Programmation en R

Alexis Gabadinho

2023-12-18

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

Supports

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 principes du langage R

Objets et classes

- En R tout est **objet**
- Chaque objet a une (ou plusieurs) classes
- La fonction `class()` renvoie la classe d'un objet

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

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

- Ce sont les fonctions génériques (voir plus loin) qui permettent ensuite d'associer un comportement à une classe
- La classe d'un objet est **modifiable** ´

```
class(a) <- c("truc","machin")
class(a)
```

```
## [1] "truc"   "machin"
```

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] "2023-12-18"
```

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

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

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

- Pour les types de base, classe et stockage sont

```
i <- 12L
storage.mode(i)
```

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

Constantes et symboles (1)

- 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"
```

Constantes et symboles (2)

- 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

```
TRUE
```

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

Fonctions et opérateurs (1)

- 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
```

Fonctions et opérateurs (2)

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

```
1+1
```

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

```
a <- 1
```

- Certains opérateurs sont fournis par des librairies optionnelles

```
library(tidyr)
fichier %>% import(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
```

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)
```

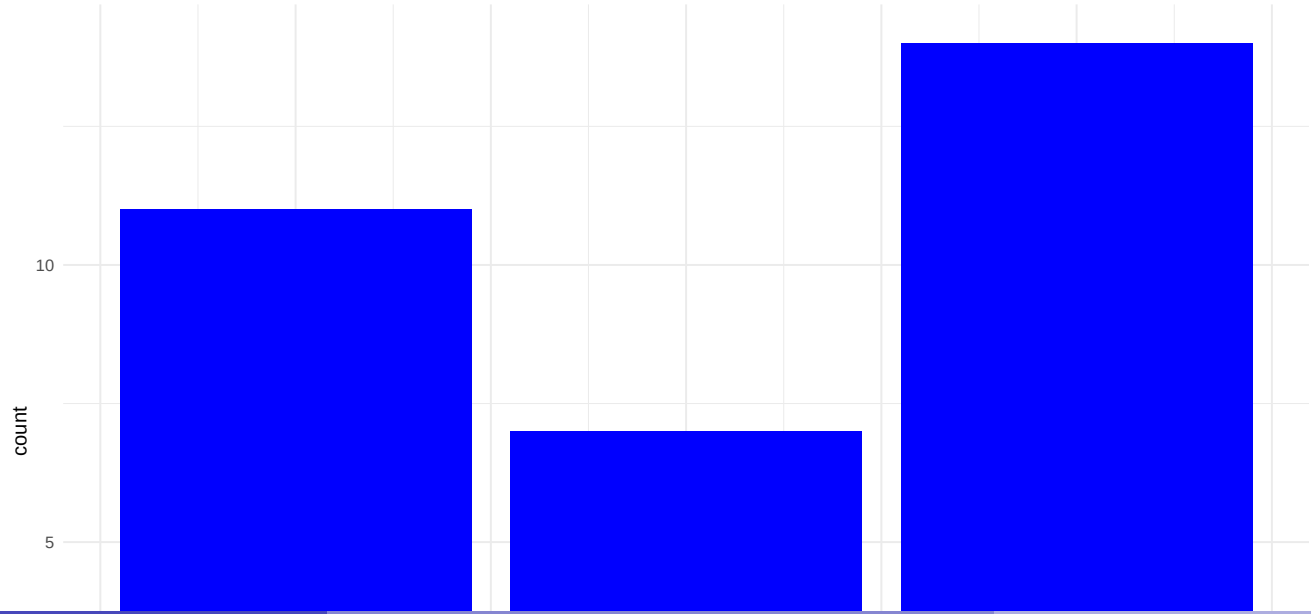
```
## Le chargement a nécessité le package : ggplot2

## 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))
```

Symboles et objets (1)

- 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 et objets (2)

