



RÉPUBLIQUE
FRANÇAISE

*Liberté
Égalité
Fraternité*



R Programmation

*Ca marche comment?
Moi aussi je peux faire!*

SAV :
jean-luc.lipatz@insee.fr

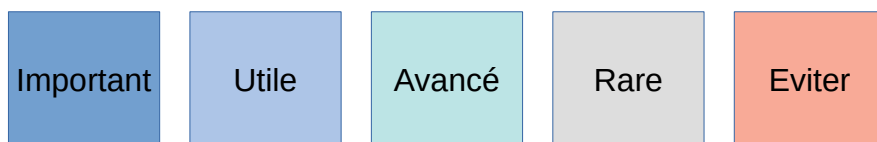
Objectifs de la formation



- Découvrir une autre façon de travailler en R en sortant du cadre des formations construites autour du package **dplyr** et en donnant des clés de **lecture des programmes écrits en R de base**.
- Se familiariser avec les **diverses façon de définir une fonction**, en particulier dans la perspective d'automatiser des travaux ou d'écrire des applications **Shiny**.
- **Acquérir les fondamentaux de R** : découvrir que R n'est pas un ensemble de formules magiques mais une construction rigoureuse autour d'une organisation des données et d'un très petit nombre de principes.

Classification des vues

Programmer



Comprendre



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)

Demandez le programme !

Principe : découvrir les fonctionnalités de base du langage R autour de la résolution d'un problème concret.

- Prologue : retour sur quelques bases du langage
- Acte 1 scène 1 : Les vecteurs
- Acte 1 scène 2 : Les listes
- Acte 1 scène 3 : Les fonctions
- Acte 1 scène 4 : Les data frames
- Acte 2 scène 1 : Les boucles
- Acte 2 scène 2 : Les attributs d'un objet
- Acte 2 scène 3 : Les traitements conditionnels
- Epilogue : quelques compléments

		page	
Zéro	Rappels ; les principes du langage	7	
R1	Les vecteurs	33	
R2	Les listes	53	
F1	Définir une fonction	65	
F2	Les paramètres d'une fonction	83	
R3	Les tables de données	95	
B	Faire des traitements itératifs	115	
R4	Les attributs d'un objet	133	
C	Faire des traitements conditionnels	143	
F3	Les environnements	155	
F4	Des fonctions qui génèrent des fonctions	165	
F5	Les fonctions génériques	175	
F6	L'évaluation retardée	183	

Important

6

Rare

1

Important

4

Rare

1

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l)!=length(n)) stop("Wrong number of arguments.")  
    names(l)<- if (is.null(names(l))) as.character(n)  
               else ifelse(names(l)=="",as.character(n),names(l))  
    e <- do.call(substitute,list(c$expr,l))  
    eval(e,envir=-2)  
  }  
}
```

Séquence Zéro

Quelques rappels

Les bases du langage
Symboles et objets

Rappel

Les types de données élémentaires

- Les types de base ('atomic'):
Numérique, virgule flottante (« **double** » , base des calculs statistiques)
 Numérique, entier (« **integer** » rare) : un nombre entier de ce type est suivi d'un L.
Complexe (« **complex** », pour les matheux)
Logique/booléen (« **logical** », résultat des tests)
Caractère (« **character** », chaînes de contenu quelconque, entre simples ou doubles quotes)
Brut (« **raw** », pour des manipulations au niveau de l'octet)
- Une donnée d'un des types atomiques ne peut exister qu'au sein d'un vecteur.**
- Ces types de base correspondent à des modes de stockage de l'information en mémoire.** En utilisant ces modes on peut construire d'autres types de données qui partageront l'organisation en mémoire mais auront des propriétés supplémentaires : les facteurs, les dates, etc..

Ce qui suit est le premier élément d'un vecteur

```
> 1
[1] 1
> 1L
[1] 1L

> 1==2
[1] FALSE

> "essai=|'|0'|"
[1] "essai=|'|0'|"

# Un type construit au dessus du numérique
# une apparence de chaîne de caractères...
> Sys.Date()
[1] "2021-07-15"

# ... et une représentation interne numérique
> storage.mode(Sys.Date())
[1] "double"
```


Mode stockage vs. utilisation

La fonction `class`

- La fonction `storage.mode` renvoie la manière dont sont représentées les données en mémoire, juste pour satisfaire sa curiosité. Pour une donnée, il n'y a qu'un mode de stockage.
- La fonction `class` renvoie, elle, la **façon dont pourront être utilisées les données**. Il peut y en avoir plusieurs : la fonction peut renvoyer un vecteur de plus d'une chaîne de caractères.
- Ce sont les fonctions génériques (voir plus loin) qui permettent ensuite d'associer un comportement à une classe.

```
# Entre chaîne de caractères et flottant
# ...un vrai type de données !
> class(Sys.Date())
[1] "Date"
> storage.mode(Sys.Date())
[1] "double"

# Mode et classe sont identiques pour les
# types de base
> class(1L)
[1] "integer"

# Le mode n'est pas modifiable sans altérer
# la donnée, la classe si
> x <- 1
> class(x)
[1] "numeric"
> class(x) <- c("truc", "machin")
> class(x)
[1] "truc"      "machin"

# Il y a d'autres types à usage interne
> class(`class`)
[1] "function"
> storage.mode(`class`)
[1] "function"
```

Quelques éléments de syntaxe non ambigus

Constantes et symboles

- Commençant par des chiffres, ou un point suivi de chiffres : un **nombre**
 - « général » **Virgule flottante** ('numeric', 'double')
12, .12, 12.0e-5
 - **Entier**, terminé par la lettre L
12L
 - **Complexe**, que nous n'utiliserons pas
- Commençant par des quotes simples (') ou doubles ("): une **chaîne de caractères**.
'essai=ניסיון'
- 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, .truc, ., ..., T, F
- Quelques séquences de caractères prédéfinies, des **mots-clés** du langage dont la signification est inaltérable... plutôt rares.

D'un point de vue syntaxique
-1 n'est pas un nombre
mais une expression !

TRUE, FALSE, les NAs

Quelques éléments de syntaxe

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

```
import("prenoms.dbf", as.is=TRUE)
import(file=file.choose())
```

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

```
1+1
a <- 1
"prenoms.dbf" %>% import(as.is=TRUE)
```

- **Tout appel produit un résultat** (pas forcément visualisé comme avec '`<-`')
- Mais la façon d'obtenir le résultat n'est jamais gravée dans le marbre.

La signification d'un nom de fonction ou d'opérateur peut varier.

En R, il n'y a pas d'instructions, uniquement des calculs.

La boucle de l'invite de commande

The screenshot shows the RStudio interface with several annotations explaining the components of the command loop:

- Lecture scan()**: Une chaîne de caractères (A string of characters)
- Interprétation parse()**: Du code à exécuter (syntaxe) (Code to be executed (syntax))
- Evaluation eval()**: Un résultat en mémoire (sémantique) (A result in memory (semantics))
- Impression print()**: Quelque chose à l'écran (Something on the screen)

The console output shows the execution of the following code:

```
> a <- 1
> library("rio", lib.loc="v:/PALETTES/R-3.5.1/library")
The following rio suggested packages are not installed: 'csvy',
'hexView', 'rmatio'
Use 'install_formats()' to install them
Warning message:
le package 'rio' a été compilé avec la version R 3.5.2
> 1+1
[1] 2
>
```

The Environment pane shows the variable 'a' with the value 1. The Packages pane shows a list of installed and suggested packages.

De l'input au résultat

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 » ordonnant les futurs calculs.
- C'est la fonction `eval` qui appliquée à une « expression », permet d'obtenir un résultat.
La fonction `eval` est le coeur de R.

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

```
> e
```

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

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

```
[[1]]  
`*`
```

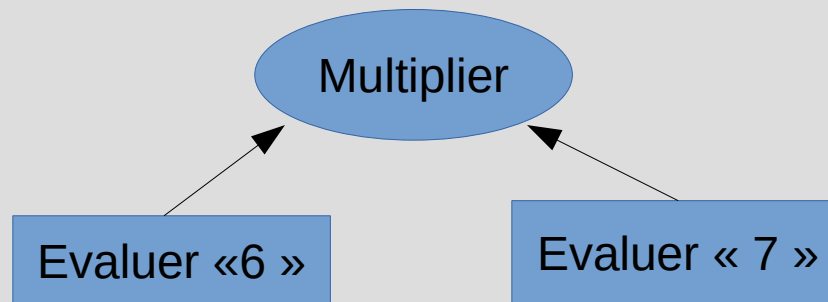
```
[[2]]  
[1] 6
```

```
[[3]]  
[1] 7
```

```
> eval(e)
```

```
[1] 42
```

L'expression contient l'arborescence des calculs à effectuer (on verra la signification de l'instruction et de la sortie plus tard)



On déclenche les calculs, le résultat est visualisé

Manipuler des expressions

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.
- 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 forme de chaîne de caractères.

```
> e <- quote(6*7)
> class(e)
[1] "call"

> e
6 * 7

> eval(e)
[1] 42
```

La visualisation du résultat

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 entourer de quotes,

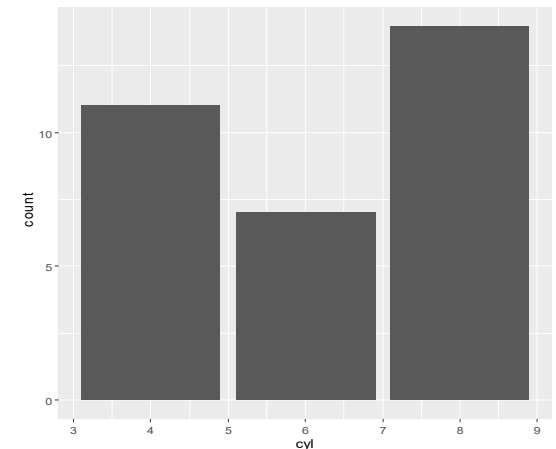
un peu plus compliqué parce que nécessitant des choix de présentation (un nombre, un data.frame)

ou être une structure encore plus complexe (un tableau, un graphique...).

- La fonction `print` est intelligente et réagit différemment selon ce qu'elle a à imprimer.

```
> library(ggformula)

> g <- mtcars %>% gf_bar( ~ cyl)
      # Il ne se passe rien car
      # '<-' n'affiche jamais rien
      # (résultat « invisible »).
> g
      # Afficher produit le dessin.
```



La visualisation du résultat

La fonction `invisible`

- Toute fonction produit un résultat, tout opérateur est une fonction, `<-` est un opérateur. Pourtant l'affectation ne montre rien.
- C'est qu'il est possible de définir des fonctions avec un résultat marqué comme invisible et qui, comme tel, ne sera pas affiché par `print`.
- La fonction `invisible` marque l'évaluation de son argument comme ne devant pas être affiché.

```
# L'affectation n'affiche rien
> x <- 6 * 7

# Pourtant elle restitue quelque chose
> y <- x <- 6*7
> y
[1] 42

# Les parenthèses affichent toujours
# le résultat du calcul
> (x <- 6*7)
[1] 42

# invisible évalue mais ne montre rien
> invisible(6*print(7))
[1] 7      # Le calcul a eu lieu
```


Symboles et objets

Principe numéro 1

Des contenus, des noms mais pas de contenants!

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** dès qu'on sait qu'on aura à la réutiliser.

```
> prenoms2017 <- import("IGoR/prenoms2017.dbf", as.is=TRUE)
```

Contrairement à la logique de beaucoup de langages de programmation, des noms comme *prenoms2017*, *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 côté des objets pour pouvoir en parler, tout comme un panneau routier indique l'entrée d'un village, mais ne contient pas le village.

Si on faisait :

```
> prenoms2017b <- prenoms2017
```

on ne ferait **aucune copie** de la table que nous venons de lire : on disposerait juste de deux noms pour identifier le même objet.

1ère étape

chargement des données en mémoire

```
prenoms2017 <-  
  import("IGoR/data/prenoms2017.dbf", as.is=TRUE)
```

mémoire

sexe	preusuel	annais	dpt	nombre
1	A	XXXX	XX	28
1	AADAM	XXXX	XX	24
1	AADEL	XXXX	XX	55
1	AADIL	1983	84	3
1	AADIL	1992	92	3
...



2ème étape enregistrement d'un nom

```
prenoms2017 <-  
  import("IGoR/data/prenoms2017.dbf", as.is=TRUE)
```

mémoire

prenoms2017

.GlobalEnv

Dictionnaire des
noms connus

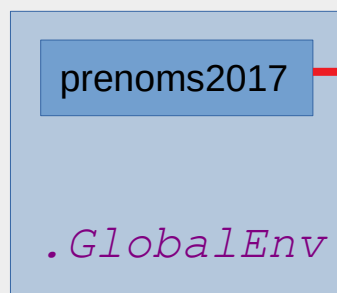
sexe	preusuel	annais	dpt	nombre
1	A	XXXX	XX	28
1	AADAM	XXXX	XX	24
1	AADEL	XXXX	XX	55
1	AADIL	1983	84	3
1	AADIL	1992	92	3
...

3ème étape

association du nom aux données

```
prenoms2017 <-  
  import("IGoR/data/prenoms2017.dbf", as.is=TRUE)
```

mémoire

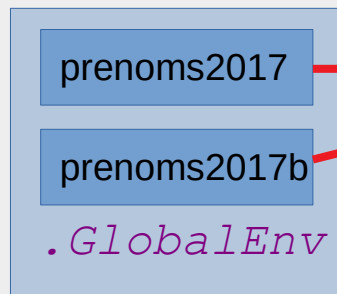


sexe	preusuel	annais	dpt	nombre
1	A	XXXX	XX	28
1	AADAM	XXXX	XX	24
1	AADEL	XXXX	XX	55
1	AADIL	1983	84	3
1	AADIL	1992	92	3
...

un autre nom pour les données

```
prenoms2017b <- prenom2017
```

mémoire



sexe	preusuel	annais	dpt	nombre
1	A	XXXX	XX	28
1	AADAM	XXXX	XX	24
1	AADEL	XXXX	XX	55
1	AADIL	1983	84	3
1	AADIL	1992	92	3
...

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"`.

