Aide-mémoire pour IFT-1902

Gabriel Crépeault-Cauchon

17 décembre 2017

Table des matières

1 Introduction

Section plus qualitative à compléter plus tard

2 Terminal

2.1 Commandes de base

Voici quelques commandes essentielles à connaître pour naviguer terminal Bash :

Important : certaines fonctions du *Shell* demandent un espace, d'autre des tirets.

- `cd` pour *change directory* (nous permet de changer de dossier dans le terminal) ;
- `pwd` pour savoir le chemin d'accès dans lequel on se trouve en ce moment ;
- `ls` pour faire apparaître la liste de tous les fichiers dans le dossier actuel ;
- `ls -a` fait apparaître **tous les dossiers**, même ceux qui commence par un point (exemple, *.Renviron*) ;
- `touch` Créer un fichier de texte brut ;
- `rm`permet
- `mkdir` Créer un dossier dans le répertoire où l'on se trouve ;
- `rmdir` Supprimer un dossier dans le répertoire où l'on se trouve ;
- `mv <nomfichier> <destination>` : dépalacer un fichier

2.2 Fonctions utiles du terminal et les Regex

2.2.1 Expressions régulières (*Regex*)

Voici un aide-mémoire sur les expressions régulières

Expression	définition	Exemples
`*`	Cherche 1 ou plusieurs occurences du caractère précédent	`ga*` va trouver ga, g et gaaaa
`?`	Cherche 1 ou 0 occurences du caractère précédent	`ga` va trouver ga et g, mais pas gaaaa
`+`	Cherche 1 ou plusieurs du caractère précédent	`ga+` va trouver
strut	Quand on cherche un caractère spécial (utilisé dans les *Regex*)	`Hungry\?` va trouver Hungry ?
`.`	Cherche n'importe quel caractères	`ga.` va trouver gab, garage, gabon, etc.
`( )`	cherche une chaîne de caractères	
`[ ]`	cherche parmi une liste de caractères	`[gb]ateaux` va trouver gateaux et bateaux
`|`	Va chercher la chaîne de caractères avant ou après le symbole	`(lun)|(mar)di` va trouver lundi et mardi
`{ }`	Spécifie le nombre (consécutif) d'occurence	

Expression	définition	Exemples
`^`	le caractère doit se trouver au début de la ligne	`^http` nous permet de trouver des URL
`$`	le caractère doit terminer la ligne	`(.com)$` nous permet de trouver les adresses internet se terminant par .com

2.2.2 `grep`

2.2.3 `awk`

```
awk -F "," '{print $1 " " $5}' IAG.TO.csv
```

```
## Date Close
## 2017-11-17 59.990002
## 2017-11-20 59.900002
## 2017-11-21 60.330002
## 2017-11-22 60.189999
## 2017-11-23 59.700001
## 2017-11-24 59.669998
## 2017-11-27 59.380001
## 2017-11-28 59.049999
## 2017-11-29 59.139999
## 2017-11-30 60.169998
## 2017-12-01 59.889999
## 2017-12-04 60.270000
## 2017-12-05 60.279999
## 2017-12-06 59.860001
## 2017-12-07 59.459999
## 2017-12-08 60.000000
## 2017-12-11 59.570000
## 2017-12-12 59.650002
## 2017-12-13 59.779999
## 2017-12-14 58.820000
## 2017-12-15 58.980000
```

2.3 Utiliser Git sur le terminal

2.3.1 Configuration

À la première utilisation de git via le terminal *Bash* il faut configurer quelques informations :

- `git config --global user.name "<Nom>"` Configurer son nom tel qu'on désire qu'il apparaisse sur Git.
- `git config --global user.email "<courriel>"` Configurer l'adresse courriel associée.
- `git config --global core.editor open` Si vous oubliez de préciser une description lors d'un *commit* (sera vu à la prochaine section), il va simplement ouvrir un fichier texte brut.
- `git config --list` Juste pour valider que les informations entrées ci-dessus sont enregistrées adéquatement.

