Data Science avec R

Fousseynou Bah

2019-03-18

Contents

Part I

Les basiques dans R

Chapter 1

Introduction

1.1 Un autre livre sur la *data science*! Vraiment?

En décidant d'écrire un livre sur la *data science*, j'ai longuement débattu dans ma propre tête, je me suis posé plusieurs questions dont une qui revenait constamment: "a-t-on vraiment besoin d'un autre livre sur la *data science*?" "N'en-t-on pas assez?" Avec le succès dont jouit la discipline, ce n'est certainement pas les ressources qui manquent, aussi bien en ligne que dans les librairies. Et surtout, je me demandais bien "qu'avais-je à dire qui n'avait pas été dit"? Et pourtant, quelques raisons m'ont poussé à reconsidérer ma position.

La première est assez égoïte. On n'apprend jamais aussi bien qu'en enseignant. Pour m'assurer que j'avais bien assimilé les connaissances que j'avais acquises dans ce domaine, il n'y avait rien de mieux que de me livrer à un exercice de pédagogue. Expliquer à d'autres ce que j'avais appris. N'est-ce pas là que réside l'ultime test pour un apprenant! C'est partant de cette idée que je me suis mis à faire des diapositives dans le cadre des mes enseignements. Très tôt, j'ai réalisé que les diapositives ne sauraient jouer leur rôle, qui est d'offrir un aperçu synthétique d'une idée développée par un narrateur, et satifaire l'apprenant qui souhaiterai obtenir des explications détaillées. Ce travail revient au narrateur, à défaut de qui l'on se tourne vers un manuel. Donc, il me fallait bien accompagner les diapositives d'un support plus détaillé pour mieux outiller mes étudiants.

La seconde raison est le contexte. Malgré l'abondance et la qualité des ressources disponibles sur la *data science* et malgré l'accès de plus en plus facile - coût faible et gratuité pour beaucoup -, il demeure que l'étudiant africain peut souvent se sentir éloigné du contexte à travers lequel la *data science* est présentée. Or, cell-ci est avant tout une discipline de contexte. Bien que mélangeant informatique, mathématiques, statistiques... et bien d'autres expertises, elle est avant tout un outil, mobilisée pour répondre à des questions. Et ces questions sont très contextuelles. Il ne fait aucun doute que le disponibilité et l'accéssibilité des données sur le monde industrialisé rend leur utilisation commode pour introduire la *data science* à un jeune africain est très commode. Mais la distance entre le contexte présenté et celui qui est vécu par le bénéficiaire pose un problème. Elle empêche l'appropriation de la discipline. De ce fait, je me suis trouvé dans ce constat une raison de m'engager dans ce projet et surtout de me forcer à utiliser des données sur le contexte local. Après tout, l'être humain n'est-il pas plus enclin à vous prêter attention quand vous lui parlez de lui-même?

1.2 La *data science*

Comme toute discipline qui connaît une expansion rapide, il est difficile de définir la *data science*. Elle est vaste et riche, tant de par les disciplines dont elle emprunte des morceaux pour se contituer en entité que de par les branches qu'elle pousse avec sa propre croissance.

Commençons par quelques exemples

Fait de la *data science*:

- l'économiste qui examine le niveau du PIB sur 30 ans et cherche à dégager des scénarii pour des futures évolutions;
- le sociologue qui s'appuie le taux de natalité et le taux de participation des femmes au marché du travail pour comprendre l'évolution de la place de la femme dans la société;
- le météorologue qui cherche à prédire la pluviométrie de la semaine à venir en modélisant les données historiques;
- l'épidémiologue qui cartographie le taux de prévalence du paludisme pour appuyer un programme stratégique;
- etc.

Le caractère transversale de la *data science* apparait ici quand on sait que ces individus sont de disciplines différentes et poursuivent des questions tout aussi distantes les unes des autres. Et pourtant, les données les réunissent tous. Ils ont chacun besoin de trouver dans celles-ci un appui pour améliorer leur propre compréhension du phénomène étudié, tester leurs hypothèses, fonder leurs recommandations ou même... reconforter leurs propres idées ou mieux s'armer pour rejetter celles de leurs adversaires (les données ne sont aussi neutres que celui qui les manipule!)

Selon Wikipédia, la *data science* est un champ interdisciplinaire qui utilise les méthode, processus, algorithmes et systèmes scientifiques pour extraire des données - tant structurées que non structurées - des informations utiles à la compréhension et à la prise de décision. De ce fait, elle s'appuie sur diverses méthodes (mathématiques, statistiques, informatiques, etc.) pour tirer des données une compréhension meilleure de phénomènes d'intérêt.

1.3 Le *data scientist*

Et le *data scientist* dans tout ça? Il est apparait désormais comme la perle rare. Un individu capable de parler aux hommes, aux machines et aux données. Aux:

- hommes, il pose les questions auxquelles il a la charge d'offrir des réponses.
- machines, il parle à travers des langages spécifiques (R, Python, Julia,...), des langages qui ressemblent à bien d'égards à ceux avec lesquels il s'entretient avec les humains car ils sont basés sur des règles précises et sont vivants et évolutifs;
- données, il applique des méthodes d'investigation où l'expérience, l'intuition, le sens artistique interviennent tout autant que la connaissance du domaine d'intervention. Dans les données disponibles, il cherche à séparer les bonnes des mauvaises, les utiles des nuisibles. A celles qu'il sélectionne, il cherche le bon format, la bonne structure. Sur celles qu'il retient, il teste des modèles, sans oublier la place importante de la visualisation à tous les niveaux. Bref, un vrai détective!

Face à la génération massive des données, le besoin de *data scientist* se fait pressant partout. De ce fait l'engouement ne manque pas pour les jeunes désireux de se lancer. Mais le portrait de super-homme généralement fait du *data scientist* (ne cherchez pas plus loin que les lignes d'en dessus!), l'on peut croire qu'il faut être spécial pour embrasser la profession. Du tout! Celà dit, certaines compétences sont utiles.

Alors, qu'est-ce qu'il faut pour être *data scientist*?

- pas nécéssairement un diplôme avancé en mathématiques ou en statistiques... quoiqu'il est utile de maîtriser des concepts de bases (les concepts algébriques comme le vecteur, la matrice,... et les notions statistiques comme la moyenne, l'écart-type, etc.);
- pas forcément un diplôme en informatique ou en programmation... quoiqu'il est utile de connaître les notions de bases (qu'est-ce qu'un objet, un environnement? quels types d'objets peut-on manipuler dans un environnement donnée...?);

- une connaissance avérée dans un domaine spécifique dans lequel l'on peut soulever des questions, mobiliser des outils théoriques auxquels on confronte les résultats de l'analyse conduite sur les données;
- un esprit curieux, quelle que soit l'avenue que l'on emprunte.

Vous pourrez avoir une meilleure idée en surfant sur le net (Google est votre ami!)

Chapter 2

Introduction à R

2.1 R

2.1.1 Qu'est-ce que c'est que R?

Voici basiquement ce que Wikipédia dit. R est un langage de programmation et un logiciel gratuit et libre. Il est surtout utilisé pour le développement de programmes statistiques et des analyses de données. Il gagne en popularité depuis quelques années avec l'émergence de la *data science* et du fait qu'il est gratuit et ouvert (*open-source*). R est née d'un projet de recherche mené par deux chercheurs, Ross Ihaka et Robert Gentleman à l'université d'Auckland (Nouvelle-Zélande) en 1993. En 1997 est mis en place le *Comprehension R Archive Network (CRAN)* qui centralise les contributions au projet

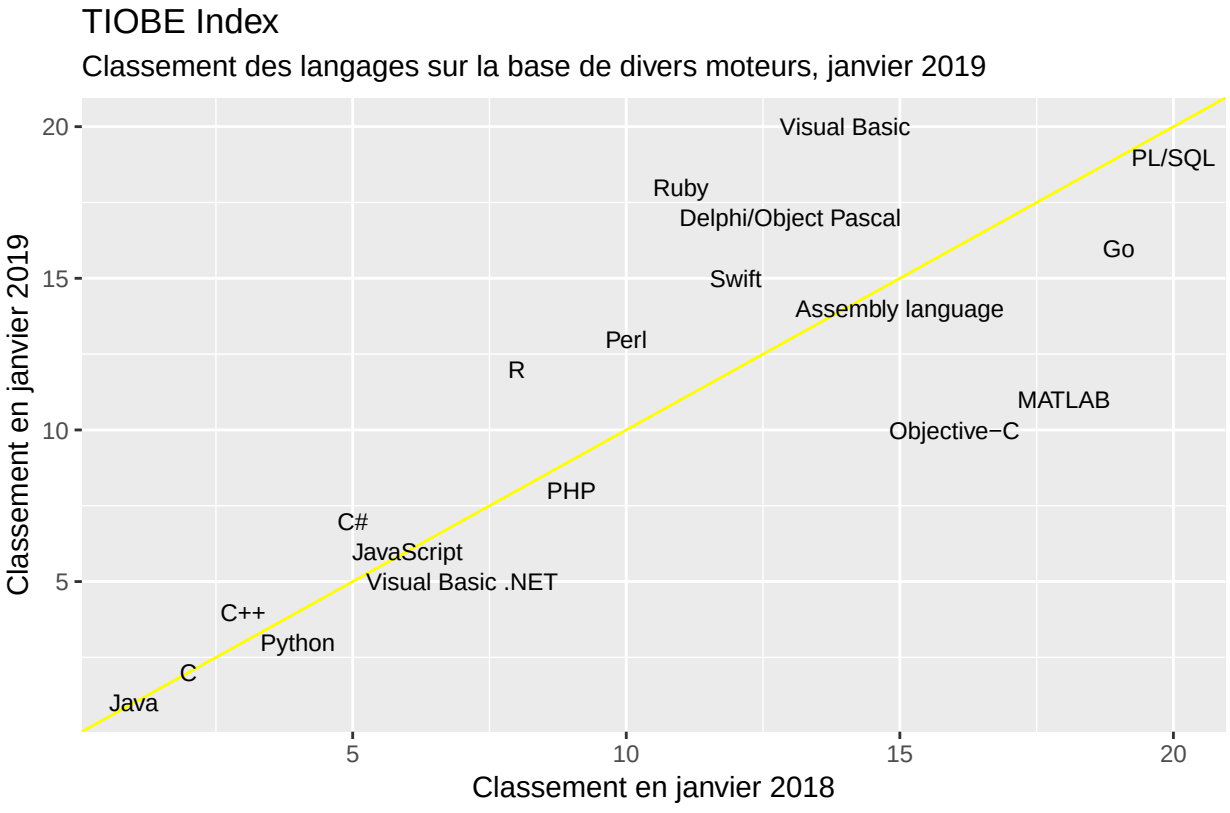
Depuis le projet connaît une croissance soutenue, grâce à des contributions de la part de milliers de personnes à travers le monde.

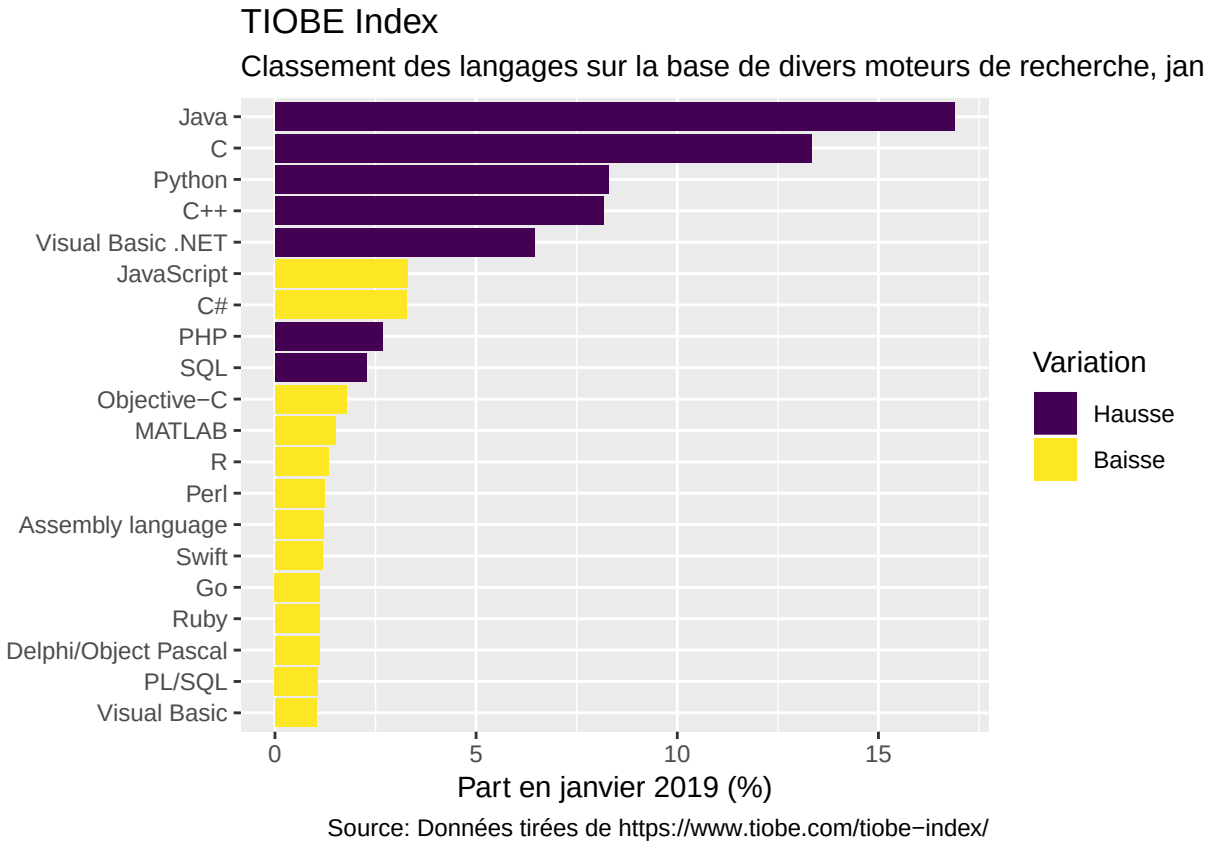
2.1.2 Pourquoi R?

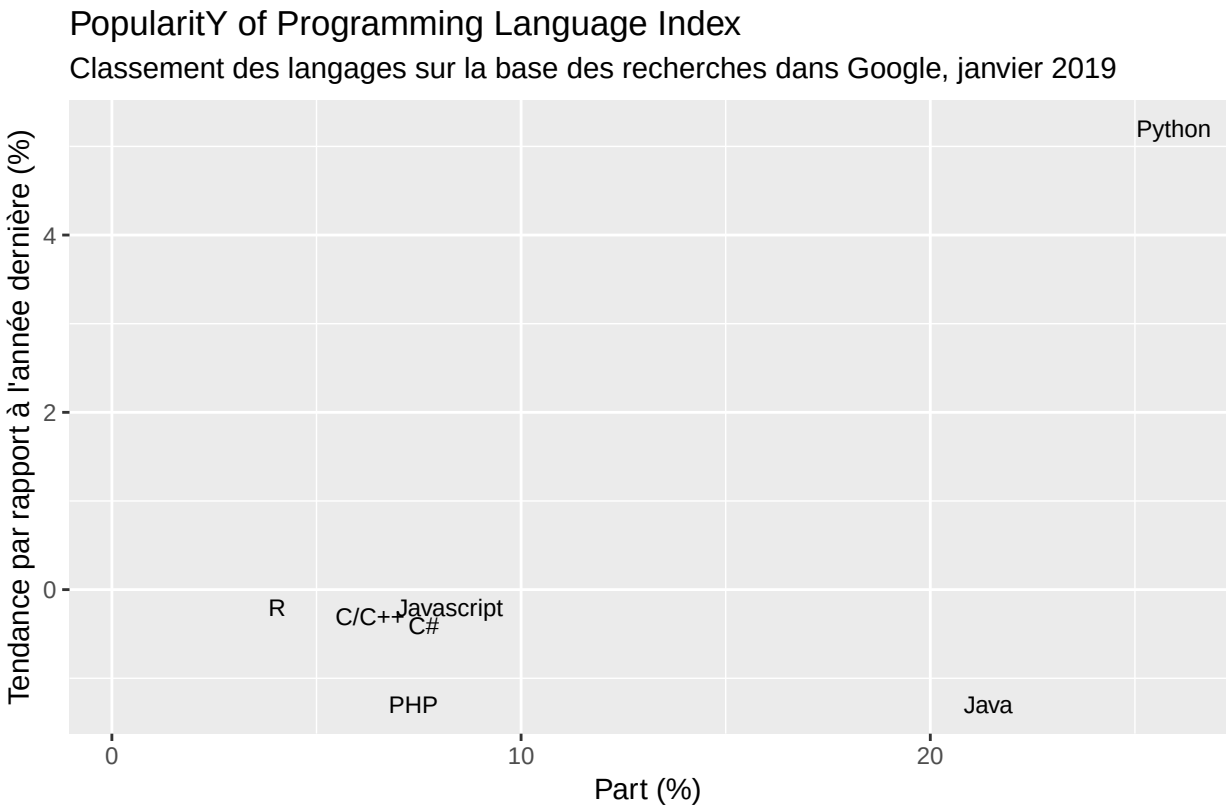
Pour un apprenti *data scientist*, le choix du langage et/ou du programme est une décision critique. Considérant le temps qu'il investira en apprentissage et le retour qu'il espéra à travers l'utilisation de ses nouvelles connaissances dans sa profession, il est utile de considerer divers critères dont:

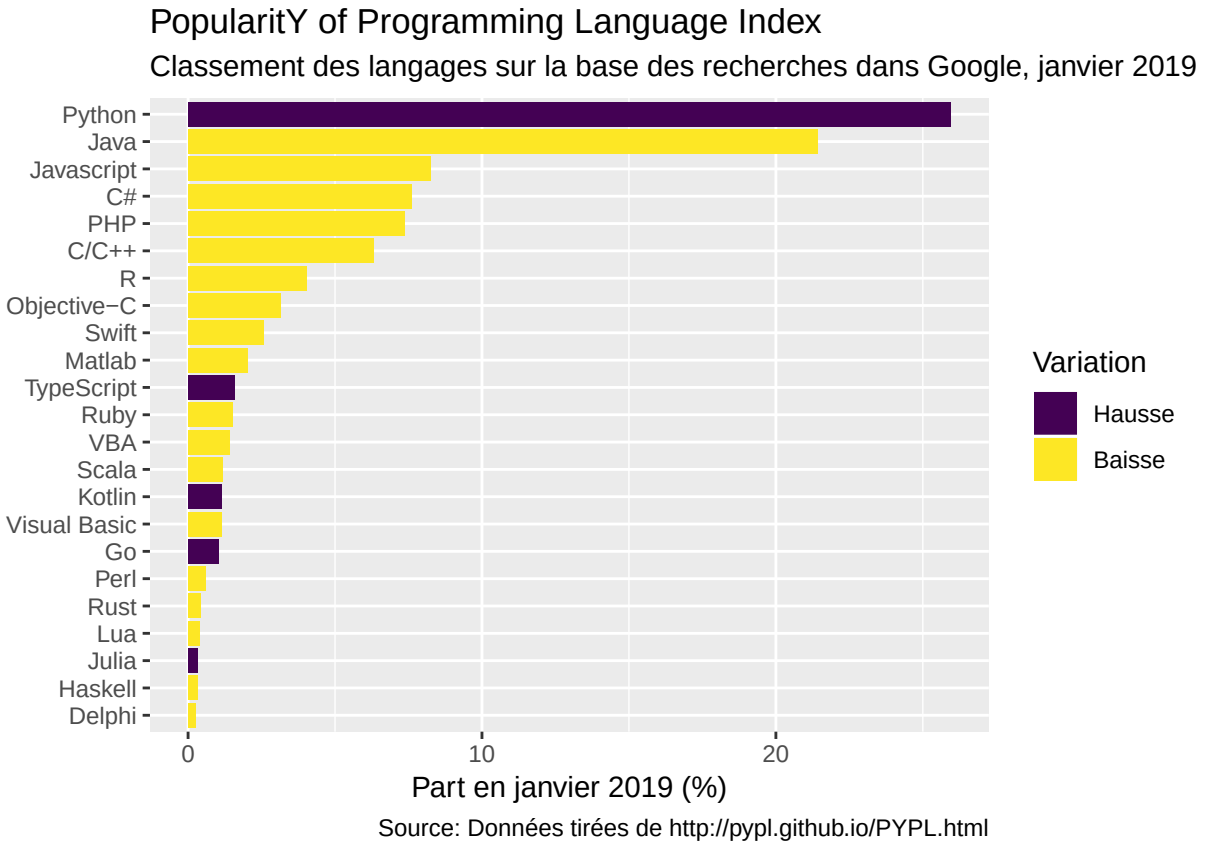
- l'accessibilité de l'outil en termes de coûts: tous les langages de programmation ne sont pas gratuits comme R! Certains coûtent... chers mêmes ;
- l'accessibilité du langage en termes de syntaxe: R est très compréhensible (surtout pour quelqu'un qui se retrouve un peu avec la langue anglaise);
- la popularité du langage parmi les paires: tout le monde s'est mis à l'anglais, même dans les pays où ce n'est pas la langue dominante. N'est-ce pas? De la même façon, il est important pour le *data scientist* d'embrasser un langage qui est aussi utilisé par ceux avec lesquels il sera amené à collaborer. A ce niveau, R est très populaire.
- la dynamique de développement du langage: le langage étant un investissement en soit, il est important de miser sur ceux qui présentent un avenir. Et ceux-ci sont ceux qui mutent avec la technologie et les besoins des utilisateurs. A ce niveau encore, R présente des arguments. Il dispose du réseau *CRAN* alimenté par des milliers de contributeurs, divers aussi bien de par leur position dans le monde que de par leur discipline.

2.1.3 R dans l'écosystème des langages




TIOBE Index
Classement des langages sur la base de divers moteurs de recherche, jan
Java
C
Python
C++
Visual Basic .NET
JavaScript
C#
PHP
SQL
Objective−C
MATLAB
R
Perl
Assembly language
Swift
Go
Ruby
Delphi/Object Pascal
PL/SQL
Visual Basic
0
5
10
15
Part en janvier 2019 (%)
Variation
Hausse
Baisse
Source: Données tirées de https://www.tiobe.com/tiobe−index/


PopularitY of Programming Language Index
Classement des langages sur la base des recherches dans Google, janvier 2019
Tendance par rapport à l'année dernière (%)
4
2
0
Python
R
C/C++
Javascript
C#
PHP
Java
0
10
20
Part (%)
Source: Données tirées de http://pypl.github.io/PYPL.html



Ce qui apparait des différentes figures, c'est que R parvient à se tailler une place parmi les langages les plus populaires au monde. Et celà, malgré le fait que c'est une langage spécialisée. Si sur les dix dernières années, le langage s'est enrichi avec la diversification de ses contributeurs, il reste à la base un langage élaboré par des statisticiens pour des statisticiens. De ce fait, il est excéllent pour l'analyse de données, mais fort peu utile pour certaines tâches... comme le développement d'un site web.

2.2 RStudio

2.2.1 Qu'est-ce que c'est que RStudio

- C'est une IDE (*Integrated Development Environment*) ou Environnement Intégré de Développement
- Il sert d'interface entre R et l'utilisateur, offre à celui diverses commodités d'utilisation

Maintenant, vous avez les outils nécéssaires pour commencer la formidable aventuRe!

Chapter 3

Objets dans R

3.1 Introduction

Dans ce chapitre, nous allons :

- introduire la notion d'objet dans R;
- présenter un certain nombre d'entre eux;
- et illustrer avec quelques exemples.

Que nous faudra-t-il?

- R (évidemment)
- RStudio (de préférence)

3.2 La notion d'objet dans R

3.2.1 Qu'est-ce qu'un objet?

Dans R, un objet représente un concept, une idée. Il se matérialise par une entité qui possède sa propre identité. Dans celle-ci, l'on compte deux aspects majeurs:

- la structure interne;
- le comportement.

Illustrons pour comprendre. Commençons par créer des objets.

Imaginez que vous voulez créer et conserver des bouts d'information dans R sur les présidents qui se sont succédés à la tête de la République du Mali. Commençons par le premier président, Mobido Keïta. Créeons des objects relatifs à son nom et son prénom.

```
nom <- "Keïta"
prenom <- "Mobido"
```

L'acte d'assignation d'une valeur à un objet se fait par le signe `<-` qui est équivalent à `=`. Chez beaucoup d'utilisateurs, la préférence est donnée à la première. Ceci peut se comprendre par le fait qu'avec `<-`, l'acte d'assignation se différencie plus facilement d'autres utilisations du signe `=` (dont notamment à l'intérieur de fonctions). Désormais, ces informations sont stockées dans notre environnement. Pour vérifier appellons-les! Ceci revient à les saisir dans notre console et à taper "Entrée"!

```
nom
```

```
## [1] "Keïta"
```

```
prenom
```

```
## [1] "Mobido"
```

3.2.2 Oranges et bananes

Enrichissons notre environnement des objets additionnels. Ajoutons l'année d'accession au pouvoir. Appelons cet objet `annee_arrivee_pouvoir`.

```
annee_arrivee_pouvoir <- 1960
```

Comme pour les objets précédent, celui-ci aussi peut être invoqué:

```
annee_arrivee_pouvoir
```

```
## [1] 1960
```

A l'instar de l'orange et de la banane, fort différentes bien que toutes les deux des fruits, ici aussi nos objets diffèrent. Peut-on les additionner?

```
nom + annee_arrivee_pouvoir
```

```
## Error in nom + annee_arrivee_pouvoir: non-numeric argument to binary operator
```

Non, en l'occurence! On a un message d'erreur. R, c'est comme la vraie vie! Les oranges et les bananes ne se mélangent.

3.2.3 Ce qui se ressemblent s'assemblent

Les choses qui diffèrent ne s'assemblent pas Illustration d'une propriété des objets: le comportement. Regardons les choses qui marchent.

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

Maintenant stockons ce résultat dans un objet.

```
objet1 <- 1 + 1
```

Créons-en un autre.

```
objet2 <- 2 + 2
```

Amusons à faire diverses opérations avec ces deux objets

```
objet1 + objet2
```

```
## [1] 6
```

```
objet1 - objet2
```

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

```
objet1 * objet2
```

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

```
objet1 / objet2
```

```
## [1] 0.5
```

Bref, vous voyez l'idée! Les propriétés des objets déterminent les intéractions auxquelles elles se prêtent. Et ce sont justement ces intéractions qui constituent le coeur de l'analyse de données. D'où l'importance de la notion d'objet.

3.2.4 Quelques objets dans R

Dans R, l'on distingue plusieurs types d'objets. Nous en retiendrons ici 5, qui nous serons utiles tout le long de l'ouvrage. Il s'agit des:

- caractères (*strings* en anglais);
- nombres (entiers ou réels);
- dates;
- valeurs logiques qui ne prennent que deux valeurs: `TRUE` (vrai) ou `FALSE` (faux);
- facteurs qui sont un format spécial dans R prévu pour les variables catégorielles.

Revenons à notre exemple *présidentiel*! Nous avons déjà le nom et le prénom...

```
# Caractères
nom <- "Keïta"
prenom <- "Mobido"
```

...ainsi que l'année d'arrivée au pouvoir.

```
# Nombre
annee_arrivee_pouvoir <- 1960
```

Ajoutons la date de naissance,

```
# Date
date_naissance <- as.Date("1915-06-04")
```

une valeur logique indiquant s'il a eu un parcours militaire ou pas,

```
# Valeur logique
parcours_militaire <- FALSE
```

et enfin la région de naissance.

```
# Facteur
region_naissance <- as.factor("Bamako")
```

3.2.5 La notion de classe et de type

Quand on a faire à des objets dont on ignore l'identité, l'on peut s'appuyer la fonction `class`. Celle-ci permet de connaître la classe de l'objet. La classe est un attribut qui contribue à la formation de l'idée d'un objet. "Avec quoi se mélange-t-il?" "A quelles règles de transformation se soumet-il?" Basiquement, la classe dicte les principes régissant la manipulation de cet objet. Testons la fonction sur les objets que nous venons de créer pour bien confirmer les identités qu'on leur a attribuées.

```
class(nom)
```

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

```
class(prenom)
```

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

```
class(annee_arrivee_pouvoir)
```

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

```
class(date_naissance)
```

```
## [1] "Date"
```

```
class(parcours_militaire)
```

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

```
class(region_naissance)
```

```
## [1] "factor"
```

Nous voyons que les résultats sont bien conformes aux dénominations que nous leur avons données plus haut.

