Concepts de base en R

Sophie Baillargeon, Université Laval

2018-01-10

Table des matières

R = une puissante calculatrice.

R est un langage interactif. La console de R est un peu comme une calculatrice.

```
1 + 1
```

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

```
2*5 + 1
```

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

Comment garder une copie en mémoire d'un résultat ?

Il faut **assigner** le résultat à un **nom** :

```
a <- 1 + 1
```

- `<-` est l'opérateur d'assignation,
- `a` est le nom (ou le symbole) référant à l'objet créé en mémoire,
- `1 + 1` est l'expression générant le résultat à enregistrer,
- `2` est la valeur contenue dans l'objet créé.

Notons que `=` est aussi un opérateur d'assignation. Cet opérateur est utilisé en R pour assigner une valeur à un argument lors de l'appel d'une fonction. Cependant, pour l'assignation d'une valeur à un nom d'objet, la pratique la plus recommandée en programmation R est d'utiliser `<-` plutôt que `=`. J'expliquerai plus tard la différence entre `<-` et `=`, qui est trop technique pour un premier cours.

Quels noms sont acceptés en R ?

Un nom d'objet R doit respecter les règles suivantes[1].

- Le nom d'un objet peut uniquement contenir des lettres, des chiffres, des points ou des barres de soulignement.
- Il commence obligatoirement par une lettre ou un point et s'il commence par un point, le deuxième caractère ne doit pas être un chiffre.
- Il ne doit pas être un des mots suivants, qui sont réservés par R[2] : `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, `NA_character_`, `...` et `..1`, `..2`, etc.

Remarques :

- Les lettres accentuées sont acceptées, mais il vaut mieux les éviter, car leur transfert d'un système d'exploitation à un autre est souvent problématique.
- Il n'y a aucune restriction sur la longueur des noms.

Comment accéder à un résultat gardé en mémoire dans un objet ?

Dans la console, il suffit de taper le nom référant à l'objet :

```
a
```

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

mais pas :

```
"a"
```

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

car les guillemets servent à créer des chaînes de caractères.

De plus, il faut **respecter la case** car R différencie les majuscules et les minuscules. Donc `A` n'est pas le même objet que `a` en R.

Soumettre une commande R contenant uniquement le nom d'un objet est équivalent à soumettre un appel à la fonction `print` en fournissant cet objet comme premier argument. Ainsi, les commandes suivantes sont équivalentes :

[1] https://stat.ethz.ch/R-manual/R-devel/library/base/html/make.names.html
[2] https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html

```
a
```

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

```
print(a)
```

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

Est-ce le seul objet accessible ?

La commande `ls()` (équivalente à `objects()`) retourne la liste des noms des objets, dans l'**environnement de travail** (parfois appelé environnement courant ou environnement global). Il s'agit donc de la liste des noms des objets créés depuis le démarrage de la session R.

```
ls()
```

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

Ces noms sont ici entre guillemets, car ils sont affichés par `ls` en tant que chaînes de caractères.

Mais est-ce vraiment le seul objet accessible ?

Non. En plus des objets que vous avez créés, vous avez accès aux objets venant avec l'installation de R. Ils sont simplement rangés ailleurs que dans l'environnement de travail. Voici la liste de tous les *groupes* d'objets auxquels vous avez accès directement.

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

Ces groupes d'objets portent le nom d'**environnements**. Il s'agit pour la plupart d'environnements de packages. Le premier environnement de cette liste, nommé `".GlobalEnv"`, est votre environnement de travail. La fonction `ls` affiche par défaut le contenu de cet environnement, mais elle peut afficher le contenu de n'importe quel environnement grâce à l'argument `name`.

```
ls(name = "package:stats")
```

```
##  [1] "acf"                  "acf2AR"               "add.scope"
##  [4] "add1"                 "addmargins"           "aggregate"
##  [7] "aggregate.data.frame" "aggregate.ts"         "AIC"
## [10] "alias"                "anova"                "ansari.test"
## [13] "aov"                  "approx"               "approxfun"
##  [ reached getOption("max.print") -- omitted 432 entries ]
```

Quelle est la nature de tous ces objets ?

Il s'agit principalement de fonctions et de jeux de données. Par exemple, dans le package `base`, il y a la fonction suivante :

```
casefold
```

```
## function (x, upper = FALSE)
## if (upper) toupper(x) else tolower(x)
## <bytecode: 0x00000000194bd530>
```

```
## <environment: namespace:base>
```

Cette fonction sert à convertir la casse de caractères alphabétiques dans une ou plusieurs chaînes de caractères, réunies dans un vecteur. La casse d'un caractère alphabétique, soit d'une lettre, est sa forme majuscule (capitale) ou minuscule. En anglais, les majuscules sont appelées *upper case characters* et les minuscules *lower case characters*.

Aussi, dans le package `datasets`, il y a le jeu de données suivant :

```
women
```

```
##    height weight
## 1      58    115
## 2      59    117
## 3      60    120
## 4      61    123
## 5      62    126
## 6      63    129
## 7      64    132
## 8      65    135
## 9      66    139
## 10     67    142
## 11     68    146
## 12     69    150
## 13     70    154
## 14     71    159
## 15     72    164
```

Il s'agit de valeurs moyennes de grandeur et de poids, calculées en 1975, pour des femmes américaines âgées entre 30 et 39 ans.

Truc :

Utilisez la fonction `str` pour avoir un aperçu d'un objet plutôt que de l'afficher en entier.

```
str(casefold)
```

```
## function (x, upper = FALSE)
```

```
str(women)
```

```
## 'data.frame':    15 obs. of  2 variables:
##  $ height: num  58 59 60 61 62 63 64 65 66 67 ...
##  $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
```

Qu'est-ce qu'une fonction ?

