Le compagnon de l'utilisateur R

Vezy Rémi

2019-09-23

Contents

Chapter 1

Prérequis

Ce livre à pour objectif de vous enseigner le plus rapidement possible comment utiliser R et son environement (RStudio, Rmarkdown, plotly. . .) pour la science des données. Ainsi, chaque chapitre peut être lu indépendemment des autres, pour que vous puissez aller droit à l'essentiel lorsque vous en avez besoin. Bien sûr, je vous recommande fortement de lire ou au moins de survoler le livre une première fois afin de vous faire gagner du temps par la suite. Pour donner un exemple, même si l'utilisation de plotly ne demande pas explicitement de savoir utiliser les pipes (tubes), leur utilisation ne rendra l'apprentissage que plus rapide.

R peut aussi être utilisé seul mais je vous recommande fortement l'utilisation de l'IDE RStudio car il dispose d'un large panel d'outils rendant l'utilisation de R plus facile. RStudio est gratuit et libre, et la dernière version du logiciel est disponible ici. Un tutorial d'installation sera décrit en introduction.

Ce livre est entièrement écrit grâce à RStudio et aux packages R **bookdown** [@R-bookdown] et **knitr** [@xie2015]. Vous pouvez donc vous-même télécharger **bookdown** et m'aider à l'écriture du livre en utilisant le lien Github. Pour cela, il vous suffit d'exécuter la commande suivante depuis la console **R**:

```
devtools::install_github("rstudio/bookdown")
```

Chapter 2

Introduction

2.1 Installation

Comme dit précédemment dans le prérequis 1, nous utiliserons **R** depuis **RStudio**, qui est un formidable environnement de développement. Il permet plusieurs choses qui seraient possibles mais plus fastidieuses depuis **R** seul:
* Le fenêtrage: avoir dans une même fenêtre ses scripts, la console R, les graphiques, les fichiers, les packages chargés et disponibles, l'aide, ainsi que l'environnement et l'historique des commandes
* Le travail en projet, que je vous décris plus tard
* Un débugger
* Un profiler
* Une gestion du contrôle de version (GIT/SVN)

Avant de commencer donc, il vous faudra installer R et RStudio. R est téléchargeable depuis le site du R-project. Il vous suffit de cliquer sur "download R", de choisir votre mirroir, qui est le dépôt depuis lequel vous allez le télécharger (si vous avez un doute, prenez n'importe lequel en France), et enfin de cliquer sur "Download R for [votre système d'exploitation]". Une fois téléchargé, installez-le comme n'importe quel programme.

2.2 Oui, mais c'est quoi R?

R est un language de programmation dérivé du language S. Un language de programmation est ce qui permet à l'homme (le développeur ou l'utilisateur) de communiquer avec un ordinateur. En effet, les ordinateurs communiquent avec le language binaire, qui n'est composé que de 0 et de 1. Ces 0 et 1 sont agregés par "mots" pour construire des instructions. Par exemple, `01010010` veut dire **R** en binaire. Difficile de communiquer comme ça, dans un sens comme dans l'autre. Les languages de programmation nous servent en quelque sorte de

language commun. Il en existe de nombreux, qui s'utilisent différemments. Je ne m'étendrais pas là-dessus ici, ce n'est pas le but. Mais vous pouvez retenir une chose ici: R vous sert d'outil de communication pour contrôler votre ordinateur, et lui demander d'exécuter des opérations à votre place.

Je pourrais aussi ajouter que R est un language orienté objet, et un language fonctionnel. Ce sont deux paradigmes qui sont devenus très populaires et sont implémentés dans de nombreux languages informatiques (C++, Java, Python...). Le premier tourne autour de l'idée de classes et d'objets. Prenons par exemple une machine à café. Une cafetière est appelé un `objet`, qui est de `classe` cafetière. Mais le nom cafetière est parfois imprécis employé seul, car il en existe plusieurs sortes. Il y a des machines à expresso, des cafetières Italiennes, des cafetières à percolation, des cafetières à piston et j'en passe. Toutes représentent le même `objet` de `classe` cafetière. On les différenciera donc par une sous-classe -le type de cafetière-, qui partagera les attribus de la classe supérieure "cafetière", mais qui se différencieront par d'autres attribus qu'ils n'auront pas en commun. Il existe trois différents systèmes objets orientés dans R: les S3, les S4 et les RC (Reference Classes). Ils découlent de la longue histoire du language R, qui prends ses racines en 1976, quatres ans seulement après l'invention du language orienté objet. Je décris plus en détails ces systèmes dans un autre chapitre du livre. J'ai aussi dit que R est un language fonctionnel, ce qui est entièrement lié au paradigme du language orienté objet. Il faut savoir qur R traite tout comme un objet, y compris les fonctions. Ce fonctionnement à de nombreux avantages, dont celui de pouvoir assigner une fonction à un objet, ou bien une fonction à une autre fonction. Si vous voulez avoir plus d'informations sur ce paragraphe, je vous recommande fortement de jeter un coup d'oeil au MOOC dont il est fortement inspiré, Advanced R programming de l'Université Johns-Hopkins (en anglais).

Lorsqu'on débute en informatique, il arrive très souvent de penser que la machine est bête car elle ne comprends pas ce que l'on veut lui dire. On lui envoi une commande, et elle nous renvoi une erreur ou un avertissement. Si vous oubliez une virgule ou une parenthèse, le mieux qu'il puisse vous arriver est que le language reconnaisse votre erreur et qu'il vous la dise. En effet, un ordinateur n'est bête, ni intelligent. Il ne fait qu'exécuter ce que vous lui demandez. Il vous faudra donc être plus clair que jamais dans votre communication. Et par ça, je veux dire qu'il vous faudra devenir un vrai nazi de la grammaire. Une seule faute d'orthographe, et vous appelez une variable différente de celle que vous pensiez. Une seule parenthèse manquante, et votre code risque d'éxecuter une portion qu'il ne fallait pas. J'insiste beaucoup sur cette partie car vous verrez que la communication avec un ordinateur est parfois très frustante au début. Mais il faudra vous y faire, et vous rendre compte que **vous** et vous seuls faites des erreurs, et que c'est vous qui contrôlez l'ordinateur et non pas le contraire.

2.3 Pourquoi R?

Voilà une grande question, à laquelle on peut répondre de plusieurs façons. Pour commencer, on peut se demander pourquoi utiliser un language de programmation plutôt qu'un outil a priori plus simple ? Utiliser un logiciel comme Excel ou Calc permet par exemple de visualiser ses données, effectuer des calculs, et même faire des graphiques. Alors où est l'intérêt de se compliquer la tâche pour le même résultat ? La réponse tient en deux mots: l'**automatisation**, et la **reproductibilité**.
En effet, un language de programmation permet à son utilisateur d'automatiser une tâche par la description de processus, de procédures et donc d'algorithmes par l'écriture de code informatique.
Mais rien de tel qu'un bon exemple pour étayer ses propos:

Imaginez donc que l'on vous envoi les mêmes données chaque semaine sous la forme de plusieurs fichiers texte, et que vous deviez effectuer des opérations arithmétiques sur certains chiffres, puis calculer quelques statistiques, et enfin envoyer des graphiques récapitulatif différents à Kevin de la compta, Amar du pôle communication ou Amandine votre patrone. Si vous utilisez Excel, vous devrez répéter chaque semaine la même procédure, inlassablement. D'abord vous devrez remplir votre tableur avec les données de chaque fichier, puis appliquer vos formules aux colonnes ou aux lignes concernées, faire plusieurs graphiques, et les envoyer à vos collègues. Peu de place pour l'imaginaire. Maintenant, en utilisant un language de programmation vous avez la possibilité de coder une seule fois un script qui fera tout ceci à votre place, et même bien plus, autant de fois que vous le désirez, instantanément. Ainsi, l'auteur d'un code aura une tâche bien précise en tête, qu'il s'efforcera d'écrire sous la forme d'un script qui contiendra la ou les tâches, et qu'il pourra réexécuter autant de fois que nécessaire, sur son propre ordinateur, ou n'importe lequel. Voici qu'une machine à remplassé la pénibilité de votre travail. Vous pouvez maintenant vous concentrer sur autre chose (d'autres dossiers en attente, la dernière série Netflix, ou la paix dans le monde que sais-je?).

Oui mais pourquoi R ? N'existe-t-il pas d'autres languages ?
Si, bien d'autres. Il existe des languages pour à peu près tout. Il y a des languages généralistes comme C++, FORTRAN, JAVA ou encore PERL, des languages spécialisés comme HTML ou CSS, des languages compilés, des languages interprétés... Mais les languages qui nous intéressent ici sont les languages pour la science des données. Et sur ce créneau c'est la même chose, le choix est très large: SAS, Pyhton, R, Matlab, STATISTICA... Alors, pourquoi R ?? D'abord, R est gratuit, open-source, et utilisable sur Winows, Mac-OS et Linux. Ces trois arguments éliminent déjà beaucoup de concurrents sauf Pyhton, qui est considéré aujourd'hui comme la seule alternative à R. En quelques points, voici les avantages de R :

- R devient petit à petit la *lingua franca* de la science des données, poussé par une énorme communauté de développeurs, toujours grandissante.

- Le nombre de packages disponibles et la généricité du language, qui permettent de faire tout ce que l'on veut, des statistiques, du machine-learning, de l'édition web... Il y a cette phrase à propos de R que j'aime beaucoup: il n'y a pas de 'est-ce que l'on peut' avec R, mais seulement des 'comment fait-on'. Elle résume à peu près le potentiel de ce language.

- R est beaucoup utilisé par les académiciens, ce qui en fait un outil toujours à la pointe des nouveaux développements.

- La documentation et l'aide. Dans R, chaque fonction d'un package à sa propre documentation standardisée. De par sa grande communauté, il existe aussi de nombreuses ressources comme des livres, des tutoriels youtube, des questions/réponses sur internet.

- R permet de faire de belles figures, de la plus simple à la plus complexe (intéractivité, D3.js, statistiques...).

- L'interpréteur. R est un language interprété, c'est à dire qu'il n'y a pas besoin de compiler un programme pour l'exécuter. Cela peut avoir plusieurs inconvénients, comme la rapidité d'éxécution. Mais avoir la possibilité de taper sa commande dans l'intérpréteur et d'obtenir la sortie directement est un grand avantage pour pouvoir coder vite (e.g. débuger, accéder aux valeurs instantanément...) et donc d'arriver à des résultats rapidements.

