FACULTÉ DES SCIENCES ECONOMIQUES

UNIVERSITÉ DE RENNES 1

M1 Statistique & Économétrie

Ewen Gallic [1]
http://egallic.fr

2015

Logiciel R et programmation

Exercices

Partie 2 : Fonctions

Exercice 1 (création d'une fonction)

1. Créer une fonction nommée `somme_n_entiers` qui retourne la somme des n premiers entiers. Son seul paramètre sera `n` ;

```
# Retourne la somme des n premiers entiers
# @n : (int)
somme_n_entiers <- function(n){
  sum(seq_len(n))
}# Fin de somme_n_entiers()
```

2. Utiliser la fonction `somme_n_entiers()` pour calculer la somme des 100 premiers entiers ;

```
somme_n_entiers(100)
```

3. Terminer la fonction par l'assignation du résultat dans un objet nommé `res`, puis évaluer l'expression suivante : `somme_n_entiers(100)`. Que peut-on constater ?

```
# Retourne la somme des n premiers entiers
# @n : (int)
somme_n_entiers <- function(n){
  res <- sum(seq_len(n))
  }# Fin de somme_n_entiers()

somme_n_entiers(100)
```

1. ewen.gallic[at]gmail.com

```r
a <- somme_n_entiers(100)
a
```

4. Charger les données `diamonds` du *package* `ggplot2` dans la session `R` à l'aide de l'expression suivante :

```r
data(diamonds, package = "ggplot2")
```

Créer une fonction que l'on appellera `prix_diamant_coupe()`, qui, quand on lui fournit la valeur de la coupe du diamant sous forme de caractères (`Fair`, `Good`, `Very Good`, `Premium`, ou `Ideal`), filtre le tableau de données `diamonds` pour ne conserver que les observations pour lesquelles la coupe du diamant correspond à celle indiquée en paramètre, et retourne le prix moyen des observations de la base ainsi filtrée ;

```r
# Retourne le prix moyen des diamants, pour une coupe donnée
# @diamond_cut: (string) coupe du diamant
# (Fair, Good, Very Good, Premium, Ideal)
prix_diamant_coupe <- function(diamond_cut){
  df <-
    diamonds %>%
    filter(cut %in% diamond_cut)
  mean(df$price)
}# Fin de prix_diamant_coupe()

prix_diamant_coupe("Fair")
prix_diamant_coupe("Good")
```

5. Reprendre le code de la fonction précédente, et le modifier pour retourner à présent une liste de deux éléments : (i) la moyenne des prix et (ii) l'écart-type ;

```r
# Pour une coupe de diamant donnée (diamond_cut), retourne une liste :
# - la moyenne des prix
# - l'écart-type des prix
# @diamond_cut: (string) coupe du diamant
# (Fair, Good, Very Good, Premium, Ideal)
resume_diamant_coupe <- function(diamond_cut){
  df <-
    diamonds %>%
    filter(cut %in% diamond_cut)
  list(moyenne = mean(df$price), ecart_type = sd(df$price))
}# Fin de resume_diamant_coupe()

resume_diamant_coupe("Fair")
resume_diamant_coupe("Good")
```

6. Créer la fonction `resume_diamant_coupe_couleur()`, qui pour une coupe et une couleur de diamant données, retourne une liste de deux éléments : (i) la moyenne des prix et (ii) l'écart-type pour les diamants possédant cette coupe et cette couleur (la couleur du diamant est une lettre allant de `J` pour les pires, à `D` pour les meilleurs). Tester la fonction pour la coupe `Fair` et la couleur `D` ;

```r
# Pour une coupe et une couleur de diamant données
# (diamond_cut et color resp.),
# retourne une liste comprenant :
# - la moyenne des prix
# - l'écart-type des prix
# @diamond_cut: (string) coupe du diamant
# (Fair, Good, Very Good, Premium, Ideal)
# @:diamond_color: (string) couleur du diamant
# (de J (pire) à D (meilleure))
resume_diamant_coupe_couleur <-
  function(diamond_cut, diamond_color){
  df <-
    diamonds %>%
    filter(cut %in% diamond_cut,
           color %in% diamond_color)
  list(moyenne = mean(df$price), ecart_type = sd(df$price))
}# Fin de resume_diamant_coupe_couleur()

resume_diamant_coupe_couleur(diamond_cut = "Fair", diamond_color = "D")
```

7. Reprendre la fonction précédente, et lui attribuer la valeur `D` (en chaîne de caractères) comme paramètre effectif pour la couleur. Tester alors l'appel à la fonction en précisant :