Une fonction est un bout de code qui produit un certain résultat lorsqu'exécuté. Une fonction prend des valeurs en entrée, nommées *arguments*. Lorsqu'elle est appelée, la fonction traite les valeurs fournies en entrée dans l'appel. Ensuite, elle retourne un objet en sortie ou encore produit un ou des résultats, comme un graphique ou l'écriture dans un fichier externe.

Un objet R de type fonction est composé des éléments suivants.

- Une liste d'arguments, avec leurs valeurs par défaut s'il y a lieu. Exemple :

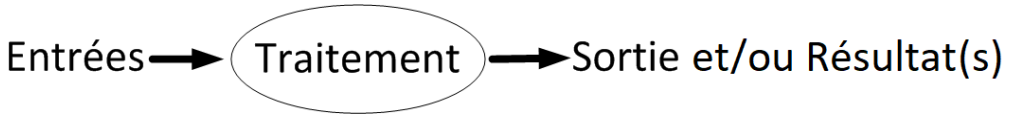


FIGURE 1 – Représentation schématique de l'exécution d'un appel à une fonction

```
args(casefold)
```

```
## function (x, upper = FALSE)
## NULL
```

La fonction `casefold` possède deux arguments : `x` et `upper`. L'argument `x` n'a pas de valeur par défaut, mais `upper` a `FALSE` comme valeur par défaut.

- Le corps de la fonction, soit le code composant la fonction. Exemple :

```
body(casefold)
```

```
## if (upper) toupper(x) else tolower(x)
```

Le corps de la fonction `casefold` est particulièrement court. Le plus souvent, le corps d'une fonction R se compose de plus d'une instruction.

Pour utiliser une fonction, il faut l'appeler en lui fournissant, au besoin, des valeurs aux arguments. Exemple :

```
casefold(x = c("Bonjour", "à", "tous!"), upper = TRUE)
```

```
## [1] "BONJOUR" "À"       "TOUS!"
```

Comment écrire un appel de fonction ?

Un appel de fonction est une instruction formée des éléments suivants, en respectant cet ordre :

1. le nom de la fonction à appeler ;
2. une parenthèse ouvrante `(` ;
3. au besoin, des valeurs pour les arguments, spécifiées ainsi :
   a) nom de l'argument, suivi de l'opérateur `=` (éléments non obligatoires),
   b) commande générant la valeur à fournir en argument (par exemple simplement le nom d'un objet),
   c) si plus d'une valeur d'argument est à donner en entrée, il faut les séparer par une virgule ;
4. une parenthèse fermante `)`.

Dans l'exemple précédent, c'est-à-dire :

```
casefold(x = c("Bonjour", "à", "tous!"), upper = TRUE)
```

deux valeurs d'arguments sont fournies, en spécifiant les noms des arguments. Le résultat suivant est obtenu :

```
## [1] "BONJOUR" "À"       "TOUS!"
```

Ici, la valeur par défaut de l'argument `upper` est `FALSE`. Pour utiliser cette valeur, il n'est pas nécessaire de fournir de valeur à l'argument dans l'appel de fonction.

```
casefold(x = c("Bonjour", "à", "tous!"))
```

```
## [1] "bonjour" "à"       "tous!"
```

Cependant, l'argument `x` n'a pas de valeur par défaut et ne lui fournir aucune valeur génère une erreur.

```
casefold()
```

```
## Error in tolower(x): argument "x" is missing, with no default
```

Comment sont interprétées les valeurs d'arguments non assignées à un nom ?

Si une valeur d'argument est passée en entrée sans être spécifiquement assignée à un nom, la valeur est assignée au premier argument non spécifié, en respectant l'ordre des arguments dans la définition de la fonction. Voici quelques exemples et les résultats obtenus.

```
casefold(c("Bonjour", "à", "tous!"))
```

```
## [1] "bonjour" "à"       "tous!"
```

Ici, la valeur `c("Bonjour", "à", "tous!")` n'est pas assignée explicitement à un nom par l'utilisateur. R l'assigne implicitement au premier argument, `x`.

```
casefold(c("Bonjour", "à", "tous!"), TRUE)
```

```
## [1] "BONJOUR" "À"       "TOUS!"
```

Ici, les deux valeurs fournies en entrée ne sont pas assignées explicitement à des noms par l'utilisateur. Elles sont donc assignées implicitement en respectant l'ordre des arguments : la première valeur fournie, soit `"Bonjour", "à", "tous!"` est assignée au premier argument, soit `x` ; la deuxième valeur fournie, soit `TRUE` est assignée au deuxième argument, soit `upper`.

```
casefold(FALSE, c("Bonjour", "à", "tous!"))
```

```
## Warning in if (upper) toupper(x) else tolower(x): the condition has length
## > 1 and only the first element will be used
```

```
## Error in if (upper) toupper(x) else tolower(x): argument is not interpretable as logical
```

Ici, l'ordre dans lequel les valeurs sont fournies implique que `FALSE` est assigné à `x` et `c("Bonjour", "à", "tous!")` est assigné à `upper`. Cela génère un avertissement et une erreur, car les arguments reçoivent alors des valeurs qui ne sont pas des types attendus.

```
casefold(FALSE, x = c("Bonjour", "à", "tous!"))
```

```
## [1] "bonjour" "à"       "tous!"
```

Si la valeur `c("Bonjour", "à", "tous!")` est explicitement assignée à l'argument `x`, la valeur `FALSE` est implicitement assignée à `upper`, même si elle est fournie en première position, car les assignations explicites ont préséance sur les assignations implicites.

