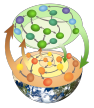

Séance 3: Fonctions

https://econumuds.github.io/BIO109/cours3/

Dominique Gravel
Laboratoire d'écologie intégrative

Retour sur l'exercice

Exercice de fin de séance

Le fichier **quadrats.csv** est un sommaire de données individuelles, où la présence de chaque espèce est mesurée. Ces données se trouvent dans **arbres.csv**. Dans le cadre de ce projet, on s'intéresse à la distribution de l'érable à sucre et des autres espèces tout au long du gradient d'élévation de la parcelle. Pour cet exercice, on vous demande de:

1. Charger les données "arbres"
2. Délimiter cinq zones au sein du gradient d'élévation : 0-200m, 201-400m, 401-600m, 601-800m, 801-1000m
3. Pour chacune de ces zones, calculer le nombre de tiges de chaque espèce
4. Enregistrer les résultats dans un tableau avec 5 rangées et S colonnes

On vous demande de rédiger un script qui réalisera l'ensemble de ces étapes, de la lecture des données à l'enregistrement du tableau final.

Solution

```
# Créer un tableau où on enregistre les résultats
resultats <- matrix(nr = 5, nc = 7)

# Lire le fichier (en assumant que vous êtes dans le bon dossier)
arbres <- read.table(file="donnees/arbres.csv", header=TRUE,sep = ";")

# Délimiter une première zone
sub_zoneA <- subset(arbres, arbres$bory < 201)

# Calculer le nombre de tiges
table(arbres$esp)
```

```
##
## abba acpe acsa beal bepa fagr piru
## 2596 1864 3326 3995 2080 2785  989
```

```
# Enregistrer le résultat dans le tableau
resultats[1,] <- table(arbres$esp)
```

Opérations mathématiques de base

Astuce: générer des nombres aléatoires

Il peut souvent être pratique de générer des chiffres au hasard, sur lesquels on souhaite faire des tests. Nous verrons plusieurs exemples au cours 5, mais pour l'instant, prenez note de la fonction suivante:

```
alea <- runif(n = 10, min = 0, max = 1)
alea
```

```
##  [1] 0.4535941 0.3331902 0.9048683 0.4931375 0.1552319 0.2690758 0.1068651
##  [8] 0.2411779 0.7547795 0.3041469
```

Ici la distribution utilisée est la distribution uniforme, toutes les autres distributions en sont dérivées.

Opérateurs de base

R peut faire toutes les opérations mathématiques de base d'une calculatrice :

```
3 + 7
```

```
## [1] 10
```

```
3 - 7
```

```
## [1] -4
```

Opérateurs de base

R peut faire toutes les opérations mathématiques de base d'une calculatrice :

```
3 * 7
```

```
## [1] 21
```

```
3 / 7
```

```
## [1] 0.4285714
```

Opérateurs de base

R respecte la séquence des opérations, tel que démontré par l'exemple suivant :

```
3 + 7 / 2 * 5
```

```
## [1] 20.5
```

```
3 + (7 / 2) * 5
```

```
## [1] 20.5
```

Opérations sur des vecteurs et matrices

Par défaut, le produit de vecteurs et de matrices est *scalaire* :

```
vec1 <- runif(10,0,1)
vec1
```

```
##  [1] 0.8544415 0.3941523 0.6655439 0.1069739 0.7691560 0.6321975 0.1774071
##  [8] 0.8201470 0.9176679 0.7544635
```

```
3*vec1
```

```
##  [1] 2.5633245 1.1824569 1.9966318 0.3209218 2.3074681 1.8965926 0.5322213
##  [8] 2.4604409 2.7530038 2.2633904
```

Opérations sur des vecteurs et matrices

La situation plus compliquée survient lorsque l'on multiplie des vecteurs et des matrices :

```
vec1 <- c(10,20,30)
mat1 <- matrix(c(1:6), nr = 3, nc = 2)
mat1
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
vec1 * mat1
```

```
##      [,1] [,2]
## [1,]   10   40
## [2,]   40  100
## [3,]   90  180
```