   (a) la coupe `Fair` et la couleur `D`,

   (b) la coupe `Fair`, mais pas de paramètre pour la couleur,

   (c) la coupe `Fair` et la couleur `E`,

   (d) la coupe non précisée mais la couleur `E`;

```r
# Pour une coupe et une couleur de diamant données
# (diamond_cut et color resp.),
# retourne une liste comprenant :
# - la moyenne des prix
# - l'écart-type des prix
# @diamond_cut: (string) coupe du diamant
# (Fair, Good, Very Good, Premium, Ideal)
# @:diamond_color: (string) couleur du diamant
# (de J (pire) à D (meilleure))
resume_diamant_coupe_couleur_2 <-
  function(diamond_cut, diamond_color = "D"){
  df <-
    diamonds %>%
    filter(cut %in% diamond_cut,
           color %in% diamond_color)
  list(moyenne = mean(df$price), ecart_type = sd(df$price))
}# Fin de resume_diamant_coupe_couleur_2()

# Infos pour les diamants avec la coupe Fair et la couleur D
resume_diamant_coupe_couleur_2(diamond_cut = "Fair", diamond_color = "D")

# Idem (par défaut, on a indiqué que la couleur vaut D)
resume_diamant_coupe_couleur_2(diamond_cut = "Fair")

# Infos pour les diamants avec la coupe Fair et la couleur E
resume_diamant_coupe_couleur_2(diamond_cut = "Fair", diamond_color = "E")

# Pas de valeur par défaut pour la coupe, paramètre manquant, donc erreur
resume_diamant_coupe_couleur_2(diamond_color = "E")
```

8. Soit le code suivant :

```r
# @x : (int)
f_test <- function(x){
  x^2
}# Fin de f_test()

# @x : (int)
f_test_2 <- function(y){
  x^2
}# Fin de f_test_2()

# @x : (int)
f_test_3 <- function(y){
  x <- y
  x^2
}# Fin de f_test_3()


x <- 3
```

Expliquez ce qui se passe dans chacun des cas suivants :

```r
f_test(x = 2)
```

On évalue la fonction `f_test()` en fournissant au paramètre `x` la valeur `2`. `R` va donc retourner le carré de 2. La valeur de l'objet appelé `x` dans l'environnement global est ignorée à l'intérieur de la fonction `f_test()`, puisqu'un objet du même nom est défini dans un environnement plus proche (à l'intérieur de la fonction).

```
f_test_2(y = 2)
```

La fonction `f_test_2()` retourne le carré de l'objet `x`. Nous lui fournissons une valeur de 2, mais au paramètre nommé `x`. Aussi, comme aucun objet appelé `x` n'est défini à l'intérieur de la fonction, `R` va chercher dans un niveau supérieur s'il trouve un objet appelé `x`. Il en trouve un dans l'environnement global, dont le contenu est `3`. Aussi, le résultat de l'évaluation du code retourne la valeur `9`.

```
f_test_2()
```

Comme le paramètre `y` de la fonction `f_test_2()` n'est pas utilisé dans le corps de la fonction, `R` l'ignore et retourne le carré de l'objet `x` défini dans l'environnement global comme dans l'exemple précédent.

```
f_test_3(4)
x
```

Cette fois, dans le corps de la fonction `f_test_3()`, la valeur du paramètre `y` est assignée à un objet vivant à l'intérieur de la fonction nommé `x`. Aussi, le carré de cette variable (et donc de la valeur fournie au paramètre `y`) est retourné.

Exercice 2 (création d'une fonction, traitement de chaînes de caractères)

Supposons que les adresses e-mails des étudiants de l'Université de Rennes 1 sont constituées de la manière suivante : le prénom et le nom de famille séparés par un point, le symbole arobase et le enfin le nom de domaine. Supposons de plus que les étudiants ont un seul prénom, et aucune particule au nom de famille. La syntaxe des adresses e-mail est donc comme suit : `nom.prenom@etudiant.univ-rennes1.fr`.

```
emails <- c("marie.petit@etudiant.univ-rennes1.fr",
            "jean.dupont@etudiant.univ-rennes1.fr",
            "isabelle.martinez@etudiant.univ-rennes1.fr",
            "pierre.moreau@etudiant.univ-rennes1.fr")
```

Créer une fonction, qui à partir d'**une** adresse e-mail d'un étudiant, retourne un `data.frame` contenant trois variables : le prénom, le nom et l'adresse e-mail de cet étudiant.

```r
library(stringr)
# A partir d'une adresse etudiant, retourne un data.frame contenant :
# - prenom : le prenom de l'etudiant
# - nom : le nom de l'etudiant
# - email : l'email de l'etudiant
parse_email <- function(email){
  nom_prenom <- str_split(email, "@")[[1]][1]
  nom_prenom <- str_split(nom_prenom, "\\.")[[1]]
  prenom <- nom_prenom[1]
  nom <- nom_prenom[2]
  data.frame(prenom = prenom, nom = nom, email = email)
}
parse_email(email[1])
```