Truc :

Pour éviter les erreurs dans un appel de fonction, il vaut mieux toujours assigner les valeurs passées en entrée à des noms d'arguments. Par contre, dans la pratique, lorsque le premier argument d'une fonction R est un objet contenant les données sur lesquelles appliquer le calcul (ce qui est fréquent en R), il est habituel de ne pas le nommer. Ainsi, dans l'exemple initial, l'appel le plus usuel est le suivant :

```r
casefold(c("Bonjour", "à", "tous!"), upper = TRUE)
```

L'argument ... et ses deux utilités.

Certaines fonctions R possèdent un argument nommé « ... ». C'est le cas par exemple de la fonction `paste`, servant à convertir des valeurs en chaînes de caractères et à concaténer ces valeurs.

```r
args(paste)
```

```
## function (..., sep = " ", collapse = NULL)
## NULL
```

C'est aussi le cas de la fonction `apply`, servant à appliquer un calcul sur des sous-sections d'un objet.

```r
args(apply)
```

```
## function (X, MARGIN, FUN, ...)
## NULL
```

L'argument ... n'a pas la même utilisé dans ces deux fonctions. Je vais utiliser ces deux exemples de fonction pour illustrer les deux utilités potentielles de ....

Utilité 1 : recevoir un nombre indéterminé d'arguments

La première utilité de ... est de permettre à une fonction de recevoir en entrée un nombre indéterminé d'arguments. Si ... est placé au tout début de la liste des arguments, il a typiquement cette utilité. C'est le cas pour la fonction `paste`. Cette fonction peut concaténer les valeurs provenant d'autant d'objets que désiré grâce à l'argument .... Voici des exemples.

```r
paste("a", 1:4, sep = "-")
```

```
## [1] "a-1" "a-2" "a-3" "a-4"
```

```r
paste("a", 1:4, c(TRUE, FALSE), collapse = " ")
```

```
## [1] "a 1 TRUE a 2 FALSE a 3 TRUE a 4 FALSE"
```

Les arguments « attrapés » par ... peuvent être assignés à des noms, tant que ceux-ci diffèrent des noms des autres arguments de la fonction. Dans un appel à la fonction `paste`, nommer les arguments attrapés par ... n'a aucun impact.

```r
paste(lettre = "a", chiffre = 1:4, sep = "-")
```

```
## [1] "a-1" "a-2" "a-3" "a-4"
```

Cependant, avec d'autres fonctions, de telles assignations ont pour effet de nommer des éléments dans le résultat. Voici un exemple avec la fonction `list`.

```r
list("a", 1:4)
```

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

```r
list(lettre = "a", chiffre = 1:4)
```

```
## $lettre
## [1] "a"
##
## $chiffre
## [1] 1 2 3 4
```

Utilité 2 : passer des arguments à une autre fonction

La deuxième utilité de l'argument `...` est de passer des arguments à une autre fonction appelée dans le corps de la première fonction. Si `...` est placé à la fin de la liste des arguments, il a typiquement cette utilité.

Considérons par exemple la fonction `apply`. Supposons que cette fonction soit utilisée pour calculer la moyenne des valeurs par colonne dans le jeu de données `women`.

```r
apply(X = women, MARGIN = 2, FUN = mean)
```

```
##   height   weight
##  65.0000 136.7333
```

Cet appel à la fonction `apply` provoque des appels à la fonction fournie comme argument `FUN`, ici la fonction `mean`. Pour demander le calcul de moyennes tronquées via l'argument `trim` de la fonction `mean`, il suffit d'ajouter cet argument, et sa valeur désirée, dans l'appel à la fonction `apply`.

```r
apply(X = women, MARGIN = 2, FUN = mean, trim = 0.3)
```

```
##   height   weight
##  65.0000 135.5714
```

Ici, tous les arguments portant des noms autres que `X`, `MARGIN` et `FUN` (les trois arguments nommés dans la liste des arguments de `apply`) ont été inclus automatiquement dans les sous-appels à la fonction `mean`.

Il est possible de passer à la sous-fonction autant d'arguments que désiré. Pour éviter toute erreur, il vaut mieux toujours assigner des noms aux valeurs passées en argument à une sous-fonction via `...`.

Les opérateurs sont aussi des fonctions.

Tous les opérateurs sont aussi des fonctions. Par exemple, l'opérateur mathématique d'addition :

```r
1 + 1
```

est en fait un raccourci pour la fonction nommée `"+"` :

```r
"+"(1, 1)
```

Comment utiliser une fonction d'ajustement de modèle ?

Étant donné que R est un langage conçu pour effectuer des calculs statistiques et créer des graphiques, il sert fréquemment à ajuster des modèles statistiques. R possède une classe d'objets particulière dédiée à la spécification de modèles : les **formules**. Comprendre comment utiliser une fonction d'ajustement de modèle en R revient principalement à comprendre comment correctement spécifier une formule R.

Par exemple, la fonction `lm` sert à ajuster un modèle de régression. Le premier argument de cette fonction est une formule. Voici un exemple d'ajustement d'une régression linéaire simple sur les données `women`, en utilisant le poids comme variable réponse et la grandeur comme variable explicative.

```
reg <- lm(weight ~ height, data = women)
```

L'objet `weight ~ height` est une formule.

```
str(weight ~ height)
```

```
## Class 'formula'  language weight ~ height
##   ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

La partie de gauche de la formule (à gaude du `~`) contient la, ou plus rarement les, noms des variables réponse. La partie de droite contient pour sa part le ou les noms des variables explicatives. Ces noms sont accompagnés, au besoin, d'opérateurs et de fonction pour désigner comment former les termes du modèle incluant les variables. Par exemple, voici la formule à utiliser dans l'appel à la fonction `lm` pour effectuer une régression polynomiale de degré deux avec les données `women`.