- 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"             "d"             "def.chunk.hook" "e"
##  [5] "fichier"       "g"             "i"              "prenoms2022"
##  [9] "prenoms2022b"  "x"
```

3. Association du nom aux données

Symboles ou chaînes de caractères (1)

- 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
```

Symboles ou chaînes de caractères (2)

- 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
```

- Par contre l'instruction suivante renvoie une erreur car cela revient à faire get(42)

```
get(x)
```

- Assignation d'une valeur à un nom dans un environement

```
assign("x",pi)
```

Les objets ne sont pas modifiables (1)

- En règle quasi-générale (exceptions : data.table, R6), il est impossible de modifier un objet de R.
- Fonctionnellement, si on veut faire une modification à un objet, par exemple une table de données, associée à un symbole donné :
  - on construit une copie modifiée de la table
  - on associe la copie modifiée au symbole
  - la précédente table associée au symbole est alors perdue parce que plus référençable (sauf si on l'avait associée à un deuxième symbole).
- En pratique, c'est R qui se charge de minimiser le nombre de réelles copies et de supprimer de la mémoire les objets "perdus" (non référençables).

Les objets ne sont pas modifiables (2)

- Par exemple, avec le package dplyr, pour convertir la colonne `sexe` de la table 'nanopop' en numérique dans une nouvelle variable sexe2, on écrira :

```
nanopop %>% mutate(sexe2=as.numeric(sexe))
```

- En sortie il y a en mémoire :
  - le résultat, une table avec la colonne supplémentaire , qui est juste affiché,
  - mais aussi la table originale, qui est toujours associée au symbole nanopop.
- Pour "modifier" l'objet, il faut écrire explicitement

```
nanopop %>% mutate(sexe2=as.numeric(sexe)) -> nanopop
```

Section 3

Les bases du langage

Les type élémentaires (atomiques)

- 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”)
- Les types de base en R (`atomic`):
  - logical
  - integer
  - numeric (double)
  - complex
  - character
  - raw
- Les données dans ces types de base ne peuvent exister en dehors d'une structure de vecteur (principe 3).

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
```

Créer un vecteur (1)

- 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 infructueuse. La notation est le type du vecteur suivi de (0) :

```
v <- logical(0)
v
```

```
## logical(0)
```

```
length(v)
```

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

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.

Accéder aux éléments d'un vecteur (3)

- 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
```

- L'opérateur %% est appliqué à tous les éléments de `x`

```
v %% 2
```

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

- Le test renvoie un vecteur de booléens

```
(v %% 2)==PARITE
```

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

Utilisation de `all()` et `any()`

- Création d'un vecteur de nombres pairs

```
x <- seq(2,12,2)
x
```

```
## [1]  2  4  6  8 10 12
```

- Pour vérifier que tous les nombres sont pairs, on peut utiliser `all()`

```
all(x %% 2 == 0)
```

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

Utilisation de `all()` et `any()`

- Pour tester si un des éléments de `x` est impair

```
any(x %% 2 != 0)
```

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

- On peut sommer les valeurs booléennes pour obtenir les nombre d'éléments concernés

```
sum( x > 6)
```

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

Le crochet n'est qu'un opérateur

- Puisque le crochet n'est qu'un opérateur, rien n'interdit d'utiliser autre chose qu'un symbole du coté gauche de l'opérateur, en particulier un résultat d'un appel de fonction

```
seq(1,50,5)[1:5]
```

```
## [1]  1  6 11 16 21
```

Exercice - Extraire une partie d'un vecteur

- En utilisant le vecteur des 7 jours à partir d'aujourd'hui compris (fonction `as.Date()` ou `Sys.Date()`), quels sont les dates dont le jour de la semaine se termine par « di » ?