Opérations avancées

```
log(100)
```

```
## [1] 4.60517
```

```
log10(100)
```

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

```
exp(10)
```

```
## [1] 22026.47
```

Opérations avancées

```
10^2
```

```
## [1] 100
```

```
sqrt(100)
```

```
## [1] 10
```

Opérations avancées

```
alea = runif(10,0,1)
alea
```

```
##  [1] 0.89029095 0.76598038 0.55277426 0.45673598 0.80244605 0.51259316
##  [7] 0.74202873 0.05310249 0.69550988 0.60267649
```

```
min(alea)
```

```
## [1] 0.05310249
```

```
max(alea)
```

```
## [1] 0.8902909
```

Arrondir

```
pi
```

```
## [1] 3.141593
```

```
floor(pi)
```

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

```
ceiling(pi)
```

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

```
round(pi,4)
```

```
## [1] 3.1416
```

Opérations sur des matrices

Parfois, on souhaite calculer des propriétés sur des colonnes et des rangées:

```
mat <- matrix(runif(4,0,1), nr = 2, nc = 2)
mat
```

```
##           [,1]      [,2]
## [1,] 0.4347471 0.5360362
## [2,] 0.1925214 0.8689882
```

```
rowSums(mat)
```

```
## [1] 0.9707833 1.0615096
```

```
colSums(mat)
```

```
## [1] 0.6272684 1.4050244
```

Opérations sur des matrices

Et de façon plus générale, on peut utiliser la fonction `apply()` qui est très pratique :

```
apply(X = mat, MARGIN = 2, FUN = sum)
```

```
## [1] 0.6272684 1.4050244
```

```
apply(X = mat, MARGIN = 1, FUN = sum)
```

```
## [1] 0.9707833 1.0615096
```

Opérations sur des matrices

Et finalement, bien que vous l'utiliserez peu dans le cours, il est toujours pratique de savoir que la particularité de R d'être un langage vectoriel permet de facilement faire des opérations sur les matrices :

```
vec1 <- c(10,20,30)
mat1 <- matrix(c(1:6),nr = 3, nc = 2)
vec1 * mat1
```

```
##      [,1] [,2]
## [1,]   10   40
## [2,]   40  100
## [3,]   90  180
```

```
vec1 %*% mat1
```

```
##      [,1] [,2]
## [1,]  140  320
```

Exercice

Abondances relatives

Au moyen du fichier **quadrats.csv**, je vous demande de faire les opérations suivantes:

1. Calculer l'abondance totale pour chacune des espèces, leur abondance moyenne et le coefficient de variation de leur abondance ;
2. Calculer le nombre total de tiges pour chaque quadrat ;
3. Transformer l'abondance absolue en abondance relative ;

Abondances relatives

```
quadrats <- read.csv2(file="donnees/quadrats.csv", header=TRUE, stringsAsFactors=FALSE,row.names = 1)
N_total <- apply(quadrats, 2, sum)
N_moy <- apply(quadrats, 2, mean)
N_sd <- apply(quadrats, 2, sd)
N_CV <- N_sd / N_moy
```

Abondances relatives

```
N_quadrats <- apply(quadrats, 1, sum)
quadrats_rel <- quadrats / N_quadrats
head(quadrats_rel)
```

```
##             abba       acpe       acsa       beal       bepa      fagr
## 0-0   0.006024096 0.33132530 0.06626506 0.04216867 0.00000000 0.5542169
## 0-100 0.000000000 0.27777778 0.22222222 0.16666667 0.00000000 0.3333333
## 0-120 0.060606061 0.21212121 0.36363636 0.12121212 0.03030303 0.2121212
## 0-140 0.160000000 0.20000000 0.16000000 0.32000000 0.04000000 0.0800000
## 0-160 0.064516129 0.06451613 0.35483871 0.25806452 0.03225806 0.1935484
## 0-180 0.178571429 0.10714286 0.32142857 0.25000000 0.00000000 0.1071429
##            piru
## 0-0   0.00000000
## 0-100 0.00000000
## 0-120 0.00000000
## 0-140 0.04000000
## 0-160 0.03225806
## 0-180 0.03571429
```

Les fonctions

Qu'est-ce qu'une fonction ?

