01 Introduction à R

Sommaire

Objectifs

Dans ce premier TP, l'objectif principal est de vous faire découvrir les bases du langage R, et l'environnement dans lequel nous allons l'utiliser, qui s'appelle RStudio.

Il s'agira principalement d'apprendre à interagir avec R dans ce qu'on appelle la Console

Vous découvrirez comment R permet de réaliser des calculs simples

Vous découvrirez des objets informatiques qu'on appelle "vecteurs" et "tableaux"

Vous apprendrez à créer, transformer et combiner des vecteurs, comment sélectionner des parties de vecteurs, et comment obtenir les statistiques élémentaires pour les décrire

Et enfin, nous verrons ce que sont les "tableaux", comment les créer, les modifier et les importer à partir de fichiers

Bon travail !

1. Les logiciels

Dans ces TPs, nous utiliserons trois logiciels : R, RStudio et JASP. Voici une rapide présentation de ces trois produits.

1.1. R

R est un logiciel puissant qui fournit un ensemble d'outils intégrés pour la manipulation, l'analyse, et la visualisation de données. C'est un projet "open source" gratuit, libre, ouvert et actif, qui se développe de manière continue grâce à une vaste communauté d'utilisateurs et de contributeurs.

Le logiciel R inclut :

- des outils pour le stockage, la préparation et la manipulation des données
- un ensemble d'opérateurs et de fonctions pour le calcul sur des tableaux, ainsi qu'une collection d'outils d'analyse quantitative et statistique
- un langage de programmation conçu et développé spécialement pour le traitement de données et les statistiques
- une gamme étendue d'outils graphiques puissants et flexibles

Donc R est à la fois un environnement informatique pour faire du calcul et du traitement de données, un logiciel d'analyse et de visualisation et un langage de programmation général.

Un tout petit peu d'histoire

Figure 1.– Robert Gentleman

R est basé sur le langage de programmation S (comme "Statistics") développé dans les années 70 (1976) à Bell Labs par John Chambers. Il a été créé par deux chercheurs de l'université de Auckland en Nouvelle-Zélande, à partir de 1996, Robert Gentleman (*Figure 1*) et Ross Ihaka (*Figure 2*). Robert Gentleman est statisticien et biologiste spécialisé en bioinformatique, et Ross Ihaka est statisticien et informaticien.

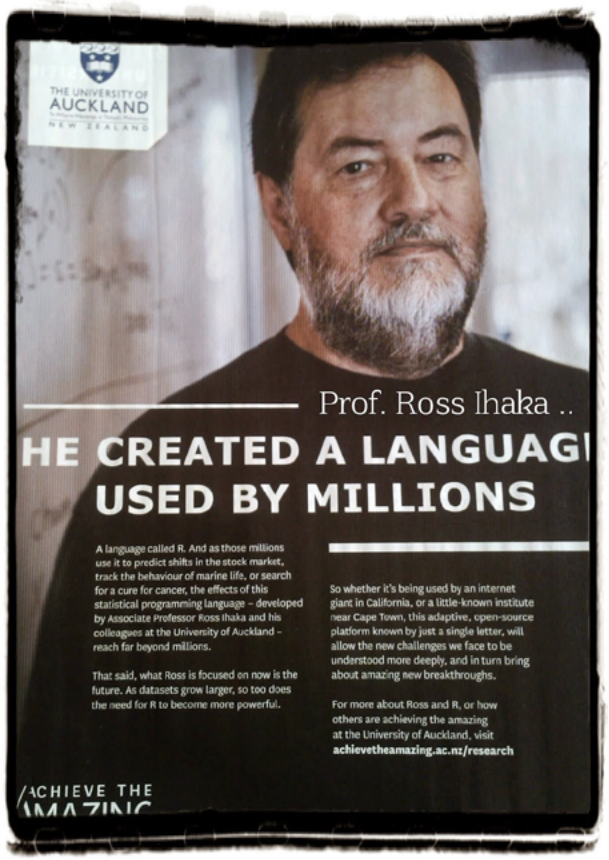


Figure 2.– Ross Ihaka - Extrait d'un article de The Economist, avril 2015; Repris de *http://blog.revolutionanalytics.com/2015/10/ross-ihaka-in-the-economist.html*

R a connu un succès grandissant depuis environ 20 ans (*Figure 3*), tandis que S et Splus n'ont jamais décollé. Il est probablement à l'heure actuelle le logiciel d'analyse statistique le plus utilisé, et le nombre d'utilisateurs continue à croître de manière rapide.

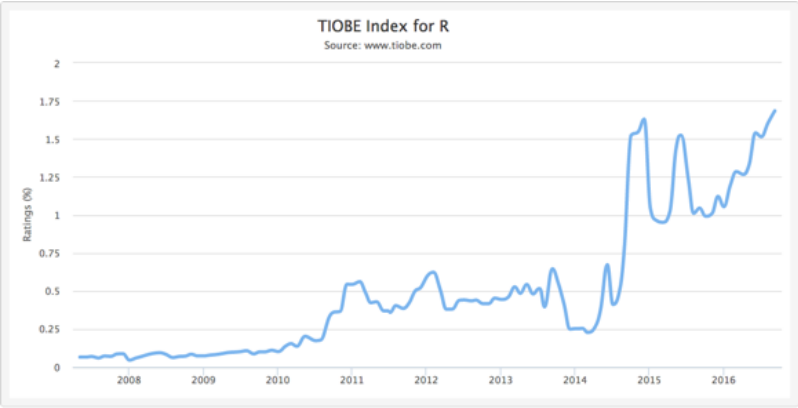


Figure 3.– Croissance du taux d'utilisation de R selon TIOBE

De nombreuses compagnies qui ont besoin d'analyser des volumes importants de données (Ex. Google) utilisent principalement R, et contribuent à son développement. Selon le *TIOBE*, un index de l'utilisation des langages de programmation, R est parmi les 20 logiciels les plus courants, et si on considère les langages orientés math/stat, il est maintenant en tête, devant Matlab, SPSS et SAS.

Le développement du logiciel est actuellement géré par une *fondation à but non lucratif*, qui rassemble des statisticiens et experts informaticiens internationaux.

R est différent…

R est donc un logiciel spécialement conçu pour l'analyse des données et le traitement statistique. C'est un logiciel ouvert et en développement permanent. Il bénéficie de la contribution de très nombreux utilisateurs, qui développent des programmes qu'on appelle librairies ("packages") qui enrichissent les fonctionnalités du logiciel de base.

Une différence importante entre R et les logiciels courants concerne l'interface offerte pour interagir avec le programme. La plupart des logiciels courants actuels utilisent ce qu'on appelle une *interface graphique* pour gérer les interactions avec l'utilisateur. Les fonctions du logiciel sont listées dans des menus et sous-menus, et les spécifications supplémentaires sont gérées par des fenêtres de dialogues avec des listes déroulantes, des boutons, etc.

Ce n'est pas le cas avec R. Les interactions se font la plupart du temps au clavier. C'est ce qu'on appelle une interface par *ligne de commande*. Un avantage de l'interface par ligne de commande est qu'on peut spécifier en une seule ligne une commande complexe (par exemple, *"Calcule une analyse de la variance sur le nombre de réponses correctes, en fonction du sexe, de l'âge et de la condition expérimentale"*), alors que les interfaces graphiques imposent une séquence d'interactions élémentaires (*"Je veux faire une analyse de la variance", "De quel type ?", "À mesures répétées", "Combien de facteurs ?", "Trois", "Quel est le premier facteur ?", ...*).

Un autre avantage est qu'on peut sauvegarder la séquence de commandes dans un fichier, et donc que les analyses sont *reproductibles* : il suffit d'exécuter la séquence de commandes sauvegardée, et on retrouve l'entièreté des analyses. Concrètement, il suffira d'un seul clic de souris pour refaire toute l'analyse !

Dans la recherche, et par exemple dans l'analyse des données d'un mémoire, il n'est pas rare que l'on doive appliquer plusieurs fois la même procédure à des jeux différents de données (parce

qu'on a réalisé plusieurs expériences en variant un paramètre, ou parce qu'on découvre qu'on a oublié un participant, etc.). Avoir sauvegardé le script est donc extrêmement précieux et évite de nombreuses erreurs.

Cela facilite en outre la collaboration. Vos collaborateurs, ou vos promoteurs, pourront très rapidement vérifier que l'analyse est parfaitement correcte, ou vous suggérer telle ou telle modification.

Enfin, comme vous le verrez dans les cours théoriques, les questions liées à la qualité des données, à leur disponibilité dans la communauté scientifique, et à la reproductibilité des analyses (à ne pas confondre avec la reproductibilité des résultats !) ont pris de plus en plus d'importance récemment, et l'utilisation d'un script sauvegardé permet de garantir qu'on exécute *exactement* la même analyse, la même série d'opérations... Par contre l'inconvénient principal de l'interface par ligne de commande est qu'il faut connaître le nom des commandes (et les introduire sans erreur...). L'effort d'apprentissage est donc généralement plus important que dans les logiciels à interface graphique.

Dans cet enseignement, l'objectif est de vous donner une introduction élémentaire à R. Cela doit vous permettre de l'utiliser principalement pour préparer vos fichiers de données en vue des analyses statistiques. Vous apprendrez à écrire des scripts, de petits programmes qui vous permettront d'automatiser les opérations de préparation des données avant les analyses proprement dites. Il est aussi possible de réaliser les analyses statistiques avec R, mais nous utiliserons également le logiciel JASP pour cela.