```
## [1] "lundi"    "mardi"    "mercredi" "jeudi"    "vendredi" "samedi"   "dimanche"
```

- On pourra utiliser la fonction `endsWith` ou la fonction `str_detect` du package `stringr` avec une expression régulière

La fonction `which`

- La fonction which permet de faire un pont entre une sélection par booléens ou une sélection par indice : elle restitue les positions des éléments vérifiant une condition.

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

- Position des lettres N à P

```
which(LETTERS>="N" & LETTERS<="P")
```

```
## [1] 14 15 16
```

- Version plus simple sans 'which'

```
length(LETTERS[which(LETTERS>="Z")])
```

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

- **Attention** : which est souvent utilisée dans des contextes où on n'en a pas besoin grâce à la possibilité de sélection par booléens

Le recyclage des éléments dans les opérations vectorielles (1)

- En R, les opérateurs arithmétiques et logiques ainsi qu'un grand nombre de fonctions dont [sont **vectorisés**
- Lorsqu'on doit faire quelque chose sur tous les éléments d'un vecteur, il n'y a pas à penser à répéter la chose sur chaque élément. C'est R qui va se charger de la "boucle"
- Et, avec les architectures modernes (multi-cores, GPUs), il est même possible que les opérations se fassent de façon simultanée !

Le recyclage des éléments dans les opérations vectorielles (2)

- Avec des vecteurs de même longueur

```
1:10 + rep(100,10)
```

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

- Avec deux vecteurs de longueurs différentes, par exemple un calcul impliquant un scalaire et un vecteur

```
1:10 + 100
```

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

- R “recycle” le vecteur le plus court pour en faire (virtuellement) un vecteur de la même longueur que le plus long

```
1:10 + rep(100,11)
```

```
## Warning in 1:10 + rep(100, 11): la taille d'un objet plus long n'est pas
## multiple de la taille d'un objet plus court
```

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

```
1:10 + c(100,200)
```

Recyclage

- Le code suivant fonctionne :

```
library(dplyr)
```

```
##
## Attachement du package : 'dplyr'

## Les objets suivants sont masqués depuis 'package:stats':
##
##     filter, lag

## Les objets suivants sont masqués depuis 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
mtcars %>% filter(cyl=="6")
```

```
## # A tibble: 7 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
## 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
## 3  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
## 4  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
## 5  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## 6  17.8     6  168.   123  3.92  3.44  18.9     1     0     4     4
## 7  19.7     6  145    175  3.62  2.77  15.5     0     1     5     6
```

```
mtcars %>% filter(cyl==c("6","4"))
```

```
## # A tibble: 8 x 11
```

La fonction paste()

- La concaténation de chaînes de caractères se fait avec la fonction paste0
- Appliquée à deux vecteurs, elle produit un vecteur de chaînes de caractères :

```
v <- c("un","deux","trois","quatre","cinq")
w <- 1:4
paste0(v,w)
```

```
## [1] "un1"     "deux2"   "trois3"  "quatre4" "cinq1"
```

- La fonction originale paste introduit un espace:

```
paste(v,w)
```

```
## [1] "un 1"     "deux 2"   "trois 3"  "quatre 4" "cinq 1"
```

- L'argument collapse permet de changer le mode de fonctionnement : la concaténation produit un vecteur d'un seul élément.

```
paste(v, collapse=";")
```

```
## [1] "un;deux;trois;quatre;cinq"
```

Modifier une partie d'un vecteur (1/3)

- Pour modifier une partie d'un vecteur on fera par exemple :

```
v <- 101:110
v[3:5] <- 0
v
```

```
##  [1] 101 102   0   0   0 106 107 108 109 110
```

- Un vecteur est extensible à souhait, R complète les trous :

