Informations techniques concernant R

Sophie Baillargeon, Université Laval

2020-02-27

Table des matières

Imprécision numérique

Attention, R ne garde pas en mémoire de façon tout à fait exacte :

- certains nombres décimaux et
- les nombres plus grands, en valeur absolue, qu'une certaine limite.

Ces imprécisions numériques sont dues à la représentation en virgules flottantes utilisée par R pour stocker en mémoire les nombres réels (voir `help(double)`).

Nombres décimaux

Voici quelques exemples de tours que l'imprécision numérique en R sur des nombre décimaux peut nous jouer, tirés de Burns, P. (2011). The R inferno, chapitre 1. http://www.burns-stat.com/pages/Tutor/R_inferno.pdf.

Pourquoi la commande suivante ne retourne-t-elle pas `TRUE` ?

```
.1 == .3/3
```

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

Pourquoi le 4^{e} élément du vecteur suivant n'est-il pas `TRUE` ?

```
seq(0,1,0.1) == .3
```

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

Pourquoi la longueur de cet objet n'est-elle pas 1 ?

```
unique(c(.3, .4 - .1, .5 - .2, .6 - .3, .7 - .4))
```

```
## [1] 0.3 0.3 0.3
```

Parce R a une précision limitée pour garder des nombres en mémoire. Nous sommes portés à croire que 0.1 en R est réellement le nombre 0.1, soit un dixième (1/10), mais en fait R a stocké en mémoire la valeur suivante :

```
print(.1, digits = 20)
```

```
## [1] 0.10000000000000001
```

alors que .3/3 a été stocké en mémoire comme suit :

```
print(.3/3, digits = 20)
```

```
## [1] 0.099999999999999992
```

donc ces deux nombres ne sont pas exactement égaux ! Par contre, ils sont approximativement égaux. D'où l'intérêt d'être « tolérant » dans nos comparaisons entre deux nombres réels.

```
all.equal(.1, .3/3)
```

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

Différences de comportement entre `unique` versus `table` ou `factor`

La fonction `table` (ainsi que `xtabs` ou `ftable`), qui calcule des fréquences, commence par trouver les valeurs uniques dans les vecteurs reçus en entrée, puis elle compte combien de fois chacune des combinaisons de valeurs uniques sont répétées dans les vecteurs. Cependant, l'exemple suivant nous montre qu'elle ne trouve pas toujours les mêmes valeurs uniques que la fonction `unique`.

```
x <- c(.3, .4 - .1, .5 - .2, .6 - .3, .7 - .4)
x # même vecteur que précédemment
```

```
## [1] 0.3 0.3 0.3 0.3 0.3
```

```
print(x, digits = 20)
```

```
## [1] 0.29999999999999999 0.30000000000000004 0.29999999999999999 0.29999999999999999
## [5] 0.29999999999999993
```

```
table(x)
```

```
## x
## 0.3
##   5
```

```
unique(x)
```

```
## [1] 0.3 0.3 0.3
```

```
print(unique(x), digits = 20)
```

```
## [1] 0.29999999999999999 0.30000000000000004 0.29999999999999993
```

Ici, `table` considère que `x` contient une seule valeur distincte (répétée 5 fois), alors que `unique` trouve 3 valeurs distinctes.

La fonction `factor` (ou `as.factor`) doit aussi trouver les valeurs uniques dans un vecteur avant de pouvoir le transformer en facteur (chaque valeur unique devient un niveau du facteur). Cependant, elle non plus ne trouve pas toujours les mêmes valeurs uniques que la fonction `unique`.

```
factor(x)
```

```
## [1] 0.3 0.3 0.3 0.3 0.3
## Levels: 0.3
```

```
nlevels(factor(x))
```

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

Il semble donc que `table` (ainsi que `xtabs` et `ftable`) et `factor` (ou `as.factor`) comparent les valeurs en ne tenant pas compte de différences vraiment minimes, alors que `unique` en tient compte.