Une fonction contient une série de commandes (i.e. lignes de code) qui sont exécutées lorsque la fonction est appelée.

Un simple exemple

```
ma_fonction <- function(argument1, argument2) {

  # Ce que l'on veut que la fonction exécute
  resultat <- argument1 * argument2

  # Optionnel. Si l'on veut accéder au résultat de la fonction
  return(resultat)
}
```

Pourquoi utiliser des fonctions ?

1. Répéter une même tâche mais en changeant ses paramètres
2. Rendre votre code plus lisible
3. Rendre votre code plus facile à modifier et à maintenir
4. Partager du code entre différentes analyses
5. Partager votre code avec d'autres personnes
6. Modifier les fonctionalités par défaut de R

La construction d'une fonction

Imaginons que l'on souhaite multiplier deux chiffres (disons, 3 et 7) et les diviser par leur somme.

$$\frac{3 \times 7}{3 + 7}$$

On peut écrire ce calcul directement dans la console comme suit

```
(3*7)/(3+7)
```

```
## [1] 2.1
```

La construction d'une fonction

Si on souhaite faire la même opération pour toutes les pairs de chiffres dans le tableau suivant, comment fait-on ?

```
tableau <- data.frame(x=rnorm(5),y=rnorm(5))
tableau
```

```
##            x          y
## 1  1.00973643  0.7076916
## 2  1.71263865 -0.4847601
## 3 -1.31853533  0.3571794
## 4 -0.21647028  0.1459792
## 5  0.05508215 -0.5965671
```

La construction d'une fonction

À noter qu'en ayant différents chiffres, la formule vue dans la diapositive précédente devient un peu plus générale :

$$\frac{x \times y}{x + y}$$

La construction d'une fonction

L'approche longue, pas efficace, mais qui marche...

```
(tableau[1,1]*tableau[1,2])/(tableau[1,1]+tableau[1,2])
(tableau[2,1]*tableau[2,2])/(tableau[2,1]+tableau[2,2])
(tableau[3,1]*tableau[3,2])/(tableau[3,1]+tableau[3,2])
```

```
## [1] 0.4160768
## [1] -0.6761409
## [1] 0.4898848
```

Problème - Ce n'est vraiment pas pratique si on a beaucoup de données ou si le format du tableau change.

La construction d'une fonction

Une boucle peut sauver du temps...

```
for(i in 1:nrow(tableau)){
  res <- (tableau[i,1]*tableau[i,2])/(tableau[i,1]+tableau[i,2])
  print(res)
}
```

```
## [1] 0.4160768
## [1] -0.6761409
## [1] 0.4898848
## [1] 0.4482861
## [1] 0.06068534
```

Problème - Qu'est-ce qu'on fait si on veut appliquer ce calcul sur plusieurs tableaux ???
C'est possible, mais ça peut être un peu plus compliqué !

La construction d'une fonction

Et si on faisait une fonction...

La fonction permet de résoudre certains problèmes car elle permet d'appliquer une série de commandes (i.e. lignes de codes) à différents types de données. En d'autre mots, la fonction généralise des commandes spécifiques.

La construction d'une fonction

Comment construire une fonction ?

On commence par écrire une version spécifique du code que l'on souhaite **généraliser**.

```
(3*7)/(3+7)
```

La construction d'une fonction

Comment construire une fonction ?

Ensuite, on définit ce code comme faisant parti d'une fonction.

```
prodsum <- function(){
  res <- (3*7)/(3+7)
  return(res)
}
```

Yéééé, on a écrit notre première fonction !! :-)

```
prodsum()
```

```
## [1] 2.1
```

Petit problème

Cette fonction a le défaut de n'être aucunement générale. Elle va toujours donner le même résultat. *:-(*

Comment construire une fonction ?

À noter qu'en utilisant la commande `return()`, on s'assure de renvoyer ce qui se trouve dans l'objet `res` à l'utilisateur.