```
> as.character(quote(x)) # on convertit le symbole x PAS sa valeur
[1] "x"
> as.name("x")          # fabriquer un symbole à partir d'une chaîne
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")                  # get(x) essaierait de faire get(42) !
[1] 42
> assign("x", pi)
> x
[1] 3.141593
```



Alliée aux fonctions de manipulation des chaînes de caractères, cette fonctionnalité est tentante pour des habitués du macro-langage SAS, (par exemple pour simuler un ensemble indicé de variables) mais il y a souvent une solution bien plus simple (les vecteurs notamment).

Principe numéro 2

Les objets sont “read only”

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 `<symbole>` donné :

- on construit une copie modifiée de la table
- on associe la copie modifiée au symbole `<symbole>`
- la précédente table associée au symbole `<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).

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`.

Une « modification » d'un objet n'est donc qu'un tour de passe-passe qu'il faut écrire explicitement

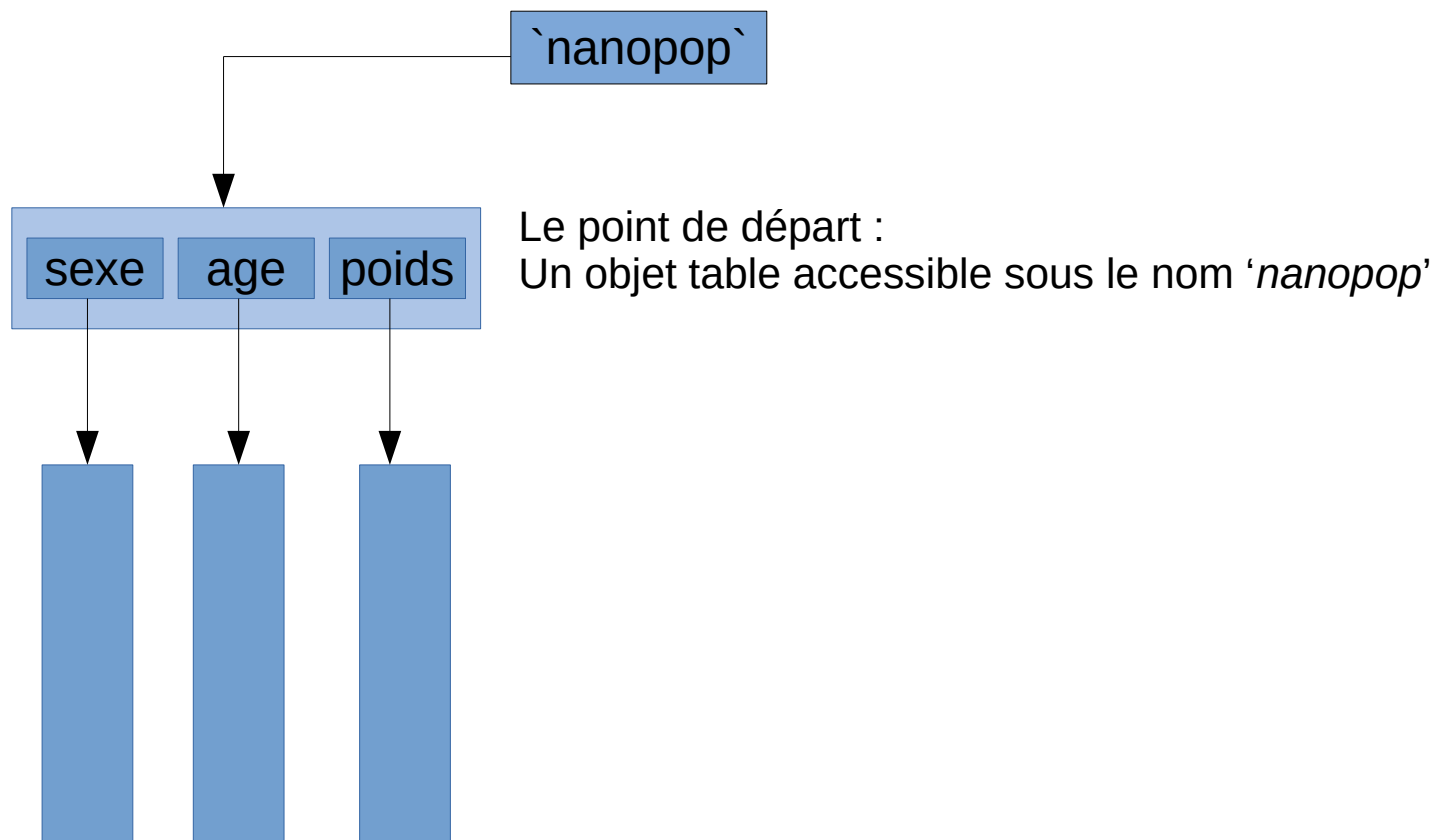
```
> nanopop %>% mutate(sexe2=as.numeric(sexe)) -> nanopop
```


Modifier un objet de R

Un exemple (1/3)

Pour rajouter une colonne à une table, ici la conversion en numérique de la colonne 'sexe' qui est du type « facteur »:

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

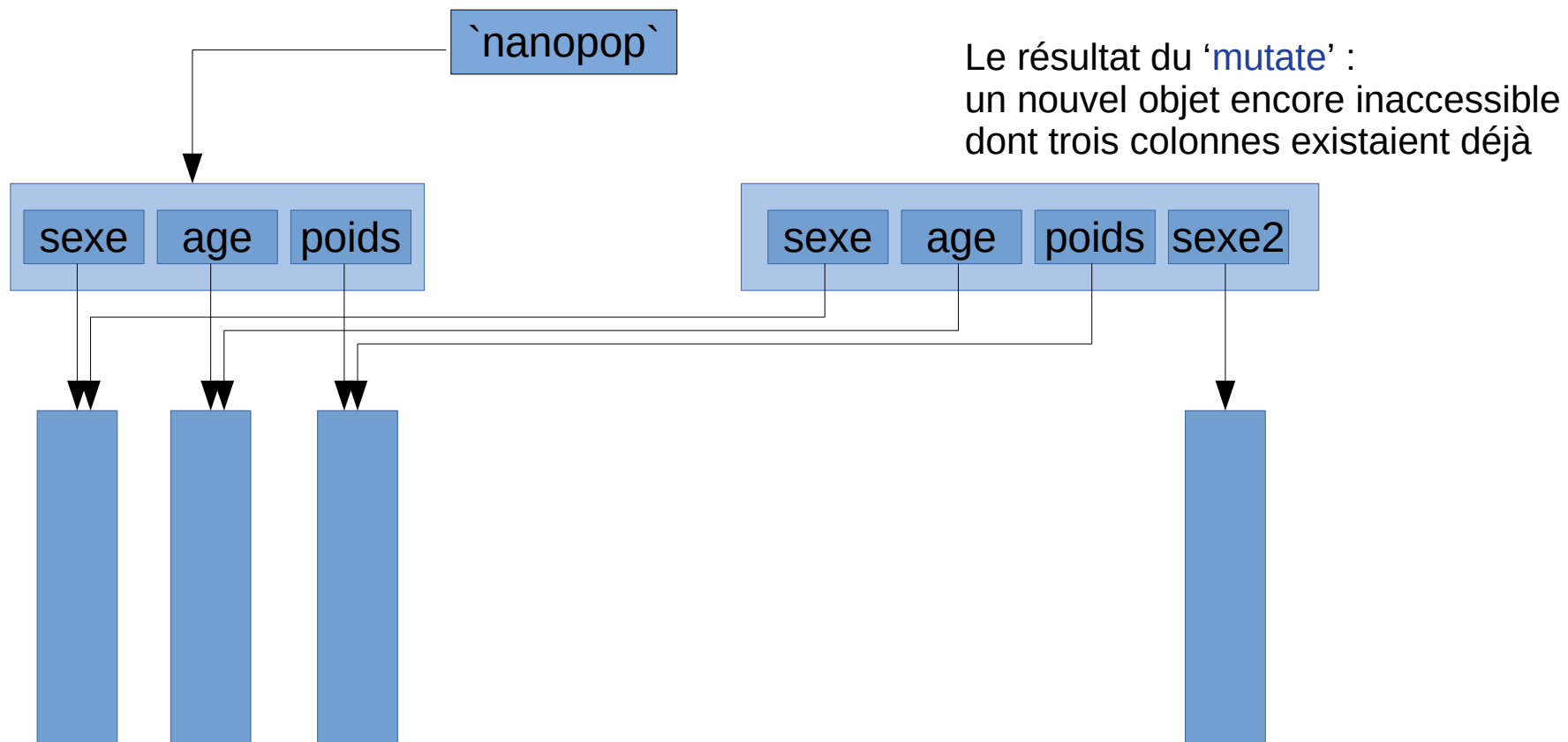


Modifier un objet de R

Un exemple (2/3)

Pour rajouter une colonne à une table, ici la conversion en numérique de la colonne 'sexe' qui est du type « facteur »:

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



Modifier un objet de R

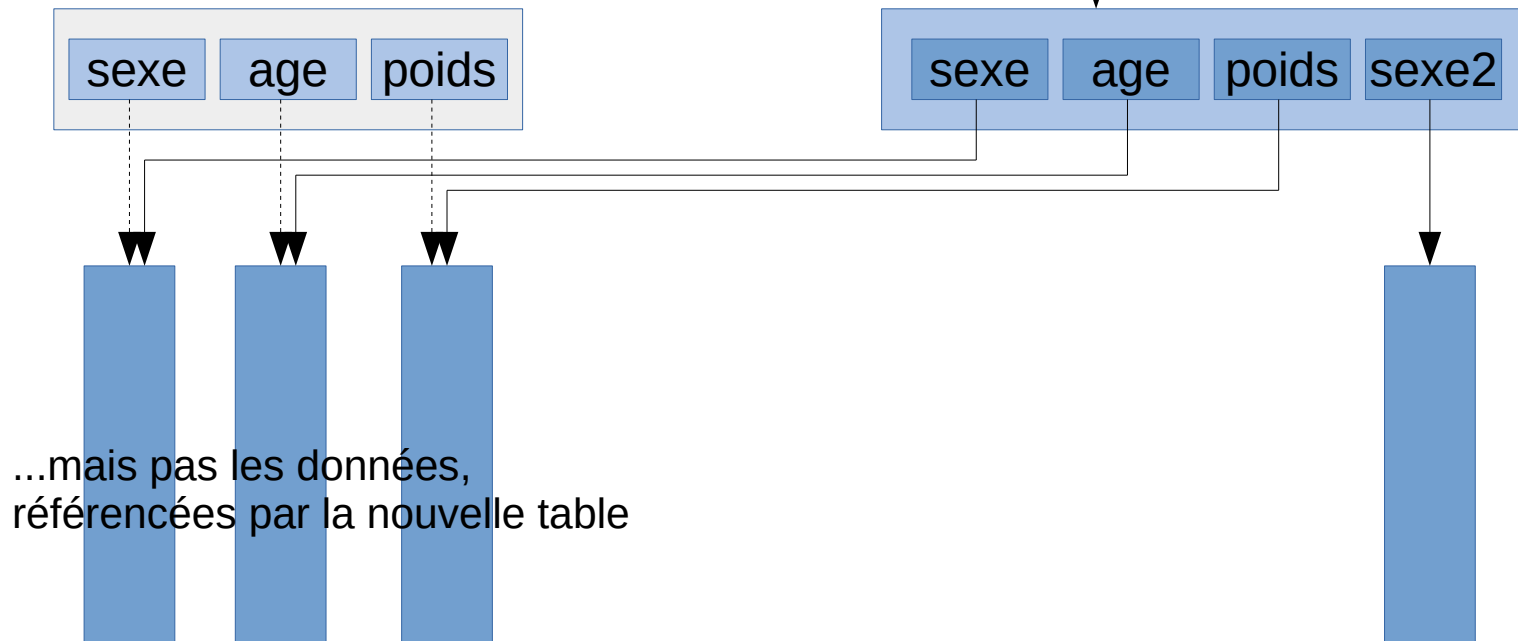
Un exemple (3/3)

Pour rajouter une colonne à une table, ici la conversion en numérique de la colonne 'sexe' qui est du type « facteur »:

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

L'objet n'est plus accessible
Il sera supprimé automatiquement...

Le résultat du 'mutate'
est rendu accessible sous le nom '*nanopop*'



...mais pas les données,
référéncées par la nouvelle table

Les grands principes du langage R

Le statique : les données

- 1) Pas de variables, mais des **objets et des symboles**
- 2) **Un objet n'est plus modifiable** après sa création
- 3) 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 n'est qu'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 unique fonction.

- **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 paraphrase alourdit les programmes et floute la vision.

- **Soignez l'esthétique !**

Votre programme est votre bébé : gardez le propre, bien proportionné et beau. Plus il sera beau plus il sera attachant et incitera à lire votre code ou à lui faire confiance.

En guise d'introduction du fil rouge de la formation

Lire 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`

Le fil rouge : le problème à résoudre

Exploiter un fichier XML complexe

L'entrant :

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

Comment faire pour :

- cartographier les points de vente
- visualiser l'évolution des prix

...

Revenir à une logique de data frame !

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><pdv_liste>
<pdv id="48000001" latitude="4451600" longitude="347800" cp="48000" pop="R">
  <adresse>ZAC DE RAMILLES</adresse>
  <ville>MENDE</ville>
  <horaires automate-24-24="1">
    <jour id="1" nom="Lundi" ferme=""/>
    <jour id="2" nom="Mardi" ferme=""/>
    <jour id="3" nom="Mercredi" ferme=""/>
    <jour id="4" nom="Jeudi" ferme=""/>
    <jour id="5" nom="Vendredi" ferme=""/>
    <jour id="6" nom="Samedi" ferme=""/>
    <jour id="7" nom="Dimanche" ferme=""/>
  </horaires>
  <services>
    <service>Restauration à emporter</service>
    <service>Carburant additivé</service>
    <service>Restauration sur place</service>
    <service>Toilettes publiques</service>
    <service>Station de gonflage</service>
    <service>Boutique non alimentaire</service>
    <service>Services réparation / entretien</service>
    <service>Vente de gaz domestique (Butane, Propane)</service>
    <service>Location de véhicule</service>
    <service>Piste poids lourds</service>
    <service>DAB (Distributeur automatique de billets)</service>
    <service>Lavage manuel</service>
    <service>Vente de fioul domestique</service>
    <service>Vente de pétrole lampant</service>
    <service>Automate CB 24/24</service>
  </services>
  <prix nom="Gazole" id="1" maj="2020-01-07T07:04:05" valeur="1468"/>
</pdv>
</pdv_liste>
```

XML acte I – scène 1

Un premier déchiffrement des points de vente

Lignes d'en tête de chacun des points de vente

```
# A tibble: 145 x 2
  pdv2                                no
  <chr>                             <int>
1 id=22000001 latitude=4852000 longitude=-279200 cp=22000 pop=R      1
2 id=22000002 latitude=4852200 longitude=-273200 cp=22000 pop=R      2
3 id=22000003 latitude=4849800 longitude=-274700 cp=22000 pop=R      3
4 id=22000004 latitude=4850836 longitude=-276785 cp=22000 pop=R      4
5 id=22000005 latitude=4851449 longitude=-273652 cp=22000 pop=R      5
6 id=22000009 latitude=4849806 longitude=-274533 cp=22000 pop=R      6
7 id=22000010 latitude=4850836 longitude=-276785 cp=22000 pop=R      7
8 id=22000011 latitude=4849806 longitude=-274533 cp=22000 pop=R      8
9 id=22100001 latitude=4846400 longitude=-208700 cp=22100 pop=R      9
10 id=22100004 latitude=4847100 longitude=-204400 cp=22100 pop=R     10
# ... with 135 more rows
```


Important	4
Utile	2
Eviter	1
Important	3
Utile	1
Rare	2

```
dm <- function(...,expr) {
  c <- as.list(match.call())
  n <- c[2:(length(c)-1)]
  t <- unlist(Map(Negate(is.name),n))
  if (any(t)) stop("Bad parameter name.")
  function(...) {
    l <- list(...)
    if (length(l)!=length(n)) stop("Wrong number of arguments.")
    names(l)<- if (is.null(names(l))) as.character(n)
               else ifelse(names(l)=="",as.character(n),names(l))
    e <- do.call(substitute,list(c$expr,l))
    eval(e,envir=-2)
  }
}
```

Séquence R1

Les vecteurs

One is a crowd

Créer un vecteur (1/2)

- 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)**.
Aussi 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
```

Ce qui suit est
le premier élément d’un vecteur

- Un vecteur peut être vide, par exemple à la suite d’une sélection infructueuse. La notation est le type du vecteur suivi de (0) :

```
logical(0)
character(0)
```

Créer un vecteur (2/2)

Au-delà de ces façons de créer un vecteur, on pourra aussi utiliser :

- `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 très utilisée 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, la fabrication de suite d'entiers consécutifs, etc...

```
> rep(TRUE, 5)
[1] TRUE TRUE TRUE TRUE TRUE
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 1:(1-1)
[1] 1 0
> seq_len(10)
[1] 1 2 3 4 5 6 7 8 9 10
> seq_len(0)
integer(0)
```

: est un opérateur !
Danger !

Exercice rapide

Autour des dates et des vecteurs

- 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 1^{er} janvier 2017 ?

Accéder à une partie d'un vecteur (1/2)

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,
- un **vecteur de nombres entiers négatifs** : pour extraire tous les éléments sauf ceux de numéro cités
- un **vecteur de booléens** pour spécifier quels éléments doivent être conservés (**TRUE**) ou non (**FALSE**).

```
> (v <- 101:110)
[1] 101 102 103 104 105 106 107 108 109 110

# Les éléments du 3ème au 5ème
> v[3:5]
[1] 103 104 105

# idem, mais en en précisant l'ordre
> v[5:3]
[1] 105 104 103

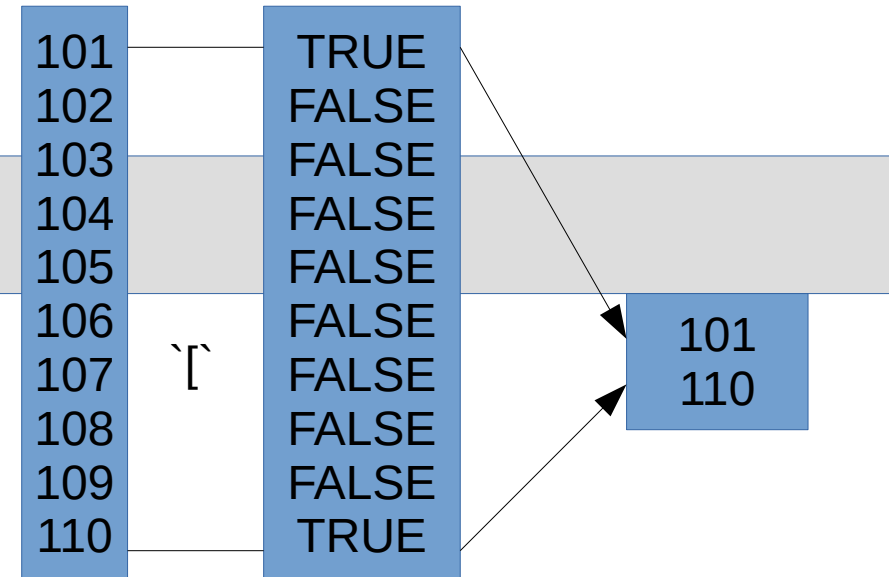
# Une sélection vide
> v[integer(0)]
integer(0)

# Tout sauf les élts du 3ème au 5ème
> v[-3:-5]
[1] 101 102 106 107 108 109 110
```

Accéder à une partie d'un vecteur (2/2)

```
> 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
```

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 côté gauche de l'opérateur, en particulier un résultat d'un appel de fonction.

```
> readLines("data2/Z1.txt") [111] %>% str_split("[,.]")  
[[1]]  
[1] "Le fichier texte"  
[2] " lorsqu'il apparait apporte la possibilité de permrmettre à un humain de soumettre un"  
[3] " Il offre également la possibilité de supprimer et d'ajouter une ligne"  
[4] " et cela dès les cartes perforées"  
[5] " <<<message(\"GAGNANT!\")>>>Cette fonctionnalité a été reprise par des logiciels comm  
[6] ""
```

`readLines` est une fonction de base qui lit un fichier texte et le restitue sous forme de vecteur où chaque élément contient une ligne de texte.

`str_split` est une fonction de **stringr** qui éclate une chaîne de caractères en une liste de chaînes de caractères, en se servant d'une **expression régulière** pour définir le séparateur à trouver entre les chaînes. Ici soit une virgule soit un point.

Exercice rapide

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 de la semaine dont le nom se termine par « di » ? 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.**



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

Mal utilisée, elle peut en outre conduire à des résultats incorrects.

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

# Position des lettres N à P
> which(LETTERS>="N" & LETTERS<="P")
[1] 14 15 16

# Lettres de N à P
# Version naïve
> LETTERS[which(LETTERS>="N" & LETTERS<="P")]
[1] "N" "O" "P"

# Version plus simple sans 'which'
> LETTERS[LETTERS>="N" & LETTERS<="P"]
[1] "N" "O" "P"

> length(LETTERS[which(LETTERS>="Z")])
[1] 1      # correct
> length(LETTERS[-which(LETTERS>="Z")])
[1] 25     # correct (le complément)
> length(LETTERS[which(LETTERS>"Z")])
[1] 0      # correct
> length(LETTERS[-which(LETTERS>"Z")])
[1] 0      # erroné !
```

Le recyclage des éléments dans les opérations vectorielles

- 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 !

```
> 1:10 + rep(100,10)
[1] 101 102 103 104 105 106 107 108 109 110
```

- Faire la même chose avec deux vecteurs de longueurs différentes ne perturbe pas R. C'est le cas quand utilise un scalaire dans un calcul impliquant un vecteur. R **"recycle" le vecteur le plus court pour en faire (virtuellement) un vecteur de la même longueur que le plus long**.

```
> 1:10 + 100
[1] 101 102 103 104 105 106 107 108 109 110
```

```
> 1:10 + c(100,200)
[1] 101 202 103 204 105 206 107 208 109 210
```

```
> 1:10 + c(100,200,300)
```

Warning message:

In 1:10 + c(100, 200, 300) :

la taille d'un objet plus long n'est pas multiple de la taille d'un objet plus court

```
[1] 101 202 303 104 205 306 107 208 309 110
```

Le recyclage : pas toujours une bénédiction !

Un exemple (malheureux)

Le code suivant fonctionne :

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

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6

Celui là aussi ?

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

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Honda Civic	30.4	4	75.0	52	4.38	1.615	17.82	0	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	16.46	0	1	4	1
Porsche 914-2	26.0	4	120.0	90	4.47	2.140	16.69	0	1	4	1
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6

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

n
1 18

Il n'y a pas que les opérateurs de vectorisés :

la fonction `paste`

- La **concaténation de chaînes de caractères** se fait avec la fonction `paste0` qui est en fait une redéfinition à la marge de la fonction plus générale `paste` qui introduit un séparateur entre les éléments concaténés. 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"

> 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 "écrase" le vecteur en 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
```

- 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
```

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

La fonction cachée

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.
- 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.
- En toute rigueur, pour le programmeur, un vecteur ne peut donc être ni modifié, ni étendu. Ce qui se passe sous le capot est bien sûr une autre histoire...

La notation `v[3:5] <- 0` est en fait strictement équivalente aux lignes 2 et 3 de :

```
> v <- 101:110
> temp <- v                # 2
> v <- `[<-`(temp,3:5,0)    # 3
> v
[1] 101 102  0   0   0 106 107 108 109 110
```

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

Démonstration

- La « modification » ne peut être optimisée parce que les données sont référencées ailleurs.
- La « modification » est optimisée : pas de copie parce que personne d'autre n'utilise l'original.

```
> (v <- LETTERS[1:5])  
[1] "A" "B" "C" "D" "E"
```

```
> tracemem(v)  
[1] "<0F57EF20>"
```

```
> w <- v  
> tracemem(w)      Assigner n'est pas copier  
[1] "<0F57EF20>"
```

```
> v[1] <- "a"  
tracemem[0x0f57ec20 -> 0x0f57eaa0]:
```

```
> tracemem(v)  
[1] "<0F57EAA0>"      Modifier c'est copier  
> tracemem(w)  
[1] "<0F57EC20>"
```

**A faire sous RGui,
Ne marche pas sous RStudio
Pourquoi ?**

```
> (v <- LETTERS[1:5])  
[1] "A" "B" "C" "D" "E"
```

```
> tracemem(v)  
[1] "<01422EA0>"
```

```
> v[1] <- "a"
```

```
> tracemem(v)  
[1] "<01422EA0>"
```

Des fonctions du côté gauche de l'assignation ? (1/2)

- La présence d'une fonction du côté gauche d'une assignation, comme avec `[`, n'est pas un cas isolé.
- De telles fonctions permettent de positionner certains champs d'une liste, d'une table etc. en accompagnant l'affectation d'une valeur, de contrôles ou de traitements complémentaires. Généralement on dispose d'un couple de fonctions :

```
> v <- 101:110
> v[3:5]
[1] 103 104 105
```

Lire, la fonction `[`

```
> v[3:5]<- 0
> v
[1] 101 102 0 0 0 106 107 108 109 110
```

Ecrire, la fonction `<-`

- Ce qui peut être réalisé par l'appel direct des fonctions correspondantes :

```
> v <- 101:110
> `[`(v,3:5)
[1] 103 104 105

> `[<-`(v,3:5,0)
[1] 101 102 0 0 0 106 107 108 109 110
```


Des fonctions du côté gauche de l'assignation ? (2/2)

- Il s'agit là d'une fonctionnalité qui peut être mobilisée par l'utilisateur. Mais **il s'agit juste d'une facilité syntaxique**. Lorsqu'un appel à une fonction f se trouve du côté gauche d'une assignation, R cherche une fonction $f<-$:

```
> x <- 1
> f(x) <- 0
impossible de trouver la fonction "f<-"
```

- Et si cette fonction existe, R travaille en trois étapes :
 - il donne un nouveau nom au premier argument de la fonction,
 - il appelle la fonction $f<-$ avec ce nom et le reste des arguments d'appels de f , plus l'opérande de droite de l'assignation,
 - il réassigne au premier argument de f le résultat de l'appel précédent.

NOTE : Le premier argument de f est donc nécessairement un symbole. Et, dans la définition de $f<-$. Par ailleurs, le dernier paramètre qui recevra l'opérande de droite, doit impérativement s'appeler 'value'.

```
> `colonnes<-` <- function(table,value) select_at(table, value)
> colonnes(mtcars)<- c("hp", "cyl")
> mtcars
```

	hp	cyl
Mazda RX4	110	6
Mazda RX4 Wag	110	6
...		