Conseils :

Lors de comparaison de nombres en R, il faut garder en tête que `==`, `identical` et `unique` sont affectés par les imprécisions numériques, car ils font des comparaisons exactes. Cependant, `all.equal`, `table` (ainsi que les autres fonctions de calcul de fréquences du R de base) et `factor` (ou `as.factor`) ne se préoccupent pas de différences inférieures à une certaine tolérance.

Petits nombres en valeur absolue

Aussi, l'imprécision numérique sur des nombres décimaux implique que R n'arrive pas à garder en mémoire des nombres vraiment très petits en valeur absolue. Ces nombres sont stockés comme s'ils étaient des zéros.

Par exemple, sur mon installation de R, le nombre 10^{-325} est ramené à 0 par R.

```
print(10^(-325))
```

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

Le nombre 10^{-310} ne l'est cependant pas.

```
print(10^(-310))
```

```
## [1] 1e-310
```

Si ces deux nombres sont divisés, nous pouvons facilement trouver à la main le résultat de la division :

- $\frac{10^{-325}}{10^{-310}} = 10^{(-325\ -\ -310)} = 10^{-15}$, ou
- $\frac{10^{-310}}{10^{-325}} = 10^{(-310\ -\ -325)} = 10^{15}$.

Cependant, si nous demandons à R de calculer ces quotients, nous obtenons :

```
10^(-325) / 10^(-310)
```

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

car ici R considère que le numérateur vaut 0,

```
10^(-310) / 10^(-325)
```

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

car cette fois R considère que le dénumérateur vaut 0.

Solution :

Lorsque nous travaillons avec de tels nombres, il est bon de les **changer d'échelle afin de les éloigner de 0**. Typiquement, c'est une **transformation logarithmique** qui est utilisée pour arriver à ça. Par exemple, en statistique, il est plus facile de travailler numériquement avec une log-vraisemblance qu'avec une vraisemblance, dont la valeur peut être très proche de zéro étant donné qu'il s'agit d'une fonction de densité conjointe.

Pour revenir à l'exemple précédent, tentons à nouveau de calculer en R le quotient $\frac{10^{-325}}{10^{-310}}$, mais cette fois en passant par une transformation logarithmique en base 10. En raison des propriétés des logarithmes, nous savons que

$$\frac{10^{-325}}{10^{-310}} = 10^{\log_{10}\left(\frac{10^{-325}}{10^{-310}}\right)} = 10^{\log_{10}(10^{-325}) - \log_{10}(10^{-310})}.$$

Ainsi, si ce que nous conservons en mémoire est le logarithme en base 10 des nombres à manipuler, soit $\log_{10}(10^{-325})$ et $\log_{10}(10^{-310})$, que nous pouvons simplifier à -325 et -310, calculer le quotient qui nous intéresse revient à effectuer le calcul suivant :

```
10^(-325 - -310)
```

```
## [1] 1e-15
```

qui n'est affecté par aucune imprécision numérique.

Grands nombres en valeur absolue

Il faut aussi se méfier lorsque nous manipulons en R de grands nombres en valeur absolue.

```
100000000000000 + 1 == 100000000000000  # Ces deux nombres sont bel et bien différents.
```

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

```
1e14 + 1 == 1e14  # Même comparaison, en notation scientifique.
```

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

```
2^53 == 2^53 - 1  # Nous obtenons encore le résultat attendu.
```

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

```
2^53 == 2^53 + 1  # Oups, c'est à partir d'ici que ça ne marche plus!
```

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

Le plus grand nombre entier positif représentable en R 64 bits sur mon ordinateur est 2^53. Quel est ce nombre ?

```
print(2^53)
```

```
## [1] 9.007199e+15
```

Modifions l'option de la session R relative à la notation scientifique de façon à voir le nombre en notation fixe.

```
options(scipen= 10)
print(2^53)
```

```
## [1] 9007199254740992
```

Maintenant, observons un phénomène étonnant. Demandons à R d'afficher le nombre 2^53 + 1, soit 9007199254740993.

```
print(9007199254740993)
```

```
## [1] 9007199254740992
```

R affiche plutôt le nombre 2^53, soit 9007199254740992. Pourtant, ceci est correct :

```
print(9007199254740994)
```

```
## [1] 9007199254740994
```

Mais pas ceci :

```
print(9007199254740995)
```

```
## [1] 9007199254740996
```

L'explication à ces inexactitudes est que les nombres sont stockés dans la mémoire d'un ordinateur sous forme de bits. Cependant, le nombre de bits pouvant être utilisé pour stocker un nombre possède une limite maximale. Si le nombre à stocker requière plus de bits pour être représenté dans son intégralité que le nombre maximum de bits utilisables, il est légèrement arrondi.