1.2. RStudio : la boîte à outils

Comme son nom l'indique, RStudio est un "environnement" informatique qui a pour fonction de faciliter les interactions entre l'utilisateur et le logiciel R. Lorsque vous ouvrez RStudio, vous constaterez que la fenêtre est divisée en trois panneaux (ou éventuellement quatre parfois). Nous découvrirons progressivement à quoi servent ces différents panneaux. Pour le moment, nous allons utiliser uniquement la **Console**, qui occupe soit la moitié gauche (voir *Figure 4*) soit le quadrant inférieur gauche de la fenêtre.

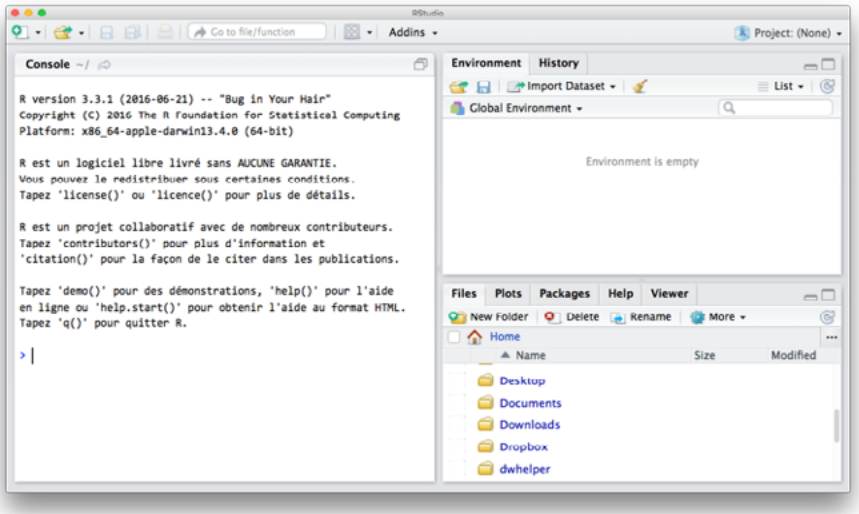


Figure 4.– La fenêtre de RStudio et la Console

La Console permet d'interagir directement avec R. Par la

suite, nous utiliserons RStudio également pour rédiger, mettre au point et corriger des scripts, pour produire les résultats d'analyses, afficher les tableaux de données et les graphiques. Patience, une chose à la fois...

1.3. JASP

Le troisième logiciel que nous allons utiliser est JASP (*Just Another Statistical Package*). JASP est également un logiciel libre, réalisé par une équipe de l'Université d'Amsterdam dirigée par un psychologue, Eric-Jan Wagenmakers. JASP est apparu récemment, en 2015, et est également encore en développement. L'ambition de l'équipe est d'offrir à la communauté scientifique un logiciel comparable à SPSS, qui permette de réaliser la plus grande partie des analyses inférentielles couramment utilisées dans les études comportementales en psychologie.

Par rapport à SPSS, outre l'avantage d'être en accès libre, JASP offre également les outils d'analyse inférentielle bayésienne. Vous verrez que l'interface graphique, assez comparable à SPSS, est plus simple et plus conviviale, et donc très facile à apprendre et à utiliser. Malheureusement à l'heure actuelle JASP n'offre aucune possibilité de manipulation des données (sélection de lignes ou de colonnes, combinaison de variables, etc.). Ces fonctions seront donc réalisées dans R.

Bon à savoir

De nouvelles versions de R apparaissent une ou deux fois par an, et il peut y avoir des changements qui rendent certaines commandes ou librairies incompatibles, si leurs auteurs ne les mettent pas à jour.

Pour éviter toute ambigüité, sachez que nous utiliserons cette année la version 3.3.1 de R qui date de juin 2016 (surnommée «Bug in Your Hair»), et la version 0.99.903 de RStudio, qui date de juillet 2016.

Les ordinateurs de la salle de TP fonctionnent sous Windows 10. Si vous le souhaitez, vous pouvez apporter votre propre ordinateur en TP pour travailler (mais à l'examen, il vous faudra de toute façon travailler sur un des PCs de la salle !)

Testez vos connaissances !

2. Premiers pas : Interagir avec R dans la Console

R est un calculateur. Vous pouvez introduire des expressions mathématiques, et le programme calculera le résultat et l'affichera à l'écran. Le *chevron* au début de la ligne (le signe *plus grand que*, >) qu'on appelle *prompt* (en anglais) ou *invite* (en français), indique que le programme est en attente, prêt à répondre à la commande que l'utilisateur va introduire. On dit parfois comme aux cartes, que c'est *l'utilisateur qui a la main*.

Dans les exemples ci-dessous, et dans tous les eBooks, ce qui apparaît à la droite de l'invite est donc ce que l'utilisateur a introduit au clavier. Si vous voulez reproduire l'exemple, il ne faut pas introduire le > ! Les lignes qui commencent par [1] indiquent la réponse du programme.

Dans la Console, lorsque vous tapez la touche ENTER ou la touche RETURN, R analyse le contenu de la ligne introduite et affiche le résultat de l'évaluation de l'expression (ou éventuellement un message d'explication, si l'expression n'est pas interprétable pour le programme).

```
> 5 + 6
[1] 11
> 3 * 2
[1] 6
> 4654 / 789
[1] 5.898606
> 3*/2
Erreur : '/' inattendu(e) in "3*/"
```

On conseille en général de séparer les opérateurs et les valeurs

par des espaces. Ce n'est pas absolument nécessaire, mais c'est une bonne habitude à prendre d'emblée, pour rendre ce que vous écrivez un peu plus lisible.

Comme toujours, on peut (et c'est une bonne idée) utiliser les parenthèses pour éviter les ambigüités d'interprétation lorsqu'on combine plusieurs opérations. R a des règles de priorité (par exemple, les opérations d'exponentiation comme élever au carré sont faites avant les multiplications et les divisions, et celles-ci sont faites avant les additions et soustractions), mais ces règles sont complexes. Le plus simple est d'adopter le principe d'ajouter des parenthèses chaque fois que vous pensez qu'il pourrait y avoir un doute.

2.1. L'opération d'assignation et l'espace de travail

Imaginons que vous voulez calculer le budget de vos futures vacances, sur base des informations suivantes :

- Une nuitée de logement coûte 65.50€.
- Vous partez 11 jours, soit 10 nuits sur place.
- Pour l'alimentation vous comptez 16€ par jour et par personne (vous êtes 4).
- Le voyage (1250 km aller-retour) vous coûtera 0.75€ du km + 240€ de péages.

Vous pouvez essayer de faire les calculs dans la Console avant de regarder la suite.

```
65.50 * 10
[1] 655
16 * 11 * 4
[1] 704
1250 * 0.75 + 240
[1] 1177.5
655 + 704 + 1177.5
[1] 2536.5
2536.50 / 4
[1] 634.12
```

Si vous avez essayé, vous aurez probablement introduit les différents calculs à peu près comme ci-dessus, et puis recopié les résultats pour faire le total et obtenir le coût par personne.

Mais il y a plus simple et plus efficace... Regardez le bloc de code ci-dessous :

```
001    logement <- 65.50 * 10
002    alim <- 16 * 11 * 4
003    voyage <- 1250 * 0.75 + 240
004    total <- logement + alim + voyage
005    prix.par.personne <- total / 4
006
007    total
008    [1] 2536.5
009    print(prix.par.personne)
010    [1] 634.12
```

Figure 5.– Un exemple d'assignation

Qu'est-ce qui s'est passé ? Les cinq premières lignes ne produisent aucun résultat visible dans la Console. Mais pourtant, R a fait plusieurs choses :

- il a créé un objet en mémoire, qui est désigné par le nom `logement`, et il a associé à ce nom le résultat de `65 * 12`
- puis un autre, `alim`
- puis un troisième, `voyage`
- il a créé un objet `tot` qui est la somme des trois précédents
- et enfin, un objet `prix.par.personne` qui divise le total par le nombre de personnes.

Vous pouvez le voir dans le panneau *Environment* à droite. Au départ, cette zone indiquait que l'espace de travail (appelé ici *Environment*) était vide (cf *Figure 4*). Vous devriez maintenant voir plusieurs éléments affichés dans le panneau *Environment*.

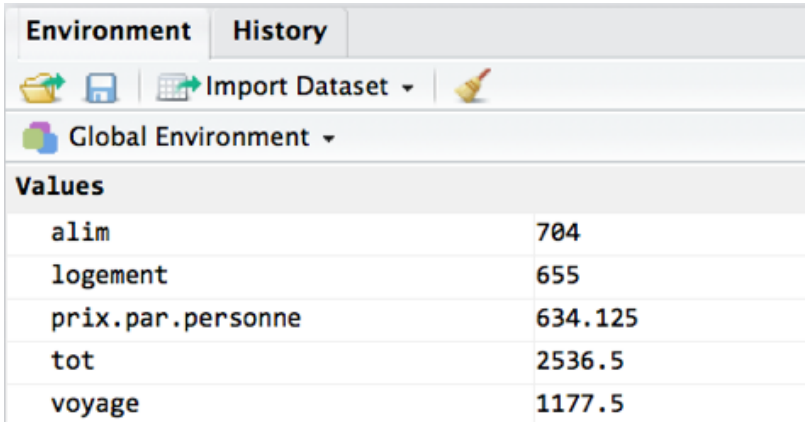


Figure 6.– L'onglet Environment montre le contenu de l'espace de travail. J'ai créé cinq objets, qui sont des variables numériques

Ces cinq premières commandes comportent ce qu'on appelle une **opération d'assignation**. La formule qui permet de commander une assignation a l'une des deux formes générales

nom <- expression
ou
nom = expression

L'opérateur d'assignation `<-` peut être introduit en tapant le signe `<` suivi immédiatement du signe `-`. Il faut faire attention de ne pas séparer les deux signes composant l'opérateur. Mais plus simplement, dans RStudio, il y a un raccourci : `<-` peut être produit en une seule fois, en appuyant ensemble les deux touches ALT et "-" (le tiret).