```
reg_poly <- lm(weight ~ height + I(height^2), data = women)
```

Je n'irai pas plus loin dans la présentation des formules R ici. L'écriture de formule est un sujet qui sera couvert dans un prochain cours.

Comment accéder aux résultats de la fonction d'ajustement de modèle ?

Un objet obtenu en sortie d'une fonction d'ajustement de modèle contient beaucoup d'informations. Beaucoup plus qu'il ne le laisse paraître à première vue.

Par exemple, affichons dans la console l'objet `reg` créé dans le premier exemple d'ajustement de modèle.

```
reg
```

```
##
## Call:
## lm(formula = weight ~ height, data = women)
##
## Coefficients:
## (Intercept)       height
##      -87.52         3.45
```

Seul un court extrait des résultats contenus dans `reg` est imprimé. Un résumé plus complet des résultats est obtenu avec la fonction `summary`.

```
summary(reg)
```

```
##
## Call:
## lm(formula = weight ~ height, data = women)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.7333 -1.1333 -0.3833  0.7417  3.1167
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) -87.51667    5.93694  -14.74 1.71e-09 ***
## height        3.45000    0.09114   37.85 1.09e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.525 on 13 degrees of freedom
## Multiple R-squared:  0.991,  Adjusted R-squared:  0.9903
## F-statistic:  1433 on 1 and 13 DF,  p-value: 1.091e-14
```

Cependant, en réalité, l'objet `reg` contient plus d'éléments que ce qui est affiché ci-dessus.

L'objet `reg` est une liste de sous-objets contenant les résultats de l'ajustement du modèle :

```
str(reg, max.level = 1, give.attr = FALSE)
```

```
## List of 12
##  $ coefficients : Named num [1:2] -87.52 3.45
##  $ residuals    : Named num [1:15] 2.4167 0.9667 0.5167 0.0667 -0.3833 ...
##  $ effects      : Named num [1:15] -529.566 57.73 -0.058 -0.486 -0.914 ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:15] 113 116 119 123 126 ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##  $ df.residual  : int 13
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = weight ~ height, data = women)
##  $ terms        :Classes 'terms', 'formula'  language weight ~ height
##  $ model        :'data.frame':   15 obs. of  2 variables:
```

Un élément de la liste `reg` peut être extrait avec l'opérateur `$` comme suit :

```
reg$coefficients
```

```
## (Intercept)      height
##   -87.51667     3.45000
```

Cependant, il existe des fonctions servant spécifiquement à extraire des éléments d'une sortie de modèle. Par exemple, la fonction `coefficients` extrait les coefficients du modèle :

```
coefficients(reg)
```

```
## (Intercept)      height
##   -87.51667     3.45000
```

La fonction `summary`, en plus d'afficher un résumé des résultats tel que vu précédemment, retourne en sortie une liste contenant quelques résultats complémentaires à ceux retournés par la fonction d'ajustement du modèle :

```
regSummary <- summary(reg)
str(regSummary, max.level = 1, give.attr = FALSE)
```

```
## List of 11
##  $ call         : language lm(formula = weight ~ height, data = women)
##  $ terms        :Classes 'terms', 'formula'  language weight ~ height
##  $ residuals    : Named num [1:15] 2.4167 0.9667 0.5167 0.0667 -0.3833 ...
##  $ coefficients : num [1:2, 1:4] -87.5167 3.45 5.9369 0.0911 -14.741 ...
##  $ aliased      : Named logi [1:2] FALSE FALSE
```

```
##  $ sigma        : num 1.53
##  $ df           : int [1:3] 2 13 2
##  $ r.squared    : num 0.991
##  $ adj.r.squared: num 0.99
##  $ fstatistic   : Named num [1:3] 1433 1 13
##  $ cov.unscaled : num [1:2, 1:2] 15.15595 -0.23214 -0.23214 0.00357
```

Qu'est-ce qu'un package ?

Un package R est un **regroupement de fonctions et/ou de données documentées**. En fait, lors de l'ouverture d'une session R, en supposant que l'environnement de travail est vide, tous les objets accessibles proviennent d'un package.

L'installation de base de R est composée d'un certain nombre de packages R. Un très grand nombre de packages supplémentaires sont aussi disponibles sur le web. Les packages sont la façon formelle de partager publiquement des fonctions R.

La majorité des développeurs de packages R les rendent disponibles sur le **CRAN** (*Comprehensive R Archive Network*) : http://www.r-project.org/ > « CRAN » dans le menu de gauche.

Le site web **Bioconductor** (http://www.bioconductor.org/) propose aussi beaucoup de packages R spécifiques au domaine de la bio-informatique.

Finalement, certains développeurs distribuent simplement leurs packages R sur leur site web personnel ou sur un site web de gestion de développement de logiciels tel que **GitHub** (https://github.com/), **Bitbucket** (https://bitbucket.org/) ou **R-Forge** (https://r-forge.r-project.org/)

Comment utiliser un package ?

Pour utiliser un package R, il faut premièrement qu'il soit installé sur notre ordinateur. Installer R provoque l'installation de certains packages. Les packages supplémentaires doivent être installés séparément. La fiche sur l'installation de R explique comment procéder à ces installations.

Ensuite, il faut charger un package dans une session R afin d'avoir accès aux fonctions d'il contient. Le démarrage d'une session R provoque automatiquement le chargement de certains packages. Par défaut, les packages suivants sont chargés automatiquement : `base`, `methods`, `datasets`, `utils`, `grDevices`, `graphics`, `stats`. Pour charger d'autres packages, il faut utiliser la fonction `library`, comme dans cet exemple :