```
v[12] <- 12
v
```

```
##  [1] 101 102   0   0   0 106 107 108 109 110  NA  12
```

Modifier une partie d'un vecteur (2/3)

- En R, on ne peut pas modifier un objet !
- Or dans l'écriture `v[3:5] <- 0`:
  - une partie du vecteur semble "recevoir" la valeur 0 comme s'il s'agissait de cases mémoire dans un langage classique,
  - l'assignation <- ne semble pas du tout être ici l'association d'un nom à une valeur : v[3:5] n'est pas un symbole.

Modifier une partie d'un vecteur (3/3)

- En fait cette écriture est un leurre, et sous une apparence d'instruction d'affectation de langage de programmation classique, elle cache deux choses :
  - le recours à une nouvelle fonction, de nom un peu particulier [<-, qui se charge de construire un (nouveau) vecteur modifié, et joue en quelque sorte le rôle d'une fonction d'écriture de données, alors que la fonction [réalise la fonction de lecture,
  - un raccourci pour une situation classique. Quand on veut, en R, modifier un objet associé à un symbole :
    1. on crée une copie modifiée de l'objet indiqué par le symbole,
    2. on réassigne le symbole à ce nouvel objet.

Supprimer des éléments d'un vecteur

- En revanche on ne peut pas éliminer un élément d'un vecteur.
- Mais il est possible de sélectionner tous les éléments sauf celui qu'on voudrait éliminer, ce qui revient au même si on appelle le résultat du même nom que l'original.

```
v <- v[-2]
v
```

```
##  [1] 101   0   0   0 106 107 108 109 110  NA  12
```

Les matrices (1)

- Pour les besoins des calculs scientifiques, les matrices existent en R sous forme d'une spécialisation de la représentation en vecteur
- Pour créer une matrice on peut commencer par créer un vecteur puis on spécifie les dimensions de la matrice en utilisant la forme "réversible" de la fonction `dim` : `dim<-`

```
m <- c(0,5,4,9,3,0,0,1,2,7)
dim(m) <- c(2,5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    4    3    0    2
## [2,]    5    9    0    1    7
```

- Une matrice peut avoir plus de deux dimensions

Accès aux éléments d'une matrice

- Les crochets permettent d'accéder à des éléments ou des portions d'une matrice qui seront soit des vecteurs soit des matrices :
  - un seul élément

```
m[2,5]
```

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

- une ligne

```
m[2,]
```

```
## [1] 5 9 0 1 7
```

- une colonne

```
m[,2]
```

```
## [1] 4 9
```

Sélection des éléments d'une matrice avec des booléens

- Les opérateurs de comparaison >, >=, '==', != s'appliquent à chaque élément de la matrice

```
m>4
```

```
##       [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] FALSE FALSE FALSE FALSE FALSE
## [2,]  TRUE  TRUE FALSE FALSE  TRUE
```

- On peut les utiliser pour sélectionner des parties de la matrice

```
m[m>4]
```

```
## [1] 5 9 7
```

- Le résultat est un vecteur

Opérations sur les matrices

- Opération entre une matrice et une valeur unique

```
m * 2
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    8    6    0    4
## [2,]   10   18    0    2   14
```

- Opération entre deux matrices

```
m - m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
```

Les fonctions rowSums et colSums

- Somme des lignes

```
rowSums(m)
```

```
## [1]  9 22
```

- Somme des colonnes

```
colSums(m)
```

```
## [1]  5 13  3  1  9
```

- Sélection

```
m[, colSums(m)>5]
```

```
##      [,1] [,2]
## [1,]    4    2
## [2,]    9    7
```

Les listes

- Une liste est une collection d'objets qui peuvent être de types **différents**
- On crée une liste avec la fonction `list()`. Une liste peut être vide : `list()`.
- Les éléments d'une liste peuvent être nommés. Cela simplifiera l'accès ultérieur en précisant des noms et non des positions
- La liste ci-dessous contient des champs des types élémentaires du langage R: double, entier, booléen, chaîne de caractères, fonction e liste. Trois des champs sont nommés : a, f, l. Les autres ne seront accessibles que par leur position.

```
liste <- list(a=1, 3L, "a", TRUE, f=sum, l=list(1, 2))
```

Utilisation des listes en R

- La liste est la structure de données la plus importante de R car elle permet de mémoriser n'importe quelle information complexe.
- Elle est à la source de la définition de nombreux types de données : **data frames**, **objets graphiques**, **résultats de régression**, ...
- Ces types de données sont souvent accompagnés d'une redéfinition de la fonction d'impression/affichage à l'écran qui cache la structure interne en produisant une "belle" sortie.
- Pour connaître la structure interne, il faut alors utiliser la fonction `str()` qui affiche un descriptif détaillé.