La limite dans le nombre de bits utilisables en R pour stocker un nombre dépend du type de données utilisé pour le stocker (réel versus entier), du nombre de bits de la version de R utilisée (32 versus 64) ainsi que des spécifications de l'ordinateur utilisé. R 32 bits utilise moins de bits que R 64 bits.

Notez que la plus grande valeur représentable de 2^53 sur mon installation de R 64 bits s'applique à des nombres entiers d'un point de vue mathématique, mais stockés sous le type réel en R. Le type entier ne permet pas de représenter des nombres supérieurs en valeur absolue à l'objet R `.Machine$integer.max` (= 2147483647 sur mon installation de R 64 bits).

```
as.integer(-2147483647)
```

```
## [1] -2147483647
```

```
as.integer(-2147483647-1)
```

```
## Warning: NAs introduced by coercion to integer range
```

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

```
options(scipen= 0)  # réattribution de sa valeur par défaut à l'option scipen
```

Évaluation d'expressions en R et environnements

Lorsque nous soumettons des instructions dans la console, R doit chercher la valeur des objets nommés dans ces instructions. Ces objets peuvent être de n'importe quel type : des fonctions, des vecteurs, des listes, etc. La façon dont R s'y prend pour trouver les valeurs de ces objets est plutôt complexe. Je ne vais pas expliquer ce point en détail. Une bonne référence pour en apprendre plus sur le sujet est le blogue suivant : http://blog.obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/

Je vais tout de même expliquer le principe général simplifié derrière l'évaluation d'expressions en R. À l'ouverture d'une session R, certains packages sont automatiquement chargés. Le contenu de chaque package chargé est placé dans ce qui est appelé un *environnement*. Les objets R que nous créons pendant une session R sont aussi conservés dans un environnement : l'environnement de travail (aussi appelé environnement courant ou environnement global). Lorsque R a besoin d'évaluer un objet apparaissant dans une expression soumise dans la console, il démarre une recherche de cet objet dans les environnements ouverts. La recherche débute dans l'environnement de travail. Si l'objet recherché n'est pas trouvé dans cet environnement, la recherche se poursuit dans les environnements des packages chargés, en commençant par les packages les plus récemment chargés.

La figure 1 illustre le chemin de recherche parcouru par R pour évaluer les expressions soumises dans la console. La recherche part de l'environnement du haut et descend dans les environnements en dessous, jusqu'à ce que l'objet recherché soit trouvé. C'est le nom d'un objet qui permet de l'identifier.

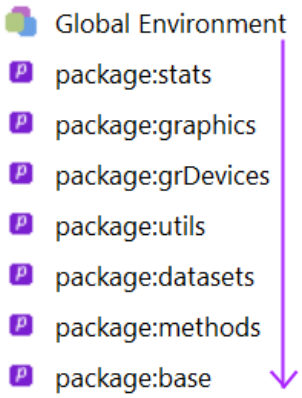


FIGURE 1 – Chemin de recherche parcouru par R pour évaluer les expressions soumises dans la console

Il existe quelques exceptions à cette façon de rechercher la valeur d'expressions en R. Les points suivants décrivent quelques-unes de ces exceptions.

Évaluation du corps d'une fonction

Lors de l'évaluation du corps d'une fonction (code composant la fonction), des environnements s'ajoutent au début du chemin de recherche décrit ci-dessous, uniquement le temps de l'évaluation de la fonction. Tous les détails seront fournis au prochain cours.