- La communication. R dispose de plusieurs packages (Rmarkdown, knitr, bookdown, shiny...) permettant la production de documents automatisée intégrant des figures, du code et bien d'autres choses.

Le principal inconvénient de R est qu'il peut être lent si l'on utilise certains paradigmes pour coder, comme par exemple l'utilisation de boucles `for`, qui sont pourtant très utilisées dans d'autres languages. Ceci-dit R peut facilement être optimisé pour la rapidité si l'on connaît les bons outils. Ces outils et ses méthodes d'écritures seront détaillés dans le chapitre sur l'optimisation des codes R (mettre ref une fois chapitre existant).

Enfin, au bout du compte, on peut dire que ce qui compte vraiment n'est pas le choix du language de programmation, mais plutôt ce que vous voulez en faire. Car le meilleur outil est l'outil que vous métrisez. Donc si vous voulez vous lancer dans l'apprentissage de R (ou de Python, ou de C++), lancez vous à fond, apprenez les possibilités qu'offre le language, utilisez-les pour des choses qui vous tiennent à coeur, et alors vous pourrez en faire ce que vous voudrez.

2.4 Apprendre R et trouver de l'aide

Apprendre un language de programmation peut être long. La vitesse à laquelle on apprend un language par rapport à l'expérience peut se représenter grâce à une courbe d'apprentissage. Il se trouve que la courbe d'apprentissage de R n'est pas facile à déterminer, car bien souvent les personnes qui commencent ce language n'ont aucune connaissance des bases en informatique. Cependant R est un language qui dispose de nombreuses ressources d'aides sur différents formats, l'aide de R, mais aussi les livres, blogs, MOOCs, ou autres sites Q&A. Pour faciliter votre apprentissage je ne peux que vous recommander de vous forcer à apprendre un peu tous les jours.
Pour se faire, vous pouvez commencer par vous abonner à la liste de diffusion du blog R-bloggers, qui est un méta-blog regroupant plusieurs centaines de blogs en une seule et unique source. Vous pourrez trouver tout ce dont vous avez besoin, comme des tutoriels ou des news sur les sorties de packages. Ensuite, lorsque vous recherchez une information, vous pouvez aller faire un tour sur le site stackoverflow, qui est un site qui propose un système de question-réponse très au point, permettant le vote des meilleures réponses, ainsi que ses pages de documentations proposant des exemples ou des tutoriels autour du language R. Enfin, une façon de trouver plus vite des réponses à ses questions est d'utiliser le moteur de recherche DuckDuckGo que j'affectionne particulièrement car il propose en haut de page la réponse la mieux notée du premier site Q&A qu'il trouve, avec le code associé. Celà permet très souvent de ne même pas avoir besoin d'entrer sur un site pour obtenir une réponse à sa question. Testez vous-même en suivant ce lien pour un exemple sur comment faire un graphique comprennant plusieurs graphiques de séries temporelles. De plus, ce moteur de recherche est un des seuls qui ne vous piste pas. Aussi dois-je préciser qu'il ne s'agit pas ici de publicité mais seulement de préférence personnelle.

Bien sûr, la première aide à consulter avant toute est celle donnée par défaut dans chaque package. Pour y accéder sous RStudio, il vous suffit de surligner (double-cliquer) le nom de la fonction, et d'appuyer sur la touche `F1` de votre clavier. Vous pouvez aussi passer par la console de R en tappant `?nom_de_la_fonction` (ou `help(nom_de_la_fonction)`), ou encore tapper directement le nom de la fonction dans la barre de recherche de l'onglet "help" dans la fenêtre de droite.

Chapter 3

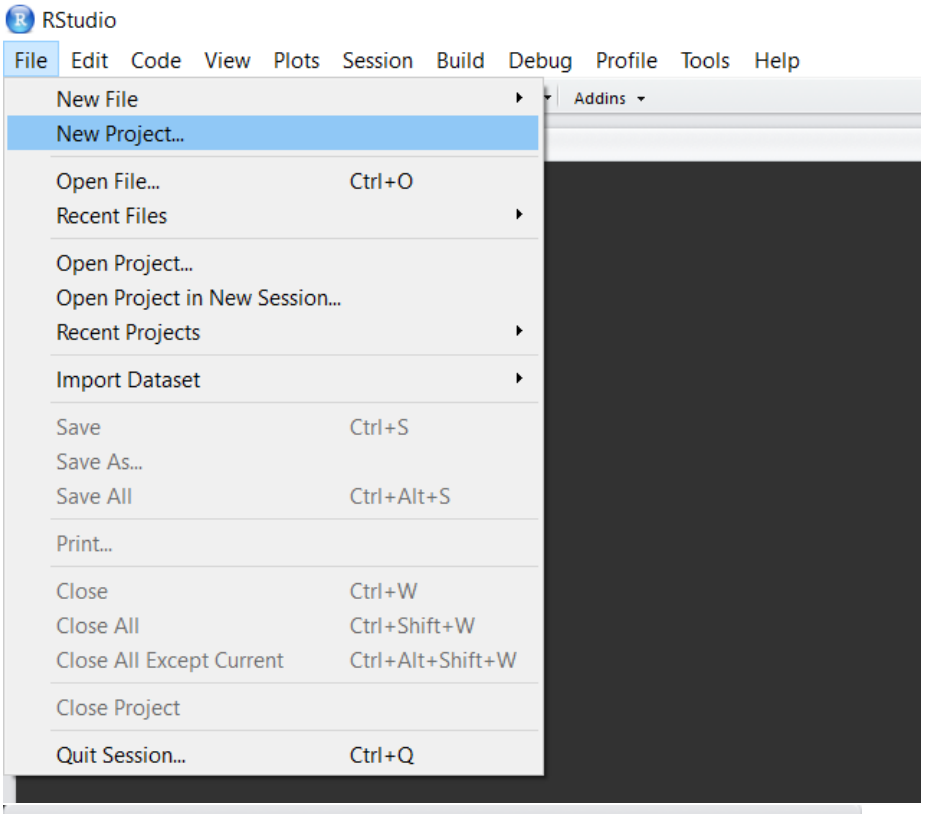
Premiers pas

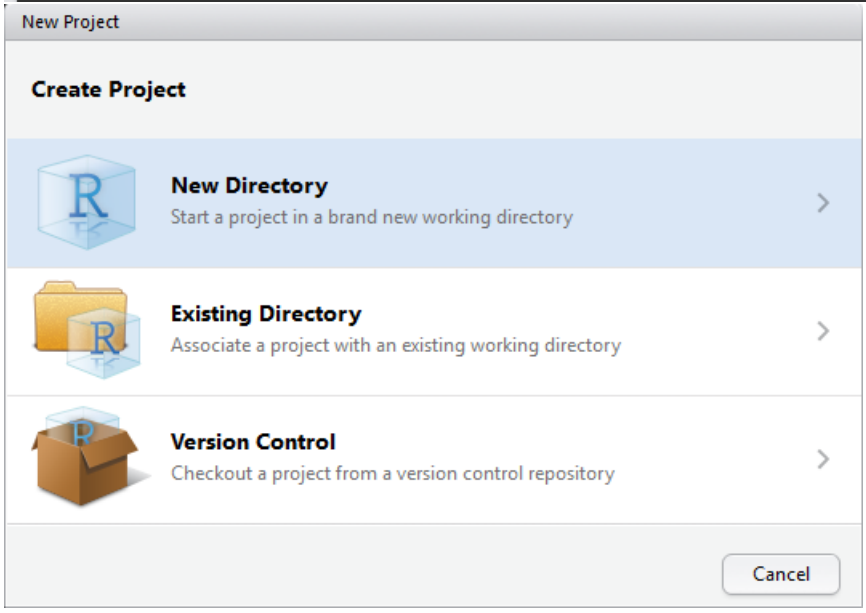
3.1 Ouvrir un projet RStudio

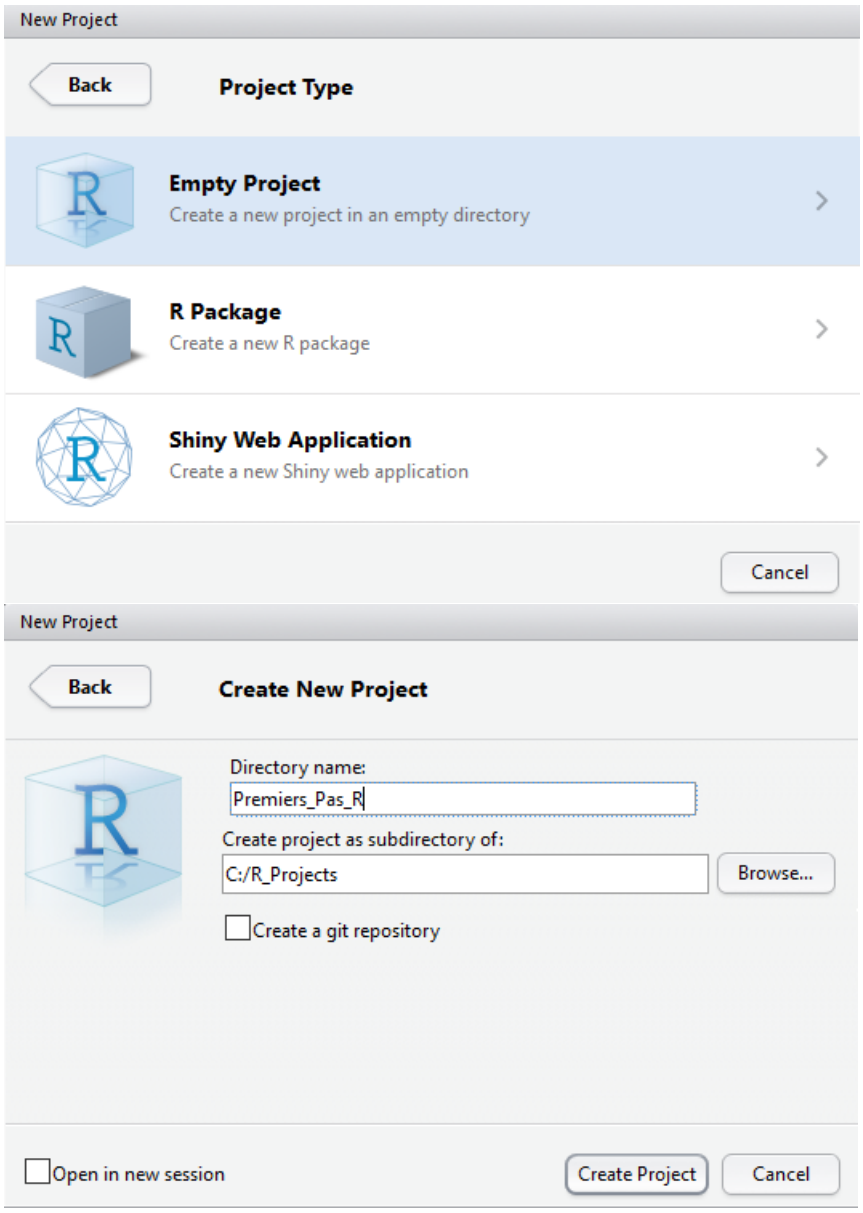
Pour commencer vous allez apprendre à créer votre premier projet RStudio. Travailler en projet est l'essence même du travail sous RStudio car cela apporte de nombreux avantages. En effet, un projet RStudio est un dossier contenant un projet quelconque, qui se doit être le plus indépendant possible. En effet, celui-ci contiendra toutes les étapes dont vous aurez besoin pour réaliser votre projet: les données, les scripts **R** ou autre, les logiciels tiers, les résultats intermédiaires et finaux, ainsi que les communications. Il est important de bien prendre connaissance de cette partie car elle conditionne de nombreuses étapes qui suivent. Croyez-moi, travailler en projet est une merveille dès qu'on a bien compris comment ça marche.

