Concepts de base en R

Sophie Baillargeon, Université Laval

2017-01-13

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, mais on le réservera pour l'assignation d'une valeur à un argument lors de l'appel d'une fonction. J'expliquerai plus tard la différence entre `<-` et `=`, qui est trop technique pour un premier cours.

Quels noms sont acceptés en R ?

Les noms d'objets R doivent respecter les règles suivantes :

- 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 ;
- 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 :

```
a
```

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

```
print(a)
```

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

Est-ce le seul objet accessible ?

```
ls()
```

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

Cette commande (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 l'ouverture de R. 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 classés ailleurs que dans l'environnement de travail. Voici la liste de tous les *groupes* d'objets auxquels vous avez accès.

```
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 :

```
colMeans
```

```
## function (x, na.rm = FALSE, dims = 1L)
## {
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     if (!is.array(x) || length(dn <- dim(x)) < 2L)
##         stop("'x' must be an array of at least two dimensions")
##     if (dims < 1L || dims > length(dn) - 1L)
##         stop("invalid 'dims'")
##     n <- prod(dn[id <- seq_len(dims)])
##     dn <- dn[-id]
##     z <- if (is.complex(x))
##         .Internal(colMeans(Re(x), n, prod(dn), na.rm)) + (0+1i) *
##             .Internal(colMeans(Im(x), n, prod(dn), na.rm))
##     else .Internal(colMeans(x, n, prod(dn), na.rm))
##     if (length(dn) > 1L) {
##         dim(z) <- dn
##         dimnames(z) <- dimnames(x)[-id]
##     }
##     else names(z) <- dimnames(x)[[dims + 1L]]
```

```
##     z
## }
## <bytecode: 0x000000001965c420>
## <environment: namespace:base>
```

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

```
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
```

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

```
str(colMeans)
```

```
## function (x, na.rm = FALSE, dims = 1L)
```

```
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 ?

C'est un bout de code qui produit un certain résultat. Une fonction prend des valeurs en entrée. On les nomme *arguments*. Elle traite ces valeurs, puis retourne un objet en sortie ou encore produit un résultat comme un graphique ou l'écriture dans un fichier externe.

FIGURE 1 – Représentation schématique d'une fonction

Un objet de type fonction est composé des éléments suivants :

- une liste d'arguments avec leurs valeurs par défaut s'il y a lieu, par exemple :

```
args(colMeans)
```

```
## function (x, na.rm = FALSE, dims = 1L)
## NULL
```

- le corps de la fonction, soit le code composant la fonction, par exemple :

```
body(colMeans)
```

```
## {
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     if (!is.array(x) || length(dn <- dim(x)) < 2L)
##         stop("'x' must be an array of at least two dimensions")
##     if (dims < 1L || dims > length(dn) - 1L)
##         stop("invalid 'dims'")
##     n <- prod(dn[id <- seq_len(dims)])
##     dn <- dn[-id]
##     z <- if (is.complex(x))
##         .Internal(colMeans(Re(x), n, prod(dn), na.rm)) + (0+1i) *
##             .Internal(colMeans(Im(x), n, prod(dn), na.rm))
##     else .Internal(colMeans(x, n, prod(dn), na.rm))
##     if (length(dn) > 1L) {
##         dim(z) <- dn
##         dimnames(z) <- dimnames(x)[-id]
##     }
##     else names(z) <- dimnames(x)[[dims + 1L]]
##     z
## }
```

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

```
colMeans(x = women, na.rm = FALSE)
```

Comment écrire un appel de fonction ?

- On tape d'abord le nom de la fonction (ex. : `colMeans`) ;
- ensuite on ouvre une parenthèse ;
- ensuite on donne des valeurs aux arguments :
  - typiquement on écrit le nom de l'argument (ex. : `x`),
  - suivi de l'opérateur `=`,
  - suivi d'une commande générant la valeur à fournir en argument (souvent le nom d'un objet, ex. : `women`),
  - si on a d'autres arguments à fournir, on les fournit exactement de la même façon, en séparant les arguments par une virgule ;
- finalement on referme la parenthèse.

