Informations techniques concernant R

Sophie Baillargeon, Université Laval

2018-03-06

Table des matières

Imprécision numérique

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

- les nombres avec une partie décimale et
- les nombres entiers supérieurs à 2^53.

R comporte une imprécision numérique.

Nombres avec une partie décimale

Voici quelques exemples de tours que l'imprécision numérique peut nous jouer, tirés de Burns, P. (2011). The R inferno, chapitre 1. URL : 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 est réellement 0.1, soit 1/10, mais en fait R a stocké en mémoire la valeur suivante :

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

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

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

```r
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.

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

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

Différences de comportement entre `table` et `unique`

La fonction `table` calcule des fréquences. Si elle reçoit en entrée un vecteur numérique, elle commence par trouver les valeurs uniques dans ce vecteur, puis elle compte combien de fois chacune des valeurs uniques est répétée dans le vecteur. Cependant, l'exemple suivant nous montre qu'elle ne trouve pas toujours les mêmes valeurs uniques que la fonction `unique`.

```r
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
```

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

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

```r
table(x)
```

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

```r
unique(x)
```

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

```r
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.

Il semble donc que `table` (ainsi que `xtabs` et `ftable`) 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` et `table` (ainsi que les autres fonctions de calcul de fréquences du R de base) ne se préoccupent pas de différences inférieures à une certaine tolérance.

Aussi, l'imprécision numérique sur des nombres avec une partie décimale implique que R n'arrive pas à garder en mémoire des nombres vraiment très petits. Ces nombres sont stockés comme s'ils étaient des zéros. Lorsque nous travaillons avec de tels nombres, il est bon de les changer d'échelle afin de les rendre plus gros. 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.

Références pertinentes sur la limite de la représentation en virgules flottantes :

- http://fr.wikipedia.org/wiki/Virgule_flottante
- http://en.wikipedia.org/wiki/Double_precision
- `help(double)`

Nombres entiers très grands

Il faut aussi se méfier lorsque l'on travaille avec des entiers très grands.

```
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 entier représentable en R est 2^53. Quel est ce nombre ?

```
2^53
```

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

pour le voir en notation fixe plutôt que scientifique :

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

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

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

```
9007199254740993
```

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

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

```
9007199254740994
```

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

Mais pas ceci :

```
9007199254740995
```

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

L'explication à cela est que pour des nombres entiers au-delà de 2^53, R a des problèmes à stocker correctement le nombre en mémoire. Il se met à faire des arrondissements (arrive seulement à écrire en mémoire des nombres pairs).

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 excellente 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 suivante 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.

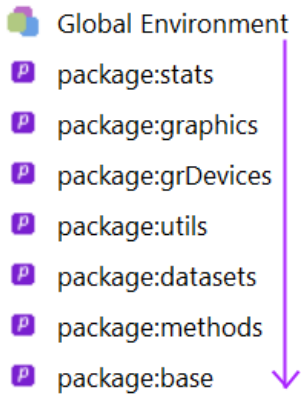


FIGURE 1 – Chemin de recherche parcouru par R pour évaluer les expressions soumises dans la console

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` ou `transform`

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, pas 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, 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. Ainsi, si nous voulons par exemple extraire du jeu de données `iris` 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.

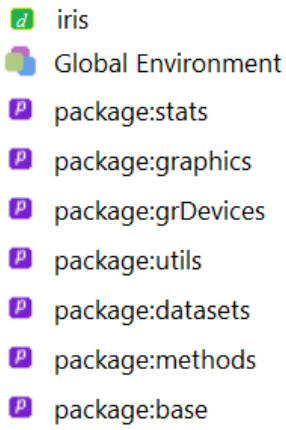


FIGURE 2 – Chemin de recherche parcouru par R pour évaluer les expressions données en argument à la fonction subset dans l'exemple

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 = monDataFrame, exp = expression)`
où `expression` est une commande R, ou encore plusieurs commandes R encadrées de crochets.

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

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

```
# équivalent à :
with(iris, {
  var_catego <- cut(Sepal.Length, right = FALSE,
                    breaks = c(-Inf, quantile(Sepal.Length, probs = c(1/3, 2/3)), Inf))
  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`, plus besoin des `iris$`.