Les matrices

- 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) # un vecteur de 10 éléments...
> dim(m) <- c(2,5)           # ...sous forme d'une matrice à deux dimensions
> m
```

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

Recours à la fonction `dim<-`
Voir aussi `matrix`

Une matrice peut avoir plus de deux dimensions.

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

```
> m[2,5] # un élément
[1] 7
```

↓

```
> m[1,] # une ligne et toutes ses colonnes
[1] 0 4 3 0 2
```

↓

```
> m[,1:2] # un ensemble de colonnes et toutes leurs lignes
```

	[,1]	[,2]
[1,]	0	4
[2,]	5	9

Fin de la séquence
XML acte I – scène 1
Tout dans un vecteur

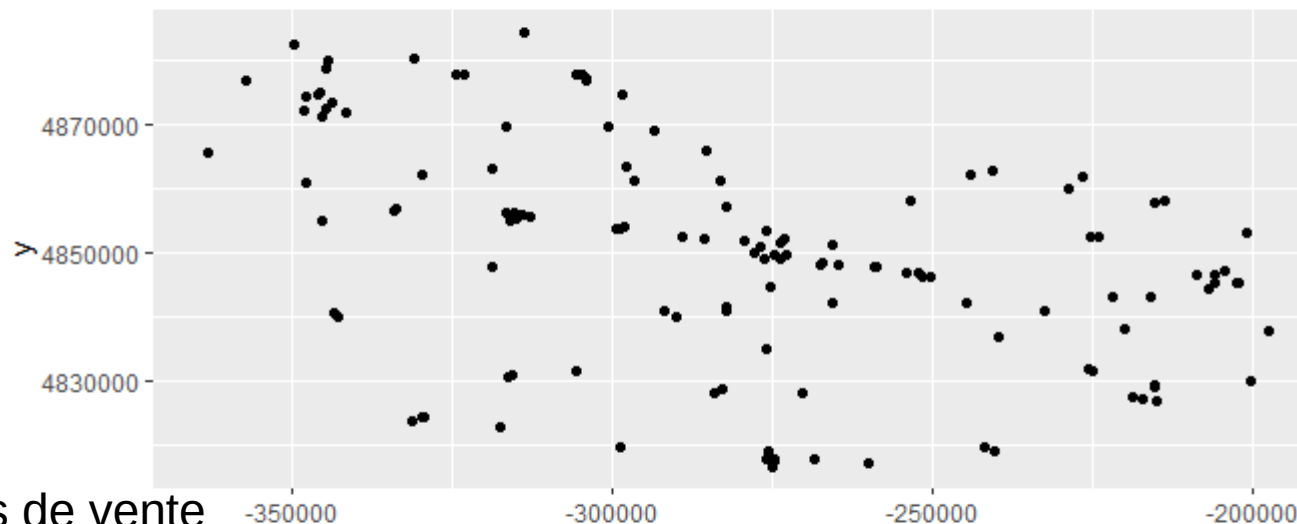
- Avec la fonction `readLines`, lire l'intégralité du fichier. Le résultat est un (très long) vecteur. La fonction `head` permet d'en visualiser le début.

⇒ **Séquence R1 « les vecteurs »**

- Avec la fonction `str_detect` de **stringr**, ne conserver que les lignes contenant 'p dv ' (avec un espace à la fin, donc ne contenant pas 'p dv_liste'). `str_detect` est une fonction qui accepte une expression régulière mais nous n'en avons pas besoin ici.
- Avec la fonction `str_replace_all`, éliminer : ' <p dv ' (deux espaces avant, un après), les '>', les double quotes.
- Transformer en `data.frame` (fonction du même nom, sans facteurs). En profiter pour ajouter une colonne contenant le numéro de la ligne (utiliser l'opérateur `:` ou mieux).

XML acte I – scène 2

La localisation des points de vente



Données caractéristiques des points de vente

A tibble: 145 x 7

	pdv2			no	cp	id	latitude	longitude	pop
	<chr>			<int>	<chr>	<chr>	<chr>	<chr>	<chr>
1	id=22000001	latitude=4852000	long~	1	22000	2200~	4852000	-279200	R
2	id=22000002	latitude=4852200	long~	2	22000	2200~	4852200	-273200	R
3	id=22000003	latitude=4849800	long~	3	22000	2200~	4849800	-274700	R
4	id=22000004	latitude=4850836	long~	4	22000	2200~	4850836	-276785	R
5	id=22000005	latitude=4851449	long~	5	22000	2200~	4851449	-273652	R
6	id=22000009	latitude=4849806	long~	6	22000	2200~	4849806	-274533	R
7	id=22000010	latitude=4850836	long~	7	22000	2200~	4850836	-276785	R
8	id=22000011	latitude=4849806	long~	8	22000	2200~	4849806	-274533	R
9	id=22100001	latitude=4846400	long~	9	22100	2210~	4846400	-208700	R
10	id=22100004	latitude=4847100	long~	10	22100	2210~	4847100	-204400	R

... with 135 more rows

Important

4

Utile

2

Rare

1

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l)!=length(n)) stop("Wrong number of arguments.")  
    names(l)<- if (is.null(names(l))) as.character(n)  
               else ifelse(names(l)== "",as.character(n),names(l))  
    e <- do.call(substitute,list(c$expr,l))  
    eval(e,envir=-2)  
  }  
}
```

Séquence R2

Les listes

str
NULL

Créer une liste

`list`, `str`

- 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 :

Une liste contenant des champs des types élémentaires du langage R: double, entier, booléen, chaîne de caractères, fonction et... 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))
```

- **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
```

Accéder à un élément 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),
 - soit **un nom d'élément** (quand la liste contient des éléments nommés),
 - 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[[5]]  
function (... , na.rm = FALSE) .Primitive("sum")  
  
> liste[["f"]]  
function (... , na.rm = FALSE) .Primitive("sum")  
  
> 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,
 - un vecteur de booléens,
 - un vecteur de noms d'éléments.

```
> liste[5]
$f
function (... , na.rm = FALSE) .Primitive("sum")
```

```
> liste[-1:-4]
$f
function (... , na.rm = FALSE) .Primitive("sum")

$l
$l[[1]]
[1] 1

$l[[2]]
[1] 2
```


« Modifier » une liste

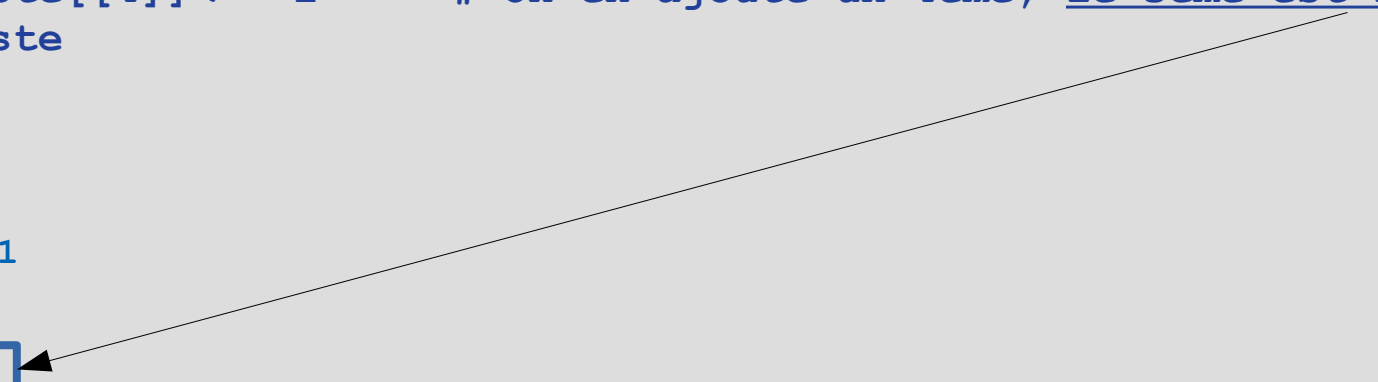
- Les opérateurs `[]` et `$` sont “réversibles”. On peut ainsi positionner un élément dans une liste par sa position ou son nom avec `[]<-`, ou par son nom `$<-`

```
> liste <- list(1,2)    # Deux éléments 1 et 2
> liste[[2]]<- -1      # On remplace le 2ème élément
> liste[[4]]<- -2      # On en ajoute un 4ème, le 3ème est indéfini
> liste
[[1]]
[1] 1

[[2]]
[1] -1

[[3]]
NULL

[[4]]
[1] -2
```



```
> liste2 <- list()
> liste2$a <- 1        # La liste doit préalablement exister
```

- NOTE : L'opérateur `[]` est également “réversible” sur les listes. **Il permet de modifier** une série d'éléments d'une liste à partir d'une série de valeurs fournies en même nombre (ou qui sera complétée par réplication). **Il ne permet pas d'insérer** des éléments (avec une liste plus longue coté droit) **ni de réduire** une partie d'une liste (avec une liste plus courte du coté droit).

Rien : NULL

- Contrairement aux vecteurs, **on peut “supprimer” un élément d’une liste** avec l’opérateur `[<-` (ou l’opérateur `$<-`). Il suffit d’affecter à l’élément à supprimer l’objet vide `NULL`.

```
> liste[[1]]<- NULL
> length(liste)
[1] 3
```

```
> liste
[[1]]
[1] -1
```

L’ancien 2ème élément est devenu le premier

```
[[2]]
[1] NULL
```

Le « trou » créé par la création de l’ex 4ème élément compte pour 1, et restera même si on supprime ce qui le suit.

```
[[3]]
[1] -2
```

- `NULL` est assimilable à une liste vide (« rien ») et ne doit pas être confondu avec `NA` (« je ne sais pas »). Cf. la réponse à « qu’avez vous mangé ce matin ? » (en général une liste d’aliments) : il faut faire la différence entre « rien » et l’absence de réponse (« j’ai oublié »).
- Le test d’existence d’un élément d’une liste peut se faire en testant l’égalité avec l’objet `NULL`** mais, comme avec les valeurs manquantes, pas directement :
soit en utilisant la fonction `is.null`,
soit en testant la longueur de l’élément qui sera alors 0.

```
> is.null(liste$a)
[1] TRUE
```

Convertir une liste en vecteur : `unlist`

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

> list(1, "a") %>% unlist()
[1] "1" "a"

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

Lorsque les éléments de la liste sont nommés, les éléments du vecteur résultat sont -par défaut (voir paramètre ad'hoc)- également nommés.

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

Comme pour les éléments d'une liste, les éléments d'un vecteur peuvent être nommés

Il y a liste et liste !

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

```
> liste <- list(a=1,b=2)
> library(pryr)
> inspect(liste)
<VECSXP 0x12952cb8>
  <REALSXP 0x12952c40> # vecteur 1
  <REALSXP 0x12952c68> # vecteur 2
attributes:
  <LISTSXP 0xcfeabf8>
tag:
  <SYMSXP 0x6091fd8>
car:
  <STRSXP 0x12952ce0>
    <CHARSXP 0x70c0860>
    <CHARSXP 0x87fe570>
cdr:
  NULL
```

Les noms

```
> is.vector(liste)
[1] TRUE
```

La liste des attributs

Pour l'exercice

RAPPEL : Les expressions régulières

- 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.
- Les éléments de syntaxe les plus fréquemment utilisés :

```

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

```

Fin de la séquence

XML acte I – scène 2

De la liste au data frame

- 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.

⇒ **Séquence R2 « Les listes »**

- 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 : une 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.

Fin de la première demi-journée

Résumé

- Les principes fondamentaux
- Les structures de base :
 - Vecteur
 - Liste
- Quelques opérateurs :
 - Vecteurs et listes

[

[<-

- Listes

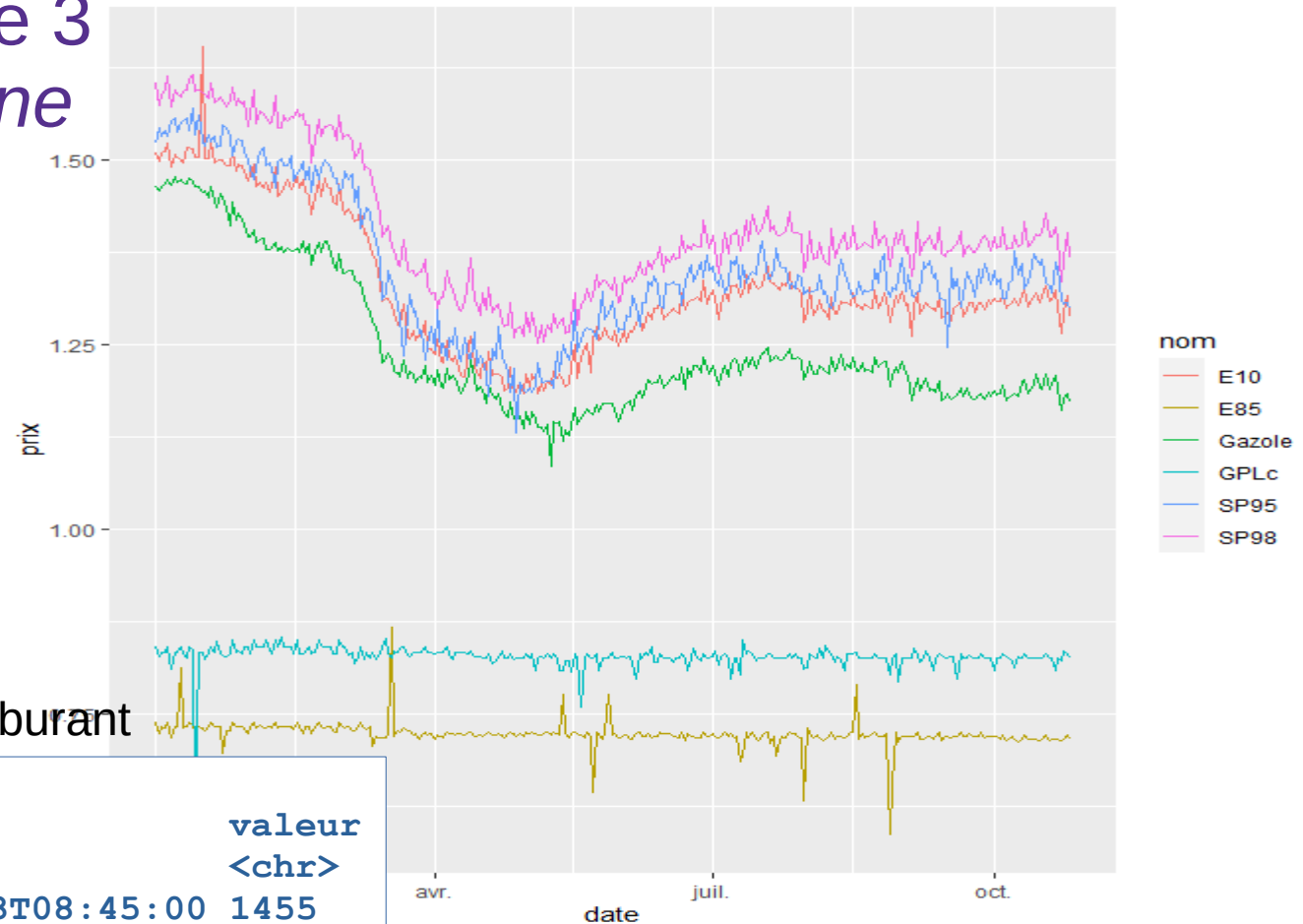
[[

[[<-

\$

XML acte 1 – scène 3

L'évolution moyenne des prix



Relevés de prix par pompe et carburant

```
# A tibble: 30,412 x 5
```

	no	nom	id	maj	valeur
	<int>	<chr>	<chr>	<chr>	<chr>
1	1	Gazole	1	2020-01-03T08:45:00	1455
2	1	Gazole	1	2020-01-07T08:45:00	1465
3	1	Gazole	1	2020-01-15T08:45:00	1459
4	1	Gazole	1	2020-01-17T08:45:00	1455
5	1	Gazole	1	2020-01-21T08:45:00	1449
6	1	Gazole	1	2020-01-22T08:45:00	1439
7	1	Gazole	1	2020-01-28T08:45:00	1425
8	1	Gazole	1	2020-01-29T08:45:00	1419
9	1	Gazole	1	2020-01-31T08:45:00	1405
10	1	Gazole	1	2020-02-03T08:45:00	1399

```
# ... with 30,402 more rows
```


Important

5

Utile

4

Eviter

1

Important

1

Utile

2

Rare

2

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l)!=length(n)) stop("Wrong number of arguments.")  
    names(l)<- if (is.null(names(l))) as.character(n)  
               else ifelse(names(l)== "",as.character(n),names(l))  
    e <- do.call(substitute,list(c$expr,l))  
    eval(e,envir=-2)  
  }  
}
```

Séquence F1

Définir une fonction

Définir une fonction c'est facile !
Tout est fonction
Tout redéfinir est facile

"To understand computations in R, two slogans are helpful:
• *Everything that exists is an object.*
• *Everything that happens is a function call."*

— John Chambers

Créer une fonction c'est facile !

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**

5)... et consommer sans modération

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

```
> Somme(1,100)  
[1] 5050
```

NOTE : `function` n'est pas vraiment un mot clé, mais... une fonction dont, certes, la syntaxe est un peu spéciale. Comme toute fonction elle crée un objet. **Une fonction est un objet, juste un objet d'un type particulier.**

L'utilisation des fonctions



Bien distinguer la définition d'une fonction et son utilisation :

- Un nom seul est interprété comme l'accès à l'objet qu'il référence, dans le cas d'une fonction c'est le **code de la fonction**.
- Un nom suivi de parenthèses, éventuellement sans rien dedans, correspond à l'appel de la fonction avec les arguments précisés entre parenthèses.

Dans un pipe, du côté droit on trouve un appel de fonction qui sera complété par le résultat du calcul spécifié coté gauche : l'usage des parenthèses devrait donc être obligatoire même s'il n'y a pas d'argument complémentaire. Avec le pipe de la 4.1 on n'aura plus le choix.

```
> Somme <- function(x,y)
  sum(c(-1,1)*(x:y)^2)

# Entrer le nom seul veut dire qu'on
# veut savoir à quoi il est associé
> Somme
function(x,y) sum(c(-1,1)*(x:y)^2)

# Un nom suivi de parenthèses est un
# appel de fonction
> Somme()
Erreur dans Somme() :
  l'argument "début" est manquant,
  avec aucune valeur par défaut

> 1 |> print
Erreur : L'opérateur pipe nécessite
un appel de fonction comme membre de
droite
> 1 |> print()
[1] 1
```

L'expression formant la fonction...

- Soit **UNE fonction** éventuellement faisant appel à d'autres fonctions ou opérateurs de façon emboîtée, enchaînés par un pipe ou appels imbriqués.
- Soit l'appel de **la fonction « accolades »** à la syntaxe spéciale où les arguments sont séparés par des point-virgules ou des passage à la ligne et qui évalue successivement tous ses arguments pour donner en résultat le résultat de l'évaluation du dernier.

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

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

```
> 100 %>%  
{t <- 1:. ;sum(c(-1,1)*t^2)}  
[1] 5050
```



Les accolades ne sont pas une notation syntaxique mais une fonction à part entière utilisable partout :

Nommer ses fonctions

Un nom de fonction est un symbole et est donc sujet aux mêmes contraintes :

- commencer par une lettre ou un point
- contenir des lettres, des chiffres, des points ou un underscore _.

La notion de “lettre” est moins simple qu’il n’y paraît car au delà des **caractères de a à z (et majuscules)**, on peut parfois utiliser des caractères spécifiques à la langue déclarée lors de l’installation (les caractères accentués en français) ou des caractères issus d’autres alphabets. Mais cette possibilité est hautement dépendante du système d’exploitation et de son paramétrage. Utiliser une fonction nommée λ est parfois possible mais pas forcément une bonne idée.

Nommer ses fonctions

Quelques bonnes pratiques

- **Utiliser des noms parlants** : 'f' n'est bon que pour une démonstration ou un test.
- **Ne pas préfixer** par quelque chose comme `fonction_` ou `f_...` . Le contexte de déclaration et d'usage est suffisamment explicite pour qu'on voie au premier coup d'oeil qu'il s'agit d'une fonction.
- **Eviter les points**. Bien qu'il y ait une fonction `file.choose` en contre exemple et que rien n'interdit leur usage, les points servent à la définition de fonctions génériques telles `print.ggplot` (fonction qui se charge de l'impression d'un objet généré par **ggplot2**). Noter que les objets dont les noms commencent par un point ne sont pas visibles dans la fenêtre environnement de RStudio.
- **Faire un usage raisonné des majuscules** : une habitude, indépendante de R, est de réserver les noms complètement en majuscules à des données constantes sur tout le programme.
- **Utiliser des noms composés** d'un verbe suivis du type d'objet, par exemple : `lire_fichier`, `generer_boxplot`...

On peut aussi utiliser le "camelCase" : `lireFichier`, `genererBoxplot`... Mais il faut prendre garde qu'en cas d'incursion dans les couches objets S4 et R6, cette notation y est l'équivalent de l'usage du point avec les fonctions génériques,

Nommer les paramètres

- Les noms des paramètres sont, comme les noms des fonctions, des symboles qu'on peut choisir arbitrairement.
- Néanmoins, si le code de la fonction est un peu long (ce qui n'est pas souhaitable!) il peut être intéressant d'avoir une convention de nommage qui permet de séparer au premier coup d'oeil ce qui est paramètre de ce qui ne l'est pas. Il n'existe pas de préconisation standard mais on peut penser à ;
 - Préfixer par p_ : p_dep
 - Préfixer par un point, c'est qui est fait dans *dplyr*
 - Préfixer par un article : leDep (theDep en anglais)
 - Utiliser une majuscule au début : Dep

Le tout sera de se tenir aux choix faits !

- La limite de ce type de convention est qu'elle ne s'applique qu'aux premiers paramètres lorsque leurs valeurs seront passées par position. **Pour les suivants où la valeur sera passée par le nom du paramètre, mieux avoir quelque chose de parlant pour l'utilisateur plutôt que pour le programmeur de la fonction.**
- Enfin, s'il y a une éventualité que la fonction sorte de l'hexagone, par exemple en faisant partie d'un package, il faut s'habituer à tout écrire en anglais, nom de la fonctions, des paramètres et, bien sûr, commentaires et documentation.

Exercice rapide

Une première fonction

- Transformez en fonction de deux paramètres le code suivant, pour qu'on puisse l'utiliser avec n'importe quel département (ou liste de départements) et une autre variable que *'indnatm'*.