Exemples d'appels de fonction

Revenons à l'exemple précédent :

```
colMeans(x = women, na.rm = FALSE)
```

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

Si on souhaite utiliser la valeur par défaut d'un argument, il suffit de ne pas spécifier sa valeur.

```
colMeans(x = women)
```

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

La valeur affectée à un argument n'est pas obligatoirement le nom d'un objet. Elle peut être une expression créant un objet, par exemple :

```
colMeans(x = cbind(1:10, 10:1))
```

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

Si on n'assigne pas de nom d'argument à une valeur, la valeur est affectée au premier argument non spécifié selon l'ordre des arguments dans la définition de la fonction, par exemple :

```
colMeans(women)
```

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

```
colMeans(women, FALSE)
```

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

```
colMeans(FALSE, women)
```

```
## Error in colMeans(FALSE, women): 'x' must be an array of at least two dimensions
```

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

```
colMeans(women, na.rm = FALSE)
```

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.

```
args(paste)
```

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

C'est aussi le cas de la fonction `mean` pour calculer une moyenne.

```
args(mean)
```

```
## function (x, ...)
## NULL
```

L'argument `...` a deux utilités potentielles.

Utilité 1 : recevoir un nombre indéterminé d'arguments

Sa première utilité est de permettre à une fonction de recevoir en entrée un nombre indéterminé d'arguments. Si `...` a cette utilité, il est typiquement placé au tout début de la liste des arguments. C'est le cas par exemple de la fonction `paste`. C'est donc grâce à l'argument `...` que la fonction `paste` peut concaténer les valeurs provenant d'autant d'objets qu'on le souhaite, comme dans ces exemples.

```
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"
```

Utilité 2 : passer des arguments à une autre fonction

Si l'argument `...` n'est pas placé au début de la liste des arguments d'une fonction, il a presque toujours une utilité différente de l'utilité 1. Dans ce cas, `...` sert plutôt à passer des arguments à une autre fonction appelée dans le corps de la première fonction. Considérons par exemple la fonction `mean`. Voici la liste des arguments ainsi que le corps de cette fonction.

```r
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000017142028>
## <environment: namespace:base>
```

La fonction `mean` possède seulement deux arguments : `x` et `....` Aussi, le corps de la fonction est un peu particulier. Il est très court. Il contient uniquement un appel à la fonction `UseMethod`. En conséquence, `mean` est ce que l'on appelle en R une *fonction générique*. Le comportement de la fonction dépend du type de l'objet fourni en entrée comme premier argument (ici `x`). Plus formellement, on appel ce type la *classe* de l'objet. Il s'agit d'une forme de *programmation orientée objet*. Nous reviendrons sur ce concept dans un prochain cours, car il s'agit d'un concept plus avancé et non d'un concept de base. Pour l'instant, nous voulons simplement comprendre l'utilité de l'argument `....` Il suffit donc de savoir qu'à travers l'appel à la fonction `UseMethod`, la fonction `mean` appelle une sous-fonction (appelée *méthode S3*), choisie en fonction de la classe de l'argument `x`. L'argument `...` sert simplement à passer des arguments autres que ceux déjà présents dans la liste des arguments de `mean` à la sous-fonction. Par exemple, on peut fournit à `mean` l'argument `trim` ou l'argument `na.rm` même si ces arguments ne sont pas dans la liste des arguments de `mean` grâce aux `....` Ces arguments sont dans la liste des arguments de la méthode `mean.default` :

```r
args(mean.default)
```

```
## function (x, trim = 0, na.rm = FALSE, ...)
## NULL
```

Si `mean` choisit d'appeler la méthode `mean.default`, il inclura automatiquement tous les arguments portant un nom autre que `x` (le seul argument nommé dans la liste des arguments de `mean`) dans l'appel à la méthode `mean.default`.

Ainsi, lors de l'appel à la fonction `mean` pour calculer la moyenne de valeurs numériques dans un vecteur, on peut fournir l'argument `trim` pour demander le calcul d'une moyenne tronquée comme dans l'exemple suivant.

```r
mean(c(1,2,3,4,5,10), trim = 0)
```

```
## [1] 4.166667
```

```r
mean(c(1,2,3,4,5,10), trim = 0.2)
```

```
## [1] 3.5
```

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 :

```
`+`(1, 1)
```

Comment utiliser une fonction d'ajustement de modèle ?

R possède une classe d'objet servant à la spécification de modèles : les **formules**.

Par exemple, la fonction `lm` sert à ajuster un modèle de régression. Le premier argument de cette fonction est une formule. Par exemple :

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

Pour plus d'information :

```
help(as.formula)
```

On reparlera de l'écriture de formules dans un prochain cours.