Dans R, il y a aussi la fonction `typeof` (ou `mode`, mais nous resterons avec la première) qui permet de connaître le mode de stockage d'un objet. Testons!

```
typeof(nom)
```

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

```
typeof(prenom)
```

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

```
typeof(annee_arrivee_pouvoir)
```

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

```
typeof(date_naissance)
```

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

```
typeof(parcours_militaire)
```

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

```
typeof(region_naissance)
```

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

Si pour les objets `nom` et `prenom` qui sont des lettres, la classe et le type se confondent, la question est tout autre pour d'autres objets. Regardons `region_naissance`, par exemple. En termes de classe, c'est un facteur. Par contre, en terme de type, R l'a coercé en entier (*integer*).

Les types sont assez génériques car présentant pratiquement les mêmes nomenclatures d'un langage à un autre. Dans R, nous allons plus nous intéressér aux types suivants:

- logique (*logical*);
- entier (*integer*);
- réel (*double*);
- caractère (*character*);
- liste (*list*);
- valeur nulle (*NULL*).

Les objets que nous avons vus là peuvent être pensés comme des briques. Ils entrent à leur tour dans la formation d'autres objets qui varient les uns des autres. Tout comme les constructions peuvent différer entre elles.

3.2.6 Vers d'autres types d'objets

Les objets que nous allons voir ici peuvent être pensés comme des objets composites. Nous en verrons quatre types:

- le vecteur;
- la matrice;
- le *data frame* (cadre de données ou données rectangulaires);
- la liste.

3.3 Vecteurs

3.3.1 Qu'est-ce qu'un vecteur?

De façon très simple, un vecteur est un ensemble d'éléments de même nature. Revenons à notre exemple pour mieux comprendre. Nous avons défini l'objet `nom`, n'est-ce pas? Est-ce un vecteur? A quoi peut-on voir si c'est un vecteur ou pas? La réponse:

```
is.vector(nom)
```

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

Donc nous avons créé des vecteurs depuis longtemps et on voit qu'un objet d'un seul élément peut être un vecteur. Maintenant, comptons le nombre d'éléments que compte ce vecteur.

```
length(nom)
```

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

C'est vraiment un singleton qu'on a là... pour le moment!

3.3.2 Créons-en, des vecteurs!

Décidons d'étendre nos observations à tous les présidents de la République du Mali. En voici de quoi nous faire revisiter nos livres d'histoire... ou juste consulter Wikipedia!

```
# Omettons les périodes de transition (la valeur pédagogique est ce qui est recherché ici!)
nom <- c("Keïta", "Traoré", "Konaré", "Touré", "Keïta")
prenom <- c("Modibo", "Moussa", "Alpha Oumar", "Amadou Toumani", "Ibrahim Boubacar")
date_naissance <- as.Date(c("1915-06-04", "1936-09-25", "1946-02-02", "1948-11-04", "1945-01-29"))
region_naissance <- as.factor(c("Bamako", "Kayes", "Kayes", "Mopti", "Koutiala"))
annee_arrivee_pouvoir <- c(1960, 1968, 1992, 2002, 2013)
parcours_militaire <- c(FALSE, TRUE, FALSE, TRUE, FALSE)
```

Maintenant, expérimentons! Commençons avec `nom` que nous avons écrasé avec de nouvelles valeurs.

```
is.vector(nom)
```

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

```
length(nom)
```

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

```
class(nom)
```

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

```
typeof(nom)
```

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

“nom” est un **vecteur**, un ensemble de **5** éléments en **charactères**. Amusez-vous à expérimenter avec les autres vecteurs.

3.3.3 Vrai pour un, vrai pour plusieurs

Vous vous rappelez que plus haut, nous voyions que les opérations n'étaient pas possibles entre de différentes natures. Et bien, cette règle, valable à l'échelle des objets élémentaires, l'est aussi aux échelles supérieures.

Prenons nos données et cherchons à déterminer l'âge des présidents à leur arrivée au pouvoir. On a les éléments nécéssaires pour ce faire, la date de naissance et l'année d'arrivée au pouvoir. Toutefois, ces deux vecteurs ne sont pas de même nature.

```
age_arrivee_pouvoir <- annee_arrivee_pouvoir - date_naissance
```

```
## Error in `-.Date`(annee_arrivee_pouvoir, date_naissance): can only subtract from "Date" objects
```

On a un message d'erreur. Apparemment l'opération n'est pas possible. Il faudrait procéder à une transformation: déduire de la date de naissance l'année pour conduire l'opération avec celle-ci.

```
annee_naissance <- as.numeric(format(date_naissance,'%Y'))
```

Testons si le nouveau vecteur est de même nature de celui de `annee_arrivee_pouvoir`.

```
class(annee_naissance)
```

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

Maintenant, nous pouvons procéder à l'opération

```
age_arrivee_pouvoir <- annee_arrivee_pouvoir - annee_naissance
age_arrivee_pouvoir
```

```
## [1] 45 32 46 54 68
```

On le confirme: les oranges et les bananes ne se mélangent pas. Toutefois, R nous fait souvent des cocktails de fruits en coerçant certains éléments. Imaginons que l'on veuille rassembler le prénom et le nom dans un seul vecteur. Collons ces éléments à l'aide d'une fonction de base dans R, `paste`, (ne vous en faites pas, vous ferez progressivement connaissance avec les fonctions!)

```
prenom_nom <- paste(prenom, nom)
prenom_nom
```

```
## [1] "Modibo Keïta"           "Moussa Traoré"
## [3] "Alpha Oumar Konaré"     "Amadou Toumani Touré"
## [5] "Ibrahim Boubacar Keïta"
```

On peut être enclin à dire que ceci est passé sans souci parce que `nom` et `prenom` sont tous les deux des vecteurs en caractères. Maintenant, et si l'on ajoutait l'année d'arrivée au pouvoir?

```
prenom_nom_age <- paste(prenom, nom, ",", age_arrivee_pouvoir)
prenom_nom_age
```

```
## [1] "Modibo Keïta , 45"           "Moussa Traoré , 32"
## [3] "Alpha Oumar Konaré , 46"     "Amadou Toumani Touré , 54"
## [5] "Ibrahim Boubacar Keïta , 68"
```

C'est passé comme une lettre à la poste (pour la génération email, voici ce qu'est la poste). Car R a une hiérarchie entre les objets. Avant de déclarer forfait avec un message d'erreur, il tente tant bien que mal

d'exécuter l'opération. Sur la base de cette hiérarchie, il coerce certains éléments à se conformer à d'autres, partant du plus flexible au moins flexible: `logique` < `entier` < `réel` < `caractère`. Pour comprendre ça, créons un vecteur de valeurs logiques.

```
vecteur_logique <- c(TRUE, FALSE)
```

Confirmons sa classe.

```
class(vecteur_logique)
```

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

Ajoutons un troisième élément qui sera un entier. Disons 1.

```
vecteur_entier <- c(vecteur_logique, 1)
```

Qu'obtenons-nous?

```
vecteur_entier
```

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

Des entiers! R a coercé `TRUE` en 1 et `FALSE` en 0.

```
class(vecteur_entier)
```

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

Ajoutons un quatrième élément, cette fois-ci une réel: 2.5 (dans R, comme en anglais, les décimales viennent après un `.`, pas une `,`, qui sert plutôt de séparateur de milliers).

```
vecteur_reel <- c(vecteur_entier, 2.5)
vecteur_reel
```

```
## [1] 1.0 0.0 1.0 2.5
```

```
class(vecteur_reel)
```

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

La mutation se voit au fait que R a affecté aux trois premiers éléments des décimales, bien qu'initialement c'étaient des entiers. Maintenant, ajoutons un cinquième élément: un prénom.

```
vecteur_caractere <- c(vecteur_reel, "Mariam")
vecteur_caractere
```

```
## [1] "1"      "0"      "1"      "2.5"    "Mariam"
```

```
class(vecteur_caractere)
```

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

Là aussi, la coercion se voit.

3.3.4 Nommer les éléments d'un vecteur

Jusque là, ce sont des objets à part intégrale que nous avons nommés. On les a assignés des noms pour les garder dans notre environnement de travail. Maintenant, nous allons donner un nom aux éléments de vecteur. Dressons l'analogie suivante. Notre environnement dans R est comme une rue. Dans celle-ci, nous avons des concessions dont les portes sont toutes numérotées: ce sont les noms des objets. A l'intérieur des concessions, nous avons des individus: ce sont les éléments à l'intérieur de nos objets. Tout comme ces individus portent des prénoms, nous pouvons donner des appélations aux éléments contenus dans nos objets.

Considerons que nous voulons associer à chaque date de naissance le nom du président en question.

```
names(date_naissance) <- prenom_nom
```

Voyons ce que ça donne

```
date_naissance
```

```
##           Modibo Keïta          Moussa Traoré     Alpha Oumar Konaré
##           "1915-06-04"           "1936-09-25"           "1946-02-02"
##   Amadou Toumani Touré Ibrahim Boubacar Keïta
##           "1948-11-04"           "1945-01-29"
```

C'est beau non! Il est intéressant de noter que quand on conduit des opérations sur des vecteurs aux éléments nommés, le résultat peut hériter de ces propriétés. Reprenons l'opération de déduction de l'âge à l'arrivée au pouvoir. Rappelons les deux vecteurs.

```
annee_naissance
```

```
## [1] 1915 1936 1946 1948 1945
```

```
annee_arrivee_pouvoir
```

```
## [1] 1960 1968 1992 2002 2013
```

Nommons juste un des deux vecteurs.

```
names(annee_naissance) <- prenom_nom
annee_naissance
```

```
##           Modibo Keïta          Moussa Traoré     Alpha Oumar Konaré
##                   1915                   1936                   1946
##   Amadou Toumani Touré Ibrahim Boubacar Keïta
##                   1948                   1945
```

Procédons à l'opération.

```
age_arrivee_pouvoir <- annee_arrivee_pouvoir - annee_naissance
age_arrivee_pouvoir
```

```
##           Modibo Keïta          Moussa Traoré     Alpha Oumar Konaré
##                     45                     32                     46
##   Amadou Toumani Touré Ibrahim Boubacar Keïta
##                     54                     68
```

Le vecteur `age_arrivee_pouvoir` a hérité des noms d'éléments.

Cette règle n'est pas toutefois immuable. Quand les éléments sont coercés à prendre une autre classe que leur classe de départ, ils peuvent perdre leur nom, qui n'est qu'un de leurs attributs (qui sont subordonnés à leur classe). Reprenons la déduction de l'année de naissance à partir de la date de naissance.

```
annee_naissance <- format(date_naissance,'%Y')
annee_naissance
```

```
##           Modibo Keïta          Moussa Traoré     Alpha Oumar Konaré
##                 "1915"                 "1936"                 "1946"
##   Amadou Toumani Touré Ibrahim Boubacar Keïta
##                 "1948"                 "1945"
```

```
class(annee_naissance)
```

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

Ici, l'année n'a pas été coercé. Elle a été extraite par la fonction sous format de caractères. En voulant conformer le vecteur à la classe de nombre (on descend dans la hiérarchie), on coerce les éléments.

```
annee_naissance <- as.numeric(format(date_naissance,'%Y'))
annee_naissance
```

```
## [1] 1915 1936 1946 1948 1945
```

```
class(annee_naissance)
```

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

Avec la coercion, les noms se perdent. Il est donc utile de se rappeler que les noms d'éléments ne sont pas immunes à la coercion. Toutefois, quand les opérations se passent entre des éléments de même nature, les noms sont bien saufs!

3.3.5 Opérations sur vecteurs

3.3.5.1 Sélection explicite

Il arrive souvent qu'on ne soit intéréssée que par un élément précis d'un vecteur. Peut-être l'on souhaite connaître seulement l'âge du premier président lors de son accès au pouvoir. C'est le premier élément du vecteur `age_arrivee_pouvoir`.

```
age_arrivee_pouvoir[1]
```

```
## Modibo Keïta
##           45
```

Peut-être nous voulons l'information pour le 1er et le 3ème présidents. Ce sont les 1er et 3ème éléments du vecteur.

```
age_arrivee_pouvoir[c(1, 3)]
```

```
##       Modibo Keïta Alpha Oumar Konaré
##                 45                 46
```

Peut-être que nous voulons l'information du 1er au 3ème président.

```
age_arrivee_pouvoir[c(1:3)]
```

```
##       Modibo Keïta      Moussa Traoré Alpha Oumar Konaré
##                 45                 32                 46
```

On peut aussi souhaiter exclure certains éléments. Imaginons que l'on veuille seulement regarder les informations sans les deux derniers éléments du vecteur.

```
age_arrivee_pouvoir[-c(4, 5)]
```

```
##       Modibo Keïta      Moussa Traoré Alpha Oumar Konaré
##                 45                 32                 46
```

Le signe `[]` agit comme une porte d'entrée à l'intérieur du vecteur tandis que les chiffres indiqués sont des index qui indiquent la position des éléments intérêt. L'opération peut consister en une sélection ou une exclusion selon que l'opérateur `c` est précédé du signe `-` (exclusion) ou pas (sélection).

3.3.5.2 Sélection à partir de logiques

La sélection à l'intérieur d'un vecteur peut aussi se faire à partir de valeurs logiques. L'on peut poser des critères auxquels certains éléments répondraient. Et sur la base de leur confirmité au(x) critère(s) posé(s), l'on pourra effectuer la sélection (ou l'exclusion). Cette fonctionnalité est très utile car elle permet au data scientist d'utiliser les questions qu'il se pose pour avoir un aperçu des données qui sont à sa disposition.

Explorons la question suivante: quels sont les présidents arrivés au pouvoir avant l'âge de 50 ans?

```
president_avant_50ans <- age_arrivee_pouvoir < 50
president_avant_50ans
```

```
##           Modibo Keïta          Moussa Traoré     Alpha Oumar Konaré
##                   TRUE                   TRUE                   TRUE
##   Amadou Toumani Touré Ibrahim Boubacar Keïta
##                  FALSE                  FALSE
```

On transforme maintenant ce vecteur de valeurs logiques en outil de sélection. On peut soir regarder le nom de ces présidents:

```
prenom_nom[president_avant_50ans]
```

```
## [1] "Modibo Keïta"       "Moussa Traoré"      "Alpha Oumar Konaré"
```

Le résultat nous donne le nom des présidents pour lesquels le vecteur de valeurs logiques affiche `TRUE`. On peut utiliser le même critère sur d'autres vecteurs. Voyons le vecteur d'âge d'arrivée au pouvoir: quel âge avec les présidents qui sont arrivés au pouvoir avant l'âge de 50 ans?

```
age_arrivee_pouvoir[age_arrivee_pouvoir < 50]
```

```
##       Modibo Keïta      Moussa Traoré Alpha Oumar Konaré
##                 45                 32                 46
```

Pendant qu'on y est, dans quelle région sont-ils nés?

```
names(region_naissance) <- prenom_nom # nommons d'abord les éléments
region_naissance[age_arrivee_pouvoir < 50]
```

```
##       Modibo Keïta      Moussa Traoré Alpha Oumar Konaré
##             Bamako              Kayes              Kayes
## Levels: Bamako Kayes Koutiala Mopti
```

Vous comprenez la logique...

3.3.5.3 Statistiques sommaires

Une fois le vecteur constitué, il peut en lui-même faire l'objet d'opérations diverses. Posons diverses questions avec le vecteur `age_arrivee_pouvoir`. Quelle est la moyenne d'âge d'arrivée au pouvoir sur la base des éléments disponibles?

```
mean(age_arrivee_pouvoir)
```

```
## [1] 49
```

```
# une alternative donnat le même résultat.
sum(age_arrivee_pouvoir)/length(age_arrivee_pouvoir)
```

```
## [1] 49
```

Quel est l'âge d'arrivée au pouvoir le plus bas ?

```
min(age_arrivee_pouvoir)
```

```
## [1] 32
```

Quel est l'âge d'arrivée au pouvoir le plus élevé ?

```
## [1] 68
```

3.3.5.4 Ajustement et recyclage

Maintenant, revenons-en un peu aux opérations entre deux vecteurs. Imaginez maintenant, que l'on veuille connaître l'âge auquel les présidents ont quitté le pouvoir. Rappellons d'abord le vecteur `age_arrivee_pouvoir` que nous avions déjà généré.

```
age_arrivee_pouvoir
```

```
##           Modibo Keïta          Moussa Traoré     Alpha Oumar Konaré
##                     45                     32                     46
##   Amadou Toumani Touré Ibrahim Boubacar Keïta
##                     54                     68
```

Construisons ensuite un vecteur avec le nombre d'années passées au pouvoir.

```
duree_au_pouvoir <- c(8, 23, 10, 10)
```

Maintenant calculons l'année de départ du pouvoir en ajoutant à l'âge d'arrivée au pouvoir le nombre d'années qui y ont été passé.

```
age_depart_pouvoir <- age_arrivee_pouvoir + duree_au_pouvoir
```

```
## Warning in age_arrivee_pouvoir + duree_au_pouvoir: longer object length is
## not a multiple of shorter object length
```

```
age_depart_pouvoir
```

```
##           Modibo Keïta          Moussa Traoré     Alpha Oumar Konaré
##                     53                     55                     56
##   Amadou Toumani Touré Ibrahim Boubacar Keïta
##                     64                     76
```

Parvenez-vous à décéler l'erreur?

Nous avons additionné un vecteur de 5 éléments, `age_arrivee_pouvoir`, avec un vecteur de 4 éléments, `duree_au_pouvoir`. R a recyclé le premier élément du vecteur court (4) pour poursuivre l'opération d'addition entre les deux vecteur et l'a ajouté au 5ème élément du vecteur long. D'où la valeur de `76`.

```
68 + 8
```

```
## [1] 76
```

R avertit, mais conduit l'opération. De ce fait, même si les opérations entre vecteurs de même nature s'exécute sans problème majeur, il reste utile de vérifier leur longueur. Pour éviter le recyclage, il faudrait ne pas laisser de vide dans le vecteur court, s'assurer que les vecteurs impliqués dans l'opération sont de la même taille. Sur nos 5 présidents, nous n'avons pas ajouté le nombre d'années passées au pouvoir (car le mandat est encore en cours pendant la rédaction du présent document). Une solution serait de remplir la position dans le vecteur avec la valeur `NA`, indiquant un valeur manquante. Ajoutons-le.

```
duree_au_pouvoir <- c(duree_au_pouvoir, NA)
duree_au_pouvoir
```

```
## [1]  8 23 10 10 NA
```

Reprenons l'opération.

```
age_depart_pouvoir <- age_arrivee_pouvoir + duree_au_pouvoir
age_depart_pouvoir
```

```
##           Modibo Keïta          Moussa Traoré     Alpha Oumar Konaré
##                     53                     55                     56
##   Amadou Toumani Touré Ibrahim Boubacar Keïta
##                     64                     NA
```

Ne sachant pas comment faire l'opération pour la dernière entrée du vecteur car l'un des composante est `NA`, R reconduit cette valeur. Ainsi le recyclage est évité.

3.4 Matrices

3.4.1 La matrice, un ensemble de vecteurs

De façon basique, une matrice n'est autre qu'une collection de vecteurs. De ce fait, elle hérite d'une propriété fondamentale du vecteur: ne peuvent former une matrice que des éléments de même nature.

Retournons à notre exemple. Associons les noms et prénoms en une matrice car tous deux sont en charactères.

Solution 1: coller horizontalement les deux vecteurs

```
prenom_nom_hmatrix <- rbind(prenom, nom)
prenom_nom_hmatrix
```

```
##        [,1]     [,2]     [,3]          [,4]             [,5]
## prenom "Modibo" "Moussa" "Alpha Oumar" "Amadou Toumani" "Ibrahim Boubacar"
## nom    "Keïta"  "Traoré" "Konaré"      "Touré"          "Keïta"
```

Solution 2: coller verticalement les deux vecteurs

```
prenom_nom_vmatrix <- cbind(prenom, nom)
prenom_nom_vmatrix
```

```
##      prenom             nom
## [1,] "Modibo"           "Keïta"
## [2,] "Moussa"           "Traoré"
## [3,] "Alpha Oumar"      "Konaré"
## [4,] "Amadou Toumani"   "Touré"
## [5,] "Ibrahim Boubacar" "Keïta"
```

On voit que la matrice hérite des noms donnés aux différents vecteurs.

Bien que l'on puisse créer une matrice en combinant différents vecteurs, horizontalement avec `rbind` ou verticalement avec `cbind`, il existe aussi une fonction qui permet de créer directement une matrice: `matrix`. Il est toutefois utile de connaitre l'ordre de positionnement des éléments. Reprenons la création avec `matrix`, horizontalement...

```
prenom_nom_hmatrix
```

```
##        [,1]     [,2]     [,3]          [,4]             [,5]
## prenom "Modibo" "Moussa" "Alpha Oumar" "Amadou Toumani" "Ibrahim Boubacar"
## nom    "Keïta"  "Traoré" "Konaré"      "Touré"          "Keïta"
```

...et verticalement

```
prenom_nom_vmatrix <- matrix(c("Modibo", "Keïta",
                               "Moussa", "Traoré",
                               "Alpha Oumar", "Konaré",
                               "Amadou Toumani", "Touré",
                               "Ibrahim Boubacar", "Keïta"),
                             byrow = TRUE,
                             ncol = 2,
                             dimnames = list(NULL, c("prenom", "nom"))
                             )
prenom_nom_vmatrix
```

```
##      prenom             nom
```

```
## [1,] "Modibo"           "Keïta"
## [2,] "Moussa"           "Traoré"
## [3,] "Alpha Oumar"      "Konaré"
## [4,] "Amadou Toumani"   "Touré"
## [5,] "Ibrahim Boubacar" "Keïta"
```

Dans la fonction `matrix`, les arguments `nrow`, `ncol`, `byrow` et `bycol` servent à celà. Fonction, arguments... ne vous en faites pas! On y viendra.

3.4.2 La matrice, un objet bidimensionnel

La matrice n'est pas seulement un ensemble de vecteurs. Elle se distingue aussi de par sa bidimensionnalité. Pendant que le vecteur est soit une ligne de plusieurs éléments (*1 x n*) soit une colonne de plusieurs éléments éléments (*n x 1*), la matrice, elle, est faite de plusieurs lignes (*n rows*) et de plusieurs colonnes (*n columns*). Ici *n* étant bien sûr supérieur à 1. Nous avions noté que pour connaître le nombre d'éléments dans un vecteur on utilisait la fonction `length`.

```
length(prenom_nom)
```

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

La même chose marche-t-elle pour le vecteur?

```
length(prenom_nom_vmatrix)
```

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

En l'occurence, non! `length` ne rend pas compte de la bidimensionnalité. Il y a une autre fonction pour ça: `dim`.

```
dim(prenom_nom_vmatrix)
```

```
## [1] 5 2
```

La bidimensionnalité se lit aussi dans le nom des rangées.

```
dimnames(prenom_nom_vmatrix)
```

```
## [[1]]
## NULL
##
## [[2]]
## [1] "prenom" "nom"
```

Comme nous avons vu plus haut, l'on peut nommer les rangées depuis la création de la matrice. Reprenons la création de `prenom_nom_vmatrix` en nommant toutes les rangées, aussi bien horizontales que verticales.

```
prenom_nom_vmatrix <- matrix(c("Modibo", "Keïta",
                               "Moussa", "Traoré",
                               "Alpha Oumar", "Konaré",
                               "Amadou Toumani", "Touré",
                               "Ibrahim Boubacar", "Keïta"),
                             byrow = TRUE,
                             ncol = 2,
                             dimnames = list(c("1er", "2ème", "3ème", "4ème", "5ème"),
                                             c("prenom", "nom"))
                             )
```

Imprimons la matrice.

```
print(prenom_nom_vmatrix)
```

```
##      prenom             nom
## 1er  "Modibo"           "Keïta"
## 2ème "Moussa"           "Traoré"
## 3ème "Alpha Oumar"      "Konaré"
## 4ème "Amadou Toumani"   "Touré"
## 5ème "Ibrahim Boubacar" "Keïta"
```

Examinons la matrice à travers différentes fonctions que nous avons vues en haut:

- la dimension, c'est-à-dire le nombre de lignes et le nombre de colonnes;

```
dim(prenom_nom_vmatrix)
```

```
## [1] 5 2
```

- les noms des lignes et des colonnes;

```
dimnames(prenom_nom_vmatrix)
```

```
## [[1]]
## [1] "1er"  "2ème" "3ème" "4ème" "5ème"
##
## [[2]]
## [1] "prenom" "nom"
```

- le nombre de lignes uniquement;

```
nrow(prenom_nom_vmatrix)
```

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

- le nom des lignes uniquement;

```
rownames(prenom_nom_vmatrix)
```

```
## [1] "1er"  "2ème" "3ème" "4ème" "5ème"
```

- le nombre de colonnes uniquement;

```
ncol(prenom_nom_vmatrix)
```

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

- le nom des colonnes uniquements.

```
colnames(prenom_nom_vmatrix)
```

```
## [1] "prenom" "nom"
```

3.4.3 Opérations sur matrices

3.4.3.1 Une autre matrice

A l'instar des vecteurs, les matrices se prêtent elles aussi à une variété d'opérations. Explorons-en quelques unes.

Reprenons notre exemple sur les présidents maliens et considérons les années d'évènements majeurs: naissance, arrivée au pouvoir et départ du pouvoir.

```
annee_evenement_matrix <- matrix(c(1915, 1936, 1946, 1948, 1945,
                                   1960, 1968, 1992, 2002, 2013,
```

```
                                  1968, 1991, 2002, 2012, NA),
                                byrow = TRUE,
                                ncol = 5,
                                dimnames = list(c("Naissance", "Arrivée", "Départ"),
                                                c("M. Keïta", "M. Traoré", "A.O. Konaré", "A.T. Touré"
annee_evenement_matrix
```

```
##           M. Keïta M. Traoré A.O. Konaré A.T. Touré I.B. Keïta
## Naissance     1915      1936        1946       1948       1945
## Arrivée       1960      1968        1992       2002       2013
## Départ        1968      1991        2002       2012         NA
```

3.4.3.2 Questions logiques

Maintenant que nous avons notre matrice, amusons-nous avec. Prenons un grand-père née vers 1949 (oui, il fait partie des *né vers*), marié à l'âge de 22 ans, père 1 an plus tard, grand-père 18 ans plus tard et décédé à l'âge de 61 ans. Quels sont les évènements qui se sont passés de son vivant?

```
ce_que_grandpa_a_vu <- annee_evenement_matrix > 1949 & annee_evenement_matrix < (1949 + 61)
ce_que_grandpa_a_vu
```

```
##           M. Keïta M. Traoré A.O. Konaré A.T. Touré I.B. Keïta
## Naissance    FALSE     FALSE       FALSE      FALSE      FALSE
## Arrivée       TRUE      TRUE        TRUE       TRUE      FALSE
## Départ        TRUE      TRUE        TRUE      FALSE         NA
```

Apparemment, il en a vu beaucoup, mais tous les présidents le dépassent en âge. On vient d'introduire ici la notion d'addition dans les critères (dans le prochain chapitre, la question sera plus développée).

3.4.3.3 Extraction par position

Comme pour les vecteurs, des éléments peuvent être explicitement sélectionnés à l'intérieur des matrices. Comme pour ceux-ci également, le signe `[]` peut être utilisé. Revenons à notre matrice `annee_evenement_matrix`.

```
annee_evenement_matrix
```

```
##           M. Keïta M. Traoré A.O. Konaré A.T. Touré I.B. Keïta
## Naissance     1915      1936        1946       1948       1945
## Arrivée       1960      1968        1992       2002       2013
## Départ        1968      1991        2002       2012         NA
```

Supposons que l'on veuille connaître l'élément qui est dans la cellule de la 3ème ligne et le 2ème colonne.

```
annee_evenement_matrix[3, 2]
```

```
## [1] 1991
```

Ou la 3ème ligne toute entière.

```
annee_evenement_matrix[3, ]
```

```
##    M. Keïta   M. Traoré A.O. Konaré  A.T. Touré  I.B. Keïta
##        1968        1991        2002        2012          NA
```

Ou la 2ème colonne toute entière.

```
annee_evenement_matrix[ , 2]
```

```
## Naissance   Arrivée    Départ
```

```
##      1936      1968      1991
```

Avec les matrices, l'on spécifie deux éléments à l'intérieur des crochets. Le premier désigne la ligne à sélectionner et le deuxième la colonne.