Il y a souvent des discussions parmi les utilisateurs de R sur la question de savoir s'il faut préférer le signe `=` ou le signe `<-` pour indiquer l'opération d'assignation. La raison principale d'utiliser `<-` est de bien faire la distinction entre l'opération d'assignation et la notion mathématique d'égalité[1].

Si vous avez introduit dans la Console les commandes de la *Figure 5*, vous pouvez faire une expérience intéressante. Marie-Angèle, avec qui vous projetez de partir, trouve que le logement envisagé n'est pas à son goût. Elle propose une autre adresse, mais le prix est de 83.50€ par nuitée. Vous allez donc recalculer le coût du logement : `logement <- 83.50 * 10`. Qu'est-ce qui a changé dans les valeurs ? Qu'est-ce qui n'a pas changé ?

1 Sachez cependant que vous ne gagnerez ni ne perdrez aucun point en utilisant l'un plutôt que l'autre ou un mélange élégant des deux.

La valeur associée à l'objet `logement` devrait avoir changé (dans le panneau *Environment*). Mais ni le total ni le prix par personne n'ont été modifiés. Pour recalculer le total, vous devez ré-exécuter la commande appropriée : `total <- logement + alim + voyage`.

A retenir !

Le résultat d'une commande sans assignation s'affiche mais il est perdu.

On utilise l'assignation chaque fois qu'il est nécessaire de conserver le résultat d'une opération pour des opérations ultérieures.

Si on assigne une nouvelle valeur à un objet qui existe déjà, son ancienne valeur est écrasée et remplacée par la nouvelle valeur.

La modification d'un objet ne modifie pas automatiquement les autres objets

Introduire le nom d'un objet produit l'affichage de sa valeur.

2.2. Qu'est-ce qu'une commande ?

Une commande est une expression qui peut être interprétée par le programme R, et ensuite exécutée.

Par exemple, l'expression `a <- 5 + 6` est une commande. R l'interprète : *Calcule la somme de la valeur numérique 5 et de la valeur numérique 6, stocke ce résultat et assigne-le à un objet désigné par le nom "a", s'il-te-plait* et l'exécute (sans rechigner).

Les éléments qui composent les expressions sont séparés par des espaces (un ou plusieurs), des virgules, ou des parenthèses. Le passage à la ligne (l'utilisateur tape ENTER ou RETURN) déclenche l'action de l'interpréteur. Si l'expression interprétée peut être exécutée, elle est exécutée. Si l'expression est incomplète, l'interpréteur attend une continuation et le signale par une invite différente : le signe `+`.

Donc une commande peut s'étaler sur plusieurs lignes, à condition que les lignes non-finales laissent une expression que l'interpréteur ne peut considérer comme une commande complète.

Il vous arrivera certainement de temps en temps de taper une ligne de code, et de ne recevoir en retour que l'invite `+`. En voici un exemple. Essayez de repérer pourquoi R ne produit pas le résultat du calcul introduit :

```
(logement + voyage + alim / 4
+
```

La réponse est simple : il manque une parenthèse droite, et l'interpréteur considère donc que la "phrase" n'est pas terminée, il attend la suite. Vous pouvez donc compléter la commande sur la ligne suivante en tapant `)`. Mais souvent, le plus simple est d'annuler la commande erronée, en appuyant sur la touche ESCAPE. Vous recevrez alors l'invite normale.

2.3. Comment nommer les variables ?

Bien choisir le nom des variables et des objets est un élément important de l'art de bien programmer : il faut que les noms soient suffisamment longs pour pouvoir être facilement compris lorsque vous (ou quelqu'un d'autre) relit le code, mais il ne faut pas qu'ils soient trop longs non plus.

Cela vaut donc **vraiment** la peine de bien réfléchir à ce qu'une variable désigne pour la nommer de façon appropriée.

Ce n'est généralement pas une bonne idée d'appeler ses variables `v1`, `v2`, `v3`... Outre que cela démontre un criant manque d'imagination, cela n'aide pas à comprendre ce que ces objets représentent.

Deux ou trois choses à retenir :

- L'interpréteur est sensible à la casse (*case-sensitive*): un `a` et un `A` sont deux symboles différents. Vraiment aussi différents que `a` et `z` ! Mettez-vous à sa place : il n'a aucun moyen de savoir que `total` et `Total` correspondent à la même variable. Pour lui, le pauvre, `total` et `Total` sont juste deux chaînes de caractères différentes.
- Les noms des objets peuvent contenir n'importe quelles lettres (y compris accentuées), chiffres, ainsi que "." (point) et "_" (le tiret bas, qu'on appelle aussi "underscore").
- Les noms d'objets doivent toujours commencer par une lettre
- On conseille de se limiter aux caractères A...Z, a...z, 0...9, . et _, parce que les caractères accentués peuvent parfois créer des difficultés entre applications ou entre environnements.
- `TotalParPers`, `Total.par.pers`, `Total_Pers`... sont des noms de variables compréhensibles et bien formés.
- On utilise très couramment `s` pour une chaîne de caractères (*string*), `i` pour un nombre entier (*integer*), `v` pour un vecteur (voir *Section 3 page 18*), `d` pour un tableau de données (*dataframe*, voir *Section 5 page 31*).

Mots magiques réservés !

Certains mots ne peuvent pas être utilisés pour nommer des variables parce qu'ils ont un sens particulier pour le programme. Voici la liste des mots réservés principaux : `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`

2.4. Différents types de variables

R distingue différentes sortes de variables.

Les trois types principaux sont :

- les variables qui prennent des valeurs **numériques**. Le point est utilisé pour séparer la partie entière de la partie décimale, et il n'y a jamais de séparateur pour les milliers, millions etc.
- les variables qui représentent des chaînes de caractères, qu'on appelle parfois variables **alphanumériques**. Elles sont spécifiées par la présence des guillemets (simples ou doubles) avant et après. Attention, tout caractère entre les guillemets fait partie de la suite, on peut avoir des espaces (l'espace est un caractère) avant, après ou au milieu. Donc "`abc`", " `abc`" et "`abc` " sont des chaînes de caractères possibles et différentes les unes des autres. Les variables alphanumériques peuvent contenir des lettres et des chiffres, mais les chiffres ne seront pas interprétés numériquement, et ne peuvent donc pas faire l'objet d'opérations arithmétiques. Par exemple, si on a assigné la valeur "320" à la variable `A`, la commande `A + 1` donnerait une erreur.
- les variables **logiques**, qui n'ont que deux valeurs possibles : `TRUE` , ou `FALSE`. On verra plus loin à quoi ces variables logiques peuvent servir. On peut aussi écrire simplement `T`et `F`.

2.5. Opérateurs

Les opérateurs arithmétiques

Comme tout logiciel de calcul qui se respecte, R est équipé d'une série d'opérateurs arithmétiques. Voici les opérateurs classiques :

- `+` : addition, `3 + 2` donnera `5`
- `-` : soustraction, `3 - 2` donnera `1`
- `*` : multiplication, `3 * 2` donnera `6`
- `/` : division, `3 / 2` donnera `1.5`
- `^` : exponentiation, `3 ^ 2` (exposant 2) donnera `9`

Les opérateurs logiques

R a aussi des opérateurs logiques. Il s'agit d'opérateurs qui permettent d'effectuer des comparaisons. Elles donnent un résultat logique : le test de comparaison est vérifié (`TRUE`) ou non (`FALSE`). Voici la liste des opérateurs principaux :

Les premiers s'appliquent normalement à des valeurs numériques puisqu'ils impliquent une comparaison de magnitude

- `expr1 > expr2` : `TRUE` si le résultat de `expr1` est plus grand que celui de `expr2`
- `expr1 < expr2` : `TRUE` si le résultat de `expr1` est plus petit que celui de `expr2`
- `expr1 >= expr2` : `TRUE` si le résultat de `expr1` est plus grand que ou égal à celui de `expr2`
- `expr1 <= expr2` : `TRUE` si le résultat de `expr1` est plus petit que ou égal à celui de `expr2`

Les deux suivants s'appliquent tant aux variables numériques qu'aux autres :

- `expr1 == expr2` : `TRUE` si le résultat de `expr1` est égal à celui

de `expr2`

- `expr1 != expr2` : `TRUE` si le résultat de `expr1` n'est pas égal à celui de `expr2`

Notez bien la différence entre le signe `=` et le signe `==`

Le premier correspond à l'assignation : l'expression `a = x * 2` signifie : *assigne à l'objet* `a` *le produit de la valeur de* `x` *par 2.*

Le deuxième correspond à un test d'égalité : l'expression `a == x * 2` signifie : *calcule si la valeur de l'objet* `a` *est identique au produit de la valeur de* `x` *par 2*

2.6. Aides à l'écriture dans la Console

Quelques ficelles à retenir d'emblée, parce qu'elles vous feront gagner énormément de temps.