Comment accéder aux résultats de la fonction ?

```
reg
```

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

On ne voit ainsi qu'un court extrait des résultats. Un résumé un peu plus complet des résultats est obtenu avec

```
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
```

L'objet `reg` est en fait une liste de sous-objets contenant les résultats :

```
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:
```

Je donnerai au prochain cours de l'information sur les listes. Pour l'instant, je me contente de dire que l'on peut extraire un élément de `reg` avec l'opérateur `$` comme suit :

```
reg$coefficients
```

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

ou encore en utilisant une fonction générique servant à extraire des éléments d'une sortie de modèle, par exemple :

```
coefficients(reg)
```

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

La fonction générique `summary` permet même d'accéder à encore plus de résultats de l'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
```

Comment avoir de l'information sur une fonction ?

Avant d'utiliser une fonction pour la première fois, on a besoin 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, on accède à cette fiche d'aide grâce à la fonction `help` (ou `?`), comme dans cet exemple :

```
help(colMeans)
```

ou

```
?colMeans
```

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("+")
```

Comment dénicher la fonction dont on a besoin ?

Il est facile d'obtenir de l'information lorsqu'on connaît 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 simplement **Google**. En entrant dans la barre de recherche Google `R function` suivi d'une courte description en anglais de la tâche à effectuer, on obtient habituellement une grande quantité de résultats. Le web contient vraiment beaucoup d'informations sur la programmation en R, principalement en anglais. On tombe souvent 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 comme suit :

```
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, le CRAN comporte une page web pour chaque package qu'il héberge (https://cran.r-project.org/web/packages/). On peut y télécharger la version PDF des fiches d'aide de toutes les fonctions publiques du package (le *reference manual*) et comporte parfois des guides d'utilisation du package, appelés *vignettes* (par exemple https://cran.r-project.org/web/packages/ggplot2/index.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 du programme 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.

Qu'est-ce qu'un package ?

Parfois, la fonction dont on a besoin ne vient pas avec l'installation de base de R, mais plutôt dans un package développé par un utilisateur de R.

Un package R est un **regroupement de fonctions et de données documentées**.

La majorité des développeurs de packages les rendent disponibles sur le **CRAN** de R : 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 ?

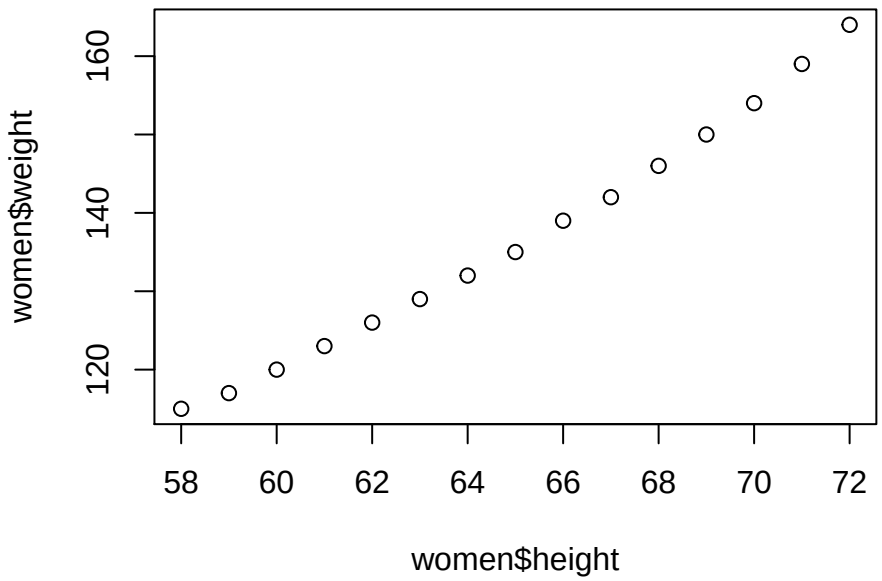
Après avoir installé le package (voir la fiche sur l'installation de R), il faut le charger dans notre session R afin d'avoir accès aux fonctions du package. On fait ce chargement avec la fonction `library`, comme dans cet exemple :

```
library(Rcmdr)
```

Comment faire un graphique ?

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

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



Si on lui donne un vecteur de données `x` et un vecteur de données `y`, elle génère un diagramme de dispersion.

Elle accepte plusieurs autres types d'arguments en entrée. 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.

R est un langage orienté objet.

Tel que mentionné lorsque l'on a parlé de la fonction `mean`, R est un langage orienté objet. La fonction `plot`, tout comme la fonction `mean`, est une fonction générique *polymorphe* propre à un langage orienté objet : le comportement de la fonction dépend du type d'argument qu'on lui donne en entrée.

Pour énumérer toutes les « versions » d'une fonction générique auxquelles on a accès, on appelle 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
```

Nous reviendrons plus tard sur ce sujet plus technique.