2.3.2 Collaborer sur un projet

Pour faire le suivi des versions d'un projet informatique, il est utile d'utiliser Git. Voici un résumé des fonctions (du terminal) à savoir utiliser :

- `git init` : Créer un répertoire (*repository*) *Git* Dans le dossier actuel. *Astuce* : il est plus simple de créer son *repository* directement sur Github puis le cloner dans le dossier désiré ;
- `git clone <https://...>` *Cloner* (ou si on préfère, télécharger) un répertoire Git dans le dossier actuel. **Attention**, on va cloner une seule fois un répertoire, car par la suite on va *pull* les modifications du dépôt ;
- `git pull` commandes qu'on utilise seulement si on a déjà cloné le répertoire. Nous permet d'avoir les mises à jour ;
- `git status` permet de voir si il y a des fichiers dans notre dossier qui n'apparaissent pas (ou que les modifications n'apparaissent pas) sur le répertoire Git. Si c'est le cas, elles seront affichés en **rouge**.
- `git add` pour ajouter les modifications dans le dépôt. Après cette étape, on doit confirmer nos modifications par la commande *commit*
  - Si on utilise `git add -A`, tous les fichiers seront ajoutés au prochain *commit*
- `git commit -m "<description de la modif.>"` On confirme notre modification au travail et on décrit *très brièvement* ce qu'on a fait comme modification
- `git push` *pousser* au serveur les modifications qu'on a fait. Après cette étape, si on va sur Github, nos modifications apparaîtront.

3 Programmation en R

3.1 Commandes et expressions de R

3.1.1 Commandes de base

- `save image()` Si on veut sauvegarder l'espace de travail et son environnement. **Rarement utilisée**, sauf si on veut sauvegarder la valeur d'une variable (qui est longue à obtenir)
- `getwd()` obtenir le répertoire de travail dans lequel on se trouve actuellement
- `setwd(<chemin d'accès>)` Changer le répertoire de travail actuel
- `help()` obtenir de l'aide sur une fonction ou une commande en particulier. On peut aussi accéder au manuel d'instruction de R avec `help.start`
- `ls()` : voir tous les objets de l'environnement global
- `rm()` : pour supprimer un objets
- `rm(list = ls())` : supprimer tous les objets dans l'environnement global

3.1.2 Opérateurs de R

Opérateur	Fonction
`$`	extraction d'une liste
`[]`	indiçage
`^`	puissance
`-`	changement de signe
`:`	génération d'une suite
`%*% %% %/%`	produit matriciel, modulo, division entière
`* /`	multiplication, division
`+ -`	addition, soustraction

Opérateur	Fonction
< <= == >= > !=	plus petit, plus petit ou égal, égal, plus grand ou égal, plus grand, différent de
!	négation logique
&	ET logique
|	OU logique
<-	affectation (méthode la plus utilisée)

3.1.3 Création d'une suite ou séquence

Séquence de chiffres

```
seq(from = 10, to = 20, by = 2)
```

```
## [1] 10 12 14 16 18 20
```

```
x <- c(1,2,3,8)
seq(x)
```

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

```
seq_len(5)
```

```
## [1] 1 2 3 4 5
```

```
y <- c(10,14,3,2)
seq_along(y)
```

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

Échantillon de données aléatoire

```
x <- sample(1:5, size = 8, replace = TRUE, prob = c(0.1,0.2,0.2,0.25,0.25)) ; x
```

```
## [1] 2 5 3 3 3 1 4 2
```

Séquence de lettres

```
x <- c(1,2,3)
letters[x]
```

```
## [1] "a" "b" "c"
```

```
x <- c(24,25,26)
LETTERS[x]
```

```
## [1] "X" "Y" "Z"
```

3.1.4 Information sur un objet de R

- `mode()` : Mode d'un objet
- `length()` : Longueur d'un objet
- `nchar()` : nombre de caractères
- `class()` : classe d'un objet
- `summary()` : beaucoup d'information sur l'objet

3.2 Objets de R

3.2.1 Mode d'un objet

Mode	Contenu de l'objet
`numeric`	nombres réels
`complex`	nombres complexes
`logical`	valeurs booléennes
`character`	chaînes de carctères
`function`	fonction
`list`	liste
`expression`	expressions non évaluées

Si on créé un vecteur qui contient des données de plus d'un mode, il va convertir les autres données dans le mode le plus «puissant», en respectant l'ordre suivant :

1. `list`
2. `character`
3. `numeric`
4. `logical`

```
a <- c(TRUE, "test",1:2,list(1)) ; a
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] "test"
##
## [[3]]
## [1] 1
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 1
```

```
mode(a)
```

```
## [1] "list"
```

```
b <- c(FALSE,0:2,"test") ; b
```

```
## [1] "FALSE" "0"     "1"     "2"     "test"
```

```
mode(b)
```

```
## [1] "character"
```

```
c <- c(FALSE,1:2) ; c
```

```
## [1] 0 1 2
```

```
mode(c)
```

```
## [1] "numeric"
```

```r
x <- c(TRUE,1,4,"GAB")
class(x)
```

```
## [1] "character"
```

```r
mode(x)
```

```
## [1] "character"
```

