



Les opérateurs arithmé-

Table 5.1: Opérateurs arithmétiques.

Label	Opérateur
Addition	+
Soustraction	-
Multiplication	*
Division	/
Puissance	^
Modulo	%%
Quotien Décimal	%/%

tiques

```
5 + 5
```

```
## [1] 10
```

Si nous écrivons `5 + 5` dans la console puis **Entrée**, le résultat apparaît précédé du chiffre `[1]` entre crochets. Ce chiffre correspond au numéro du résultat (dans notre cas, il n'y a qu'un seul résultat ; nous reviendrons sur cet aspect plus tard). Nous pouvons également noter dans cet exemple l'utilisation d'espaces avant et après le signe `+`. Ces espaces ne sont pas nécessaires mais permettent au code d'être plus lisible par les humains (i.e., plus agréable à lire pour nous comme pour les personnes avec qui nous serons amenés à partager notre code). Les opérateurs arithmétiques disponibles sous R sont résumés dans la table 5.1.

Classiquement, les multiplications et les divisions sont prioritaires sur les additions et les soustractions. Au besoin nous pouvons utiliser des parenthèses.

```
5 + 5 * 2
```

```
## [1] 15
```

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

```
## [1] 20
```

L'opérateur modulo correspond au reste de la division euclidienne. Il est souvent utilisé en informatique par exemple pour savoir si un nombre est pair ou impair (un nombre modulo 2 va renvoyer 1 si il est impair et 0 si il est pair).

```
451 %% 2
```

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

```
288 %% 2
```

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

```
(5 + 5 * 2) %% 2
```

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

```
((5 + 5) * 2) %% 2
```

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

Table 5.2: Opérateurs de comparaison.

Label	Operador
plus petit que	<
plus grand que	>
plus petit ou égal à	<=
plus grand ou égal à	>=
égal à	==
différent de	!=

R intègre également certaines constantes dont `pi`. Par ailleurs le signe infini est représenté par `Inf`

```
pi
```

```
## [1] 3.141593
```

```
pi * 5^2
```

```
## [1] 78.53982
```

```
1/0
```

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



le *style* du code est important car le code est destiné à être lisible par nous plus tard et par d'autres personnes de manière générale. Pour avoir un style lisible il est recommandé de mettre des espaces avant et après les opérateurs arithmétiques. Les informations concernant le *style* seront toujours représentées avec ce pictogramme afin qu'elles soient facilement identifiables.

5.2.1 Les opérateurs de comparaison

R est cependant bien plus qu'une simple calculatrice puisque'il permet un autre type d'opérateurs : les opérateurs de comparaison. Ils servent comme leur nom l'indique à *comparer* des valeurs entre elles (Table 5.2).

Par exemple si nous voulons savoir si un chiffre est plus grand qu'un autre, nous pouvons écrire :

```
5 > 3
```

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

R renvoie la valeur `TRUE` si la comparasion est vraie et `FALSE` si la comparaison est fausse.

```
5 > 3
```

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

```
2 < 1.5
```

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

```
2 <= 2
```

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

```
3.2 >= 1.5
```

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

Nous pouvons combiner les opérateurs arithmétiques avec les opérateurs de comparaison.

```
(5 + 8) > (3 * 45/2)
```

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



Dans la comparaison $(5 + 8) > (3 * 45/2)$ les parenthèses ne sont pas nécessaires mais elles permettent au code d'être plus facile à lire.

Un opérateur de comparaison particulier est *égal à*. Nous verrons dans la section suivante que le signe $=$ est réservé à un autre usage : il permet d'affecter une valeur à un objet. L'opérateur de comparaison *égal à* doit donc être différent, c'est pour cela que R utilise `==`.

```
42 == 53
```

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

```
58 == 58
```

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

Un autre opérateur particulier est *différent de*. Il est utilisé avec *un point d'interrogation* suivi de *égal*, `!=`. Cet opérateur permet de d'obtenir la réponse inverse à `==`.

```
42 == 53
```

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

```
42 != 53
```

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

```
(3 + 2) != 5
```

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

```
10/2 == 5
```

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

R utilise TRUE et FALSE qui sont aussi des valeurs qui peuvent être testées avec les opérateurs de comparaison. Mais R attribue également une valeur à TRUE et FALSE :

```
TRUE == TRUE
```

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

```
TRUE > FALSE
```

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

```
1 == TRUE
```

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

```
0 == FALSE
```

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

```
TRUE + 1
```

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

```
FALSE + 1
```

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

```
(FALSE + 1) == TRUE
```

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

La valeur de TRUE est de 1 et la valeur de FALSE est de 0. Nous verrons plus tard comment utiliser cette information dans les prochains chapitres.

R est aussi un langage relativement permissif, cela veut dire qu'il admet une certaine flexibilité dans la manière de rédiger le code. Débattre du bien fondé de cette flexibilité sort du cadre de ce livre mais nous pourrions trouver dans du code R sur Internet ou dans d'autres ouvrages le raccourcis T pour TRUE et F pour FALSE.

```
T == TRUE
```

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

```
F == FALSE
```

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

```
T == 1
```

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

```
F == 0
```

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

```
(F + 1) == TRUE
```

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

Table 5.3: Opérateurs logiques.

Label	Operador
n'est pas	!
et	&
ou	
ou exclusif	xor()

Bien que cette façon de se référer à TRUE et FALSE par T et F soit assez répandue, dans ce livre nous utiliserons toujours TRUE et FALSE afin que le code soit plus facile à lire. Encore une fois l'objectif d'un code est de non seulement être fonctionnel mais aussi d'être facile à lire et à relire.

5.2.2 Les opérateurs logiques

Il existe un dernier type d'opérateur, les opérateurs logiques. Ils sont utiles pour combiner des opérateurs de comparaison (Table 5.3).

```
!TRUE
```

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

```
!FALSE
```

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

```
((3 + 2) == 5) & ((3 + 3) == 5)
```

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

```
((3 + 2) == 5) & ((3 + 3) == 6)
```

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

```
(3 < 5) & (5 < 5)
```

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

```
(3 < 5) & (5 <= 5)
```

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

L'opérateur logique `xor()` correspond à un *ou exclusif*. C'est à dire que l'un des deux **arguments** de la **fonction** `xor()` doit être vrai, mais pas les deux. Nous reviendrons plus tard sur les **fonctions** et leurs **arguments**, mais retenons que l'on identifie une fonction par ses parenthèses qui contiennent des arguments séparés par des virgules.

```
xor((3 + 2) == 5, (3 + 3) == 6)
```

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

```
xor((3 + 2) == 5, (3 + 2) == 6)
```

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

```
xor((3 + 3) == 5, (3 + 2) == 6)
```

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

```
xor((3 + 3) == 5, (3 + 3) == 6)
```

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



Il est recommandé que les virgules , soient suivies par un espace afin que le code soit plus agréable à lire.

5.2.3 Aide sur les opérateurs

Le fichier d'aide en anglais sur les opérateurs arithmétiques peut être obtenue avec la commande ? '+' celui sur les opérateurs de comparaison avec la commande ? '==' et celui sur les opérateurs logiques avec la commande ? '&'.

5.3 La notion d'objet

Un aspect important de la programmation avec R, mais aussi de la programmation en général est la notion d'objet. Comme indiqué sur la page web de wikipedia ([https://fr.wikipedia.org/wiki/Objet_\(informatique\)](https://fr.wikipedia.org/wiki/Objet_(informatique))), en informatique, un objet est un *conteneur*, c'est à dire quelque chose qui va contenir de l'information. L'information contenue dans un objet peut être très diverse, mais pour le moment nous allons contenir dans un objet le chiffre 5. Pour ce faire (et pour pouvoir le réutiliser par la suite), il nous faut donner un nom à notre objet. Avec R le nom des objets ne doit pas comprendre de caractères spéciaux comme `^$?|+(){}[]`, ne doit pas commencer par un chiffre ni contenir d'espaces. Le nom de l'objet doit être représentatif de ce qu'il contient, tout en étant ni trop court ni trop long. Imaginons que notre chiffre 5 corresponde au nombre de répétitions d'une expérience. Nous voudrions lui donner un nom faisant référence à *nombre* et à *répétition*, que nous pourrions réduire à *nbr* et *rep*, respectivement. Il existe plusieurs possibilités qui sont toutes assez répandues sous R :

- la séparation au moyen du caractère *tiret bas* : `nbr_rep`
- la séparation au moyen du caractère *point* : `nbr.rep`
- l'utilisation de lettres minuscules : `nbrrep`
- le style *lowerCamelCase* consistant en un premier mot en minuscules et des suivants avec une majuscule : `nbrRep`
- le style *UpperCamelCase* consistant à mettre une majuscule au début de chacun des mots : `NbrRep`

Toutes ces formes de nommer un objet sont équivalentes. Dans ce livre nous utiliserons le style *lowerCamelCase*. De manière générale il faut éviter les noms trop longs comme `leNombreDeRepetitions` ou trop courts comme `nR`, et les noms ne permettant pas d'identifier le contenu comme `maVariable` ou `monChiffre`, mais aussi `a` ou `b...`



Il existe différentes façons de définir un nom pour les objets que nous allons créer avec R. Dans ce livre il est utilisé le style *lowerCamelCase*. L'important n'est pas le choix du style mais la consistance dans son choix. L'objectif est d'avoir un code fonctionnel mais également un code facile et agréable à lire.

Maintenant que nous avons choisi un nom pour notre objet, il faut le créer et faire comprendre à R que notre objet doit contenir le chiffre 5. Il existe trois façons de créer un objet sous R :

- avec le signe `<-`
- avec le signe `=`
- avec le signe `->`

```

nbrRep <- 5
nbrRep = 5
5 -> nbrRep

```

Dans ce livre nous utiliserons toujours la forme `<-` par souci de consistance et aussi parce que c'est la forme la plus répandue.

```

nbrRep <- 5

```

Nous venons de créer un objet `nbrRep` et de lui affecter la valeur 5. Cet objet est désormais disponible dans notre environnement de calcul et peut donc être utilisé. Voici quelques exemples :

```

nbrRep + 2

```

```

## [1] 7

```

```

nbrRep * 5 - 45/56

```

```

## [1] 24.19643

```

```

pi * nbrRep^2

```

```

## [1] 78.53982

```

La valeur associée à notre objet `nbrRep` peut être modifiée de la même manière que lors de sa création :

```

nbrRep <- 5
nbrRep + 2

```

```

## [1] 7

```

```

nbrRep <- 10
nbrRep + 2

```

```

## [1] 12

```

```

nbrRep <- 5 * 2 + 7/3
nbrRep + 2

```

```

## [1] 14.33333

```

L'utilisation des objets prend tout son sens lorsque nous avons des opérations complexes à réaliser et rend le code plus agréable à lire et à comprendre.

```

(5 + 9^2 - 1/18) / (32 * 45/8 + 3)

```

```

## [1] 0.4696418

```

```

terme01 <- 5 + 9^2 - 1/18
terme02 <- 32 * 45/8 + 3
terme01 / terme02

```

```

## [1] 0.4696418

```

5.4 Les scripts

R est un langage de programmation souvent dénommé *langage de script*. Cela fait référence au fait que la plupart des utilisateurs vont écrire des petits bouts de code plutôt que des programmes entiers. R peut être utilisé comme une simple calculatrice, et dans ce cas il ne sera pas nécessaire de conserver un historique des opérations qui ont été réalisées. Mais si les opérations à réaliser sont longues et complexes, il peut devenir nécessaire de pouvoir sauvegarder ce qui a été fait à un moment donné pour pouvoir poursuivre plus tard. Le fichier dans lequel seront conservées les opérations constitue ce que l'on appelle communément le script. Un script est donc un fichier contenant une succession d'informations compréhensibles par R et qu'il est possible d'exécuter.