3.4.3.4 Extraction par nom

Si les rangées sont nommés, alors il est aussi possible de passer par ces noms pour les sélectionner. Vous vous rappelez `rownames` ou `colnames`? Si la réponse est non, je saurai que vous n'avez pas tout suivi! Passons par ces fonctions pour sélectionner des lignes et colonnes d'intérêt dans notre matrice.

```
rownames(annee_evenement_matrix)
```

```
## [1] "Naissance" "Arrivée"   "Départ"
```

```
colnames(annee_evenement_matrix)
```

```
## [1] "M. Keïta"    "M. Traoré"   "A.O. Konaré" "A.T. Touré"  "I.B. Keïta"
```

Séléctionnons la ligne relative aux années de naissance.

```
annee_evenement_matrix[rownames(annee_evenement_matrix) == "Naissance", ]
```

```
##    M. Keïta   M. Traoré A.O. Konaré  A.T. Touré  I.B. Keïta
##        1915        1936        1946        1948        1945
```

Et cherchons les éléments concernant le président Modibo Keïta.

```
annee_evenement_matrix[, colnames(annee_evenement_matrix) == "M. Keïta"]
```

```
## Naissance   Arrivée    Départ
##      1915      1960      1968
```

3.4.3.5 Consolidation

Il arrive souvent que l'on souhaite consolider une matrice en y ajoutant de nouvelles informations. Ces nouvelles informations peuvent même être dérivées d'éléments déjà existants à l'intérieur de la matrice. Considérons ici que nous voulions ajouter à notre matrice l'âge à l'arrivée au pouvoir et l'âge au départ du pouvoir. Nous passons tout simplement par les techniques que nous avons déjà vues pour générer ces nouveaux éléments.

```
# Un vecteur pour l'âge d'arrivée au pouvoir
age_arrivee_pouvoir <-
  annee_evenement_matrix[rownames(annee_evenement_matrix) == "Arrivée", ] -
  annee_evenement_matrix[rownames(annee_evenement_matrix) == "Naissance", ]
# Un vecteur pour l'age de départ du pouvoir
age_depart_pouvoir <-
  annee_evenement_matrix[rownames(annee_evenement_matrix) == "Départ", ] -
  annee_evenement_matrix[rownames(annee_evenement_matrix) == "Naissance", ]
# Un vecteur pour la durée au poivoir
duree_au_pouvoir <-
  annee_evenement_matrix[rownames(annee_evenement_matrix) == "Départ", ] -
  annee_evenement_matrix[rownames(annee_evenement_matrix) == "Arrivée", ]
# Ajoutons maintenant ces trois nouveaux vecteurs à notre matrice
annee_evenement_matrix_cons <- rbind(annee_evenement_matrix,
                                     "Âge d'arrivée au pouvoir" =  age_arrivee_pouvoir,
                                     "Âge de départ du pouvoir" = age_depart_pouvoir,
                                     "Durée au pouvoir" = duree_au_pouvoir)
# Voyons la matrice
annee_evenement_matrix_cons
```

```
##                           M. Keïta M. Traoré A.O. Konaré A.T. Touré
## Naissance                     1915      1936        1946       1948
## Arrivée                       1960      1968        1992       2002
## Départ                        1968      1991        2002       2012
## Âge d'arrivée au pouvoir        45        32          46         54
## Âge de départ du pouvoir        53        55          56         64
## Durée au pouvoir                 8        23          10         10
##                           I.B. Keïta
## Naissance                       1945
## Arrivée                         2013
## Départ                            NA
## Âge d'arrivée au pouvoir          68
## Âge de départ du pouvoir          NA
## Durée au pouvoir                  NA
```

```
# Nous sommes passés par la fonction "rbind()". Sachez qu'il y a plusieurs solutions!
# Remarquez-vous "NA" dans une nouvelle cellule? Vous rappelez-vous pourquoi?
```

3.4.3.6 Calculs

Comme pour les vecteurs, des calculs sont possibles sur les matrices. Pour ce faire, limitons-nous à deux informations de la matrice: les âges et les durées. Calculons les moyennes. D'abord l'âge moyen d'arrivée au pouvoir.

```
mean(annee_evenement_matrix_cons["Âge d'arrivée au pouvoir", ])
```

```
## [1] 49
```

Ensuite, l'age moyen de départ du pouvoir.

```
mean(annee_evenement_matrix_cons["Âge de départ du pouvoir", ])
```

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

Nous voyons que R nous donne une valeur `NA`. Ne sachant quoi faire en présence de cette valeur dans la matrice sélectionnée, R s'est résigné à ne rien faire. D'où la sortie de `NA` comme résultat. Heureusement, les fonctions comportent aussi des moyens pour contourner ce problème, l'exclusion des valeurs `NA`.

```
mean(annee_evenement_matrix_cons["Âge de départ du pouvoir", ], na.rm = TRUE)
```

```
## [1] 57
```

La même technique nous permet de contourner la présence de `NA` dans le vecteur `Durée au pouvoir`.

```
mean(annee_evenement_matrix_cons["Durée au pouvoir", ], na.rm = TRUE)
```

```
## [1] 12.75
```

Il existe nombreuses fonctions qui permettent de faire des calculs sur les matrices: `colSums`, `rowSums`, `colMeans` et `rowMeans`.

3.5 *Data frames*

3.5.1 Le *data frame*, au-délà de la matrice

Jusque là, nous avons travaillé avec des éléments de même nature. Et pourtant le *data scientist* ne peut pleinement mener ses investigations avec une telle contrainte. Il a besoin d'explorer en même temps des informations de diverses natures. D'où le data frame. Qu'est-ce que c'est au juste? Un format d'organisation de données en forme rectangulaire, tout comme la matrice. Toutefois, contrairement à la matrice, elle respecte

la nature des données qu'elle contient. Explorons l'idée. Rassemblons verticalement les différents vecteurs que nous avons créés. Re-créons d'abord les vecteurs.

```
nom <- c("Keïta", "Traoré", "Konaré", "Touré", "Keïta")
prenom <- c("Modibo", "Moussa", "Alpha Oumar", "Amadou Toumani", "Ibrahim Boubacar")
date_naissance <- as.Date(c("1915-06-04", "1936-09-25", "1946-02-02", "1948-11-04", "1945-01-29"))
region_naissance <- as.factor(c("Bamako", "Kayes", "Kayes", "Mopti", "Koutiala"))
annee_arrivee_pouvoir <- c(1960, 1968, 1992, 2002, 2013)
duree_au_pouvoir <- c(8, 23, 10, 10, NA)
parcours_militaire <- c(FALSE, TRUE, FALSE, TRUE, FALSE)
```

Puis, rassemblons-les.

```
presidents_df <- cbind(nom,
                       prenom,
                       date_naissance,
                       region_naissance,
                       parcours_militaire,
                       annee_arrivee_pouvoir,
                       duree_au_pouvoir)
```

Qu'est-ce que ça donne?

```
presidents_df
```

```
##      nom      prenom             date_naissance region_naissance
## [1,] "Keïta"  "Modibo"           "-19935"       "1"
## [2,] "Traoré" "Moussa"           "-12151"       "2"
## [3,] "Konaré" "Alpha Oumar"      "-8734"        "2"
## [4,] "Touré"  "Amadou Toumani"   "-7728"        "4"
## [5,] "Keïta"  "Ibrahim Boubacar" "-9103"        "3"
##      parcours_militaire annee_arrivee_pouvoir duree_au_pouvoir
## [1,] "FALSE"            "1960"                "8"
## [2,] "TRUE"             "1968"                "23"
## [3,] "FALSE"            "1992"                "10"
## [4,] "TRUE"             "2002"                "10"
## [5,] "FALSE"            "2013"                NA
```

Nous remarquons que certaines informations ont été dénaturées. Certaines données ont été coercées à se transformer en autre chose. Regardons la classe de l'objet `presidents_df`.

```
class(presidents_df)
```

```
## [1] "matrix"
```

```
typeof(presidents_df)
```

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

Les vecteurs ont été rassemblés en matrice (`class`). Les éléments ont toutefois été coercés en charactères (`typeof`). Ceci signifie que nous ne pouvons pas manipuler les éléments qui sont des entières ou des dates. C'est en celà que le *data frame* révèle son premier avantage: l'unité dans la diversité.

Reprenons l'opération. Cette fois-ci, toutefois, indiquons qu'il s'agit d'un *data frame* avec la fonction `data.frame`.

```
presidents_df <- data.frame(nom,
                            prenom,
                            date_naissance,
                            region_naissance,
```

```
                          parcours_militaire,
                          annee_arrivee_pouvoir,
                          duree_au_pouvoir,
                          stringsAsFactors = FALSE)
```

Regardons à nouveau

```
presidents_df
```

```
##      nom           prenom date_naissance region_naissance
## 1  Keïta           Modibo     1915-06-04           Bamako
## 2 Traoré           Moussa     1936-09-25            Kayes
## 3 Konaré      Alpha Oumar     1946-02-02            Kayes
## 4  Touré   Amadou Toumani     1948-11-04            Mopti
## 5  Keïta Ibrahim Boubacar     1945-01-29         Koutiala
##   parcours_militaire annee_arrivee_pouvoir duree_au_pouvoir
## 1              FALSE                  1960                8
## 2               TRUE                  1968               23
## 3              FALSE                  1992               10
## 4               TRUE                  2002               10
## 5              FALSE                  2013               NA
```

Qu'en est-il de la classe et du type?

```
class(presidents_df)
```

```
## [1] "data.frame"
```

Maintenant que nous savons à quoi ressemble un *data frame*, essayons de le définir. Un data frame est une forme d'organisation de données en format rectangulaire où les lignes sont des observations et les colonnes des attributs de ceux-ci. Ici par exemple, nous organisons diverses informations sur les individus qui ont assumé le poste de Président de la République du Mali. Chaque ligne sera dédiée à un président et rassemblera tous les informations sur lui (attributs). Chaque colonne sera dédiée à un seul attribut et couvrira tous les présidents (observations).

A l'instar de la matrice, le *data frame* se prête lui aussi aux fonctions qui renseignent sur ses dimensions.

```
dim(presidents_df)
```

```
## [1] 5 7
```

```
dimnames(presidents_df)
```

```
## [[1]]
## [1] "1" "2" "3" "4" "5"
##
## [[2]]
## [1] "nom"                 "prenom"              "date_naissance"
## [4] "region_naissance"    "parcours_militaire"  "annee_arrivee_pouvoir"
## [7] "duree_au_pouvoir"
```

```
colnames(presidents_df)
```

```
## [1] "nom"                 "prenom"              "date_naissance"
## [4] "region_naissance"    "parcours_militaire"  "annee_arrivee_pouvoir"
## [7] "duree_au_pouvoir"
```

```
rownames(presidents_df)
```

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

Quand les lignes n'ont pas de nom, R affiche tout simplement les index. Généralement, on s'intéresse à deux éléments avec les *data frame*: les dimensions et les noms des colonnes (variables ou attributs). En ce qui concerne les lignes, il est rare qu'on les nomme vu que les observations peuvent être de nombre très élevé (milliers voire millions). De ce fait, l'on peut s'en tenir à deux fonctions.

```
dim(presidents_df)
```

```
## [1] 5 7
```

```
names(presidents_df)
```

```
## [1] "nom"                   "prenom"                "date_naissance"
## [4] "region_naissance"      "parcours_militaire"    "annee_arrivee_pouvoir"
## [7] "duree_au_pouvoir"
```

La particularité du *data frame* se lit à travers la fonction `str` qui montre sa structure. Cette fonction montre le classe des colonnes qui le constituent.

```
str(presidents_df)
```

```
## 'data.frame':    5 obs. of  7 variables:
##  $ nom                  : chr  "Keïta" "Traoré" "Konaré" "Touré" ...
##  $ prenom               : chr  "Modibo" "Moussa" "Alpha Oumar" "Amadou Toumani" ...
##  $ date_naissance       : Date, format: "1915-06-04" "1936-09-25" ...
##  $ region_naissance     : Factor w/ 4 levels "Bamako","Kayes",..: 1 2 2 4 3
##  $ parcours_militaire   : logi  FALSE TRUE FALSE TRUE FALSE
##  $ annee_arrivee_pouvoir: num  1960 1968 1992 2002 2013
##  $ duree_au_pouvoir     : num  8 23 10 10 NA
```

On a là une synthèse montrant nombre d'observations et nombre de variables comme avec la fonction `dim`; On voit aussi que pour chaque variable, on a le nom, la classe et quelques observations.

3.5.2 Opérations sur *data frame*

3.5.2.1 Sélection de cellules

En matière de sélection, le *data frame* hérite beaucoup de la matrice. Les principes demeurent les mêmes. Si l'on veut la ligne 2 de la colonne 4, on fait:

```
presidents_df[2, 4]
```

```
## [1] Kayes
## Levels: Bamako Kayes Koutiala Mopti
```

Si l'on veut la ligne 5 (un président, une observation):

```
presidents_df[2, ]
```

```
##      nom prenom date_naissance region_naissance parcours_militaire
## 2 Traoré Moussa     1936-09-25            Kayes               TRUE
##   annee_arrivee_pouvoir duree_au_pouvoir
## 2                  1968               23
```

Ou encore, la colonne 4 (une variable, un attribut)

```
presidents_df[, 4]
```

```
## [1] Bamako   Kayes    Kayes    Mopti    Koutiala
## Levels: Bamako Kayes Koutiala Mopti
```

3.5.2.2 Sélection de variables

Les techniques de sélections de colonnes sur la matrice sont valables pour le *data frame* aussi. Regardons, à titre d'exemple, la variable `date_naissance`. On peut y acceder à partir de sa position dans l'ordre des variables.

```
presidents_df[, c(3)]
```

```
## [1] "1915-06-04" "1936-09-25" "1946-02-02" "1948-11-04" "1945-01-29"
```

Ou la désigner par son nom directement.

```
presidents_df[, "date_naissance"]
```

```
## [1] "1915-06-04" "1936-09-25" "1946-02-02" "1948-11-04" "1945-01-29"
```

Le *data frame* offre en plus une alternative: les variables y sont accessibles avec le signe `$`.

```
presidents_df$date_naissance
```

```
## [1] "1915-06-04" "1936-09-25" "1946-02-02" "1948-11-04" "1945-01-29"
```

3.5.2.3 Création de variables

Comme avec les matrices, souvent, l'analyste de données peut souhaiter ajouter une nouvelle variable à son *data frame*. Procédons comme avec les matrices à la génération de deux nouvelles variables: l'âge d'arrivée au pouvoir et l'âge de départ du pouvoir. Pour commencer, générons l'année de naissance.

```
presidents_df$annee_naissance <- as.numeric(format(presidents_df$date_naissance,'%Y'))
```

Ensuite on génère l'âge d'arrivée au pouvoir.

```
presidents_df$age_arrivee_pouvoir <- presidents_df$annee_arrivee_pouvoir - presidents_df$annee_naissanc
```

Ensuite l'âge de départ du pouvoir.

```
presidents_df$age_depart_pouvoir <- presidents_df$age_arrivee_pouvoir + presidents_df$duree_au_pouvoir
```

Regardons notre nouveau data frame.

```
str(presidents_df)
```

```
## 'data.frame':    5 obs. of  10 variables:
##  $ nom                  : chr  "Keïta" "Traoré" "Konaré" "Touré" ...
##  $ prenom               : chr  "Modibo" "Moussa" "Alpha Oumar" "Amadou Toumani" ...
##  $ date_naissance       : Date, format: "1915-06-04" "1936-09-25" ...
##  $ region_naissance     : Factor w/ 4 levels "Bamako","Kayes",..: 1 2 2 4 3
##  $ parcours_militaire   : logi  FALSE TRUE FALSE TRUE FALSE
##  $ annee_arrivee_pouvoir: num  1960 1968 1992 2002 2013
##  $ duree_au_pouvoir     : num  8 23 10 10 NA
##  $ annee_naissance      : num  1915 1936 1946 1948 1945
##  $ age_arrivee_pouvoir  : num  45 32 46 54 68
##  $ age_depart_pouvoir   : num  53 55 56 64 NA
```

A travers cette création, on voit comment on peut mener des opérations entre des colonnes d'un *data frame*.

3.5.2.4 Suppression de variables

Dans notre exemple, nous avons créé l'année de naissance comme étape transitoire vers une autre variable. Sachant que nous avons la même information dans la date de naissance, l'on peut éviter la redondance, donc la supprimer. Comment s'y prend-on dans R?

```
presidents_df$annee_naissance <- NULL
```

Vérifions si cette colonne est partie.

```
str(presidents_df)
```

```
## 'data.frame':    5 obs. of  9 variables:
##  $ nom                  : chr  "Keïta" "Traoré" "Konaré" "Touré" ...
##  $ prenom               : chr  "Modibo" "Moussa" "Alpha Oumar" "Amadou Toumani" ...
##  $ date_naissance       : Date, format: "1915-06-04" "1936-09-25" ...
##  $ region_naissance     : Factor w/ 4 levels "Bamako","Kayes",..: 1 2 2 4 3
##  $ parcours_militaire   : logi  FALSE TRUE FALSE TRUE FALSE
##  $ annee_arrivee_pouvoir: num  1960 1968 1992 2002 2013
##  $ duree_au_pouvoir     : num  8 23 10 10 NA
##  $ age_arrivee_pouvoir  : num  45 32 46 54 68
##  $ age_depart_pouvoir   : num  53 55 56 64 NA
```

Mission accomplie!

3.5.2.5 Sélection d'observations

Nous avons vu que comme la matrice, les éléments du *data frame* sont accessibles grâce aux numéros de lignes. Ici, nous allons voir qu'il est aussi possible de passer par des critères spécifiques aux variables pour sélectionner des observations. Cherchons seulement les noms et prénoms des présidents nés dans la région de "Kayes".

```
presidents_df[presidents_df$region_naissance == "Kayes", c("nom", "prenom")]
```

```
##      nom      prenom
## 2 Traoré      Moussa
## 3 Konaré Alpha Oumar
```

Il est possible d'aboutir au même résultat avec une fonction intégrée à R: `subset`. Cette fonction offre la commodité de sélectionner à la fois des observations à partir de critères et des variables.

```
subset(x = presidents_df, subset = region_naissance == "Kayes", select = c(nom, prenom))
```

```
##      nom      prenom
## 2 Traoré      Moussa
## 3 Konaré Alpha Oumar
```

Un autre exemple! Voyons le nom, le prénom et la date de naissance pour les présidents arrivés au pouvoir avant l'âge de 50 ans.

```
subset(x = presidents_df, subset = age_arrivee_pouvoir < 50, select = c(nom, prenom, date_naissance))
```

```
##      nom      prenom date_naissance
## 1  Keïta      Modibo     1915-06-04
## 2 Traoré      Moussa     1936-09-25
## 3 Konaré Alpha Oumar     1946-02-02
```

Un autre exemple! Voyons le nom, le prénom et la durée au pouvoir pour les présidents ayant fait moins de 10 ans au poste.

```
subset(x = presidents_df, subset = duree_au_pouvoir < 10, select = c(nom, prenom, duree_au_pouvoir))
```

```
##     nom prenom duree_au_pouvoir
## 1 Keïta Modibo                8
```

Vous voyez? Avec R, tous les chemins mènent... à Roundé. (Rome est trop loin pour moi! Même s'il comment par R).

3.5.2.6 Ordonner les observations

On peut souvent souhaiter ordonner son *data frame* selon une variable donnée. Rearrangeons nos données selon l'année de naissance des présidents

```
ordre_age <- order(presidents_df$date_naissance)
ordre_age
```

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

Nous pouvons voir que si les premier et second présidents se succèdent en ainesse, le troisième lui il est devancé par le cinquième. Pour mieux comprendre, utilons cet ordre pour afficher le *data frame*.

```
presidents_df[ordre_age, ]
```

```
##      nom           prenom date_naissance region_naissance
## 1  Keïta           Modibo     1915-06-04           Bamako
## 2 Traoré           Moussa     1936-09-25            Kayes
## 5  Keïta Ibrahim Boubacar     1945-01-29         Koutiala
## 3 Konaré      Alpha Oumar     1946-02-02            Kayes
## 4  Touré   Amadou Toumani     1948-11-04            Mopti
##   parcours_militaire annee_arrivee_pouvoir duree_au_pouvoir
## 1              FALSE                  1960                8
## 2               TRUE                  1968               23
## 5              FALSE                  2013               NA
## 3              FALSE                  1992               10
## 4               TRUE                  2002               10
##   age_arrivee_pouvoir age_depart_pouvoir
## 1                  45                 53
## 2                  32                 55
## 5                  68                 NA
## 3                  46                 56
## 4                  54                 64
```

Pour plus de commodité, regardons seulement les nom, prénom et date de naissance.

```
presidents_df[ordre_age, c("nom", "prenom", "date_naissance")]
```

```
##      nom           prenom date_naissance
## 1  Keïta           Modibo     1915-06-04
## 2 Traoré           Moussa     1936-09-25
## 5  Keïta Ibrahim Boubacar     1945-01-29
## 3 Konaré      Alpha Oumar     1946-02-02
## 4  Touré   Amadou Toumani     1948-11-04
```

3.5.3 Le meilleur reste à venir

Le *data frame* est la pièce maîtresse de l'analyse dans R, comme dans beaucoup d'autres langages. D'ailleurs, d'autres langages ont développé des concepts similaires. En prenant Python par exemple, on trouve la notion de *DataFrame*, une adaption du concept de *data frame* tel que défini dans R. Pour dire combien l'idée englobée dans le *data frame* est puissante. D'où son rôle capital dans le reste de ce cours et de l'ouvrage.

C'est avec le *data frame* que nous:

- procéderons à des manipulations de données: du nettoyage à la transformation ;

- explorerons des données par la visualisation ;
- introduirons l'application de modèles à des données.

3.6 Listes

3.6.1 Oublier l'ordre et la structure

La liste (*list* en anglais et dans R) apporte elle aussi sa particularité. Elle permet de créer un espace pour les données non structurées dans R. Créons de nouveaux éléments. Commençons par les pays voisins du Mali: un vecteur en caractères.

```
voisins_vec_char <- c( "Algérie", "Burkina-Faso", "Côte d'Ivoire", "Guinée", "Mauritanie", "Niger", "Sé
```

Ajoutons des données sur la population à partir des données de recensement de 1976, 1987, 1998 et 2009. C'est une matrice d'entiers.

```
population_matrix_int <- matrix(data = c(3123733, 3269185, 6392918,
                                         3760711, 3935638, 7696349,
                                         4856023, 4954889, 9810912,
                                         7204990, 7323672, 14528662),
                                byrow = TRUE,
                                nrow = 4,
                                dimnames = list(c(1976, 1987, 1998, 2009),
                                                c("Hommes", "Femmes", "Total")))
```

Ajoutons un dernier élément: lesquels de nos présidents sont encore vivants? Mettons ça sous forme boléén.

```
presidents_en_vie_vec_logi <- c(FALSE, TRUE, TRUE, TRUE, TRUE)
```

Nous avons là un beau monde. Rassemblons tout ça dans une liste!

```
mali_list <- list(presidents = presidents_df,
                  voisins = voisins_vec_char,
                  population = population_matrix_int,
                  presidents_en_vie = presidents_en_vie_vec_logi)
```

De par leurs différences en nature, forme et taille, rien ne prédispose ses objets à être contenus dans le même objet! Et pourtant ça tient dans notre liste. Explorons celle-ci!

3.6.2 Un contenant de contenants

Commençons par la structure de la liste. Que voit-on?

```
str(mali_list)
```

```
## List of 4
##  $ presidents       :'data.frame':   5 obs. of  9 variables:
##   ..$ nom                 : chr [1:5] "Keïta" "Traoré" "Konaré" "Touré" ...
##   ..$ prenom              : chr [1:5] "Modibo" "Moussa" "Alpha Oumar" "Amadou Toumani" ...
##   ..$ date_naissance      : Date[1:5], format: "1915-06-04" ...
##   ..$ region_naissance    : Factor w/ 4 levels "Bamako","Kayes",..: 1 2 2 4 3
##   ..$ parcours_militaire  : logi [1:5] FALSE TRUE FALSE TRUE FALSE
##   ..$ annee_arrivee_pouvoir: num [1:5] 1960 1968 1992 2002 2013
##   ..$ duree_au_pouvoir    : num [1:5] 8 23 10 10 NA
##   ..$ age_arrivee_pouvoir : num [1:5] 45 32 46 54 68
##   ..$ age_depart_pouvoir  : num [1:5] 53 55 56 64 NA
##  $ voisins          : chr [1:7] "Algérie" "Burkina-Faso" "Côte d'Ivoire" "Guinée" ...
```

```
##  $ population       : num [1:4, 1:3] 3123733 3760711 4856023 7204990 3269185 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:4] "1976" "1987" "1998" "2009"
##   .. ..$ : chr [1:3] "Hommes" "Femmes" "Total"
##  $ presidents_en_vie: logi [1:5] FALSE TRUE TRUE TRUE TRUE
```

Qu'en est-il des noms

```
names(mali_list)
```

```
## [1] "presidents"        "voisins"           "population"
## [4] "presidents_en_vie"
```

Les noms assignés aux objets sont bien reconduits. Voyons voir si à l'instar des matrices et des data frames, ces noms peuvent être utilisés pour accéder aux éléments qui y sont stockés. Prenons le vecteur sur les pays voisins.

```
mali_list[["voisins"]]
```

```
## [1] "Algérie"       "Burkina-Faso"  "Côte d'Ivoire" "Guinée"
## [5] "Mauritanie"    "Niger"         "Sénégal"
```

Le même résultat doit être possible par l'ordre de l'objet dans la liste, le 2ème.

```
mali_list[[2]]
```

```
## [1] "Algérie"       "Burkina-Faso"  "Côte d'Ivoire" "Guinée"
## [5] "Mauritanie"    "Niger"         "Sénégal"
```

Peut-on utiliser le signe $ comme avec les *data frame*.

```
mali_list$voisins
```

```
## [1] "Algérie"       "Burkina-Faso"  "Côte d'Ivoire" "Guinée"
## [5] "Mauritanie"    "Niger"         "Sénégal"
```

Donc, on a l'embarra du choix.

Maitenant qu'on peut accéder aux objets à l'intérieur d'une liste, qu'en est-il des éléments stockés à l'intérieur de cette liste elle-même. Cherchons le 2ème élément du vecteur des pays voisins.

```
mali_list[["voisins"]][2]
```

```
## [1] "Burkina-Faso"
```

On peut y accéder avec d'autres voies dont les suivantes.

```
mali_list$voisins[2]
```

```
## [1] "Burkina-Faso"
```

```
mali_list[[2]][2]
```

```
## [1] "Burkina-Faso"
```

Un autre exemple: la 3ème colonne de la matrice sur la population

```
mali_list[["population"]][, 3]
```

```
##     1976     1987     1998     2009
##  6392918  7696349  9810912 14528662
```

Pour le même résultat, le code suivant aussi marche.

```
mali_list[["population"]][, "Total"]
```

```
##     1976     1987     1998     2009
##  6392918  7696349  9810912 14528662
```

3.7 Conclusion

3.7.1 Et ce n'est que le début

Avec une introduction à ces objets, on pose les bases de l'analyse de données dans R. Bien que, pour des raisons pédagogiques, chaque objet ait été présenté par rapport aux limites du précédent, ils demeurent tous utiles, chacun avec ses avantages (compétitifs). Il revient au *data scientist* de connaître quand, où et comment faire intervenir un au lieu des autres. Contribuer à vous outiller pour faire ces choix - parmi tant d'autres - est l'un des objectifs de ce cours / cet ouvrage.

Chapter 4

S'exprimer dans R

4.1 Introduction

4.1.1 Objectif

Dans le chapitre précédent, il a été question d'objets dans R. Certains types ont été présentés. Il a surtout été fait état de la différence qui les sépare, de ce en quoi ils se démarquent les uns des autres. Ici, nous allons continuer en explorant l'expression dans R.

Les objets permettent de stocker des données. Celles-ci ne deviennent vivantes et parlantes qu'à travers le dialogue que le *data scientist* entretient avec elles. Et en quels termes ce dialogue se pose-t-il? Là est le début de notre démarche ici.

Nous allons:

- revenir sur les questions logiques;
- introduire déclarations conditionnelles;
- introduire la notion de boucle et de fonction.

4.1.2 Outils

Que nous faut-il?

- R (évidemment);
- RStudio (de préférence);
- Les données utilisées dans le cadre du présent chapitre.

4.1.3 Données

Dans le présent chapitre, nous allons utiliser des données tirées des Recensements Généraux de la Population et de l'Habitat au Mali en 1976, 1987, 1998 et 2009. Des rapports sont disponibles cette adresse.