1. La Console connaît plein de choses, et notamment les objets qui existent dans l'espace de travail. Si vous n'avez plus en mémoire le nom complet d'un objet, il suffit d'introduire les deux ou trois premiers caractères, et vous vous verrez apparaitre une liste de propositions de complétions. Vous pouvez utiliser les flèches haut-bas du clavier pour vous déplacer dans cette liste, pour trouver l'objet ou la fonction que vous cherchez, et ENTER ou RETURN pour confirmer votre sélection.

2. La Console a une mémoire de l'histoire de vos commandes. Vous pouvez retrouver les commandes récentes en "remontant" dans le temps. Dans la Console, remonter dans le temps se fait en appuyant sur la touche ↑ du clavier. Vous pouvez ensuite utiliser les flèches gauche et droite pour déplacer le curseur et modifier, compléter ou corriger la commande.

3. Combinaison des deux, si vous introduisez le début d'une commande et ensuite (simultanément) CMD ou CTRL et la flèche vers le haut, vous verrez apparaître une fenêtre flottante avec la cohorte des commandes passées compatibles avec les caractères introduits. Vous pouvez également circuler et choisir, et éventuellement modifier ensuite.

Bon à savoir

Il y a généralement plusieurs façons de résoudre un problème avec R. La solution proposée dans l'eBook n'est sans doute pas la seule, et elle n'est pas nécessairement la meilleure.

Et cela deviendra de plus en plus fréquent, au fur et à mesure que vous avancerez.

Si vous pensez avoir trouvé une autre manière de procéder, tant mieux ! Testez-la, et proposez-la sur le forum pour en faire profitez les autres !

Testez vos connaissances !

3. Vecteurs et scalaires

Qu'est ce qu'un vecteur ?

Un vecteur est simplement une variable qui est constituée d'une série d'éléments, et dont tous les éléments sont du même type. Les vecteurs sont les objets de base de R.

R ne fait pas de différence entre vecteurs et nombres simples (ce qu'on appelle parfois des "scalaires"). Dans R, les nombres simples sont simplement considérés comme des vecteurs qui ne comportent qu'un seul élément.

Comme tous les éléments d'un vecteur doivent être du même type, on distinguera les vecteurs de type numérique (une série de nombres entiers ou décimaux), de type alphanumérique (une série de suites de caractères), et de type logique (une série de valeurs `TRUE` ou `FALSE`).

3.1. Comment créer un vecteur

On verra par la suite différentes façons de créer des vecteurs. L'une des manières les plus courantes est la fonction `combine()`. Elle s'écrit simplement `c(…)` avec entre les parenthèses une liste de valeurs, séparées par des virgules.

Voici un exemple de vecteur numérique :

```
v <- c(4, 5, 7, 2, 3, 1, 6, 8, 9)
v
[1] 4 5 7 2 3 1 6 8 9
```

Voici un exemple de vecteur alphanumérique :

```
prenoms <- c("Pierre", "Jacques", "Paul", "Henri", "Pierre",
"Paul")
prenoms
[1] "Pierre" "Jacques" "Paul"   "Henri"
[5] "Pierre" "Paul"
```

Et un vecteur logique :

```
vraiFaux <- c(T, F, F, F, T, T, T, T)
vraiFaux
[1] TRUE FALSE FALSE FALSE TRUE TRUE TRUE
[8] TRUE
```

Attention à ne pas mélanger des éléments de types différents. R essayerait de trouver le "meilleur compromis" et vous pourriez avoir des surprises...

3.2. Créer une séquence...

... et comment faire pour créer une séquence de valeurs ? Il existe plusieurs façons de faire. Créer une séquence correspond à générer un vecteur numérique. Voici quelques façons de générer des vecteurs numériques :

Avec l'opérateur ":"

L'opérateur : (deux-points) permet de générer une suite numérique, en indiquant son début et sa fin. Le premier chiffre correspond au début de la suite, l'autre à son dernier élément, et les deux-points signifie : énumère la suite des nombres entiers de ... à ...

En fonction des paramètres introduits comme début et fin de séquence, la longueur du vecteur va naturellement changer. Voici un exemple :

```
# Un vecteur de 9 éléments
1:9
[1] 1 2 3 4 5 6 7 8 9
# Et un autre
# On peut construire des séries décroissantes
4:-4
[1] 4 3 2 1 0 -1 -2 -3 -4
```

Bon à savoir

Pourquoi les lignes de R commencent toujours par [1] ?

R est basé sur la manipulation de vecteurs. Le numéro entre crochets indique la position du premier élément affiché sur la ligne dans le vecteur.

Donc si vous affichez un scalaire, ce sera toujour [1].

Mais si vous affichez un vecteur un peu plus long, vous aurez un autre numéro pour la deuxième ligne de l'affichage, par exemple [8], qui veut dire que l'élément qui suit est le huitième élément du vecteur affiché.

Avec la fonction "seq()"

Vous pouvez aussi créer des séquences en utilisant des fonctions. Comme son nom l'indique, la fonction `seq(from = …, to = …, by = …)` vous permet de créer des séquences d'éléments avec plus de souplesse :

```
seq(from = 4, to = 16)
[1] 4 5 6 7 8 9 10 11 12 13 14 15 16
```

Si on spécifie un seul paramètre, comme ci-dessous, il est interprété comme le nombre d'éléments désirés et R crée un vecteur de la longueur indiquée à partir de 1.

```
seq(4)
[1] 1 2 3 4
```

Et on peut également spécifier la valeur de l'incrément ou du décrément avec le paramètre `by …`.

```
seq(from = 4, to = 16, by = 4)
[1] 4 8 12 16
seq(from = 16, to = 8, by = -2)
[1] 16 14 12 10 8
```

Avec la fonction "rep()"

La fonction `rep(x = …, times = …)` (rep comme... répéter) permet de créer des séquences en répétant un élément un certain nombre de fois. Le premier paramètre `x = …` indique ce qu'on veut répéter, et le paramètre `times = …` spécifie combien de fois on doit répéter :

```
rep(4, 3)
[1] 4 4 4
```

Vous pouvez aussi créer des vecteurs de caractères en répétant des lettres, ou des suites de lettres :

```
rep('Condition 1', 3)
[1] "Condition 1" "Condition 1" "Condition 1"
```

Avec la fonction `rep()` on peut répéter non seulement des éléments simples (nombre, chaîne de caractères), mais aussi des vecteurs, puisque un élément simple est considéré par le programme comme un vecteur formé d'un seul élément.

On peut donc combiner `rep()` avec l'opérateur `:`, avec la fonction `seq()`, ou avec la fonction `c()`. Mais du coup, il y a deux possibilités : répéter plusieurs fois *tout le vecteur* (ex. `1 2 3 4 1 2 3 4`) ou répéter plusieurs fois *chaque élément* du vecteur (ex. `1 1 2 2 3 3 4 4`). Les deux sont possibles. Dans le premier cas, on utilise le paramètre `times = …`, tandis que pour répéter chaque élément, on utilisera `each = …`.

```
rep(c(1, 3, 5), times = 4)
[1] 1 3 5 1 3 5 1 3 5 1 3 5
rep(c(1, 3, 5), each = 4)
[1] 1 1 1 1 3 3 3 3 5 5 5 5
```

Par exemple, `rep(seq(from = 1, to = 5, by = 2), times =`

`2)` produira un vecteur formé de deux répétitions de la suite 1, 3, 5. L'expression `rep(seq(from = 1, to = 5, by = 2), each = 2)` produira un vecteur formé de deux répétitions de *chaque élément* de la suite 1, 3, 5[2].

Plus généralement, les arguments d'une fonction peuvent eux-mêmes être des expressions comprenant des fonctions (avec leurs arguments).

3.3. Opérations vectorielles

Une particularité très importante et très utile de R est que beaucoup d'opérations sont *vectorielles*. Cela veut simplement dire que **l'opération s'applique d'un seul coup à l'ensemble des éléments du vecteur**.

Par exemple, vous pouvez additionner un nombre à un vecteur ou multiplier un vecteur par un nombre :

```
v <- c(1, 5, 10, 15, 20)
v
[1] 1 5 10 15 20
v + 2
[1] 3 7 12 17 22
v * 5
[1]  5 25 50 75 100
```

On voit que chaque élément du vecteur est modifié. Vous pouvez également soustraire un nombre d'un vecteur ou diviser un vecteur par un nombre :

```
v <- c(5, 10, 15, 20)
v
[1] 5 10 15 20
v - 2
[1] 3 8 13 18
v / 5
[1] 1 2 3 4
```

On peut également faire des opérations arithmétiques avec deux ou plusieurs vecteurs. Encore une fois, les opérations se font élément par élément -- cela suppose donc nécessairement que les vecteurs ont la même longueur, le même nombre d'éléments !

```
v1 <- c(1, 2, 3, 4, 5, 6)
v2 <- c(2, 4, 6, 8, 1, 1)
v1 + v2
[1] 3 6 9 12 6 7
v1 - v2
[1] -1 -2 -3 -4 4 5
v1 * v2
[1] 2 8 18 32 5 6
v1 / v2
[1] 0.5 0.5 0.5 0.5 5.0 6.0
```

2 Essayez (évidemment) !

3.4. Combinaison de vecteurs

Comment faire pour combiner ensemble deux ou plusieurs vecteurs ? En utilisant la fonction qui sert à combiner : `c(…)`, puisqu'il n'y a pas de différence entre vecteurs et scalaires.

```
v1 <- 1:3
v2 <- 11:13
v3 <- c(v1, v2)
v3
[1] 1 2 3 11 12 13
```

3.5. Accès aux éléments d'un vecteur

Comme on vient de le voir, l'avantage énorme des opérations vectorielles est qu'elles s'appliquent d'un seul coup à toutes les données, à tous les éléments du vecteur. Par exemple, vous voulez transformer des scores bruts en pourcentages, sommer les scores à plusieurs épreuves pour avoir un score global, calculer la différence entre deux conditions...ce sera chaque fois une commande en une ligne, qui s'appliquera d'office à tous les éléments.

