Programmation en R

Alexis Gabadinho

2023-12-13

RÉPUBLIQUE
FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Section 1

Introduction

Objectifs pédagogiques

- Définir les fondamentaux de R
- Créer une fonction
- Gérer les attributs d'un objet
- Gérer les paramètres d'une fonction

Prérequis

- Savoir travailler en R : RStudio ou Rgui
- Savoir installer, charger un package
- Savoir naviguer dans la documentation des packages
- Savoir visualiser la structure d'une table
- Connaître le sens de l'assignation
- Connaître la fonction data.frame
- Avec le package `dplyr` : savoir créer une colonne, faire des statistiques
- Savoir utiliser le pipe %>%
- Savoir faire un graphique simple (courbe, graphique en barre)

Section 2

Les bases du langage (rappels)

Les type élémentaires (atomiques)

- Les types de base en R (`atomic`):
  - logical
  - integer
  - numeric (double)
  - complex
  - character
  - raw

Les types `integer` et `numeric`

- Numérique, virgule flottante (`double` ou `numeric`), base des calculs statistiques

```
1
```

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

```
a <- 1
class(a)
```

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

- Numérique, entier (« integer ») : un nombre entier de ce type est suivi d'un L.

```
b <- 1L
class(b)
```

```
## [1] "integer"
```

Le type `logical` (booleen)

- Logique/booléen (« logical », résultat des tests)

```
a == b
```

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

```
c <- FALSE
class(c)
```

```
## [1] "logical"
```

Le type character

- Caractère (« character », chaînes de contenu quelconque, entre simples ou doubles quotes

```
d <- 'Je suis une chaîne de caractères'
class(d)
```

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

Les types de données élémentaires: `raw`

- Brut (« raw », pour des manipulations au niveau de l'octet)

```
as.raw(2)
```

```
## [1] 02
```

- La fonction `charToRaw()` convertit un caractère de longueur 1 en `raw`

```
charToRaw("a")
```

```
## [1] 61
```

Les vecteurs

- Un donnée d'un des types atomiques ne peut exister qu'au sein d'un **vecteur**

```
a <- 1
is.vector(a)
```

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

- L'objet a est un vecteur de longueur 1

```
length(a)
```

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

- L'objet b est également de type `numeric`

```
b <- c(1,2)
class(b)
```

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

- Sa longueur est de 2

```
length(b)
```

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

Stockage des objets

- La fonction `storage.mode()` renvoie la manière dont sont représentées les données en mémoire (pour une donnée, il n'y a qu'un mode de stockage)

```
d <- Sys.Date()
d
```

```
## [1] "2023-12-13"
```

- Une date est stockée sous la forme d'un nombre décimal

```
storage.mode(d)
```

```
## [1] "double"
```

Classe d'un objet

- La fonction `class()` renvoie la façon dont pourront être utilisées les données (méthodes)

```
class(d)
```

```
## [1] "Date"
```

- Un objet peut avoir plusieurs classes (héritage), la fonction `class()` peut renvoyer un vecteur
- Ce sont les fonctions génériques (voir plus loin) qui permettent ensuite d'associer un comportement à une classe

Constantes et symboles

- Commençant par des chiffres, ou un point suivi de chiffres : un nombre (`numeric`, `double`)

```
12
```

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

- Commençant par des quotes simples (') ou doubles (") : une chaîne de caractères

```
'abcd'
```

```
## [1] "abcd"
```

- Pas de quotes, commençant par une lettre ou un point, suivi de lettres, chiffres, points, ou blanc soulignés : le nom de quelque chose, un « symbole » dont la signification peut varier

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

- Séquences de caractères prédéfinies, mots-clés du langage dont la signification est inaltérable

Fonctions et opérateurs

- Il y a deux façons d'obtenir un résultat :
  - Par des appels de **fonction** : un nom de fonction (un « symbole ») et, entre parenthèses, séparés par des virgules, les paramètres de la fonction passés soit par **position** soit par **nom**

```
library(rio)
fichier <- "/home/alex/Devel/Cours-R-Programmation/data/dpt2022.csv"
import(fichier, as.is=TRUE)
```

```
## # A tibble: 3,835,767 x 5
##     sexe preusuel       annais dpt   nombre
##    <int> <chr>          <chr>  <chr>  <int>
##  1     1 _PRENOMS_RARES 1900   02         7
##  2     1 _PRENOMS_RARES 1900   04         9
##  3     1 _PRENOMS_RARES 1900   05         8
##  4     1 _PRENOMS_RARES 1900   06        23
##  5     1 _PRENOMS_RARES 1900   07         9
##  6     1 _PRENOMS_RARES 1900   08         4
##  7     1 _PRENOMS_RARES 1900   09         6
##  8     1 _PRENOMS_RARES 1900   10         3
##  9     1 _PRENOMS_RARES 1900   11        11
## 10     1 _PRENOMS_RARES 1900   12         7
## # i 3,835,757 more rows
```