Quant aux données extraites et formatées pour le présent cours, elles sont disponibles à cette adresse.

Nous

Balayons du regard les objets qui meublent notre environnement.

```
ls()
```

```
## [1] "pop_groupage_list"
```

Regardons la structure de cet objet.

```
str(pop_groupage_list)
```

```
## List of 4
##  $ 1976:'data.frame':    18 obs. of  5 variables:
##   ..$ annee   : num [1:18] 1976 1976 1976 1976 1976 ...
##   ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ femme   : num [1:18] 589394 482851 321959 333508 265842 ...
##   ..$ homme   : num [1:18] 587015 492272 342807 308607 218391 ...
##   ..$ total   : num [1:18] 1176409 975123 664766 642115 484233 ...
##  $ 1987:'data.frame':    18 obs. of  5 variables:
##   ..$ annee   : num [1:18] 1987 1987 1987 1987 1987 ...
##   ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ femme   : num [1:18] 713507 611562 414302 379522 315753 ...
##   ..$ homme   : num [1:18] 719804 633206 452166 348200 260215 ...
##   ..$ total   : num [1:18] 1433311 1244768 866468 727722 575968 ...
##  $ 1998:'data.frame':    18 obs. of  5 variables:
##   ..$ annee   : num [1:18] 1998 1998 1998 1998 1998 ...
##   ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ femme   : num [1:18] 824505 797057 589603 529270 409584 ...
##   ..$ homme   : num [1:18] 839795 830211 637495 492480 364333 ...
##   ..$ total   : num [1:18] 1664300 1627268 1227098 1021750 773917 ...
##  $ 2009:'data.frame':    18 obs. of  5 variables:
##   ..$ annee   : num [1:18] 2009 2009 2009 2009 2009 ...
##   ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ femme   : num [1:18] 1321275 1178850 882725 799081 624565 ...
##   ..$ homme   : num [1:18] 1353418 1225145 935796 745757 538927 ...
##   ..$ total   : num [1:18] 2674693 2403995 1818521 1544838 1163492 ...
```

Il s'agit d'une liste. Les données portent sur la population par groupe d'âge.

4.2 Les déclarations

Nous allons présenter ici la notion de déclaration et l'illustrer à partir de nos données. Qu'est-ce qu'une déclaration? Tout simplement une affirmation que l'on formule et que l'on soumet à la machine... Pour être plus exact, nous soumettons la déclaration aux données et regardons leur réaction!

4.2.1 Formulations simples

Affirmons qu'au Mali, pour les groupes d'âge identifiés, il y a plus de femmes que d'hommes. Est-ce vrai ou faux? Qu'en disent nos données? Pour faire simple, prenons le recensement le plus récent, celui de 2009, pour vérifier la véracité de notre déclaration.

Pour commencer, tirons de la liste les données relative à l'année d'intérêt.

```
pop_groupage_2009 <- pop_groupage_list[["2009"]]
```

Maintenant, regardons la structure de ce *data frame*.

```
str(pop_groupage_2009)
```

```
## 'data.frame':    18 obs. of  5 variables:
##  $ annee   : num  2009 2009 2009 2009 2009 ...
##  $ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ femme   : num  1321275 1178850 882725 799081 624565 ...
##  $ homme   : num  1353418 1225145 935796 745757 538927 ...
```

```
##  $ total   : num  2674693 2403995 1818521 1544838 1163492 ...
```

Regardons la tête, les 3 premières observations par exemple.

```
head(x = pop_groupage_2009, n = 3)
```

```
##    annee groupage   femme   homme   total
## 55  2009      0-4 1321275 1353418 2674693
## 56  2009      5-9 1178850 1225145 2403995
## 57  2009    10-14  882725  935796 1818521
```

Regardons la queue, les 3 dernières observations par exemple.

```
tail(x = pop_groupage_2009, n = 3)
```

```
##    annee groupage femme homme total
## 70  2009    75-79 36949 41667 78616
## 71  2009      80+ 44504 42779 87283
## 72  2009       ND     0     0     0
```

Nous voyons qu'il y a une colonne pour les hommes `homme` et une autre pour les femmes, `femme`. Tirons du *data frame* les vecteurs relatifs à ces deux groupes.

```
pop_femme_2009 <- pop_groupage_2009$femme
pop_homme_2009 <- pop_groupage_2009$homme
```

Maintenant posons la condition suivante: `pop_femme_2009 > pop_homme_2009`.

```
pop_femme_2009 > pop_homme_2009
```

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

Pour une meilleur lisibilité, insérons ce résultat dans le data frame

```
pop_groupage_2009$femme_sup_homme <- pop_groupage_2009$femme > pop_groupage_2009$homme
```

En assignant le résultat de l'opération à une nouvelle variable du *data frame*, R crée lui-même une variable boléenne (`TRUE`/`FALSE`). Regardons les groupes d'âge qui répondent au critère posé.

```
pop_groupage_2009[pop_groupage_2009$femme_sup_homme, ]
```

```
##    annee groupage  femme  homme   total femme_sup_homme
## 58  2009    15-19 799081 745757 1544838            TRUE
## 59  2009    20-24 624565 538927 1163492            TRUE
## 60  2009    25-29 557627 457139 1014766            TRUE
## 61  2009    30-34 436501 391919  828420            TRUE
## 62  2009    35-39 333542 330907  664449            TRUE
## 63  2009    40-44 281004 276149  557153            TRUE
## 65  2009    50-54 196356 192875  389231            TRUE
## 71  2009      80+  44504  42779   87283            TRUE
```

Le même résultat s'obtient avec la fonction `subset`.

```
subset(x = pop_groupage_2009, subset = femme_sup_homme == TRUE)
```

```
##    annee groupage  femme  homme   total femme_sup_homme
## 58  2009    15-19 799081 745757 1544838            TRUE
## 59  2009    20-24 624565 538927 1163492            TRUE
## 60  2009    25-29 557627 457139 1014766            TRUE
## 61  2009    30-34 436501 391919  828420            TRUE
## 62  2009    35-39 333542 330907  664449            TRUE
```

```
## 63  2009    40-44 281004 276149  557153            TRUE
## 65  2009    50-54 196356 192875  389231            TRUE
## 71  2009      80+  44504  42779   87283            TRUE
```

L'on peut utiliser directement introduire le critère à l'intérieur du *data frame*...

```
pop_groupage_2009[pop_groupage_2009$femme > pop_groupage_2009$homme, ]
```

```
##    annee groupage  femme  homme   total femme_sup_homme
## 58  2009    15-19 799081 745757 1544838            TRUE
## 59  2009    20-24 624565 538927 1163492            TRUE
## 60  2009    25-29 557627 457139 1014766            TRUE
## 61  2009    30-34 436501 391919  828420            TRUE
## 62  2009    35-39 333542 330907  664449            TRUE
## 63  2009    40-44 281004 276149  557153            TRUE
## 65  2009    50-54 196356 192875  389231            TRUE
## 71  2009      80+  44504  42779   87283            TRUE
```

...ou à l'intérieur de la fonctio `subset`.

```
subset(x = pop_groupage_2009, subset = femme > homme)
```

```
##    annee groupage  femme  homme   total femme_sup_homme
## 58  2009    15-19 799081 745757 1544838            TRUE
## 59  2009    20-24 624565 538927 1163492            TRUE
## 60  2009    25-29 557627 457139 1014766            TRUE
## 61  2009    30-34 436501 391919  828420            TRUE
## 62  2009    35-39 333542 330907  664449            TRUE
## 63  2009    40-44 281004 276149  557153            TRUE
## 65  2009    50-54 196356 192875  389231            TRUE
## 71  2009      80+  44504  42779   87283            TRUE
```

Cette dernière approche se révèle simple. A partir de maintenant, nous allons privilégier la fonction `subset`.

Sur la base de ces résultats, on voit clairement que R sait comparer des valeurs numériques. Juste pour confirmer, reprenons sur le groupe d'âge 0-4 ans.

```
1353418 < 1321275
```

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

Qu'en est-il des rééls?

```
1.000002 > 1
```

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

```
# (Notez que l'assignation ses fait avec "=", mais le test d'égalité se fait avec "==")
```

De toute évidence, ça marche avec les nombres. Qu'en est-il des caractères? Testons!

```
"MALI" == "Mali"
```

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

Cette égalité est rejetée par que R est sensible à la taille des lettres (majuscule/minuscule). Maintenant regardons la logique.

```
TRUE == 1
```

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

Vous rappelez-vous quand, dans le cours précédent, nous avons coercé une vecteur de valeurs logiques en y ajoutant un réel comment `TRUE` est devenu `1` et `FALSE` `0`? Et bien, c'est la preuve que pour R, `TRUE` == `1`.

4.2.2 Critères additifs: *et* = &

Il est souvent possible que l'on souhaite combiner plusieurs critères dans la même déclaration. Supposons que l'on veuille connaître les groupes d'âge pour lesquels:

- les femmes sont plus nombreuses que les hommes; et
- la population totale (hommes + femmes) est en dessous de 1 millions de personnes.

Nous commençons par définir nos critères.

```
# femme > homme
pop_groupage_2009$femme_sup_homme <- pop_groupage_2009$femme > pop_groupage_2009$homme
# total > 1000000
pop_groupage_2009$moins_de_1_million <- pop_groupage_2009$total < 1000000
```

Maintenant, combinons les!

```
## subset(x = pop_groupage_2009, subset = femme_sup_homme & moins_de_1_million)
```

Avec l'insertion directe des résultats, l'on obtient la même chose.

```
subset(x = pop_groupage_2009, subset = femme > homme & total < 1000000)
```

```
##    annee groupage  femme  homme  total femme_sup_homme moins_de_1_million
## 61  2009    30-34 436501 391919 828420            TRUE               TRUE
## 62  2009    35-39 333542 330907 664449            TRUE               TRUE
## 63  2009    40-44 281004 276149 557153            TRUE               TRUE
## 65  2009    50-54 196356 192875 389231            TRUE               TRUE
## 71  2009      80+  44504  42779  87283            TRUE               TRUE
```

L'addition de critères se fait avec l'opérateur `&`. Le résultat donne les observations qui répondent à toutes les conditions posées.

4.2.3 Critères alternatifs: *ou* = |

La combinaison de critères dans une déclaration ne se pose pas toujours sous la forme additive. Il arrive qu'on veuille procéder sur la base de: soit…soit… Dans ce cas, il faut une autre expression.

Cherchons par exemple, à connaître les groupe pour lequels:

- soit les femmes sont plus nombreuses que les hommes;
- soit la population totale (hommes + femmes) est en dessous de 1 millions de personnes.

Au lieu du signe `&`, nous utilisons le signe `|`

```
subset(x = pop_groupage_2009, subset = femme > homme | total < 1000000)
```

```
##    annee groupage  femme  homme   total femme_sup_homme moins_de_1_million
## 58  2009    15-19 799081 745757 1544838            TRUE              FALSE
## 59  2009    20-24 624565 538927 1163492            TRUE              FALSE
## 60  2009    25-29 557627 457139 1014766            TRUE              FALSE
## 61  2009    30-34 436501 391919  828420            TRUE               TRUE
## 62  2009    35-39 333542 330907  664449            TRUE               TRUE
## 63  2009    40-44 281004 276149  557153            TRUE               TRUE
## 64  2009    45-49 221709 232779  454488           FALSE               TRUE
## 65  2009    50-54 196356 192875  389231            TRUE               TRUE
## 66  2009    55-59 136852 151319  288171           FALSE               TRUE
## 67  2009    60-64 126022 129916  255938           FALSE               TRUE
## 68  2009    65-69  78677  89929  168606           FALSE               TRUE
## 69  2009    70-74  67433  68569  136002           FALSE               TRUE
```

```
## 70  2009    75-79  36949  41667   78616           FALSE                TRUE
## 71  2009      80+  44504  42779   87283            TRUE                TRUE
## 72  2009       ND      0      0       0           FALSE                TRUE
```

Ici, la validation de l'une des conditions suffit. On voit des groupes au dessus de 1 million de personnes (violation du critère n°2). Toutefois, les femmes y sont plus nombreuses (validation du critère n°1). A l'inverse, certains groupes ont moins de femmes (violation du critère n°1), mais comptent moins d'1 millions de personnes (validation du critère n°2).

Souvent, il arrive qu'on veuille accumuler des critères à l'intérieur d'une seule variable. Supposons que l'on souhaite voir les informations concernant juste les moins de 15 ans. On sait que, dans ce cas, on aura à sélectionner trois groupes d'âge: 0-4, 5-9, et 10-14. La variable `groupage` doit être égale à l'une de ses valeurs. Reprenons la logique des critères alternatifs (soit... soit...).

```
subset(x = pop_groupage_2009, subset = groupage == "0-4" | groupage == "5-9" | groupage == "10-14")
```

```
##    annee groupage   femme   homme   total femme_sup_homme
## 55  2009      0-4 1321275 1353418 2674693           FALSE
## 56  2009      5-9 1178850 1225145 2403995           FALSE
## 57  2009    10-14  882725  935796 1818521           FALSE
##    moins_de_1_million
## 55              FALSE
## 56              FALSE
## 57              FALSE
```

Maintenant, ajoutons au critère de *moins de 15 ans* un autre, celui d'un total de moins de 2 millions, donc `total < 2000000`.

```
subset(x = pop_groupage_2009, subset = (groupage == "0-4" | groupage == "5-9" | groupage == "10-14") &
```

```
##    annee groupage  femme  homme   total femme_sup_homme moins_de_1_million
## 57  2009    10-14 882725 935796 1818521           FALSE              FALSE
```

Il a suffit d'isoler les critères alternatifs entre parenthèses et d'y le critère additif.

4.2.4 Critères opposés: *contraire* = !

Souvent, il arrive que l'on souhaite sélectionner sur la base de l'opposition à un critère. Explorons à travers un exemple.

Plus haut, nous avons défini les groupes où `femme > homme`. Ceci revient à définir les groupes où la condition `homme >= femme` est violée. Voyons comment on part de la négation pour parvenir à ce même résultat.

Rappelons

```
subset(x = pop_groupage_2009, subset = femme > homme)
```

```
##    annee groupage  femme  homme   total femme_sup_homme moins_de_1_million
## 58  2009    15-19 799081 745757 1544838            TRUE              FALSE
## 59  2009    20-24 624565 538927 1163492            TRUE              FALSE
## 60  2009    25-29 557627 457139 1014766            TRUE              FALSE
## 61  2009    30-34 436501 391919  828420            TRUE               TRUE
## 62  2009    35-39 333542 330907  664449            TRUE               TRUE
## 63  2009    40-44 281004 276149  557153            TRUE               TRUE
## 65  2009    50-54 196356 192875  389231            TRUE               TRUE
## 71  2009      80+  44504  42779   87283            TRUE               TRUE
```

Passons maintenant par l'opposé.

```
subset(x = pop_groupage_2009, subset = !(femme <= homme))
```

```
##    annee groupage  femme  homme   total femme_sup_homme moins_de_1_million
## 58  2009    15-19 799081 745757 1544838            TRUE              FALSE
## 59  2009    20-24 624565 538927 1163492            TRUE              FALSE
## 60  2009    25-29 557627 457139 1014766            TRUE              FALSE
## 61  2009    30-34 436501 391919  828420            TRUE               TRUE
## 62  2009    35-39 333542 330907  664449            TRUE               TRUE
## 63  2009    40-44 281004 276149  557153            TRUE               TRUE
## 65  2009    50-54 196356 192875  389231            TRUE               TRUE
## 71  2009      80+  44504  42779   87283            TRUE               TRUE
```

En termes d'aperçu, nous avons le même résultat. Pour confirmer, sauvegardons les deux résultats sous forme de nouvelles variables dans le *data frame*, puis comparons-les.

```
pop_groupage_2009$femme_sup_homme <- pop_groupage_2009$femme > pop_groupage_2009$homme
pop_groupage_2009$homme_pas_sup_femme <- !(pop_groupage_2009$homme >= pop_groupage_2009$femme)
pop_groupage_2009$femme_sup_homme == pop_groupage_2009$homme_pas_sup_femme
```

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

Souvent le nombre d'observations est trop grand pour que l'on puisse inspecter à l'oeil le résultat de la déclaration pour toutes les observations. Il existe des fonctions qui permettent de conduire l'examen au niveau global. C'est le cas de la fonction `identical`.

```
identical(pop_groupage_2009$femme_sup_homme, pop_groupage_2009$homme_pas_sup_femme)
```

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

Les deux vecteurs sont donc identiques. Les deux procédés mènent donc au même résultat.

La négation revèle toute son utilité quand on cherche à examiner les données sur la base de l'exclusion plutôt que celle de la sélection. Prenons un exemple dans notre cas. Supposons que nous souhaitions faire la somme des populations sans les enfants de moins de 5 ans. Dans ce cas, plutôt que de sélectionner les groupes qui sont au dessus de 5 ans, il s'avère plus commode d'exclure les moins de 5 ans.

```
# Les groupes homme, femme et total, avec l'exclusion de 0-4 ans
pop_groupage_2009_plus5ans <- subset(x = pop_groupage_2009, subset = groupage != "0-4", select = c(homme
```

Le résultat est la même chose que la ligne suivante.

```
# Les groupes homme, femme et total, avec l'exclusion de 0-4 ans
pop_groupage_2009_plus5ans <- subset(x = pop_groupage_2009, subset = !(groupage == "0-4"), select = c(h
```

La preuve.

```
identical(subset(x = pop_groupage_2009, subset = groupage != "ND"),
          subset(x = pop_groupage_2009, subset = !(groupage == "ND")))
```

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

Faisons les sommes pour la population restrainte.

```
# Vous rappelez-vous la fonction colSums du chapitre précédent?
colSums(pop_groupage_2009_plus5ans)
```

```
##    homme    femme    total
##  5851572  6002397 11853969
```

Et maintenant, juste pour comparer, regardons sur la population globale.

```
# Les groupes homme, femme et total, sans aucun critère
pop_groupage_2009_avec5ans <- subset(x = pop_groupage_2009, select = c(homme, femme, total))
# Les sommmes
colSums(pop_groupage_2009_avec5ans)
```

```
##    homme    femme    total
##  7204990  7323672 14528662
```

4.2.5 Conditionalités

Jusque là, nous avons parlé de déclarations dans une formulation simple. Nous les avons pas inscrites dans le cadre d'un arbre de décision. Il s'agit du schéma suivant: "si condition remplie, alors action 1, sinon action 2". On délègue à la machine l'exécution de tâches sur la base de critères définis... ce qui est pratiquement le début de l'intelligence artificielle.

Dans notre example, nous avons vu qu'entre les hommes et les femmes, la supériorité numérique varie d'un groupe d'âge à un autre. Nous pouvons souhaiter générer une variable qui indiquera lequel des groupes est plus nombreux. Pour ce faire, R dispose de la fonction `ifelse`.

```
pop_groupage_2009$sup_num <- ifelse(# condition
                                    test = pop_groupage_2009$femme > pop_groupage_2009$homme,
                                    # action si condition satisfaite
                                     yes = "femme > homme",
                                    # action si condition non satisfaite
                                     no = "femme <= homme"
                                     )
```

Regardons ce que celà donne.

```
head(pop_groupage_2009)
```

```
##    annee groupage   femme   homme   total        sup_num
## 55  2009      0-4 1321275 1353418 2674693 femme <= homme
## 56  2009      5-9 1178850 1225145 2403995 femme <= homme
## 57  2009    10-14  882725  935796 1818521 femme <= homme
## 58  2009    15-19  799081  745757 1544838  femme > homme
## 59  2009    20-24  624565  538927 1163492  femme > homme
## 60  2009    25-29  557627  457139 1014766  femme > homme
```

Avec cette nouvelle variable, nous pouvons déterminer, par exemple, le nombre de groupes pour lesquels il y a plus de femmes que d'hommes et vice-versa.

```
table(pop_groupage_2009$sup_num)
```

```
##
## femme <= homme  femme > homme
##             10              8
```

4.3 Les boucles

4.3.1 La solution aux tâches répétititives

Un grand avantage de la programmation est la capacité de déléguer à la machine l'exécution de tâches répétitives. R dispose de diverses fonctions qui permettent d'effectuer celles-ci en boucle. Ceci est très commode surtout quand le nombre de répétitions est élevé. Toutefois, la nécessité des boucles varie d'un objet à un autre. Si pour certains, des solutions alternatives et plus simples existent, pour d'autres, elles sont la meilleure option.

Dans ce chapitre, nous allons nous limiter à la fonction *for*. Vous pouvez regarder la fonction *while*. Entrez dans la console: *help("while")*.

4.3.2 La fonction *for*

La fonction `for` est très pratique pour l'exécution des boucles dans `R`. Elle est structurée de la façon suivante:

```
for(var in seq){
  expr
}
```

où `var` désigner une variable dans la séquence `seq` et `expr` la transformation à laquelle l'on soumet les éléments de cette dernière. Un exemple.

```
for(i in c(1:10)){
  print(i^2)
  }
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

Pour chaque i élément de la séquence allant de 1 à 10, nous affichons le carré de i.

4.3.3 Application sur vecteurs

Prenons un vecteur de chiffres.

```
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Utilisons une boucle avec la fonction `for` pour élever les éléments à leur carré et stockons dans un vecteur nommé `y`.

```
# Création d'une coquille vide de vecteur.
y <- c()
  # Pour chaque élément dans le vecteur x,
for(i in x){
  # créer un élément dans le vecteur y qui en serait le carré.
  y[[i]] <- i^2
  }
```

Regardons y

```
y
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Ici, la boucle marche parfaitement, mais on peut s'en passer. Reprenons l'opération, mais avec une approche différente.

```
# Le vecteur de départ
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# L'élévation au carré
```

```
y <- x^2
y
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Même résultat. Moins de codage. Donc solution optimale! La fonction native (^) s'exécute déjà en boucle sur tous les éléments du vecteur.

Pour nous assurer que cette règle n'est pas limité qu'aux chiffres, testons avec les lettres. Prenons un vecteur de caractères.

```
x <- c("Mamadou", "Amadou", "Ahmed", "Ahmad", "Abdoul", "Zan", "Tchiè", "Mady")
```

Cherchons à détecter les prénoms qui contiennent la lettre "a" (en minuscule). R a des fonctions natives qui peuvent exécuter cette tâche dont `grepl`.

```
# Création d'une coquille vide de vecteur.
y <- c()
  # Pour chaque élément dans le vecteur x,
for(i in x){
  # identifier les éléments contenant la lettre "a".
  y[i] <- grepl(pattern = "(a)", x = i)
}
```

Regardons y.

```
y
```

```
## Mamadou  Amadou   Ahmed   Ahmad  Abdoul     Zan   Tchiè    Mady
##    TRUE    TRUE   FALSE    TRUE   FALSE    TRUE   FALSE    TRUE
```

Encore une fois, on peut remarquer que R est sensible à la taille de la lettre (minuscule/majuscule). Regardez les résultats pour `Ahmad` et `Ahmed`. Reprenons en appliquant directement la formule au vecteur directement.

```
y <- grepl(pattern = "(a)", x)
y
```

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

Même résultat. Moins de codage. Donc solution optimale! La fonction native `grepl` s'exécute déjà en boucle sur tous les éléments du vecteur. Leçon: chaque fois, qu'une fonction native existe et peut exécuter une tâche, il est préférable de se passer de la boucle.

4.3.4 Application sur matrices

Maintenant, essayons sur une matrice.

```
x <- matrix(data = c(1:12), nrow = 3, byrow = TRUE)
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

Comme avant, élévons les éléments à leur carré et stockons dans une matrice nommée `y`.

```
# Création d'une coquille vide de matrice
y <- c()
  # Pour chaque élément dans la matrice x,
for(i in x){
```

```
  # créer un élément dans la matrice y qui en serait le carré.
  y[[i]] <- i^2
}
```

Regardons y.

```
y
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100 121 144
```

Ici la boucle donne le bon résultat, mais pas le bon format. Nous cherchons une matrice, mais c'est un vecteur que nous avons eu. Apparemment, la boucle doit aussi tenir compte du format. Ajustons-donc le format de la matrice qui recevra les résultats. Créons une coquille vide de matrice.

```
y <- matrix(data = rep(NA, times = 12), nrow = 3, byrow = TRUE)
y
```

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

Et reprenons la boucle.

```
  # Pour chaque ligne (i) de la matrice x, et
for(i in 1:nrow(x)){
    # pour chaque colonne (j) de la matrice x,
  for(j in 1:ncol(x)){
    # créer un élément dans la matrice y qui en serait le carré.
    y[i, j] <- x[i, j]^2
  }
}
```

Regardons y.

```
y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    9   16
## [2,]   25   36   49   64
## [3,]   81  100  121  144
```

Nous avons le bon résultat et le bon format. Mais que de lignes de codes!!!! Il doit y avoir une voie plus simple!

Maintenant, regardons une autre solution: l'implémentation directe du la formule (^2) sur la matrice de départ.

```
y <- x^2
y
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    9   16
## [2,]   25   36   49   64
## [3,]   81  100  121  144
```

A l'instar du vecteur, l'on peut appliquer des formules directements aux matrices. L'objet qui en résulte hérite de la structure et du format de la matrice de départ.

Ce qui marche pour les chiffres, marche-t-il pour les lettres aussi? Comme pour les vecteurs, testons avec une matrice de caractères. Considérons la matrice suivante.

```
x <- matrix(data = c("Zégoua", "Hamdallaye", "Zanbougou",
                     "Farimaké","Cinzani", "Tinzawatene",
                     "Nara", "Hawa Dembaya", "Bozobougou"),
            nrow = 3, byrow = TRUE)
x
```

```
##      [,1]       [,2]           [,3]
## [1,] "Zégoua"   "Hamdallaye"   "Zanbougou"
## [2,] "Farimaké" "Cinzani"      "Tinzawatene"
## [3,] "Nara"     "Hawa Dembaya" "Bozobougou"
```

Cherchons-y dans les éléments qui contiennent la lettre "z" (minuscule!). Appliquons directement la formule à la matrice `x`.

```
y <- grepl(pattern = "(z)", x)
y
```

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

Nous avons le bon résultat, mais pas le bon format. R a généré le résultat sous format de vecteur. Ce qui en érode fortement la lisibilité. Ajustons! Nous pouvons générer le résultat et le déclarer sous le format de matrice.

```
# étape 1
y <- grepl(pattern = "(z)", x)
# étape 2
y <- matrix(data = y, nrow = 3, byrow = TRUE)
y
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,] FALSE  TRUE FALSE
## [3,] FALSE  TRUE  TRUE
```

Ou tout simplement combiner les deux étapes.

```
# combinaison des 2 étapes
y <- matrix(data = grepl(pattern = "(z)", x), nrow = 3, byrow = TRUE)
y
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,] FALSE  TRUE FALSE
## [3,] FALSE  TRUE  TRUE
```

Malgré cet ajustement, l'application directe de la formule est préférable à la boucle car une fonction native existe déjà pour l'exécution de la tâche souhaitée. Sachant que les matrices sont fortement sollicitées en algèbre, il n'est pas surprenant de trouver que le format est respecté quand les opérations pour sur des chiffres, mais défait quand il s'agit de lettres ou caractères.

4.3.5 Application sur *data frame*

Partant de ce qu'on a vu avec les vecteurs et les matrices, on peut se douter que les boucles ne sont pas toujours le meilleur choix pour les data frame non plus.

Supposons que l'on veuille calculer pour chaque groupe d'âge de notre data frame l'écart entre les femmes et les hommes: `femme - homme`. On pourrait faire une boucle:

```
ecart_femme_homme <- c()
for(i in 1:nrow(pop_groupage_2009)){
  ecart_femme_homme[i] <- pop_groupage_2009[i, "femme"] - pop_groupage_2009[i, "homme"]
}
ecart_femme_homme
```

```
##  [1] -32143 -46295 -53071  53324  85638 100488  44582   2635   4855 -11070
## [11]   3481 -14467  -3894 -11252  -1136  -4718   1725      0
```

Un détour for peu utile quand on peut faire plus simple.