```
fst("IGoR/data/nais2017.fst") %>%  
  [. $depnais %in% c("17","86"), c("depnais","indnatm")] %>%  
  group_by_at(c("depnais","indnatm")) %>%  
  summarise(n=n())
```

- Testez sur 976, *'indnatp'*. Résultat attendu :

```
> setwd("V:/PALETTES")  
> compterDepts("976","indnatp")  
`summarise()` regrouping output by 'depnais' (override with `groups` argument)  
# A tibble: 2 x 3  
# Groups:   depnais [1]  
  depnais indnatp      n  
  <chr>   <chr>   <int>  
1 976     1       4747  
2 976     2       4728
```

NOTE :

Le message sur `summarise` est émis par les dernières versions de *dplyr* : il est juste informatif pour signaler que la table finale est encore groupée.

Au delà du standard...

- Au delà des noms constitués de façon normale, il est possible d'avoir des symboles qui contiennent des caractères en principe interdits. Ceci arrive parfois quand on génère des tables sans prendre trop garde aux noms des colonnes. :

```
> import("IGoR/data/poplegale_6815.sas7bdat") %>%
  group_by(D) %>%
  summarise(sum(PMUN15, na.rm=TRUE)) -> r
> r %>% head(1)
# A tibble: 1 x 2
  D      `sum(PMUN15, na.rm = TRUE)`
<chr>      <dbl>
1 01      631877
```

- De tels noms sont généralement (pas avec *ggformula*, par exemple!) **manipulables à condition de les inclure entre backticks (ou backquote, l'accent grave, alt-Gr-7)**, pas vraiment commodes. Ou parfois entre quotes, plus commodes mais avec des risques de confusion plus qu'élevés entre nom de symbole et chaîne de caractère!

```
> r %>% select(`sum(PMUN15, na.rm = TRUE)` ) %>% head(1)
# A tibble: 1 x 1
  `sum(PMUN15, na.rm = TRUE)`
      <dbl>
1      631877
```

Des fonctions absolument partout !

mais des fonctions aux noms non standard

- Les noms de fonction non standard ne font que compliquer leur usage. Néanmoins **les multiples notations syntaxiques de R cachent en fait des appels à des fonctions**. Et ces fonctions ont systématiquement des noms qui sortent franchement du standard.
 - Les opérateurs : +, -, *, / et d'autres qu'on verra ensuite

```
> `*`  
function (e1, e2) .Primitive("*")  
> .Primitive("*") (6,7)  
[1] 42
```

- L'assignation, les accolades, les crochets, les parenthèses, le « mot-clé » function...

```
> `<-`  
.Primitive("<-")  
> `{`  
.Primitive("{")  
> `(`  
.Primitive("(")  
> `function`  
.Primitive("function")
```

- `.Primitive` est la fonction qui réalise l'interface entre le code R et le code C du noyau.

Construire un opérateur : % ... %

- Un cas particulier de syntaxe dans les noms de fonction est l'usage du caractère “pourcent” comme dans `%>%` (on peut mettre n'importe quel caractère différent de % entre les %, y compris des espaces). **Le caractère “pourcent” offre la possibilité de définir ses propres opérateurs binaires**, Cela fournit juste une syntaxe alternative à tout appel de fonction à deux arguments.

Par exemple, pour définir un opérateur de concaténation de chaînes de caractères :

```
> `%+%` <- function(left,right) paste0(left, right)
> "un" %+% " " %+% "petit" %+% " " %+% "essai"
[1] "un petit essai"
```

- Enfin, il a existé un opérateur `:=`. Bien qu'il ne soit plus défini en R, cette notation continue à être reconnue comme syntaxiquement valide et l'opérateur a été (re)défini dans certains packages comme **dplyr** (marginale), **data.table** (massivement), et donc ne disparaîtra pas. Pour ceux qui ne font pas de complexes, ce peut être une opportunité de définir quelque chose porté par une syntaxe synthétique. On verra un exemple plus tard.

Exercice rapide

Un opérateur

- Construire l'opérateur manquant `%not in%` : non appartenance à une liste.
- Que donne ?

```
"width" %not in% names(options())
```

Pour rire !

Redéfinir des éléments clés du langage

L'existence d'une fonction donne le droit de la redéfinir... à ses risques et périls. Les fonctions utilisant la fonction redéfinie ne seront peut être pas préparées à subir les derniers outrages !

```
> `( ` <- function (e) 2*e # La fonction appelée quand on met quelque chose (e)
> 1+(1+1)                  # entre une parenthèse ouvrante et une fermante
[1] 5

> rm("(")                  # On supprime la définition précédente
> 1+(1+1)                  # pour retrouver un fonctionnement normal
[1] 3
```

Néanmoins ce genre de technique (avec « un peu » de précautions pour ne rien casser) peut se justifier le cas d'un objet de type particulier où on veut une syntaxe qui sorte de l'ordinaire.

RAPPEL : les packages

- Les packages sont un composant essentiel de l'éco-système R. Même les fonctions statistiques ou utilitaires de base sont dans des packages chargés automatiquement avec le noyau : 'stats' et 'util'.
- **Pour utiliser les fonctionnalités développées par des milliers de contributeurs dans le monde, on aura nécessairement besoin de plusieurs packages complémentaires plus ou moins recommandés par les formations, les collègues, etc. La démarche est en deux temps :**
 - 1) Installer dans le répertoire de packages du **disque** de l'ordinateur
 - En provenance d'internet (site CRAN), la fonction : `install.packages`, ou le bouton 'install' de l'ongle 'package' de **RStudio**.
 - En provenance d'un répertoire de l'ordinateur, c'est à dire dans un contexte de développement de package , il y a plusieurs procédures possibles, dont une est en démonstration en annexe du support.
 - 2) Utiliser le package c'est à dire installer le package en **mémoire**
 - La fonction `library` ou cocher la case dans la liste des packages connus (installés).
- Remarque : Les deux opérations précédentes peuvent être annulées (ce n'est utile qu'en développement) : respectivement `detach` et `remove.packages`.

L'accès direct à une fonction d'un package

Le chargement d'un package avec `library` n'est pas une fatalité. Si le package voulu est installé, on peut invoquer ses fonctions avec l'opérateur `::`

```
> rio::import("IGoR/data/poplegale_6815.sas7bdat") %>%
  dplyr::group_by(D) %>%
  dplyr::summarise(sum(PMUN15, na.rm=TRUE)) %>%
  base::head(1)
# A tibble: 1 x 2
  D      `sum(PMUN15, na.rm = TRUE)`
<chr>      <dbl>
1 01      631877
```

Les fonctions du noyau sont dans le pseudo package 'base'.

- ATTENTION : ne pas conclure trop vite autour de l'usage du terme « chargement » pour `library`. En terme d'occupation mémoire l'économie proposée par les `::` est minime : juste le dictionnaire des fonctions du package.
- L'usage de `::` est une pratique courante et conseillée à l'intérieur de code destiné à être partagé.
 - Cela rend explicite la référence aux packages utilisées par le code
 - Cela évite qu'une fonction de même nom mais dans un autre package soit utilisée à tort.

Représentation interne (1/2)

- La représentation interne d'une fonction est une « expression » : une arborescence de symboles.

```
> p <- parse(text="function(x,y) x + y", keep.source=TRUE)
> getParseData(p)
```

	line1	col1	line2	col2	id	parent	token	terminal	text
17	1	1	1	19	17	0	expr	FALSE	
1	1	1	1	8	1	17	FUNCTION	TRUE	function
2	1	9	1	9	2	17	'('	TRUE	(
3	1	10	1	10	3	17	SYMBOL_FORMALS	TRUE	x
4	1	11	1	11	4	17	','	TRUE	,
6	1	12	1	12	6	17	SYMBOL_FORMALS	TRUE	y
7	1	13	1	13	7	17	')'	TRUE)
15	1	15	1	19	15	17	expr	FALSE	
9	1	15	1	15	9	11	SYMBOL	TRUE	x
11	1	15	1	15	11	15	expr	FALSE	
10	1	17	1	17	10	15	'+'	TRUE	+
12	1	19	1	19	12	14	SYMBOL	TRUE	y
14	1	19	1	19	14	15	expr	FALSE	

Représentation interne (2/2)

- Néanmoins depuis la R3.4, un compilateur à la volée transforme automatiquement un code souvent utilisé en instructions d'une machine virtuelle, du « **bytecode** ». C'est ce code, plus efficace, qui sera désormais utilisé.

```
> f <- function(x,y) x + y
> f
function(x,y) x + y
> f(1,2)
[1] 3
> f
function(x,y) x + y
> f(2,3)
[1] 5
> f
function(x,y) x + y
<bytecode: 0x0a72c378>
```

Il y a fonction et fonction !

- En pratique il y a trois types de fonctions, dont deux premiers types de fonctions complètement internes (pas de code R) :
 - des **fonctions qui évaluent tous leurs arguments** a priori, avant le calcul du résultat : les `builtin`,
 - des **fonctions qui évaluent leurs arguments en fonction de leur besoin**, donc pas a priori, les `special`,
 - des **fonctions définies en R** avec `function`, y compris dans le noyau : les `closure`. L'évaluation des arguments n'est pas systématique.
- La fonction `typeof` retourne la représentation interne, contrairement à `class` qui peut être modifié par l'utilisateur .

```
> typeof(`+`)
[1] "builtin"

> typeof(`{`)
[1] "special"

> typeof(quote)
[1] "special"

> typeof(library)
[1] "closure"

> f <- function (x) x+1
> typeof(f)
[1] "closure"
```

Important

1

Utile

3

Avancé

1

Utile

1

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l)!=length(n)) stop("Wrong number of arguments.")  
    names(l)<- if (is.null(names(l))) as.character(n)  
               else ifelse(names(l)== "",as.character(n),names(l))  
    e <- do.call(substitute,list(c$expr,l))  
    eval(e,envir=-2)  
  }  
}
```

Séquence F2

Les paramètres des fonctions

Promesses
missing
l'ellipse
do.call

Paramètres non renseignés : les valeurs par défaut

- Lors de la définition d'une fonction, on peut préciser des valeurs par défaut :

```
> f <- function(x=1, y=2) x+y
> f()
[1] 3
> f(y=1)
[1] 2
```

- Grace à l'évaluation retardée (détaillée dans une séquence ultérieure), **les valeurs par défaut peuvent être des expressions et même faire référence à d'autres paramètres**. Le calcul ne sera fait qu'en cas de besoin, dans l'environnement de la fonction. Ces expressions sont du type « promesse » ('promise'), une structure spécifique.

```
> f <- function(x=y/2, y=2*x) x+y
> f(1)      # x est fourni, y sera calculé à partir de x
[1] 3
> f(y=1)    # y est fourni, x sera calculé à partir de y
[1] 1.5
> f()
```

Error in f() :

la promesse est déjà en cours d'évaluation : référence récursive d'argument par défaut ou problème antérieur ?

Paramètres non renseignés : la fonction `missing`

- En l'absence de valeur par défaut, l'appel d'une fonction sans alimenter un paramètre est encore possible car en raison de l'évaluation retardée c'est à la fonction de décider de ce qu'elle fait de ses paramètres.

Par défaut :

```
> f <- function(x=1, y) x+y
> f(1)
Error in f(1) :
  l'argument "y" est manquant, avec aucune valeur par défaut
```

- La fonction `missing`, appelée dans le corps de la fonction, permet d'intercepter les cas de paramètres non renseignés.**

```
> f <- function(x, y) if (missing(x) | missing(y)) 42 else x+y
> f(2)
[1] 42
> f(2,3)
[1] 5
```

Exercice rapide

Paramètres avec valeur par défaut

- Transformez en fonction de deux paramètres le code suivant, pour qu'on puisse l'utiliser avec n'importe quel département (ou liste de départements) et une autre variable que *'indnatm'*.

Faites en sorte que par défaut, la fonction travaille sur le département 974 .

```
fst("IGoR/data/nais2017.fst") %>%  
  [. $depnais %in% c("17", "86"), c("depnais", "indnatm")] %>%  
  group_by_at(c("depnais", "indnatm")) %>%  
  summarise(n=n())
```

Des paramètres sans nom : l'ellipse '...'

- Le symbole '...' spécifié dans les paramètres d'une fonction joue un rôle particulier : il collecte le reste des arguments après que les autres paramètres aient été alimentés par les arguments de l'appel de la fonction.

```
> f <- function (x,...) str(get("..."))
> f(1,TRUE,z=2)
length 2, mode "...": TRUE 2
```

Ni une liste, ni un vecteur !

- Il s'agit d'un objet d'un nouveau type et structure très spécifiques, mais **cet objet peut être transféré à une autre fonction** qui pourra, par exemple, le mettre sous une forme exploitable : `list` ou `c`, pour ensuite permettre de parcourir l'ensemble des valeurs.

```
> f <- function (x,...) list(...)
> f(1,TRUE,z=2)
[[1]]
[1] TRUE

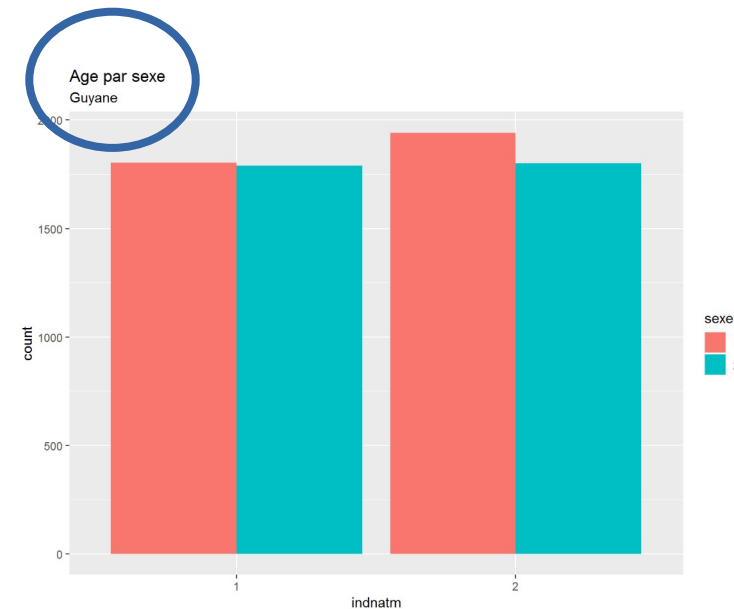
$z
[1] 2
```

```
list(TRUE, z=2)
```

Une utilisation classique de l'ellipse

L'ellipse peut aussi servir à transporter des paramètres destinés à une autre fonction sans préciser quels sont ces paramètres. C'est le mécanisme utilisé par les fonctions `'import'` et `'export'` de `rio` pour, sans programmation complémentaire, autoriser la spécification d'options propres aux fonctions des divers packages qui sont appelés suivant le suffixe du fichier.

Dans l'exemple ci dessous la fonction 'realiser_graphique' est définie avec un unique paramètre fournissant le département de travail mais on peut lui passer n'importe quoi d'autre qui sera alors transmis à la fonction 'gf_bar'.



```
> realiser_graphique <- function(.dep = "13", ...)
  Naissances %>%
  filter(depnaiss==.dep) %>%
  gf_bar( ~ indnatm, fill=~ sexe, position= "dodge", ...)

> realiser_graphique(.dep = "973", title = "Age par sexe", subtitle = "Guyane")
```


Appeler une fonction avec ses paramètres dans une liste : la fonction `do.call`

En R il est facile de construire dynamiquement un appel de fonction :

- on peut paramétrer le nom de la fonction à appeler :

```
> f <- `*`  
> f(6,7)  
[1] 42
```

- Il est trivial de paramétrer chacun des arguments,

```
> p <- 6 ; q <- 7  
> f(p,q)  
[1] 42
```

- la fonction `do.call` permet de paramétrer l'ensemble des arguments qu'on doit fournir en les spécifiant sous forme d'une unique liste :

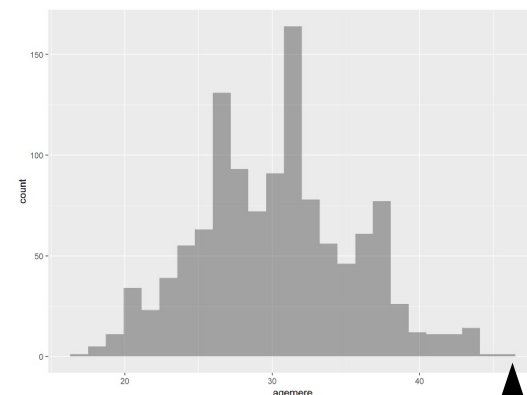
```
> l <- list(6,7)      # 'c' construirait un vecteur non une liste  
> do.call(f,l)  
[1] 42
```

Cette fonction est souvent utile dans des contextes où les valeurs des arguments proviennent de fonctions de type 'Map' et où la fonction travaille uniformément sur ses arguments :

```
> do.call(sum,Map(function(x) x^2, 1:10))  
[1] 385
```

La reconnaissance des noms des paramètres : la fonction `match.arg`

- Lorsqu'on appelle une fonction en donnant le nom des paramètres il n'est pas nécessaire de les donner en entier : toute abréviation non ambiguë fait l'affaire.
- La fonction qui est derrière ce mécanisme est `match.arg` qui permet de convertir une abréviation en un nom complet appartenant à une liste prédéfinie.



```
> realiser_graphique <- function(table, dep, type_graphique, ...) {
  type_graphique <- match.arg(type_graphique,
                               choices = c("histogramme", "densite"))

  table %>%
    filter(depnaiss %in% dep) %>%
    mutate(agemere=as.numeric(agemere)) %>%
    (if (type_graphique == "histogramme") gf_histogram else gf_density)
    (~ agemere, ...)
}
```

La fonction à appliquer est calculée !

```
> realiser_graphique(naissances, type = "truc", dep = "04")
Error in match.arg(type_graphique, choices = c("histogramme", "densite")) :
  'arg' should be one of "histogramme", "densite"

> realiser_graphique(naissances, type = "hist", dep = "04")
```

Fin de la séquence
XML acte 1 – scène 3
« On ne copie pas ! »

⇒ **Séquence F1 « Les fonctions »**

- Rassembler toutes les étapes, sauf la première et la dernière, dans une seule fonction sans argument. Tester le graphique.

=> **Séquence F2 « Les paramètres des fonctions »**

- Paramétrer cette fonction pour qu'elle puisse fonctionner sur les lignes « pdv » mais aussi « prix » ou d'autres (attention le marqueur de fin est différent, ' />' avec un espace avant, mais ce sont les lignes point de vente qui constituent une exception).

Rappel : Un peu de statistiques

- Rajouter une colonne 'date' (de type date) à partir du début de la colonne 'maj'. Mettre en numérique le contenu de 'valeur' (le prix en millièmes d'euros), puis calculer le prix moyen du SP98 par jour (fonctions `group_by` et `summarise` de **dplyr**). Avec la fonction `gf_line` tracer l'évolution du prix moyen.
- Faire la même chose en superposant les courbes obtenues pour chaque type de carburant.