```
str(liste)
```

```
## List of 6
##  $ a: num 1
##  $  : int 3
##  $  : chr "a"
##  $  : logi TRUE
##  $ f:function (..., na.rm = FALSE)
##  $ l:List of 2
##   ..$ : num 1
##   ..$ : num 2
```

Accès aux éléments d'une liste

- Deux opérateurs (cf. principe numéro 4) sont disponibles pour accéder à un seul élément d'une liste :
  - le double crochet `[[...]]`, avec :
    - soit un numéro d'élément (surtout en l'absence de noms)

```
liste[[5]]
```

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

- soit un nom d'élément (quand la liste contient des éléments

```
liste[["f"]]
```

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

  - le $ qui n'évalue pas son argument de droite (qui doit être un nom) et ne permet donc pas de paramétrer l'élément à récupérer

```
liste$f
```

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

Accéder à une partie d'une liste

- L'opérateur simple crochet permet d'extraire une partie d'une liste systématiquement sous forme de liste, même lorsqu'il n'y a qu'un seul élément.
- Le fonctionnement est similaire à l'opérateur sur les vecteurs. On a le choix entre lui fournir :
  - un vecteur de nombres, éventuellement tous négatifs,

```
liste[5]
```

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

  - un vecteur de booléens,
  - un vecteur de noms d'éléments.

```
liste[-1:-4]
```

```
## $f
## function (..., na.rm = FALSE)  .Primitive("sum")
##
## $l
## $l[[1]]
## [1] 1
##
## $l[[2]]
```

« Modifier » une liste

- Les opérateurs [[et $ sont "réversibles". On peut ainsi positionner un élément dans une liste par sa position ou son nom avec [[<-, ou par son nom $<-
- Dans l'exemple ci-dessous, on ajoute un 4 ème élément, le troisième est indéfini

```
liste <- list(1,2)
liste[[2]]<- -1
liste[[4]]<- -2
liste
```

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

- Pour ajouter un élément à unbe liste, celle-ci doit préalablement exister

```
liste2 <- list()
liste2$a <- 1
```

Convertir une liste en vecteur : `unlist` (1)

- Lorsque tous les éléments d'une liste sont de même type, la fonction `unlist` permet de construire un vecteur en "écrasant" tous les éléments de la liste.

```
list(1, 2, 3) %>% unlist()
```

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

- Ici les éléments sont convertis en chaîne de caractère

```
list(1, "a") %>% unlist()
```

```
## [1] "1" "a"
```

- Listes imbriquées

```
list(1, list(2,3)) %>% unlist()
```

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

Convertir une liste en vecteur : `unlist` (2)

- Lorsque les éléments de la liste sont nommés, les éléments du vecteur résultat sont - par défaut (voir paramètre ad'hoc) - également nommés (comme pour les éléments d'une liste, les élements d'un vecteur peuvent être nommés)

```
list(a=1, b=2, c=3) %>% unlist()
```

```
## a b c
## 1 2 3
```

Stockage des listes en mémoire (1)

- En mémoire la structure de liste n'est guère différente de celle d'un vecteur afin de permettre le même type d'accès aléatoire.