```
library(dplyr)
```

Comment obtenir de l'information ?

Comment obtenir de l'information sur une fonction ?

Avant d'utiliser une fonction pour la première fois, il est utile de savoir :

- ce que la fonction fait ;
- la signification de ses arguments et les formats d'objets acceptés comme valeur en argument ;
- le format du résultat en sortie si la fonction retourne quelque chose.

Toutes les fonctions ont une fiche d'aide, toujours structurée de la même façon, procurant ces informations. En R, la fonction `help` (ou l'opérateur `?`) permet d'accéder à cette fiche d'aide, comme dans cet exemple :

```
help(lm)
# ou
?lm
```

Notons que pour ouvrir la fiche d'aide d'un opérateur, il faut encadrer son nom de guillemets simples ou doubles dans l'appel a la fonction `help`, par exemple :

```
help("+")
```

La majorité des fiches d'aide de fonctions R comportent des exemples d'utilisation de la fonction à la fin. Ces exemples sont souvent très utiles pour comprendre comment utiliser la fonction.

Aussi, le CRAN comporte une page web pour chaque package R qu'il héberge (https://cran.r-project.org/web/packages/). Celle-ci contient la version PDF des fiches d'aide de toutes les fonctions publiques du package (le *reference manual*). Dans certains cas, cette page comporte un lien vers un site web documentant davantage le package. Elle comporte aussi parfois des guides d'utilisation du package, appelés *vignettes*, créés par les développeurs du package (par exemple https://cran.r-project.org/web/packages/dplyr/index.html).

Les vignettes d'un package sont aussi consultables hors-ligne, à partir de R. Dans le bas d'une fiche d'aide R, il y a toujours un lien vers l'index du package contenant la fonction documentée dans la fiche. Dans la première partie de cet index, il y a un lien vers les vignettes du package, si celui-ci en possède.

Il n'y a pas que les développeurs de R et de packages R qui fournissent de l'information sur des fonctions R. La communauté R étant très active, il est fréquent de trouver sur le web de l'information à propos d'une fonction R provenant d'utilisateurs de la fonction. Par exemple, lancez la recherche « `R function lm` » dans Google. Le premier résultat est la version en ligne de la fiche d'aide de la fonction. Les résultats suivants sont des pages web très pertinentes pour apprendre à utiliser la fonction, créées par des utilisateurs.

Comment dénicher la fonction pour réaliser une certaine tâche ?

Il est relativement facile d'obtenir de l'information lorsque nous connaissons le nom de la fonction à utiliser. Par contre, il est parfois moins facile de trouver le nom de la fonction qui répond à notre besoin !

Afin de trouver une fonction R qui effectue une certaine tâche, le premier endroit pour chercher est encore **Google**. En entrant dans la barre de recherche Google « `R function` », suivi d'une courte description en anglais de la tâche à effectuer, il est fort probable que des résultats pertinents seront retournés. Le web contient vraiment beaucoup d'informations sur la programmation en R, principalement en anglais. Google suggère souvent des pages sur des sites sociaux de questions/réponses tels que **StackOverflow** (http://stackoverflow.com/questions/tagged/r) ou **CrossValidated** (http://stats.stackexchange.com/questions/tagged/r).

Il existe aussi des outils de recherche spécifiques à R. Tout d'abord, la fonction `help.start` appelée sans lui fournir d'arguments en entrées :

```
help.start()
```

ouvre une table des matières contenant des liens vers la documentation R. Toute cette information est aussi accessible en ligne sur le site web http://stat.ethz.ch/R-manual/R-patched/doc/html/.

Aussi, la fonction R `help.search` permet de rechercher la présence de chaînes de caractères dans les fiches d'aide R, par exemple :

```
help.search("regression")
```

Cependant, les fonctions `help`, `help.start` et `help.search` donnent uniquement accès à la documentation des packages R installés pour la version de R à partir duquel la commande est soumise. Pour faire une

recherche dans les fiches d'aide de tous les packages R sur le CRAN, le site web

http://www.rdocumentation.org/

est un bon outil. Il permet d'effectuer des recherches avancées, par exemple en spécifiant le champ de la fiche d'aide dans lequel restreindre la recherche.

Le site web http://rseek.org/ permet d'élargir la recherche à :

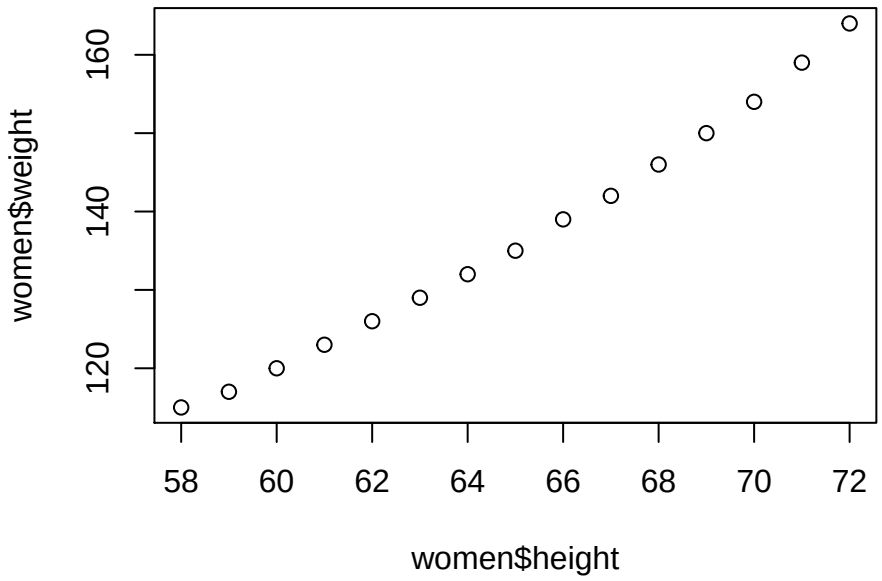
- toute la documentation distribuée par l'équipe de R, incluant les manuels du R core team (http://cran.r-project.org/manuals.html) ;
- les messages publiés sur les listes courriel de R (http://www.r-project.org/mail.html), dont le R-help où les abonnés peuvent poser des questions concernant R ;
- le R Journal (http://journal.r-project.org/) ;
- des blogues et autres sites web consacrés à R, dont R-bloggers (http://www.r-bloggers.com/) et Quick-R (http://www.statmethods.net/) ;
- et même des titres de livres traitant de programmation en R.

Il existe en fait plusieurs outils de recherche de documentation R. Le site web http://search.r-project.org/ en répertorie plusieurs.

Comment faire un graphique ?

La fonction graphique de base en R est `plot`.