Évaluation d'expressions dans une formule ou dans un appel à la fonction `subset`, `transform` ou une fonction du `tidyverse` possédant un argument `data` ou `.data`

Les fonctions R prenant en entrée une formule, prennent aussi toutes en entrée un argument `data`. Si un data frame est donné à cet argument, ce data frame devient un environnement de plus, qui s'ajoute à la liste des environnements parcourus lors de la recherche de la valeur d'un objet R. Cet environnement vient en premier dans la liste, la recherche débute donc par lui. Dans une formule, les noms des colonnes de `data` deviennent donc des objets directement accessibles (plus besoin d'utiliser la syntaxe `data$nomColonne`). Il en est de même dans l'expression fournie à l'argument `subset` souvent présent dans les fonctions prenant en entrée une formule.

Les fonctions `subset` et `transform`, que nous avons déjà vues, ainsi que toutes les fonctions du `tidyverse` possédant un argument `data` ou `.data`, ont le même genre de comportement. Le data frame donné au premier argument devient l'environnement d'où part la recherche des valeurs associées aux noms d'objets dans les expressions fournies aux arguments suivants.

Illustrons ce type d'évaluation avec la fonction `subset`. Supposons que nous voulons extraire du jeu de données `iris` du package `datasets` les observations de `Petal.Length` et `Petal.Width` pour lesquelles `Sepal.Width` est supérieure à 4, nous pouvons utiliser la commande

```
subset(iris, subset = Sepal.Width > 4, select = c(Petal.Length, Petal.Width))
```

```
##    Petal.Length Petal.Width
## 16          1.5         0.4
## 33          1.5         0.1
## 34          1.4         0.2
```

plutôt que

```
subset(iris, subset = iris$Sepal.Width > 4, select = c("Petal.Length", "Petal.Width"))
```

ou

```
iris[iris$Sepal.Width > 4, c("Petal.Length", "Petal.Width")]
```

La commande suivante ne fonctionne pas

```
iris[Sepal.Width > 4, c(Petal.Length, Petal.Width)]
```

```
## Error in `[.data.frame`(iris, Sepal.Width > 4, c(Petal.Length, Petal.Width)) :
##   object 'Petal.Length' not found
```

parce que `Sepal.Width`, `Petal.Length` et `Petal.Width` sont des noms d'objets dont R doit trouver les valeurs. Le chemin de recherche parcouru par R pour trouver ces valeurs débute dans l'environnement de travail et se poursuit dans les environnements des packages chargés, en commençant par les packages les plus récemment chargés. R retourne une erreur, car il ne trouve nulle part ces objets. Il y a bien des colonnes du data frame `iris` qui portent les noms de ses objets, mais il ne s'agit pas d'objets directement accessibles. Ce sont des éléments dans un objet.

Dans la commande `subset(iris, subset = Sepal.Width > 4, select = c(Petal.Length, Petal.Width))`, R doit aussi trouver la valeur des objets `Sepal.Width`, `Petal.Length` et `Petal.Width`. Il trouve cette fois ces objets, car la fonction `subset` modifie le chemin de recherche de R (mais uniquement le temps de l'évaluation de l'appel à cette fonction). Elle ajoute au tout début du chemin de recherche un environnement, contenant les colonnes du data frame fourni comme premier argument. Dans l'exemple, l'environnement ajouté au chemin de recherche contient donc des vecteurs nommés `"Sepal.Length"`, `"Sepal.Width"`, `"Petal.Length"`, `"Petal.Width"` et `"Species"`, soit les colonnes de `iris`.

Le chemin de recherche utilisé par R pour évaluer les expressions données à `subset` dans cet exemple est illustré dans la figure 2.

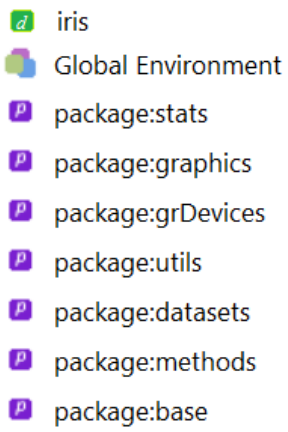


FIGURE 2 – Chemin de recherche parcouru par R pour évaluer les expressions données en argument à la fonction subset dans l'exemple

Les fonctions `with` et `within`

Les fonctions `with` et `within` permettent d'étendre la possibilité d'ajouter un data frame au début de la liste des environnements dans le chemin de recherche à n'importe quelles instructions R. Elles s'emploient avec la syntaxe suivante :
`with(data = nom_data_frame, expr = expression)`
où `expression` est une commande R, ou encore plusieurs commandes R encadrées d'accolades.