```
liste <- list(a=1,b=2)
is.vector(liste)
```

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

- Utilisation de `inspect` pour afficher les attributs internes

```
library(pryr)

inspect(liste)
```

```
## <VECSXP 0x564cf60430e8>
##   <REALSXP 0x564cf43484f8>
##   <REALSXP 0x564cf4348530>
## attributes:
##   <LISTSXP 0x564cf4903428>
##   tag:
##     <SYMSXP 0x564cebd878d0>
##   car:
##     <STRSXP 0x564cf6043128>
##       <CHARSXP 0x564cec08c7b0>
##       <CHARSXP 0x564cec31e928>
##   cdr:
##     NULL
```

Stockage des listes en mémoire (2)

- Mais d'autres éléments de R nécessitent une structure de liste : un programme est une liste d'appels de fonctions.
- Pour ses besoins internes, R utilise une structure de liste (« pair list ») formée de cellules chaînées entre-elle, directement héritée de LISP. Chaque cellule a un « tag » spécifiant son contenu, un « car » donnant le contenu et « cdr » indiquant la suite de la liste.
- La navigation dans ce type de structure n'est pas possible sans recours aux fonctions internes, mais des conversions sont possibles avec la fonction as.list.

Section 4

Importer des données en R

Types de données (1)

- 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
- `read.csv` pour les fichiers `.csv` avec un séparateur `,`ou `;`

Types de données (2)

- 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

Types de données (3)

- Fichiers binaires (autres que images, sons, films) : programmer autour du type de données “raw”
- Fonctions utiles : open, close, readBin

Exemple: Prix des carburtants

- Un fichier réel de mise à disposition d'informations sur les points de vente de carburant :
  - localisation
  - période d'ouverture
  - services annexes
  - historique des prix

Lecture du fichier

- On lit l'intégralité du fichier avec la fonction `readLines()`

```
con <- file("../data/PrixCarburants_quotidien_20231211.xml", encoding = "latin1")
carburants <- readLines(con)
close(con)
unique(Encoding(carburants))
```

```
## [1] "unknown" "UTF-8"
```

- Le résultat est un (très long) vecteur

```
head(carburants)
```

```
## [1] "<?xml version=\"1.0\" encoding=\"ISO-8859-1\" standalone=\"yes\"?>"
## [2] "<pdv_liste>"
## [3] "  <pdv id=\"1000001\" latitude=\"4620100\" longitude=\"519800\" cp=\"01000\" pop=\"R\">"
## [4] "    <adresse>596 AVENUE DE TREVOUX</adresse>"
## [5] "    <ville>SAINT-DENIS-LèS-BOURG</ville>"
## [6] "    <horaires automate-24-24=\"\">"
```

Sélection des lignes

- Avec la fonction str_detect de stringr, ne conserver que les lignes contenant 'pdv' (avec un espace à la fin, donc ne contenant pas 'pdv_liste').
- str_detect est une fonction qui accepte une expression régulière mais nous n'en avons pas besoin ici.

```
library(stringr)
carburants <- carburants[str_detect(carburants, "pdv ")]
head(carburants)
```

```
## [1] "  <pdv id=\"1000001\" latitude=\"4620100\" longitude=\"519800\" cp=\"01000\" pop=\"R\">"
## [2] "  <pdv id=\"1000002\" latitude=\"4621842\" longitude=\"522767\" cp=\"01000\" pop=\"R\">"
## [3] "  <pdv id=\"1000004\" latitude=\"4618800\" longitude=\"524500\" cp=\"01000\" pop=\"R\">"
## [4] "  <pdv id=\"1000007\" latitude=\"4622100\" longitude=\"524500\" cp=\"01000\" pop=\"R\">"
## [5] "  <pdv id=\"1000008\" latitude=\"4619900\" longitude=\"524100\" cp=\"01000\" pop=\"R\">"
## [6] "  <pdv id=\"1000009\" latitude=\"4619600\" longitude=\"522900\" cp=\"01000\" pop=\"R\">"
```

- Avec la fonction `str_replace_all`, éliminer : <pdv espaces avant, un après), les > , les double quotes

```
carburants <- str_replace_all(carburants,"  <pdv " , "")
carburants <- str_replace_all(carburants, "\\\\", "")
head(carburants)
```