5.4.1 Créer un script et le documenter

Pour ouvrir un nouveau script il suffit de créer un fichier texte vide qui sera édité par un éditeur de texte comme le bloc note sous Windows ou Mac OS, ou encore Gedit ou même nano sous Linux. Par convention ce fichier prend l'extension ".r" ou plus souvent ".R". C'est cette dernière convention qui sera utilisée dans ce livre. Depuis l'interface graphique de R il est possible de créer un nouveau script sous Mac OS et Windows via *fichier* puis *nouveau script* et *enregistrer sous*. Tout comme le nom des objets, le nom du script est important pour que nous puissions facilement identifier son contenu. Par exemple nous pourrions créer un fichier `formRConceptsBase.R` contenant les objets que nous venons de créer et les calculs effectués. Mais même avec des noms de variables et un nom de fichier bien définis, il sera difficile de se rappeler le sens de ce fichier sans une documentation accompagnant ce script. Pour documenter un script nous allons utiliser des *commentaires*. Les commentaires sont des éléments qui seront identifiés par R comme tel et qui ne seront pas exécutés. Pour spécifier à R que nous allons faire un commentaire, il faut utiliser le caractère octothorpe (croisillon) `#`. Les commentaires peuvent être insérés sur une nouvelle ligne ou en fin de ligne.

```
# creation objet nombre de repetitions
nbrRep <- 5 # commentaire de fin de ligne
```

Les commentaires peuvent aussi être utilisés pour qu'une ligne ne soit plus exécutée.

```
nbrRep <- 5
# nbrRep + 5
```

Pour en revenir à la documentation du script, il est recommandé de commencer chacun de ses scripts par une brève description de son contenu, puis lorsque le script devient long, de le structurer en différentes parties pour faciliter sa lecture.

```
# -----
# Voici un script pour acquérir les concepts de base
# avec R
# date de création : 25/06/2018
# auteur : François Rebaudo
# -----

# [1] création de l'objet nombre de répétitions
# -----

nbrRep <- 5

# [2] calculs simples
# -----

pi * nbrRep^2

## [1] 78.53982
```




Pour aller plus loin sur le style de code, un guide complet de recommandations est disponible en ligne (en anglais ; <http://style.tidyverse.org/>).

5.4.2 Exécuter un script

Depuis que nous avons un script, nous ne travaillons plus directement dans la console. Or seule la console est capable d'interpréter le code R et de nous renvoyer les résultats que nous souhaitons obtenir. Pour l'instant la technique la plus simple consiste à copier-coller les lignes que nous souhaitons exécuter depuis notre script vers la console. A partir de maintenant nous n'allons plus utiliser les éditeurs de texte comme le bloc note mais des éditeurs spécialisés pour la confection de scripts R. C'est l'objet du chapitre suivant.

Chapter 6

Choisir un environnement de développement

6.1 Editeurs de texte et environnement de développement

Il existe de très nombreux éditeurs de texte, le chapitre précédent a permis d'en introduire quelque uns parmi les plus simples comme le bloc note de Windows. Rapidement les limites de ces éditeurs ont rendu la tâche d'écrire un script fastidieuse. En effet, même en structurant son script avec des commentaires, il reste difficile de se repérer dans celui-ci. C'est là qu'interviennent les éditeurs de texte spécialisés qui vont permettre une écriture et une lecture agréable et simplifiée. L'éditeur de texte pour R certainement le plus répandu est Rstudio, mais il en existe bien d'autres. Faire une liste exhaustive de toutes les solutions disponibles sort du cadre de ce livre, ainsi nous nous focaliserons sur les trois solutions que j'utilise au quotidien que sont **Notepad++**, **Rstudio**, et **Geany**.

6.2 RStudio

6.2.1 Installer RStudio

Le programme pour installer Rstudio se retrouve dans la partie *Products* du site web de Rstudio (<https://www.rstudio.com/>). Nous allons installer RStudio pour un usage local (sur notre ordinateur), donc la version qui nous intéresse est *Desktop*. Nous allons utiliser la version *Open Source* qui est gratuite. Ensuite il nous suffit de sélectionner la version qui correspond à notre système d'exploitation, de télécharger le fichier correspondant et de l'exécuter pour lancer l'installation. Nous pouvons conserver les options par défaut tout au long de l'installation.

6.2.2 Un script avec RStudio

Nous pouvons alors ouvrir RStudio. Lors de la première ouverture, l'interface est divisée en deux avec à gauche la console R que nous avons vu au chapitre précédent (Figure 6.2). Pour ouvrir un nouveau script, nous allons dans le menu *File*, *New File*, *R script*. Par défaut ce fichier a comme nom *Untitled1*. Nous avons vu au chapitre précédent l'importance de donner un nom pertinent à nos scripts, c'est pourquoi nous allons le renommer *selecEnvDev.R*, dans le menu *File*, avec l'option *Save As....* Nous



Figure 6.1: Logo RStudio.

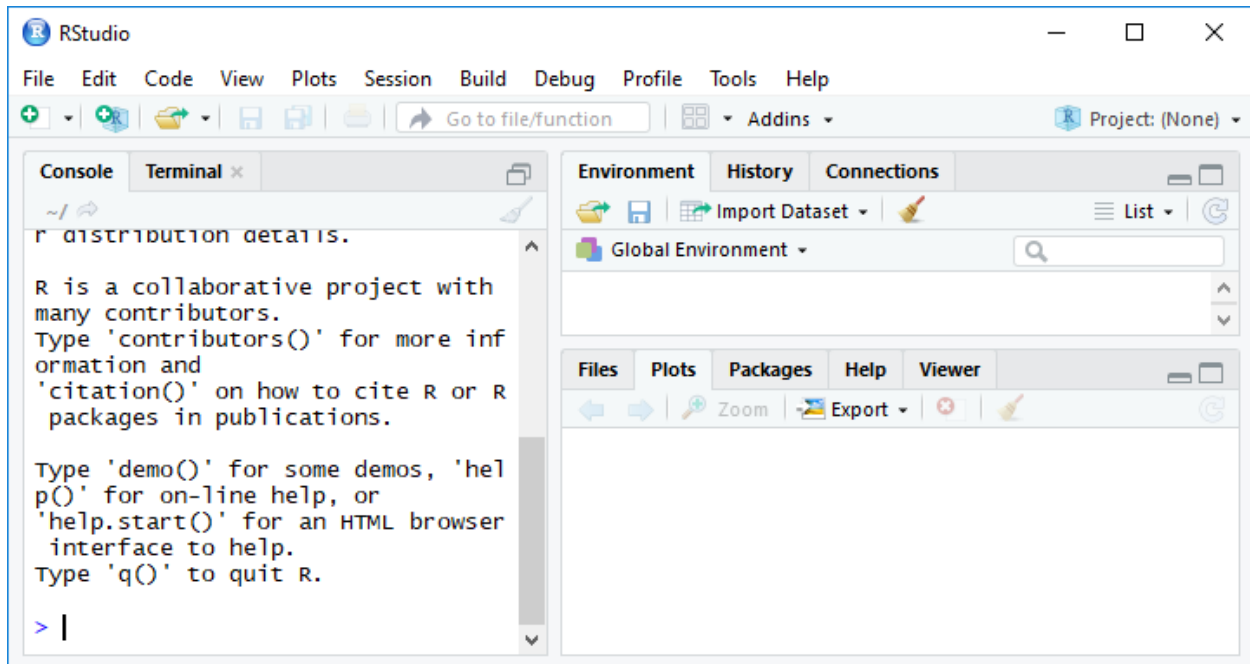


Figure 6.2: Capture d'écran de RStudio sous Windows : fenêtre par défaut.

avons pu noter que la partie gauche de RStudio est désormais séparée en deux, avec en bas de l'écran la console et en haut de l'écran le script.

Nous pouvons alors commencer l'écriture de notre script avec les commentaires décrivant ce que nous allons y trouver, et y ajouter un calcul simple. Une fois que nous avons recopié le code suivant, nous pouvons sauvegarder notre script avec la commande `CTRL + S` ou en se rendant dans *File*, puis *Save*.

```
# -----
# Un script para seleccionar su entorno de desarrollo
# fecha de creación : 27/06/2018
# autor : François Rebaudo
# -----

# [1] cálculos simples
# -----
nbrRep <- 5
pi * nbrRep^2

## [1] 78.53982
```

Pour exécuter notre script, il suffit de sélectionner les lignes que nous souhaitons exécuter et d'utiliser la combinaison de touches `CTRL + ENTER`. Le résultat apparaît dans la console (Figure 6.3).

Nous pouvons voir que par défaut dans la partie du script les commentaires apparaissent en vert, les chiffres en bleu, et le reste du code en noir. Dans la partie de la console ce qui a été exécuté apparaît en bleu et les résultats de l'exécution en noir. Nous pouvons également noter que dans la partie du code chaque ligne comporte un numéro correspondant au numéro de ligne à gauche sur fond gris. Il s'agit de la coloration syntaxique par défaut avec RStudio. Cette coloration syntaxique peut être modifiée en se rendant dans le menu *Tools, Global Options..., Appearance*, puis en choisissant un autre thème dans la liste *Editor theme:*. Nous allons choisir le thème *Cobalt*, puis *OK* (Figure 6.4).