Avant tout, je vous conseille de créer un dossier général nommé "R_Projects" sur votre ordinateur dans un emplacement de votre choix (qui pourra contenir beaucoup de données).
Ensuite, pour commencer un projet, ouvrez RStudio, et cliquez sur File/New Project/New Directory/Empty project:

Ici, RStudio vous demande un nom de projet et un emplacement dans lequel ranger votre projet. Le nom de projet doit toujours être court mais le plus explicite possible, tout en évitant les caractères spéciaux (&/%-) ou les espaces (remplacez-les par "_"). Ensuite, cliquer sur browse puis choisissez l'emplacement de votre dossier R_Projects si vous l'avez déjà crée (fortement recommandé). Pour notre exemple, on nommera notre projet"Premiers_Pas_R", et on le rangera dans le dossier"C:/R_Projects" comme suit:


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Addins
New File
New Project...
Open File... Ctrl+O
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save Ctrl+S
Save As...
Save All Ctrl+Alt+S
Print...
Close Ctrl+W
Close All Ctrl+Shift+W
Close All Except Current Ctrl+Alt+Shift+W
Close Project
Quit Session... Ctrl+Q


New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel



Tada ! Vous venez de créer un projet RStudio. Maintenant il ne manque plus qu'à travailler.

3.2 Organiser son projet

Un des avantages des projets est le fait que RStudio défini automatiquement le répertoire de travail à la racine du projet. Le répertoire de travail est le chemin

du dossier dans lequel est situé votre projet. Dans mon exemple, celui-ci est "C:/R_Projects/Premiers_Pas_R". Dans **R**, on peut modifier le répertoire de travail manuellement en utilisant la commande `setwd`. Par exemple, si je ne veux travailler que dans mon dossier code, je pourrais utiliser la commande:

```
setwd("02-Code/")
```

Le répertoire de travail sert à spécifier des chemins d'accès relatifs. En effet, pour lire un fichier qui se situe dans le répertoire `01-DATA`, il existe deux façons de le faire.
Soit en spécifiant le chemin complet, depuis la racine du disque :

```
fichier= read.table("C:/R_Projects/01-DATA/fichier.txt")
```

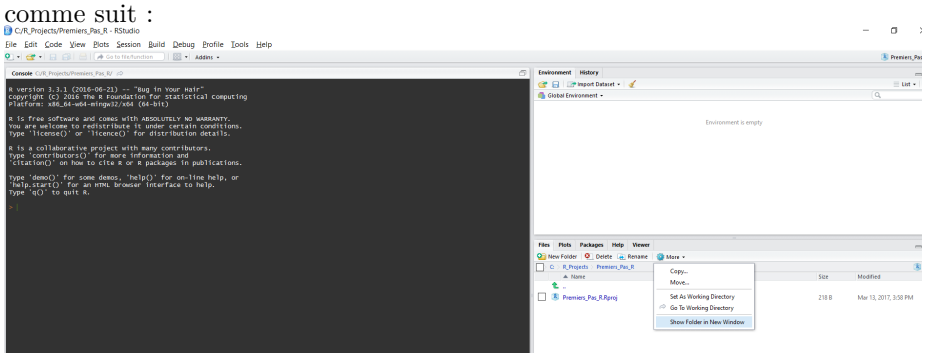
Soit en utilisant un chemin relatif par rapport au projet :

```
fichier= read.table("01-DATA/fichier.txt")
```

Cela peut paraître trivial à première vue, mais cette astuce devient très pratique lorsque les chemins deviennent longs (pensez aux chemins de la vie réelle, comme "C:/Users/username/Desktop/Docs_R/Simulations/22-01-1990/test/test2"), ou lorsque l'on partage des scripts entre collègues (pas besoin de modifier les chemins lorsqu'on passe son projet). Pour voir quel est votre répertoire de travail, vous pouvez soit le trouver écrit en haut de la console, soit simplement tapper la commande suivante:

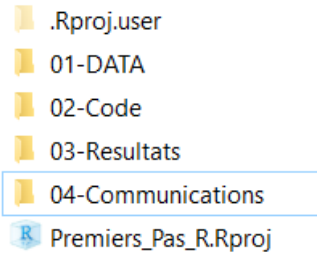
```
getwd()
```

Maintenant que le répertoire de travail n'a plus de secret pour vous, nous allons l'organiser. Pour mieux comprendre comment un projet s'organise, il faut d'abord l'ouvrir dans le gestionnaire de fichiers. Pour ouvrir la fenêtre contenant le projet, vous pouvez cliquer sur More/Show Floder in New Window comme suit :



Vous allez donc voir apparaître le dossier du projet dans une nouvelle fenêtre. Il n'y a rien de très intéressant à l'intérieur du projet pour l'instant: le fichier Premiers_Pas_R.Rproj et un dossier caché ".Rproj.user" (sa visibilité n'est absolument pas requise). Les fichiers .Rproj servent à ouvrir RStudio avec les paramètres du projet. Lorsque vous voulez ré-ouvrir votre projet après l'avoir

quitté, il vous faudra double-cliquer sur ce fichier. Maintenant nous allons commencer à pré-organiser ce dossier. Personnellement j'aime créer une suite de dossiers numérotés dans lesquels je range des fichiers triés selon l'avancée du projet comme ceci:



Le premier dossier 01-DATA contient toutes les données de départ dont à besoin le projet. Il peut s'agir d'images, de fichiers textes ou de tableurs, il seront tous présents dans ce dossier.
Le second dossier 02-Code contient tous les scripts pour **R** ou pour tout autre logiciel, numérotés eux aussi dans l'ordre d'éxécution. En général je regroupe aussi les fonctions utilisées dans les scripts dans un seul et même script **R** appelé 0-Fonctions, que j'appelle ensuite depuis les autres scripts. Si vous ne comprenez rien à la phrase précédente, ne vous en faites pas nous aborderons les fonctions très vite (référence chapitre fonctions).
Le troisième dossier 03-Resultats est le dossier qui contient tous les résultats, intermédiaires ou finaux. il peut s'agir de figures, de données, de modèles ou tout autre résultat du projet.
Enfin, le dernier dossier 04-Communications est un dossier contenant les communications sur le projet. En effet, RStudio permet l'édition de documents pdf, word et html afin de faciliter la communication. Ce livre est d'ailleurs écrit avec Rstudio. Nous parlerons des communications dans le chapitre (mettre référence chapitre communications).
Je vous incite à créer ces 4 dossiers vides à chaque fois que vous créez un nouveau projet Rstudio. Vous pouvez le faire soit manuellement, soit en utilisant la commande ci-dessous:

```
lapply(X = list("01-DATA","02-Code","03-Resultats","04-Communications"), FUN = dir.create)
```

Attention, avant d'exécuter la commande précédente veuillez vous assurer d'avoir ouvert le projet dans RStudio, et donc que votre répertoire de travail soit bien à la source du projet !

3.3 Créer un script R

R est un language intérprété, c'est à dire qu'il interprête (exécute) vos commandes en temps réel. Vous pouvez dès à présent intéragir avec **R** dans l'onglet *console* de RStudio, qui se trouve en bas à gauche. Vous pouvez par exemple

taper la commande suivante:

```
1+3
```

R vous retournera alors le résultat du calcul juste après votre commande :

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

On vient donc de voir que **R** peut être utilisé comme une calculatrice, mais celui à bien plus à offrir. En effet, vous pouvez par exemple créer des objets que **R** sauvera dans la mémoire vive[1] durant toute votre session[2].
Voici un premier exemple assez simple et concret: vous êtes partis en mission avec votre collègue qui a payé la totalité des frais. Pour le rembourser vous avez noté toutes les dépenses: 53.26€ de trajet, 108.30€ de repas, et 145.50€ pour la nuitée. Vous devez donc sommer toutes les dépenses et diviser par deux le total pour connaître votre part, comme ceci:

```
(53.26+108.30+145.50)/2
```

```
## [1] 153.53
```

Maintenant, si vous partez régulièrement ensemble, il peut être pratique de garder une trace de chaque élément, avec quelques explications pour chacune. Pour celà, il vous suffit de créer un script **R**. Un script est en fait simplement un fichier texte qui présente l'extension `.R` au lieu de `.txt`. Ce fichier sert de support pour l'écriture de commandes **R**.
Pour créer votre premier script **R**, cliquez sur fichier (file) -> nouveau fichier (New File) -> R script, ou alors en utilisant le raccourci clavier (`ctrl+shit+N`). RStudio va alors ouvrir un script vierge sans titre et non sauvegardé. La première chose à faire est d'écrire un titre. Bien sûr vous ne voulez pas que **R** exécute le titre ou les commantaires. Pour celà, vous devez lui indiquer que la ligne que vous insérez ne doit pas être interprétée en commençant la ligne par un `#`. Voici un exemple, la première ligne est un commentaire, **R** ne l'interprète pas, alors que la seconde oui:

```
# 2+3
5+6
```

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

Comme vous pouvez le voir **R** retourne 11, qui est le résultat de la deuxième ligne.
Vous pouvez donc écrire votre titre de la même façon, sans que **R** ne s'en préoccupe:

```
# Script R pour remboursement des frais:
```

[1] **R** fonctionne exclusivement avec la mémoire vive (ajouter renvoi vers chapitre optimiser **R**)
[2] voir définition wikipédia

```
# Lundi 24/04/2017: 53.26€ de trajet, 108.30€ de repas, et 145.50€ pour la nuitée
(53.26+108.30+145.50)/2 # Vous pouvez aussi taper un comentaire après une ligne de commande
```

```
## [1] 153.53
```

Pour éxécuter une ligne de commande (i.e. envoyer une ligne à **R**) depuis votre script, il vous suffit d'utiliser le raccourcis clavier `ctrl+r`, ou d'utiliser le bouton "run" dans la barre en haut de votre script.