Corrigé

```
lire <- function(tag)
  z[str_detect(z,paste0('<',tag,' '))] %>%
  str_replace_all(paste0(" +<",tag,' '), "") %>%
  str_replace_all(paste0(' */?>'), "") %>%
  str_replace_all("\\", "") %>%
  data.frame(
    d=.,
    no=seq_along(.),
    stringsAsFactors=FALSE) %>%
  mutate(l=str_split(d, " +")) %>%
  unnest(l) %>%
  separate(l, c("clé", "donnée"), "=") %>%
  spread(clé, donnée)

lire("prix") %>%
  filter(nom=="SP98") %>%
  mutate(date=as.Date(substr(maj,1,10)),
    prix=as.numeric(valeur)/1000) %>%
  group_by(date) %>%
  summarise(prix=mean(prix)) %>%
  gf_line(prix ~ date)
```

R version $\geq 4.1.0$

Une implémentation rigoureuse du pipe dans le noyau de R.

```
lire <- \(tag)
  z[str_detect(z,paste0('<',tag,' '))] |>
  str_replace_all(paste0(" +<",tag,' '), "") |>
  str_replace_all(paste0(' */?>'), "") |>
  str_replace_all("\\", "") |>
  (\(x) data.frame(
    d=x,
    no=seq_along(x),
    stringsAsFactors=FALSE))() |>
  mutate(l=str_split(d, " +")) |>
  unnest(l) |>
  separate(l, c("clé", "donnée"), "=") |>
  spread(clé, donnée)

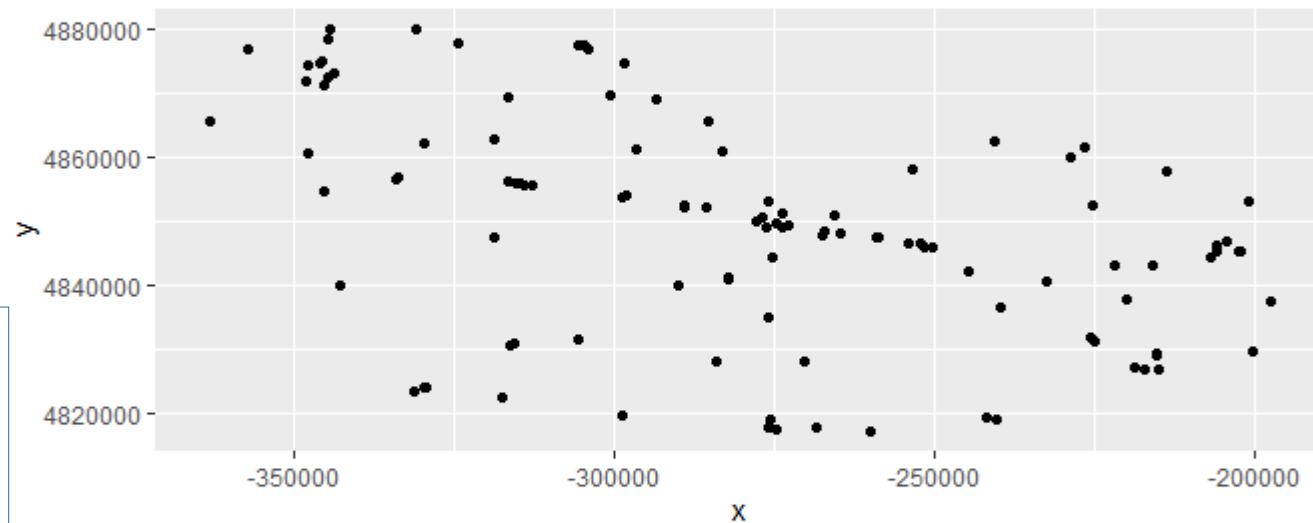
lire("prix") |>
  filter(nom=="SP98") |>
  mutate(date=as.Date(substr(maj,1,10)),
    prix=as.numeric(valeur)/1000) |>
  group_by(date) |>
  summarise(prix=mean(prix)) |>
  gf_line(prix ~ date)
```


Acte I - scène 4

Les pompes ouvertes le dimanche

Jours d'ouverture par pompe

```
# A tibble: 1,015 x 4
  no ferme id  nom
  <int> <chr> <chr> <chr>
1     1  1 ""    1  Lundi
2     1  1 ""    2  Mardi
3     1  1 ""    3  Mercredi
4     1  1 ""    4  Jeudi
5     1  1 ""    5  Vendredi
6     1  1 ""    6  Samedi
7     1  1 "1"    7  Dimanche
8    336 "1"    1  Lundi
9    336 "1"    2  Mardi
10   336 "1"    3  Mercredi
# ... with 1,005 more rows
```



Important

5

Utile

5

Rare

1

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l) != length(n)) stop("Wrong number of arguments.")  
    names(l) <- if (is.null(names(l))) as.character(n)  
                  else ifelse(names(l) == "", as.character(n), names(l))  
    e <- do.call(substitute, list(c$expr, l))  
    eval(e, envir=-2)  
  }  
}
```

Séquence R3

Les tables de données

data.frame

Un data frame est une liste

Pour les besoins du statisticien, R introduit le `data.frame`, objet rectangulaire constitué d'une **liste de vecteurs** :

Un vecteur pour chacune des variables. Chaque variable mesurant la même chose a donc un type unique.

Chaque vecteur a un nom et peut porter des attributs supplémentaires (des "labels" par exemple).

Autant d'éléments dans chaque vecteur que d'observations. Tous les vecteurs ont donc la même hauteur.

Un exemple : une petite table de 4 observations (lignes) et 4 variables (colonnes = vecteurs)

CODGEO	SUPERF	POP	URBAIN
"16015"	22	42081	TRUE
"17300"	28	75404	TRUE
"79191"	68	58952	TRUE
"86091"	17	423	FALSE

NOTE : En principe, les données présentes dans les colonnes sont issues des types de base et les colonnes sont donc des vecteurs. Mais ce principe peut être violé avec des données de complexité quelconque

Les colonnes ont des noms, les lignes aussi

- La fonction `colnames` permet de construire un vecteur à partir des noms des colonnes.

```
> mtcars %>% head(1)
      mpg cyl disp  hp drat   wt  qsec vs am gear carb
Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4

> mtcars %>% head(1) %>% colnames()
[1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

- Les lignes aussi peuvent avoir des noms. Ceux ci sont initialisés par défaut au numéro de la ligne. La fonction `rownames` permet d'extraire le vecteur des noms.

```
> mtcars %>% head(1) %>% rownames()
[1] "Mazda RX4"
```

Accéder à une partie d'un data frame : sélectionner des lignes

La sélection d'une partie d'un data frame peut se faire comme s'il s'agissait d'une **matrice** : on fournit entre crochets deux arguments. Le premier sert à sélectionner les lignes selon les possibilités suivantes :

- **Rien** : toutes les lignes
- **Un vecteur numérique** : uniquement les lignes de numéro cité (déconseillé)
- **Un vecteur de nombres négatifs** : toutes les lignes sauf celles de numéro cité (déconseillé)
- **Un vecteur de chaînes de caractères** : uniquement les lignes de nom cité
- **Un vecteur de booléens**, exprimant généralement une condition sur les colonnes. Les colonnes doivent alors être citées par leur nom complet en indiquant la table d'origine, même si cela semble redondant : n'importe quel objet connu de R pourrait intervenir dans la condition.

```
> mtcars[mtcars$hp>100 & mtcars$cyl==4,]
      mpg  cyl  disp  hp drat   wt  qsec vs  am  gear  carb
Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.9  1   1     5     2
Volvo 142E   21.4   4 121.0 109 4.11 2.780 18.6  1   1     4     2
```

Tandis que :

```
> mtcars[hp>100 & cyl==4,]
Error in `[.data.frame]'(mtcars, hp > 100 & cyl == 4, ) :
  objet 'hp' introuvable
```

Des crochets, une fonction, mais des crochets adaptés aux data frames (cf. principe numéro 6 : toute fonction peut être surchargée)

Accéder à une partie d'un data frame : sélectionner des colonnes

Les colonnes sont sélectionnées par le deuxième argument présent entre crochets, sur le même principe que pour les lignes :

- **Rien** : toutes les colonnes
- **Un vecteur numérique** : uniquement les colonnes de numéro cité (déconseillé)
- **Un vecteur de nombres négatifs** : toutes les colonnes sauf celles de numéro cité (déconseillé)
- **Un vecteur de chaînes de caractères** : uniquement les colonnes de nom cité
- **Un vecteur de booléens**, exprimant généralement une condition sur les noms (la fonction 'colnames' fournit le vecteur des noms de colonnes)

```
> mtcars[1,c("hp","cyl")]
      hp cyl
Mazda RX4 110   6
```

```
> mtcars[1,startsWith(colnames(mtcars),"c")]
      cyl carb
Mazda RX4   6   4
```



ATTENTION : Sauf si on rajoute le paramètre `drop=FALSE` à l'appel de la fonction `[`, lorsqu'on sélectionne une seule colonne, le résultat est, par défaut, "simplifié". Il n'est plus un data frame mais un vecteur:

```
> mtcars[, "cyl"]      #mtcars[, "cyl", drop=FALSE] fait autre chose
[1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

Accéder à une partie d'un data frame : sélectionner une unique colonne

Un data frame étant une liste, l'opérateur `$` ou l'opérateur `[[` permettent de restituer un champ de la liste, donc un vecteur.

```
> mtcars$hp
```

```
[1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52  65  97 150 150 245
[25] 175  66  91 113 264 175 335 109
```

```
> mtcars[["hp"]]
```

```
[1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52  65  97 150 150 245
[25] 175  66  91 113 264 175 335 109
```

```
> mtcars[[4]]
```

```
[1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52  65  97 150 150 245
[25] 175  66  91 113 264 175 335 109
```

La fonction `pull` de *dplyr* est une alternative à ces deux opérateurs. Son principal avantage est qu'elle s'intègre naturellement dans une syntaxe à base de pipes :

```
> mtcars %>% pull(hp) # ou pull("hp") ou pull(4)
```

```
[1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52  65  97 150 150 245
[25] 175  66  91 113 264 175 335 109
```

Exercice rapide

Filtrer un data frame sans *dplyr*

1) Sans utiliser *dplyr* (donc avec les crochets), à partir du fichier de naissances ("IGoR/data/nais2017.fst"), récupérer le vecteur des ages *agemere* de la mère, pour les naissances :

- de filles (sexe est à 2)
- des départements 16 et 18 (*depnais*)
- hors mariage (*amar* est à '0000'),
- où la mère a plus de 40 ans.

2) Ne produire que les valeurs distinctes (utiliser deux méthodes différentes : avec *dplyr* ou sans, grâce à un peu de recherche).

Corrigé de l'exercice

```
> library(rio)
> naissances <- import("IgoR/data/nais2017.fst")
> with(naissances,
      naissances[(sexe=='2') & (depnais %in% c("16", "18")) & (amar=='0000')
                  & (agemere>40), "agemere"]
    )
[1] "42" "44" "43" "46" "42" "41" "42" "42" "43" "41" "45" "45" "41" "45" "43"
[16] "41" "41" "41" "43" "41" "45" "45" "42" "43" "43" "41" "41" "43" "42" "43"
[31] "41"
```

> naissances %>%
 with(.[(sexe=='2') & (depnais %in% c("16", "18")) & (amar=='0000')
 & (agemere>40), "agemere", drop=FALSE] %>%
 distinct(agemere) %>%
 pull(agemere)
)

▲
Une seule colonne mais on ne veut pas de la réduction automatique à un vecteur

```
[1] "42" "44" "43" "46" "41" "45"
```

> naissances %>%
 with(.[(sexe=='2') & (depnais %in% c("16", "18")) & (amar=='0000')
 & (agemere>40), "agemere"] %>%
 unique() # Voir documentation de 'distinct'
)

```
[1] "42" "44" "43" "46" "41" "45"
```

La fonction 'with'

- La fonction `with` permet de raccourcir l'écriture des conditions en omettant le nom de la table d'origine. Les noms cités sont cherchés en priorité parmi les colonnes de la table fournie en premier argument de `with` et ne sont pas perturbés par une éventuelle variable portant le même nom.

```
> cyl <- "n'importe quoi"
> with(mtcars,mtcars[(cyl==6) & (hp>150) ,])
      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
Ferrari Dino 19.7   6  145 175 3.62 2.77 15.5  0  1    5    6
```

- Avec un pipe qui limite les redondances et rapproche de la simplicité de *dplyr* :

```
> mtcars %>% with(.[(cyl==6) & (hp>150) ,])
      mpg cyl  disp  hp drat   wt  qsec vs am gear carb
Ferrari Dino 19.7   6  145 175 3.62 2.77 15.5  0  1    5    6
```

Modifier un data frame

- La logique pour modifier les cellules existantes d'un data-frame est la même que pour les vecteurs : on met la sélection de cellules à modifier du côté gauche de l'assignation.
- Deux solutions existent selon qu'on considère un data frame comme :
 - une liste de vecteurs
 - ou comme une matrice.

La seconde possibilité permet en outre de modifier simultanément plusieurs colonnes.

- Ajouter une colonne est également possible : la cohérence est assurée.
- Pour ajouter des lignes il faut faire appel à la fonction de concaténation de data frames : `rbind`.

```
> df <- read.table(h=TRUE, text=
"x      y  z
un      1 NA
deux    NA 2")
```

```
> df$z[df$x=="un"] <- 1
> df
```

	x	y	z
1	un	1	1
2	deux	NA	2

```
> df[df$x=="un", "z"] <- 2
> df
```

	x	y	z
1	un	1	2
2	deux	NA	2

```
> df[df$x=="un", c("y", "z")] <- 0
> df
```

	x	y	z
1	un	0	0
2	deux	NA	2

```
> df$new <- "a"
> df
```

	x	y	z	new
1	un	0	0	a
2	deux	NA	2	a

Les avatars des data frames : les 'tibble'

- Par défaut, l'affichage d'un gros « data frame » n'est guère commode : il y a bien une limite en termes de lignes affichées, mais celle ci (certes paramétrable) est élevée. De plus, lorsqu'on a de nombreuses variables, toutes sont affichées conduisant à une présentation sur plusieurs lignes.
- Le monde du "tidyverse" autour du package *dplyr* introduit la notion de **tibble**, un « data frame » qui s'affiche "proprement" : uniquement 10 lignes (c'est paramétrable) et uniquement les colonnes qui peuvent tenir dans la fenêtre. Le reste est mis en commentaire. Enfin, taille du « data frame » et type des variables sont affichés.

```
> mtcars %>% as_tibble() -> a
```

```
> a
```

```
# A tibble: 32 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	22.8	4	108	93	3.85	2.32	18.6	1	1	4	1
4	21.4	6	258	110	3.08	3.22	19.4	1	0	3	1
5	18.7	8	360	175	3.15	3.44	17.0	0	0	3	2
6	18.1	6	225	105	2.76	3.46	20.2	1	0	3	1
7	14.3	8	360	245	3.21	3.57	15.8	0	0	3	4
8	24.4	4	147.	62	3.69	3.19	20	1	0	4	2
9	22.8	4	141.	95	3.92	3.15	22.9	1	0	4	2
10	19.2	6	168.	123	3.92	3.44	18.3	1	0	4	4

```
# ... with 22 more rows
```

ATTENTION : le « tidyverse » n'aime pas les noms d'observations, ils sont perdus lors de la transformation.

- REMARQUE : Le mode "tibble" est en fait un exemple d'un nouveau type de données créé comme spécialisation d'un autre plus général (*data.frame*).
- La fonction '*as.data.frame*' permet de faire la conversion inverse.

Les avatars des data frames : accéder à une partie d'un fichier **fst**

- La fonction `fst` du package **fst** permet d'établir une connexion avec un fichier créé par ce package et de **travailler dessus comme s'il s'agissait d'un « data frame » en mémoire, mais sans le charger, avec l'essentiel des mécanismes de sélection de lignes ou de colonnes décrits précédemment.**

```
> library(fst)
> f1 <- fst("IGoR/data/nais2017.fst")
> str(f1)
List of 4
 $ meta      :List of 7
  ..$ path      : chr "D:\\h2izgk\\PALETTES\\IGoR\\data\\nais2017.fst"
  ..$ nrOfRows   : num 792868
  ..$ keys       : NULL
  ..$ columnNames : chr [1:10] "comnais" "depnais" "regnais" "sexe" ...
  ..$ columnBaseTypes: int [1:10] 2 2 2 2 5 2 2 2 2 2
  ..$ keyColIndex  : NULL
  ..$ columnTypes  : int [1:10] 2 2 2 2 10 2 2 2 2 2
  ..- attr(*, "class")= chr "fstmetadata"
 $ col_selection: NULL
 $ row_selection: NULL
 $ old_format   : logi FALSE
```

'f1' n'est pas un data frame

```
> f1[1,c("depnais", "agemere")]
  depnais agemere
1      01      34
```

'f1' peut être utilisé comme un data frame

- ATTENTION : Le raccourci avec `with` ne marche pas, pas plus que les fonctions de **dplyr**. Un objet issu de **fst** n'est pas un « data frame ».

length et ncol, nrow colnames et names

Un « data frame » est une liste, certes, mais des packages comme **fst** ou **dbplyr** (vers des bases de données) mettent à disposition des objets qui se comportent comme des « data frame » sans en être. Il est donc judicieux de ne pas trop faire confiance à l'équivalence avec une liste et d'utiliser les fonctions explicitement construites pour récupérer les métadonnées.

- Pour obtenir le nombre de colonnes, `length` ne donne pas toujours le bon résultat. **C'est `ncol` qui donne le nombre de colonnes d'une table :**

```
> length(f1)
[1] 4
> ncol(f1)
[1] 32
```

- Pour obtenir le nombre des colonnes, `names` donne le noms des éléments d'une liste ou d'un vecteur mais pourrait ne pas avoir de sens pour un data.frame. C'est `colnames` qu'il faut utiliser.
- De même, regarder la longueur d'un vecteur de la liste constituant le data frame pourrait ne pas avoir de sens. **C'est `nrow` qui donne la taille d'une table :**

```
> nrow(f1)
[1] 769553
```

Des data frames hors norme : I

- Le schéma usuel où au croisement d'une ligne et d'une colonne on trouve une donnée élémentaire n'est pas une fatalité.

Les fonctions `nest` de *tidyr*, `str_split` de *stringr* sont des exemples de fonctions qui peuvent amener à des structures plus complexes.

- Invquée à l'intérieur de la fonction `data.frame` la fonction `I` (pour « as is ») est le moyen élémentaire de créer une table où une des colonnes n'est pas un vecteur.

```
> (df <- data.frame(no=1:2,
+   x=I(list(list("a","b"),
+             list("c","d","e")))))
```

```
  no      x
1  1    a, b
2  2  c, d, e
```

Attention : la présentation peut être trompeuse : les virgules n'appartiennent pas aux données !

```
> str(df)
'data.frame':  2 obs. of  2 variables:
 $ no: int  1 2
 $ x :List of 2
 ..$ :List of 2
 .. ..$ : chr "a"
 .. ..$ : chr "b"
 ..$ :List of 3
 .. ..$ : chr "c"
 .. ..$ : chr "d"
 .. ..$ : chr "e"
 ..- attr(*, "class")= chr "AsIs"
```

Des data.frames hors normes

Démythifier le data frame

- Un « data frame » n'est rien d'autre qu'une liste de colonnes (des vecteurs ou des listes), agrémentée de propriétés supplémentaires :
 - Des noms aux colonnes
 - Des noms aux lignes
 - Une classe "data.frame"
- Il est donc tout à fait possible d'en créer de façon complètement artisanale.

```
> m <- matrix(1:6,nrow=2,ncol=3)
> m
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> df <- list(x = c("un","deux"), y=m)
> class(df) <- "data.frame"
> row.names(df) <- 1:2

> df
      x y.1 y.2 y.3
1  un    1    3    5
2 deux    2    4    6

> str(df)
'data.frame':  2 obs. of  2 variables:
 $ x: chr  "un" "deux"
 $ y: int [1:2, 1:3] 1 2 3 4 5 6

> df[1,]
      x y.1 y.2 y.3
1 1    1    3    5
```

Fin de la séquence

Acte I - scène 4

Une architecture complète

⇒ **Séquence R3 « data frames »**

- 1) En revenant au résultat brut du `readLines`, deux premières lignes exclues, construire un data frame de deux colonnes : le numéro de la ligne et le contenu de la ligne.
- 2) Sans utiliser **dplyr**, mettre à valeur manquante le numéro de toutes les lignes autres que celles d'un point de vente.
- 3) Avec la fonction `na.locf` du package **zoo**, remplacer chaque valeur manquante par la dernière valeur non manquante.
- 4) Modifier la fonction écrite précédemment pour utiliser la table qu'on vient de construire au lieu du vecteur issu de `readLines`. L'appliquer au niveau point de vente, puis au niveau « *jour* ».
- 5) Quelles sont les pompes ouvertes le dimanche ?

Retour sur la deuxième demi-journée

Les fonctions

Définir une fonction : **function**

Juste une substitution des constantes par des noms de paramètres.

Ni accolades, ni 'return' sauf cas compliqués.

Les paramètres des fonctions :

Valeurs par défaut

L'ellipse ..., dans les fonctions des packages.

XML acte II

Un nouveau personnage, mais des listes rien que des listes...

- Avec la fonction `read_xml` du package **xml2**, lire la totalité du contenu du fichier. Afficher la structure obtenue. Puis convertir le résultat en liste à l'aide de la fonction `as_list` du même package.

Le résultat est une très très longue liste similaire à la structure interne du fichier XML. Hormis la première ligne d'en tête, celui ci est en fait constitué d'une unique liste de points de vente, et chaque élément de la liste est une liste décrivant les données du point de vente.

Utiliser le paramétrage `max.level=2` de la fonction `str` pour visualiser le sommet de la structure.

- Si 'xml2' est le résultat de l'opération précédente `xml2[[1]]` (aussi nommé `xml2$pdv_liste`) est la liste des points de vente et `xml2[[1]][[42]]` est le 42ème point de vente et ses données.
- Visualiser `xml2[[1]][[42]]`.