Nous savons comment créer un nouveau script, le sauvegarder, exécuter son contenu, et changer l'apparence de RStudio. Nous

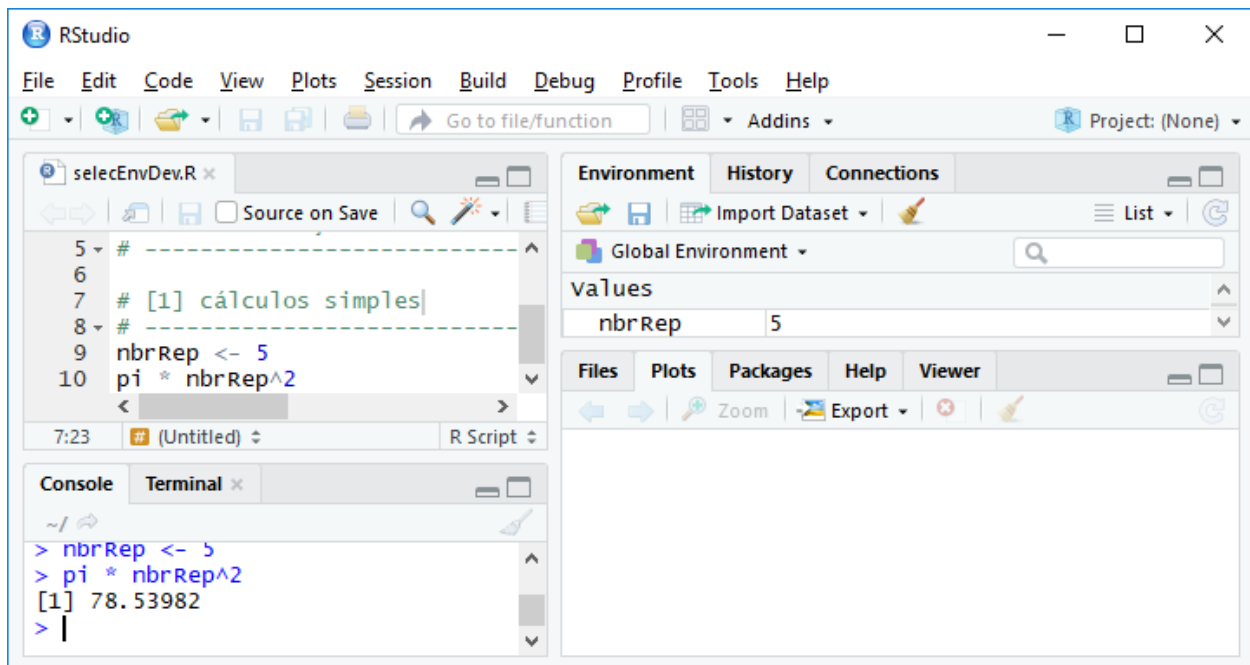


Figure 6.3: Capture d'écran de RStudio sous Windows : exécuter un script avec CTRL + ENTER.

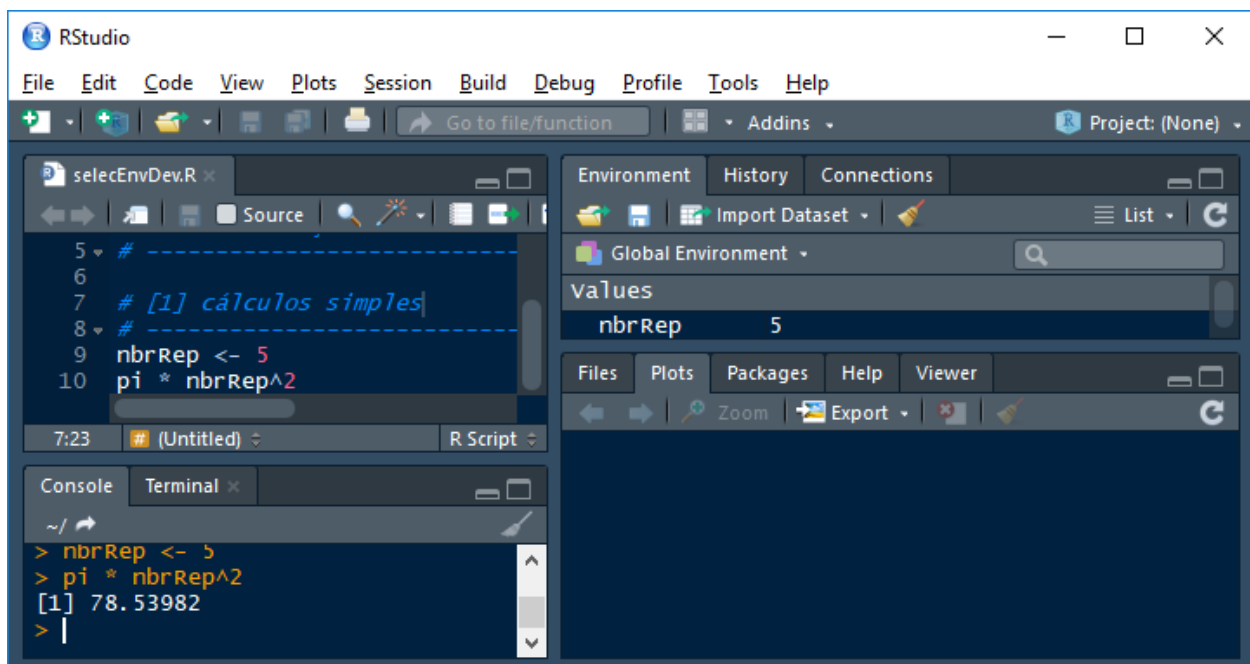


Figure 6.4: Capture d'écran de RStudio sous Windows : changer les paramètres de coloration syntaxique.



Figure 6.5: Logo Notepad++

verrons les nombreux autres avantages de RStudio tout au long de ce livre car c'est l'environnement de développement qui sera utilisé. Nous serons néanmoins particulièrement vigilants à ce que tous les scripts développés tout au long de ce livre s'exécutent de la même façon quel que soit l'environnement de développement utilisé.

6.3 Notepad++ avec Npp2R

6.3.1 Installer Notepad++ (pour Windows uniquement)

Le programme pour installer Notepad++ se trouve dans l'onglet *Downloads* (<https://notepad-plus-plus.org/download/>). Vous pouvez choisir entre la version 32-bit et 64-bit (64-bit si vous ne savez pas quelle version choisir). Notepad++ seul est suffisant pour écrire un script, mais il est encore plus puissant avec *Notepad to R* (*Npp2R*) qui permet d'exécuter automatiquement nos scripts dans une console en local sur notre ordinateur ou à distance sur un serveur.

6.3.2 Installer Npp2R

Le programme pour installer Npp2R est hébergé sur le site de Sourceforge (<https://sourceforge.net/projects/npp2r/>). Npp2R doit être installé après Notepad++.

6.3.3 Un script avec Notepad++

Lors de la première ouverture Notepad++ affiche un fichier vide *new 1* (Figure 6.6).

Puisque nous avons déjà créé un script pour le tester avec RStudio, nous allons l'ouvrir à nouveau avec Notepad++. Dans *Fichier*, sélectionnons *Ouvrir...* puis choisir le script *selecEnvDev.R* créé précédemment. Une fois le script ouvert, allons dans *Langage*, puis *R*, et encore une fois *R*. La coloration syntaxique apparaît (Figure 6.7).

L'exécution du script ne peut se faire que si Npp2R est en cours d'exécution. Pour se faire il est nécessaire de lancer le programme Npp2R depuis l'invite de Windows. Un icône devrait apparaître en bas de votre écran. L'exécution automatique du code depuis Notepad++ se fait en sélectionnant le code à exécuter puis en utilisant la commande F8. Si la commande ne fonctionne pas et que vous venez d'installer Notepad++, il est peut-être nécessaire de redémarrer votre ordinateur. Si la commande fonctionne, une nouvelle fenêtre va s'ouvrir avec une console exécutant les lignes souhaitées (Figure 6.8).

Comme pour RStudio, la coloration syntaxique peut être modifiée depuis le menu *Paramètres*, et un nouveau thème peut être sélectionné (par exemple *Solarized* dans la Figure 6.9)

Par rapport aux autres éditeurs de texte, Notepad++ a l'avantage d'être très léger et offre une vaste gamme d'options pour personnaliser l'écriture du code.

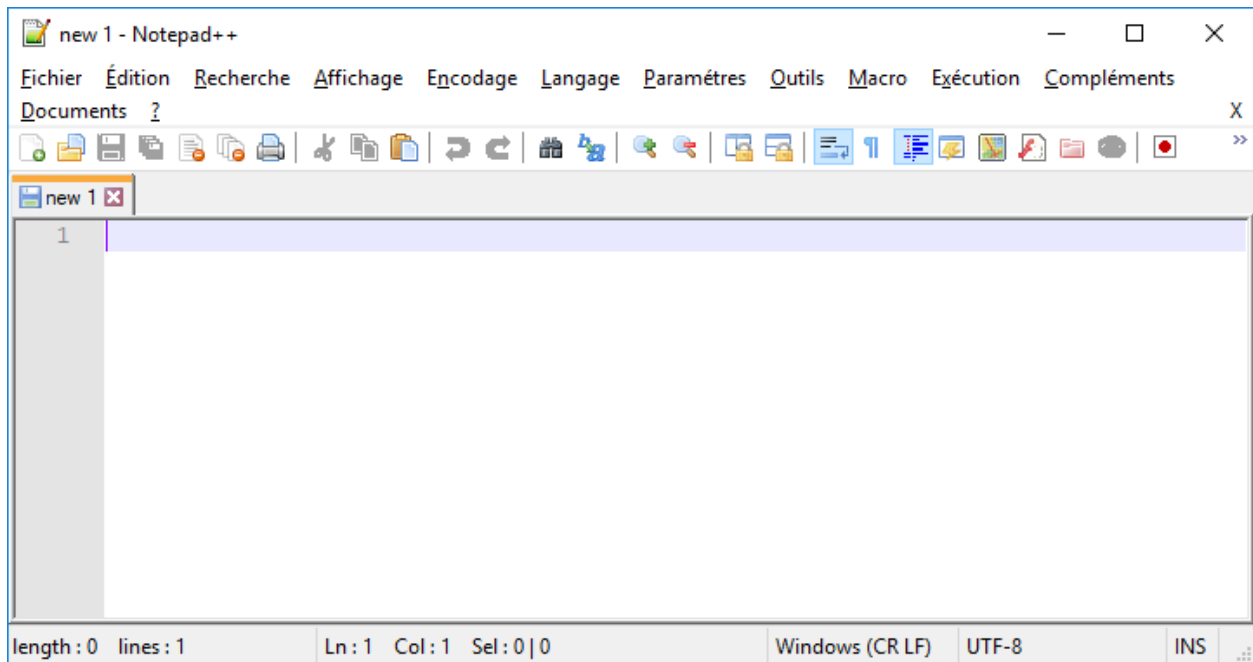


Figure 6.6: Capture d'écran de Notepad++ sous Windows : fenêtre par défaut.