Vous pouvez maintenant enregistrer votre script en cliquant sur Fichier (File) -> Enregistrer sous (save as), ou par le raccourci clavier `Ctrl+s`. Ensuite vous devrez nommer votre script, et je vous encourage fortement à l'enregistrer dans le dossier `02-Code` si vous avez suivi le précédent point: Organiser son projet.

3.4 Créer des objets et les manipuler

3.4.1 Objet unique

Maintenant que vous savez utiliser les calculs sous **R** et les commenter, vous allez voir comment simplifier vos scripts et vos calculs grâce à la création d'objets. Pour reprendre l'exemple précédent, nous allons créer plusieurs objet qui déterminent chacun une dépense :

```
Trajet= 53.26
repas= 108.30
hotel= 145.50
```

Cette écriture est très intuitive car elle ressemble à celle que l'on écrirait sur un bout de papier. Ce qui est intéressant avec **R**, c'est que celui-ci enregistre la valeur associée à chaque objet, ainsi, si on éxécute le nom de l'objet, **R** nous retournera sa valeur :

```
Trajet
```

```
## [1] 53.26
```

On peut maintenant utiliser des opération mathématiques directement sur nos objets, comme l'addition:

```
Trajet+repas+hotel
```

```
## [1] 307.06
```

Vous remarquerez que `Trajet` comporte une majuscule. **R** est un language sensible à la casse, c'est à dire qu'il fait la différence entre une majuscule et une minuscule, si bien que `Trajet` est différent de `trajet`. Regardez par vous même:

```
trajet= 8
Trajet
```

```
## [1] 53.26
```

```
trajet
```

```
## [1] 8
```

Ensuite, on peut assigner ce nouveau résultat à un nouvel objet, appelé pour l'occasion `Depenses_07_Avril`, qui contient les dépenses du 07 du mois d'avril:

```
Depenses_07_Avril= Trajet+repas+hotel
```

Notez bien la structure du code : l'objet `Depenses_07_Avril` est suivi d'un `=`, puis chaque nombre est additionné un à un. Ainsi, **R** fait d'abord le calcul de l'addition des objets à droite, puis assigne le résultat à l'objet nouvellement créé. On peut maintenant demander à **R** la valeur des dépenses :

```
Depenses_07_Avril
```

```
## [1] 307.06
```

3.4.2 Vecteur

La manière précédente est intéressante, mais ne permet malheureusement pas de garder une trace du calcul dans l'objet. En effet, `Depenses_07_Avril` ne contient que la somme des dépenses de la journée, mais pas leur détail. Pour se faire, nous pouvons créer un vecteur contenant chaque dépense de la journée :

```
Depenses_07_Avril= c(Trajet,repas,hotel)
Depenses_07_Avril
```

```
## [1]  53.26 108.30 145.50
```

Notez bien la structure du code ci-dessus. Chaque dépense est séparée par une virgule, et le tout est mis entre parenthèses précédées d'un `c`. Il est important de bien comprendre ce qu'il se passe ici. Pour faire court, nous avons créé un vecteur comportant trois nombres : 53.26, 108.30 et 145.50 rangés des objets appelés Trajet,repas et hotel repsectivement, puis nous les avons assignés à un nouvel objet `Depenses_07_Avril`. Maintenant, `Depenses_07_Avril` ne contient plus la somme de chaque dépense, mais chacune de leur valeur. En réalité, nous avons utilisé ici deux fonctions: la fonction d'assignation `=`, et la fonction de coercion `c`. Nous avons déjà vu la première, qui sert à assigner une ou des valeurs à un objet. La seconde quand à elle permet de concaténer plusieurs éléments en un seul objet qui sera un vecteur. Une vecteur est donc une suite de valeurs indépendantes. L'avantage d'un vecteur est de garder chaque valeur, et de pouvoir opérer des opérations mathématiques sur l'objet entier. Par exemple, si on veut obtenir la somme des dépenses de la journée, on pourra utiliser la fonction associée `sum`:

```
Depenses_07_Avril= c(Trajet,repas,hotel)
sum(Depenses_07_Avril)
```

```
## [1] 307.06
```

Cette fonction additione tous les éléments du vecteur.
Les opérations mathématiques usuelles ont toutes leurs propres fonctions:

```
sum(Depenses_07_Avril)  # somme des éléments
```

```
## [1] 307.06
```

```
mean(Depenses_07_Avril) # moyenne des éléments
```

```
## [1] 102.3533
```

```
Depenses_07_Avril*2     # multiplication de chaque élément par 2
```

```
## [1] 106.52 216.60 291.00
```

```
Depenses_07_Avril/2     # division de chaque élément par 2
```

```
## [1] 26.63 54.15 72.75
```

```
Depenses_07_Avril%%2    # modulo 2 de la divison (retourne le reste de la division par deux)
```

```
## [1] 1.26 0.30 1.50
```

D'autres objets plus complexes existent dans **R**, comme les matrices, les tableaux, ou les listes. Ils seront détaillés dans le prochain chapitre.

3.5 Introduction aux fonctions, aux packages et à l'aide

R est un language fonctionel, c'est à dire qu'il utilise un paradigme de programmation basé sur les fonctions. **R** est d'ailleurs particulièrement orienté fonctions car pratiquement tout est considéré comme tel (nous verrons cela dans le chapitre dédié aux fonctions). Le très grand intérêt de **R** est donc de disposer de nombreuses fonctions qui servent d'utilitaires. On dit qu'avec **R** la question n'est jamais "Est-il possible de . . . ?" mais toujours "Comment fait-on . . . ?". La raison est qu'il y a presque toujours une personne qui a déjà fait ce que vous voulez faire, et que la solution est très souvent disponible sous la forme d'une fonction disponible dans un package, sur un dépôt de type Gitlab/Github ou dans des commentaires sur Stackoverflow.com.
Une fonction comporte un nom, et est suivie d'arguments entre parenthèse. Nous avons déjà utilisé plusieurs fonctions précédemment: la fonction `c`, la fonction `sum` et la fonction `mean`. Pour connaître les arguments d'une fonction, il vous suffit d'exécuter son nom précédé d'un point d'intérogation, ou bien de surligner son nom et d'appuyer sur la touche `F1` de votre clavier dans RStudio. Apparaîtra alors la documentation de la fonction, qui est toujours structuré de la même manière: En haut se trouve le nom de la fonction suivi du nom de son package entre crochets (nb: base est le package intégré de base dans R).

Ensuite vient une description brève, puis son utilisation. Ici apparaitra donc les différentes façons d'utiliser la fonction, ainsi que les valeurs par défaut de ses paramètres. Vient ensuite une description de chaque arguments (synonyme de paramètres), puis de certains détails d'utilisation, des valeurs retournées par la fonction (Value), puis enfin les notes, les références, les fonctions liées (see also) et, le plus important, d'exemples. Alons voir l'aide de la fonction `sum`:

```
?sum
```

Voici le résultat anotté:

Comme on peut le voir, la fonction `sum` vient du package `base`, qui est en fait un package qui est chargé dès l'ouverture de R. De nombreuses fonctions sont disponibles dans d'autres packages. Les packages sont regroupés dans un dossier appelé "libraire". Pour connaître l'emplacement de votre librairie sur votre ordinateur, vous pouvez exécuter la fonction `.libPaths()` :

```
.libPaths()
```

```
## [1] "D:/Mes Donnees/R/win-library/3.6"
## [2] "C:/Program Files/R/R-3.6.1/library"
```

Certains packages sont téléchargés en même temps que votre installation de **R** et placés dans votre librairie. Pour utiliser les fonctions de ces packages, il faut d'abord les importer dans votre session en utilisant la fonction `library` ou la fonction `require` comme ceci :

```
library(MASS)
```

`library` va charger le package et retourner des messages d'importation, comme la version de R lors de la compilation du package, ou des avertissements. Si le package n'est pas trouvé, la fonction retourne une erreur. `require` importe aussi les packages mais retourne `TRUE` si le package est trouvé et chargé, et `FALSE` dans l'autre cas. De manière générale, `require` sera plutôt utilisé à l'intérieur d'autres fonctions, alors que `library` sera utilisé lors de programmation intéractive.

D'autres packages peuvent ne pas être présent dans votre librairie. Pour télécharger un nouveau package, on utilisera la commande `install.packages`, en spécifiant le nom du package comme charactère, entre guillemets, comme ceci:

```
install.packages("data.table")
```

Cette fonction va chercher le package sur le dépôt officiel du projet **R**, appelé CRAN, et va le télécharger, ainsi que tous les autres packages dont dépend le package concerné. Une fois téléchargé, vous pourrez charger le package comme précédemment avec la fonction `library`. Notez bien que le téléchargement n'a besoin d'être fait qu'une seule fois.

Chapter 4

Structures et types de données

Il existe plusieurs types de données dans R, et différentes façons de les manipuler en découle. Le type de structure ou d'objet utilisé dépendra principalement de la nature de l'information à y stocker et des manipulations que l'on veut effectuer dessus. Trois principaux types de données et quatre principales structures sont utilisées régulièrement dans R.