3.2.2 NA, NaN, NULL et Inf

```r
0/0
```

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

```r
Inf/Inf
```

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

```r
Inf-Inf
```

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

```r
1/0
```

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

```r
Inf
```

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

```r
Inf^Inf
```

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

```r
5 + NULL
```

```
## numeric(0)
```

3.3 Vecteurs

3.3.1 Création d'un Vecteur

```r
x <- c(1,2,3) ; x
```

```
## [1] 1 2 3
```

```r
y <- vector(mode = "numeric", length = 5) ; y
```

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

3.3.2 Ajout d'étiquettes au vecteur

il existe 2 façon :

```r
x <- c(1,2)
names(x) <- c("a","b") ; x
```

```
## a b
## 1 2
```

ou bien

```
x <- c(a=1, b=2) ; x
```

```
## a b
## 1 2
```

3.3.3 Indiçage du vecteur

```
x <- c(A = 1, B = 2, C = 3, D = 4, E = TRUE, G = NA) ; x
```

```
##  A  B  C  D  E  G
##  1  2  3  4  1 NA
```

```
x[3]            # 3e élément
```

```
## C
## 3
```

```
x[c(1,5)]       # 1er et 5e élément
```

```
## A E
## 1 1
```

```
x[-3]           # tout sauf le 3e élément
```

```
##  A  B  D  E  G
##  1  2  4  1 NA
```

```
x[-c(3,4)]      # tous sauf le 3e et 4e élément
```

```
##  A  B  E  G
##  1  2  1 NA
```

```
x[!is.na(x)]    # tout ce qui n'est pas NA
```

```
## A B C D E
## 1 2 3 4 1
```

```
x[x<4]          # tous les éléments qui sont <4
```

```
##    A    B    C    E <NA>
##    1    2    3    1   NA
```

```
x[c("G","B")]   # seulement les éléments taggé "G" et "B"
```

```
##  G  B
## NA  2
```

```
x[]             # tous les éléments
```

```
##  A  B  C  D  E  G
##  1  2  3  4  1 NA
```

3.3.4 Opérations sur les vecteurs

3.4 Fonctions

3.4.1 Quelques fonctions déjà programmées dans R

3.4.1.1 Statistiques

Il existe plusieurs fonctions déjà installées pour des calculs statistiques simples :

```
x <- sample(1:20, 10, replace=T)
y <- sample(1:30, 10, replace = T)
sum(x)
```

```
## [1] 89
```

```
mean(x)
```

```
## [1] 8.9
```

```
var(x)
```

```
## [1] 39.43333
```

```
cov(x,y)
```

```
## [1] 2.055556
```

Lorsqu'on a un vecteur ou une matrice, on peut utiliser des fonctions qui font certains calculs par ligne ou par colonne :

```
tableau <- matrix(1:6, nrow = 3, ncol = 2, byrow = T)
rowSums(tableau)
```

```
## [1]  3  7 11
```

```
rowMeans(tableau)
```

```
## [1] 1.5 3.5 5.5
```

```
colSums(tableau)
```

```
## [1]  9 12
```

```
colMeans(tableau)
```

```
## [1] 3 4
```

3.4.2 Fonctions d'applications

Lorsqu'on travaille avec des *data frame* ou bien des tableaux dans R, on est mieux d'utiliser des fonctions d'applications (plutôt que des boucles qui ralentissent le temps d'éxécution d'une fonction).

- le premier argument est la vecteur ou le tableau sur lequel on veut *appliquer* la fonction
- le 2e argument est la dimension (1 = ligne, 2= colonne, etc.) sur laquelle on veut appliquer itérativement la fonction
- La fonction désirée sur chaque ligne ou colonne est donnée en 3e argument
- si la fonction a besoin de plusieurs arguments de spécifiés, on peut les spécifier par la suite en 4e, 5e argument etc...

Exemples

```r
apply(tableau, 1, sum)    # fait la somme des lignes
```

```
## [1]  3  7 11
```

```r
apply(tableau, 2, mean)    # idem pour colonne
```

```
## [1] 3 4
```