Mais il arrive aussi qu'on ait besoin d'isoler un élément, ou un sous-ensemble d'éléments, pour les traiter séparément. Il y a deux manières principales de procéder : sur base de la position des éléments dans le vecteur, ou par une liste explicite sous la forme d'un vecteur logique.

Accès par l'index de position

L'idée de base est simple : on indique entre crochets le numéro d'ordre des éléments à extraire.

```
v <- c('a', 'b', 'c', 'd')
v[3]
[1] "c"
```

Bien sûr, l'expression entre crochets est elle-même un vecteur. On peut donc extraire plusieurs éléments en indiquant une liste de numéros d'ordre entre les crochets. Voici quelques illustrations. Le vecteur `LETTERS` est formé de toutes les lettres majuscules.

```
LETTERS[1:5]
[1] "A" "B" "C" "D" "E"
LETTERS[c(1, 5, 9, 15, 21)]
[1] "A" "E" "I" "O" "U"
LETTERS[seq(1,26, by = 4)]
[1] "A" "E" "I" "M" "Q" "U" "Y"
```

L'expression `LETTERS[1:5]` signifie *dis-moi (s'il-te-plaît) quelle sont les valeurs des éléments 1 à 5 du vecteur LETTERS*. On peut bien sûr assigner le résultat à un nouvel objet, si on souhaite utiliser ce sous-ensemble pour d'autres opérations.

Les crochets ne servent pas uniquement à sélectionner une partie d'un vecteur pour l'afficher ou l'assigner à un autre vecteur. Point très important, ils permettent également de *modifier sélectivement certains éléments d'un vecteur*. En voici un exemple. Je veux créer un vecteur `CATEGORY` de 26 éléments, qui indique pour

chaque lettre s'il s'agit d'une consonne ou d'une voyelle. Comme il y a beaucoup plus de consonnes que de voyelles, je commence par créer une série de 26 "C".

```
CATEGORY <- rep("C", 26)
```

Et je vais ensuite modifier sélectivement les éléments de la série qui correspondent aux voyelles : A est la première lettre, E la cinquième, etc.

```
CATEGORY[ c(1, 5, 9, 15, 21) ] <- "V"
```

L'expression entre les crochets est un vecteur, qui énumère les positions des éléments que je veux modifier en "V".

Sélection d'éléments par un vecteur logique

Les crochets peuvent aussi être utilisés d'une autre manière, en indiquant par un *vecteur logique* quels sont les éléments à extraire (ou à modifier).

```
v <- LETTERS[1:5]
v[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
[1] "A" "C" "E"
```

Ici, on a créé un vecteur, `v`, qui reprend les cinq premières lettres de l'alphabet. Dans la commande suivante on spécifie explicitement quels éléments on veut extraire (`TRUE`) et quels éléments on ne veut pas (`FALSE`). Le vecteur entre les crochets `c(TRUE, FALSE, TRUE, FALSE, TRUE)` indique à R de sélectionner la première lettre (`TRUE`), pas la deuxième (`FALSE`), la troisième (`TRUE`), et ainsi de suite.

Bien sûr, il n'est pas très pratique de spécifier une longue liste explicite de `TRUE, FALSE…` ou même `T, F,…`. Mais cette manière de procéder est très utile lorsqu'on la combine avec les opérations logiques (voir *Section 2.5 page 15*). En voici une illustration. Le vecteur `CATEGORY` que j'ai créé précédemment indique pour chaque lettre si elle est Consonne ou Voyelle.

```
LETTERS[1:6] ; CATEGORY[1:6]
[1] "A" "B" "C" "D" "E" "F"
[1] "V" "C" "C" "C" "V" "C"
```

Voici donc une manière rapide de sélectionner les voyelles :

```
LETTERS[CATEGORY == "V"]
[1] "A" "E" "I" "O" "U"
```

La logique de ce type de commande n'est pas simple à saisir. Pour bien comprendre, voyez ce que produit la commande qui correspond à l'expression à l'intérieur des crochets, `CATEGORY == "V"`. Vous verrez que c'est un vecteur, formé de 26 éléments, qui est le résultat de l'application du test `== "V"` successivement à chaque élément du vecteur `CATEGORY` (Rappelez-vous que les opérations et les comparaisons sont *vectorielles*, c'est-à-dire qu'elles s'appliquent automatiquement à chaque élément des vecteurs).

Voici un autre exemple, qui utilise les fonctions de génération de nombres aléatoires qui sont décrites dans la section qui suit. Il peut arriver qu'on doive sélectionner une partie des données récoltées, par exemple, pour équilibrer les effectifs entre diffé-

rentes conditions. Le vecteur `Id` reprend les numéros d'ordre des 10 participants, de 1 à 10. Le vecteur `Score` pourrait correspondre au résultat obtenu par ces 10 participants dans un certain test. Pour créer `Score`, j'ai imaginé que sa distribution était normale, autour d'une moyenne de 50, et avec un écart-type de 10. Pour la facilité, j'ai ensuite arrondi les valeurs obtenus à l'unité supérieure avec la fonction `round()`.

```
Id <- 1:12
Score <- rnorm(n = 12, mean = 50, sd = 10)
Score <- round(Score)
Id ; Score
[1] 1 2 3 4 5 6 7 8 9 10 11 12
[1] 44 52 42 66 53 42 55 57 56 47 65 54
```

Comment sélectionner au hasard 6 participants ? Nous allons d'abord créer une liste (un vecteur) qui mélange les nombres de 1 à 10. Nous utiliserons ensuite cette liste pour décider quelles observations sélectionner. Nous prendrons les observations qui correspondent aux nombres 1 à 6 dans la liste mélangée.

```
Selection <- sample (12)
Selection
[1] 4 5 1 12 7 3 8 6 2 10 9 11
Selection <= 6
[1] TRUE TRUE TRUE FALSE FALSE TRUE FALSE
[8] TRUE TRUE FALSE FALSE FALSE
Id.5p <- Id[Selection <= 6]
Score.5p <- Score[Selection <= 6]
Id.5p ; Score.5p
[1] 1 2 3 6 8 9
[1] 44 52 42 42 57 56
```

3.6. Créer des vecteurs aléatoires

Comme on vient de le voir, il est utile de pouvoir créer des vecteurs de valeurs tirées au hasard. Nous allons examiner deux manières de faire cela.

Mélanger une liste : `sample()`

Dans sa version la plus simple la fonction `sample(x =…)` permet de mélanger les éléments d'un vecteur :

```
sample(1:8)
[1] 7 1 5 3 4 2 6 8
sample (c("a", "b", "c", "d", "e"))
[1] "c" "d" "a" "e" "b"
```

On peut également utiliser `sample()` pour créer un échantillon, par tirage aléatoire. Il suffit de spécifier le paramètre `size = …` qui détermine la taille de l'échantillon. Cela correspond à un tirage aléatoire sans remplacement.

```
# tirer au hasard cinq nombres entre 1 et 1000
sample(1 : 1000, size = 5)
[1] 693 478 860 437 244
# tirer quatre lettres majuscules au hasard parmi tout
l'alphabet
sample(LETTERS, size = 4)
```

```
[1] "B" "C" "H" "L"
```

Enfin, `sample()` permet de tirer un échantillon *avec remplacement*. Il faut alors spécifier le paramètre `replace = TRUE`, pour que les valeurs puissent apparaître plus d'une fois dans l'échantillon sélectionné

```
sample(LETTERS[1:4], size = 10, replace = T)
[1] "C" "B" "D" "B" "B" "B" "C" "B" "B" "D"
```

Créer une série de valeurs selon une distribution définie

L'autre technique pour créer des séries aléatoires est basée sur les distributions de probabilités. R contient un éventail de fonctions statistiques relatives aux différentes lois de probabilité connues et étudiées (Binomiale, Normale, Chi-carré, Student, F, et beaucoup d'autres moins familières). Comme nous le verrons plus tard, dans la 2ème partie des TPs, ces fonctions permettent notamment de calculer les *p-valeurs* (et donc de remplacer avec plus de précision l'utilisation des anciennes tables statistiques). Mais elles offrent également la possibilité de créer un échantillon de valeurs tirée d'une population de distribution connue.

Il peut être utile de savoir que les noms de ces fonctions ont tous la même structure : `r` (comme random) + le nom de la distribution (`unif`, pour la distribution uniforme, `norm`, pour la distribution normale, etc.). Nous en verrons deux exemples, `runif()` et `rnorm()`.