```
plot(x = women$height, y = women$weight)
```



Pour un vecteur de données `x` et un vecteur de données `y` fournis en entrée, elle génère un diagramme de dispersion.

Elle accepte plusieurs autres types d'objets comme premier argument. Par exemple, si on lui donne en entrée un objet créé par la fonction `lm`, elle génère quelques graphiques de résidus (non affichés ici).

```
plot(reg)
```

Un cours sera consacré à ce sujet.

Est-ce que de la programmation orientée objet est possible en R ?

La fonction `plot`, mentionnée ci-dessus, a un comportement qui s'adapte à la classe de l'objet assigné à son premier argument. Il s'agit d'une fonction générique *polymorphe* propre à un langage orienté objet. Ainsi, la programmation orientée objet est possible en R. Il y a même plus d'un système R de programmation orientée objet. La fonction `plot` du package `graphics` est une fonction générique issue du système orienté objet S3, le plus simple et encore le plus courant en R.

Jetons un coup d'oeil au corps de la fonction `plot`.

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x0000000015c69a00>
## <environment: namespace:graphics>
```

La fonction `plot` possède seulement trois arguments : `x`, `y` et `...`. Aussi, le corps de la fonction est un peu particulier. Il est très court. Il contient uniquement un appel à la fonction `UseMethod`. Cette caractéristique indique que `plot` est une fonction générique S3.

À travers l'appel à la fonction `UseMethod`, la fonction `plot` appelle une sous-fonction (appelée *méthode S3*), choisie en fonction de la classe de l'objet fourni en entrée comme premier argument (ici `x`). Pour énumérer toutes les « versions » d'une fonction générique auxquelles notre installation de R nous donne accès, il faut appeler la fonction `methods` comme suit.

```
methods(plot)
```

```
##  [1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
##  [4] plot.default        plot.dendrogram*    plot.density*
##  [7] plot.ecdf           plot.factor*        plot.formula*
## [10] plot.function       plot.hclust*        plot.histogram*
## [13] plot.HoltWinters*   plot.isoreg*        plot.lm*
## [16] plot.medpolish*     plot.mlm*           plot.ppr*
## [19] plot.prcomp*        plot.princomp*      plot.profile.nls*
## [22] plot.raster*        plot.spec*          plot.stepfun
## [25] plot.stl*           plot.table*         plot.ts
## [28] plot.tskernel*      plot.TukeyHSD*
## see '?methods' for accessing help and source code
```

Les caractères se trouvant après le point dans le nom d'une méthode S3 représentent une classe d'objets. Par exemple, `lm` est la classe des objets retournés par la fonction du même nom. C'est la méthode `plot.lm` qui est appelée lorsque `plot` reçoit en entrée un objet de classe `lm`.

Les méthodes sont souvent documentées, au même titre que les fonctions. Par exemple, la fiche d'aide ouverte par `help(plot.lm)` fourni de l'information à propos du comportement de la fonction `plot` avec un objet de classe `lm`,

La programmation orientée objet est un sujet technique plutôt avancé, certainement pas un concept de base. Nous reviendrons sur ce concept vers la fin de la session.

Comment conserver une trace des commandes soumises au cours d'une session R ?

Il est possible de sauver l'**historique des commandes** soumises au cours d'une session R avec la fonction `savehistory`. Le fichier ainsi créé (d'extension `.Rhistory` par défaut) contient une énumération de toutes les commandes soumises dans la console depuis l'ouverture de R.

Il est aussi possible de sauver l'**environnement de travail** avec la fonction `save.image`. Le fichier ainsi créé (d'extension `.RData` par défaut) contient tout le contenu de l'environnement de travail, soit tous les objets créés depuis l'ouverture de R.

Truc :

En fait, la console R garde toujours une trace en mémoire des commandes que vous avez soumises. Vous pouvez parcourir cet historique, dans la console, avec les touches « flèche vers le haut » et « flèche vers le bas » de votre clavier. Ainsi, pour soumettre de nouveau une commande, vous pouvez éviter de la retaper en y accédant à l'aide de la touche « flèche vers le haut ».

Comment quitter R ?

La commande `q()` provoque la fin de la session R, tout comme la fermeture de la fenêtre de la console R. R demande alors si nous souhaitons sauvegarder une **image de la session**. Si nous répondons oui, il sauvegarde l'historique des commandes dans un fichier nommé « `.Rhistory` » et l'environnement de travail dans un fichier nommé « `.RData` ».

Qu'est-ce que le répertoire courant ?

R a parfois besoin de lire ou d'écrire dans des fichiers externes. Il le fait par exemple :

- lors de l'importation de données provenant d'un fichier externe (lecture dans un fichier) ;
- lors de l'enregistrement de l'historique des commandes ou de l'environnement de travail (écriture dans un fichier).

R se définit un emplacement par défaut pour ces fichiers. Cet emplacement se nomme **répertoire courant** (en anglais *current working directory*). Lorsqu'aucun emplacement n'est spécifié lors d'une communication entre R et un fichier externe, R considérera par défaut que le fichier se trouve dans le répertoire courant.