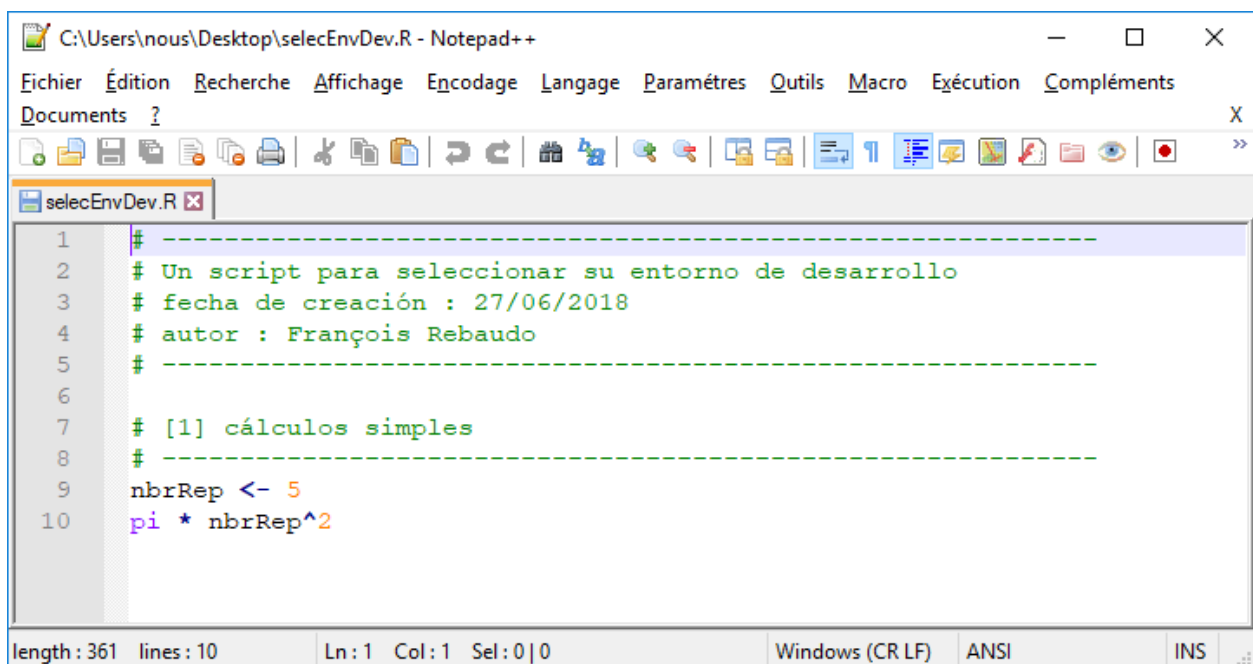
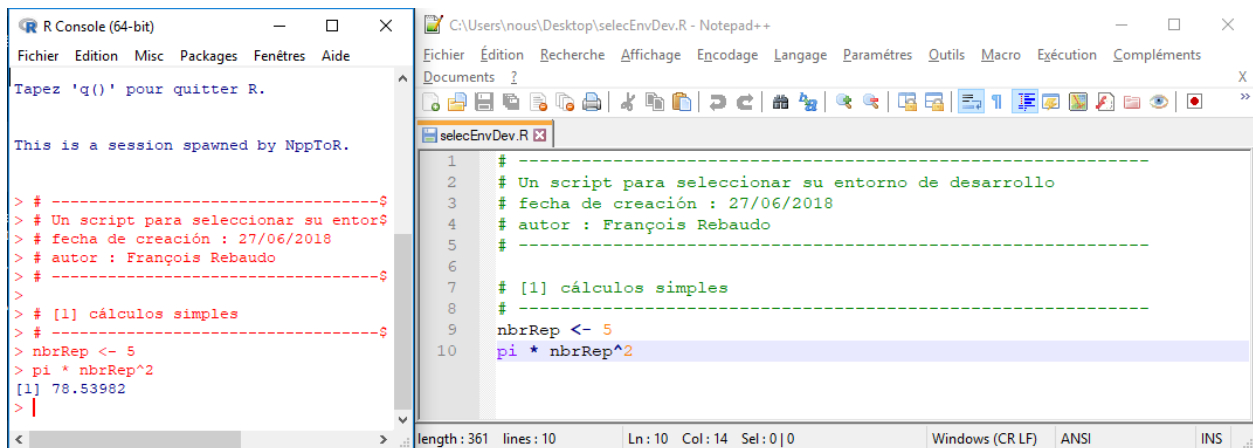


Figure 6.7: Capture d'écran de Notepad++ sous Windows : exécuter un script avec F8.



The screenshot shows two windows. The left window is 'R Console (64-bit)' with the following text:

```
Tapez 'q()' pour quitter R.

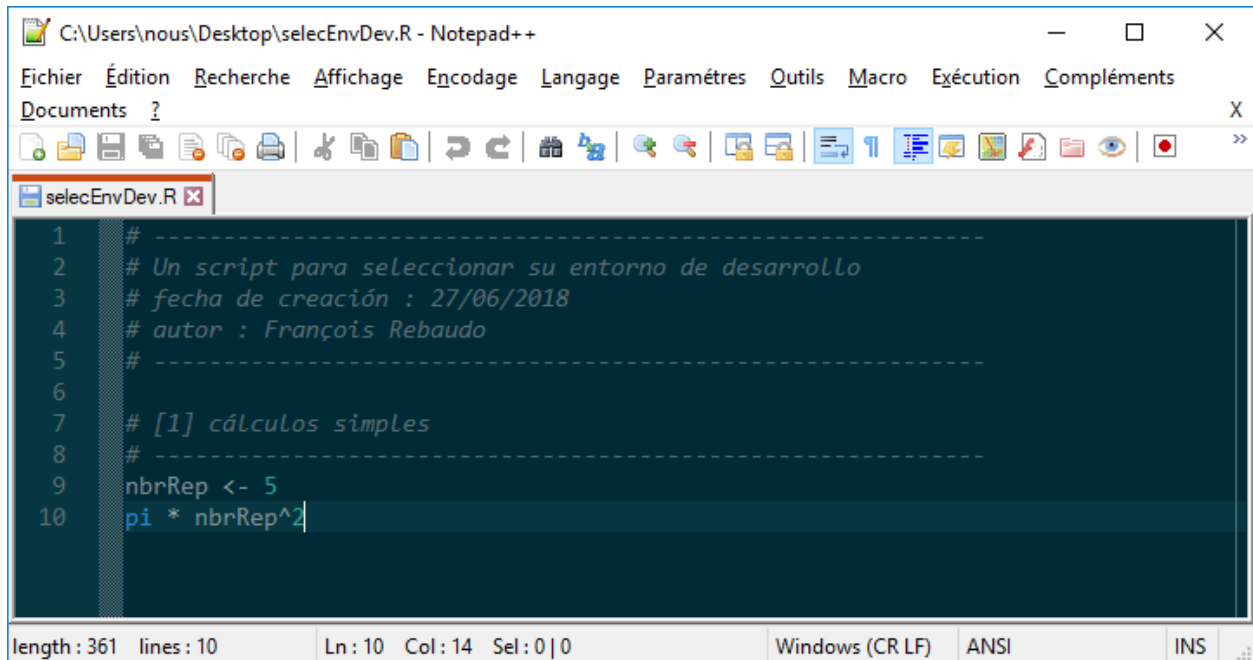
This is a session spawned by NppToR.

> # -----$
> # Un script para seleccionar su entorno de desarrollo
> # fecha de creación : 27/06/2018
> # autor : François Rebaudo
> # -----$
> # [1] cálculos simples
> # -----$
> nbrRep <- 5
> pi * nbrRep^2
[1] 78.53982
> |
```

The right window is 'C:\Users\nous\Desktop\selecEnvDev.R - Notepad++' showing a script file with the following content:

```
1 # -----$
2 # Un script para seleccionar su entorno de desarrollo
3 # fecha de creación : 27/06/2018
4 # autor : François Rebaudo
5 # -----$
6
7 # [1] cálculos simples
8 # -----$
9 nbrRep <- 5
10 pi * nbrRep^2
```

Figure 6.8: Capture d'écran de Notepad++ sous Windows : la console avec F8.



The screenshot shows a single window 'C:\Users\nous\Desktop\selecEnvDev.R - Notepad++' displaying the same script file as in Figure 6.8, but with syntax highlighting. The text is as follows:

```
1 # -----$
2 # Un script para seleccionar su entorno de desarrollo
3 # fecha de creación : 27/06/2018
4 # autor : François Rebaudo
5 # -----$
6
7 # [1] cálculos simples
8 # -----$
9 nbrRep <- 5
10 pi * nbrRep^2
```

Figure 6.9: Capture d'écran de Notepad++ sous Windows : coloration syntaxique avec le thème Solarized.



Figure 6.10: Logo Geany

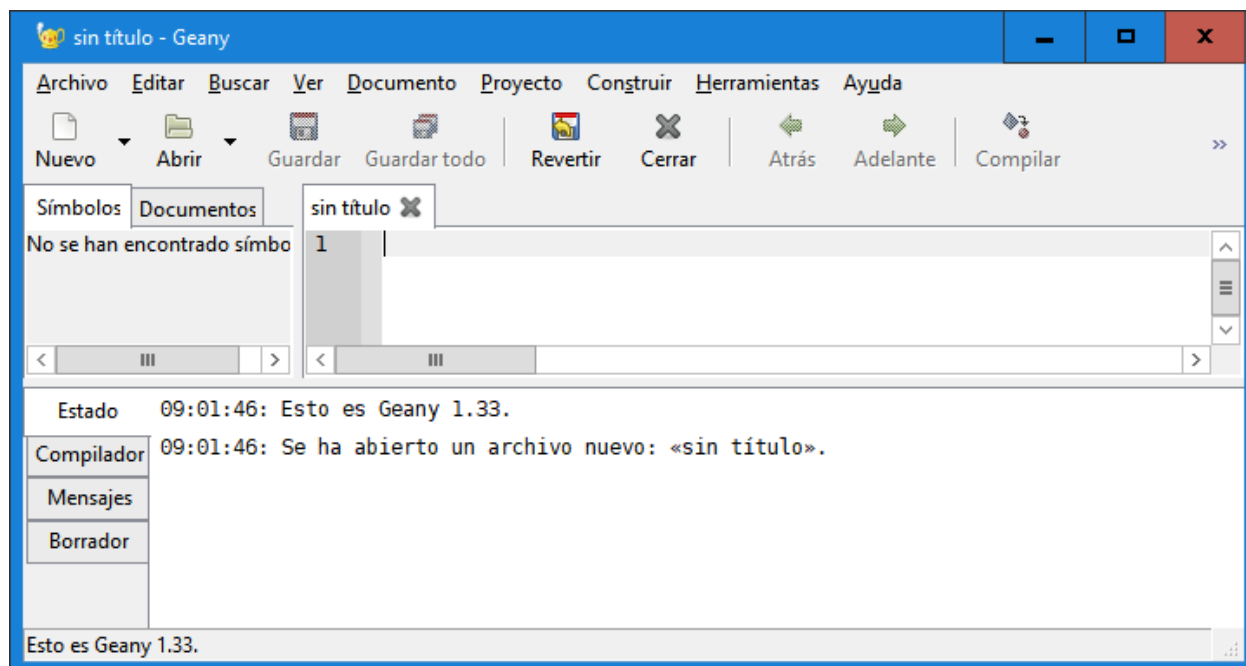


Figure 6.11: Capture d'écran de Geany sous Windows : fenêtre par défaut.