Chacune des données se présente sous forme de liste , par exemple :

`xml2[[1]][[42]]$adresse` qui est une liste d'une seule chaîne de caractères, l'adresse. L'adresse est donc en

`xml2[[1]][[42]]$adresse[[1]]`.

La structure du fichier vue par *xml2*

Premier niveau de liste

Un seul élément : l'en-tête du fichier

```
xml2[[1]]
```

```
xml2$pdv_liste
```

Deuxième niveau de liste

Un élément par point de vente, de numéro <i>
i

```
xml2$pdv_liste[[i]]
```

Troisième niveau de liste

Un élément par donnée

```
xml2$pdv_liste[[i]]$ville
```

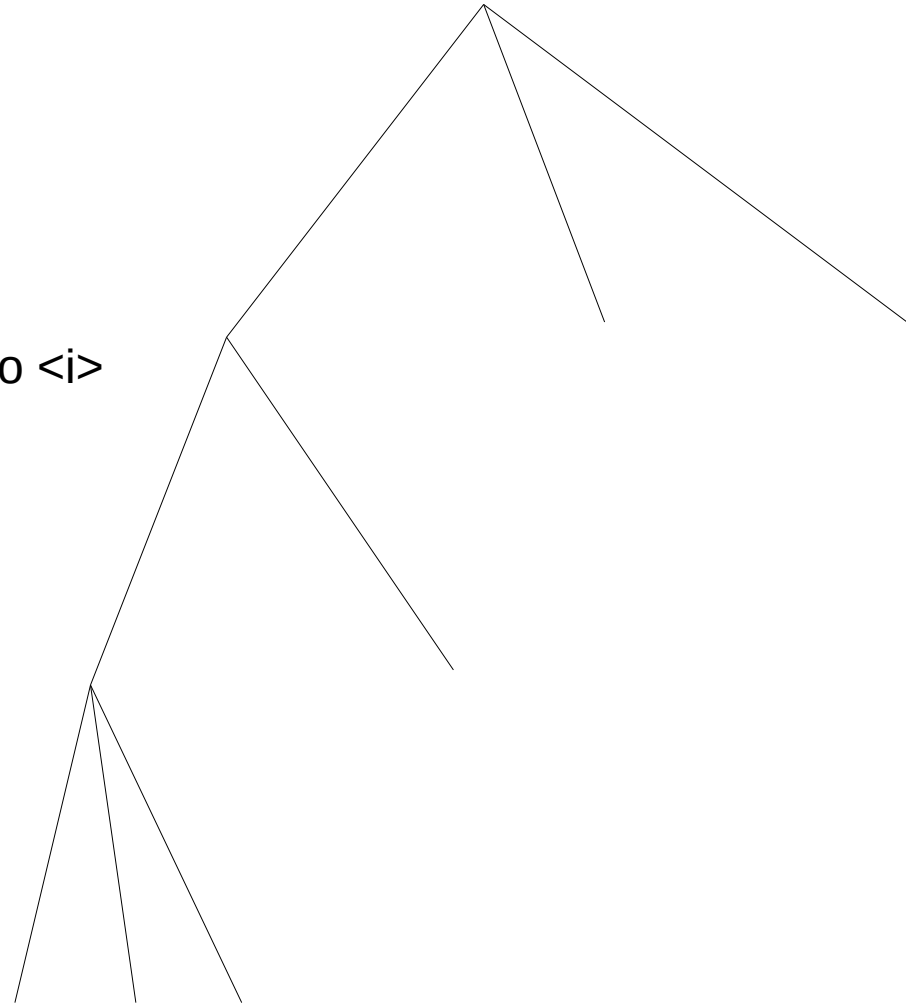
```
xml2$pdv_liste[[i]]$adresse
```

...

Quatrième niveau : les données

Soit des attributs des nœuds précédents

Soit un élément de la liste présente au niveau précédent



XML acte II scène 1

Les adresses des points de vente

```
# A tibble: 145 x 3
  no ville      adresse
  <int> <chr>      <chr>
1     1 Saint-Brieuc "LES VILLAGES"
2     2 SAINT-BRIEUC "3 Rue Champlain"
3     3 SAINT-BRIEUC "Rue Guillaume Apollinaire"
4     4 SAINT-BRIEUC "44 Boulevard Charner"
5     5 Saint-Brieuc "28 Rue Edmond Rostand"
6     6 SAINT-BRIEUC " Rue Guillaume Apollinaire"
7     7 Saint-Brieuc "44, BOULEVARD CHARNER"
8     8 Saint-Brieuc "RUE GUILLAUME APOLLINAIRE"
9     9 QUEVERT     "C CIAL LE CHENE"
10    10 Taden      "Le Pré des Landes"
# ... with 135 more rows
```

Important

4

Utile

5

Rare

2

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l) != length(n)) stop("Wrong number of arguments.")  
    names(l) <- if (is.null(names(l))) as.character(n)  
                  else ifelse(names(l) == "", as.character(n), names(l))  
    e <- do.call(substitute, list(c$expr, l))  
    eval(e, envir = -2)  
  }  
}
```

Séquence B

Faire des traitements itératifs

Comment répéter un appel de fonction un nombre fixe de fois

Itérer un traitement un nombre fixe de fois (1/3)

R est un langage fonctionnel.

Le principe le plus naturel en R pour répéter un traitement consiste à :

- définir le traitement sous forme d'**une fonction** dont les paramètres seront ce qui va varier entre les différents traitements,
- **appliquer cette fonction** aux différents cas, en spécifiant ceux ci sous forme de **liste**.

Il existe de nombreuses fonctions permettant d'appliquer une fonction à chacun des éléments d'une liste. En utilisant le R de base, pour effectuer la lecture successive des deux départements de Charente (16) et Charente-Maritime (17), on pourra faire :

```
> ldf <- Map(lire1, c("16","17"))
```

Le résultat est une **liste de résultats**, un pour chacune des lectures :

```
> str(ldf)
List of 2
 $ 16:'data.frame':  3027 obs. of  2 variables:
  ..$ depnais: chr [1:3027] "16" "16" "16" "16" ...
  ..$ agemere: chr [1:3027] "34" "34" "31" "21" ...
 $ 17:'data.frame':  4826 obs. of  2 variables:
  ..$ depnais: chr [1:4826] "17" "17" "17" "17" ...
  ..$ agemere: chr [1:4826] "29" "36" "27" "37" ...
```

Itérer un traitement un nombre fixe de fois (2/3)

- Les arguments de la fonction `Map` :
 - Une fonction (les fonctions sont des objets comme les autres on peut donc les passer en paramètre):
 - Soit par son nom (donc un symbole)

```
> Map(import, c("nais2017.dbf", "nais2017.fst"))
```

- Soit par sa seule définition, une fonction sans nom (cela arrive très souvent!)

```
> Map(function(x) paste0("nais2017.",x) %>% import(), c("dbf","fst"))
```

- Une « liste » de valeurs :
 - Soit une liste, quelle qu'elle soit (p.e. les colonnes d'un data.frame)
 - Soit un vecteur
- Le résultat de la fonction `Map`:
 - Est **systématiquement une liste** (pas un vecteur), voir `unlist` si besoin,
 - **dont les composants peuvent avoir un nom** (si la liste de valeurs est un vecteur de chaînes de caractères)

cf. l'exemple précédent.

Itérer un traitement un nombre fixe de fois (3/3)

Le résultat d'un appel de `Map` étant une liste, il n'est pas directement utilisable par les fonctions de `dplyr`. Par contre on peut continuer à travailler dans une logique d'itération et de résultats sous forme de liste :

```
> Map(count, ldf)
$`16`
# A tibble: 1 x 1
      n
  <int>
1  3027

$`17`
# A tibble: 1 x 1
      n
  <int>
1  4826
```

Et si on désire réellement revenir à une logique de table unique, le R de base offre la fonction `Reduce` qui combine successivement tous les éléments d'une liste pour obtenir un élément unique. :

```
> df <- Reduce(bind_rows, ldf)
> str(df)
'data.frame':   7853 obs. of  2 variables:
 $ depnais: chr  "16" "16" "16" "16" ...
 $ agemere: chr  "34" "34" "31" "21" ...
```

Rappel :
`bind_rows` = concaténation

Remarque : 'map' et 'reduce' forment une logique réutilisée en « big data », où 'map' distribue le travail à faire sur les différentes machines et 'reduce' collecte l'ensemble des résultats.

Exercice rapide

Une première boucle avec `Map`

- 1-** Charger le fichier
« *IGoR/data/poplegale_6815.sas7bdat* » et en afficher la structure.
- 2-** Avec les fonctions `Map` et `class`, récupérer une liste d'autant d'éléments que de colonnes et contenant la classe de chaque colonne.
- 3-** Enigme : comment fabriquer un `data.frame` à partir du résultat précédent (deux colonnes « nom », « classe »)?

La fonction `Filter`

- Alors que le résultat de la fonction `Map` a autant d'éléments que la liste qui lui est fournie,

la fonction `Filter` permet de boucler sur une liste en ne retenant que les éléments vérifiant une condition.

- `Filter` transforme son résultat en vecteur si c'est possible.

```
> p <- import("poplegale_6815.sas7bat")
> Map(function(x)
      if (is.numeric(p[[x]])) x,
      colnames(p))

$DC
NULL

$NCC
NULL

$PMUN15
[1] "PMUN15"
# (affichage partiel)
# ... Une liste avec des NULL

> Filter(function(x)
      is.numeric(p[[x]]),
      colnames(p))
[1] "PMUN15" "PMUN10" "PMUN06" "PSDC99"
[5] "PSDC90" "PSDC82" "PSDC75" "PSDC68"

# Une solution sans Filter
# names(p)[unlist(Map(is.numeric,p))]
```


mapply, lapply, sapply

- La fonction `Map` n'est qu'une des interfaces à une fonction plus complète `mapply` qui permet de traiter le cas de **traitements devant être paramétrés par plus d'un paramètre**.
- Il existe aussi les fonctions suivantes, qui ne travaillent que sur une seule liste et donc une fonction à un seul paramètre.
 - `lapply`: comme `Map` à l'ordre des arguments près, mais avec une **possibilité de passer des arguments complémentaires à la fonction**
 - `sapply`: comme la précédente, le « s » signifiant « simplifier » c'est à dire **transformer en vecteur si c'est possible**.

```
> mapply(`+`,1:2,3:4)
[1] 4 6
```

```
> mapply(`+`,1:2,3:4, SIMPLIFY=FALSE)
[[1]]
[1] 4
```

```
[[2]]
[1] 6
```

```
> a <- list(c(1,2,NA),c(3,4))
> lapply(a,mean,na.rm=TRUE)
[[1]]
[1] 1.5
```

```
[[2]]
[1] 3.5
```

```
> sapply(a,mean,na.rm=TRUE)
[1] 1.5 3.5
```

Itérer un nombre fixe de fois

le package **purrr** (1/3)

- Le package **purrr** offre une fonction similaire à `Map`, `map` où les arguments sont à fournir dans l'ordre inverse : liste en premier, ceci pour simplifier l'écriture de « pipes ».
- Cette fonction est utilement complétée par des fonctions qui, au lieu de listes, restituent des vecteurs ou mieux. Le nom de la fonction indique le type désiré.

`map_chr` restituera un vecteur de chaînes de caractères (« **character** »)

`map_dbl` restituera un vecteur de nombres (flottants **double** précision)

`map_dfr` restituera un **data frame** après concaténation par `bind_rows`. Chaque étape de la boucle est sensée produire une liste nommée, avec toujours les mêmes noms. Ces noms seront les noms des colonnes de la table résultat.

- Lorsque l'effet souhaité de la boucle n'est pas de restituer un résultat, mais juste d'effectuer une opération à effet de bord, la fonction `walk` peut être utile.

```
walk(  
  list.files("pages/", full.names=TRUE) ,  
  function (x) source(x, echo=FALSE, encoding="UTF-8"))
```

Charge tout le répertoire 'pages/'

Itérer un nombre fixe de fois

le package **purrr** (2/3)

- Le package offre également des fonctions permettant d'**appliquer successivement une fonction** non à une liste mais à **deux listes** (ou vecteurs), voire à **une liste de listes** (ou de vecteurs, donc aussi à un data frame).

`map2` applique une fonction de deux arguments à deux listes

```
> map2(list(1,2), list(3,4), `+`)
[[1]]
[1] 4

[[2]]
[1] 6
```

`pmap` applique une fonction de n arguments à une liste de n listes

```
> pmap(list(list(1,2), list(3,4), list(5,6)), function (x,y,z) x+y+z)
[[1]]
[1] 9

[[2]]
[1] 12
```

- Ces fonctions sont complétées par des fonctions à suffixe comme dans `map_chr`.

Itérer un nombre fixe de fois le package **purrr** (3/3)

- Les éléments d'une liste pouvant être nommés, des fonctions permettent de **boucler sur les éléments tout en récupérant leur nom**. La fonction à fournir sera une fonction de deux arguments, le second recevant le nom de l'élément à traiter.

`imap`

```
> imap_dfr(list(a=1,b=2),function(x,n) list(valeur=x,nom=n))
# A tibble: 2 x 2
  valeur nom
  <dbl> <chr>
1     1 a
2     2 b
```

`iwalk`

- Ces fonctions sont complétées par des fonctions à suffixe comme dans `map_chr`.

Un exemple D'un cube Excel vers un format long

T202.xlsx - LibreOffice Calc

Fichier Édition Affichage Insertion Format Données Outils Fenêtre Aide

Arial 10

B7 = 20375036

	A	B	C	D
1	T202 : Emploi salarié en fin d'année par département et régi			
2	SEXE : Ensemble			
3	NA38 : Tous secteurs			
4				
5	Année	1989	1990	
6	Niveau géographique			
7	France (hors Mayotte)	20 375 036	20 623 156	
8	Total DOM	335 232	353 870	
9	971-Guadeloupe	77 818	79 527	
10	972-Martinique	95 688	94 008	
11	973-Guyane	26 015	26 488	
12	974-La Réunion	135 711	153 847	
13	France métropolitaine	20 039 804	20 269 286	
14	Auvergne-Rhône-Alpes	2 395 622	2 428 880	

Feuille 1 sur 144 PageStyle_E - T

Lecture de l'ensemble des feuilles

```
> readxl::excel_sheets("T202.xlsx") %>%
```

```
Map(function(x) import("T202.xlsx", skip=4, sheet=x), .) %>%
```

```
map(function(x) {colnames(x)[1] <- "Geo"; x[-1,]}) %>%
```

```
map(function(x) gather(x, Annee, Valeur, -Geo)) %>%
```

```
imap_dfr(function(x, n) {
  x$Secteur <- n
  x$Valeur <- as.numeric(x$Valeur)
  x}) -> r
```

Ajout du nom de chaque feuille, conservé par `Map`
Et concaténation du tout

RGui (64-bit) - [Data: r]

Fichier

	Geo	Annee	Valeur	Secteur
1	France (hors Mayotte)	1989	20375036	E - T
2	Total DOM	1989	335232	E - T
3	971-Guadeloupe	1989	77818	E - T
4	972-Martinique	1989	95688	E - T
5	973-Guyane	1989	26015	E - T
6	974-La Réunion	1989	135711	E - T
7	France métropolitaine	1989	20039804	E - T
8	Auvergne-Rhône-Alpes	1989	2395622	E - T

Exercice rapide

Les fonctions map de *purrr*

- 1- Utilisez la fonction `map_chr` pour **récupérer directement le vecteur des chaînes de caractères** donnant les classes des colonnes de la table de populations.
- 2- Utilisez la fonction `map_dfr` à la place de `map_chr`. La fonction s'attend à ce que chaque itération produise une liste de valeurs correspondant aux variables de la ligne à rajouter.
- 3- Le résultat précédent ne comporte pas le nom de la colonne initiale, or, puisque qu'un data frame est une liste de colonnes nommées, on peut utiliser à chaque itération de la boucle à la fois le contenu de la colonne et son nom en passant par la fonction `imap_dfr` et une fonction à deux arguments.

Construisez la table complète : nom, classe.

Itérer un nombre fixe de fois

La fonction `for`

- Pour les adeptes d'une programmation moins fonctionnelle, R offre la fonction `for` qui répète un traitement suivant les valeurs d'un « indice ».


Malgré sa syntaxe, il s'agit bien d'une fonction !

- Syntaxe :

`for (<var> in <liste>) <expression>`

⇒ Utiliser la fonction accolades lorsqu'une seule expression ne suffit pas.

⇒ `<var>` est l'« indice » conducteur de la boucle : c'est **une variable (de type quelconque) qui sera créée dans l'environnement englobant la boucle** (une variable globale dans les exemples ci joints)

 Le `for` est donc non seulement plus « rétro » mais aussi moins « propre » que les solutions à base de « map ».. Il est par ailleurs plus difficile à paralléliser.

- Le résultat de l'appel de la fonction `for` est toujours `NULL`

```
> ( `for` (i, list("a", "b")1:2, print(i)) )
[1] "a"
[1] "b"
NULL
```

```
> (for (i in list("a", "b")) print(i))
[1] "a"
[1] "b"
NULL
> i
[1] "b"
```

```
> class(`for`)
[1] "function"
```

```
> library(ggformula)
> for (x in c("hp", "wt", "qsec"))
  print(gf_boxplot(mtcars, ~ .data[[x]]))
```

Le `print` est indispensable pour générer les graphiques, car `for` n'imprime rien et a un résultat vide.



Avertissement Ne bouclez pas !

- Ecrire une boucle n'est pas une fatalité. En R **la structure de base n'est pas la donnée élémentaire mais quasi systématiquement un ensemble de données**, sous forme soit de liste, soit de vecteur.
- De nombreuses opérations fonctionnent donc spontanément sur l'intégralité de vecteurs en produisant d'autres vecteurs. Et ce mode de travail se propage dans les fonctions définies par l'utilisateur.

```
> x <- c("a", "b", "c")  
> for (i in seq_along(x)) x[i] <- str_to_upper(x[i])  
> x  
[1] "A" "B" "C"
```

Se simplifie en :

```
> x <- c("a", "b", "c")  
> x <- str_to_upper(x)  
> x  
[1] "A" "B" "C"
```

Il faut donc se méfier de tout code qui contient une boucle autour de la référence à un unique élément d'un vecteur. Il est possible que le code écrit pour ce seul élément marche naturellement sur l'ensemble du vecteur sans nécessité de boucler.

Exercice rapide

La fonction `for`

Intuitivement, ce qu'on a réalisé avec `imap_dfr`, revient à ajouter successivement à la table résultat une ligne pour chacune des colonnes.

- 1- Commencez par créer la structure de la table résultat (nom, classe), initialisée avec des vecteurs de chaînes de caractères de longueur nulle.
- 2- Puis avec une boucle `for` indicée par le numéro de colonne et la fonction `add_row` ajoutez successivement à cette table chacune des lignes formée par le nom de la colonne fourni par la fonction `colnames` ainsi que sa classe.
- 3- Pour n'utiliser que du R de base, reportez la création du data frame à la fin et utilisez le `for` pour construire les futures colonnes. Qu'est ce qui n'a plus sa place dans la boucle ?

while : Itérer jusqu'à ce qu'une condition se réalise

`while` fait partie des mots réservés du langage et propose une fonction qui évalue une expression tant qu'une condition est réalisée.

while (<condition>) <expression>

Cette situation est assez fréquente lorsqu'on lit un fichier petit à petit ou pour lire une grosse table de données d'un SGBD par morceaux :

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
```

Création d'une base en mémoire avec une copie de mtcars

```
dbWriteTable(con, "mtcars", mtcars)
```

```
rs <- dbSendQuery(con, "SELECT * FROM mtcars")
```

Initialisation de la lecture

```
while (!dbHasCompleted(rs)) {
  chunk <- dbFetch(rs, 10)
  print(nrow(chunk))
}
```

Le traitement à réaliser :

- Récupération de 10 observations maximum
- Affichage du nombre d'observations récupérées
- ...jusqu'à ce qu'il n'y en ait plus :
- `dbHasCompleted` vrai

```
dbClearResult(rs)
dbDisconnect(con)
```

XML acte II scène 1

« Voulez vous répéter, s'il vous plait ? »

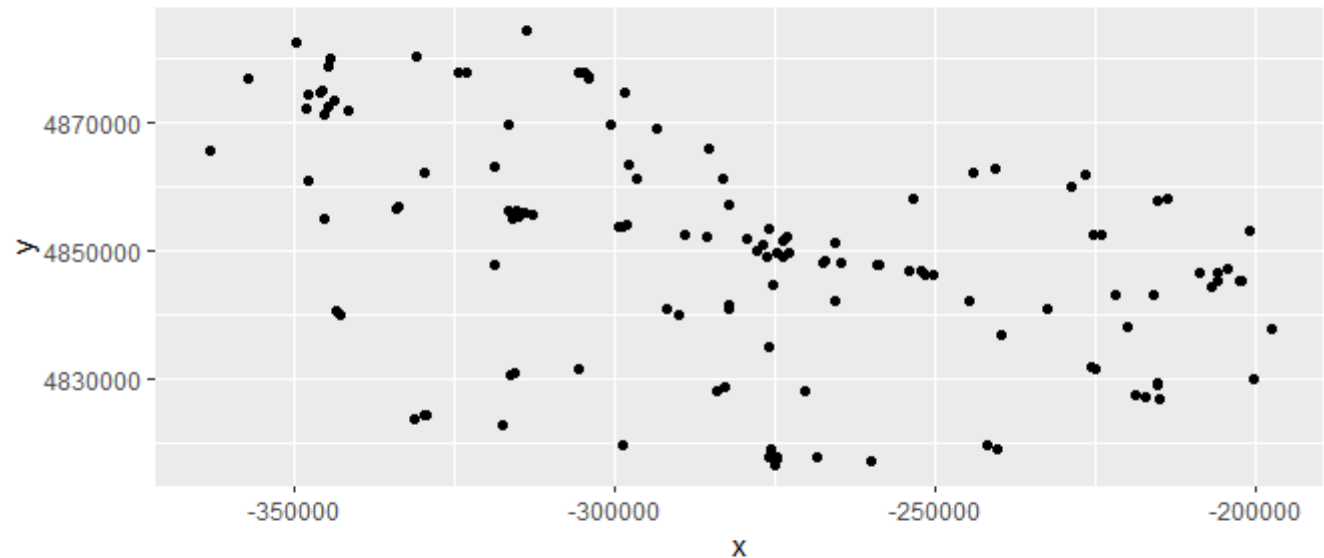
⇒ Séquence B « les boucles »

- 1- Avec la fonction `Map` récupérez l'ensemble des noms de ville. Le résultat est une liste où chaque élément est nommé. Utiliser la fonction `unnname` pour enlever ces noms superflus.
- 2- Recommencez avec la fonction `map_chr` du package ***purrr*** pour récupérer un vecteur de chaînes de caractères au lieu d'une liste.
- 3- Les résultats précédents ne contiennent aucune référence au point de vente, empêchant tout rapprochement, par exemple de la ville et de l'adresse d'un même point de vente.