Comment conserver une trace de ce que l'on a fait ?

On peut sauver l'**historique des commandes** avec la fonction `savehistory` ou par le menu « Fichier » du R GUI (Graphical User Interface, soit interface graphique en français). 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.

On peut aussi sauver l'**environnement de travail** avec la fonction `save.image` ou par le menu « Fichier » du R GUI. 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.

Bon à savoir : En fait, R garde toujours une trace en mémoire des commandes que vous avez soumises dans la console. 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 ?

On quitte R en fermant le GUI de R ou en soumettant la commande `q()`. On met alors fin à une **session R**.

R demande alors si on souhaite sauvegarder une **image de la session**. Si on répond oui, il sauvegarde l'historique des commandes dans un fichier nommé « .Rhistory » et l'environnement de travail dans un fichier nommé « .RData ».

Mais où se retrouvent ces fichiers sur notre ordinateur ?
(R n'a pas demandé dans quel emplacement on souhaitait enregistrer les fichiers créés.)
Réponse : dans le **répertoire courant** actif à la fermeture de R.

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 (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). Lorsque l'on ne spécifie pas d'emplacement 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 ?

On peut connaître le répertoire courant avec la commande :

```
getwd()
```

et le modifier avec la commande :

```
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

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 ?

On peut avoir accès aux objets sauvegardés dans un fichier d'extension .RData en utilisant la fonction `load` ou par le menu « Fichier » du R GUI.

Attention : Si le répertoire courant de R à son ouverture contient un fichier .RData, les objets contenus dans ce fichier seront 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. On peut y insérer des commentaires pour expliquer ce que font les commandes (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. Mais même dans ce cas, il existe d'autres solutions que l'image de session, que nous verrons plus loin.

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, que l'on peut ouvrir par le menu « Fichier > Nouveau script ». Par convention, on donne l'**extension .R** à un programme R. À partir de cet éditeur, le **raccourci clavier Ctrl-R** permet de soumettre dans la console des lignes de code dans 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 »).

RStudio facilite le travail en R grâce à :

- un éditeur de code R, notamment pourvu de coloration syntaxique ;

- 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 ;
- la possibilité de lancer des commandes de construction de package simplement à partir d'un menu ;
- 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. Cela n'a aucune importance.
- 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 case).

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.

```
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. On peut modifier cette option 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 on a modifié ce paramètre dans le fichier de configuration de R). On peut aussi redonner à `digits` sa valeur par défaut comme suit.

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

D'options `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.

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

- à partir de la console au cours d'une session R grâce aux fonctions `help.start`, `help` et `help.search` ;
- 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 (2016). 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 :

- 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` ;
- en ligne :
  - https://cran.r-project.org/web/packages/ ;
  - http://bioconductor.org/packages ;
- pour connaître les packages les plus téléchargés :
  - https://awesome-r.com/index.html.

Outils sociaux de questions/réponses, notamment :

- 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).

Blogues et autres sites web à propos de R, notamment :

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

Formations ou exercices R, notamment

- 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/,

Livres et notes de cours

Référence officielle pour les notes de ce cours :

- Baillargeon, S. (2017). R pour scientifique : notes de cours, STT-4230/STT-6230. Université Laval, Département de mathématiques et de statistique. URL http://archimede.mat.ulaval.ca/dokuwiki/doku.php?id=r

Livres que j'utilise le plus :

- Wickham, H. et Grolemund, G. (2015). R for Data Science. O'Reilly Media, Inc. URL http://r4ds.had.co.nz/
- 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/
- Matloff, N. (2011). The art of R programming : A tour of statistical software design. No Starch Press.
- Adler, J. (2010). R in a nutshell : A desktop quick reference. 2e edition. O'Reilly Media, Inc.

Autres livres :

- 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.
- 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.