4.1 Les types de données

Les principaux types de données sont:

- les charactères `c("character","another one")`
- les doubles `c(1.5,2.248,3.01)`
- les entiers `c(1,2,3)`
- les booléens `c(TRUE,FALSE)`
  Il existe aussi d'autres types, dérivés ou non des types précédents comme les dates `as.POSIXct(c("1990-01-01","2017-06-03"))`, les complexes, ou les "raw" (séquences de bits), qui ne seront pas discutés tout de suite. Il est à noter qu'un manque de donnée se note généralement `NA` (= not applicable: non applicable) ou bien `NULL` par exemple pour un vecteur ou une liste de longueur nulle (i.e. vide). Il existe un type de `NA` pour chaque type de données : `NA_character_`, `NA_real_`, `NA_integer_` et `NA_complex_`, mais l'utilisateur n'a généralement pas en soucier, car **R** applique automatiquement le type de `NA` associé au type de donnée.

4.1.1 Les charactères (character)

Les charactères (character) peuvent prendre n'importe quelle forme (même celle d'un chiffre), et comporter n'importe quel charactère, même un espace. Pour créer un charactère, il suffit de le déclarer entre guillemets `"` (ou `'`) comme ceci: `"character"`. Ils peuvent être composés d'un seul charactère `"a"`, de plusieurs `"abc"`, ou même d'une phrase `"ceci est un seul charactère"`. Pour créer un vecteur de plusieurs charactères, il faut utiliser la fonction `c()` comme nous l'avons déjà vu: `c("a","b","c")`. Voici un exemple incluant une déclaration et un test :

```
a= "a"       # a est un charactère
a
```

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

```
is.character(a)  # on teste si a est un charactère (retourne TRUE si la condition est
```

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

4.1.2 Les doubles (double)

Les doubles sont des nombres décimaux. Ils sont le type de données que vous utiliserez probablement le plus souvent. Pour les déclarer, rien de plus simple, il suffit d'écrire votre chiffre, puis de l'utiliser comme bon vous semble. Voici un petit exemple :

```
a= 1.56       # a est un double
a*83.550      # il peut être multiplié par un autre double
```

```
## [1] 130.338
```

```
is.double(a)  # on teste si a est un double (équivalent de is.numeric())
```

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

Les doubles ont en plus d'autres valeurs possibles qui correspondent plutôt à des erreurs de calculs ou de précision:

- le `NaN`, qui signifie Not a Number (= Pas un nombre). Qui est souvent le résultat d'une opération mathématique interdite, comme un logarithme ou une racine carrée d'un nombre négatif.

- le `-Inf` et le `Inf`, qui sont le résultat d'une opération mathématique qui ne peut pas être calculée car trop grande ou trop petite. Exemple :

```
log(-1)
```

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

```
10^89999
```

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

```
-10^89999
```

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

Pour bien comprendre ce type d'erreurs, et comment ils sont utilisés par R, vous devez d'abord comprendre comment un ordinateur perçoit les doubles. Je comprends que ce type d'information puisse moins vous intéresser, mais je pense fermement que vous devez connaître ceci avant de passer à la suite. Cependant, vous pouvez sauter le paragraphe qui suit pour une première lecture, et aller directement à Les entiers (integer).

Cette partie peut vous paraître difficile et éloignée du sujet de prime abord, mais vous verrez par la suite à quel point elle peut être importante lors de certains calculs.
Si vous lisez ce livre depuis un navigateur avec une connexion internet, je vous renvoie à l'article wikipédia sur le sujet en français ou celui en anglais encore plus complet, sinon, je vais vous faire ici un petit résumé simplifié.

En informatique, un double est un type de donnée utilisé pour représenter les nombres décimaux (nombre à virgule). Comme un ordinateur utilise des suites de bits (0 ou 1) pour fonctionner, il doit représenter les nombres décimaux avec une précision donnée, c'est à dire un nombre de bits donné pour représenter un nombre. Nous les humains faisons de même. En effet, pour écrire en décimale le nombre pi, nous n'écrirons certainement pas tous les chiffres après la virgule (je vous met au défi), mais nous en feront plutôt une approximation. Parfois nous écrirons 3.14 pour aller très vite, parfois lorsque la précision s'impose nous écrirons 3.141593 (ou plus). Comme un ordinateur ne peut pas savoir à l'avance avec quelle précision nous voulons travailler, nous lui imposons une précision fixe, qui aujourd'hui est le format double précision (d'où le nom de double) 64 bits de la norme (IEEE 754). Ces nombres sont représentés grâce à trois informations: le signe du nombre `s` (- ou +), la mantisse `m` (~les chiffres du nombre) et un exposant `e` (entier relatif qui donne la position de la virgule dans la mantisse, ce qui donne le nom de virgule flottante). Il s'agit un peu d'un équivalent de la notation scientifique. Un nombre peut donc être représenté comme ceci[1] :

[1] Source de l'image

En double précision, ces trois informations ensemble ne doivent pas dépasser 64 bits. L'encodage est donc divisé comme suit: 1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse, ce qui donne 53 bits de précision, donc environ 16 chiffres significatifs. Si vous avez bien compris comment les doubles sont encodés, alors vous aurez compris que R peut donc représenter des nombres avec beaucoup de précision, mais pas avec une précision infinie.
Que vous ayez bien compris ou non, retenez ces deux conclusion:

1- La précision n'est pas infinie:
Il est important de toujours garder en tête que la précision des calculs dans R n'est pas infinie, et donc que sous certaines conditions R peut présenter un comportement non valide mathématiquement, comme par exemple lors d'une soustraction de nombres extrêmement proches de zéro comme suit :

```
10^-325-10^-324
```

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

Comme vous pouvez le voir R retourne 0, ce qui est faux. Un calcul de cette précision peut paraître superflu mais ce genre d'erreur peut s'accumuler de calculs en calculs et devenir bien plus grande par la suite.
Bien sûr, il est très rare d'utiliser de telles précisions, et les erreurs engendrées n'auront aucun impact réel sur vos calculs. Pour l'exemple, voici un autre calcul très précis, que R peut gérer (si votre ordinateur le permet) :

```
10^-324-10^-323
```

```
## [1] -9.881313e-324
```

NB: pour voir quel degré de précision votre ordinateur (et R) peut gérer, vous pouvez regarder le contenu de l'objet `.Machine` dans la console de R.

2- La précision en informatique = chiffres significatifs
Il faut retenir que **les grands nombres ont moins de chiffres après la virgule**. En effet, la précision en informatique se réfère aux chiffres significatifs, c'est à dire au nombre de chiffres que comporte un nombre au **total** (avant ou après la virgule). Ceci est différent de la "précision décimale", qui se réfère au

nombre de chiffres après la virgule.
Comme la précision ne change pas avec le nombre considéré, plus un nombre est grand (positif ou négatif), moins on peut représenter de chiffres après sa virgule. En effet, 0.003 comporte 4 chiffres, dont 3 après la virgule. 3000 comporte 4 chiffres aussi, mais aucun après la virgule. Ils ont la même précision arithmétique au sens de chiffres significatifs (4), mais pas au sens de "précision décimale" qui donnerait 3000.0000 pour une précision de 4.
Regardez par vous même, on voit que R est capable de calculer ceci :

```
a= 10^-324-10^-323
# On teste si "a" est différent de 0:
a!=0
```

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

(a est bien différent de 0)

Mais il ne pourra pas calculer ceci :

```
a= 1-10^-323
# On teste si "a" est différent de 1:
a!=1
```

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

(a est considéré égal à 1, donc la soustraction `1-10^-323` n'est même pas prise en compte)

On peu aussi noter que même si R arrive à utiliser des nombres avec de grandes précisions, celui-ci arrondi souvent le résultat lorsqu'il l'affiche dans la console. Pour voir un nombre complet, on peut utliser la fonction `print()` avec son argument `digits`, qui donne le nombre de décimale voulues, comme ceci:

```
print(1+ 2.2e-14,digits = 22)
```

```
## [1] 1.000000000000022
```

Enfin, si la précision vous importe vraiment et que vous devez utiliser des chiffres plus précis que la double précision, vous pouvez jeter un oeil au package `Rmpfr` et le package `Brobdingnag`.

4.1.3 Les entiers (integer)

En informatique, les entiers sont des entiers relatifs (nombre avec signe). Pour être différenciés des doubles, les entiers (integer) sont déclarés suivis de la lettre `L` comme suit :

```
a= 1L
b= 24L
c= -500L
paste(a,b,c)
```

```
## [1] "1 24 -500"
```

```
is.integer(a) # teste si a est un entier
```

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

4.1.4 Les booléens (boolean or logical)

Les booléens sont des nombres logiques, aussi appelés variable d'état. Les booléens ont deux valeurs possibles uniquement, `TRUE` (vrai) et `FALSE` (faux). Dans R `T` est un raccourci pour `TRUE`, et `F` un raccourci pour `FALSE`. Voici comment déclarer et tester un booléen :

```
a= TRUE
b= T          # équivalent de a
c= FALSE
d= F          # équivalent de c
is.logical(a) # teste si a est un booléen
```

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

4.1.5 En bonus, les facteurs (factors)

Les facteurs sont un type de données un peu spéciaux. Ils permettent d'utiliser des catégories de données dans une même variable. On peut prendre l'exemple d'une personne qui regarde de quelle couleur sont les voitures qui passent devant son entrée, s'il relève 3 voitures rouges, deux bleues et quatre blanches, alors on peut dire que nous avons trois catégories de couleurs de voitures: rouge, bleue, blanche. Pour déclarer ceci dans R, nous ferions comme suit :

```
couleurs= factor(c("rouge","rouge","rouge","bleue","bleue","blanche",
                   "blanche","blanche","blanche"))
couleurs          # nous avons un vecteur de charactères, et R reconnait automatiqueme
```

```
## [1] rouge   rouge   rouge   bleue   bleue   blanche blanche blanche blanche
## Levels: blanche bleue rouge
```

On voit que `couleurs` se présente comme un vecteur, mais contient un méta-donnée de plus, les `levels`, qui résument les valeurs uniques du vecteur, et donc qui décrivent les différentes valeurs possible du vecteur.

La fonction level permet de retrouver les catégories :

```
levels(couleurs)
```

```
## [1] "blanche" "bleue"   "rouge"
```

Les facteurs sont souvent utilisés dans d'autres fonctions comme pour les graphiques ou les modèles. En général, il est préférable d'éviter leur utilisation et de les remplacer par un autre type de donnée, à moins d'en avoir un réel besoin.

4.2 Les transformations

R est un language extrèmement maléable, si bien qu'il existe des "ponts" entre les types de données. On pourra donc très facilement transformer un type de données en un autre dans certains cas.

