R base

programmation orientée objet

Benjamin Louis

15/10/2019 (MàJ: 18/10/2019)

Orientée objet*

objet : structure de données auquelle est attachée une classe.

classe : définit le comportement d'un objet en décrivant ses attributs et relations avec les autres classes.

méthodes : fonctions qui vont agir différemment selon la classe de l'objet.

	Homogène	Hétérogène
1d	Atomic Vector	List
2d	Matrix	Data Frame
nd	Array	

La fonction `str()` donne la structure d'un objet et la fonction `class()` donne sa classe.

[*] https://adv-r.hadley.nz/

Opérateur arithmétique

Opérateur	Description
+	Addition
-	Soustraction
*	Multiplication
/	Division
^ ou **	Exponentiation
%%	Modulo
%/%	Division entière
%*%	Multiplication de matrice

```
2 + 3
5 / 2
5 %/% 2
5 %% 2
```

```
## [1] 5
## [1] 2.5
## [1] 2
## [1] 1
```

Fonctions

- Des fonctions permettent d'appliquer des méthodes à des objets
- Utilisation `ma_fonction(arg1, arg2)`
- Exemple :

```
sum(1:5)
```

```
## [1] 15
```

- Pour voir l'aide d'une fonction :

```
?sum #ou help(sum)
```

Packages

- Un package est un groupement de fonctions documentées autour d'une thématique
- Packages natifs de R : **base**, **utils**, **stats**

```
# Installation de nouveau package depuis le CRAN
install.packages("nom_du_package")
#Charger un package installé non natif
library(nom_du_package)
```

- https://cran.r-project.org/web/packages/available_packages_by_name.html
- Autres sources :
  - **Bioconductor** : https://www.bioconductor.org/
  - **Github** : https://github.com/

Structure des données

Vecteurs : *Atomic*

Construction

Éléments de même type, construit avec `c()`

```
vec1 <- c(1L, 2L, 3L) #ou 1:3
vec2 <- c(1.2, 5/8, 3*2, 2^3)
vec3 <- c(TRUE, FALSE, FALSE)
vec4 <- c("a", "c", "d")
```

Fonctions utiles : `typeof()`, `length()`, `names()`

Les types principaux sont : `logical` < `integer` < `double` < `character`

Sous-ensemble

```
vec1[1]
vec3[2:3]
```

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

Factors

Vecteurs qui ne peuvent prendre que certaines valeurs pré-déterminées

```
fac1 <- factor(c("a", "a", "b", "b", "c", "c"))
fac1
```

```
## [1] a a b b c c
## Levels: a b c
```

On peut préciser les valeurs pré-determinées (*levels*)

```
fac2 <- factor(c("a", "a", "b", "b", "c", "c"),
               levels = c("a", "b", "c", "d"))
fac2
```

```
## [1] a a b b c c
## Levels: a b c d
```

On peut transformer n'importe quel vecteur atomique en *factor* avec **`as.factor()`**

Vecteurs : *List*

Construction

Éléments de type différent possible, construit avec `list()`

```
li <- list(vec1, vec2, c = vec3)
li <- list(a = vec1, b = list(vec2, vec3))
```

Fonctions utiles : `str()`, `length()`, `unlist()`, `is.recursive()`, `names()`

Sous-ensemble

```
li[1]
```

```
## $a
## [1] 1 2 3
```

```
li[[1]] # ou li$a
```

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

Matrice

Construction

Éléments de même type, construit avec **matrix()**

```
m1 <- matrix(1:20, nrow = 4, ncol = 5, byrow = FALSE)
```

Fonctions utiles : **dim()**, **nrow()**, **ncol()**, **rownames()**, **colnames()**

Sous-ensemble

```
m1[1,] #1ere colonne
m1[,2] #2eme colonne
m1[1,2] #valeur de la 1ere ligne et 2eme colonne
```

```
## [1]  1  5  9 13 17
## [1] 5 6 7 8
## [1] 5
```

Data Frame

Construction

Éléments de type différent, construit avec `data.frame()`

```
df <- data.frame(x = 1:3,
                 y = c("a", "b", "c"),
                 stringsAsFactors = FALSE)
```

Fonctions utiles : `dim()`, `nrow()`, `ncol()`, `rownames()`, `colnames()`, `cbind()`, `rbind()`

Sous-ensemble

```
# Comme les matrices
df[1,]
# Les data.frame sont des listes organisées
df$x
df[[1]]
df["x"]
```

Exercices

- Installer le package `cowplot`
- Charger le package `cowplot`, regarder l'aide de la fonction `plot_grid` et essayer quelques exemples
- Dans la console, essayer quelques opérations de base
- Dans un script :
  - Créer 3 vecteurs (atomiques) : 1 numérique, 1 logique et 1 caractère
  - Faire différentes combinaisons des vecteurs 2 à 2 avec `c()` et observer le résultat. Que se passe t-il ?
  - À partir de ces vecteurs, créer une liste puis créer un data.frame
  - Donner comme nom au data.frame : *vecteur1* (numérique), *vecteur2* (logique), *vecteur3* (caractère)
  - Appliquer la fonction `mean()` à chaque colonne du data.frame. Que se passe t-il ?