Recommencez en bouclant non sur les éléments de la liste mais sur le numéro d'item dans la liste et en produisant à chaque itération une liste nommée de deux composants : le numéro d'item '*no*' et la ville '*ville*'. Utiliser la fonction `map_dfr` pour faire la boucle et construire un résultat final sous forme de data.frame. Appliquer aux champs '*ville*' et '*adresse*'.

XML acte II scène 2

La localisation des points de vente



```
# A tibble: 145 x 6
  no id      latitude longitude cp      pop
  <int> <chr>    <chr>    <chr>    <chr> <chr>
1     1  1 220000001 4852000 -279200 22000 R
2     2  2 220000002 4852200 -273200 22000 R
3     3  3 220000003 4849800 -274700 22000 R
4     4  4 220000004 4850836 -276785 22000 R
5     5  5 220000005 4851449 -273652 22000 R
6     6  6 220000009 4849806 -274533 22000 R
7     7  7 220000010 4850836 -276785 22000 R
8     8  8 220000011 4849806 -274533 22000 R
9     9  9 221000001 4846400 -208700 22100 R
10    10 10 221000004 4847100 -204400 22100 R
# ... with 135 more rows
```

Rare

1

Important

1

Utile

1

Rare

2

```
dm <- function(...,expr) {
  c <- as.list(match.call())
  n <- c[2:(length(c)-1)]
  t <- unlist(Map(Negate(is.name),n))
  if (any(t)) stop("Bad parameter name.")
  function(...) {
    l <- list(...)
    if (length(l) != length(n)) stop("Wrong number of arguments.")
    names(l) <- if (is.null(names(l))) as.character(n)
    else ifelse(names(l) == "", as.character(n), names(l))
    e <- do.call(substitute, list(c$expr, l))
    eval(e, envir=-2)
  }
}
```

Séquence R4

Les attributs d'un objet

Un objet contient des données... et plus

Nom

Données

Compléments d'info

Lire les attributs d'un objet (1/2)

attributes

- Tous les objets de R peuvent être complétés par des informations supplémentaires (« attributs ») qui se présentent sous la forme de couples nom - valeur.
- La fonction `attributes` restitue une liste de l'ensemble des attributs d'un objet.
- En pratique, pour le fonctionnement courant de R, ces attributs sont massivement utilisés.

Par exemple, un data frame, c'est juste une liste avec quelques attributs :

```
> "IGoR/data/poplegale_6815.sas7bdat"%>%
  import() %>% head() %>%
  attributes()
```

```
$label
[1] "POPLEGALE_6815"
```

Un effet secondaire de la fonction de lecture des fichiers SAS

Les noms des colonnes sont là !

```
$names
[1] "DC"      "NCC"      "PMUN15" "PMUN10" "PMUN06" "PSDC99" "PSDC90" "PSDC82" "PSDC75" "PSDC68"
[11] "D"      "REGION"
```

```
$row.names
[1] 1 2 3 4 5 6
```

Les lignes aussi ont des noms, par défaut leur numéro

```
$class
[1] "data.frame"
```

L'attribut clé : l'objet est un peu plus qu'une liste

Lire les attributs d'un objet (2/2)

attr

- Il existe des fonctions spécifiques pour accéder aux attributs les plus importants :

```
> "IGoR/data/poplegale_6815.sas7bdat"%>% import() %>%
  names()
[1] "DC"      "NCC"      "PMUN15" "PMUN10" "PMUN06" "PSDC99" "PSDC90" "PSDC82" "PSDC75" "PSDC68"
[11] "D"       "REGION"
```

```
> "IGoR/data/poplegale_6815.sas7bdat"%>% import() %>% head() %>%
  row.names()
[1] "1" "2" "3" "4" "5" "6"
```

```
> "IGoR/data/poplegale_6815.sas7bdat"%>% import() %>%
  class()
[1] "data.frame"
```

- Pour des attributs quelconques, la fonction 'attr' fournit un outil général :

```
> "IGoR/data/poplegale_6815.sas7bdat"%>% import() %>%
  attr('label')
[1] "POPLEGALE_6815"
```

```
> "IGoR/data/poplegale_6815.sas7bdat"%>% import() %>%
  attr('class')
[1] "data.frame"
```

Positionner les attributs d'un objet

- Les fonctions de lecture d'un attribut sont "réversibles" : on peut s'en servir pour positionner ou changer un attribut donné. En fait, comme on ne modifie jamais un objet en R, on va créer une copie avec l'attribut ajouté ou modifié et la substituer à l'original : ceci impose que l'argument (l'original) soit un symbole et non un objet.
- Pour effacer un attribut, on affectera la valeur `NULL`.

```
> p <- import("IGoR/data/poplegale_6815") %>% head(1)
```

```
> attr(p, 'label') <- 'essai'
```

```
> class(p) <- NULL
```

← La table ne sera plus qu'une liste ordinaire

```
> str(p)
```

```
List of 12
```

```
$ DC : chr "01001"
```

```
$ NCC : chr "L' Abergement-Clémenciat"
```

```
$ PMUN15: num 767
```

```
$ PMUN10: num 784
```

```
$ PMUN06: num 811
```

```
$ PSDC99: num 728
```

```
$ PSDC90: num 579
```

```
$ PSDC82: num 477
```

```
$ PSDC75: num 368
```

```
$ PSDC68: num 347
```

```
$ D : chr "01"
```

```
$ REGION: chr "82"
```

```
- attr(*, "label")= chr "essai"
```

```
- attr(*, "row.names")= int 1
```

row.names est un souvenir du data frame

ATTENTION : N'utiliser `attr` qu'en l'absence de fonction spécifique comme `names`, `class`... ces dernières font toujours plus de contrôles de cohérence.

class

ne donne pas forcément un résultat unique

Le système de définition de types qui existe par défaut, qu'on appelle **S3**, est très rudimentaire.

- **Les types de données construits au dessus des types de base** (les types dit atomiques, les vecteurs et les listes) n'existent pas en tant qu'objets, et **ne sont que des attributs positionnés sur les objets de leur famille**. On a vu précédemment qu'il suffisait d'effacer l'attribut `"class"` d'un data frame pour que celui redevienne une simple liste.
- La hiérarchie des types de données est réalisée par le même moyen rudimentaire : pour donner de nouvelles propriétés à un objet par le biais de l'appartenance à un nouveau type, on se contente de rajouter le nom de ce nouveau type à la liste (vecteur) des noms des classes associées à l'objet. C'est ce qui se produit pour les 'tibble' :

```
> haven::read_sas("IGoR/data/poplegale_6815.sas7bdat") %>% class()  
[1] "tbl_df"      "tbl"        "data.frame"
```



La conséquence pratique est que pour tester l'appartenance à un type de données, il vaut toujours mieux passer par les fonctions spécifiques de test de type comme `'is.data.frame'`.

La séquence sur les « fonctions génériques » **S3** démontrera le processus.

Plusieurs résultats pour une fonction

Il n'est pas rare qu'une fonction produise plusieurs résultats : un résultat principal et des résultats secondaires, produits fatals du calcul. C'est le cas de la division qui produit essentiellement un quotient mais aussi, comme sous produit, un reste. Il existe plusieurs stratégies :

- **Faire plusieurs fonctions**, mais le calcul devra être relancé pour chacun des résultats (cf. la division entière : `%/%` et `%%`),
- **Collecter tous les résultats dans une liste**, mais il y a un surcoût en termes de codage pour extraire de la liste le résultat principal.
- **Restituer le résultat principal en lui adjoignant sous forme d'attributs les résultats secondaires**, de cette façon l'accès au résultat principal est immédiat, mais il faudra extraire les résultats secondaires.

Un exemple , la fonction `regexpr` restitue la position d'une expression régulière mais aussi la longueur du « match » :

```
> i <- regexpr(".$", "abc")           # N'importe quel caractère en fin
> str(i)
int 3                                La position obtenue (le résultat principal)
- attr(*, "match.length")= int 1    Le nombre de caractères du « match »
- attr(*, "index.type")= chr "chars"
- attr(*, "useBytes")= logi TRUE
```

Exercice rapide

Métadonnées d'un fichier 'feather'

Les fichiers '**feather**' (package *feather*) sont une alternative aux fichiers '**fst**' en termes de fonctionnalités d'accès aux données par morceaux ou d'accès aux métadonnées sans charger le fichier.

1- Charger le fichier 'poplegale_6815.sas7bdat' du répertoire 'lGoR/data', et le **sauvegarder en format 'feather'**.

2- Avec la fonction *feather_metadata* du package *feather*, **récupérez l'information sur le contenu du fichier**. Cette fonction ne charge pas les données en mémoire, mais uniquement les métadonnées. Affichez la structure de cette information.

Les types de colonnes sont contenus dans un vecteur nommé de chaînes de caractères : les noms des colonnes sont des attributs des types de données.

3- Pour savoir quelles sont les colonnes de type "character", sélectionnez les éléments correspondants du vecteur des types et extrayez en l'attribut donnant le nom. Résultat attendu :

```
[1] "DC"      "NCC"      "D"      "REGION"
```

4) Transformez le code précédent en fonction et listez les colonnes de type "double".

```
> contenu("poplegale_6815.feather", "double")  
[1] "PMUN15" "PMUN10" "PMUN06" "PSDC99" "PSDC90" "PSDC82" "PSDC75" "PSDC68"
```

XML acte II scène 2

« Vous avez de bien beaux attributs... »

⇒ Séquence R4 « attributs d'un objet »

Les données fixes relatives à un point de vente (longitude, latitude...) se présentent non sous forme de liste mais sous forme d'attributs d'une liste vide.

- 1- Avec la fonction `attributes`, afficher les caractéristiques fixes du premier point de vente.
- 2- Ces caractéristiques commencent par un attribut de gestion des listes indépendant du contenu du XML, le sauter.
- 3- Avec la fonction `map_dfr`, collecter toutes les caractéristiques de tous les points de vente dans un seul data frame.
- 4- Faire une carte pour vérifier le résultat.
- 5- La table obtenue ne va pas pouvoir être appariée aux données d'adressage précédemment obtenues. Il faut enrichir chaque ligne avec un numéro de point de vente. Utiliser la fonction `append` pour compléter la liste issue des attributs par le numéro d'ordre de point de vente et boucler sur les numéros d'ordre.

XML acte 2 scène 3

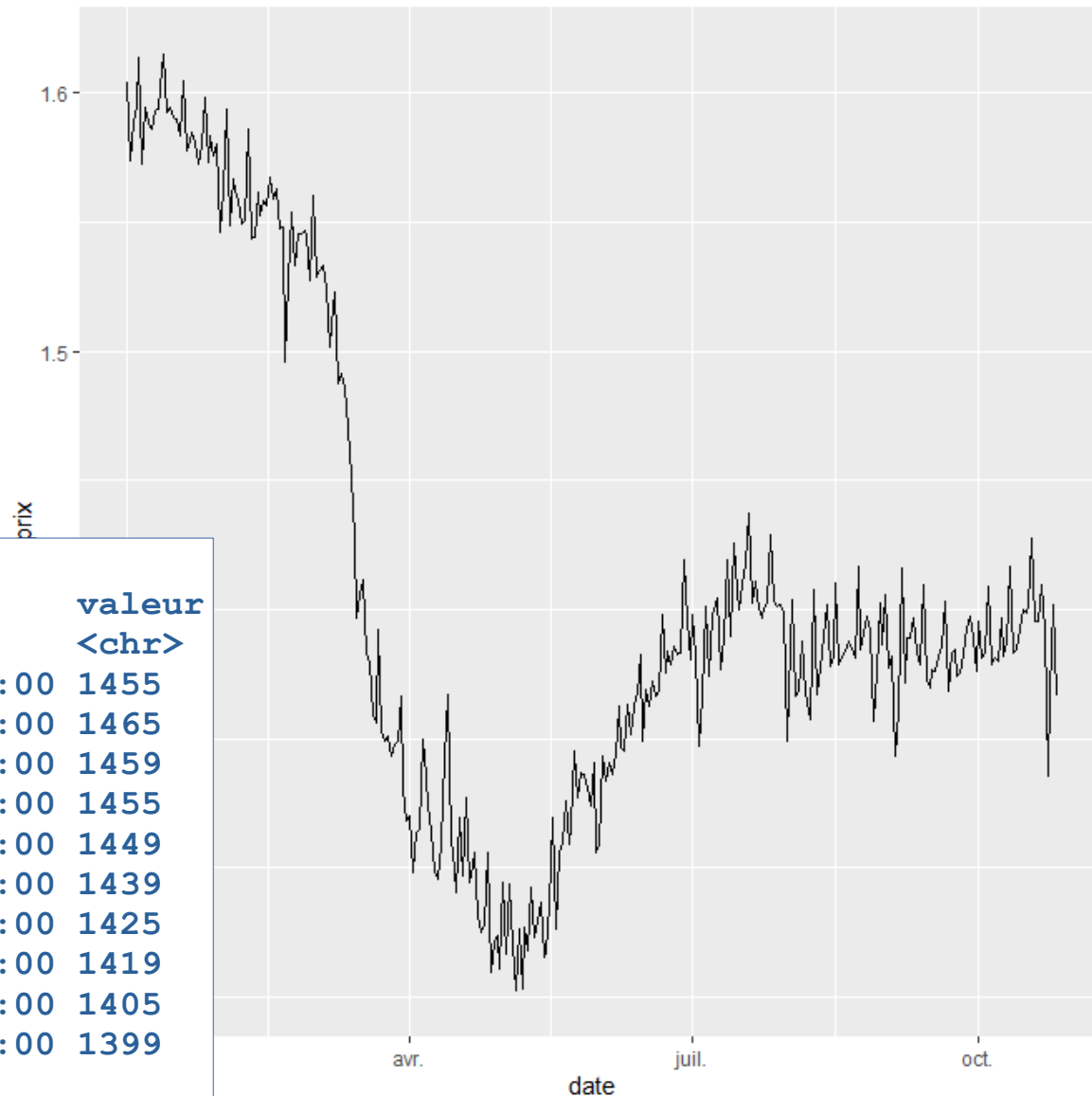
L'évolution du prix du « sans plomb 98 »

Relevés de prix par pompe et carburant

```
# A tibble: 30,412 x 5
```

	no	nom	id	maj	valeur
	<int>	<chr>	<chr>	<chr>	<chr>
1	1	Gazole	1	2020-01-03T08:45:00	1455
2	1	Gazole	1	2020-01-07T08:45:00	1465
3	1	Gazole	1	2020-01-15T08:45:00	1459
4	1	Gazole	1	2020-01-17T08:45:00	1455
5	1	Gazole	1	2020-01-21T08:45:00	1449
6	1	Gazole	1	2020-01-22T08:45:00	1439
7	1	Gazole	1	2020-01-28T08:45:00	1425
8	1	Gazole	1	2020-01-29T08:45:00	1419
9	1	Gazole	1	2020-01-31T08:45:00	1405
10	1	Gazole	1	2020-02-03T08:45:00	1399

```
# ... with 30,402 more rows
```



Important

3

Utile

2

Rare

2

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l)!=length(n)) stop("Wrong number of arguments.")  
    names(l)<- if (is.null(names(l))) as.character(n)  
               else ifelse(names(l)=="",as.character(n),names(l))  
    e <- do.call(substitute,list(c$expr,l))  
    eval(e,envir=-2)  
  }  
}
```

Séquence C

Réaliser des traitements conditionnels

Les alternatives
La gestion des erreurs

Des traitements conditionnels avec `if` (1/2)

La fonction `if` (c'est une fonction , comme le reste) restitue le **résultat de l'évaluation d'une expression ou d'une autre** selon que le résultat d'un calcul booléen renvoie vrai ou faux.

`if (<expression>) <expression1> else <expression2>`

Les parenthèses sont obligatoires.

La branche 'else' est facultative.

En l'absence de branche `else`, `if` restitue `NULL` lorsque la condition est fausse. C'est pratique quand on construit une liste dont certains éléments seront peut être absents :

```
> z <- 8
> c("Le nombre z est",
    if (z%%3==0) " divisible par 3",
    if (z%%4==0) " divisible par 4",
    if (z%%5==0) " divisible par 5")
[1] "Le nombre z est" " divisible par 4"
```

→ `c("Le nombre z est",
NULL,
" divisible par 4"
NULL)`



ATTENTION : `if` ne doit pas être confondue avec `ifelse` qui travaille sur des vecteurs. Appliquée à un vecteur, `if` ne testera que le premier élément (avec avertissement).

```
> z <- c(1,2,3)
> if (z%%2==1) -z else z
[1] -1 -2 -3
Warning message:
In if (z%%2 == 1) -z else z :
la condition a une longueur > 1 et seul le premier élément est utilisé
```


Des traitements conditionnels avec `if` (2/2)

ATTENTION : Les accolades ne sont pas une notation syntaxique pour regrouper des instructions mais une vraie fonction sans lien particulier avec `if`. Les accolades sont utiles lorsqu'on a plusieurs traitements indépendants à réaliser dans chaque branche, mais les vrais délimiteurs du `if` sont les fins de ligne !

```
> if (TRUE) {  
  0  
} else {  
  2  
} + 3  
[1] 0
```

Le '+3' est relié au 'else' malgré la présence des accolades

```
> if (TRUE) 0  
else 1  
[1] 0  
Erreur : 'else' inattendu(e) in "else"
```

L'instruction se termine à '0'
et 'else' semble en démarrer une autre

La fonction `switch`

- La fonction `switch` permet d'écrire des alternatives à plus de deux possibilités, mais sur la base d'un résultat qui est une chaîne de caractères : le premier argument de la fonction, donnera le sélecteur, les autres devront se présenter sous la forme d'une correspondance entre une chaîne de caractères et une valeur.
- On peut préciser le même résultat pour plusieurs étiquettes consécutives en omettant la valeur. Le résultat sera la première valeur présente.

Ici FST et fst sont synonymes :

```
> f <- function(file)
  switch(str_extract(file,"(?<=[.]).*$"), # le type du fichier
    fst = ,                               # si c'est 'fst', faire comme suit
    FST = metadata_fst(file),             # si c'est 'FST'...
    feather = metadata_feather(file) # si c'est 'feather'...
  )
```

- Si aucune branche de l'alternative n'est prise, la fonction `switch` ne fait rien (sans avertissement, mais elle restitue `NULL`)

Les opérateurs non vectoriels && et ||

Les opérateurs `&` et `|` sont vectoriels et particulièrement adaptés à des fonctions comme `ifelse`. Avec la fonction `if` qui, au contraire, ne travaille que sur le premier élément d'un vecteur lorsqu'on lui passe un vecteur, des opérateurs vectoriels ne présentent pas d'intérêt

Ils sont donc complétés par `&&` et `||` qui ont en outre la propriété de **ne faire que les calculs absolument nécessaires à l'obtention du résultat**.

Par exemple :

```
> x <- 1
> if ((x==1) || oops(42)) message("ok")
ok
```

Alors que :

```
> x <- 1
> if ((x==1) | oops(42)) message("ok")
Error in oops(42) : impossible de trouver la fonction "oops"
```



ATTENTION : A l'inverse, il ne faut pas utiliser ces opérateurs dans un cadre vectoriel, comme un filtre dans un data.frame. Le résultat serait très probablement faux.

Exercice rapide

La fonction if

1- Ecrivez une fonction qui teste si son argument est `NULL` (qu'on appellera `recode.null`) et qui restitue l'argument si ce n'est pas vrai, une chaîne vide sinon.

Appliquez votre fonction à l'attribut `"label"` des colonnes `PSDC99`, `DC` de la table de populations.

2- Puis, en s'inspirant du premier exercice sur les boucles, construire un data frame contenant le nom et l'étiquette de chaque colonne de la table de populations.