- Par des appels à des **opérateurs** : une expression, l'opérateur et une autre expression

La boucle de l'invite de commande

- Lecture d'une chaîne de caractères: `scan()`
- Interprétation du code à exécuter (syntaxe): `parse()`
- Evaluation (produit un résultat en mémoire): `eval()`
- Impression (affichage du résultat dans la console): `print()`

Les fonctions `parse()` et `eval()`

- Le rôle de la fonction `parse()` est de faire les contrôles de syntaxe. Au passage, elle transforme une séquence de caractères en une structure interne, une « expression » ordonnançant les futurs calculs

```
e <- parse(text="6*7")
e
```

```
## expression(6 * 7)
```

- L'expression contient l'arborescence des calculs à effectuer

```
as.list(e[[1]])
```

```
## [[1]]
## `*`
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 7
```

- La fonction `eval()` appliquée à une **expression**, permet d'obtenir un résultat. Elle est le coeur de R.

```
eval(e)
```

La fonction `quote()`

- Comme une majorité de fonctions, la fonction `eval()` commence par prendre la valeur de son argument puis travaille sur cette valeur. Au final, comme la fonctionnalité est d'évaluer, l'argument est évalué deux fois.
- La fonction `quote()` fait, elle, partie des fonctions qui ne commencent pas par prendre la valeur de leur argument. La fonction quote restitue son argument sans tenter de l'évaluer.

```
e <- quote(6*7)
class(e)
```

```
## [1] "call"
```

```
e
```

```
## 6 * 7
```

```
eval(e)
```

```
## [1] 42
```

- Il existe de nombreuses fonctions qui travaillent ainsi sur des objets de nature « expression » autorisant ainsi la construction de programmes constructeurs de programmes sans passer par une représentation sous

La fonction `print()`

- Toute opération en R produit un résultat, c'est à dire un objet stocké quelque part en mémoire.
- La dernière étape de la boucle est donc la visualisation de ce résultat.
- Celui ci peut être simple (une chaîne de caractères) : juste entouré de quotes ou ...
- ... un peu plus compliqué parce que nécessitant des choix de présentation (un nombre, un data.frame) ou être une structure encore plus complexe (un tableau, un graphique...)

La fonction print(): Exemple

```
library(ggformula)
```

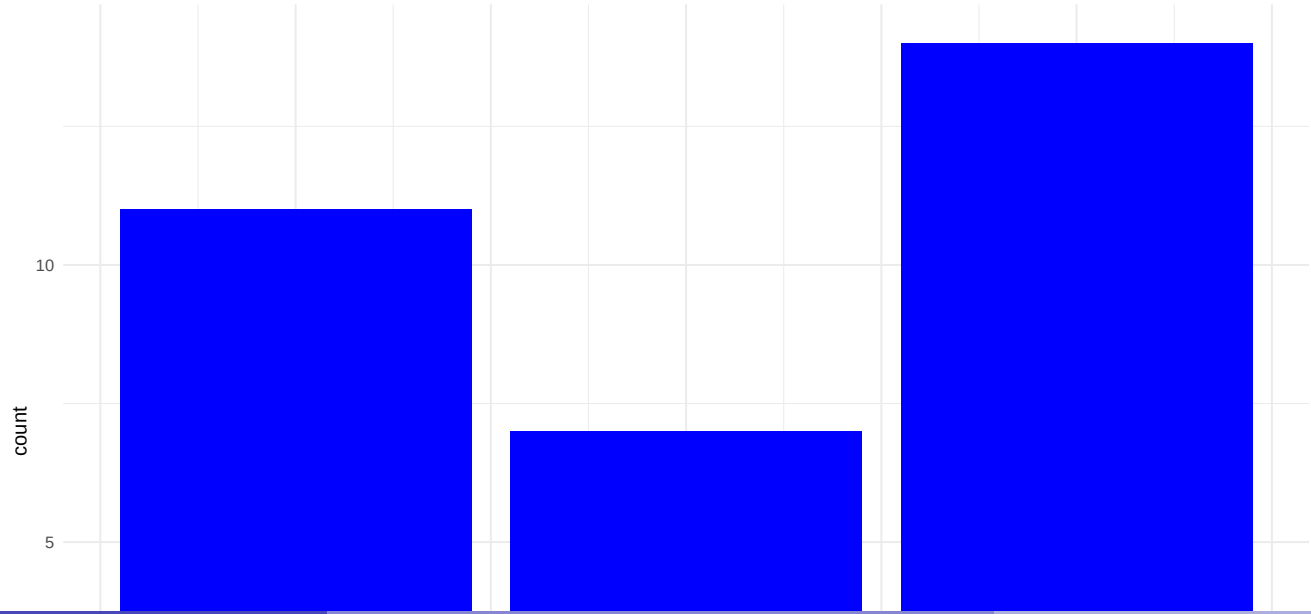
```
## Le chargement a nécessité le package : ggplot2

## Le chargement a nécessité le package : scales

## Le chargement a nécessité le package : ggridges

##
## New to ggformula?  Try the tutorials:
##  learnr::run_tutorial("introduction", package = "ggformula")
##  learnr::run_tutorial("refining", package = "ggformula")
```

```
g <- mtcars %>% gf_bar( ~ cyl, fill="blue") + theme_minimal()
g
```