Par exemple, les deux bouts de code suivants retournent le même résultat.

```r
var_catego <- cut(
  x = iris$Sepal.Length,
  breaks = c(-Inf, quantile(iris$Sepal.Length, probs = c(1/3, 2/3)), Inf),
  right = FALSE
)
table(var_catego, iris$Species)

# équivalent à :

with(iris, {
  var_catego <- cut(
    x = Sepal.Length,
    breaks = c(-Inf, quantile(Sepal.Length, probs = c(1/3, 2/3)), Inf),
    right = FALSE
  )
  table(var_catego, Species)
})
```

```
##             Species
## var_catego   setosa versicolor virginica
##   [-Inf,5.4)     40          5         1
##   [5.4,6.3)      10         31        12
##   [6.3, Inf)      0         14        37
```

Avec `with`, les préfixes `nom_data_frame$` ne sont plus requis.

La fonction `within` est pour sa part une sorte d'alternative à la fonction `transform`. Les expressions fournies en entrée vont modifier le data frame ou la liste donné comme premier argument. La fonction retourne une copie modifiée de cet objet. Voici un exemple.

```r
head(women)
```

```
##   height weight
## 1     58    115
## 2     59    117
## 3     60    120
## 4     61    123
## 5     62    126
## 6     63    129
```

```r
women2 <-
  within(data = women, expr = {
    height <- height / 12
    weight2 <- weight^2
  })
head(women2)
```

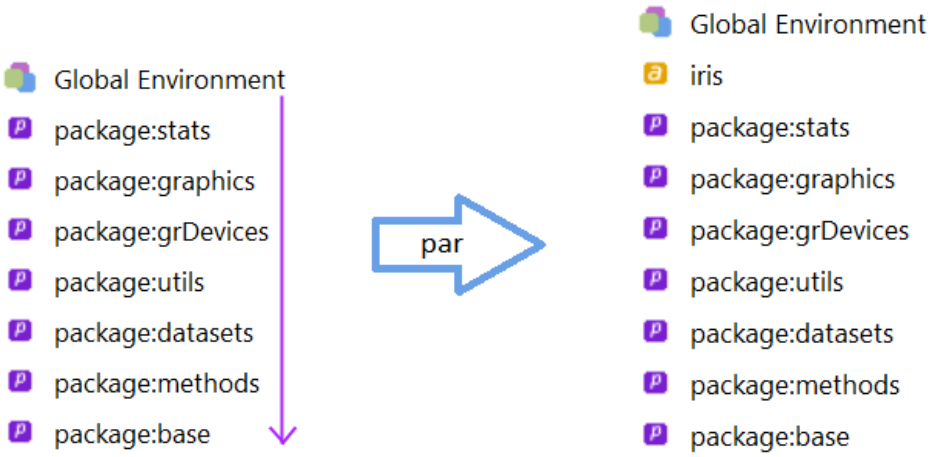
```
##     height weight weight2
## 1 4.833333    115   13225
## 2 4.916667    117   13689
## 3 5.000000    120   14400
## 4 5.083333    123   15129
## 5 5.166667    126   15876
## 6 5.250000    129   16641
```

La fonction `attach`

Certains d'entre vous connaissent peut-être la fonction `attach`. Cette fonction permet d'ajouter un data frame dans le chemin de recherche, juste **en dessous** de l'environnement de travail. Par exemple, la commande

```
attach(iris)
```

modifie le chemin suivant



Je ne recommande pas l'utilisation de `attach`, car si l'environnement de travail contient déjà un objet portant le même nom qu'une colonne du data frame attaché, c'est l'objet dans l'environnement de travail qui sera retourné plutôt que la colonne du data frame si nous utilisons ce nom dans nos instructions. Ce comportement s'explique par le fait que l'environnement de travail demeure le point de départ du chemin de recherche avec `attach`, alors qu'avec `with` c'est le data frame qui est le point de départ du chemin de recherche.