Comment contrôler le répertoire courant ?

La commande suivante permet de connaître le répertoire courant :

```
getwd()
```

La suivante permet de le modifier :

```
setwd("cheminAccesRepertoire")
```

où `"cheminAccesRepertoire"` est le chemin d'accès d'un répertoire sur votre ordinateur, par exemple `"C:/MesDocuments"` ou autre.

Attention :

En R, les chemins d'accès contenant le caractère \ ne sont pas acceptés. Ce n'est pas très pratique pour ceux qui travaillent sous Windows, car un chemin d'accès copié à partir d'un explorateur de fichiers Windows contient ce caractère. Il faut toujours remplacer les \ par / ou \\ (ex. : `"C:\MesDocuments"` devient `"C:/MesDocuments"` ou `"C:\\MesDocuments"`).

Comment charger un fichier d'extension `.RData` ?

Les objets sauvegardés dans un fichier d'extension `.RData` deviennent accessibles en R après que le fichier soit chargé avec la fonction `load`.

Attention :

Si le répertoire courant de R à son ouverture contient un fichier `.RData`, les objets contenus dans ce fichier sont automatiquement chargés en R. Dans ce cas, l'environnement de travail n'est pas vide en début de session.

Pourquoi créer des programmes R plutôt que de travailler directement dans la console ?

Retrouver, dans un ancien historique, certaines commandes que l'on cherche à réutiliser peut être difficile. Une meilleure méthode de travail que de travailler directement dans la console R consiste à rédiger des *programmes R*, aussi appelés *scripts R*.

Les programmes facilitent la réutilisation du code. Des commentaires peuvent y être insérés pour expliquer ce que font les instructions (en utilisant le symbole `#`).

```
# Exemple de commentaire
1 + 1 # calcul simple
```

Faire rouler un ancien programme recrée tous les objets produits par les commandes dans le programme. Plus besoin d'image de session, sauf si la création de ces objets est longue à rouler.

Quel outil utiliser pour éditer un programme R ?

Un programme R s'édite dans une fenêtre séparée de la console. Le GUI de la version Windows de R possède un éditeur de code R minimaliste, qui s'ouvre par le menu « Fichier > Nouveau script ». Par convention, l'**extension `.R`** est donnée à un programme R. À partir de cet éditeur, le **raccourci clavier Ctrl-R** permet de soumettre dans la console des lignes de code de l'éditeur.

Il existe cependant des outils bien plus performants que cet éditeur pour programmer en R. Plusieurs éditeurs de code R ou environnements de développement R (en anglais IDE pour Integrated Development Environment) sont offerts, notamment :

- RStudio (http://www.rstudio.com/),
- Emacs avec ESS (http://ess.r-project.org/),
- Eclipse avec StatET (http://www.walware.de/goto/statet),
- Tinn-R (http://sourceforge.net/projects/tinn-r/),
- l'extension NppToR pour Notepad++ (http://sourceforge.net/projects/npptor/),
- le module d'extension R Tools pour Visual Studio (https://www.visualstudio.com/fr/vs/rtvs/)
- etc.

RStudio est probablement le plus populaire de ces outils. Il s'agit d'un environnement de développement déjà très complet et qui continue d'être activement développé. Une version gratuite du logiciel est offerte (https://www.rstudio.com/products/rstudio/download/, version « Desktop, Open Source License »).

RStudio facilite le travail en R grâce à :

- un éditeur de code R, notamment pourvu de
  - coloration syntaxique,
  - complètement automatique,
  - pliage de code ;
- un environnement de travail qui divise la fenêtre en sous-fenêtres, dont une pour l'éditeur R et une autre pour la console R, et qui permet la communication entre les fenêtres (raccourci clavier Ctrl-Enter et autres) ;
- la possibilité de lancer certaines commandes à partir de menus, notamment :
  - des appels aux fonctions courantes `setwd`, `install.packages`, `help`, etc.,
  - des commandes de construction de package ;
- plusieurs autres fonctionnalités permettant de travailler de façon plus productive.

Quelles sont les règles de syntaxe du langage R ?

Lors de l'écriture de code R, il y a quelques règles de syntaxe à respecter. En plus des règles concernant les choix de noms d'objets énumérées précédemment, voici les règles de base du langage R.

- Chaque commande doit se terminer par un retour de chariot ou un point-virgule (moins usuel).
- Une commande peut s'étaler sur plusieurs lignes.
- Il est possible de mettre plusieurs commandes sur une même ligne, à condition de séparer les commandes par des points-virgules.
- Les différents éléments d'une commande peuvent ou non être séparés par un ou plusieurs espaces. C'est uniquement une question de style.
- Dans un nombre réel, la décimale est représentée par un point et non par une virgule.
- Le symbole `#` sert à insérer des commentaires dans un programme R. Sur une ligne, tout ce qui se trouve après un `#` est ignoré par R lors de l'exécution d'un programme.
- R fait toujours la distinction entre les majuscules et les minuscules (il respecte la casse des caractères).

Comment connaître et modifier les paramètres d'une session R ?

Une session R comporte un certain nombre de paramètres nommés options. La fonction R `options` permet de connaître les paramètres de notre session, ainsi que de les modifier. Il faut d'abord aller voir la fiche d'aide de la fonction, que l'on peut ouvrir avec la commande `help(options)`, pour obtenir de l'information sur ces paramètres.