1. integer vers double :
   il s'agit de la transformation la plus facile. R ne requiert aucun préalable à cette transformation, si bien que celle-ci se fait sans même y réfléchir:

```
a= 1L           # a est déclaré comme un entier égal à 1
is.integer(a)   # a est bien considéré comme un entier
```

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

```
b= a*2.5        # on multiplie a par un double et on le stocke dans b
is(b)           # b est lui-même un double
```

```
## [1] "numeric" "vector"
```

2. double to integer :
   L'inverse n'est pas aussi simple et doit se faire en utilisant une fonction de `R` semblable à la fonction de test (`is.integer`) que nous avons déjà vu : `as.integer`. En voici un exemple :

```
a= c(2,2.5,3.8)   # a est déclaré comme un double
is.double(a)      # a est bien considéré comme un double
```

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

```
b= as.integer(a)  # on force a à devenir un entier
b                 # b est maintenant un integer (R à tronqué le chiffre)
```

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

3. double (ou integer) to character :
   Cette transformation est aussi assez simple et peut se faire en utilisant la fonction `as.character` :

```
a= c(2,2.5,3.8)     # a est déclaré comme un double
is.double(a)        # a est bien considéré comme un double
```

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

```
b= as.character(a)  # on force a à devenir un character
b
```

```
## [1] "2"   "2.5" "3.8"
```

`b` est maintenant un charactère (notez les guillemets qui entourent chaque chiffre)

4. character to double (ou integer) :

Comme vous l'avez deviné, cette transformation peut se faire grâce à la fonction `as.double`. Cependant, il faut être très prudent lorsque l'on fait une transformation dans ce sens, car R peut parfois présenter des comportements inattendus.

```
a= c("1","2","3")    # a est déclaré comme un vecteur de 3 charactères
is.character(a)      # a est bien considéré comme un charactère
```

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

```
b= as.double(a)      # on force a à devenir un double
b                    # b est une version de a en double
```

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

Maintenant un exemple avec des facteurs, qui peuvent donner des résultats inattendus lorsqu'ils sont composés de charactères comportant des chiffres et des lettres :

```
a= factor(c("1","a","3"))    # a est déclaré comme un facteur de logueur 3
b= as.numeric(a)             # on force a à devenir un double
b                            # b est bien un double, mais le résultat est quelque peu d
```

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

```
is(b)
```

```
## [1] "numeric" "vector"
```

Comme on peut le voir, la transformation est déroutante : le "3" du facteur est devenu un 2, et le "a" un 3. Pour comprendre ce qu'il s'est passé, il faut savoir que pour transformer un facteur en numérique, R prends les catégories comme des charactères, puis les trie alphanumériquement (i.e. de 0 à 9 puis de A à Z), puis alloue l'index de chaque catégorie comme nouvelle valeur. Donc ici R à décidé de l'ordre suivant dans les catégories : "1"<"3"<"a", puis leur à alloué un index : "1"=1, "3"=2, "a"=3, et a remplacé les valeurs de chaque catégorie par son index pour que "1","a","3" devienne 1,3,2. Ce type d'erreur peut être évité en utilisant la combinaison de fonctions suivantes :

```
a= factor(c("1","a","3"))    # a est déclaré comme un facteur de longueur 3
b= as.numeric(levels(a))[a] # on force a à devenir un double
```

```
## Warning: NAs introduits lors de la conversion automatique
```

```
b
```

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

`b` est bien un double, le résultat est bon, et le charactère "a" est transformé en `NA`.

Maintenant que vous connaissez mieux les charactères, les doubles, les entiers, et les booléens, vous allez voir comment les organiser dans des structures de

données qui vous permettront de les stocker dans des objets plus ou moins complexes.

4.3 Les Structures de données

R met à disposition de ses utilisateurs quatre grands types de structures de données plus ou moins liées: les vecteurs, les matrices, les tableaux et les listes.

4.3.1 Vecteurs

Nous avons déjà vu un premier exemple de vecteurs dans le chapitre précédent (Premiers pas), qui introduisait la fabrication d'un vecteur contenant des chiffres. Les vecteurs sont une structure de données à une dimension, c'est à dire qu'il s'agit d'une suite de données rangées en "ligne". Un vecteur ne peut contenir qu'un seul type de données à la fois (charactère, booléen...). Si l'on construit un vecteur avec plusieurs types de données consécutifs, ils sont alors transformés de la même façon que nous avons vu les tranformations précédement dans Les transformations. Il est à retenir que **R** essaye toujours de prendre le types de donnée qui gardera le plus d'information. Par exemple un mélange de charactères et de doubles sera transformé en charactère pour ne pas perdre l'information des charactères (qui seraient transformés en NA autrement).
Il existe plusieurs moyens de créer un vecteur. Tous ont un avantage associé, et seront utilisés en fonction du contexte:

- En utilisant as.vector pour créer un vecteur de la taille désirée, que l'on remplira par la suite. Comme les vecteurs ne peuvent contenir qu'un seul type de données, il faut déclarer le type dès la création grâce à l'argument "mode". Les différents modes sont `logical` (= booléen), `integer` (= entier), `numeric` (= double), `complex` (= complexe), `character` (= charactère) and `raw` (= bits). Voici un exemple de vecteur numérique:

```
a= vector(mode = "numeric", length = 10)   # a est un vecteur de longueur 10
a
```

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

- En utilisant la fonction `c()`, qui concatène ses arguments :

```
a= c(1,10,8)
a
```

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

- En utilisant d'autres fonction comme `seq()`, `rep()`, `:` etc... :

```
a= seq(from = 1, to = 10, by = 1)
a= rep(x = 0, times= 10)
a= rnorm(n = 10, mean = 1, sd = 2)
```

4.3.2 Matrices

Une matrice est très semblable à un vecteur, sauf qu'elle comporte deux dimensions, il s'agit là d'une matrice dans son sens mathématique. Comme les vecteurs, les matrices ne peuvent contenir qu'un seul type de données. Voici un exemple de création de matrice:

```
matrix(data = 1:9, nrow = 3)   # une matrice
```

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

La fonction `matrix` est très utile car on peut choisir le nombre de lignes et/ou de colonnes, ainsi que le sens de remplissage des données avec l'argument `by.row`. La fonction `matrix` est un très bon utilitaire pour construire des structures 2D vides notemment, comme suit:

```
matrix(data = NA, nrow = 8, ncol=2)   # une matrice 8*2 comportant la valeur NA
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
## [3,]   NA   NA
## [4,]   NA   NA
## [5,]   NA   NA
## [6,]   NA   NA
## [7,]   NA   NA
## [8,]   NA   NA
```

Une matrice peut aussi comporter une seule colonne, ou une seule ligne.

4.3.3 Tableaux

Il existe deux types de tableaux dans R: `array` et `data.frame`.
Le premier est rarement utilisé. Il s'agit ni plus ni moins d'un vecteur à n dimensions (englobe donc vecteur et matrices).

```
array(data = 1:9, dim = 9)   # un array tel un vecteur
```

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

```
array(data = 1:9, dim = c(3,3))   # un array tel une matrice 3*3
```

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

```
array(data = 1:9, dim = c(3,3,2))   # un array tel une liste de 2 matrices de 3*3 (dimension 3)
```

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

Le second, le `data.frame`, est peut-être la structure la plus utilisée dans R. Cette structure représente le sens commun que l'on se fait d'un tableau. Structurellement, le `data.frame` se présente donc comme un tableau comportant un nombre défini de colonnes et de lignes. Chaque colonne comporte le même nombre de lignes. Dans **R**, un `data.frame` est une liste de vecteurs de même longueur. Le plus grand avantage du `data.frame` est que chaque colonne peut contenir un type de donnée différent.

```
data.frame(a= c("Jour 1", "Jour 2", "Jour 3"), b=c(40, 50, 45))   # un data.frame
```

```
##        a  b
## 1 Jour 1 40
## 2 Jour 2 50
## 3 Jour 3 45
```

Les `data.frame` sont utilisés par la grande majorité des fonctions dans R.

4.3.4 Listes

Les listes (`list`) sont aussi un type de structure très important dans R. Il faut penser la liste comme étant au `data.frame` ce qu'est l'`array` à la matrice. Il s'agit d'un `data.frame` de dimension n. On peut mettre n'importe quel type de données de n'importe quelle taille dans une liste, même une autre liste (= nested list).

```
list(a= c("Jour 1", "Jour 2", "Jour 3"), b=c(40, 50, 45), c= 1, d= list(1:3, 8:4, "a"))
```

```
## $a
## [1] "Jour 1" "Jour 2" "Jour 3"
##
## $b
## [1] 40 50 45
```

```
##
## $c
## [1] 1
##
## $d
## $d[[1]]
## [1] 1 2 3
##
## $d[[2]]
## [1] 8 7 6 5 4
##
## $d[[3]]
## [1] "a"
```

Les listes sont un peu complexe à appréhender mais sont certainement le type de structure le plus maléable, et finalement le plus simple à travailler. Il est à noter que le `data.frame` est en fait une liste de vecteurs. On peut donc remplir l'une de ses colonnes par une liste, mais ce type d'utilisation est plutôt déconseillé dans un premier temps car il limite l'utilisation de fonctions spéciales adaptées aux `data.frame`, mais aussi à celles adaptées aux listes.

4.3.5 Attributs

Il est possible d'associer n'importe quelle métadonnée à un objet dans R. Ces métadonnées sont appelés attributs. Ces attributs n'ont aucune influence sur la valeur de la donnée elle-même, mais elle peut apporter des informations complémentaires à l'utilisateur ou à une fonction.

Voici un exemple d'attribut qui permet à l'utilisateur de se rappeler l'unité d'un objet:

```
a= 1   # a vaut 1 cm
attr(a, "unit")= "cm"
```

Certains attributs sont utilisés par des fonctions. C'est le cas du nom et de la dimension d'un objet.