```
pop_groupage_2009$ecart_femme_homme <- pop_groupage_2009$femme - pop_groupage_2009$homme
head(x = pop_groupage_2009, n = 3)
```

```
##    annee groupage   femme   homme   total        sup_num ecart_femme_homme
## 55  2009      0-4 1321275 1353418 2674693 femme <= homme            -32143
## 56  2009      5-9 1178850 1225145 2403995 femme <= homme            -46295
## 57  2009    10-14  882725  935796 1818521 femme <= homme            -53071
```

4.3.6 Application sur listes

C'est avec les listes que les boucles prennent tout leur sens. Les vecteurs, matrices et data frame constituent tous des objets unitaires eux-mêmes. Ils ont leur propriétés propres à eux-mêmes (structure et comportements). Ceci veut dire qu'ils prêtent à l'assimilation par les fonctions. Celles-ci vont systématiquement s'appliquer sur tous les éléments désignés au sein de l'objet. Qu'il s'agisse d'une opération mathématiques (élévation au carré) ou de l'examen de texte (détection d'un caractère), l'objet peut servir d'intrant direct à la fonction utilisée dans la boucle.

Avec la liste, les choses sont différentes. La liste est un objet *hôte*. Bien qu'elle ait ses propriétés, elle sert de contenant à d'autres objets. De ce fait, elle peut abriter plusieurs objets sur lesquels l'on peut souhaiter exécuter la même opération en boucle. Et c'est là, qu'on est content que les boucles existent!

Illustrons!

Rappelons d'abord les noms des objets contenus dans notre liste `pop_groupage_list`.

```
names(pop_groupage_list)
```

```
## [1] "1976" "1987" "1998" "2009"
```

Maintenant, commençons avec le simple affichage de la première observation de tous les *data frame* de la liste.

```
  # Pour chaque élément de la liste
for(i in pop_groupage_list) {
  # Assigner l'affichage de la 1ère observation à une variable
  obs1 <- head(x = i, n = 1)
  # Affiche toutes les 1ères variables extraites
  print(obs1)
}
```

```
##   annee groupage  femme  homme   total
## 1  1976      0-4 589394 587015 1176409
##    annee groupage  femme  homme   total
## 19  1987      0-4 713507 719804 1433311
##    annee groupage  femme  homme   total
## 37  1998      0-4 824505 839795 1664300
##    annee groupage   femme   homme   total
## 55  2009      0-4 1321275 1353418 2674693
```

Nous avons vu plus haut qu'avec les déclarations conditionnelles, l'on peut exécuter des tâches sur la base d'un arbre de décision. Maintenant, imaginez que vous avez à répéter une même tâche sur plusieurs objets. Nous avons vu que la liste contient 4 data frames, tirés de 4 recensements (1976, 1987, 1998 et 2009). Imaginez que vous souhaitez déterminer qui des hommes et des femmes sont les plus nombreux et ce pour tous les années de recensement. Là, vous allez devoir définir une tâche et l'exécuter en boucle. Pensez-vous comme un agent de vaccination qui passe dans toutes les concessions (*data frame*) d'une rue (liste) pour vacciner des enfants (le test `femme > homme`). Générons dans chacun des *data frame* une variable `femme_sup_homme` qui est vrai (`TRUE`) quand `femme > homme` et faux (`FALSE`) dans le cas contraire.

```
  # Pour chaque élément "i" de la liste "pop_groupage_list"
for(i in pop_groupage_list){
  # Exécuter l'opération "femme > homme"
  i[, "femme_sup_homme"] <- i[, "femme"] > i[, "homme"]
  # Assigner l'affichage des 3 premières observations à une variable
  obs3 <- head(x = i, n = 3)
  # Afficher toutes les 3 premières observations extraites.
  print(obs3)
}
```

```
##   annee groupage  femme  homme   total femme_sup_homme
## 1  1976      0-4 589394 587015 1176409            TRUE
## 2  1976      5-9 482851 492272  975123           FALSE
## 3  1976    10-14 321959 342807  664766           FALSE
##    annee groupage  femme  homme   total femme_sup_homme
## 19  1987      0-4 713507 719804 1433311           FALSE
## 20  1987      5-9 611562 633206 1244768           FALSE
## 21  1987    10-14 414302 452166  866468           FALSE
##    annee groupage  femme  homme   total femme_sup_homme
## 37  1998      0-4 824505 839795 1664300           FALSE
## 38  1998      5-9 797057 830211 1627268           FALSE
## 39  1998    10-14 589603 637495 1227098           FALSE
##    annee groupage   femme   homme   total femme_sup_homme
## 55  2009      0-4 1321275 1353418 2674693           FALSE
## 56  2009      5-9 1178850 1225145 2403995           FALSE
## 57  2009    10-14  882725  935796 1818521           FALSE
```

Allons plus loin en enrichissant les conditions. Voici la démarche:

- sélectionnons seulement les moins de 15 ans: groupage est 0-4 ou 5-9 ou 10-14;
- créons ensuite une colonne `test_max` qui indique qui des hommes ou des femmes a la supériorité numérique;
- créons ensuite une colonne `valeur_max` qui donne la valeur de la population.

```
  # Pour chaque élément "i" de la liste "pop_groupage_list"
for(i in pop_groupage_list){
  # sélection des groupes d'âge dans 0-15 ans.
  i <- i[i["groupage"]=="0-4" | i["groupage"]=="5-9" | i["groupage"]=="10-14",]
  # déclarations conditionnelles pour les variables "test_max" et "valeur_max".
  i[, "test_max"] <- ifelse(# condition
                            test = i[,"femme"] > i[,"homme"],
                            # action si condition satisfaite
                            yes = "femme",
                            # action si condition non satisfaite
                            no = "homme")
  i[, "valeur_max"] <- ifelse(# condition
```

```
                              test = i[,"femme"] > i[,"homme"],
                              # action si condition satisfaite
                              yes = i[,"femme"],
                              # action si condition non satisfaite
                              no = i[,"homme"])
  # Afficher toutes les 2 premières extraites.
  print(head(x = i, n = 2))
}
```

```
##   annee groupage  femme  homme   total test_max valeur_max
## 1  1976      0-4 589394 587015 1176409    femme     589394
## 2  1976      5-9 482851 492272  975123    homme     492272
##    annee groupage  femme  homme   total test_max valeur_max
## 19  1987      0-4 713507 719804 1433311    homme     719804
## 20  1987      5-9 611562 633206 1244768    homme     633206
##    annee groupage  femme  homme   total test_max valeur_max
## 37  1998      0-4 824505 839795 1664300    homme     839795
## 38  1998      5-9 797057 830211 1627268    homme     830211
##    annee groupage   femme   homme   total test_max valeur_max
## 55  2009      0-4 1321275 1353418 2674693    homme    1353418
## 56  2009      5-9 1178850 1225145 2403995    homme    1225145
```

4.3.7 Arrêtons-nous un instant!

Qu'avons-nous vu jusque là? Nous avons vu comment:

- poser des critères et les insérer dans des déclarations ;
- poser un raisonnement en arbre de décision avec les déclarations conditionnelles ;
- les boucles marchent avec divers objets (vecteurs, matrices, data frame et listes).

Nous avons vu que c'est avec les listes que les boucles révèlent leur plus grande utilité. Il se trouve que R contient aussi des fonctions taillées spécialement pour tourner des fonctions en boucle sur les éléments d'une liste. Dans R-base seulement, il y a une grande famille de fonction dont `lapply`, `sapply`, `vapply`, `tapply`, `mapply`, `rapply`, `eapply`... Toutes ces fonctions sont des outils du paradigme `split-apply-combine` qui consiste à :

- diviser des données en morceaux ;
- à appliquer sur chaque morceau une fonction donnée ;
- à rassembler les résultats en un nouveau morceau.

Nous allons nous limiter à `lapply` ici. Explorons `lapply` avec quelques exemples.

4.3.8 Paradigme *split-apply-combine*: illustration avec *lapply*

Considérons la liste suivante avec deux vecteurs et deux matrices:

```
malist <- list(monvect2 = seq(from = 0, to = 20, by = 0.5),
               monvect1 = rnorm(20, mean = 9.88, sd = 1.23),
               mamat1 = matrix(data = c(1:20), nrow = 4, byrow = TRUE),
               mamat2 = matrix(data = rnorm(20, mean = 7.43, sd = 1.80), nrow = 4, byrow = TRUE)
               )
```

Regardons le contenu.

```
str(malist)
```

```
## List of 4
##  $ monvect2: num [1:41] 0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 ...
##  $ monvect1: num [1:20] 9.75 10.53 9.34 11.43 8.51 ...
##  $ mamat1  : int [1:4, 1:5] 1 6 11 16 2 7 12 17 3 8 ...
##  $ mamat2  : num [1:4, 1:5] 8.65 7.59 9.9 6.88 7.49 ...
```

Pour chaque objet de la liste, procédons à une agrégation avec la fonction `sum`.

```
for(i in malist){
  print(sum(i))
  }
```

```
## [1] 410
## [1] 200.8776
## [1] 210
## [1] 160.1925
```

Faisons la même chose avec "lapply"

```
lapply(X = malist, FUN = sum)
```

```
## $monvect2
## [1] 410
##
## $monvect1
## [1] 200.8776
##
## $mamat1
## [1] 210
##
## $mamat2
## [1] 160.1925
```

Le même résultat est obtenu avec `lapply`, sous la forme d'une nouvelle liste.

Testons encore! Au lieu des sommes, générons cette fois-ci les moyennes de chaque objet de la liste. Avec la boucle...

```
for(i in malist){
  print(mean(i))
  }
```

```
## [1] 10
## [1] 10.04388
## [1] 10.5
## [1] 8.009627
```

...et avec `lapply`

```
lapply(X = malist, FUN = mean)
```

```
## $monvect2
## [1] 10
##
## $monvect1
## [1] 10.04388
##
```

```
## $mamat1
## [1] 10.5
##
## $mamat2
## [1] 8.009627
```

Vous voyez la logique?

Nous avons vu que, pour les matrices et les vecteurs, l'on trouve souvent des fonctions qui sont déjà capables d'exécuter les tâches souhaitées (ne paniquez pas, avec la pratique, votre répertoire de fonctions s'agrandira!) Et quand celà est possible, une boucle n'est pas nécessaire. Le même principe va pour les listes. Quand il y a des fonctions natives qui peuvent a) exécuter la tâche souhaitée (générer une somme ou une moyenne par exemple) et b) insérer cette tâche dans une boucle (exécuter sur tous les objets d'une liste), alors, il est préférable d'embrasser cette voie. Les exemples précédents ont clairement illustré celà.

Contiuons avec d'autres exemples pour illustrer davantage. Cherchons à connaître les dimensions des objets de la liste (nombre de lignes, nombres de colonnes).

```
lapply(X = pop_groupage_list, FUN = dim)
```

```
## $`1976`
## [1] 18  5
##
## $`1987`
## [1] 18  5
##
## $`1998`
## [1] 18  5
##
## $`2009`
## [1] 18  5
```

Et le nom des variables?

```
lapply(X = pop_groupage_list, FUN = colnames)
```

```
## $`1976`
## [1] "annee"    "groupage" "femme"    "homme"    "total"
##
## $`1987`
## [1] "annee"    "groupage" "femme"    "homme"    "total"
##
## $`1998`
## [1] "annee"    "groupage" "femme"    "homme"    "total"
##
## $`2009`
## [1] "annee"    "groupage" "femme"    "homme"    "total"
```

C'est simple et efficace. N'est-ce pas?

Maintenant, supposons qu'on veut déterminer la population totale pour chaque année en faisant la somme de la colonne `total`. Comment faire ? Avec une boucle, c'est facile.

```
for(i in pop_groupage_list){
  print(sum(i["total"]))
  }
```

```
## [1] 6392918
## [1] 7696349
```

```
## [1] 9810912
## [1] 14528662
```

Avec `lapply`.

```
lapply(X = pop_groupage_list, FUN = sum)
```

```
## Error in FUN(X[[i]], ...): only defined on a data frame with all numeric variables
```

`lapply` n'arrive pas à s'exécuter car nous n'avons pas spécifié qu'à l'intérieur de chaque *data frame*, il fallait prendre la variable `total`! Certes, `lapply` est destinée à exécuter les tâches en boucle, mais encore faudrait-il que celles-ci soient bien définies. Et c'est ça que fait une fonction. Elle exécute des tâches! Et ça, c'est le début d'un autre pan de la Data Science: la programmtion fonctionnelle. Dans ce terme, on va englober, l'art de faire des fonctions. Tout y passe: de la conception à la rapidité. Dans la prochaine, nous allons reprendre les idées déjà présentées, mais cette fois-ci en raisonnant en termes de fonctions.

4.4 Les fonctions

4.4.1 L'épine dorsale de R

Pour un data scientist, les fonctions sont d'une importance capitale car son flux de travail consiste à faire passer les données d'une fonction à une autre pour recadrer ses questions ou trouver des réponses à celles-ci. Depuis le début, nous parlons de fonctions. Qu'est-ce que c'est? Dans R, la fonction agit comme dans les mathématiques. C'est un règle ou une procédure qui détermine la transformation d'un intrant en extrant. Prenons l'exemple suivant:

$$y = f(x) = x^2$$

Dans cet exemple, la fonction élève les intrants au carré pour donner les extrants. Dans R, c'est la même chose! Nous avons déjà mentionnées certaines fonctions et avons montré ce qu'elles font. Revenons sur quelques unes.

4.4.2 Retour sur quelques fonctions

Prenons le vecteur suivant:

```
monvect <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Si nous voulons compter le nombre d'éléments composants cet éléments, une fonction...

```
length(monvect)
```

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

... faire la somme de ces éléments, une fonction...

```
sum(monvect)
```

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

... faire la moyenne de ces éléments, une fonction...

```
mean(monvect)
```

```
## [1] 5.5
```

Revenons a notre liste. Pour voir sa structure, une fonction...

```
str(pop_groupage_list)
```

```
## List of 4
##  $ 1976:'data.frame':    18 obs. of  5 variables:
##   ..$ annee   : num [1:18] 1976 1976 1976 1976 1976 ...
##   ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ femme   : num [1:18] 589394 482851 321959 333508 265842 ...
##   ..$ homme   : num [1:18] 587015 492272 342807 308607 218391 ...
##   ..$ total   : num [1:18] 1176409 975123 664766 642115 484233 ...
##  $ 1987:'data.frame':    18 obs. of  5 variables:
##   ..$ annee   : num [1:18] 1987 1987 1987 1987 1987 ...
##   ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ femme   : num [1:18] 713507 611562 414302 379522 315753 ...
##   ..$ homme   : num [1:18] 719804 633206 452166 348200 260215 ...
##   ..$ total   : num [1:18] 1433311 1244768 866468 727722 575968 ...
##  $ 1998:'data.frame':    18 obs. of  5 variables:
##   ..$ annee   : num [1:18] 1998 1998 1998 1998 1998 ...
##   ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ femme   : num [1:18] 824505 797057 589603 529270 409584 ...
##   ..$ homme   : num [1:18] 839795 830211 637495 492480 364333 ...
##   ..$ total   : num [1:18] 1664300 1627268 1227098 1021750 773917 ...
##  $ 2009:'data.frame':    18 obs. of  5 variables:
##   ..$ annee   : num [1:18] 2009 2009 2009 2009 2009 ...
##   ..$ groupage: Ord.factor w/ 18 levels "0-4"<"5-9"<"10-14"<..: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ femme   : num [1:18] 1321275 1178850 882725 799081 624565 ...
##   ..$ homme   : num [1:18] 1353418 1225145 935796 745757 538927 ...
##   ..$ total   : num [1:18] 2674693 2403995 1818521 1544838 1163492 ...
```

Pour voir le nom des éléments qu'elle contient, une fonction

```
names(pop_groupage_list)
```

```
## [1] "1976" "1987" "1998" "2009"
```

Vous voyez l'idée?

4.4.3 Pourquoi faire une fonction?

Vu la richesse de R on peut bien être amené à se demander pourquoi se donner la peine de faire une fonction. N'en exist-il pas déjà dans R ? La plupart du temps, oui! Mais pas tout le temps.

Autant, sont nombreuses les questions que la data scientist soulève, autant les voies qui s'offrent à lui pour y répondre sont variées. Les particularités de la question peuvent faire qu'il est souhaitable voire indispensable de `personnaliser` la réponse. D'où la nécéssité de créer de nouvelles fonctions. Celles-ci peuvent aussi bien s'incorporer dans le flux de travail que prendre intégralement celui-ci en charge.

4.4.4 Les basiques de la fonction

Bien que tous les sept milliards d'humains peuplant la terre partagent cette appélation commune, il demeure que l'on s'attache à donner à chacun d'entre eux une appélation particulière: le prénom! N'est-ce pas? De même, une fonction a besoin d'un *nom*! A ce niveu, il est utile d'indiquer qu'il y a des mots réservés qui ne peuvent pas être utilisés. Voir:

```
help("reserved")
```

Après le nom, il y a les *arguments.* Ceci est l'appélation donnée aux intrants. Ensuite on a le *corps* qui est la procédure à laquelle sont soumis ces intrants. Schématisons tout ça!

```
mafonction <- function(x){
  x^2
  }
```

Nous avons défini ici une fonction, `mafonction`, où `x` est l'argument et la procédure à laquelle il est soumis est l'élévation au carré. Testons le résultat.

```
mafonction(x = 25)
```

```
## [1] 625
```

Juste pour ironiser un peu, rappelons que c'est par une fonction, `function`, que nous venons de créer une fonction. Trop méta, R !!!!!!

Avançons un peu ici en créant une fonction avec deux arguments: x et y.

```
mafonction <- function(x, y){
  x + y
  }
```

Testons

```
mafonction(x = 1, y = 2)
```

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

Souvent, il est possible d'assigner à un argument ou à tous une valeur par défaut.

```
mafonction <- function(x, y = 10){
  x + y
  }
```

Regardons ce qu'on obtient quand on ne spécifie pas la valeur passée à l'argument y.

```
mafonction(x = 3)
```

```
## [1] 13
```

Certaines fonctions contiennent plusieurs arguments. Par commodité, on a assigne à beaucoup des valeurs par défaut qui sont validées sauf si l'utilisateur en décide autrement.

Souvent, la fonction comprends des étapes intermédiaires. A vrai dire, c'est dans ça que réside la nécéssité des fonctions, le séquençage de procédures multiples en une seule commande. Revenant à notre example, nous pouvons assigner le résultat à une variable intermédiaire z.

```
mafonction <- function(x, y = 10){
  z <- x + y
  }
mafonction(x = 3)
```

L'exécution de la fonction sur le chiffre 3 n'a pas donné de résultat car nous n'avons pas demandé à la fonction d'afficher celui-ci. Pour que le résultat sorte, il faut expliciter.

```
mafonction <- function(x, y = 10){
  z <- x + y
  return(z)
  }
mafonction(x = 3)
```

```
## [1] 13
```

`return()` est très commode quand on doit passer par plusieurs étapes à l'intérieur de la fonction.

4.4.5 Fonctions et boucles

Les fonctions (écriture, évaluation, etc.) constitue un domaine vaste de la data science. Nous ne pourrons pas tout voir d'un seul coup. La maîtrise des règles (les do's et les don't's) viennent avec la pratique. Ayant couvert les basiques, nous allons retourner à nos données pour illustrer. Vous vous rappelez la boucle suivante...

```
for(i in pop_groupage_list){
  print(sum(i["total"]))
  }
```

```
## [1] 6392918
## [1] 7696349
## [1] 9810912
## [1] 14528662
```

...qu'on avait pas réussi à insérer dans la fonction `lapply`? Elle marche. La raison est simple. `lapply` exécute des fonctions sur les objets contenus dans une liste. Elle ne peut pas systématiquement atteindre les éléments contenus dans ces objets. Nous avons pu faire des moyennes et des médianes sur des vecteurs et matrices à partir de `lapply`. La raison était simple: ces objets sont des ensembles homogènes. Ils contenaient tous des chiffres, qui sont des éléments assimilables par `mean` et `median`. Or, le `data frame` est un objet hétérogène, contenant des chiffres et des lettres. Les fonctions natives qu'on a utilisées ne peuvent faire la différence d'elles-mêmes. Elles doivent être guidées. De ce fait, nous devons inscrire la procédure souhaitée dans une fonction avant de passer celle-ci à `lapply` qui va l'exécuter en boucle sur tous les objets de la liste. Voici la fonction:

```
pop_somme <- function(df){
  sum(df["total"])
  }
```

Nous venons de définir une fonction où `df` est l'argument principal. On s'attend à un *data frame* comme intrant. On s'attend à ce que celui-ci ait une colonne nommée `total` dont les éléments seront agrégés par la fonction `sum`. Vous voyez ? C'est une solution très personnalisée! Regardons les résultats!

```
for(i in pop_groupage_list){
  print(sum(i["total"]))
}
```

```
## [1] 6392918
## [1] 7696349
## [1] 9810912
## [1] 14528662
```

Voici le résultat pour `lapply`.

```
lapply(X = pop_groupage_list, FUN = pop_somme)
```

```
## $`1976`
## [1] 6392918
##
## $`1987`
## [1] 7696349
##
## $`1998`
## [1] 9810912
##
## $`2009`
## [1] 14528662
```

Qu'est-ce qui est préférable? Comme avant, les fonctions existantes sont toujours meilleures. `lapply` est

intégrée à R. Elle constitue une meilleure boucle. Aussi, elle peut génère en extrant une liste, qui peut être assignée à un objet donné.

Part II

Manipuler des données dans R

Chapter 5

Importer des données dans R

5.1 Introduction

5.1.1 Objectif de ce cours

Dans le flux de travail (*workflow*) du *data scientist*, l'importation constitue très généralement le point de départ. Les données ne sont toujours disponibles sous le format qui se prête à l'analyse souhaitée. Elles peuvent exister dans un classeur Excel sous format *xls*, *xlsx* ou *csv*. Elles peuvent aussi se trouver dans une base de données relationnelles, où diverses tables sont connectées entres elles. Elles peuvent même être disponibles sur Internet (page Wikipédia, Twitter, Facebook, etc.) Dans tous les cas, il revient au *data scientist* de connaitre la techique appropriée pour les importer dans son environnement de travail, les organiser et les analyser selon l'objectif qu'il s'assigne.

Dans ce chapitre, nous allons voir quelques techniques d'importation de données dans R.

5.1.2 Que nous faut-il?

- R (évidemment) et RStudio (de préférence) installés sur le poste de travail
- les fichiers fournis dans le cadre du module
- un connexion Internet pour illustrer les exemples d'importation depuis la toile

5.1.3 Données

Nous allons illustrer ce chapitre avec une compilation de données tirées des Recensements Généraux de la Population et de l'Habitat au Mali, menés respectivement en 1976, 1987, 1998 et 2009. Le tableau suivant affiche un aperçu (l'année 1976 seulement) des données dont il est question.

Notre fichier se nomme: *"RGPH_MLI"*. Nous l'avons enregistré sous divers formats, qui seront traduits par diverses extensions (.csv, .txt, .xls, .xlsx, etc). Nous allons les ouvrir progressivement.

Pour permettre l'exécution des codes depuis n'importe quel poste de travail, nous allons utiliser la plateforme Github. Toutefois, il convient de noter que les codes seront tout aussi valides que si l'on accède aux fichiers au niveau local.

Nous allons spécifier la source.

```
masource <- "https://raw.githubusercontent.com/fousseynoubah/dswr_slides/master/4_Importer_Donnees_dans_
```

A partir de celle-ci, nous allons opérer comme au niveau local (à condition d'avoir une connexion Internet active bie sûr). Pour accéder à un fichier quelconque, `monfichier`, disponible dans le dossier, il suffit d'orienter sa requête vers:

Table 5.1: Un aperçu des données démographiques

annee	id	groupage	Homme-Urbain	Homme-Rural	Femme-Urbain	Femme-Rural	source	office
1976	1	0-4	100416	486599	99453	489941	RGPH	DNSI
1976	2	5-9	80816	411456	82665	400186	RGPH	DNSI
1976	3	10-14	60647	282160	65075	256884	RGPH	DNSI
1976	4	15-19	58662	249945	60765	272743	RGPH	DNSI
1976	5	20-24	46239	172152	47487	218355	RGPH	DNSI
1976	6	25-29	36390	163705	43143	223875	RGPH	DNSI
1976	7	30-34	30895	154834	33504	190446	RGPH	DNSI
1976	8	35-39	27926	133457	28505	137444	RGPH	DNSI
1976	9	40-44	22479	116947	22007	125822	RGPH	DNSI
1976	10	45-49	17864	93466	15445	83008	RGPH	DNSI
1976	11	50-54	14692	89927	13238	90369	RGPH	DNSI
1976	12	55-59	10659	66919	8928	53989	RGPH	DNSI
1976	13	60-64	8630	67990	9337	72129	RGPH	DNSI
1976	14	65-69	5043	35236	5032	31800	RGPH	DNSI
1976	15	70-74	3460	28430	4610	33137	RGPH	DNSI
1976	16	75-79	2084	15065	2650	14080	RGPH	DNSI
1976	17	80+	2188	25958	3677	29082	RGPH	DNSI
1976	99	ND	125	272	93	281	RGPH	DNSI
1976	18	Total	529215	2594518	545614	2723571	RGPH	DNSI

```
paste0(masource, "monfichier" )
```

```
## [1] "https://raw.githubusercontent.com/fousseynoubah/dswr_slides/master/4_Importer_Donnees_dans_R/da
```

5.2 Fichiers plats: cas du format *CSV*

5.2.1 Aperçu

Comme son nom l'indique, le format *CSV* (*Comma Separated Values*) est un format où les valeurs de données rectangulaires sont séparées par une virgule (`,`). Cette règle de base est commode pour les anglophones, pour lesquels les décimales viennent après un point (`.`) et non après une virgule. Par contre, pour les francophones, le format de sauvegarde de données réquiert une règle différente. Pour ne pas confondre le séparateur de valeurs et la virgule de la décimale, l'on utilise le point-virgule (`;`). Nous verrons les fonctions qui conviennent pour chacun de ces deux formats.

5.2.2 Importation avec *R-base*: `read.csv`

La version d'installation de R, couramment appelée *R-base* (ou *base-R*), vient avec un ensemble de fonctions qui sont disponibles par défaut. Parmi celles-ci: `read.csv`. L'importation d'un fichier *csv* avec *R-base* se fait de la façon suivante:

```
RGPH_MLI <- read.csv(
  # Spécifier le nom du fichier
  file = paste0(masource, "RGPH_MLI.csv"),
  # Déclarer la première ligne comme nom de colonne/variable
  header = TRUE,
  # Déclarer la virgule comme séparateur
  sep = ",",
  # Ne pas systématiquement transformer les caractères en facteurs.
  stringsAsFactors = TRUE
```

```
)
```

Les éléments à l'intérieur de la fonction (`file`, `header`, `sep`, `stringsAsFactors`, etc.) sont les arguments de la fonction `read.csv`. Celles-ci ont des valeurs défaut, qui peuvent toutefois être altérer par l'utilisateur. Pour voir ces valeurs par défaut, consulter la documentation intégrer à R en entrant:

```
?read.csv
```

Regardons la structure de nos données.

```
str(RGPH_MLI)
```

```
## 'data.frame':    77 obs. of  14 variables:
##  $ ï..annee    : int  1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 ...
##  $ id          : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ groupage    : Factor w/ 20 levels "","0-4","10-14",..: 2 11 3 4 5 6 7 8 9 10 ...
##  $ Homme.Urbain: int  100416 80816 60647 58662 46239 36390 30895 27926 22479 17864 ...
##  $ Homme.Rural : int  486599 411456 282160 249945 172152 163705 154834 133457 116947 93466 ...
##  $ Femme.Urbain: int  99453 82665 65075 60765 47487 43143 33504 28505 22007 15445 ...
##  $ Femme.Rural : int  489941 400186 256884 272743 218355 223875 190446 137444 125822 83008 ...
##  $ Homme       : int  587015 492272 342807 308607 218391 200095 185729 161383 139426 111330 ...
##  $ Femme       : int  589394 482851 321959 333508 265842 267018 223950 165949 147829 98453 ...
##  $ Urbain      : int  199869 163481 125722 119427 93726 79533 64399 56431 44486 33309 ...
##  $ Rural       : int  976540 811642 539044 522688 390507 387580 345280 270901 242769 176474 ...
##  $ Total       : int  1176409 975123 664766 642115 484233 467113 409679 327332 287255 209783 ...
##  $ source      : Factor w/ 2 levels "","RGPH": 2 2 2 2 2 2 2 2 2 2 ...
##  $ office      : Factor w/ 2 levels "","DNSI": 2 2 2 2 2 2 2 2 2 2 ...
```

5.2.3 Importation avec *R-base*: `read.csv2`

Avec read.csv déjà, le séparateur peut être spécifié pour permettre la prise en charge des fichiers qui ont point-virgule comme séparateur (`;`). Toutefois, il existe une fonction bâtie au dessus de celle-ci, qui prend en charge les spécificités de tels fichiers. Il s'agit de `read.cvs2`.

```
RGPH_MLI2 <- read.csv2(
  # Spécifier le nom du fichier
  file = paste0(masource, "RGPH_MLI2.csv"),
  # Déclarer la première ligne comme nom de colonne/variable
  header = TRUE,
  # Déclarer le point-virgule comme séparateur
  sep = ";",
  # Ne pas systématiquement transformer les caractères en facteurs.
  stringsAsFactors = TRUE
)
```

Ici aussi, les valeurs par défaut des arguments sont maintenues. Pour en savoir plus sur `read.csv2`, explorer la documentation R en tapant:

```
?read.csv2
```

Regardons la structure de ces données aussi.

```
str(RGPH_MLI2)
```

```
## 'data.frame':    77 obs. of  14 variables:
##  $ Ã...annee   : int  1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 ...
##  $ id          : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ groupage    : Factor w/ 20 levels "","0-4","10-14",..: 2 11 3 4 5 6 7 8 9 10 ...
```

```
##  $ Homme.Urbain: int  100416 80816 60647 58662 46239 36390 30895 27926 22479 17864 ...
##  $ Homme.Rural : int  486599 411456 282160 249945 172152 163705 154834 133457 116947 93466 ...
##  $ Femme.Urbain: int  99453 82665 65075 60765 47487 43143 33504 28505 22007 15445 ...
##  $ Femme.Rural : int  489941 400186 256884 272743 218355 223875 190446 137444 125822 83008 ...
##  $ Homme       : int  587015 492272 342807 308607 218391 200095 185729 161383 139426 111330 ...
##  $ Femme       : int  589394 482851 321959 333508 265842 267018 223950 165949 147829 98453 ...
##  $ Urbain      : int  199869 163481 125722 119427 93726 79533 64399 56431 44486 33309 ...
##  $ Rural       : int  976540 811642 539044 522688 390507 387580 345280 270901 242769 176474 ...
##  $ Total       : int  1176409 975123 664766 642115 484233 467113 409679 327332 287255 209783 ...
##  $ source      : Factor w/ 2 levels "","RGPH": 2 2 2 2 2 2 2 2 2 2 ...
##  $ office      : Factor w/ 2 levels "","DNSI": 2 2 2 2 2 2 2 2 2 2 ...
```

5.2.4 Importation avec `readr`

5.2.4.1 Aperçu

readr est un package créé par Hadley Wickham. Ses fonctions sont similaires à celles de `read.csv` et de `read.csv2`. `readr` présente l'avantage de faire partie du *tidyverse*. Il travaille harmonieusement avec les autres packages de cet écosystème. Sa syntaxe est très simple: pour importer un fichier csv, on utilise `read_csv` au lieu de `read.csv`. Cette logique est valable pour d'autres formats, dont il suffit seulement d'ajouter l'extension après le tiret d'en bas ("_"). Ainsi, on a :

- `read_csv2`: pour les fichiers *CSV* ayant le point-virgule (`;`) comme séparateur;
- `read_tsv`: pour les fichiers avec les valeurs séparées par des tabulations;
- `read_fwf`: pour les fichiers avec les valeurs séparées par des espaces fixes.