```
## [1] "id=\"1000001\" latitude=\"4620100\" longitude=\"519800\" cp=\"01000\" pop=\"R\">"
```

Exercice : Utilisation des expressions régulières (1)

- Les expressions régulières sont un outil de test de chaînes de caractères permettant de reconnaître la présence d'un ensemble de chaînes possibles grâce à une syntaxe spécifique : un des caractères `^ $ . * + ? | ( ) [ ] { } \`
- La fonction `str_detect` du package `stringr` est une des fonctions réalisant ce genre de tests : elle cherche si son premier argument contient quelque chose ressemblant à son second argument et répond vrai ou faux.
- Le premier argument peut être un vecteur, le résultat sera un vecteur de booléens.

Exercice : Utilisation des expressions régulières (2)

- Les éléments de syntaxe les plus fréquemment utilisés :

```
str_detect(arg,"^0") # commence par '0'
str_detect(arg,"0$") # se termine par '0'
str_detect(arg,"^\\$") # commence par '$' non interprété

str_detect(arg,"^.0") # commence par n'importe quel caractère puis un 0
str_detect(arg,"^[1-6]") # commence par un caractère de '1' à '6'
str_detect(arg,"^[1-68]") # commence par un caractère de '1' à '6' ou un '8'
str_detect(arg,"^[^1-6]") # commence par tout sauf un caractère de '1' à '6'
str_detect(arg,"^\\d") # commence par un chiffre décimal

str_detect(arg,"^.*9") # commence par 0 et + caractères quelconques puis un 9
str_detect(arg,"^.+9") # commence par 1 et + caractères quelconques puis un 9
str_detect(arg,"^01?") # commence par 0 et éventuellement un 1
str_detect(arg,"^0(19)?") # commence par 0 et éventuellement le groupe '19'
str_detect(arg,"^(19|20)")# commence par '19' ou '20'
```

Exercice

- Avec la fonction str_split, éclater la colonne issue du data.frame précédent pour séparer en une liste de couples nom=valeur.
- Afficher la structure de la première ligne de la table résultat.
- La table obtenue précédemment n'est pas « propre » (des données non élémentaires dans la dernière colonne).
- Avec la fonction `unnest` de `tidyr`, répartir la nouvelle colonne de type liste sur plusieurs lignes.
- Avec la fonction `separate`, séparer les deux composants des couples nom=valeur dans deux nouvelles colonnes.
- Le résultat peut être considéré comme une table en format long : un colonne indique le nom de la donnée, une autre la valeur. Mettre le résultat en format « large » (fonction spread ou équivalent): une colonne par modalité du nom.
- Avec la fonction `gf_point` de `ggformula`, faire une carte des latitudes - longitudes, en les ayant préalablement converties en numérique.

Section 5

Programmation avec 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 (1)

- 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.

Quelques principes de programmation (2)

- 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.

Quelques principes de programmation (3)

- 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 unique fonction.

Quelques principes de programmation (4)

- Respectez la symétrie. Si le problème à traiter présente une forme de symétrie, celle-ci doit se retrouver dans le code, sinon c'est un indice de cas mal couverts. -Ne vous préoccupez pas d'optimisation, sauf dans les cas critiques -Testez Commencer par les cas extrêmes. Les autres ont plus de chances de fonctionner.
- Adoptez un style et tenez vous y.
- Il y a plusieurs façons d'écrire du R, de nommer ses objets ou de présenter les programmes.
- Ne mélangez les patois R qu'en cas de nécessité.
- Un nom d'objet parlant évite bien des commentaires.
- Décrivez ce que font vos fonctions.
- Commentez les passages difficiles.
- Mais uniquement les passages difficiles ; un commentaire de type

Section 6

Définir des fonctions

Création d'une fonction

1. Lister les noms des paramètres de la fonction
2. Ecrire une expression utilisant ces paramètres
3. Donner un nom à la fonction
4. Exécuter le code donnant la définition pour créer la fonction dans l'environnement courant

```
Somme <- function(début,fin)
  sum(c(-1,1)*(début:fin)^2)
```

5. Utilisation de la fonction

```
Somme(1,100)
```

```
## [1] 5050
```