Une notion importante à avoir lorsqu'on construit une fonction est que tout ce qui se trouve à l'**intérieur** d'une fonction est *entièrement* indépendent de ce qui se trouve à l'**extérieur** d'une fonction.

Par exemple, l'objet `res` à l'extérieur de la fonction **n'existe pas**. Il a un sense uniquement à l'intérieur de la fonction.

Comment rendre la fonction plus générale ?

On peut généraliser cette fonction, en implémentant directement la formule générale présentée précédemment:

$$\frac{x \times y}{x + y}$$

Pour ce faire, il faut ajouter des **arguments** à notre fonction.

Comment rendre la fonction plus générale ?

Les **arguments** peuvent varier selon ce que l'utilisateur souhaite calculer. Il faut donc s'assurer que les mêmes opérations soient réalisées sur ces arguments.

```
prodsum <- function(x,y){
  res <- (x*y)/(x+y)
  return(res)
}
```

Avec cette fonction on peut faire le calcul qui nous intéresse avec différentes séries de chiffres.

Comment rendre la fonction plus générale ?

Autre charactéristique importante des arguments d'une fonction: les objets passés en argument n'ont pas besoin d'avoir le même nom que les arguments. En fait, c'est rarement le cas :

```
a <- 1
b <- 2
prodsum(x = a, y = b)
```

```
## [1] 0.6666667
```

Utilisons notre fonction avec des chiffres

```
prodsum(x = 3, y = 7)
```

```
## [1] 2.1
```

```
prodsum(y = 7, x = 3)
```

```
## [1] 2.1
```

Comme on le constate, dans le language R, les arguments peuvent être définis de deux façons.

- ✓ En suivant l'ordre des arguments
- ✓ En utilisant le noms des arguments

Utilisons notre fonction avec des vecteurs

Le language de programmation R permet de faire aussi le calcul sur des vecteurs :

```
vecA <- tableau[,1]
vecB <- tableau[,2]
prodsum(vecA, vecB)
```

```
## [1]  0.41607680 -0.67614086  0.48988480  0.44828606  0.06068534
```

```
prodsum(tableau[,1], tableau[,2])
```

```
## [1]  0.41607680 -0.67614086  0.48988480  0.44828606  0.06068534
```

Utilisons notre fonction avec des vecteurs

L'exemple précédent fonctionne bien car `vecA` et `vecB` contiennent le même nombre de chiffres, ils ont la même longueur. Que ce passe-t-il si les vecteurs n'ont pas la même longueur ?

```
vec2 <- tableau[1:2,1]
vec3 <- tableau[1:3,1]
vec4 <- tableau[1:4,2]
prodsum(vec3, vec4)
```

```
## Warning in x * y: la taille d'un objet plus long n'est pas multiple de la
## taille d'un objet plus court
```

```
## Warning in x + y: la taille d'un objet plus long n'est pas multiple de la
## taille d'un objet plus court
```

```
## [1]  0.4160768 -0.6761409  0.4898848  0.1275405
```

La construction d'une fonction

Le **recyclage** est une propriété des fonctions mathematiques de base du language R (e.g. `+`, `-`, `*` et `/`). Lorsque deux vecteurs sont de longueurs différentes, les valeurs du vecteur le plus court sont réutilisées, dans l'ordre, pour combler le nombre de valeurs manquantes entre les deux vecteurs. Cette propriété du langage R peut être très pratique mais aussi **générer beaucoup de problèmes**.

Comme on le voit dans l'exemple, un message d'avertissement est envoyé si la longueur du plus petit vecteur n'est pas un multiple du vecteur le plus long. Par contre, si le vecteur le plus court est un multiple du vecteur le plus long, aucun message d'avertissement n'est envoyé et ce même si le résultat n'a pas de sense.

Comment faire pour régler ce problème ?