La fonction `invisible()`

- Toute fonction produit un résultat, tout opérateur est une fonction, <- est un opérateur.
- Pourtant l'affectation ne montre rien

```
x <- 6 * 7
```

- L'utilisation des parenthèses permet de montrer le résultat du calcul

```
(x <- 6*7)
```

```
## [1] 42
```

- Il est possible de définir des fonctions avec un résultat marqué comme invisible et qui, comme tel, ne sera pas affiché par print.

```
6*print(7)
```

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

```
## [1] 42
```

- La fonction `invisible()` marque l'évaluation de son argument comme ne devant pas être affiché.

```
invisible(6*print(7))
```

Section 3

Symboles et objets

Contenus, noms

- De façon quasi systématique, l'appel à une fonction R crée quelque chose quelque part en mémoire.
- La quantité de mémoire occupée n'est pas définie a priori, c'est la fonction qui se charge de "prendre" ce dont elle a besoin.
- Le résultat de l'appel d'une fonction devrait donc être une indication de l'endroit où la fonction a créé quelque chose, mais cela ne serait guère pratique de manipuler des adresses mémoire.
- A la place, on utilise l'assignation <- pour associer un **nom** à l'**objet** créé (ce qui permet de le réutiliser)

```
prenoms2022 <- import("../data/dpt2022.csv", as.is=TRUE)
```

Symboles

- Contrairement à la logique de beaucoup de langages de programmation, des noms comme `prenoms2022`, `import` ne correspondent pas à des cases contenant des choses (une table, une fonction) qu'il aurait fallu pré-déclarer.
- Ce sont plutôt des étiquettes (des symboles), qu'on appose a posteriori à coté des objets pour pouvoir en parler
- L'instruction suivante ne fait aucune copie, mais créé un second nom pour identifier le même objet

```
prenoms2022b <- prenoms2022
```

Association d'un nom

1. Chargement des données en mémoire

```
prenoms2022 <- import("../data/dpt2022.csv", as.is=TRUE)
```

2. Enregistrement d'un nom, l'objet apparaît dans l'environnement `.GlobalEnv`

```
ls()
```

```
##  [1] "a"              "b"              "c"              "d"
##  [5] "def.chunk.hook" "e"              "fichier"        "g"
##  [9] "prenoms2022"    "prenoms2022b"   "x"
```

3. Association du nom aux données

Symboles ou chaînes de caractères

- Les symboles ne sont pas que des étiquettes apposées sur des objets, ils peuvent aussi être utilisés comme arguments sans faire référence à l'objet qu'ils pourraient désigner (il pourraient n'en désigner aucun!). Et des ponts existent vers le type "character".
- Ici on convertit le symbole `x` PAS sa valeur

```
as.character(quote(x))
```

```
## [1] "x"
```

- Ici on fabrique un symbole à partir d'une chaîne de caractère

```
as.name("x")
```

```
## x
```

- On peut aussi explicitement naviguer dans le lien entre un symbole, exprimé comme chaîne de caractères, et l'objet associé.

```
x <- 42
get("x")
```

```
## [1] 42
```

Les objets ne sont pas modifiables

Section 4

Les principes du langage R

Introduction

- Le statique: les données
  1. Pas de variables, mais des **objets** et des **symboles** 2, Un objet n'est **plus modifiable** après sa création
  2. Les données des types de base n'existent qu'au sein de **vecteurs**
- Le dynamique: les fonctions
  4. Une fonction derrière toute opération
  5. Une fonction est une forme particulière d'objet
  6. Toute fonction peut être surchargée
  7. Chaque fonction est libre d'interpréter ses arguments comme elle le veut