6.4 Geany (pour Linux, Mac OSX et Windows)

6.4.1 Installer Geany

Le programme pour l'installation de Geany se trouve sous l'onglet *Downloads* dans le menu de gauche *Releases* de la page web (<https://www.geany.org/>). Ensuite il suffit de télécharger l'exécutable pour Windows ou le dmg pour Mac OSX. Les utilisateurs de Linux préféreront un `sudo apt-get install geany`.

6.4.2 Un script avec Geany

Lors de la première ouverture, comme pour RStudio et Notepad++, un fichier vide est créé (Figure 6.11).

Nous pouvons ouvrir notre script avec *Fichier, Ouvrir* (Figure 6.12).

Pour exécuter notre script, la version de Geany pour Windows ne dispose pas d'un terminal intégré, ce qui rend son utilisation limitée sous ce système d'exploitation. L'exécution d'un script peut se faire en ouvrant R dans une fenêtre à part et en copiant et collant les lignes à exécuter. Sous Linux et Mac OSX, il suffit d'ouvrir R dans le terminal situé dans la partie basse de la fenêtre de Geany avec la commande `R`. Nous pouvons ensuite paramétrer Geany pour qu'une combinaison de touches permette d'exécuter le code sélectionné (par exemple `CTRL + R`). Pour cela il faut tout d'abord autoriser l'envoi de sélection vers le terminal (`send_selection_unsafe=true`) dans le fichier `geany.conf` puis choisir la commande d'envoi vers le terminal (dans *Editar, Preferencias, Combinaciones*). Pour changer le thème de Geany, il existe une collection de thèmes accessibles sur GitHub (<https://github.com/geany/geany-themes/>). Le thème peut ensuite être changé via le menu *Ver, Cambiar esquema de color...* (un exemple avec le thème *Solarized*, Figure 6.13).

6.5 Autres solutions

Il existe beaucoup d'autres solutions, certaines spécialisées pour R comme **Tinn-R** (<https://sourceforge.net/projects/tinn-r/>), et d'autres plus généralistes pour la programmation comme **Atom** (<https://atom.io/>),

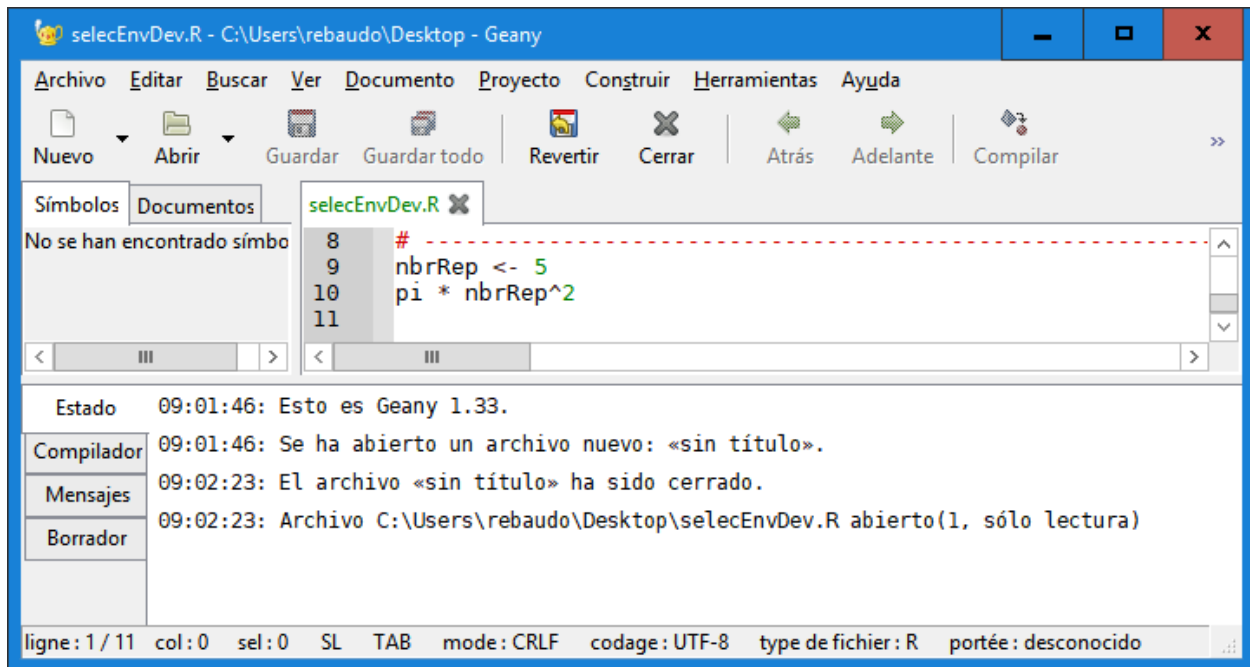


Figure 6.12: Capture d'écran de Geany sous Windows : ouvrir un script.

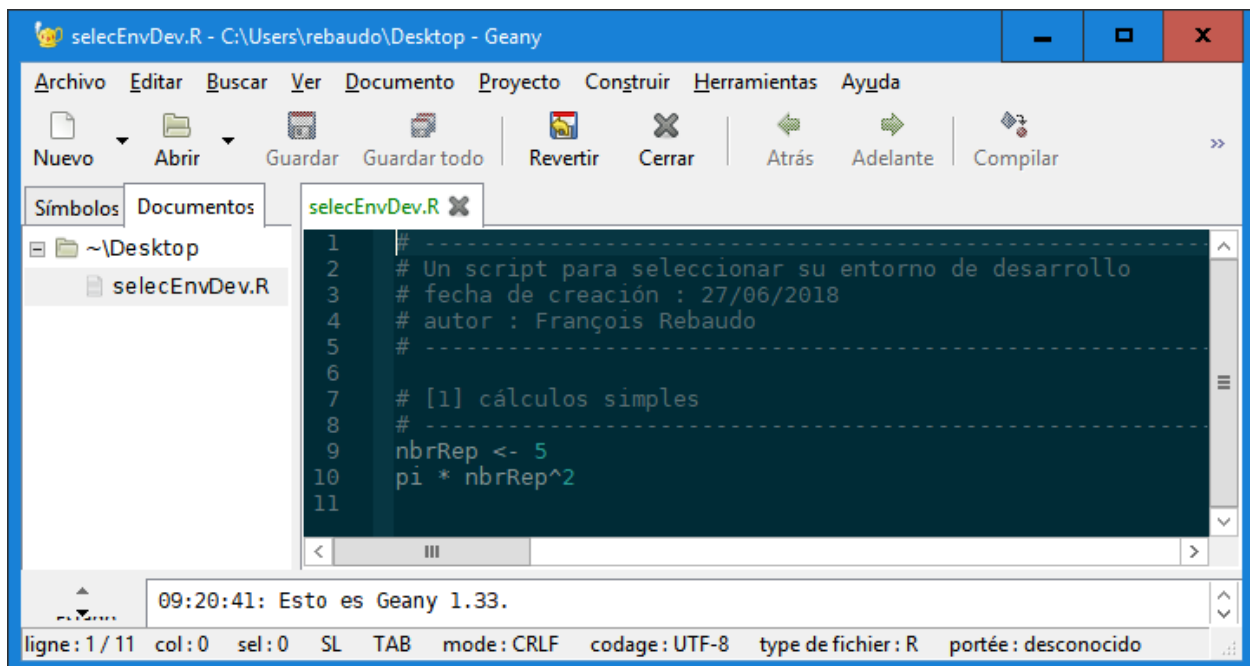


Figure 6.13: Capture d'écran de Geany sous Windows : changer les paramètres de coloration syntaxique.

Sublime Text (<https://www.sublimetext.com/>), **Vim** (<https://www.vim.org/>), **Gedit** (<https://wiki.gnome.org/Apps/Gedit>), **GNU Emacs** (<https://www.gnu.org/software/emacs/>), ou encore **Brackets** (<http://brackets.io/>) et **Eclipse** (<http://www.eclipse.org/>).

6.6 Conclusion

Félicitations, nous sommes arrivés au bout de ce chapitre sur environnements de développement pour utiliser R. Nous savons désormais :

- Installer RStudio, Geany ou Notepad++
- Reconnaître et choisir notre environnement préféré

A partir d'ici nous allons pouvoir nous concentrer sur le langage de programmation R dans un environnement facilitant le travail de lecture et d'écriture du code. C'est un grand pas en avant pour maîtriser R.

Chapter 7

Les types de données

Nous avons vu précédemment comment créer un objet. Un objet est comme une boîte dans laquelle nous allons *stocker* de l'information. Jusqu'à présent nous n'avons stocké que des nombres mais dans ce chapitre nous allons voir qu'il est possible de stocker d'autres informations et nous allons nous attarder sur les types les plus courants. Dans ce chapitre nous allons utiliser des **fonctions** sur lesquelles nous reviendrons plus tard.