La fonction `runif(n = …, min = …, max = …)` permet donc de créer une série de `n` valeurs tirées aléatoirement d'une distribution uniforme dans un intervalle entre `min` et `max`. Rappelons qu'une distribution uniforme est une distribution continue dont toutes les valeurs ont la même probabilité. Par exemple, la commande ci-dessous produit 10 valeurs, tirées au hasard parmi tous les nombres possibles entre 0 et 2.

```
runif(n=10, min=0, max=2)
[1] 0.16849 1.75064 0.67815 1.67888 0.69337
[6] 0.66755 0.95270 1.78440 1.72868 0.77998
```

Vous pourriez vous demander quelle est la différence entre `sample()` et `runif()` : effectivement, `sample (0:2, size = 10, replace = T)` donnerait également 10 valeurs comprises entre 0 et 2. Mais `sample()` échantillonne parmi un ensemble de valeurs particulières distinctes, les nombres entiers, tandis que `runif()` échantillonne parmi tous les nombres possibles dans l'intervalle continu entre `min` et `max`. Voici par exemple ce qu'on obtient en générant 10 valeurs entre 0 et 5 avec les deux fonctions :

```
sample(0:5, size=10, replace = T)
[1] 4 5 2 4 2 1 4 1 4 0
runif(min=0, max=5, n=10)
[1] 1.22744 0.71652 1.19815 0.29467 3.21144
[6] 4.38135 3.89457 3.98654 2.27637 2.05042
```

La fonction `rnorm(n = …, mean = …, sd = …)` permet de créer une série de valeurs tirées au hasard selon une distribution normale (gaussienne) de moyenne et d'écart-type déterminés. Les valeurs par défaut correspondent à la distribution normale standard réduite (moyenne = 0, écart-type = 1).

```
rnorm(10)
[1]  0.88111  0.39811 -0.61203  0.34112 -1.12936
[6]  1.43302  1.98040 -0.36722 -1.04413  0.56972
```

Testez vos connaissances !

4. Statistiques descriptives élémentaires

Imaginez que vous avez fait passer un certain test à 100 personnes, et que vous souhaitez en faire une première analyse exploratoire parce que vous avez le lendemain un rendez-vous avec votre promoteur. Vous pourriez introduire les 100 résultats sous la forme d'un vecteur. Nous allons maintenant découvrir comment obtenir les statistiques élémentaires à propos de vecteurs et comment les visualiser. Nous verrons dans la section suivante (*Section 5 page 31*) de manière plus complète comment on organise des données et comment importer dans R des données récoltées via d'autres outils, par exemple PsychoPy.

Pour le moment, avec un brin d'imagination, supposons que nous disposons des performances des 100 participants. Je vais à nouveau les générer au hasard.

```
perf <- rnorm(n=100, mean=10, sd=3)
```

4.1. Visualisation des données

On peut très facilement obtenir un histogramme montrant la distribution des valeurs d'un vecteur avec la fonction `hist()`. Cela permet de vérifier si les observations sont rassemblées autour d'une valeur centrale, et si la distribution semble symétrique.

```
hist(perf)
```

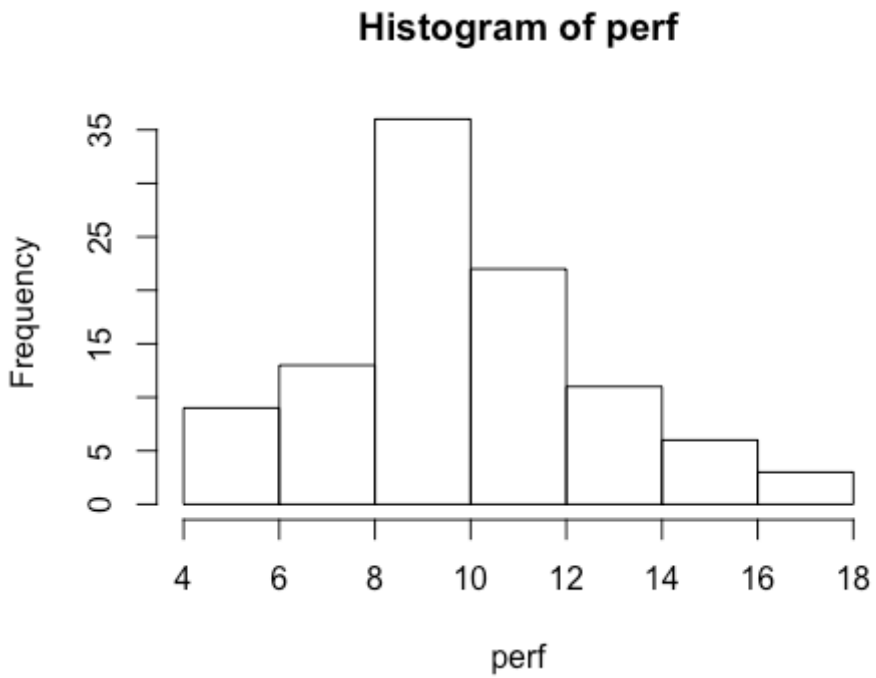


Figure 7.– Exemple d'histogramme

Le paramètre `breaks = …` détermine le nombre de classes et peut être utile pour obtenir une représentation moins grossière de la distribution

```
hist(perf, breaks = 20)
```

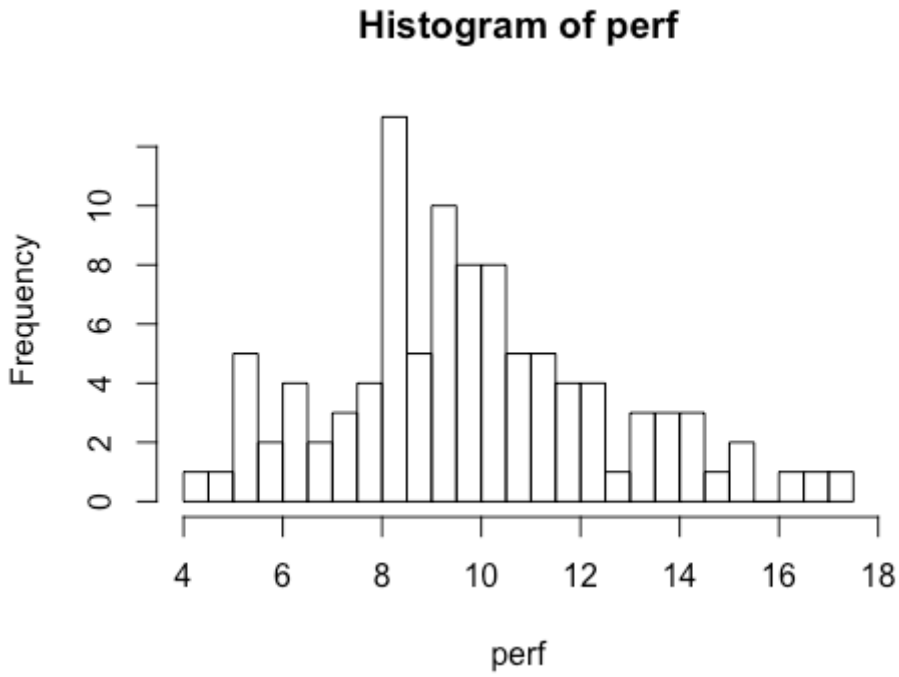


Figure 8.– Un histogramme plus détaillé

4.2. Les statistiques descriptives de base

Fonctions générales

(1) La fonction `length(v)` renvoie la longueur du vecteur `v`, c'est-à-dire le nombre d'éléments du vecteur.

```
length(perf)
[1] 100
```

Lorsque les vecteurs sont composés de valeurs numériques, et que leur distribution indique une tendance centrale, on peut calculer plusieurs descripteurs de cette tendance et de la dispersion des valeurs autour de la tendance centrale.

(2) La fonction `sum(v)` renvoie la somme des valeurs des éléments du vecteur `v`.

```
sum(perf)
[1] 976.86
```

Lorsque les vecteurs sont composés de valeurs numériques, et que leur distribution indique une tendance centrale, on peut calculer plusieurs descripteurs de cette tendance et de la dispersion des valeurs autour de la tendance centrale.

Tendance centrale : moyenne et médiane

(3) La fonction `mean(v)` calcule la valeur de la moyenne des éléments de `v`.

```
mean(perf)
[1] 9.7686
```

on pourrait évidemment faire autrement…

```
sum(perf)/length(perf)
[1] 9.7686
```

(4) La fonction `median(v)` calcule la valeur de la médiane des éléments de `v`.

```
median(perf)
[1] 9.5312
```

Certains d'entre vous se souviendront qu'on mentionne parfois un troisième indicateur de tendance centrale, le *mode*. Le mode est la valeur qui a l'effectif le plus élevé. Il faudrait donc d'abord calculer les effectifs de toutes les valeurs rencontrées dans le vecteur. Dans le cas des échantillons provenant de distributions normales ou uniformes, cela n'a pas de sens puisque toutes les valeurs sont uniques, à moins de constituer des classes.