Quelques principes de programmation

- Commencez petit: -Pour résoudre un problème 'truc', ne pas commencer par écrire 'truc <- function. . . '.
  - Mais commencer par décomposer le problème en sous questions élémentaires et ensuite commencer par coder et tester ses sous-questions.
  - L'assemblage n'est que la dernière étape.
- Écrivez des petites fonctions.
  - Une fonction pour chaque opération élémentaire : ne pas tenter de faire plusieurs choses disjointes dans une même fonction.
  - Le code d'une fonction doit tenir sur un seul écran pour permettre d'en suivre la logique de déroulement sans toucher au clavier et à la souris.
  - Les fonctions potentiellement neutres (accolades, return) ne sont pas de bons amis quand elle sont sur-utilisées.
- Faites la chasse aux clones.
  - Ne JAMAIS dupliquer de code : cela alourdit le programme et introduit un point de faiblesse en cas de modification ultérieure.
- Factoriser au maximum.
  - R permet une grande souplesse dans les arguments des fonctions :des codes presque identiques peuvent toujours être réduits à l'usage d'une

Importer des données en R

- Données tabulaires (lignes x colonnes avec au croisement une donnée « élémentaire ») – import du package rio : l'outil tous terrains en fonction du suffixe – Paramétrage spécifique pour cas particulier : suivre la piste rio, la documentation indique le package utilisé (donc préconisé) et ses options – read.fwf pour les formats à largeur de champ fixe
- Données structurées non tabulaires : un peu de programmation autour de packages standard (cf. rio) – Classeurs XLSX de plus d'une feuille – Fichiers XML : packages XML, xml2
- Fichiers texte non structurés : programmer à l'aide des fonctions de manipulation de chaînes de caractères (package stringr) – Fonctions utiles : readLines , read_file du package readr
- Fichiers binaires (autres que images, sons, films) : programmer autour du type de données "raw" – Fonctions utiles : open, close, readBin

Section 5

Les vecteurs

Créer un vecteur (1)

- En R, la structure de données de base est le vecteur : ensemble de données qui sont toutes du même “type” de base (aussi dit “atomique”):
  - booléen : “logical”,
  - numérique entier : “integer”,
  - numérique, virgule flottante : “double”, date : “date”
  - numérique complexe : “complex”,
  - chaîne de caractères : “character”, facteur : “factor”
  - octet : “raw”.
- Les données dans ces types de base ne peuvent exister en dehors d'une structure de vecteur (principe 3).
- La plus simple façon de créer un vecteur (de longueur 1) est juste d'écrire une constante dans un type de base :

```
42
```

```
## [1] 42
```

Un vecteur peut être vide, par exemple à la suite d'une sélection

Créer un vecteur (2)

- On peut également utiliser:
  - la fonction `vector()` qui crée un vecteur de type donné et de longueur donnée

```
vector("complex",5)
```

```
## [1] 0+0i 0+0i 0+0i 0+0i 0+0i
```

- La fonction de collecte `c()` qui cherche à combiner ses arguments en vecteur (lorsque c'est possible : quand ils peuvent être mis au même type, sinon elle produit une liste)

```
c(6,7)
```

```
## [1] 6 7
```

– de nombreuses autres fonctions : la répétition,

```
rep(TRUE,5)
```

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

la fabrication de suite d'entiers consécutifs, etc. . .

```
1:10
```

Exercice

- Avec la fonction `as.Date()` convertir la chaîne de caractères “2017-01-01” en donnée de type « date » -Lui ajouter 1. On passe au 2 janvier 2017. -Lui ajouter la suite des nombres de 1 à 7.
- Appliquer la fonction `weekdays()` au résultat que l'on mémorisera sous le nom `jour`. Quel jour de la semaine était le 1er janvier 2017 ?

Accéder aux éléments d'un vecteur (1)

- La sélection d'une partie d'un vecteur se fait avec l'opérateur "crochet" `[...]` (c'est à dire une fonction, cf. principe numéro 4) au sein duquel on peut préciser :
  - un vecteur de nombres entiers : pour extraire les éléments de numéros cité

```
(v <- 101:110)
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

par exemple les éléments 3 à 5

```
v[3:5]
```

```
## [1] 103 104 105
```

  - ou les éléments 5 à 3

```
v[5:3]
```

```
## [1] 105 104 103
```

  - un vecteur de nombres entiers négatifs : pour extraire tous les éléments sauf ceux de numéro cités

Accéder aux éléments d'un vecteur (2)

- un vecteur de booléens pour spécifier quels éléments doivent être conservés (TRUE) ou non (FALSE)

```
v[c(TRUE,rep(FALSE,8),TRUE)]
```

```
## [1] 101 110
```

- Le crochet ne fait pas de l'indexation, c'est un véritable opérateur entre deux vecteurs. - L'intérêt majeur de la sélection par vecteur de booléens est que ce vecteur peut provenir d'un calcul logique, où on exprime une condition sur les éléments. Cette condition peut utiliser n'importe quel élément connu de R, et en particulier le vecteur lui-même. - Exemple : une sélection des éléments dont la parité (pair/impair) dépend du positionnement d'un paramètre de nom PARITE (%% est le reste de la division entière)

```
PARITE <- 1
v[(v %% 2)==PARITE]
```

```
## [1] 101 103 105 107 109
```

Les listes