7.1 Le type `numeric`

Le type `numeric` correspond à ce que nous avons fait jusqu'à présent, stocker des nombres. Il existe deux principaux types de nombres avec R: les nombres entiers (*integers*), et les nombres à virgule (*double*). Par défaut R considère tous les nombres comme des nombres à virgule et attribue le type `double`. Pour vérifier le type de données nous allons utiliser la fonction `typeof()` qui prend comme argument un objet (ou directement l'information que nous souhaitons tester). Nous pouvons également utiliser la fonction `is.double()` qui va renvoyer `TRUE` si le nombre est au format `double` et `FALSE` dans le cas contraire. La fonction générique `is.numeric()` va quant à elle renvoyer `TRUE` si l'objet est au format `numeric` et `FALSE` dans le cas contraire.

```
nbrRep <- 5
typeof(nbrRep)
```

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

```
typeof(5.32)
```

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

```
is.numeric(5)
```

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

```
is.double(5)
```

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

Si nous voulons spécifier à R que nous allons travailler avec un nombre entier, alors il nous faut transformer notre nombre à virgule en nombre entier avec la fonction `as.integer()`. Nous pouvons également utiliser la fonction `is.integer()` qui va renvoyer `TRUE` si le nombre est au format `integer` et `FALSE` dans le cas contraire.

```
nbrRep <- as.integer(5)
typeof(nbrRep)
```

```
## [1] "integer"
```

```
typeof(5.32)
```

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

```
typeof(as.integer(5.32))
```

```
## [1] "integer"
```

```
as.integer(5.32)
```

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

```
as.integer(5.99)
```

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

```
is.numeric(nbrRep)
```

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

Nous voyons ici que transformer un nombre comme 5.99 au format `integer` va renvoyer uniquement la partie entière, soit 5.

```
is.integer(5)
```

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

```
is.numeric(5)
```

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

```
is.integer(as.integer(5))
```

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

```
is.numeric(as.integer(5))
```

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

La somme d'un nombre entier et d'un nombre à virgule renvoie un nombre à virgule.

```
sumIntDou <- as.integer(5) + 5.2
typeof(sumIntDou)
```

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

```
sumIntInt <- as.integer(5) + as.integer(5)
typeof(sumIntInt)
```

```
## [1] "integer"
```

Pour résumer, le type `numeric` contient deux sous-types, les types `integer` pour les nombres entiers et le type `double` pour les nombres à virgule. Par défaut R attribue le type `double` aux nombres.

7.2 Le type character

Le type `character` correspond au texte. En effet, R permet de travailler avec du texte. Pour spécifier à R que l'information contenue dans un objet est au format texte (ou de manière générale pour tous les textes), il faut utiliser les guillemets doubles (`"`), ou simples (`'`).

```
myText <- "azerty"
myText2 <- 'azerty'
myText3 <- 'azerty uiop qsdg hjklm'
typeof(myText3)
```

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

Les guillemets doubles ou simples sont utiles si l'on souhaite mettre des guillemets dans notre texte. Nous pouvons également *échapper* un caractère spécial comme un guillemet grâce au signe backslash `\`.

```
myText <- "a 'ze' 'rt' y"
myText2 <- 'a "zert" y'
myText3 <- 'azerty uiop qsdg hjklm'
myText4 <- "qwerty \" azerty "
myText5 <- "qwerty \\ azerty "
```

Par défaut lorsque nous créons un objet, son contenu n'est pas renvoyé par la console. Sur Internet ou dans de nombreux ouvrages nous pouvons retrouver le nom de l'objet sur une ligne pour renvoyer son contenu :

```
myText <- "a 'ze' 'rt' y"
myText
```

```
## [1] "a 'ze' 'rt' y"
```

Dans ce livre nous n'utiliserons jamais cette façon de faire et préférons l'utilisation de la fonction `print()`, qui permet d'afficher dans la console le contenu d'un objet. Le résultat est le même mais le code est alors plus facile à lire et plus explicite sur ce qui est fait.

```
myText <- "a 'ze' 'rt' y"
print(myText)
```

```
## [1] "a 'ze' 'rt' y"
```

```
nbrRep <- 5
print(nbrRep)
```

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

Nous pouvons également mettre des chiffres au format texte, mais il ne faut pas oublier de mettre des guillemets pour spécifier le type `character` ou utiliser la fonction `as.character()`. Une opération entre du texte et un nombre renvoie une erreur. Par exemple si l'on ajoute 10 à "5", R nous signale qu'un **argument** de la **fonction** `+` n'est pas de type `numeric` et que donc l'opération n'est pas possible. Nous ne pouvons pas non plus ajouter du texte à du texte, mais verrons plus tard comment *concaténer* deux chaînes de texte.

```
myText <- "qwerty"
typeof(myText)
```

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

```
myText2 <- 5
typeof(myText2)
```

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

```
myText3 <- "5"
typeof(myText3)
```

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

```
myText2 + 10
```

```
## [1] 15
```

```
as.character(5)
```

```
## [1] "5"
```

```
# myText3 + 10 # Error in myText3 + 10 : non-numeric argument to binary operator
# "a" + "b" # Error in "a" + "b" : non-numeric argument to binary operator
```

Pour résumer, le type `character` permet la saisie de texte, nous pouvons le reconnaître grâce aux guillemets simples ou doubles.

7.3 Le type factor

Le type `factor` correspond aux facteurs. Les facteurs sont un choix parmi une liste finie de possibilités. Par exemple les pays sont des facteurs car il y a une liste finie de pays dans le monde à un temps donné. Un facteur peut être défini avec la fonction `factor()` ou transformé en utilisant la fonction `as.factor()`. Comme pour les autres types de donnée nous pouvons utiliser la fonction `is.factor()` pour vérifier le type de donnée. Pour avoir la liste de toutes les possibilités, il existe la fonction `levels()` (cette fonction prendra plus de sens quand nous aurons abordé les types de conteneur de l'information).

```
factor01 <- factor("aaa")
print(factor01)
```

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

```
## Levels: aaa
```



```
typeof(factor01)
```

```
## [1] "integer"
```

```
is.factor(factor01)
```

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

```
levels(factor01)
```

```
## [1] "aaa"
```

Un facteur peut être transformé en texte avec la fonction `as.character()` mais également en nombre avec `as.numeric()`. Lors de la transformation en nombre chaque facteur prend la valeur de sa position dans la liste des possibilités. Dans notre cas il n'y a qu'une seule possibilité donc la fonction `as.numeric()` va renvoyer 1:

```
factor01 <- factor("aaa")  
as.character(factor01)
```

```
## [1] "aaa"
```

```
as.numeric(factor01)
```

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

7.4 Le type logical

Le type `logical` correspond aux valeurs `TRUE` et `FALSE` (et `NA`) que nous avons déjà vu avec les opérateurs de comparaison.

```
aLogic <- TRUE  
print(aLogic)
```

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

```
typeof(aLogic)
```

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

```
is.logical(aLogic)
```

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

```
aLogic + 1
```

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

```
as.numeric(aLogic)
```

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

```
as.character(aLogic)
```

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

7.5 A propos de NA

La valeur NA peut être utilisée pour spécifier l'absence de données ou les données manquantes. Par défaut NA est de type `logical` mais il peut être utilisé pour du texte, ou des nombres.

```
print(NA)
```

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

```
typeof(NA)
```

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

```
typeof(as.integer(NA))
```

```
## [1] "integer"
```

```
typeof(as.character(NA))
```

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

```
NA == TRUE
```

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

```
NA == FALSE
```

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

```
NA > 1
```

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

```
NA + 1
```

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

7.6 Conclusion

Félicitations, nous sommes arrivés au bout de ce chapitre sur les type de données. Nous savons désormais :

- Reconnaître et faire des objets dans les principaux types de données
- Transformer les types de données d'un type à un autre

Ce chapitre un peu fastidieux est la base pour aborder le prochain chapitre sur les conteneurs des données.

Chapter 8

Les conteneurs de données

Jusqu'à présent nous avons fait des objets simples ne contenant qu'une seule valeur. Nous avons néanmoins pu voir qu'un objet avait différents attributs, comme sa valeur, mais aussi le type de donnée contenue. maintenant nous allons voir qu'il existe différents types de conteneurs permettant de stocker plusieurs données.

8.1 Le conteneur vector

Dans R, un `vector` est une combinaison de données avec la particularité que toutes les données contenues dans un `vector` sont du même type. Nous pouvons donc stocker plusieurs `numeric` ou `character` dans un `vector`, mais pas les deux. Le conteneur `vector` est important car c'est l'élément de base de R.

8.1.1 Créer un vector

Pour créer un `vector` nous allons utiliser la fonction `c()` qui permet de combiner des éléments en un `vector`. Les éléments à combiner doivent être séparés par des virgules.

```
miVec01 <- c(1, 2, 3, 4) # un vecteur de 4 éléments de type numeric ; double
print(miVec01)
```

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

```
typeof(miVec01)
```

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

```
is.vector(miVec01)
```

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

La fonction `is.vector()` permet de vérifier le type de conteneur.

```
miVec02 <- c("a", "b", "c")
print(miVec02)
```

```
## [1] "a" "b" "c"
```

```
typeof(miVec02)
```

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

```
is.vector(miVec02)
```

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

```
miVec03 <- c(TRUE, FALSE, FALSE, TRUE)  
print(miVec03)
```

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

```
typeof(miVec03)
```

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

```
is.vector(miVec03)
```

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

```
miVecNA <- c(1, NA, 3, NA, 5)  
print(miVecNA)
```

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

```
typeof(miVecNA)
```

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

```
is.vector(miVecNA)
```

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

```
miVec04 <- c(1, "a")  
print(miVec04)
```

```
## [1] "1" "a"
```

```
typeof(miVec04)
```

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

```
is.vector(miVec04)
```

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

Si l'on combine différents types de données, par défaut R va chercher à transformer les éléments en un seul type. Si comme ici dans l'objet `miVec03` nous avons des `character` et des `numeric`, R va transformer tous les éléments en `character`.

```
miVec05 <- c(factor("abc"), "def")
print(miVec05)
```

```
## [1] "1"    "def"
```

```
typeof(miVec05)
```

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

```
miVec06 <- c(TRUE, "def")
print(miVec06)
```

```
## [1] "TRUE" "def"
```

```
typeof(miVec06)
```

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

```
miVec07 <- c(factor("abc"), 55)
print(miVec07)
```

```
## [1] 1 55
```

```
typeof(miVec07)
```

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

```
miVec08 <- c(TRUE, 55)
print(miVec08)
```

```
## [1] 1 55
```

```
typeof(miVec08)
```

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

Nous pouvons aussi combiner des objets existants au sein d'un vector.

```
miVec09 <- c(miVec02, "d", "e", "f")
print(miVec09)
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
miVec10 <- c("aaa", "aa", miVec09, "d", "e", "f")
print(miVec10)
```

```
## [1] "aaa" "aa"  "a"   "b"   "c"   "d"   "e"   "f"   "d"   "e"   "f"
```

```
miVec11 <- c(789, miVec01, 564)
print(miVec11)
```

```
## [1] 789 1 2 3 4 564
```

8.1.2 Opérations sur un vector

Nous pouvons également effectuer des opérations sur un vector.

```
print(miVec01)
```

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

```
miVec01 + 1
```

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

```
miVec01 - 1
```

```
## [1] 0 1 2 3
```

```
miVec01 * 2
```

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

```
miVec01 /10
```

```
## [1] 0.1 0.2 0.3 0.4
```