Voici un exemple de la hauteur de 10 arbres dans une forêt:

```
hauteur= rnorm(n = 10, mean = 25)
```

Cette hauteur n'a pas de nom (`names()` retourne `NULL`):

```
names(hauteur)
```

```
## NULL
```

On peut assigner un nom comme suit:

```
names(hauteur)= paste0("arbre_",1:10)
names(hauteur)
```

```
##  [1] "arbre_1"  "arbre_2"  "arbre_3"  "arbre_4"  "arbre_5"  "arbre_6"
##  [7] "arbre_7"  "arbre_8"  "arbre_9"  "arbre_10"
```

On peut aussi retrouver le nom de chaque valeur grâce à la fonction `attributes()`:

```
attributes(hauteur)
```

```
## $names
##  [1] "arbre_1"  "arbre_2"  "arbre_3"  "arbre_4"  "arbre_5"  "arbre_6"
##  [7] "arbre_7"  "arbre_8"  "arbre_9"  "arbre_10"
```

Idem pour la dimension d'une matrice avec `dim()`. On crée une matrice de dimension 2,3:

```
matrice= matrix(1:8, 2,3)
```

```
## Warning in matrix(1:8, 2, 3): la longueur des données [8] n'est pas un
## diviseur ni un multiple du nombre de colonnes [3]
```

```
matrice
```

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

Sa dimension est 2,3:

```
dim(matrice)
```

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

On peut la modifier comme ceci:

```
dim(matrice)= c(3,2)
matrice
```

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

Ou en utilisant la fonction `attr()`:

```
attr(matrice,"dim")= c(2,3)
matrice
```

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

Les fonctions `names()` et `dim()` sont en fait des raccourcis pour `attr(matrice,"names")` et `attr(matrice,"dim")` respectivement.

4.4 Manipuler les structures

La manipulation des structures se fait en accord avec les spécificités de chacunes.

4.4.1 Index et Test

La manipulation la plus courante est l'indexage. En d'autres termes, il s'agit là de récupérer les valeurs d'une structure en fonction de sa position, de son nom ou d'une condition. L'index se fait le plus souvent en utilisant la fonction `[`, appelée fonction `extract`. Par exemple, pour récupérer la troisième valeur d'un vecteur, on utilisera la fonction comme suit: `a[3]`. Il est à noter que l'indexage se fait à partir de 1, car la première valeur du vecteur est en position 1. Cela peut porter à confusion pour les utilisateurs venant d'autres languages de programmation tels que Python, qui indexe à partir de 0. Une autre particularité de R est l'utilisation du `-` comme un **sauf**. Donc `a[-3]` récupèrera le vecteur `a` sauf sa troisième valeur. Lorsque l'on a plus d'une dimension (e.g. matrice ou data.frame), on sépare l'index de chaque dimension par une virgule (e.g. [ligne,colonne]). Pour récupérer les objets d'une liste, on utilisera la fonction `[[`. On peut aussi indexer un vecteur par le nom de son objet comme suit `a["nom de l'objet"]` mais aussi en testant une condition comme ceci `a[c(T,T,F)]`. Une deuxième fonction que l'on peut utiliser pour le `data.frame` et la `list` est la fonction `$`. Celle-ci indexe les colonnes du premier et les objets du second par leur nom.

- Vecteur

```
a= c(10:15) # a est un vecteur numérique de 10 à 20.
a[3]        # on veut récupérer la troisième valeur du vecteur a
```

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

```
a[c(1,5,3)] # on veut récupérer la 1e, la 5e et la 3e valeur de a
```

```
## [1] 10 14 12
```

```
a[-1]       # on veut récupérer toutes les valeurs de a sauf la première.
```

```
## [1] 11 12 13 14 15
```

```
a[c(T,F,F,T,T)] # on veut récupérer toutes les valeurs de a sauf la 2e et 3e valeur qu
```

```
## [1] 10 13 14 15
```

- matrice:

```
a= matrix(10:21, nrow= 3) # a est une matrice
a[3,]                     # on veut récupérer la troisième ligne entière
```

```
## [1] 12 15 18 21
```

```
a[c(1,3), 1]              # on veut récupérer la 1e et la 3e ligne, mais seulement la 1e colonne
```

```
## [1] 10 12
```

```
a[-1,-2]                  # on veut récupérer toutes les lignes sauf la 1e et toutes les colonnes
```

```
##      [,1] [,2] [,3]
## [1,]   11   17   20
## [2,]   12   18   21
```

- data.frame

```
a= data.frame(Num= c(1:10), Prix= c(11:20)) # a est un data.frame
a[3,]          # comme sur les matrices, on veut récupérer la troisième ligne entière
```

```
##   Num Prix
## 3   3   13
```

```
a[c(1,3), 1]  # on veut récupérer la 1e et la 3e ligne, mais seulement la 1e colonne
```

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

```
a[-1,-2]      # on veut récupérer toutes les lignes sauf la 1e et toutes les colonnes sauf la 2e
```

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

```
a$Prix        # on veut récupérer la deuxième colonne par son nom avec la fonction `$`.
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
a["Num"]      # on veut récupérer la 1e colonne par son nom avec la fonction `[`
```

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

- liste

```
a= list(Num= c(1:3), Prix= c(11:13), Id_et_serie= list(Id= c(1,1,1), serie= c(2,2,2)), Classe= da
a[1:2]        # on veut récupérer une liste des 1er et 2nd objets de la liste
```

```
## $Num
## [1] 1 2 3
```

```
##
## $Prix
## [1] 11 12 13
```

```
a[[1]]        # on veut récupérer les valeurs du 1er objet de la liste
```

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

```
a[-1]         # on veut récupérer une liste de tous les objets de la liste sauf le 1er
```

```
## $Prix
## [1] 11 12 13
##
## $Id_et_serie
## $Id_et_serie$Id
## [1] 1 1 1
##
## $Id_et_serie$serie
## [1] 2 2 2
##
##
## $Classe
##   a b
## 1 1 1
## 2 2 2
## 3 3 3
```

```
a[[3]][1]     # on veut récupérer les valeurs du premier objet de la liste en 3e posit
```

```
## $Id
## [1] 1 1 1
```

```
a[[4]][,1]    # on veut récupérer les valeurs de la 1e colonne du data.frame en 4e pos
```

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

```
a$Classe[,1]  # on mélange l'utilisation des deux fonctions `$` et `[` pour faire de m
```

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

```
a$Prix        # on veut récupérer les valeurs de l'objet nommé Prix
```

```
## [1] 11 12 13
```

```
a$Id_et_serie$serie # on veut récupérer les valeurs de l'objet nommé serie dans l'obje
```

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

```
a[c("Num","Classe")]# on veut récupérer le 1er et 4e objets de la liste a
```

```
## $Num
```

```
## [1] 1 2 3
##
## $Classe
##   a b
## 1 1 1
## 2 2 2
## 3 3 3
```

Donc pour résumer, l'indexage commence à 1, et l'index se fait comme suit: vecteurs [i], matrices/tableaux [i,j] et listes [[i,j]] ou [1] ou [[1]] Pour indexer plusieurs éléments:

- selon numéro de l'index lignes/colonnes [1:10], [,1:10]
- selon condition test [c(T,T,F,T)]

4.4.2 Remplacer, assigner

Il est possible de remplacer des valeurs dans une structure en spécifiant l'index de la valeur à remplacer et lui ré-assignant une nouvelle valeur :

```
a= c(10:15) # a est un vecteur numérique de 10 à 20.
a[3]        # la troisième valeur de a est de 12
```

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

On remplace sa valeur par 5:

```
a[3]= 5
a           # La 3e valeur est modifiée à 5
```

```
## [1] 10 11  5 13 14 15
```

Attention, si on remplace une valeur par un autre type de données, cela peut changer le type de tout le vecteur:

```
a[1]= "a"
a           # a a été modifié en charactere
```

```
## [1] "a"  "11" "5"  "13" "14" "15"
```

On peut aussi modifier plusieurs valeurs à la fois:

```
a[1:3]= 5
a
```

```
## [1] "5"  "5"  "5"  "13" "14" "15"
```

4.4.3 Concaténation

On peut aussi concaténer les différentes structures de données:

- vecteur, en utilisant la fonction `c()` :

```
a= 1:10
b= c(2,8)
vec= c(a,b)
vec
```

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

- matrice/data.frame en utilisant `rbind` pour concatener par lignes, `cbind` pour concatener par colonnes :

```
a= matrix(data= 1:9, nrow = 3)
b= matrix(data= 10:18, nrow = 3)
mat_c= cbind(a,b)
mat_c
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    4    7   10   13   16
## [2,]    2    5    8   11   14   17
## [3,]    3    6    9   12   15   18
```

```
a= matrix(data= 1:9, nrow = 3)
b= matrix(data= 10:18, nrow = 3)
mat_r= rbind(a,b)
mat_r
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
## [4,]   10   13   16
## [5,]   11   14   17
## [6,]   12   15   18
```

4.4.4 Transformation

On peut transformer des structures simples très facilement dan R, comme passer d'un charactère à un numérique par exemple (voir [transform_datatype]), mais aussi pour des structures plus complexes, comme les matrices, les tableaux et les listes. Les structures ont très souvent une fonction associée pour passer de l'une vers l'autre.

Par exemple pour transformer quelque chose en matrice, on utilisera `as.matrix`, `as.data.frame` pour les tableaux, `as.vector` pour les vecteurs, ou encore `as.list` pour les listes. Bien sûr l'utilisateur doit s'efforcer de présenter les données de façons structurées pour que ces fonctions puissent effectuer la transformation.

Une liste pourra par exemple être tranformée en `data.frame` seulement si son contenu est de même longueur:

```
liste= list(a= 1:2, b= 3:4)
as.data.frame(liste)
```

```
##   a b
## 1 1 3
## 2 2 4
```

Ici le `data.frame` est bien crée car `a` et `b` dans `liste` ont la même longueur. Par contre si `b` était plus long, la fonction `as.data.frame` ne sait pas a priori ce qu'elle doit faire, donc elle retourne une erreur.

4.4.5 Transposition

Une matrice peut être transposée grâce à la fonction `t()` comme suit:

```
matrice= matrix(1:9, nrow = 3)
t(matrice)
```

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

On peut aussi transposer un `data.frame` avec `t()`, mais celui-ci est alors transformé en matrice:

```
df= as.data.frame(matrice)
t(df)
```

```
##    [,1] [,2] [,3]
## V1    1    2    3
## V2    4    5    6
## V3    7    8    9
```

La transformation en matrice peut être un problème si l'on utilise des colonnes de types différent, car la fonction va uniformiser le type des données:

```
df= data.frame(a= 1:2, b= c("1","2"))
t(df)
```

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

Ici toutes les données sont transformées en charactère.