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.

```
head(women)
```

```
##   height weight
## 1     58    115
## 2     59    117
## 3     60    120
```

```
## 4     61    123
## 5     62    126
## 6     63    129
```

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

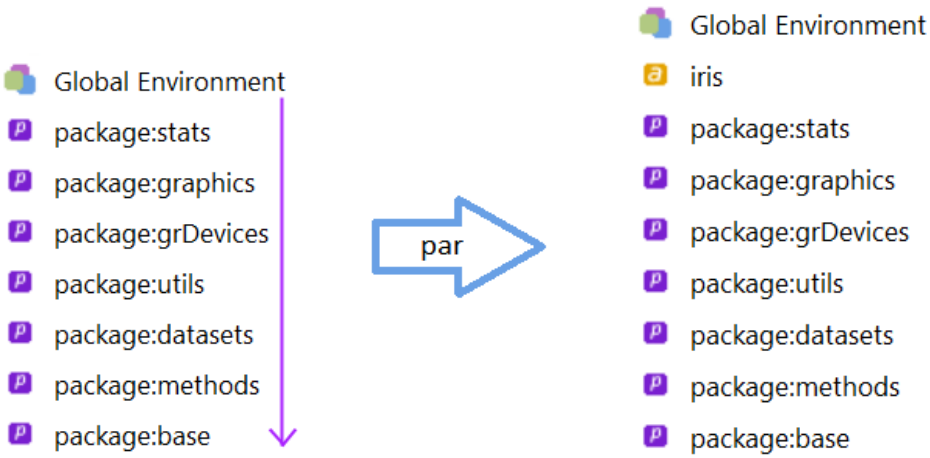
```
##     height weight weight2
## 1 4.833333    115   13225
## 2 4.916667    117   13689
## 3 5.000000    120   14400
## 4 5.083333    123   15129
## 5 5.166667    126   15876
## 6 5.250000    129   16641
```

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



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 on utilise 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 `::`. 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`.

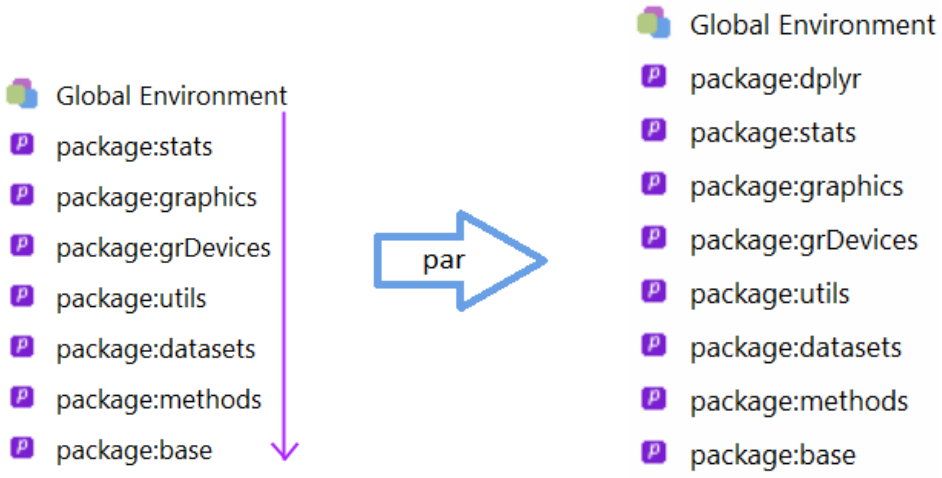
```
library(dplyr)
```

```
##
## 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



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
```

```
## function (x, y, ...)
## UseMethod("union")
## <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
```

```
## function (x, y)
## unique(c(as.vector(x), as.vector(y)))
## <bytecode: 0x0000000015cb72f8>
## <environment: namespace:base>
```

Notez que l'auteur du package `dplyr` a délibérément repris des noms de fonctions du R de base. Il s'agit de versions de ces fonctions qu'il juge meilleures.

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

```
detach("package: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* or *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.

Par exemple, en RStudio, lorsque l'utilisateur saisit une commande dans un script R ou dans la console R, 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 `$`, 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 R Studio peuvent être trouvées sur la page web suivante : https://support.rstudio.com/hc/en-us/articles/205273297-Code-Completion