(5) Lorsque le vecteur est *discret*, c'est-à-dire, constitué d'un nombre limité de valeurs possibles, la fonction `table()` permet d'obtenir les effectifs pour chaque valeur

Dispersion : le minimum, le maximum, l'écart-type, et la variance

(6) La fonction `min(v)` renvoie la plus petite valeur parmi les éléments du vecteur `v`. La fonction `max(v)` renvoie la plus grande valeur.

```
min(perf)
[1] 4.2569
max(perf)
[1] 17.205
```

(7) La fonction `var()` -- fournit la *variance* de la distribution des valeurs.

```
var(perf)
[1] 7.755
```

(8) La fonction `sd()` -- *standard deviation* fournit l'écart-type de la distribution des valeurs. L'écart-type est la racine carrée de la variance.

```
sd(perf)
[1] 2.7848
```

Distribution : les quantiles

(9) La fonction `quantile()` permet d'obtenir l'estimation de certains quantiles de la distribution du vecteur.

Par défaut, si on indique comme seul argument le nom du vecteur de données, R affiche le minimum (c'est-à-dire le quantile à 0%), le maximum (le quantile à 100%), ainsi que les valeurs qui correspondent au premier, deuxième et troisième quartile.

```
quantile(perf)
  0%    25%    50%    75%   100%
4.2569 8.0816 9.5312 11.4405 17.2049
```

On peut spécifier les quantiles souhaités en indiquant une ou plusieurs valeurs pour le paramètre `probs =...`. Notez que les quantiles souhaités doivent être indiqués en nombres décimaux et pas en pourcentages.

```
quantile(perf, probs = seq(0, 1, by = 0.10))
  0%    10%    20%    30%    40%    50%
4.2569 6.1673 7.8502 8.2661 8.9449 9.5312
  60%    70%    80%    90%   100%
10.0649 10.9434 11.8796 13.5432 17.2049
```

(10) Une autre fonction qui donne plusieurs indications résumées sur les caractéristiques d'un vecteur est `summary()`.

```
summary(perf)
 Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 4.26  8.08  9.53  9.77  11.40  17.20
```

Testez vos connaissances !

5. Tableaux de données

5.1. Notion de base

La plupart du temps, les "vraies" données comprennent évidemment plusieurs variables :

- Vous réalisez une enquête et vous avez une série de variables signalétiques (l'âge, le sexe, l'état civil, la nationalité, etc.), et une série de variables indiquant les réponses à chacun des éléments du questionnaire
- Vous conduisez une expérience comportementale, et vous devez coder les variables signalétiques propres à chaque participant, les conditions qu'il ou elle a rencontrées dans l'expérience, et les caractéristiques mesurées (temps de réponse, nature de la réponse, durée, etc.)
- Vous avez mené une étude avec des patients et vous souhaitez mettre en relation certaines caractéristiques de leur diagnostic neurologique avec certains aspects de leurs performances

Dans tous ces exemples, les données prennent naturellement la forme d'un tableau. Nous verrons, au fur et à mesure, qu'il y a différentes façons d'organiser les tableaux de données. Mais cette section a pour objectif d'introduire ce qu'on appelle dans R des *data frames*, "tableaux de données", que je traduirai dorénavant plus simplement par "tableaux"[3].

3 Sachez cependant qu'il existe une autre sorte d'objets, qu'on appelle ***array***, ce qu'on traduirait aussi par "tableau". Mais nous ne rencontrerons probablement jamais des ***arrays*** dans le cadre de cette introduction.

Un *data frame* est une liste de vecteurs qui ont tous la même longueur.

Les vecteurs sont les colonnes du tableau. Il faut donc nécessairement qu'ils aient la même longueur pour que les éléments qui se trouvent sur une même ligne correspondent à la même observation.

Pour rendre les choses un peu plus concrètes, reprenons l'exemple utilisé supra, *Section page 23*. Nous avions dix participants, identifiés dans la variable `Id` et leur score, dans le vecteur `Score`.

```
Id
[1] 1 2 3 4 5 6 7 8 9 10 11 12
Score
[1] 44 52 42 66 53 42 55 57 56 47 65 54
```

Nous allons combiner ces deux vecteurs pour construire un nouvel objet, le *data frame* `d`.

```
d <- data.frame(Id, Score)
```

Nous avons assigné au nouvel objet `d` le résultat de la fonction `data.frame()`. Celle-ci construit un tableau qui combine les vecteurs qu'elle reçoit comme arguments, ici, `Id` et `Score`.

Si vous exécutez cette commande, vous devriez voir apparaître dans la zone *Environment*, en plus des vecteurs `Id` et `Score`, un nouvel objet `d` sous l'étiquette *Data*. L'information affichée indique que `d` comporte 10 *observations* et est composé de deux *variables*.

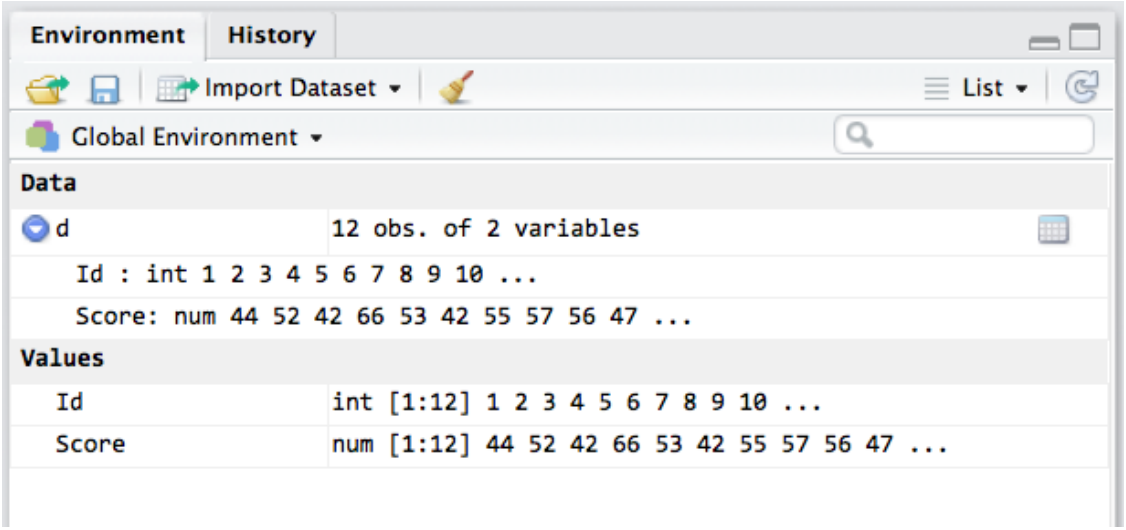


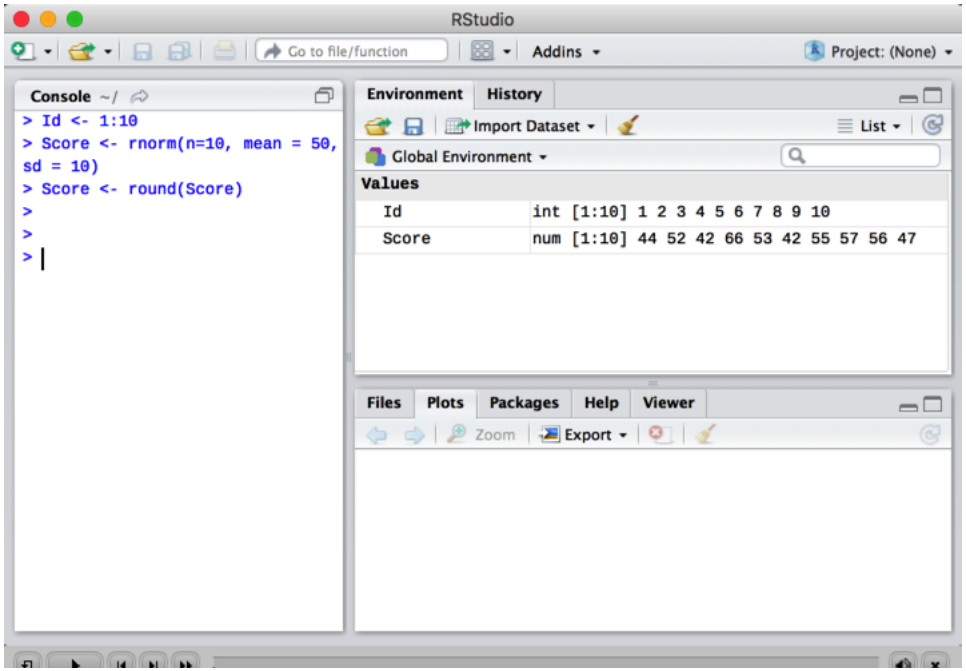
Figure 9.– Création d'un data frame

Si vous cliquez sur le petit triangle à gauche du nom de l'objet (voir *Figure 9*), vous verrez apparaître les vecteurs (variables) qui composent le tableau, `Id` et `Score`, avec l'indication de leur type (`Id` est `int`, c'est-à-dire *integer*, nombre entier, tandis que Score est `num`, *numeric*. La distinction est subtile et n'a pas d'importance pour ce qui nous concerne), et les valeurs des premières observations.

Vous pouvez voir que les valeurs des vecteurs (colonnes) de `d` correspondent à celles des objets-vecteurs, ceux qui figurent sous l'étiquette *Values*. La commande `data.frame()` a réalisé des copies des objets-vecteurs d'origine pour faire les colonnes du tableau `d`. Cela signifie que si vous modifiez ensuite les valeurs des *objets-vecteurs*, cela n'aura pas d'effet sur les *vecteurs-colonnes* et vice-versa.