5.2.4.2 `read_csv` et `read_csv2`

Ouvrons le fichier csv avec `read_csv`.

```
# Chargement du package "readr"
library(readr)
# Importation du fichier
RGPH_MLI <- read_csv(
  # Spécifier le nom du fichier
  file = paste0(masource, "RGPH_MLI.csv"),
  # Déclarer la première ligne comme nom de colonne/variable
  col_names = TRUE,
  # Indiquer la valeur à attribuer aux cellules vides
  na = "",
  # Nombre de lignes à ne pas importer, partant du ficher
  skip = 0
  )
```

et `read_csv2`

```
# Chargement du package "readr"
library(readr)
# Importation du fichier
RGPH_MLI <- read_csv2(
  # Spécifier le nom du fichier
  file = paste0(masource, "RGPH_MLI2.csv"),
  # Déclarer la première ligne comme nom de colonne/variable
  col_names = TRUE,
  # Indiquer la valeur à attribuer aux cellules vides
  na = "",
```

```
  # Nombre de lignes à ne pas importer, partant du ficher
  skip = 0
  )
```

5.2.4.3 Généralisation

`read_csv` est un cas spécifique d'une fonction généraliste qui couvre un large spectre de formats : `read_delim`.

```
# Chargement du package "readr"
library(readr)
# Importation du fichier
RGPH_MLI <- read_delim(
  # Spécifier le nom du fichier
  file = paste0(masource, "RGPH_MLI.csv"),
  # Indiquer le séparateur
  delim = ","
  )

RGPH_MLI2 <- read_delim(
  # Spécifier le nom du fichier
  file = paste0(masource, "RGPH_MLI2.csv"),
  # Indiquer le séparateur
  delim = ";"
  )
```

Avec `read_delim`, divers format de fichiers peuvent être importés, à partir du moment où le séparateur est bien spécifié:

- virgule (`,`);
- point-virgule (`;`);
- tabulation (`\t`);
- barre verticale (`|`);
- espace (" "); etc.

Les autres arguments peuvent être ajustés pour prendre en compte les spécificités des données. Toutefois, il faut noter que `read_delim` est assez intuitive pour détecter le type des données (entiers, réels, caractères, etc.). Elle inspecte jusqu'à 1000 lignes (et peut aller jusqu'à la nième ligne) pour déterminer le type de données d'une colonne.

5.3 Excel: `xls`, `xlsx`

5.3.1 Aperçu

Faisant partie de la suite MS Office, Excel est l'un des tableurs les plus populaires. Il est difficile de l'exclure du *workflow* du *data scientist* car c'est l'outil de prédilection dans beaucoup de domaines et secteurs (entreprises, administrations) en matière d'organisation et de sauvegarde des données . Les *data scientist* opérant exclusivement sur des bases de données constituent une niche. Pour la majorité, amenée à interagir avec des spécialistes d'autres domaines, il est important de pourvoir disposer de techniques pour importer les données collectées et organisées par leurs soins dans Excel.

Dans R, plusieurs solutions existent. Nous verrons deux packages:

- *xlsx*; et
- *readxl*.

5.3.2 Importation avec `xlsx`: `read.xlsx`

Le package `xlsx` offre deux fonctions majeures pour lire des formats `xls` et `xlsx`. Il s'agit de `read.xlsx` et `read.xlsx2` (plus rapide sur les fichiers lourds).

Pour le format `xls`...

```
# Chargement du package "xlsx"
library(xlsx)

# Ne pouvant accéder au fichier depuis le net,
# il faut le télécharger localement d'abord.
library(downloader)
download(url = paste0(masource, "RGPH_MLI.xls?raw=true"),
         dest = "RGPH_MLI.xls",
         mode="wb")

# Exemple avec fichier Excel 97-2003
RGPH_MLI_xls <- read.xlsx(
  # Spécifier le nom du fichier (format "xls")
  file = "RGPH_MLI.xls",
  # Indiquer le numéro d'ordre de l'onglet à importer
  sheetIndex = 1,
  #  sheetName = "RGPH_MLI" # indiquer le nom de l'onglet (alternative au numéro d'ordre)
  # Déclarer la première ligne comme nom de colonne/variable
  header = TRUE
)
```

...et pour le format `xlsx`

```
# Chargement du package "xlsx"
library(xlsx)

# Ne pouvant accéder au fichier depuis le net,
# il faut le télécharger localement d'abord.
library(downloader)
download(url = paste0(masource, "RGPH_MLI.xlsx?raw=true"),
         dest = "RGPH_MLI.xlsx",
         mode="wb")

# Exemple avec fichier Excel 2007-plus
RGPH_MLI_xlsx <- read.xlsx(
  # Spécifier le nom du fichier  (format "xlsx")
  file = "RGPH_MLI.xlsx",
  # Indiquer le numéro d'ordre de l'onglet à importer
  sheetIndex = 1,
  #  sheetName = "RGPH_MLI" # indiquer le nom de l'onglet (alternative au numéro d'ordre)
  # Déclarer la première ligne comme nom de colonne/variable
  header = TRUE
)
```

5.3.3 Importation avec `readxl`: `read_excel`

Le package `readxl` est bâti sur les mêmes principes que `readr`. Il permet d'importer des formats Excel avec la même logique syntaxique. Il contient des fonctions spécifiques, `read_xls` et `read_xlsx`, et une fonction généraliste, `read_excel`.

```r
# Chargement du package "readxl"
library(readxl)

# Exemple avec fichier Excel 97-2003
RGPH_MLI_xls <- read_excel(
# Spécifier le nom du fichier (format "xls")
path = "RGPH_MLI.xls",
# indiquer le nom de l'onglet ou le numéro d'ordre, les deux sont acceptés
sheet = "RGPH_MLI",
# Déclarer la première ligne comme nom de colonne/variable
col_names = TRUE
)

# Exemple avec fichier Excel 2007-plus
RGPH_MLI_xlsx <- read_excel(
# Spécifier le nom du fichier (format "xls")
path = "RGPH_MLI.xlsx",
# indiquer le nom de l'onglet ou le numéro d'ordre, les deux sont acceptés
sheet = "RGPH_MLI",
# Déclarer la première ligne comme nom de colonne/variable
col_names = TRUE
)
```

5.4 Formats issues d'autres logiciels statistiques: Stata et SPSS

5.4.1 Aperçu

Le data scientist peut aussi être amené à travailler sur des données sauvegardées à partir d'autres programmes de traitement de données tels que `Stata`, `SPSS`, `SAS`, etc. Dans R, les solutions sont nombreuses. Ici, nous allons voir deux packages: `foreign` et `haven`.

5.5 Importation avec `foreign`

Le package `foreign` permet d'ouvrir des fichiers `dta`, issus de `Stata`...

```r
# Chargement du package "foreign"
library(foreign)

# Ne pouvant accéder au fichier depuis le net,
# il faut le télécharger localement d'abord.
library(downloader)
download(url = paste0(masource, "RGPH_MLI.dta?raw=true"),
         dest = "RGPH_MLI.dta",
         mode="wb")

# Importation
RGPH_MLI_stata <- read.dta(
  # Spécifier le nom du fichier
  file = "RGPH_MLI.dta",
  # Conversion des dates du format Stata au format R (pour dire simple)
  convert.dates = TRUE,
  # Conversion des étiquettes de variables catégorielles en facteurs
  convert.factors = TRUE,
```

```
  # Convertir "_" de Stata en "." dans R (surtout nom des variables)
  convert.underscore = FALSE,
  # Donner ou pas une valeur particulière aux cellules vides?
  missing.type = FALSE,
  # Si oui, cet argument indique si cette valeur est présente ou pas
  warn.missing.labels = TRUE
)
```

et des fichiers `sav`, issus de `SPSS`.

```
# Chargement du package "foreign"
library(foreign)

# Ne pouvant accéder au fichier depuis le net,
# il faut le télécharger localement d'abord.
library(downloader)
download(url = paste0(masource, "RGPH_MLI.sav?raw=true"),
         dest = "RGPH_MLI.sav",
         mode="wb")

# Importation

RGPH_MLI_spss <- read.spss(
  # Spécifier le nom du fichier
  file = "RGPH_MLI.sav",
  # Conservation des étiquettes de variables catégorielles (facteurs)
  use.value.labels = TRUE,
  # Importation en data frame au lieu de matrice
  to.data.frame = TRUE
  )
```

5.6 Base de données relationnelles

5.6.1 Aperçu

R peut aussi importer des données depuis une base de données relationnelles. Celle-ci peut aussi bien être locale, installée sur le poste de travail, que distante, installée sur un serveur accessible via Internet. Nous allons illustrer ici avec une base locale.

Les packages varient d'un type de base à un autre:

- *RMySQL* pour MySQL,
- *RPostgreSQL* pour PostgreSQL
- *RSQLite*: pour SQLite
- etc.

Pour plus de détails, consulter cette page.

5.6.2 Importation avec *RODBC*

Commençons par ouvrir le chaine de communication entre R et la base via ODBC (*Open Data Base Connectivity*).

```
# Chargement du package "RODBC"
library(RODBC)
dswr <- odbcConnect(
  # Indiquer le nom de la chaîne de connection
  dsn = "dswr",
  # Indiquer l'identifiant (s'il y'en a)
  uid = "",
  # Indiquer le mot de passe (s'il y'en a)
  pwd = ""
  )
```

Maintenant, importons la table qui nous intéresse, *RGPH_MLI*. Pour celà, deux méthodes sont possibles.

```
# Méthode 1: extraction de la table
RGPH_MLI_rodbc_tbl <- sqlFetch(
  # Indiquer le nom de la chaîne de connection
  dswr,
  # Indiquer le nom de la table
  sqtable = "RGPH_MLI"
  )

# Méthode 2: requête SQL
RGPH_MLI_rodbc_sql <- sqlQuery(
  # Indiquer le nom de la chaîne de connection
  channel = dswr,
  # Selectionner toutes les colonnes et lignes de la table "RGPH_MLI"
  query = "SELECT * FROM RGPH_MLI;"
  )
```

Une fois les extractions de données finie, il faut penser à briser la chaîne de connection, fermer la porte.

```
odbcClose(dswr)
```

5.6.3 Importation avec `odbc`

Une autre solution est de passe par le package `odbc`.

```
# Chargement du package "odbc"
library(odbc)
# Chargement du package "DBI"
library(DBI)

# Importation
dswr <- dbConnect(
  # Indiquer le package utilisé par l'interfacee "DBI"
  odbc::odbc(),
  # Indiquer le nom de la chaîne de connection
  "dswr"
  )
```

Comme avant, on peut importer la table d'intérêt par deux méthodes.

```
# Méthode 1: extraction de la table
RGPH_MLI_odbc <- dbReadTable(
  # Indiquer le nom de la chaîne de connection
  conn = dswr,
  # Indiquer le nom de la table indiquer
```

```
  name = "RGPH_MLI"
  )

# Méthode 2: requête SQL
RGPH_MLI_odbc_sql <- dbGetQuery(
  # Indiquer le nom de la chaîne de connection
  conn = dswr,
  # Selectionner toutes les colonnes et lignes de la table "RGPH_MLI"
  "SELECT * FROM RGPH_MLI;"
  )
```

Comme avant, en bon invité, on ferme la porte en sortant.

```
dbDisconnect(dswr)
```

5.7 Depuis Internet

5.7.1 Aperçu

Les données peuvent aussi être tirées de la toile mondiale. Les outils disponibles dans R varient selon le type de données.

5.7.2 Chargement de fichier CSV

Pour un fichier CSV, le chargement dans l'environnement R se fait de la même façon que pour des fichiers locaux.

```
url <- "https://raw.githubusercontent.com/fousseynoubah/dswr_slides/master/4_Importer_Donnees_dans_R/da
RGPH_MLI_csv_online <- read.csv(url)
# ou
RGPH_MLI_csv_online <- read_csv(url)
```

Chapter 6

Transformer de données avec dplyr

6.1 Introduction

6.1.1 Objectif de ce chapitre

Le *data scientist* a très rarement les données structurées dans la forme qui lui convient. Il lui revient de les mettres dans cette forme. De ce fait, il lui est indispensable de savoir manipuler les données. Il s'agit de la maîtrise d'une série de tâches parmi lesquelles nous pouvons citer:

- la simple sélection d'un sous-ensemble à l'intérieur d'un large groupe ;
- la sélection d'un nombre déterminé de variables (attributs) ;
- la combinaison d'informations conservées dans différentes `data frame`;
- la suppresion et la création variables;
- la réorganisation des données à l'intérieur d'un `data frame`.

Dans ce chapitre, nous allons voir quelques techniques de manipulation de données avec le package **dplyr**.

6.1.2 Que nous faut-il?

- R (évidemment) et RStudio (de préférence) installés sur le poste de travail;
- le package **dplyr** installés;
- les fichiers fournis dans le cadre du module.

6.1.3 Données

Nous allons illustrer ce chapitre avec une compilation de données tirées des Recensements Généraux de la Population et de l'Habitat au Mali, menés respectivement en 1976, 1987, 1998 et 2009. Il s'agit:

- d'un côté, de tableaux sur la population par groupe d'âge (tranche de 5 ans);
- de l'autre, de tableaux sur la population par commune.

6.2 Aperçu de dplyr

6.2.1 Installer dplyr

Le package **dplyr** est un package créé par Hadley Wickham. Il fait partie de l'écosystème **tidyverse** qui est un ensemble de packages conçus pour la *data science* et partageant tous les mêmes philoshophie, grammaire

et structure. Nous allons, avec l'exploration des packages du **tidyverse**, comprendre l'importance de cette grammaire. Elle traduit la volonté des auteurs de condenser dans le nom des fonctions l'idée de la tâche que celles-ci exécutent. Dans **dplyr**, les fonctions majeures sont des verbes (comme nous allons le voir).

Alors, qu'est-ce que **dplyr** fait? Voici une simple analogie. Si les données constituent un tissu, nous pouvons voir **dplyr** comme à la fois la paire de ciseaux pour en faire la coupe, et l'aiguille et le fil pour le coudre. Pour le prouver, voici le logo associé au package.

Pour travailler avec **dplyr**, l'on commence par l'installer sur son poste de travail. Etant intégré au **tidyverse**, il est possible de l'installer en même temps que les autres composantes de celui-ci.

```
install.packages("tidyverse")
```

L'on peut aussi l'installer tout seul.

```
install.packages("dplyr")
```

Cette commande télécharge depuis le réseau d'archivage des packages R la dernière version stable publiée. Toutefois, il est aussi possible d'acquérir les versions en développement depuis Github.

```
install.packages("devtools") # au cas où ce package n'est pas installé
devtools::install_github("tidyverse/dplyr")
```

Une fois l'installation effectuée, l'on peut charger le package.

```
library(dplyr)
```

6.2.2 L'objet `tibble`

Avant d'aller loin, il est utile d'introduire la notion de `tibble` qui est une partie intégrante des packages du **tidyverse**. Il s'agit de la même chose que le `data frame`, mais celui-ci repensé. Le `tibble` n'altère pas la structure ni la classe des données contenue dans un `data frame`. Par contre, il agit sur l'affichage des données et, comme nous le verrons plus tard, leur permet de stocker de nouvelles sortes de données (`colonne-list`, à noter quelque part... on y reviendra). Pour l'instant regardons-le à l'oeuvre en termes d'affichage.

Prenons les données sur les populations des communes du Mali en 2009 (RGPH, 2009). Nous allons charger les données depuis la source suivante:

```
source_donnees <- "https://raw.githubusercontent.com/fousseynoubah/dswr_slides/master/5_Manipuler_Donnees_dans_R/data/adm3_pop_2009.csv"
```

```
# Importation des données
adm3_pop_2009 <- read.csv(source_donnees)
```

Quelle est la classe de cet objet?

```
class(adm3_pop_2009)
```

```
## [1] "data.frame"
```

C'est un `data frame`! Regardons sa structure ainsi que les six premières observations

```
str(adm3_pop_2009)
```

```
## 'data.frame':    703 obs. of  9 variables:
##  $ id        : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ admin0_nom: Factor w/ 1 level "Mali": 1 1 1 1 1 1 1 1 1 1 ...
##  $ admin1_nom: Factor w/ 9 levels "Bamako","Gao",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ admin2_nom: Factor w/ 50 levels "Abeibara","Ansongo",..: 23 23 23 23 23 23 23 23 23 23 ...
##  $ admin3_nom: Factor w/ 687 levels "Abeibara","Adarmalane",..: 25 74 116 156 192 210 231 238 249 257
##  $ annee     : int  2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 ...
##  $ homme     : num  6123 6144 7115 11466 5141 ...
##  $ femme     : int  5974 6353 7015 12091 5017 1934 3927 1903 10234 3445 ...
```

```
##  $ source    : Factor w/ 1 level "RGPH": 1 1 1 1 1 1 1 1 1 1 ...
```

```
head(adm3_pop_2009)
```

```
##   id admin0_nom admin1_nom admin2_nom admin3_nom annee homme femme source
## 1  1       Mali      Kayes      Kayes   Bangassi  2009  6123  5974   RGPH
## 2  2       Mali      Kayes      Kayes  Colimbine  2009  6144  6353   RGPH
## 3  3       Mali      Kayes      Kayes     Diamou  2009  7115  7015   RGPH
## 4  4       Mali      Kayes      Kayes   Djelebou  2009 11466 12091   RGPH
## 5  5       Mali      Kayes      Kayes     Faleme  2009  5141  5017   RGPH
## 6  6       Mali      Kayes      Kayes      Fegui  2009  1999  1934   RGPH
```

Maintenant, déclarons cet objet comme un `tibble`.

```
adm3_pop_2009 <- as_tibble(adm3_pop_2009)
```

Revoyons la classe.

```
class(adm3_pop_2009)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

La classe a changé, ou plutôt elle s'est consolidée. De nouvelles caractéristiques ont été ajoutées à `data.frame`. Maintenant, imprimons le jeu de données lui-même.

```
adm3_pop_2009
```

```
## # A tibble: 703 x 9
##       id admin0_nom admin1_nom admin2_nom admin3_nom annee homme femme
##    <int> <fct>      <fct>      <fct>      <fct>      <int> <dbl> <int>
##  1     1 Mali       Kayes      Kayes      Bangassi    2009  6123  5974
##  2     2 Mali       Kayes      Kayes      Colimbine   2009  6144  6353
##  3     3 Mali       Kayes      Kayes      Diamou      2009  7115  7015
##  4     4 Mali       Kayes      Kayes      Djelebou    2009 11466 12091
##  5     5 Mali       Kayes      Kayes      Faleme      2009  5141  5017
##  6     6 Mali       Kayes      Kayes      Fegui       2009  1999  1934
##  7     7 Mali       Kayes      Kayes      Gory Gope~  2009  3939  3927
##  8     8 Mali       Kayes      Kayes      Goumera     2009  1918  1903
##  9     9 Mali       Kayes      Kayes      Guidimaka~  2009  9798 10234
## 10    10 Mali       Kayes      Kayes      Hawa Demb~  2009  3406  3445
## # ... with 693 more rows, and 1 more variable: source <fct>
```

Avec les `tibble`, l'affichage d'un jeu de données donne par défaut les dix premières observations et indique la classe et/ou le type des colonnes. Nous voyons par exemple que la variable `id` est composée d'entiers tandis que la variable `admin0_nom` est déclarée en facteur. De ce fait, l'impression d'un `tibble` permet de combiner les résultats qu'on obtiendrait avec les commandes `str` et `head`.

Le `tibble` est l'output par défaut des fonctions du **tidyverse**. Reprenons l'opération d'importation. Cette fois-ci, au lieu de passer par `read.csv` qui est une fonction basique, nous utiliserons `read_csv` du package **readr**, lui-même membre du club **tidyverse**.

```
# Chargement du package "readr"
library(readr)

# Importation des données
adm3_pop_2009 <- read_csv(source_donnees)

# Classe
class(adm3_pop_2009)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
# Impression
adm3_pop_2009
```

```
## # A tibble: 703 x 9
##       id admin0_nom admin1_nom admin2_nom admin3_nom annee homme femme
##    <dbl> <chr>      <chr>      <chr>      <chr>      <dbl> <dbl> <dbl>
##  1     1 Mali       Kayes      Kayes      Bangassi    2009  6123  5974
##  2     2 Mali       Kayes      Kayes      Colimbine   2009  6144  6353
##  3     3 Mali       Kayes      Kayes      Diamou      2009  7115  7015
##  4     4 Mali       Kayes      Kayes      Djelebou    2009 11466 12091
##  5     5 Mali       Kayes      Kayes      Faleme      2009  5141  5017
##  6     6 Mali       Kayes      Kayes      Fegui       2009  1999  1934
##  7     7 Mali       Kayes      Kayes      Gory Gope~  2009  3939  3927
##  8     8 Mali       Kayes      Kayes      Goumera     2009  1918  1903
##  9     9 Mali       Kayes      Kayes      Guidimaka~  2009  9798 10234
## 10    10 Mali       Kayes      Kayes      Hawa Demb~  2009  3406  3445
## # ... with 693 more rows, and 1 more variable: source <chr>
```

Vous voyez!

Même si l'on passe un `data frame` à une fonction du **tidyverse**, le résultat est un `tibble`. Prenons la fonction `glimpse` du package **dplyr** (notez que *glimpse* veut dire aperçu en anglais).

```
# Importation des données
adm3_pop_2009 <- read.csv(source_donnees)

# Aperçu avec "glimpse"
glimpse(adm3_pop_2009)
```

```
## Observations: 703
## Variables: 9
## $ id         <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...
## $ admin0_nom <fct> Mali, Mali, Mali, Mali, Mali, Mali, Mali, Mali, Mal...
## $ admin1_nom <fct> Kayes, Kayes, Kayes, Kayes, Kayes, Kayes, Kayes, Ka...
## $ admin2_nom <fct> Kayes, Kayes, Kayes, Kayes, Kayes, Kayes, Kayes, Ka...
## $ admin3_nom <fct> Bangassi, Colimbine, Diamou, Djelebou, Faleme, Fegu...
## $ annee      <int> 2009, 2009, 2009, 2009, 2009, 2009, 2009, 2009, 200...
## $ homme      <dbl> 6123, 6144, 7115, 11466, 5141, 1999, 3939, 1918, 97...
## $ femme      <int> 5974, 6353, 7015, 12091, 5017, 1934, 3927, 1903, 10...
## $ source     <fct> RGPH, RGPH, RGPH, RGPH, RGPH, RGPH, RGPH, RGPH, RGP...
```

```
# Comparons à "str"
str(adm3_pop_2009)
```

```
## 'data.frame':    703 obs. of  9 variables:
##  $ id        : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ admin0_nom: Factor w/ 1 level "Mali": 1 1 1 1 1 1 1 1 1 1 ...
##  $ admin1_nom: Factor w/ 9 levels "Bamako","Gao",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ admin2_nom: Factor w/ 50 levels "Abeibara","Ansongo",..: 23 23 23 23 23 23 23 23 23 23 ...
##  $ admin3_nom: Factor w/ 687 levels "Abeibara","Adarmalane",..: 25 74 116 156 192 210 231 238 249 257
##  $ annee     : int  2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 ...
##  $ homme     : num  6123 6144 7115 11466 5141 ...
##  $ femme     : int  5974 6353 7015 12091 5017 1934 3927 1903 10234 3445 ...
##  $ source    : Factor w/ 1 level "RGPH": 1 1 1 1 1 1 1 1 1 1 ...
```

Nous voyons que `glimpse` répartit différemment l'espace en les observations affichées. Il se préoccupe plus de

l'équilibre entre celles-ci que les fonctions comme `str` ou `head`.

Nous allons maintenant explorer les verbes majeurs du package **dplyr**. Dans une démarche pédagogique - pour rappeler les bases déjà vue et enrichir celles-ci, au besoin - nous tenterons autant que possible de reproduire les exemples avec R-base aussi afin de mettre en exergue les différences.

6.3 Selection et/ou exclusion de variables: `select`

Comme nous avons vu précédemment, il arrive souvent que l'on veuille sélectionner certaines variables d'un jeu de données. Dans notre cas, supposons que nous sommes seulement intéressés par les chiffres. Il s'agit des colonnes: `homme` et `femme`.

Rappelons notre jeu de données.

```
# Chargement du package "readr"
library(readr)

# Importation des données
adm3_pop_2009 <- read_csv(source_donnees)

# Impression
adm3_pop_2009
```

```
## # A tibble: 703 x 9
##       id admin0_nom admin1_nom admin2_nom admin3_nom annee homme femme
##    <dbl> <chr>      <chr>      <chr>      <chr>      <dbl> <dbl> <dbl>
##  1     1 Mali       Kayes      Kayes      Bangassi    2009  6123  5974
##  2     2 Mali       Kayes      Kayes      Colimbine   2009  6144  6353
##  3     3 Mali       Kayes      Kayes      Diamou      2009  7115  7015
##  4     4 Mali       Kayes      Kayes      Djelebou    2009 11466 12091
##  5     5 Mali       Kayes      Kayes      Faleme      2009  5141  5017
##  6     6 Mali       Kayes      Kayes      Fegui       2009  1999  1934
##  7     7 Mali       Kayes      Kayes      Gory Gope~  2009  3939  3927
##  8     8 Mali       Kayes      Kayes      Goumera     2009  1918  1903
##  9     9 Mali       Kayes      Kayes      Guidimaka~  2009  9798 10234
## 10    10 Mali       Kayes      Kayes      Hawa Demb~  2009  3406  3445
## # ... with 693 more rows, and 1 more variable: source <chr>
```

Maintenant, passons d'abord par les méthodes de R-base, comme nous avons vu avant.