Pour retirer du chemin de recherche un data frame attaché, il faut utiliser la fonction `detach`.

```
detach(iris)
```

L'opérateur `::`

Plutôt que de laisser R parcourir tout son chemin de recherche pour retrouver un objet dans un environnement associé à un package, il est possible d'indiquer à R dans quel environnement chercher un objet avec l'opérateur `::` (double deux-points). Cet opérateur est utile lorsque des fonctions sont masquées par un nouveau package chargé.

Par exemple, le package `dplyr` contient des fonctions portant le même nom que des fonctions des packages `stats` et `base`.

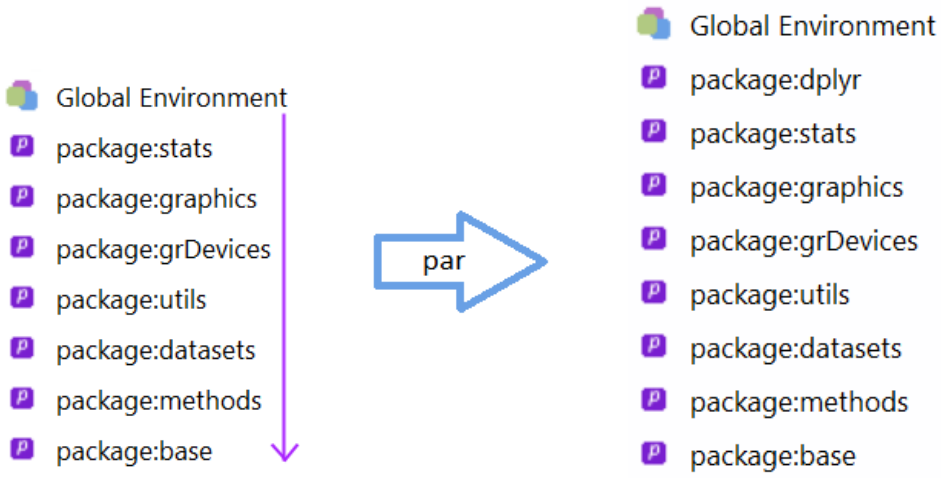
```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Le chargement d'un nouveau package ajoute un environnement dans le chemin de recherche, juste en dessous de l'environnement de travail. Ainsi, la commande `library(dplyr)` a modifié le chemin suivant



Un appel à un des noms de fonction en double, par exemple à `union`, va utiliser par défaut la fonction `union` du package `dplyr` puisque l'environnement de ce package est plus haut dans le chemin de recherche que l'environnement du package `base`.

```
union  # remarquez le nom de l'environnement de provenance de la fonction
```

```
## function (x, y, ...)
## UseMethod("union")
## <bytecode: 0x000000001889fff0>
## <environment: namespace:dplyr>
```

Pour accéder à la fonction `union` du package `base` avec ce chemin d'accès, il faut utiliser l'opérateur `::` comme suit.

```
base::union  # remarquez le nom de l'environnement de provenance de la fonction
```

```
## function (x, y)
## unique(c(as.vector(x), as.vector(y)))
## <bytecode: 0x0000000014176be0>
## <environment: namespace:base>
```

Notez que les auteurs du package `dplyr` ont délibérément repris des noms de fonctions du R de base. Il s'agit de versions de ces fonctions qu'ils jugent meilleures.

Un package peut être retiré du chemin de recherche avec la fonction `detach` comme suit :

```
detach("package:dplyr")
```

Notons que l'utilisation d'une fonction d'un package via l'opérateur `::` ne requière pas que ce package soit chargé dans la session R. Il a simplement besoin d'être installé pour la version de R utilisée. Ainsi, même si nous venons de retirer le package `dplyr` du chemin de recherche, nous pouvons encore accéder aux fonctions qu'il contient en utilisant l'opérateur `::`, comme le prouve cet exemple.