Comme illustré dans la *Vidéo 1*, vous pouvez aussi cliquer sur l'icone à l'extrême droite sur la ligne correspondant à l'objet `d`

dans la zone d'Environnement. Cela fera apparaître un nouvel onglet dans le panneau gauche de la fenêtre, dans lequel vous pouvez voir le contenu du *dataframe* sous la forme d'un tableau.



Vidéo 1.– Visualisation d'un data frame

Dans la version actuelle de R-Studio, vous n'avez pas la possibilité de modifier les cellules de ce tableau dans le tableau. Par contre, si vous faites des modifications via la Console, elles se reflèteront automatiquement dans la visualisation du tableau.

Nous allons donc maintenant examiner comment on peut accéder aux colonnes, aux lignes et aux cellules. Pour la facilité, nous allons opérer avec le tout petit tableau `d` qui comporte 12 lignes et 2 colonnes, mais si vous avez un peu d'imagination, vous pouvez rêver que le tableau comporte 60 variables et des milliers de lignes...

5.2. Accès aux colonnes

Le plus souvent, on accèdera aux colonnes d'un tableau sur la base de leur nom, en utilisant l'opérateur spécial **`$`**.

> **L'expression `tbl$col` réfère au vecteur-colonne `col` du tableau `tbl`**

On peut utiliser cette règle pour visualiser le contenu de toute une colonne, mais également pour le modifier. Voyons ce qui se produit si on modifie la colonne `Id`, par exemple.

```
d$Id <- 1001:1012
```

Vous pourrez constater par vous-même dans le panneau Environnement et dans l'onglet de visualisation que les valeurs de la variable Id *du tableau* ***`d`*** a été modifiée, mais que par contre le vecteur `Id` n'a pas été affecté et a conservé ses valeurs antérieures. L'objet *vecteur* ***`Id`*** et l'objet *colonne* ***`Id`*** *du tableau* ***`d`*** sont des choses différentes sans aucun lien informatique ! Il n'y a d'ailleurs aucune obligation à ce que ces deux choses portent le même nom.

Il en est de même si vous modifiez le vecteur `Score` :

```
# calculons le score standardisé
Score <- (Score - mean(Score)) / sd(Score)
```

Cette fois c'est l'inverse : `Score` est modifié, mais `d$Score` n'a

pas été affecté.

Autre chose d'utile : on peut utiliser la même règle pour faire référence à une colonne qui n'existe pas et lui assigner des valeurs. Il faut évidemment, encore une fois, que le nombre d'éléments de la nouvelle colonne corresponde au nombre de lignes du tableau.

```
d$ScoreStd <- Score
# et une autre façon de faire la même chose :
d$ScoreStd1 <- (d$Score - mean (d$Score))/sd( d$Score)
```

R permet également de traiter les tableaux en utilisant le principe des crochets, mais cette fois, avec les deux dimensions, lignes et colonnes. Cela permet d'accéder aux lignes, et aux cellules des tableaux.

L'expression tbl[vlignes, vcols] réfère à la partie du tableau `tbl` formée des lignes correspondant au vecteur `vlignes` et des colonnes correspondant au vecteur `vcols`

5.3. Accès aux lignes

```
# Les trois premières lignes de d
d[1:3, ]
  Id Score ScoreStd ScoreStd1
1 1001   44 -1.095112 -1.095112
2 1002   52 -0.093867 -0.093867
3 1003   42 -1.345423 -1.345423
```

```
# La ligne correspondant au participant 1007
d[d$Id==1007, ]
  Id Score ScoreStd ScoreStd1
7 1007   55   0.2816   0.2816
```

A noter !

Un tableau est un objet à deux dimensions. Il faut donc toujours que l'expression de sélection comporte deux séries de valeurs, séparées par une virgule.

On peut laisser vide l'une ou l'autre de ces spécifications, pour indiquer que toutes les valeurs doivent être retenues.

Ainsi `tbl[v, ]` refère au tableau comprenant toutes les lignes spécifiées par `v` dans leur entièreté.

Et `tbl[ , v]` est une autre manière de référer au tableau comportant un sous-ensemble des colonnes de `tbl`.

5.4. Importer des tableaux à partir d'un fichier

Nous verrons dans le deuxième chapitre d'autres outils pour manipuler des tableaux de données. Un dernier point important est qu'on dispose de fonctions pour importer des tableaux de données à partir de fichiers et inversement, pour exporter un tableau sous la forme d'un fichier qui pourra ensuite être utili-

sé dans d'autres logiciels – JASP, Excel ou SPSS pour prendre quelques exemples.

La fonction qui permet d'importer un fichier est `read.table()`. Elle demande

- de spécifier le chemin d'accès au fichier,
- de déclarer quel est le caractère qui sert de séparateur entre les colonnes dans le fichier.
- et d'indiquer par le paramètre `header = …` si le fichier comporte ou non une première ligne avec les noms des colonnes

```
read.table(file = "tableau.txt", sep = "\t", header=T)
   Id Score ScoreStd ScoreStd1
1 1001    44 -1.095112 -1.095112
2 1002    52 -0.093867 -0.093867
3 1003    42 -1.345423 -1.345423
4 1004    66 1.658312 1.658312
5 1005  <NA> 0.031289 0.031289
6 1006    42 -1.345423 -1.345423
7 1007    55 0.281600 0.281600
8 1008     . 0.531912 0.531912
9 1009    56 0.406756 0.406756
10 1010   47 -0.719645 -0.719645
11 1011   65 1.533157 1.533157
12 1012   54 0.156445 0.156445
```

Comme pour la plupart des fonctions que nous avons rencontrées, `read.table()` importe le contenu du fichier spécifié, l'affiche dans la Console, et le résultat est perdu. Il faut donc ici également assigner le résultat de la lecture à un objet

```
d1 <- read.table(file = "tableau.txt", sep = "\t", header=T)
```

Pour que la lecture soit possible, il faut évidemment que le fichier existe, et qu'il se trouve à l'endroit de votre disque dur où R va chercher. Le plus simple pour vous assurer que R regardera dans le bon répertoire est de vérifier le *répertoire de travail (Working Directory)*. Une des manières de procéder est d'utiliser l'entrée *Set Working Directory* du menu *Session* (voir *Figure 10*)

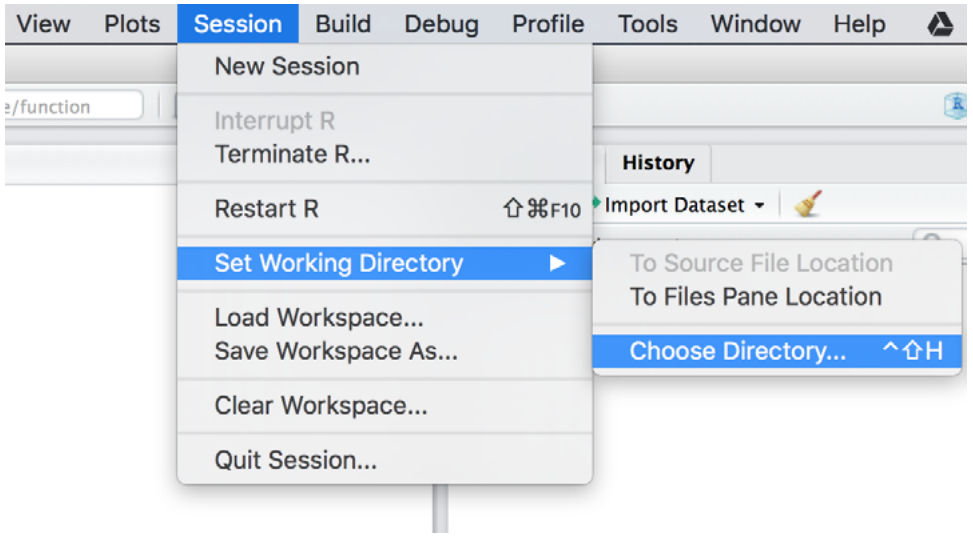


Figure 10.– Comment définir le répertoire de travail

Testez vos connaissances !

6. Prêt.e pour le TP ?

Ma Checklist

- ☐ J'ai repéré la Console dans RStudio et j'ai compris comment introduire des commandes
- ☐ J'ai compris la différence entre commandes avec et sans assignation
- ☐ J'ai créé différentes variables et je les vois dans la zone d'environnement
- ☐ Je peux naviguer dans la ligne de commande pour la corriger et la modifier
- ☐ J'ai compris la distinction entre variables numériques, alphanumériques et logiques
- ☐ Je peux créer des vecteurs de différentes manières
- ☐ J'ai compris le principe des opérations vectorielles
- ☐ Je vois comment on peut extraire un sous-ensemble d'éléments d'un vecteur
- ☐ Je vois comment on peut modifier un sous-ensemble d'éléments d'un vecteur
- ☐ Je peux visualiser la distribution d'une série de mesures
- ☐ Je peux obtenir les stats descriptives élémentaires d'une série de mesures
- ☐ J'ai compris le principe des tableaux de données
- ☐ Je peux accéder aux vecteurs-colonnes d'un tableau
- ☐ Je peux isoler certaines lignes d'un tableau sur base d'un test de comparaison
- ☐ Je vois comment importer un fichier de données sous la forme d'un tableau