```
adm3_pop_2009[ , c("homme", "femme")]
```

```
## # A tibble: 703 x 2
##    homme femme
##    <dbl> <dbl>
##  1  6123  5974
##  2  6144  6353
##  3  7115  7015
##  4 11466 12091
##  5  5141  5017
##  6  1999  1934
##  7  3939  3927
##  8  1918  1903
##  9  9798 10234
## 10  3406  3445
## # ... with 693 more rows
```

ou

```
subset(x = adm3_pop_2009, select = c(homme, femme))
```

```
## # A tibble: 703 x 2
##    homme femme
##    <dbl> <dbl>
##  1  6123  5974
##  2  6144  6353
##  3  7115  7015
##  4 11466 12091
##  5  5141  5017
##  6  1999  1934
##  7  3939  3927
##  8  1918  1903
##  9  9798 10234
## 10  3406  3445
## # ... with 693 more rows
```

Dans **dplyr**, l'on utilise pour ce faire la fonction `select` qui va se rapprocher de la fonction `subset` de R-base.

```
select(adm3_pop_2009, homme, femme)
```

```
## # A tibble: 703 x 2
##    homme femme
##    <dbl> <dbl>
##  1  6123  5974
##  2  6144  6353
##  3  7115  7015
##  4 11466 12091
##  5  5141  5017
##  6  1999  1934
##  7  3939  3927
##  8  1918  1903
##  9  9798 10234
## 10  3406  3445
## # ... with 693 more rows
```

Le premier argument de la fonction est le jeu de données. Ceci est général pour toutes les fonctions de **dplyr**. Ensuite viennent les autres arguments. Ici, nous avons énuméré les variables qui nous intéressent. La fonction `select` présente une grande simplicité par rapport aux alternatives de R-base. A l'instar de `subset`, elle ne requiert pas que les noms de variables soient entre griffes (`" "`). Aussi, elle ne requiert pas que l'on spécifie le nombre d'observations à afficher, que l'input soit un `data frame` ou un `tibble`. Avec les fonctions de R-base, quand l'input est une `tibble`, l'affichage est limité à dix observations. Toutefois, cette règle est violée quand l'input est un `data.frame`.

6.3.1 Sélection simple

Il y a un autre avantage avec `select`, mais qui est commun au **tidyverse** tout entier. C'est l'opérateur: **%>%** que l'on appelle *pipe opérator*. Ce signe permet d'unir les lignes de codes en chaîne en leur servant de maillon. Il prend l'objet issu d'une ligne de départ ou d'une opération et fournit le résultat comme intrant à la ligne suivante. De ce fait, en l'utilisant, on n'a pas besoin de mettre dans une fonction l'argument spécifique à l'objet intrant. On peut se limiter aux arguments spécifiques à la tâche exécutée par la fonction. Réécrivons l'opération précédente.

```
adm3_pop_2009 %>%
  select(homme, femme)
```

```
## # A tibble: 703 x 2
##    homme femme
##    <dbl> <dbl>
##  1  6123  5974
##  2  6144  6353
##  3  7115  7015
##  4 11466 12091
##  5  5141  5017
##  6  1999  1934
##  7  3939  3927
##  8  1918  1903
##  9  9798 10234
## 10  3406  3445
## # ... with 693 more rows
```

Avec %>%, nous verbalisons l'idée que nous partons du jeu de données, adm3_pop_2009, que nous soumettons ensuite à une opération de sélection à travers la fonction select. Nous obtenons le même résultat, mais la lisibilité est améliorée. Le signe %>% nous permet d'ordonner les lignes de codes de sorte à rendre compte du séquençage des opérations. A partir de maintenant, nous allons progressivement adoptée cette façon d'écrire les opérations.

6.3.2 Sélection groupée

Un autre avantage de select est de permettre la sélection de plusieurs variables à travers leur communalité. Dans notre jeu, nous avons trois variables qui portent toutes le terme admin comme préfixe. Plutôt que de les sélectionner une à une nous pouvons les appeler toutes ensembles. Commençons par R-base.

```
names_df <- names(adm3_pop_2009)
select_df <- startsWith(x = names_df, prefix = "adm")
adm3_pop_2009[, select_df]
```

```
## # A tibble: 703 x 4
##    admin0_nom admin1_nom admin2_nom admin3_nom
##    <chr>      <chr>      <chr>      <chr>
##  1 Mali       Kayes      Kayes      Bangassi
##  2 Mali       Kayes      Kayes      Colimbine
##  3 Mali       Kayes      Kayes      Diamou
##  4 Mali       Kayes      Kayes      Djelebou
##  5 Mali       Kayes      Kayes      Faleme
##  6 Mali       Kayes      Kayes      Fegui
##  7 Mali       Kayes      Kayes      Gory Gopela
##  8 Mali       Kayes      Kayes      Goumera
##  9 Mali       Kayes      Kayes      Guidimakan Keri Kaff
## 10 Mali       Kayes      Kayes      Hawa Dembaya
## # ... with 693 more rows
```

Cette opération nous a pris trois lignes différentes. Avec select, nous en faisons juste de deux lignes séquencées (même une si l'on veut).

```
adm3_pop_2009 %>%
  select(starts_with("adm"))
```

```
## # A tibble: 703 x 4
```

```
##    admin0_nom admin1_nom admin2_nom admin3_nom
##    <chr>      <chr>      <chr>      <chr>
##  1 Mali       Kayes      Kayes      Bangassi
##  2 Mali       Kayes      Kayes      Colimbine
##  3 Mali       Kayes      Kayes      Diamou
##  4 Mali       Kayes      Kayes      Djelebou
##  5 Mali       Kayes      Kayes      Faleme
##  6 Mali       Kayes      Kayes      Fegui
##  7 Mali       Kayes      Kayes      Gory Gopela
##  8 Mali       Kayes      Kayes      Goumera
##  9 Mali       Kayes      Kayes      Guidimakan Keri Kaff
## 10 Mali       Kayes      Kayes      Hawa Dembaya
## # ... with 693 more rows
```

L'on peut faire la même chose avec les suffixes...

```
adm3_pop_2009 %>%
  select(ends_with("nom"))
```

```
## # A tibble: 703 x 4
##    admin0_nom admin1_nom admin2_nom admin3_nom
##    <chr>      <chr>      <chr>      <chr>
##  1 Mali       Kayes      Kayes      Bangassi
##  2 Mali       Kayes      Kayes      Colimbine
##  3 Mali       Kayes      Kayes      Diamou
##  4 Mali       Kayes      Kayes      Djelebou
##  5 Mali       Kayes      Kayes      Faleme
##  6 Mali       Kayes      Kayes      Fegui
##  7 Mali       Kayes      Kayes      Gory Gopela
##  8 Mali       Kayes      Kayes      Goumera
##  9 Mali       Kayes      Kayes      Guidimakan Keri Kaff
## 10 Mali       Kayes      Kayes      Hawa Dembaya
## # ... with 693 more rows
```

... ou chercher des termes indépendemment de leur position dans le nom des variables.

```
adm3_pop_2009 %>%
  select(contains("nom"))
```

```
## # A tibble: 703 x 4
##    admin0_nom admin1_nom admin2_nom admin3_nom
##    <chr>      <chr>      <chr>      <chr>
##  1 Mali       Kayes      Kayes      Bangassi
##  2 Mali       Kayes      Kayes      Colimbine
##  3 Mali       Kayes      Kayes      Diamou
##  4 Mali       Kayes      Kayes      Djelebou
##  5 Mali       Kayes      Kayes      Faleme
##  6 Mali       Kayes      Kayes      Fegui
##  7 Mali       Kayes      Kayes      Gory Gopela
##  8 Mali       Kayes      Kayes      Goumera
##  9 Mali       Kayes      Kayes      Guidimakan Keri Kaff
## 10 Mali       Kayes      Kayes      Hawa Dembaya
## # ... with 693 more rows
```

dplyr contient une variétés de fonctions similaires dont: `matches()`, `num_range()`, `one_of()`, `everything()`, `group_cols()`. Avec R, l'on peut aboutir au même résultat en utilisant différentes méthodes. La force de **dplyr** - et de `tidyverse` dans une large mesure - est que la méthode est plus économe en écriture de codes

et le séquençage bien explicité à l'aide de l'opérateur %>%.

6.3.3 Index et nom

La sélection peut aussi se faire sur la base de l'index des colonnes, c'est-à-dire leur position. Sélectionnons de la deuxième à la cinquième colonne.

```
# Avec R-base
adm3_pop_2009[, c(2:5)]
```

```
## # A tibble: 703 x 4
##    admin0_nom admin1_nom admin2_nom admin3_nom
##    <chr>      <chr>      <chr>      <chr>
##  1 Mali       Kayes      Kayes      Bangassi
##  2 Mali       Kayes      Kayes      Colimbine
##  3 Mali       Kayes      Kayes      Diamou
##  4 Mali       Kayes      Kayes      Djelebou
##  5 Mali       Kayes      Kayes      Faleme
##  6 Mali       Kayes      Kayes      Fegui
##  7 Mali       Kayes      Kayes      Gory Gopela
##  8 Mali       Kayes      Kayes      Goumera
##  9 Mali       Kayes      Kayes      Guidimakan Keri Kaff
## 10 Mali       Kayes      Kayes      Hawa Dembaya
## # ... with 693 more rows
```

```
# Avec dpylr
adm3_pop_2009 %>%
  select(2:5)
```

```
## # A tibble: 703 x 4
##    admin0_nom admin1_nom admin2_nom admin3_nom
##    <chr>      <chr>      <chr>      <chr>
##  1 Mali       Kayes      Kayes      Bangassi
##  2 Mali       Kayes      Kayes      Colimbine
##  3 Mali       Kayes      Kayes      Diamou
##  4 Mali       Kayes      Kayes      Djelebou
##  5 Mali       Kayes      Kayes      Faleme
##  6 Mali       Kayes      Kayes      Fegui
##  7 Mali       Kayes      Kayes      Gory Gopela
##  8 Mali       Kayes      Kayes      Goumera
##  9 Mali       Kayes      Kayes      Guidimakan Keri Kaff
## 10 Mali       Kayes      Kayes      Hawa Dembaya
## # ... with 693 more rows
```

La même chose peut se faire avec le nom des variables.

```
# Avec R-base
adm3_pop_2009[, c("admin0_nom", "admin1_nom", "admin2_nom", "admin3_nom")]
```

```
## # A tibble: 703 x 4
##    admin0_nom admin1_nom admin2_nom admin3_nom
##    <chr>      <chr>      <chr>      <chr>
##  1 Mali       Kayes      Kayes      Bangassi
##  2 Mali       Kayes      Kayes      Colimbine
##  3 Mali       Kayes      Kayes      Diamou
##  4 Mali       Kayes      Kayes      Djelebou
##  5 Mali       Kayes      Kayes      Faleme
```

```
##  6 Mali       Kayes      Kayes      Fegui
##  7 Mali       Kayes      Kayes      Gory Gopela
##  8 Mali       Kayes      Kayes      Goumera
##  9 Mali       Kayes      Kayes      Guidimakan Keri Kaff
## 10 Mali       Kayes      Kayes      Hawa Dembaya
## # ... with 693 more rows
```

```
# Avec dplyr
adm3_pop_2009 %>%
  select(admin0_nom, admin1_nom, admin2_nom, admin3_nom)
```

```
## # A tibble: 703 x 4
##    admin0_nom admin1_nom admin2_nom admin3_nom
##    <chr>      <chr>      <chr>      <chr>
##  1 Mali       Kayes      Kayes      Bangassi
##  2 Mali       Kayes      Kayes      Colimbine
##  3 Mali       Kayes      Kayes      Diamou
##  4 Mali       Kayes      Kayes      Djelebou
##  5 Mali       Kayes      Kayes      Faleme
##  6 Mali       Kayes      Kayes      Fegui
##  7 Mali       Kayes      Kayes      Gory Gopela
##  8 Mali       Kayes      Kayes      Goumera
##  9 Mali       Kayes      Kayes      Guidimakan Keri Kaff
## 10 Mali       Kayes      Kayes      Hawa Dembaya
## # ... with 693 more rows
```

L'on peut faire plus simple avec **dplyr** quand les variables recherchées se suivent.

```
adm3_pop_2009 %>%
  select(admin0_nom:admin3_nom)
```

6.3.4 Exclusion

Comme avec R-base, la sélection peut aussi se faire sur la base de l'exclusion. Imaginez que l'on souhaite exclure les quatre premières variables.

```
# Avec R-base
adm3_pop_2009[, -c(1:4)]
```

```
## # A tibble: 703 x 5
##    admin3_nom           annee homme femme source
##    <chr>                <dbl> <dbl> <dbl> <chr>
##  1 Bangassi              2009  6123  5974 RGPH
##  2 Colimbine             2009  6144  6353 RGPH
##  3 Diamou                2009  7115  7015 RGPH
##  4 Djelebou              2009 11466 12091 RGPH
##  5 Faleme                2009  5141  5017 RGPH
##  6 Fegui                 2009  1999  1934 RGPH
##  7 Gory Gopela           2009  3939  3927 RGPH
##  8 Goumera               2009  1918  1903 RGPH
##  9 Guidimakan Keri Kaff  2009  9798 10234 RGPH
## 10 Hawa Dembaya          2009  3406  3445 RGPH
## # ... with 693 more rows
```

```
# Avec dplyr
adm3_pop_2009 %>%
  select(-c(1:4))
```

```
## # A tibble: 703 x 5
##    admin3_nom           annee homme femme source
##    <chr>                <dbl> <dbl> <dbl> <chr>
##  1 Bangassi              2009  6123  5974 RGPH
##  2 Colimbine             2009  6144  6353 RGPH
##  3 Diamou                2009  7115  7015 RGPH
##  4 Djelebou              2009 11466 12091 RGPH
##  5 Faleme                2009  5141  5017 RGPH
##  6 Fegui                 2009  1999  1934 RGPH
##  7 Gory Gopela           2009  3939  3927 RGPH
##  8 Goumera               2009  1918  1903 RGPH
##  9 Guidimakan Keri Kaff  2009  9798 10234 RGPH
## 10 Hawa Dembaya          2009  3406  3445 RGPH
## # ... with 693 more rows
```

L'on voit qu'à l'instar de R-base, l'on a juste à précédér les éléments à exclure du signe `-`.

6.3.5 Une pierre deux coups

dplyr compte une fonction `rename` qui permet de renommer les variables. Prise indépendemment, elle agit comme `select`. Supposons que nous voulions changer les noms de variables en majuscules.

```
## Avec R-base
# Sauvegarde des données dans un nouveau data frame
pop_df <- adm3_pop_2009[ , c("homme", "femme")]
# Voir les noms
names(pop_df)
```

```
## [1] "homme" "femme"
```

```
# Changer les noms
names(pop_df) <- c("HOMME", "FEMME")
# Vérification
names(pop_df)
```

```
## [1] "HOMME" "FEMME"
```

```
## Avec dplyr
# Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(homme, femme) %>%
  # Modification des noms de variables
  rename(HOMME = homme, FEMME = femme)
```

```
## # A tibble: 703 x 2
##    HOMME FEMME
##    <dbl> <dbl>
##  1  6123  5974
##  2  6144  6353
##  3  7115  7015
##  4 11466 12091
##  5  5141  5017
##  6  1999  1934
##  7  3939  3927
##  8  1918  1903
##  9  9798 10234
```

```
## 10  3406  3445
## # ... with 693 more rows
```

Nous voyons qu'une telle opération qu'avec R-base requiert la création d'un objet intermédiaire tandis qu'avec **dplyr** elle s'insère tout simplement dans la séquence. Mais `select` peut elle-même prendre en charge la tâche de changement de noms.

```
# Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt et changement de noms
  select(HOMME = homme, FEMME = femme)
```

```
## # A tibble: 703 x 2
##    HOMME FEMME
##    <dbl> <dbl>
##  1  6123  5974
##  2  6144  6353
##  3  7115  7015
##  4 11466 12091
##  5  5141  5017
##  6  1999  1934
##  7  3939  3927
##  8  1918  1903
##  9  9798 10234
## 10  3406  3445
## # ... with 693 more rows
```

L'on indique le nouveau nom suivi du signe `=` et ensuite le nouveau nom.

6.4 Création et/ou suppression de variables: `mutate`

La sélection et la suppression de variables peuvent s'inscrire dans le cadre d'une stratégie d'exploration plus large qui peut elle-même impliquer la création de nouvelles variables.

6.4.1 Création de variable

Revenons à l'exploration de la supériorité numérique entre hommes et femmes, sujet que nous avons abordé dans le chapitre précédent. Supposons que nous souhaitions:

- sélectionner les populations pour les hommes et les femmes pour chaque commune;
- calculer un ratio `femme`/`homme` pour chaque commune.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable
pop_df$ratio <- pop_df$femme / pop_df$homme
# Aperçu
pop_df
```

```
## # A tibble: 703 x 4
##    admin3_nom          homme femme ratio
##    <chr>               <dbl> <dbl> <dbl>
##  1 Bangassi             6123  5974 0.976
##  2 Colimbine            6144  6353 1.03
##  3 Diamou               7115  7015 0.986
```

```
##  4 Djelebou             11466 12091 1.05
##  5 Faleme                5141  5017 0.976
##  6 Fegui                 1999  1934 0.967
##  7 Gory Gopela           3939  3927 0.997
##  8 Goumera               1918  1903 0.992
##  9 Guidimakan Keri Kaff  9798 10234 1.04
## 10 Hawa Dembaya          3406  3445 1.01
## # ... with 693 more rows
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(ratio = homme/femme)
```

```
## # A tibble: 703 x 4
##    admin3_nom           homme femme ratio
##    <chr>                <dbl> <dbl> <dbl>
##  1 Bangassi              6123  5974 1.02
##  2 Colimbine             6144  6353 0.967
##  3 Diamou                7115  7015 1.01
##  4 Djelebou             11466 12091 0.948
##  5 Faleme                5141  5017 1.02
##  6 Fegui                 1999  1934 1.03
##  7 Gory Gopela           3939  3927 1.00
##  8 Goumera               1918  1903 1.01
##  9 Guidimakan Keri Kaff  9798 10234 0.957
## 10 Hawa Dembaya          3406  3445 0.989
## # ... with 693 more rows
```

Le résultat est le même, mais le gain avec **dplyr** est visible. Le séquençage rend la lecture du code facile. Il évite aussi la création d'un objet intermédiaire, comme ce fut le cas de `pop_df` avec R-base.

Dans le cas précédent, la variable était numérique. Elle peut aussi prendre la forme catégorielle. Considérons par exemple des intervalles de population qu'on souhaiterait créer pour séparer les communes en catégories. Nous allons d'abord faire la somme des deux groupes, hommes et femmes, et ensuite créer la variable catégorielle.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Création d'une nouvelle variable: catégorielle
pop_df$pop_cat <- ifelse(pop_df$total < 10000, "<10000", ">=10000")
# Aperçu
pop_df
```

```
## # A tibble: 703 x 5
##    admin3_nom           homme femme total pop_cat
##    <chr>                <dbl> <dbl> <dbl> <chr>
##  1 Bangassi              6123  5974 12097 >=10000
##  2 Colimbine             6144  6353 12497 >=10000
##  3 Diamou                7115  7015 14130 >=10000
##  4 Djelebou             11466 12091 23557 >=10000
```

```
##  5 Faleme                5141  5017 10158 >=10000
##  6 Fegui                 1999  1934  3933 <10000
##  7 Gory Gopela           3939  3927  7866 <10000
##  8 Goumera               1918  1903  3821 <10000
##  9 Guidimakan Keri Kaff  9798 10234 20032 >=10000
## 10 Hawa Dembaya          3406  3445  6851 <10000
## # ... with 693 more rows
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable: numérique
  mutate(total = homme + femme) %>%
  # Création d'une nouvelle variable: catégorielle
  mutate(pop_cat = ifelse(total < 10000, "<10000", ">=10000"))
```

```
## # A tibble: 703 x 5
##    admin3_nom           homme femme total pop_cat
##    <chr>                <dbl> <dbl> <dbl> <chr>
##  1 Bangassi              6123  5974 12097 >=10000
##  2 Colimbine             6144  6353 12497 >=10000
##  3 Diamou                7115  7015 14130 >=10000
##  4 Djelebou             11466 12091 23557 >=10000
##  5 Faleme                5141  5017 10158 >=10000
##  6 Fegui                 1999  1934  3933 <10000
##  7 Gory Gopela           3939  3927  7866 <10000
##  8 Goumera               1918  1903  3821 <10000
##  9 Guidimakan Keri Kaff  9798 10234 20032 >=10000
## 10 Hawa Dembaya          3406  3445  6851 <10000
## # ... with 693 more rows
```

Il est intéréssant de rélever qu'avec cet exemple l'on vient de montrer que la fonction `mutate` prend en charge les déclarations conditionnelles.

A l'instar de `select` qui accepte la liste de toutes les variables à sélection, `mutate` aussi peut, avec un seul appel, exécuter plusieurs opérations de création de variables.

```
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création de deux nouvelles variables: numérique et catégorielle
  mutate(total = homme + femme,
         pop_cat = ifelse(total < 10000, "<10000", ">=10000"))
```

```
## # A tibble: 703 x 5
##    admin3_nom           homme femme total pop_cat
##    <chr>                <dbl> <dbl> <dbl> <chr>
##  1 Bangassi              6123  5974 12097 >=10000
##  2 Colimbine             6144  6353 12497 >=10000
##  3 Diamou                7115  7015 14130 >=10000
##  4 Djelebou             11466 12091 23557 >=10000
##  5 Faleme                5141  5017 10158 >=10000
##  6 Fegui                 1999  1934  3933 <10000
##  7 Gory Gopela           3939  3927  7866 <10000
```

```
##  8 Goumera               1918  1903  3821 <10000
##  9 Guidimakan Keri Kaff  9798 10234 20032 >=10000
## 10 Hawa Dembaya          3406  3445  6851 <10000
## # ... with 693 more rows
```

L'appel de `mutate` n'a pas à être spécifique à une seule variable. La fonction peut effectuer plusieurs opérations lors d'un seul appel.

6.4.2 Suppression de variables

Comme dans R-base, l'on supprime une variable en lui affectant la valeur `NULL`.

```
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  mutate(
    # Création de deux nouvelles variables: numérique et catégorielle
    total = homme + femme,
    pop_cat = ifelse(total < 10000, "<10000", ">=10000"),
    # Suppression de variables
    homme = NULL,
    femme = NULL
    )
```

```
## # A tibble: 703 x 3
##    admin3_nom           total pop_cat
##    <chr>                <dbl> <chr>
##  1 Bangassi             12097 >=10000
##  2 Colimbine            12497 >=10000
##  3 Diamou               14130 >=10000
##  4 Djelebou             23557 >=10000
##  5 Faleme               10158 >=10000
##  6 Fegui                 3933 <10000
##  7 Gory Gopela           7866 <10000
##  8 Goumera               3821 <10000
##  9 Guidimakan Keri Kaff 20032 >=10000
## 10 Hawa Dembaya          6851 <10000
## # ... with 693 more rows
```

6.4.3 Ne garder que le résultat

Une variante de la fonction `mutate` est la fonction `transmute` qui a la particularité de ne garder que les résultats issus des tâches qui lui ont été confiées. Elle présente certes une approche radicale par rapport à une suppression sélective des variables, mais elle est très commode pour celui que n'est intéréssé que par les résultats. Elle se distingue ainsi de `mutate` qui, quant à elle, préserve les variables préexistantes à l'opération de création de nouvelles colonnes

Reprenons le dernier exemple.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Création d'une nouvelle variable: catégorielle
pop_df$pop_cat <- ifelse(pop_df$total < 10000, "<10000", ">=10000")
```

```
# Suppression de variables
pop_df$admin3_nom <- NULL
pop_df$homme <- NULL
pop_df$femme <- NULL
# Alternative: sélection des variables créées
# Aperçu
pop_df
```

```
## # A tibble: 703 x 2
##    total pop_cat
##    <dbl> <chr>
##  1 12097 >=10000
##  2 12497 >=10000
##  3 14130 >=10000
##  4 23557 >=10000
##  5 10158 >=10000
##  6  3933 <10000
##  7  7866 <10000
##  8  3821 <10000
##  9 20032 >=10000
## 10  6851 <10000
## # ... with 693 more rows
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création de deux nouvelles variables: numérique et catégorielle
  transmute(total = homme + femme,
            pop_cat = ifelse(total < 10000, "<10000", ">=10000"))
```

```
## # A tibble: 703 x 2
##    total pop_cat
##    <dbl> <chr>
##  1 12097 >=10000
##  2 12497 >=10000
##  3 14130 >=10000
##  4 23557 >=10000
##  5 10158 >=10000
##  6  3933 <10000
##  7  7866 <10000
##  8  3821 <10000
##  9 20032 >=10000
## 10  6851 <10000
## # ... with 693 more rows
```

Vous voyez! Comme résultat, nous avons les deux nouvelles colonnes. Toutes les autres ont été omises. Toutefois, le nombre d'observations n'a pas varié.

Les avantages de **dplyr** deviennent évident avec l'augmentation du nombre d'opérations à effectuer sur une intrant donné, très généralement un jeu de données rectangulaires (`data frame` ou `tibble`).

6.5 Sélection d'observations: `filter`

Dans le chapitre précédent, nous avions vu quelques techniques de sélections d'observations. Souvent à partir de la position, souvent à partir de critères définis. Re-appliquons certaines de ces techniques à nos données et comparons à la démarche de **dplyr**.

6.5.1 Sur la base de critères numériques

Regardons à nouveau notre jeu de données.

```
glimpse(adm3_pop_2009)
```

```
## Observations: 703
## Variables: 9
## $ id         <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...
## $ admin0_nom <chr> "Mali", "Mali", "Mali", "Mali", "Mali", "Mali", "Ma...
## $ admin1_nom <chr> "Kayes", "Kayes", "Kayes", "Kayes", "Kayes", "Kayes...
## $ admin2_nom <chr> "Kayes", "Kayes", "Kayes", "Kayes", "Kayes", "Kayes...
## $ admin3_nom <chr> "Bangassi", "Colimbine", "Diamou", "Djelebou", "Fal...
## $ annee      <dbl> 2009, 2009, 2009, 2009, 2009, 2009, 2009, 2009, 200...
## $ homme      <dbl> 6123, 6144, 7115, 11466, 5141, 1999, 3939, 1918, 97...
## $ femme      <dbl> 5974, 6353, 7015, 12091, 5017, 1934, 3927, 1903, 10...
## $ source     <chr> "RGPH", "RGPH", "RGPH", "RGPH", "RGPH", "RGPH", "RG...
```

Nous avons 703 observations, donc 703 communes. Regardons les plus grandes en matière de population. Disons, celles qui sont au dessus du seuil de 100000 habitants.

```
## Avec R-base.
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Critère de sélection: valeurs logiques (TRUE/FALSE)
pop_100000_plus <- pop_df$total > 100000
# Mise en oeuvre de la sélection
pop_df[pop_100000_plus, ]
```

```
## # A tibble: 12 x 4
##    admin3_nom        homme  femme  total
##    <chr>             <dbl>  <dbl>  <dbl>
##  1 Kayes Commune     65135  61184 126319
##  2 Kalabancoro       81018  80864 161882
##  3 Koutiala Commune  70905  70539 141444
##  4 Sikasso Commune  114171 112447 226618
##  5 Segou Commune     66819  66682 133501
##  6 Mopti Commune     60080  60706 120786
##  7 Commune I        168308 166587 334895
##  8 Commune II        78690  80670 159360
##  9 Commune III       63792  64874 128666
## 10 Commune IV       152153 152373 304526
## 11 Commune V        206749 206517 413266
## 12 Commune VI       237951 231702 469653
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
```

```
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(total > 100000)
```

```
## # A tibble: 12 x 4
##    admin3_nom        homme  femme  total
##    <chr>             <dbl>  <dbl>  <dbl>
##  1 Kayes Commune     65135  61184 126319
##  2 Kalabancoro       81018  80864 161882
##  3 Koutiala Commune  70905  70539 141444
##  4 Sikasso Commune  114171 112447 226618
##  5 Segou Commune     66819  66682 133501
##  6 Mopti Commune     60080  60706 120786
##  7 Commune I        168308 166587 334895
##  8 Commune II        78690  80670 159360
##  9 Commune III       63792  64874 128666
## 10 Commune IV       152153 152373 304526
## 11 Commune V        206749 206517 413266
## 12 Commune VI       237951 231702 469653
```

Là aussi, nous voyons que la séquence est plus économe en écriture.

Il arrive souvent aussi que la sélection porte sur un interval ou une région. Dans ce genre de cas, **dplyr** a des fonctions spécialisées comme `between` ou `near`.