```
arrange
```

```
## Error in eval(expr, envir, enclos): object 'arrange' not found
```

```
dplyr::arrange
```

```
## function (.data, ...)
## {
##     UseMethod("arrange")
## }
## <bytecode: 0x00000000186594b0>
## <environment: namespace:dplyr>
```

Complètement automatique

Ceux qui utilisent `attach`, mentionné précédemment, le font souvent pour que le code soit moins long à taper. Si c'est votre cas, vous pourriez aimer utiliser les capacités de complètement automatique (en anglais *completion* ou *autocompletion*) d'un éditeur de code R. Cette fonctionnalité a justement pour but d'aider l'utilisateur à programmer plus rapidement, tout en diminuant les risques d'erreur de frappe.

Comme il a déjà été mentionné dans les notes sur concepts de base en R, RStudio offre du complètement automatique. Lorsque l'utilisateur saisit une commande en RStudio, que ce soit dans la console R, dans un script R ou même dans un bloc de code R à l'intérieur du document R Markdown, une fenêtre de complètement automatique s'ouvrent souvent automatiquement pour suggérer des suites à la commande. Par exemple, si nous tapons le nom d'un data frame suivi de l'opérateur `$`, la fenêtre de complètement automatique contient tous les noms des colonnes du data frame. Il suffit de sélectionner la colonne désirée avec les flèches, puis d'enfoncer la touche Enter ou Tab, pour que R s'occupe de compléter notre commande.

Les fenêtres de complètement automatique peuvent aussi être ouvertes avec la touche Tab. S'il y a seulement une possibilité de complètement, la touche Tab ne fait pas qu'ouvrir la fenêtre, elle complète aussi la commande.

Plus d'informations sur les capacités de complètement automatique de RStudio peuvent être trouvées sur la page web suivante : https://support.rstudio.com/hc/en-us/articles/205273297-Code-Completion

---

Résumé

Imprécision numérique

R ne garde pas en mémoire de façon tout à fait exacte :

- certains nombres décimaux et
- les nombres plus grands, en valeur absolue, qu'une certaine limite.

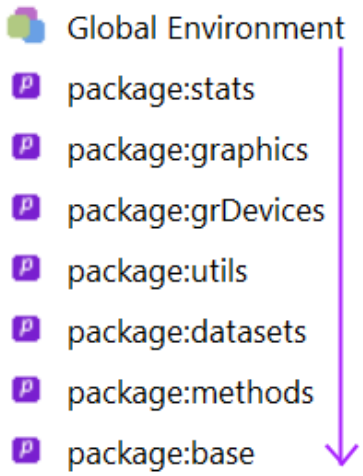
⟶ Attention aux comparaison avec `==`, `identical` ou `unique` : elles sont exactes.

Les comparaison avec `all.equal`, `table` et `factor` sont plus souples.

⟶ Attention aux très petits nombres en valeur absolue : ils peuvent être ramenés à zéro par R.

Évaluation d'expressions en R et environnements

Chemin de recherche parcouru par R pour évaluer les expressions soumises dans la console :



Environnement = groupe d'objets

Ajouter un jeu de données **au début** du chemin de recherche :

- argument `data` venant avec un argument `formula`,
- fonctions `subset` et `transform` (déjà vues),
- fonctions du `tidyverse` possédant un argument `data` ou `.data`,
- fonction `with` et `within`.

Ajouter un environnement provenant d'un package dans le chemin de recherche (en d'autres mots, charger un package) : `library`.

- L'environnement du package est ajouté en 2^{e} position, soit sous l'environnement de travail.

`::` = opérateur pour identifier l'environnement où aller chercher un objet plutôt que de laisser R effectuer une recherche dans tous les environnements ouverts.

Références

- Imprécision numérique en R : Burns, P. (2011). The R inferno, chapitre 1. http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
- Représentation en virgules flottantes :
  - http://fr.wikipedia.org/wiki/Virgule_flottante
  - https://en.wikipedia.org/wiki/Double-precision_floating-point_format
- Évaluation d'expressions en R : http://blog.obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/
- Environnements : Wickham, H. (2019). *Advanced R. Second Edition.* Chapman and Hall/CRC. Chapitre 7. https://adv-r.hadley.nz/environments.html
- R Core Team (2019). The R Language Definition. https://cran.r-project.org/doc/manuals/r-release/R-lang.html