Les opérations d'un vector sur un autre sont aussi possibles, mais il faut veiller à ce que le nombre d'éléments d'un vector soit le même que l'autre, sinon R va effectuer le calcul en repartant du début. Voici un exemple pour illustrer ce que R fait:

```
miVec12 <- c(1, 1, 1, 1, 1, 1, 1, 1)
print(miVec12)
```

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

```
miVec13 <- c(10, 20, 30)
print(miVec13)
```

```
## [1] 10 20 30
```

```
miVec12 + miVec13 # vecteurs de tailles différentes : attention au résultat
```

```
## [1] 11 21 31 11 21 31 11 21 31
```

```
miVec14 <- c(10, 20, 30, 40, 50, 60, 70, 80, 90)
print(miVec14)
```

```
## [1] 10 20 30 40 50 60 70 80 90
```

```
miVec12 + miVec14 # les vecteurs sont de la même longueur
```

```
## [1] 11 21 31 41 51 61 71 81 91
```

```
miVec15 <- c(1, 1, 1, 1)
print(miVec15)
```

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

```
miVec15 + miVec13 # vecteurs de tailles différentes et non multiples
```

```
## Warning in miVec15 + miVec13: la taille d'un objet plus long n'est pas
## multiple de la taille d'un objet plus court
```

```
## [1] 11 21 31 11
```

8.1.3 Accéder aux valeurs d'un vector

Il est souvent nécessaire de pouvoir accéder aux valeurs d'un vector, c'est à dire de récupérer une valeur ou un groupe de valeurs au sein d'un vector. Pour accéder à un élément dans un vector nous utilisons les crochets []. Entre les crochets, nous pouvons utiliser un numéro correspondant au numéro de l'élément dans le vector.

```
miVec20 <- c(10, 20, 30, 40, 50, 60, 70, 80, 90)
miVec21 <- c("a", "b", "c", "d", "e", "f", "g", "h", "i")
print(miVec20)
```

```
## [1] 10 20 30 40 50 60 70 80 90
```

```
print(miVec21)
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

```
print(miVec20[1])
```

```
## [1] 10
```

```
print(miVec21[3])
```

```
## [1] "c"
```

Nous pouvons aussi utiliser la combinaison de différents éléments (un autre vector).

```
print(miVec20[c(1, 5, 9)])
```

```
## [1] 10 50 90
```

```
print(miVec21[c(4, 3, 1)])
```

```
## [1] "d" "c" "a"
```

```
print(miVec21[c(4, 4, 3, 4, 3, 2, 5)])
```

```
## [1] "d" "d" "c" "d" "c" "b" "e"
```

Nous pouvons aussi sélectionner des éléments en utilisant un opérateur de comparaison ou un opérateur logique.

```
print(miVec20[miVec20 >= 50])
```

```
## [1] 50 60 70 80 90
```

```
print(miVec20[(miVec20 >= 50) & ((miVec20 < 80))])
```

```
## [1] 50 60 70
```

```
print(miVec20[miVec20 != 50])
```

```
## [1] 10 20 30 40 60 70 80 90
```

```
print(miVec20[miVec20 == 30])
```

```
## [1] 30
```

```
print(miVec20[(miVec20 == 30) | (miVec20 == 50)])
```

```
## [1] 30 50
```

```
print(miVec21[miVec21 == "a"])
```

```
## [1] "a"
```

Une autre fonctionnalité intéressante est de conditionner les éléments à sélectionner dans un `vector` en fonction d'un autre `vector`.

```
print(miVec21[miVec20 >= 50])
```

```
## [1] "e" "f" "g" "h" "i"
```

```
print(miVec21[(miVec20 >= 50) & ((miVec20 < 80))])
```

```
## [1] "e" "f" "g"
```

```
print(miVec21[miVec20 != 50])
```

```
## [1] "a" "b" "c" "d" "f" "g" "h" "i"
```

```
print(miVec21[miVec20 == 30])
```

```
## [1] "c"
```

```
print(miVec21[(miVec20 == 30) | (miVec20 == 50)])
```

```
## [1] "c" "e"
```



```
print(miVec21[(miVec20 == 30) | (miVec21 == "h")])
```

```
## [1] "c" "h"
```

Il est aussi possible d'exclure certains éléments plutôt que de les sélectionner.

```
print(miVec20[-1])
```

```
## [1] 20 30 40 50 60 70 80 90
```

```
print(miVec21[-5])
```

```
## [1] "a" "b" "c" "d" "f" "g" "h" "i"
```

```
print(miVec20[-c(1, 2, 5)])
```

```
## [1] 30 40 60 70 80 90
```

```
print(miVec21[-c(1, 2, 5)])
```

```
## [1] "c" "d" "f" "g" "h" "i"
```

Les éléments d'un vector peuvent aussi être sélectionné sur la base d'un vector de type `logical`. Dans ce cas seuls les éléments avec une valeur `TRUE` seront sélectionnés.

```
miVec22 <- c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, TRUE)
print(miVec21[miVec22])
```

```
## [1] "a" "b" "d" "f" "h" "i"
```

8.1.4 Nommer les éléments d'un vector

Les éléments d'un vector peuvent être nommé pour pouvoir s'y référer par la suite et opérer une sélection. La fonction `names()` permet de récupérer les noms des éléments d'un vecteur.

```
miVec23 <- c(aaa = 10, bbb = 20, ccc = 30, ddd = 40, eee = 50)
print(miVec23)
```

```
## aaa bbb ccc ddd eee
## 10 20 30 40 50
```

```
print(miVec23["bbb"])
```

```
## bbb
## 20
```

```
print(miVec23[c("bbb", "ccc", "bbb")])
```

```
## bbb ccc bbb
## 20 30 20
```

```
names(miVec23)
```

```
## [1] "aaa" "bbb" "ccc" "ddd" "eee"
```

8.1.5 Modifier les éléments d'un vector

Pour modifier un vecteur, nous opérons de la même façon que pour modifier un objet simple, avec le signe `<-` et l'élément ou les éléments à modifier entre crochets.

```
print(miVec21)
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

```
miVec21[3] <- "zzz"
print(miVec21)
```

```
## [1] "a" "b" "zzz" "d" "e" "f" "g" "h" "i"
```

```
miVec21[(miVec20 >= 50) & ((miVec20 < 80))] <- "qwerty"
print(miVec21)
```

```
## [1] "a" "b" "zzz" "d" "qwerty" "qwerty" "qwerty" "h"
## [9] "i"
```

```
print(miVec23)
```

```
## aaa bbb ccc ddd eee
## 10 20 30 40 50
```

```
miVec23["ccc"] <- miVec23["ccc"] + 100
print(miVec23)
```

```
## aaa bbb ccc ddd eee
## 10 20 130 40 50
```

Nous pouvons aussi changer les noms associés aux éléments d'un vector.

```
print(miVec23)
```

```
## aaa bbb ccc ddd eee
## 10 20 130 40 50
```

```
names(miVec23)[2] <- "bb_bb"
print(miVec23)
```

```
##   aaa bb_bb   ccc   ddd   eee
##   10   20  130   40   50
```

Nous pouvons faire bien plus avec un vector et reviendrons sur leur manipulations et les opérations lors du chapitre sur les fonctions.

8.2 Le conteneur list

Le deuxième type de conteneur que nous allons introduire est le conteneur `list`, qui est également le deuxième conteneur après le type `vector` de part son importance dans la programmation avec R. Le conteneur de type `list` permet de stocker une **liste** d'éléments. Contrairement à ce que nous avons vu précédemment avec le type `vector`, les éléments du type `list` peuvent être différents (par exemple un `vector` de type `numeric`, puis un vecteur de type `character`). Les éléments du type `list` peuvent aussi être des conteneurs différents (par exemple un `vector`, puis une `list`). Le type de conteneur `list` prendra tout son sens lorsque nous aurons étudié les **boucles** et les **fonctions** de la famille `apply`.

8.2.1 Créer une list

Pour créer une `list` nous allons utiliser la fonction `list()` qui prend comme argument des éléments (objets).

```
miList01 <- list()
print(miList01)
```

```
## list()
```

```
miList02 <- list(5, "qwerty", c(4, 5, 6), c("a", "b", "c"))
print(miList02)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c"
```

```
miList03 <- list(5, "qwerty", list(c(4, 5, 6), c("a", "b", "c")))
print(miList03)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [[3]][[1]]
## [1] 4 5 6
##
## [[3]][[2]]
## [1] "a" "b" "c"
```

La fonction `is.list()` permet de tester si nous avons bien créer un objet de type `list`.

```
is.list(miList02)
```

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

```
typeof(miList02)
```

```
## [1] "list"
```

8.2.2 Accéder aux valeurs d'une list

Les éléments du conteneur `list` sont identifiables grâce aux double crochets `[]`.

```
print(miList02)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c"
```

Dans l'objet `miList02` de type `list`, il y a quatre éléments identifiables avec `[[1]]`, `[[2]]`, `[[3]]`, et `[[4]]`. Chacun des éléments est de type `vector` de taille 1 et de type `double` pour le premier élément, de taille 1 et de type `character` pour le deuxième élément, de taille 3 et de type `double` pour le troisième élément, et de taille 3 et de type `character` pour le quatrième élément.

```
typeof(miList02)
```

```
## [1] "list"
```

```
print(miList02[[1]])
```

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

```
typeof(miList02[[1]])
```

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

```
print(miList02[[2]])
```

```
## [1] "qwerty"
```

```
typeof(miList02[[2]])
```

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

```
print(miList02[[3]])
```

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

```
typeof(miList02[[3]])
```

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

```
print(miList02[[4]])
```

```
## [1] "a" "b" "c"
```

```
typeof(miList02[[4]])
```

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

L'accès au deuxième élément du vector situé en quatrième position de la liste se fait donc avec `miList02[[4]][2]`. Nous utilisons un double crochet pour le quatrième élément de la liste, puis un simple crochet pour le deuxième élément du vector.

```
print(miList02[[4]][2])
```

```
## [1] "b"
```

Comme une liste peut contenir elle-même une ou plusieurs listes, nous pouvons accéder à l'information recherchée en combinant les doubles crochets. L'objet `miList04` est une liste de deux éléments, les listes `miList02` et `miList03`. L'objet `miList03` contient lui-même une liste comme élément en troisième position. Pour accéder au premier élément du vector en première position de l'élément en troisième position du deuxième élément de la liste `miList04`, nous pouvons utiliser `miList04[[2]][[3]][[1]][1]`. Il n'y a pas de limite quant à la profondeur des listes mais dans la pratique il n'y a que rarement besoin de faire des listes de listes de listes.

```
miList04 <- list(miList02, miList03)
print(miList04)
```

```
## [[1]]
## [[1]][[1]]
## [1] 5
##
## [[1]][[2]]
## [1] "qwerty"
##
## [[1]][[3]]
## [1] 4 5 6
##
## [[1]][[4]]
## [1] "a" "b" "c"
##
##
## [[2]]
## [[2]][[1]]
## [1] 5
##
## [[2]][[2]]
```

```
## [1] "qwerty"
##
## [[2]][[3]]
## [[2]][[3]][[1]]
## [1] 4 5 6
##
## [[2]][[3]][[2]]
## [1] "a" "b" "c"
```

```
print(miList04[[2]][[3]][[1]][1])
```