Ajouter des trappes dans une fonction

Pour régler le problème subtile présenté dans la diapositive précédente, on peut ajouter des conditions de sorties dans la fonction qui retournent un message d'erreur. Voici un exemple :

```
prodsum <- function(x, y){
  if(length(x) != length(y)){
    stop("'x' est de taille différente de 'y'")
  }
  res <- (x*y)/(x+y)
  return(res)
}
```

Les lignes de codes ajoutées mesure la longeur de x et y et lorsqu'ils sont d'une longueur différente, un message d'erreur est envoyé et le reste du code dans la fonction n'est pas évalué.

Ajouter des trappes dans une fonction

La condition de sortie `stop("'x' est de taille différente de 'y'")` permet de corriger le problème mentionné dans la diapositive précédente :

```
prodsum(vec2,vec4)
```

```
## Error in prodsum(vec2, vec4): 'x' est de taille différente de 'y'
```

```
prodsum(vec3,vec4)
```

```
## Error in prodsum(vec3, vec4): 'x' est de taille différente de 'y'
```

Notez que le message d'erreur envoyé est composé par le programmeur. Par contre, il est important que le message d'erreur soit court et précis.

Utilisons notre fonction avec des tableaux

Le language de programmation R permet de faire aussi le calcul sur des matrices

```
tableau2 <- data.frame(x_2=rnorm(5)^2,y_2=rnorm(5)^2)
prodsum(tableau, tableau2)
```

```
##            x        y
## 1  0.11447934 0.4209701
## 2  0.63563452 4.0581913
## 3  0.24325559 0.1896481
## 4 -0.25967075 0.1352183
## 5  0.02946152 0.1213222
```

Utilisons notre fonction avec des tableaux

Le language de programmation R permet de faire aussi le calcul sur des matrices

```
tableau2 <- data.frame(x_2=rnorm(5)^2,y_2=rnorm(5)^2)
prodsum(tableau, tableau2)
```

```
##            x          y
## 1  0.61615930  0.03431620
## 2  0.95373963 -0.67447815
## 3 -1.60011888  0.08471254
## 4  0.17433216  0.14264923
## 5  0.04319223 -9.04416182
```

La sortie

Une fonction peut réaliser plusieurs opérations avec les mêmes éléments d'entrée et on peut souhaiter retourner ces arguments dans une liste :

```
ma_fonction <- function(x) {

  # Calcul de la moyenne
  res1 <- mean(x)

  # Calcul de l'écart-type
  res2 <- sd(x)

  # Calcul du coefficient de variation
  res3 <- res2/res1

  # On regroupe les résultats dans une liste
  final = list(moyenne = res1, ecart_type = res2, CV = res3)

  # Et on retourne le tout hors de la fonction
  return(final)
}
```

La sortie

Une fonction peut réaliser plusieurs opérations avec les mêmes éléments d'entrée et on peut souhaiter retourner ces arguments dans une liste :

```
test <- ma_fonction(rnorm(n = 100, mean = 1, sd = 0.5))
test
```

```
## $moyenne
## [1] 1.064566
##
## $ecart_type
## [1] 0.4979353
##
## $CV
## [1] 0.4677353
```

Quelques règles utiles lorsqu'on construit une fonction

- ✓ Tout ce qui se trouve à l'intérieur d'une fonction est **entièrement** indépendent de ce qui se trouve à l'extérieur de cette fonction ;
- ✓ Donner une nom représentatif à la fonction, qui résume ce qu'elle fait ;
- ✓ Tous les arguments de la fonction doivent être utilisés;
- ✓ Les arguments doivent aussi avoir des noms représentatifs ;
- ✓ Commentez les étapes du code ;
- ✓ Prenez le temps de décrire les arguments sous forme de commentaires en début de fonction ;

Exercice de fin de séance

Abondances relatives

Toujours à partir du fichier **quadrats.csv**, calculer une fonction qui vous retournera les statistiques descriptives suivantes pour une série de données (une espèce en l'occurrence) :

1. L'abondance moyenne
2. L'abondance totale
3. Le coefficent de variation de l'abondance
4. La densité maximale dans un quadrat
5. La densité minimale dans un quadrat

Ensuite, appliquez cette fonction sur l'ensemble des espèces au moyen de la fonction *apply()*