Si l'on cherche les communes dont la population est comprise entre 50000 et 60000

```
## Avec R-base.
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Critère de sélection: valeurs logiques (TRUE/FALSE)
pop_50k_60k <- pop_df$total >= 50000 & pop_df$total <= 60000
# Mise en oeuvre de la sélection
pop_df[pop_50k_60k, ]
```

```
## # A tibble: 13 x 4
##    admin3_nom       homme femme total
##    <chr>            <dbl> <dbl> <dbl>
##  1 Massigui         26249 28003 54252
##  2 Baguineda Camp   26406 25415 51821
##  3 Mande            28734 28615 57349
##  4 Ouelessebougou   24694 25345 50039
##  5 Kadiolo          25394 26622 52016
##  6 Koury            26842 27448 54290
##  7 Kolondieba       25976 27404 53380
##  8 Wassoulou Balle  25532 25941 51473
##  9 Bougouni Commune 29581 28957 58538
## 10 Koumantou        24828 26381 51209
## 11 Pelengana        27796 28051 55847
## 12 Tombouctou       27915 26714 54629
## 13 Tonka            25698 27578 53276
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(between(x = total, left = 50000, right = 60000))
```

```
## # A tibble: 13 x 4
##    admin3_nom       homme femme total
##    <chr>            <dbl> <dbl> <dbl>
##  1 Massigui         26249 28003 54252
##  2 Baguineda Camp   26406 25415 51821
##  3 Mande            28734 28615 57349
##  4 Ouelessebougou   24694 25345 50039
##  5 Kadiolo          25394 26622 52016
##  6 Koury            26842 27448 54290
##  7 Kolondieba       25976 27404 53380
##  8 Wassoulou Balle  25532 25941 51473
##  9 Bougouni Commune 29581 28957 58538
## 10 Koumantou        24828 26381 51209
## 11 Pelengana        27796 28051 55847
## 12 Tombouctou       27915 26714 54629
## 13 Tonka            25698 27578 53276
```

Et pour les communes autour de 50000 habitants. Disons que nous prendrons en compte les communes dans les valeurs environnantes et ce jusqu'au 2500 personnes

```
## Avec R-base.
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Critère de sélection: valeurs logiques (TRUE/FALSE)
pop_50k_tol2500 <- pop_df$total >= 50000 - 2500 & pop_df$total <= 50000 + 2500
# Mise en oeuvre de la sélection
pop_df[pop_50k_tol2500, ]
```

```
## # A tibble: 9 x 4
##   admin3_nom      homme femme total
##   <chr>           <dbl> <dbl> <dbl>
## 1 Kita Commune    24054 24989 49043
## 2 Guegneka        23949 24092 48041
## 3 Baguineda Camp  26406 25415 51821
## 4 Ouelessebougou  24694 25345 50039
## 5 Kadiolo         25394 26622 52016
## 6 Misseni         28443 19230 47673
## 7 Wassoulou Balle 25532 25941 51473
## 8 Koumantou       24828 26381 51209
## 9 Sony Aliber     23472 24146 47618
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
```

```
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(near(x = total, y = 50000, tol = 2500))
```

```
## # A tibble: 9 x 4
##   admin3_nom      homme femme total
##   <chr>           <dbl> <dbl> <dbl>
## 1 Kita Commune    24054 24989 49043
## 2 Guegneka        23949 24092 48041
## 3 Baguineda Camp  26406 25415 51821
## 4 Ouelessebougou  24694 25345 50039
## 5 Kadiolo         25394 26622 52016
## 6 Misseni         28443 19230 47673
## 7 Wassoulou Balle 25532 25941 51473
## 8 Koumantou       24828 26381 51209
## 9 Sony Aliber     23472 24146 47618
```

6.5.2 Sur la base de critères catégoriels.

La sélection peut aussi porter sur des variables en caractères ou catégorielles. Supposons que nous souhaitons connaître la population d'une commune dont nous avons le nom: *Bambara Maoudé*.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Critère de sélection: valeurs logiques (TRUE/FALSE)
bambara_maoude <- pop_df$admin3_nom == "Bambara Maoude"
# Mise en oeuvre de la sélection
pop_df[bambara_maoude, ]
```

```
## # A tibble: 1 x 4
##   admin3_nom     homme femme total
##   <chr>          <dbl> <dbl> <dbl>
## 1 Bambara Maoude  8315  8170 16485
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(admin3_nom == "Bambara Maoude")
```

```
## # A tibble: 1 x 4
##   admin3_nom     homme femme total
##   <chr>          <dbl> <dbl> <dbl>
## 1 Bambara Maoude  8315  8170 16485
```

En ce qui concernent les critères alternatifs, **dplyr** offre une commodité avec le signe `%in%`. Imaginez qu'au

lieu d'une seule commune, *Bambara Maoude*, que l'on souhaite sélectionner les résultats d'un groupe de communes dont on a connait le nom. Plutôt que d'utiliser `|`, l'on peut passer par `%in%`.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Critère de sélection: valeurs logiques (TRUE/FALSE)
groupe_commune <- pop_df$admin3_nom == "Bambara Maoude" |
                  pop_df$admin3_nom == "Segou Commune" |
                  pop_df$admin3_nom == "Soumpi" |
                  pop_df$admin3_nom == "Ansongo"
# Mise en oeuvre de la sélection
pop_df[groupe_commune, ]
```

```
## # A tibble: 4 x 4
##   admin3_nom     homme femme  total
##   <chr>          <dbl> <dbl>  <dbl>
## 1 Segou Commune  66819 66682 133501
## 2 Bambara Maoude  8315  8170  16485
## 3 Soumpi          8547  8401  16948
## 4 Ansongo        14707 15384  30091
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(admin3_nom %in% c("Bambara Maoude", "Segou Commune", "Soumpi", "Ansongo"))
```

```
## # A tibble: 4 x 4
##   admin3_nom     homme femme  total
##   <chr>          <dbl> <dbl>  <dbl>
## 1 Segou Commune  66819 66682 133501
## 2 Bambara Maoude  8315  8170  16485
## 3 Soumpi          8547  8401  16948
## 4 Ansongo        14707 15384  30091
```

Moins de lignes! Même résultat!

6.5.3 Sur la base d'expressions régulières

Comme avec `select`, les fonctions relatives aux préfixes et suffixes (et similaires) peuvent être mobilisées dans `filter` aussi. Supposons que l'on veuille connaître la population de toutes les communes dont le nom se termine par *dougou*.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Sélection des communes répondant au critère
select_nom <- endsWith(x = pop_df$admin3_nom, suffix = "dougou")
```

```
# Mise en oeuvre de la sélection
pop_df[select_df, ]
```

```
## Warning: Length of logical index must be 1 or 703, not 9
```

```
## # A tibble: 312 x 4
##    admin3_nom        homme femme  total
##    <chr>             <dbl> <dbl>  <dbl>
##  1 Colimbine          6144  6353  12497
##  2 Diamou             7115  7015  14130
##  3 Djelebou          11466 12091  23557
##  4 Faleme             5141  5017  10158
##  5 Karakoro           7436  7770  15206
##  6 Kayes Commune     65135 61184 126319
##  7 Kemene Tambo       8381  8574  16955
##  8 Khouloum           9734  9260  18994
##  9 Marena Diombougou  8996  9676  18672
## 10 Marintoumania      4001  4060   8061
## # ... with 302 more rows
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(endsWith(x = admin3_nom, suffix = "dougou"))
```

```
## # A tibble: 28 x 4
##    admin3_nom    homme femme total
##    <chr>         <dbl> <dbl> <dbl>
##  1 Gomitradougou  3638  3658  7296
##  2 Dinandougou   10571 10789 21360
##  3 Diedougou     17541 18351 35892
##  4 Dolendougou    6831  7389 14220
##  5 Kaladougou    19429 19742 39171
##  6 Kilidougou     7578  7930 15508
##  7 N'dlondougou  10326 10667 20993
##  8 N'garadougou   7633  7960 15593
##  9 Tenindougou    7681  7725 15406
## 10 Maramandougou  7237  7302 14539
## # ... with 18 more rows
```

La fonction `filter` épouse aussi bien les fonctions spécifiques aux expressions régulières dans R-base - comme dans l'exemple précédent - que celles présentes dans les packages dédiés du **tidyverse**. Les fonctions de **stringr** peuvent s'avérer très commode dans la sélection d'observations. Considérons les observations pour les communes qui ont la lettre z dans leur nom.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Sélection des communes répondant au critère
```

```
detect_z <- grepl(pattern = "z", x = tolower(pop_df$admin3_nom))
# Mise en oeuvre de la sélection
pop_df[detect_z, ]
```

```
## # A tibble: 21 x 4
##    admin3_nom    homme femme total
##    <chr>         <dbl> <dbl> <dbl>
##  1 Zan Coulibaly  9235  9255 18490
##  2 Zegoua        16480 16638 33118
##  3 Zanfigue       7492  7747 15239
##  4 Zangasso       9391 10105 19496
##  5 Zanina         3619  3806  7425
##  6 Zebala         8293  8997 17290
##  7 Zantiebougou  17667 18100 35767
##  8 Kafouziela     3108  3304  6412
##  9 Sanzana        5479  5820 11299
## 10 Zanferebougou  2237  2416  4653
## # ... with 11 more rows
```

```
## Avec dplyr et stringr
# Chargement du package stringr
library(stringr)
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(str_detect(string = tolower(admin3_nom), pattern = "z"))
```

```
## # A tibble: 21 x 4
##    admin3_nom    homme femme total
##    <chr>         <dbl> <dbl> <dbl>
##  1 Zan Coulibaly  9235  9255 18490
##  2 Zegoua        16480 16638 33118
##  3 Zanfigue       7492  7747 15239
##  4 Zangasso       9391 10105 19496
##  5 Zanina         3619  3806  7425
##  6 Zebala         8293  8997 17290
##  7 Zantiebougou  17667 18100 35767
##  8 Kafouziela     3108  3304  6412
##  9 Sanzana        5479  5820 11299
## 10 Zanferebougou  2237  2416  4653
## # ... with 11 more rows
```

6.5.4 Sur la base d'index

Comme nous l'avons vu pour les variables, avec les observations aussi, la sélection peut se faire à partir de l'index.

Chercons à afficher les observations pour les 10 premières observations impaires.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
```

```
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Mise en oeuvre de la sélection
pop_df[seq(from = 1, to = 19, by = 2), ]
```

```
## # A tibble: 10 x 4
##    admin3_nom           homme femme total
##    <chr>                <dbl> <dbl> <dbl>
##  1 Bangassi              6123  5974 12097
##  2 Diamou                7115  7015 14130
##  3 Faleme                5141  5017 10158
##  4 Gory Gopela           3939  3927  7866
##  5 Guidimakan Keri Kaff  9798 10234 20032
##  6 Karakoro              7436  7770 15206
##  7 Kemene Tambo          8381  8574 16955
##  8 Koniakary             4061  4080  8141
##  9 Koussane             10472 11118 21590
## 10 Logo                  6022  5967 11989
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(row_number() %in% seq(from = 1, to = 19, by = 2))
```

```
## # A tibble: 10 x 4
##    admin3_nom           homme femme total
##    <chr>                <dbl> <dbl> <dbl>
##  1 Bangassi              6123  5974 12097
##  2 Diamou                7115  7015 14130
##  3 Faleme                5141  5017 10158
##  4 Gory Gopela           3939  3927  7866
##  5 Guidimakan Keri Kaff  9798 10234 20032
##  6 Karakoro              7436  7770 15206
##  7 Kemene Tambo          8381  8574 16955
##  8 Koniakary             4061  4080  8141
##  9 Koussane             10472 11118 21590
## 10 Logo                  6022  5967 11989
```

6.5.5 Filtres multiples

Dans le chapitre précédent, nous avons qu'en matière de sélections d'observations, il y a plusieurs façons de combiner des critères. On peut :

- les ajouter avec le signe `&`;
- les présenter comme des alternatives avec le signe `|`; ou
- opérer avec une logique de négation avec le signe `!`.

Avec **dplyr**, ces signes demeurent valides.

Supposons que nous voulons connaître les communes de plus de 100000 où il y a plus de femmes que d'hommes.

Ici, les deux critères s'accumulent.

```
## Avec R-base.
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Critère de sélection (1): valeurs logiques (TRUE/FALSE)
pop_100000_plus <- pop_df$total > 100000
# Critère de sélection (1): valeurs logiques (TRUE/FALSE)
femmes_sup_hommes <- pop_df$femme > pop_df$homme
# Mise en oeuvre de la sélection
pop_df[pop_100000_plus & femmes_sup_hommes, ]
```

```
## # A tibble: 4 x 4
##   admin3_nom     homme  femme  total
##   <chr>          <dbl>  <dbl>  <dbl>
## 1 Mopti Commune  60080  60706 120786
## 2 Commune II     78690  80670 159360
## 3 Commune III    63792  64874 128666
## 4 Commune IV    152153 152373 304526
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(
    # Critère 1: population > 100000
    total > 100000,
    # Critère 2: femme > homme
    femme > homme
    )
```

```
## # A tibble: 4 x 4
##   admin3_nom     homme  femme  total
##   <chr>          <dbl>  <dbl>  <dbl>
## 1 Mopti Commune  60080  60706 120786
## 2 Commune II     78690  80670 159360
## 3 Commune III    63792  64874 128666
## 4 Commune IV    152153 152373 304526
```

Ce sont quatre lignes qui répondent à nos critères.

Maintenant regardons ces mêmes critères, mais en termes d'alternatives. Les communes qui ont soit plus de 100000 habitants soit plus de femmes que d'hommes.

```
## Avec R-base.
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Critère de sélection (1): valeurs logiques (TRUE/FALSE)
pop_100000_plus <- pop_df$total > 100000
```

```
# Critère de sélection (1): valeurs logiques (TRUE/FALSE)
femmes_sup_hommes <- pop_df$femme > pop_df$homme
# Mise en oeuvre de la sélection
pop_df[pop_100000_plus | femmes_sup_hommes, ]
```

```
## # A tibble: 538 x 4
##    admin3_nom           homme femme  total
##    <chr>                <dbl> <dbl>  <dbl>
##  1 Colimbine             6144  6353  12497
##  2 Djelebou             11466 12091  23557
##  3 Guidimakan Keri Kaff  9798 10234  20032
##  4 Hawa Dembaya          3406  3445   6851
##  5 Karakoro              7436  7770  15206
##  6 Kayes Commune        65135 61184 126319
##  7 Kemene Tambo          8381  8574  16955
##  8 Koniakary             4061  4080   8141
##  9 Konsiga               2308  2620   4928
## 10 Koussane             10472 11118  21590
## # ... with 528 more rows
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(
    # Critère 1: population > 100000
    total > 100000 |
    # Critère 2: femme > homme
    femme > homme
    )
```

```
## # A tibble: 538 x 4
##    admin3_nom           homme femme  total
##    <chr>                <dbl> <dbl>  <dbl>
##  1 Colimbine             6144  6353  12497
##  2 Djelebou             11466 12091  23557
##  3 Guidimakan Keri Kaff  9798 10234  20032
##  4 Hawa Dembaya          3406  3445   6851
##  5 Karakoro              7436  7770  15206
##  6 Kayes Commune        65135 61184 126319
##  7 Kemene Tambo          8381  8574  16955
##  8 Koniakary             4061  4080   8141
##  9 Konsiga               2308  2620   4928
## 10 Koussane             10472 11118  21590
## # ... with 528 more rows
```

Pendant qu'avec R-base, le nombre d'objets intermédiaires continue d'augmenter, avec **dplyr**, nous nous contentons de l'opérateur `%>%` et de nouvelles lignes.

6.5.6 Tri d'observations: arrange

Souvent, il arrive qu'à la suite d'opération de sélections (aussi bien de variables que de colonnes) que l'on souhaite ordonner les résultats selon un ordre bien précis. Ceci peut servir souvent en matière d'affichage ou même servir de base pour des sélections.

Considérons qu'après avoir filtré pour ne garder que les communes qui ont plus de 100000 habitants et plus de femmes que d'hommes, que nous souhaitions ordonner la population.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin3_nom", "homme", "femme")]
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Critère de sélection (1): valeurs logiques (TRUE/FALSE)
pop_100000_plus <- pop_df$total > 100000
# Critère de sélection (1): valeurs logiques (TRUE/FALSE)
femmes_sup_hommes <- pop_df$femme > pop_df$homme
# Mise en oeuvre de la sélection
pop_df_filtre <- pop_df[pop_100000_plus | femmes_sup_hommes, ]
# Tri: ordre croissant
pop_decroissant <- order(pop_df_filtre$total)
# Mise en oeuvre du tri
pop_df_filtre[pop_decroissant, ]
```

```
## # A tibble: 538 x 4
##    admin3_nom            homme femme total
##    <chr>                 <dbl> <dbl> <dbl>
##  1 Adarmalane              463   492   955
##  2 Bassirou                825   893  1718
##  3 Arham                  1358  1461  2819
##  4 Soignebougou           1547  1552  3099
##  5 Ouro Modi              1566  1762  3328
##  6 Sokourani Missirikoro  1632  1740  3372
##  7 Diou                   1744  1760  3504
##  8 Kende                  1805  1921  3726
##  9 Somo                   1853  1891  3744
## 10 M'bouna                1894  1937  3831
## # ... with 528 more rows
```

```
# Tri: ordre décroissant
pop_decroissant <- order(pop_df_filtre$total, decreasing = TRUE)
# Mise en oeuvre du tri
pop_df_filtre[pop_decroissant, ]
```

```
## # A tibble: 538 x 4
##    admin3_nom        homme  femme  total
##    <chr>             <dbl>  <dbl>  <dbl>
##  1 Commune VI       237951 231702 469653
##  2 Commune V        206749 206517 413266
##  3 Commune I        168308 166587 334895
##  4 Commune IV       152153 152373 304526
##  5 Sikasso Commune  114171 112447 226618
##  6 Kalabancoro       81018  80864 161882
##  7 Commune II        78690  80670 159360
##  8 Koutiala Commune  70905  70539 141444
```

```
##  9 Segou Commune     66819  66682 133501
## 10 Commune III       63792  64874 128666
## # ... with 528 more rows
```

```
## Avec dplyr
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(
    # Critère 1: population > 100000
    total > 100000 |
    # Critère 2: femme > homme
    femme > homme
    ) %>%
  # Tri: ordre décroissant
  arrange(desc(total))
```

```
## # A tibble: 538 x 4
##    admin3_nom        homme  femme  total
##    <chr>             <dbl>  <dbl>  <dbl>
##  1 Commune VI       237951 231702 469653
##  2 Commune V        206749 206517 413266
##  3 Commune I        168308 166587 334895
##  4 Commune IV       152153 152373 304526
##  5 Sikasso Commune  114171 112447 226618
##  6 Kalabancoro       81018  80864 161882
##  7 Commune II        78690  80670 159360
##  8 Koutiala Commune  70905  70539 141444
##  9 Segou Commune     66819  66682 133501
## 10 Commune III       63792  64874 128666
## # ... with 528 more rows
```

L'on peut agrémenter `arrange` avec une fonction qui spécifie le nombre d'observations à afficher ou à sauvegarder: `top_n`. Celle-ci peut opérer par le haut (les valeurs élevées)...

```
  # Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(
    # Critère 1: population > 100000
    total > 100000 |
    # Critère 2: femme > homme
    femme > homme
    ) %>%
  # Tri: décroissant
  arrange(total) %>%
  # Sélection des 5 premières observations
  top_n(n = 5, wt = total)
```

```
## # A tibble: 5 x 4
##   admin3_nom       homme  femme  total
##   <chr>            <dbl>  <dbl>  <dbl>
## 1 Sikasso Commune 114171 112447 226618
## 2 Commune IV      152153 152373 304526
## 3 Commune I       168308 166587 334895
## 4 Commune V       206749 206517 413266
## 5 Commune VI      237951 231702 469653
```

. . . ou par le bas (les valeurs faibles).

```
# Jeu de données de départ
adm3_pop_2009 %>%
  # Sélection des variables d'intérêt
  select(admin3_nom, homme, femme) %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Sélection d'observations d'intérêt
  filter(
    # Critère 1: population > 100000
    total > 100000 |
    # Critère 2: femme > homme
    femme > homme
    ) %>%
  # Tri: décroissant
  arrange(total) %>%
  # Sélection des 5 dernières observations
  top_n(n = -5, wt = total)
```

```
## # A tibble: 5 x 4
##   admin3_nom   homme femme total
##   <chr>        <dbl> <dbl> <dbl>
## 1 Adarmalane     463   492   955
## 2 Bassirou       825   893  1718
## 3 Arham         1358  1461  2819
## 4 Soignebougou  1547  1552  3099
## 5 Ouro Modi     1566  1762  3328
```

6.6 Vers l'agrégation: `group_by` et `summarize`

Jusque là, nous avons opéré en ajoutant ou enlevant des variables, en sélectionnant ou excluant des observations. Ces opérations ont toutes été intra-individuelles c'est-à-dire qu'à aucun moment il n'a été nécéssaire de mélanger les valeurs de deux ou plusieurs observations. Or, il arrive souvant que le *data scientist* ait besoin d'agréer des valeurs pour approfondir sa propre compréhension ou tout simplement synthétiser ses résultats. **dplyr** comptent deux fonctions sont pratiques pour ce faire. Il s'agit de `group_by` et `summarize` (`summarise` aussi marche). Elles viennent consolider les quatre que nous avons vues. La première `group_by` définit les groupes sur lesquels les opérations d'agrégation doivent être exécutées. Quant à `summarize`, elle explicite ces opérations. Illustrons pour mieux comprendre.

Supposons que l'on veuille agréger la population totale par région (`admin1_nom`) et ordonner celle-ci par ordre décroissant. Ceci reviendrait à faire la somme de la population totale en définissant `admin1_nom` comme variable de groupage.

```
## Avec R-base
# Sélection des variables d'intérêt
pop_df <- adm3_pop_2009[ , c("admin1_nom", "homme", "femme")]
```

```
# Création d'une nouvelle variable: numérique
pop_df$total <- pop_df$femme + pop_df$homme
# Opération d'agrégation
pop_df_adm1 <- aggregate(formula = total ~ admin1_nom, data = pop_df, FUN = sum)
# Tri: ordre décroissant
pop_decroissant <- order(pop_df_adm1$total, decreasing = TRUE)
# Mise en oeuvre du tri
pop_df_adm1[pop_decroissant, ]
```

```
##   admin1_nom   total
## 8    Sikasso 2644458
## 5 Koulikouro 2419212
## 7      Segou 2321651
## 6      Mopti 2038855
## 3      Kayes 2013076
## 1     Bamako 1810366
## 9 Tombouctou  671005
## 2        Gao  542304
## 4      Kidal   67739
```

```
## Avec dplyr
# Jeu de données de départ
adm3_pop_2009 %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Spécification du niveau d'agrégation
  group_by(admin1_nom) %>%
  # Opération d'agrégation
  summarize(population = sum(total)) %>%
  # Tri: décroissant
  arrange(desc(population))
```

```
## # A tibble: 9 x 2
##   admin1_nom population
##   <chr>           <dbl>
## 1 Sikasso       2644458
## 2 Koulikouro    2419212
## 3 Segou         2321651
## 4 Mopti         2038855
## 5 Kayes         2013076
## 6 Bamako        1810366
## 7 Tombouctou     671005
## 8 Gao            542304
## 9 Kidal           67739
```

Et si voulions connaitre en même temps le nombre de communes par région? Nous ajoutons une opération supplémentaire à l'intérieur de `summarize`.

```
# Jeu de données de départ
adm3_pop_2009 %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Spécification du niveau d'agrégation
  group_by(admin1_nom) %>%
  # Opération d'agrégation
```

```
  summarize(
    # Population totale par région
    population = sum(total),
    # Nombre de communes par région
    nombre_communes = n()
    ) %>%
  # Tri: décroissant
  arrange(desc(population))
```

```
## # A tibble: 9 x 3
##   admin1_nom population nombre_communes
##   <chr>           <dbl>           <int>
## 1 Sikasso       2644458             147
## 2 Koulikouro    2419212             108
## 3 Segou         2321651             118
## 4 Mopti         2038855             108
## 5 Kayes         2013076             129
## 6 Bamako        1810366               6
## 7 Tombouctou     671005              52
## 8 Gao            542304              24
## 9 Kidal           67739              11
```

Nous pouvons même ajouter la population moyenne par commune.

```
# Jeu de données de départ
adm3_pop_2009 %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Spécification du niveau d'agrégation
  group_by(admin1_nom) %>%
  # Opération d'agrégation
  summarize(
    # Population totale par région
    population = sum(total),
    # Nombre de communes par région
    nombre_communes = n()
    ) %>%
  # Création d'une nouvelle variable %>%
  mutate(pop_par_commune = population / nombre_communes) %>%
  # Tri: décroissant
  arrange(desc(population))
```

```
## # A tibble: 9 x 4
##   admin1_nom population nombre_communes pop_par_commune
##   <chr>           <dbl>           <int>           <dbl>
## 1 Sikasso       2644458             147          17990.
## 2 Koulikouro    2419212             108          22400.
## 3 Segou         2321651             118          19675.
## 4 Mopti         2038855             108          18878.
## 5 Kayes         2013076             129          15605.
## 6 Bamako        1810366               6         301728.
## 7 Tombouctou     671005              52          12904.
## 8 Gao            542304              24          22596
## 9 Kidal           67739              11           6158.
```

`summarize` acceptent la majorité des fonctions statistiques de R-base: `sum` pour la somme, `mean` pour la moyenne, `sd` pour l'écart-type, `min` pour le minimum, `max` pour le maximum, etc.

Il est utile de noter que, bien que les deux fonctions opèrent généralement en paire, elles ne sont pas toutefois obligées d'être ensemble. . . enfin, pas tout le temps.

Considérons par exemple que l'on veut agréger la population totale aussi bien pour les hommes que pour les femmes. Comme le groupe de référence est l'ensemble des observations (toutes les communes), l'on n'a pas besoin de `group_by`.

```
# Jeu de données de départ
adm3_pop_2009 %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Opérations d'agrégation
  summarize(
    # Hommes
    homme = sum(homme),
    # Femmes
    femme = sum(femme),
    # Total
    total = sum(total)
    )
```

```
## # A tibble: 1 x 3
##     homme   femme    total
##     <dbl>   <dbl>    <dbl>
## 1 7204994 7323672 14528666
```

Là, nous avons utilisé `summarize` sans `group_by`. Maintenant, faisons l'inverse. Déterminons la part de chaque commune dans la population régionale. Et gardons seulement celles qui représentent plus de 5% de la population de leur région.

```
# Jeu de données de départ
adm3_pop_2009 %>%
  # Création d'une nouvelle variable
  mutate(total = homme + femme) %>%
  # Spécification du niveau d'agrégation
  group_by(admin1_nom) %>%
  # Création d'une nouvelle variable
  mutate(
    # Population au niveau de admin1_nom
    population_region = sum(total),
    # Ratio population de la commune / population de la région
    part_commune = total / population_region
    ) %>%
  # Filtre: part > 5%
  filter(part_commune > 0.05) %>%
  # Tri des résultats: ordre décroissant
  arrange(desc(part_commune)) %>%
  # Sélection des variables d'intérêt
  select(contains("nom"), part_commune)
```

```
## # A tibble: 28 x 5
## # Groups:   admin1_nom [9]
##    admin0_nom admin1_nom admin2_nom admin3_nom      part_commune
##    <chr>      <chr>      <chr>      <chr>                 <dbl>
```

```
##  1 Mali       Kidal      Kidal      Kidal                0.383
##  2 Mali       Bamako     Bamako     Commune VI           0.259
##  3 Mali       Bamako     Bamako     Commune V            0.228
##  4 Mali       Bamako     Bamako     Commune I            0.185
##  5 Mali       Bamako     Bamako     Commune IV           0.168
##  6 Mali       Gao        Gao        Gao                  0.159
##  7 Mali       Kidal      Tessalit   Adjelhoc             0.117
##  8 Mali       Bamako     Bamako     Commune II           0.0880
##  9 Mali       Gao        Gao        Sony Aliber          0.0878
## 10 Mali       Sikasso    Sikasso    Sikasso Commune      0.0857
## # ... with 18 more rows
```

6.7 Conclusion

Nous venons de voir que **dplyr** est un outil très riche. Avec un vocabulaire simple et accessible, il met à la disposition du *data scientist* une panoplie d'outil qui facilite la manipulation de données.

Dans le prochain chapitre, nous allons explorer un autre package du **tidyverse**: **tidyr**.