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

Pour rendre concret l'exemple précédent, nous pouvons imaginer des espèces de foreurs de maïs (*Sesamia nonagrioides* et *Ostrinia nubilalis*), échantillonnées dans différents sites, avec différentes abondances à quatre dates. Ici nous allons donner des noms aux éléments des listes.

```
bddInsect <- list(Snonagrioides = list(site01 = c(12, 5, 8, 7), site02 = c(5, 23, 4, 41), site03 = c(12, 0, 0, 0)),
  Onubilalis = list(site01 = c(12, 1, 2, 3), site02 = c(0, 0, 0, 1), site03 = c(1, 1, 2, 3)))
print(bddInsect)
```

```
## $Snonagrioides
## $Snonagrioides$site01
## [1] 12 5 8 7
##
## $Snonagrioides$site02
## [1] 5 23 4 41
##
## $Snonagrioides$site03
## [1] 12 0 0 0
##
##
## $Onubilalis
## $Onubilalis$site01
## [1] 12 1 2 3
##
## $Onubilalis$site02
## [1] 0 0 0 1
##
## $Onubilalis$site03
## [1] 1 1 2 3
```



La lecture d'une ligne de code longue comme celle de la création de l'objet `bddInsect` est difficile à lire car la profondeur des éléments ne peuvent se déduire que grâce aux parenthèses. C'est pourquoi nous allons reorganiser le code pour lui donner plus de lisibilité grâce à l'**indentation**. L'indentation consiste à mettre l'information à des niveaux différents de telle manière que nous puissions rapidement identifier les différents niveaux d'un code. L'indentation se fait au moyen de la touche de tabulation du clavier. Nous reviendrons sur l'indentation avec plus de précisions lors du chapitre sur les **boucles**. Nous retiendrons pour le moment que si une ligne de code est trop longue, nous gagnons en lisibilité en passant à la ligne et que R va lire l'ensemble comme une seule ligne de code.

```
bddInsect <- list(
  Snonagrioides = list(
    site01 = c(12, 5, 8, 7),
    site02 = c(5, 23, 4, 41),
```

```

    site03 = c(12, 0, 0, 0)
  ),
  Onubilalis = list(
    site01 = c(12, 1, 2, 3),
    site02 = c(0, 0, 0, 1),
    site03 = c(1, 1, 2, 3)
  )
)

```

Nous pouvons sélectionner les données d'abondance du deuxième site de la première espèce comme précédemment `bddInsect[[1]][[2]]`, ou alternativement en utilisant les noms des éléments `bddInsect$Snonagrioides$site02`. Pour ce faire nous utilisons le signe `$`, ou alors le nom des éléments avec des guillemets simples ou doubles `bddInsect[['Snonagrioides']][[2]]`.

```
print(bddInsect[[1]][[2]])
```

```
## [1] 5 23 4 41
```

```
print(bddInsect$Snonagrioides$site02)
```

```
## [1] 5 23 4 41
```

```
print(bddInsect[['Snonagrioides']][['site02']])
```

```
## [1] 5 23 4 41
```

Comme pour les vecteurs nous pouvons récupérer les noms des éléments avec la fonction `names()`.

```
names(bddInsect)
```

```
## [1] "Snonagrioides" "Onubilalis"
```

```
names(bddInsect[[1]])
```

```
## [1] "site01" "site02" "site03"
```

Lorsque nous utilisons les doubles crochets `[[]]` ou le signe `$`, R renvoie le contenu de l'élément sélectionné. Dans notre exemple les données d'abondance sont contenues sous la forme d'un `vector`, donc R renvoie un élément de type `vector`. Si nous souhaitons sélectionner un élément d'une `list` mais en conservant le format `list`, alors nous pouvons utiliser les crochets simples `[]`.

```
print(bddInsect[[1]][[2]])
```

```
## [1] 5 23 4 41
```

```
typeof(bddInsect[[1]][[2]])
```

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

```
is.list(bddInsect[[1]][[2]])
```

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

```
print(bddInsect[[1]][2])
```

```
## $site02
## [1]  5 23  4 41
```

```
typeof(bddInsect[[1]][2])
```

```
## [1] "list"
```

```
is.list(bddInsect[[1]][2])
```

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

L'utilisation des crochets simples `[]` est utile lorsque nous souhaitons récupérer plusieurs éléments d'une liste. Par exemple pour sélectionner les abondances d'insectes des deux premiers sites de la première espèce, nous utiliserons `bddInsect[[1]][c(1, 2)]` ou alternativement `bddInsect[[1]][c("site01", "site02")]`.

```
print(bddInsect[[1]][c(1, 2)])
```

```
## $site01
## [1] 12  5  8  7
##
## $site02
## [1]  5 23  4 41
```

```
print(bddInsect[[1]][c("site01", "site02")])
```

```
## $site01
## [1] 12  5  8  7
##
## $site02
## [1]  5 23  4 41
```

8.2.3 Modification d'une list

Une liste peut être modifiée de la même façon que pour le conteneur `vector`, c'est à dire en se référant avec des crochets à l'élément que nous souhaitons modifier.

```
print(miList02)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c"
```



```
miList02[[1]] <- 12
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c"
```

```
miList02[[4]] <- c("d", "e", "f")
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "d" "e" "f"
```

```
miList02[[4]] <- c("a", "b", "c", miList02[[4]], "g", "h", "i")
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

```
miList02[[4]][5] <- "eee"
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
```

```
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"
```

```
miList02[[3]] <- miList02[[3]] * 10 - 1
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 39 49 59
##
## [[4]]
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"
```

```
miList02[[3]][2] <- miList02[[1]] * 100
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 39 1200 59
##
## [[4]]
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"
```

```
print(bddInsect)
```

```
## $Snonagrioides
## $Snonagrioides$site01
## [1] 12 5 8 7
##
## $Snonagrioides$site02
## [1] 5 23 4 41
##
## $Snonagrioides$site03
## [1] 12 0 0 0
##
##
## $Onubilalis
## $Onubilalis$site01
```

```
## [1] 12  1  2  3
##
## $Onubilalis$site02
## [1] 0 0 0 1
##
## $Onubilalis$site03
## [1] 1 1 2 3

bddInsect[['Snonagrioides']][['site02']] <- c(2, 4, 6, 8)
print(bddInsect)
```

```
## $Snonagrioides
## $Snonagrioides$site01
## [1] 12  5  8  7
##
## $Snonagrioides$site02
## [1] 2 4 6 8
##
## $Snonagrioides$site03
## [1] 12  0  0  0
##
##
## $Onubilalis
## $Onubilalis$site01
## [1] 12  1  2  3
##
## $Onubilalis$site02
## [1] 0 0 0 1
##
## $Onubilalis$site03
## [1] 1 1 2 3
```

Pour combiner deux list, il suffit d'utiliser la fonction `c()` que nous avons utilisée pour créer un vector.

```
miList0203 <- c(miList02, miList03)
print(miList0203)

## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 39 1200 59
##
## [[4]]
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"
##
## [[5]]
## [1] 5
##
## [[6]]
## [1] "qwerty"
```

```
##
## [[7]]
## [[7]][[1]]
## [1] 4 5 6
##
## [[7]][[2]]
## [1] "a" "b" "c"
```

Un objet de type `list` peut être transformé en `vector` avec la fonction `unlist()` si le format des éléments de la liste le permet (un `vector` ne peut contenir que des éléments du même type).

```
miList05 <- list("a", c("b", "c"), "d")
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
```

```
miVec24 <- unlist(miList05)
print(miVec24)
```

```
## [1] "a" "b" "c" "d"
```

```
miList06 <- list(c(1, 2, 3), c(4, 5, 6, 7), 8, 9, c(10, 11))
print(miList06)
```

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

```
miVec25 <- unlist(miList06)
print(miVec25)
```

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

Pour ajouter un élément à une `list`, nous pouvons utiliser la fonction `c()` ou alors les crochets doubles `[[]]`.

```
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
```

```
miList05 <- c(miList05, "e")
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
##
## [[4]]
## [1] "e"
```

```
miList05[[5]] <- c("fgh", "ijk")
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
##
## [[4]]
## [1] "e"
##
## [[5]]
## [1] "fgh" "ijk"
```

Pour supprimer un élément à une `list`, la technique la plus rapide consiste à attribuer la valeur `NULL` à l'élément à supprimer.

```
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
```

```
##  
## [[3]]  
## [1] "d"  
##  
## [[4]]  
## [1] "e"  
##  
## [[5]]  
## [1] "fgh" "ijk"
```

```
miList05[[2]] <- NULL  
print(miList05)
```

```
## [[1]]  
## [1] "a"  
##  
## [[2]]  
## [1] "d"  
##  
## [[3]]  
## [1] "e"  
##  
## [[4]]  
## [1] "fgh" "ijk"
```

8.3 Le conteneur `data.frame`

XXX

8.4 Le conteneur `matrix`

XXX

8.5 Le conteneur `array`

XXX

8.6 Plus d'information sur les objets

XXX rm ls ...

Chapter 9

Les fonctions

9.1 Quest-ce qu'une fonction

XXX

9.2 Les fonctions les plus courantes

9.2.1 Visualiser les données

str head tail class

9.2.2 fonctions mathématiques

exp sqrt

9.2.3 Statistiques descriptives

mean sd max min quartile summary meadian

9.3 Autres fonctions utiles

XXX

9.4 Ecrire une fonction

XXX

Chapter 10

Importer et exporter des données

10.1 Lire des données depuis un fichier tableur

XXX

10.2 Sauver des données pour R

save load

10.3 Exporter des données

write XXX

Chapter 11

Les boucles

11.1 Pourquoi faire des boucles

XXX un peu d'algo

11.2 La boucle if

XXX

11.3 La boucle switch

XXX

11.4 La boucle for

XXX

11.5 La boucle while

XXX

11.6 repeat, next, break, stop

XXX

11.7 Les boucles de la famille apply

11.7.1 apply

XXX

11.7.2 sapply

XXX

11.7.3 lapply

XXX

11.7.4 tapply

XXX

11.7.5 mapply

XXX