Par exemple, la façon dont les nombres sont affichés dans la console est notamment contrôlée par les options `digits` et `scipen`. Ces options prennent par défaut les valeurs suivantes.

```
# Afin de garder une copie des valeurs par défaut avant de les modifier
optionsDefaut <- options()
```

```
optionsDefaut$digits
```

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

```
optionsDefaut$scipen
```

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

L'option `digits` représente le nombre maximal de chiffres affichés, par exemple :

```
print(1.23456789)
```

```
## [1] 1.234568
```

ou plus simplement

```
1.23456789
```

```
## [1] 1.234568
```

Le nombre est arrondi pour respecter ce nombre maximal de chiffres. Cette option peut être modifiée comme suit.

```
options(digits = 4)
1.23456789
```

```
## [1] 1.235
```

Pour toute nouvelle session R démarrée, la valeur par défaut de `digits` retombe à 7 (sauf si ce paramètre est modifié dans le fichier de configuration de R). Si nous avons préalablement enregistré les valeurs par défaut des options, il est facile de redonner à `digits` sa valeur par défaut comme suit.

```
options(digits = optionsDefaut$digits)
```

L'option `scipen` contrôle pour sa part l'affichage en format scientifique. R affiche les très grands nombres dans ce format. Par exemple, par défaut

```
100000
```

```
## [1] 1e+05
```

est affiché en format scientifique, mais pas

```
10000
```

```
## [1] 10000
```

En modifiant la valeur de `scipen` comme suit

```
options(scipen = -1)
```

10000 sera lui aussi affiché en format scientifique.

```
10000
```

```
## [1] 1e+04
```

Où trouver plus d'information ?

Voici des sources d'information sur l'utilisation de R et la programmation R. Cette liste n'est pas exhaustive. Les sources d'information R sont très nombreuses !

Documentation de R : fiches d'aide des fonctions et manuels du R core team :

- accessible à partir de la console au cours d'une session R grâce aux fonctions `help.start`, `help` et `help.search` ;
- accessible en ligne :
  - http://stat.ethz.ch/R-manual/R-patched/doc/html/,
  - https://cran.r-project.org/manuals.html,
  - http://www.r-project.org/ > « Documentation » dans le menu de gauche.

Référence officielle pour le logiciel R et sa documentation :

R Core Team (2017). R : A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Documentation de packages R qui ne se retrouvent pas dans l'installation de base :

- accessible à partir de la console au cours d'une session R lorsque les packages sont installés grâce aux fonctions `help.start`, `help` et `help.search` ;
- accessible en ligne :
  - https://cran.r-project.org/web/packages/,
  - http://bioconductor.org/packages ;
- pour connaître les packages les plus téléchargés dans le dernier mois :
  - https://www.rdocumentation.org/trends.

Outils de recherche dans la documentation de R :

- http://www.rdocumentation.org/,
- http://rseek.org/.

Sites web à propos de R :

- http://www.statmethods.net/,
- http://www.cookbook-r.com/,
- http://www.r-graph-gallery.com/.

Formations ou exercices R :

- http://r-exercises.com/,
- http://tryr.codeschool.com/,
- http://swirlstats.com,
- http://swcarpentry.github.io/r-novice-inflammation/,
- http://swcarpentry.github.io/r-novice-gapminder/.

Outils sociaux de questions/réponses :

- StackOverflow (http://stackoverflow.com/questions/tagged/r),
- CrossValidated (http://stats.stackexchange.com/questions/tagged/r),
- la liste courriel R-help (https://stat.ethz.ch/mailman/listinfo/r-help).

Outils pour suivre la communauté R :

- http://www.r-bloggers.com/,
- https://rweekly.org/,
- hashtag #rstats sur Twitter : https://twitter.com/hashtag/rstats,
- le groupe « The R Project for Statistical Computing » sur LinkedIn : https://www.linkedin.com/groups/77616.

Livres et notes de cours

Référence officielle pour les notes de ce cours :

- Baillargeon, S. (2018). R pour scientifique : notes de cours, STT-4230/STT-6230. Université Laval, Département de mathématiques et de statistique. URL https://stt4230.rbind.io/

Livres que j'utilise le plus :

- Matloff, N. (2011). The art of R programming : A tour of statistical software design. No Starch Press.
- Wickham, H. (2014). Advanced R. CRC Press. URL http://adv-r.had.co.nz/
- Wickham, H. (2015). R packages. O'Reilly Media, Inc. URL http://r-pkgs.had.co.nz/
- Grolemund, G. et Wickham, H. (2016). R for Data Science. O'Reilly Media, Inc. URL http://r4ds.had.co.nz/

Autres livres :

- Adler, J. (2010). R in a nutshell : A desktop quick reference. 2e edition. O'Reilly Media, Inc.
- Braun, W. J. et Murdoch, D. (2007). A first Course in Statistical Programming with R. Cambridge University Press.
- Chambers, J. M. (2016). Extending R. Chapman and Hall/CRC.
- Chambers, J. M. (2008). Software for Data Analysis : Programming with R. Springer.
- Cotton, R. (2013). Learning R : A Step-by-Step Function Guide to Data Analysis, O'Reilly Media.
- Davies, T. M. (2016). The Book of R : A First Course in Programming and Statistics. No Starch Press.
- Gillespie, C. et Lovelace, R. (2016). Efficient R Programming : A Practical Guide to Smarter Programming. O'Reilly Media, Inc. version en ligne : https://csgillespie.github.io/efficientR/
- Gentleman, R. (2009). R Programming for Bioinformatics. Chapman and Hall/CRC.
- Kabacoff, R. (2015), R in action. 2e édition. Manning Publications.
- Muenchen, R. A. (2011). R for SAS and SPSS Users. Second edition. Springer.
- Teetor, P. (2011). R Cookbook. O'Reilly Media.
- Zumel, N. et Mount, J. (2014). Practical Data Science with R. Manning Publications Co.
- Zuur, A. F., Ieno, E. N. et Meesters, E. H.W.G. (2009). A Beginner's Guide to R. Springer.
- les livres répertoriés par le R core team : https://www.r-project.org/doc/bib/R-jabref.html.