4.4.6 Opérations

Il est très simple d'appliquer des opérations mathématiques sur des structures dans R car ces fonctions sont vectorisées, c'est à dire que les fonctions `+`, `-`, `*`, `/`,... peuvent s'appliquer à plusieurs valeurs d'un seul coup. Par exemple on peut multiplier un vecteur par un autre comme ceci:

```
a= c(1:5)
b= c(6:10)
a * b
```

```
## [1]  6 14 24 36 50
```

On peut aussi multiplier des colonnes de `data.frame`:

```
df= data.frame(a= c(1:5), b= c(6:10))
df$a * df$b
```

```
## [1]  6 14 24 36 50
```

Pour plus de détails, voir le chapitre "[datascience1]".

Chapter 5

Structure de contrôle

Comme tout language de programmation, R dispose de strcutures de contrôle du flux. Il s'agit principalement des boucles `for` et `while`, et des conditions de type `if` ou `switch`.

5.1 Logical operators

Les opérateurs logiques permettent de tester des conditions. Il en existe de nombreux, et les principaux sont:

- `&` et `&&`: `condition1 & condition2` teste si les deux conditions sont vraies. `&` teste les deux conditions à chaque fois, alors que `&&` teste d'abord la première, puis la seconde si la première est **vraie**. Cela permet de mettre des conditions qui pourraient ne pas avoir à être testées si la première condition est déjà fausse, et de gagner du temps. Exemple:

```
a= TRUE ; b= FALSE
a&b
```

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

```
b&&(b= 2)
```

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

Ici seulement la partie de gauche es executée (b), donc b=2 n'est pas exécutée, b est toujours égal à FALSE. Dans la deuxième expression (`b&&(b= 2)`), b vaut `FALSE` donc `b=2` n'est pas exécuté, donc b reste égal à FALSE:

```
b
```

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

- `|` et `||`: `condition1 | condition2` teste si l'une ou l'autre des deux conditions est vraie. `|` teste les deux conditions à chaque fois, alors que `||` teste d'abord la première, puis la seconde si la première est **fausse**. Exemple:

```
a= TRUE ; b= FALSE
a|b
```

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

```
a||(b= 2)
```

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

Dans la deuxième expression (`a||(b= 2)`), a vaut `TRUE` donc `b=2` n'est pas exécuté, donc b reste égal à FALSE:

```
b
```

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

- `==`, `<`, `<=`, `>`, `>=`: tests d'égalité (noteza bien le double `=`), d'infériorité ou de supériorité inclusive ou exclusive. Exemple:

```
a= c(1,2)
a==1
```

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

```
a<=2
```

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

- `%in%` et `match()`: teste si une valeur est présente dans un vecteur. Exemple:

```
a= c(1,2)
a%in%1
```

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

- `all()`: teste si toutes les valeurs d'un vecteur sont vraies. Exemple:

```
a= c(1,1)
any(a==1)
```

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

- `any()`: teste si au moins une valeur d'un vecteur est vraie. Exemple:

```
a= c(1,2)
any(a==1)
```

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

- `!`: teste si une valeur est fausse (sert aussi à inverser un test). Exemple:

```
!FALSE
```

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

ou encore:

```
a= c(1,2)
a!=1
```

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

5.2 Boucles

5.2.1 Boucle `for`

Une boucle `for` permet d'exécuter une partie de code de façon répétée une nombre de fois donné. Elle présente une structure particulière comme suit:

```
for(variable in vector){
  ...
}
```

On note d'abord le code `for(variable in vector)`. Le `for` sert à déclarer le début de la boucle, et `(variable in vector)` permet de donner le nombre de fois que la boucle sera répétée, ainsi que la valeur de `variable` à chaque pas de temps. Le `...` représente la séquence de code à exécuter.

Voici un exemple pour mieux comprendre:

```
for(i in 1:3){
  print(i)
}
```

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

On peut lire cette boucle `for` comme suit: pour tous les i dans la séquence de 1 à 3 (`for(i in 1:3)`), écrit i (`print(i)`) . On peut donc logiquement voir que la fonction a successivement écrit 1, puis 2 puis 3. On peut aussi faire des calculs simples:

```
a= 0
for(i in 1:3){
  a= a + i
}
```

Ici, on crée un objet `a` avec pour valeur 0, puis on lui ajoute cumulativement 1 puis 2 puis 3. Cela revient à faire `a= 1 + 2 + 3`.

Les boucles `for` sont très utilisées en informatique car elles permettent de répéter des séquences de codes en ne les écrivant qu'une seule fois. Cependant il est préférable d'utiliser d'autres techniques lorsqu'on le peut dans R car celles-ci sont très lentes. On peut par exemple utiliser la famille des `apply`, ou la vecorisation.

5.2.2 Boucle while

Une boucle `while` exécute une portion de code tant qu'une condition reste vraie. En voici la construction:

```
while(condition){
  ...
}
```

Tant que `condition==TRUE`, alors `while` va itérer l'exécution de `...`. Voici un exemple qui écrit incrémente et écrit la valeur de `a` à chaque itération jusqu'à ce que `a` vaille 3.

```
a= 0
while(a<3){
  a= a+1
  print(a)
}
```

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

Notez bien que la condition est exécutée au début et non à la fin de la portion de code. D'où la conditon `a<3` car la boucle ira jusqu'à `a==3`, puis s'arrêtera.

5.3 Conditions

Il est possible de n'exécuter une séquence de code que lorsqu'une condition est vraie (ou fausse). Pour se faire il existe plusieurs fonctions dans R.

La plus basique est le `if`. Il se construit comme suit:

```
if(condition){
  ...
}
```

`...` représente la séquence de code à exécuter. Cette séquence n'est exécutée que si `condition` est vraie. On peut aussi exécuter une autre séquence en ajoutant un `else` :

```
if(condition){
  # 1e séquence de code
```

```
}else{
  # 2e séquence de code
}
```

Ici, la 1e séquence de code s'exécute si `condition` est vraie, et la 2e s'exécute si elle est fausse.

On peu corser la chose en utilisant des `if else` comme suit:

```
if(condition1){
  # 1e séquence de code
}else if(condition2){
  # 2e séquence de code
}else{
  # 3e séquence de code
}
```

Dans ce cas on teste d'abord si `condition1` est vrai. Si oui, on exécute la 1e séquence de code, sinon on teste `condition2`. Si elle est vraie on exécute la 2e séquence de code, et si elle est fausse, on exécute la 3e séquence de code.

Voici un exemple plus concret:

```
Nom= "Louis"

if(Nom == "Jacques"){
  print("Bonjour Jacques !")
}else if(Nom=="Louis"){
  print("Bonsoir Louis!")
}else{
  print("Bonsoir monsieur!")
}
```

On peut ajouter autant de `else if` que l'on veut, mais il faut garder en tête que cette construction devient rapidement illisible et difficile à tester.

Pour garder une bonne lisibilité, on peut utiliser le `switch`:

```
switch(object,
  case = action
)
```

Il s'agit d'une fonction dont le premier argument est l'objet à tester (`object`), et les arguments suivants contiennent les cas à tester (`case`) et la valeur à retourner si on est dans ce cas. Donc lorsque `case==object`, alors on exécute l'`action` qui suit. Reprenons notre exemple précédent:

```
Nom= "Pierre"

switch(Nom,
```

```
  "Jacques" = print("Bonjour Jacques"),
  "Pierre"  = print("Bonjour Pierre"),
  "Louis"   = print("Bonjour Louis")
)
```

```
## [1] "Bonjour Pierre"
```

5.4 Fonctions utiles

Voici quelques petites fonctions utiles pour controler le flux d'exécution:

- `break`: permet de sortir d'une boucle, e.g.:

  ```
  for(i in 1:100){
    if(i>2){
      print(i)
    }else{
      break
    }
  }
  ```

  Ici on exécute la boucle `for` de 1 à 100, mais lorsque i est supérieur à 2, alors on termine la boucle. Cela revient à déclarer une boucle `for(i in 1:2)` dans ce cas.

- `next`: permet de passer à l'itération suivante de la boucle, e.g.:

  ```
  for(i in 1:3){
    if(i==2){
      next
    }else{
      print(i)
    }
  }
  ```

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

  Ici on exécute la boucle `for` de 1 à 3, mais lorsque i est égal à 2, alors on passe directement à `i= 3` sans exécuter le `print`.

Chapter 6

Fonctions

- Appel de fonction
- Function scope
- Programmer une fonction
- Sourcer un script

Disons que vous êtes en voyage au Brésil, où la monnaie est le Real, mais comme vous êtes habitué à payer Euro, vous aimeriez pouvoir convertir entre les deux simplement.

A l'heure où cette version du livre à été mis à jour pour la dernière fois, 1 Euro (€) valait 4.5899 Reals (R$)[1]. Grâce à R, vous pouvez calculer cela très facilement en créant un objet contenant la conversion :

##autres chapitres:

05- Environnements

06- dates

07- charactères (utiliser les charactères, chercher un charactère, remplacer,. . .)

09- Science des données: importer, manipuler des données * La famille des *apply * tidyverse (+ pipe) * data.table

10- Science des données: graphiques * Base plots * Ggplot2 * Interactivité (plotly)

11- Communications * Rmarkdown (knitr + pandoc)
* Sites web / blogs * Applications Shiny * Présentations

13- Errors, Warnings and Debugging

14- Packages

[1] données Yahoo Finance téléchargées grâce à la fonction `getQuote` du package `quantmod`

15- Git

Chapter 7

Environnements

Quand on crée un objet dans R, il est sauvegardé dans un environnement particulier.

Chapter 8

Travailler avec des dates

- POSIXct, POSIXlt
- lubridates
- ...

Chapter 9

Travailler avec des chaînes de charactère

utiliser les charactères, chercher un charactère, remplacer,...

- grep, grepl, match
- substr
- stringr
- ...

Chapter 10

Science des données: importer, manipuler des données

- La famille des *apply
- tidyverse (+ pipe)
- data.table

Chapter 11

Science des données: graphiques

- Base plots
- Ggplot2
- Interactivité (plotly)

Chapter 12

Communications

- Rmarkdown (knitr + pandoc)

- Sites web / blogs
- Applications Shiny
- Présentations

Chapter 13

Errors, Warnings and Debugging

Chapter 14

Packages

Writing your own R package.

Chapter 15

GIT

Using Git + Github|Gitlab

Chapter 16

Interfacing other languages

Using other languages from R such as C, C++ or Julia.