Programmation

Opérateurs logiques (suite)

Opérateur	Description
==	est égal à
!=	est différent de
<	est strictement inférieur à
<=	est inférieur ou égal à
>	est strictement supérieur à
>=	est supérieur ou égal à
|	ou
&	et
is.element(x, y)	les éléments de x sont éléments de y
any	Il y a au moins un élément vrai
all	Tous les éléments sont vrais

Opérateurs logiques (suite)

x	y	x | y	x & y
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	TRUE	FALSE
FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE

```
x <- 2
y <- 1
x == 2
y > 1
x == 2 & y > 1
x == 2 | y > 1
any(c(x == 2, y > 1))
all(c(x == 2, y > 1))
```

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

Déclaration `if`

- Permet d'appliquer du code en fonction d'une condition.

```
x <- 2
if (x < 1) {
  1 + 1
} else if (x == 2) {
  print("Hello")
} else {
  0 + 0
}
```

```
## [1] "Hello"
```

- Les déclarations sont plus ou moins complexes :
  - `if` seulement
  - `if ... else`
  - `if ... else if ... else`
  - `if ... else if ... else if ... else`
- Les déclarations peuvent être imbriquées

Boucles* `while`

- Applique du code tant qu'une condition est remplie

```
x <- 0
while (x < 5) {
  print("Non")
  x <- x + 1
}
print("ok")
```

```
## [1] "Non"
## [1] "Non"
## [1] "Non"
## [1] "Non"
## [1] "Non"
## [1] "ok"
```

*Les boucles `for` que l'on trouve dans d'autres langages sont à éviter (et peuvent souvent l'être) dans R !

Programation fonctionnelle

Écrire ses propres fonctions

- Pour écrire ses propres fonctions, il existe la fonction `function()`

```
ma_function <- function(arg1, arg2, arg3 = valeur_defaut) {
  #code qui utilise arg1, arg2 et arg3
}
```

- Exemple

```
somme <- function(x, y) { x + y }
somme(x = 2, y = 5)
```

```
## [1] 7
```

```
somme_et_multiplication <- function(x, y = 5) {
  c(somme = x + y, multiplication = x * y)
}
somme_et_multiplication(x = 2)
```

```
##          somme multiplication
##              7             10
```

Exercices

- Écrire une fonction qui prend en entrée un vecteur et qui :
  - renvoie la moyenne et l'écart-type si le vecteur est numérique
  - renvoie les éléments uniques si le vecteur est de type caractère (ou *factor*)
  - renvoie la proportion d'élément vraie si le vecteur est de type logique
  - renvoie une erreur sinon
- Tester la fonction sur les vecteurs précédemment créés

Fonctions utiles : `mean()`, `sd()`, `unique()`, `is.numeric()`, `is.factor()`, `is.character()`, `is.logical()`

Solution

```r
summarize_vec <- function(x) {
  # Si numérique
  if (is.numeric(x)) {
    c(moyenne = mean(x), ecart_type = sd(x))
  # Si character ou factor
  } else if (is.character(x) | is.factor(x)) {
    unique(x)
  # Si logique
  } else if (is.logical(x)) {
    mean(x)
  # Sinon
  } else {
    stop("Erreur")
  }
}
```

Autre solution

```r
summarize_vec <- function(x) {
  # Si numérique
  if (is.numeric(x)) {
    result <- c(moyenne = mean(x), ecart_type = sd(x))
    cat("La moyenne est ", result[1],
        "\net l'écart-type est ", result[2], "\n")
  # Si character ou factor
  } else if (is.character(x) | is.factor(x)) {
    result <- unique(x)
    cat("Les valeurs dans x sont : ", as.vector(result), "\n")
  # Si logique
  } else if (is.logical(x)) {
    result <- mean(x)
    cat("La proportion d'élément vraie est de ", result, "\n")
  # Sinon
  } else {
    stop("Erreur")
  }
  result
}
```