```
> recode.null(attr(p$DC, 'label'))  
[1] ""
```

```
> recode.null(attr(p$PSDC99, 'label'))  
[1] "Population sans double compte 1999"
```

```
> map...  
# A tibble: 9 x 2  
  nom      label  
  <chr>   <chr>  
1 DC      ""  
2 PMUN15  "Population municipale 2015"  
3 PMUN10  "Population municipale 2010"  
4 PMUN06  "Population municipale 2006"  
5 PSDC99  "Population sans double compte 1999"  
6 PSDC90  "Population sans double compte 1990"  
7 PSDC82  "Population sans double compte 1982"  
8 PSDC75  "Population sans double compte 1975"  
9 PSDC68  "Population sans double compte 1968"
```

Erreurs, avertissements et messages

R offre trois façons d'émettre des messages depuis un programme :

- avec la fonction `message`, juste **un simple message en passant**, comme fait `print` mais en plus concis,
- avec la fonction `warning` **un message d'avertissement sur une situation anormale qui n'interrompt pas le fonctionnement du programme**,
- avec la fonction `stop` **un message d'erreur suivi de l'arrêt du programme**

Ces messages ne sont pas que des impressions à la console, ils sont également accompagnés d'un signal qui peut être intercepté, ou comme avec le package **`log4r`**, qui peut être utilisé pour garder une trace du comportement du programme.

⇒ **Lorsqu'il s'agit de suivre le comportement d'un programme, la fonction `message` doit donc être systématiquement préférée à la fonction `print`.**

Exemple

```
> f <- function (t) {  
  if (t<25) message("Tout va bien.")  
  else if (t<40) warning("Il commence à faire chaud!")  
  else stop("Arrêt d'urgence déclenché : température hors limites!!!")  
  print("ok")  
  paste0("Temperature = ",t,"°C")  
}
```

```
> f(20)  
Tout va bien.  
[1] "ok"  
[1] "Temperature = 20°C"
```

La différence entre 'print' et 'message'

```
> f(35)  
[1] "ok"  
[1] "Temperature = 35°C"  
Warning message:  
In f(35) : Il commence à faire chaud!
```

Le résultat,
puis les warnings

```
> f(40)  
Error in f(40) : Arrêt d'urgence déclenché : température hors limites!!!
```

Le programme n'est pas allé plus loin.
Pas de résultat

La fonction `tryCatch`

- Les messages d'erreur cachent une structure plus complexe qu'une simple chaîne de caractères. En R, **une erreur est un objet d'une classe particulière "condition"**. C'est également un "évènement" qu'il est possible d'intercepter.
 - Pour éviter que le programme ne stoppe
 - Pour déclencher un traitement complémentaire
- R ne propose pas de nomenclature de codes d'erreur et chaque programmeur peut émettre le message d'erreur qu'il veut. Cela rend l'interception d'une erreur particulière délicate, voire impossible en contexte multi-langue.

Le fait d'intercepter une erreur n'est donc généralement utile que pour assurer une sortie propre à une section de programme lorsque les interactions avec un utilisateur peuvent conduire à n'importe quel comportement.

Exemple

```
> f <- function (x){  
  r <- tryCatch(eval(parse(text=x)), # ce qu'il faut tenter de faire  
               error=function(e) e) # quoi faire des éventuelles erreurs  
               #error=identity  
  if ("condition" %in% class(r)) {  
    message(paste0("Echec : ", r$message))  
    r <- NULL  
  }  
  message("Fin.")  
  r  
}
```

En cas d'erreur le résultat du tryCatch
est le résultat de la fonction du paramètre 'error',
ici, l'erreur, une structure contenant le message d'erreur
sinon c'est le résultat du calcul

```
> f("6*7")  
Fin.  
[1] 42
```

```
> f("6*")  
Echec : <text>:2:0: unexpected end of input  
1: 6*  
   ^
```

En anglais

```
Fin.  
NULL
```

Et pourtant tous deux
viennent du noyau !

```
> f("truc")  
Echec : objet 'truc' introuvable  
Fin.  
NULL
```

En français

Terminer une fonction prématurément : la fonction `return`

- Le résultat d'une fonction définie par
 - une unique expression, est la valeur de cette expression
 - une liste d'expressions entre accolades, est la valeur de la dernière expression de l'accolade
- Dans certains cas on peut souhaiter que le résultat soit produit avant la fin de l'évaluation de l'expression (1^{er} cas) ou sans évaluer le reste de la liste d'expressions (2nd cas) .
- La fonction `return` permet de quitter prématurément une fonction. Elle permet de préciser au passage le résultat de la fonction.

Exemple

```
> f <- function (x) {  
  if (x<=1) return(1)      # Sortie prématurée : pas d'évaluation de la suite  
  x * f(x-1)              # Sortie normale : pas besoin de 'return'  
}  
  
# f <- function (x) if (x<=1) 1 else x*f(x-1)  
  
> f(3)  
[1] 6
```

ATTENTION : La fonction `return` semble provenir directement des langages de programmation classiques où exprimer explicitement le résultat d'une fonction est toujours obligatoire. **Mais R n'est pas un langage de programmation classique et l'abus de `return` peut être considéré comme un manque de style.**

Important

2

Avancé

1

Rare

2

Eviter

1

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l) != length(n)) stop("Wrong number of arguments.")  
    names(l) <- if (is.null(names(l))) as.character(n)  
                  else ifelse(names(l) == "", as.character(n), names(l))  
    e <- do.call(substitute, list(c$expr, l))  
    eval(e, envir=-2)  
  }  
}
```

Séquence F3

Les environnements

Le lien entre objets et symboles peut être éphémère

Des assignations locales

- **Les associations valeur-nom réalisées par la fonction `<-` n'ont qu'une portée locale** : à l'intérieur de la définition d'une fonction elles ne sont effectives qu'à l'exécution de cette fonction, même si un symbole de même nom préexistait en dehors de la fonction.

```
> z <- 1
> f <- function(x) z <- x
> f(0)
> z
[1] 1
```

- Ceci n'est pas une particularité des symboles : **toute référence à `<-` n'est que locale y compris lorsqu'il s'agit d'une fonction de « modification ».**

```
> v <- 1:10
> f <- function(x) v[1] <- x
> f(0)
> v
[1] 1 2 3 4 5 6 7 8 9 10
```

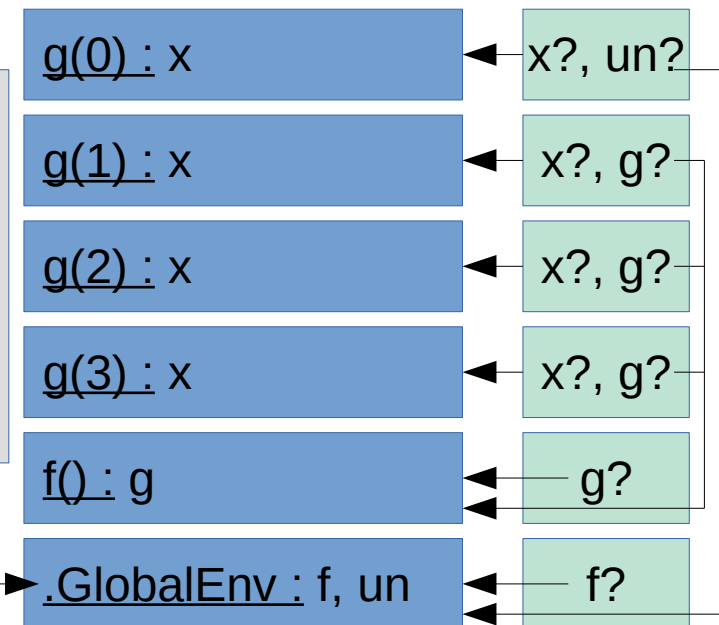
Un empilement d'environnements

- Chaque exécution de fonction, y compris l'évaluation de ses paramètres se passe dans un contexte propre, un « environnement » ('environment'). On obtient généralement un **empilement d'environnements dont la base de la pile est l'environnement global** (cf. la fenêtre de RStudio), aussi accessible via le symbole '.GlobalEnv'.

```
> un <- 1
> f <- function () {
  g <- function (x) if (x>0) x*g(x-1) else un
  g(3)
}

> f()
[1] 6
```

La recherche de la valeur associée à un symbole implique le parcours de la pile d'environnements



- REMARQUE : l'utilisation d'un package implique aussi l'utilisation d'un environnement spécifique afin de cloisonner le contexte de travail du package.

Viols de frontière (1/2)

- Plusieurs fonctions permettent de ne pas respecter le cloisonnement créé par l'exécution d'une fonction. Ce sont des fonctions qui acceptent un **argument supplémentaire spécifiant l'environnement précis où doit être créée ou lue l'association symbole – valeur**.

```
> z <- 1
> f <- function(x) assign("z",x,envir=.GlobalEnv)
> f(0)
> z
[1] 0
```

```
> z <- 1
> f <- function(z) get("z",envir=.GlobalEnv)
> f(0)
[1] 1
```

```
> z <- 1
> f <- function(z) eval(parse(text="z+z"),envir=.GlobalEnv)
> f(0)
[1] 2
```

- NOTE : L'argument 'envir' accepte aussi des références relatives (l'environnement supérieur, deux niveaux au dessus, etc.). A manier avec soin et à n'utiliser qu'en cas de nécessité absolue !

Viols de frontière (2/2)

L'opérateur <<-

L'assignation spéciale <<- permet d'associer une valeur à un symbole appartenant à un environnement supérieur : en remontant, le premier où le symbole est défini.

```
> a <- 1
> b <- 2
> c <- 3
> f <- function(x) {
  a <- 0
  g <- function(y) {a <<- y; b <<- y}
  g(-1)
  c <<- x
  sprintf("a=%d, b=%d",a,b)    # Détaillé en annexe
}
```

```
> f(0)
[1] "a=-1, b=-1"
```

```
> a
```

```
[1] 1
```

```
> b
```

```
[1]
```

```
-1
```

```
> c
```

```
[1] 0
```

'a' : Pas modifié : il y a un 'a' dans l'environnement de 'f', dans 'g', on remonte d'un seul niveau

'b' : Modifié, pas de référence autre que dans l'environnement global dans 'g', on remonte de deux niveaux

'c' : Modifié, pas de référence autre que dans l'environnement global on remonte d'un niveau



A éviter, cela nuit à la lisibilité des programmes, la cible n'étant pas identifiable immédiatement. Et cela ne correspond pas au concept de variable globale !

Exercice rapide

Un contre-exemple pour `<<-`

- Reprendre le code suivant, avec la même logique de traitement (une liste qu'on remplit petit à petit), mais en utilisant la fonction `walk` au lieu du `for` (transformer en fonction la section en rouge)

```
> ldf <- list()
> for (d in c("16","17")) ldf[[d]] <- lire1(d)
> str(ldf)
```

- Avec :

```
lire1 <- function(Dep)
  fst("V:/PALETTES/IGoR/data/nais2017.fst") %>%
  .$depnais==Dep, c("depnais", "agemere")]
```


Un autre contre-exemple

Assignations locales : les fermetures

- **Lors de son exécution** une fonction R travaille sur un univers de symboles connus constitué de :
 - une **copie des associations (symbole - objet) existantes** au moment de la création de la fonction, copie...
 - enrichie des **associations (paramètre - argument)** dues à l'appel et...
 - enrichie, au fil de l'exécution, de toutes les **associations locales** générées par l'usage de l'assignation `<-`.
- **L'environnement d'une fonction est par contre initialisé au moment de la définition de la fonction et non à son appel.**
- La fonction `local`, qui simule l'exécution au sein d'une fonction, permet de créer des fonctions « fermetures » qui contiennent des variables cachées.

```
> local({
  s <- 0
  h <-<- function (x) if (missing(x)) s else s <-<- x
})
> h(42)
> h()
[1] 42
> s
Erreur : objet 's' introuvable
```

Pour sortir de l'environnement de 'local', où la fonction est créée

Pour sortir de l'environnement de la fonction 'h' et « modifier » 's' qui est dans l'environnement de 'local'

Les environnements : un type d'objet

- Les environnements sont le moyen pour R de mémoriser une liste de symboles à utiliser dans un contexte particulier :
 - Le contenu de l'espace de travail `.GlobalEnv`
 - Les symboles d'un package
 - Les symboles locaux lors de l'évaluation d'une fonction
- C'est aussi **une façon de stocker une correspondance entre un nom et une valeur, en permettant un accès particulièrement rapide** (hash code) à la valeur de n'importe quel symbole.
- La fonction `new.env` permet de créer un environnement utilisable comme une base de données clé – valeur.

```
> a <- new.env()
> str(a)
<environment: 0x02efaf10>

> assign("x", 1, envir=a)

> ls(envir=a)
[1] "x"

> get("x", envir=a)
[1] 1
> eval(quote(2*x), envir=a)
[1] 2

# et comme dans une liste
> a$x
[1] 1
> a$x <- 2
> a[["x"]]
[1] 2
```

Fin de la séquence

*« Je ne sais pas bien où aller !
D'ailleurs où suis-je ? »*

⇒ **Séquence C « Traitements conditionnels »**

En revenant à l'affichage du premier point de vente, on constate que les données de prix se trouvent sous une forme similaire à celles d'identification des points de vente. Il y a une liste par relevé, mais cette liste est vide et les données ne sont qu'en attributs de cette liste vide. Chaque liste a pour nom 'prix' (cf. le premier attribut d'un point de vente).

- Utiliser la fonction `imap_dfr` pour récupérer dans un data frame les relevés du 42ème point de vente. On bouclera sur tous les items du point de vente en ne conservant les attributs (fonction `attributes`) que pour ceux provenant d'items nommés 'prix'.

⇒ **Séquence F3 « Environnements »**

- Boucler sur tous les points de vente pour récupérer l'ensemble des relevés.
- Faire un graphique pour vérifier.

Important

1

Rare

2

Avancé

1

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l) != length(n)) stop("Wrong number of arguments.")  
    names(l) <- if (is.null(names(l))) as.character(n)  
                  else ifelse(names(l) == "", as.character(n), names(l))  
    e <- do.call(substitute, list(c$expr, l))  
    eval(e, envir=-2)  
  }  
}
```

Séquence F4

Les fonctions sont des objets

On sait déjà écrire des fonctions mais celles ci ne sont que des objets parmi d'autres,
il s'agit ici d'utiliser des fonctions qui ont des fonctions en entrée
et fournissent des fonctions en sortie

Le cinquième principe

Des objets et des fonctions... qui sont aussi des objets

- Selon la formule “en R tout ce qui existe est objet...”, **les fonctions sont également des objets**. Même si elles ont une représentation interne qui n’a rien à voir avec les données manipulées par le statisticien, les fonctions peuvent apparaître dans les mêmes conditions :
 - comme arguments d’autres fonctions (le premier argument de `Map`),
 - comme éléments de structures de données,
 - comme résultat d’un calcul, ou d’un appel de fonction.

```
> f <- function(x) if (x) `*` else `+`           # une fonction en résultat
> f(TRUE) (6,7)
[1] 42

> (function (x) if (x) `*` else `+`)(TRUE) (6,7)  # un calcul direct
[1] 42

> creer_compteur <- function(){i <- 0; function() { i <- i+1; i} }
> compteur <- creer_compteur()                   # un générateur de fonctions
> compteur(); compteur()
[1] 1
[1] 2
```

La fonction `Vectorize`

La fonction `Vectorize` permet de couvrir le gap entre les fonctions et opérateurs qui travaillent naturellement sur l'ensemble de vecteurs, et ceux qui ne peuvent travailler que sur une donnée à la fois. La fonction `Vectorize` transforme une fonction en une autre qui, elle, est vectorisée (i.e. un vecteur en entrée, un vecteur en sortie).

La fonction 'compter1' n'est pas vectorisée, la valeur de l'argument est transférée directement dans le paramètre .dep et le test revient à faire `.$depnais==c("16","17")`, ce qui donne un résultat, mais un résultat incorrect (`==` n'est pas `%in%`).

```
> compter1 <- function(.dep)
  fst("IGoR/data/nais2017.fst") %>%
  .[$depnais==.dep, "agemere"] %>% # La simplification conduit à un vecteur
  as.numeric() %>%
  length()
> compter1(c("16", "86"))
[1] 3655
Warning message:
In .$depnais == .dep :
  la taille d'un objet plus long n'est pas multiple de la taille d'un objet p$
> f <- Vectorize(compter1)
> f(c("16", "86"))
  16   86
3027 4271
```

REMARQUE : La fonction `Vectorize` ne peut pas être utilisée sur des primitives (comme `class`) : le passage d'arguments à des primitives se fait de façon optimisée. Une solution consiste à les encapsuler dans une définition de fonction usuelle.

R est moins complet que SAS ?

La fonction `Negate`

Le ``not in`` de SAS n'existe pas en standard.

Mais il peut être facilement rajouté aux opérateurs de R car ceux ci ne sont que des fonctions à deux arguments. La **notation entre signes % est utilisée pour définir de nouveaux opérateurs** qui auront tous obligatoirement cette forme.

`Negate` est une fonction qui prend une fonction en entrée et produit une fonction donnant le résultat opposé.

```
> `%not in%` <- Negate(`%in%`)  
  
> liste <- 1:5  
  
> 6 %not in% liste  
[1] TRUE
```

REMARQUE : comme les nom de fonction `%in%` et `%not in%` ne contiennent pas que des lettres, on utilise les "backticks" ``` (altGr-7) qui servent à définir des noms de symbole contenant des caractères spéciaux.

La curryfication

- La « curryfication » consiste à **prendre une fonction à plusieurs arguments et à en geler certains** (mais pas tous) sur des valeurs prédéfinies : le résultat est encore une fonction, mais à nombre d'arguments réduit. La fonction `partial` du package *purrr* réalise cette opération.
- Il s'agit juste d'une question de style pour simplifier les programmes dans un esprit de programmation fonctionnelle. Les deux écritures sont **presque** fonctionnellement équivalentes :

```
> f <- function(x) sprintf("%5.2f",x)
> f(5)
[1] " 5.00"
```

Pourquoi « presque » ?

```
> library(purrr)
> g <- partial(sprintf, "%5.2f")
> g(5)
[1] " 5.00"
```

- Il est possible de « curryfier » à gauche (les premiers paramètres : le défaut) ou à droite (les derniers paramètres)

Exercice rapide

Des fonctions comme paramètres

1- Ecrire une fonction qui extrait, pour un département donné, du fichier **fst** des naissances, la colonne 'agemere' (voir vue sur Vectorize) sous forme numérique puis en calcule la moyenne.

Appliquer au département de Mayotte (976).

Paramétrer la fonction pour qu'elle calcule une statistique quelconque fournie à l'appel et par défaut la moyenne.

Appliquer à la médiane.

Calculer le premier quartile.

2- Modifier la fonction pour qu'elle puisse accepter une liste de fonctions statistiques à calculer.

```
> f <- fst::fst("nais2017.fst")

> calcul <- function(.dep) ???
> calcul("976")
[1] 28.22976

> calcul <- function(.dep, .fun ???)
> calcul("976")
[1] 28.22976
> calcul("976",median)
[1] 28
> calcul("976", ???)
25%
23

> calcul <- function(.dep, .funs ???)
> calcul("976",c(mean,median))
[[1]]
[1] 28.22976

[[2]]
[1] 28

> calcul("976")
[1] 28.22976
```

Corrigé de l'exercice

```
> calcul <- function(.dep, .fun=mean) {  
  data <- fst("IGoR/data/nais2017.fst") %>%  
    [. $depnais==.dep, "agemere"] %>%  
    as.numeric()  
  .fun(data)  
}
```

```
> calcul("976")  
[1] 28.22976
```

```
> calcul("976", median)  
[1] 28
```

```
> calcul("976", function (x) quantile(x, p=.25))  
25%  
23
```

```
> calcul("976", partial(quantile, p=.25))  
25%  
23
```

Corrigé de l'exercice

```
> calcul <- function(.dep,.funcs=mean) {  
  data <- fst("IGoR/data/nais2017.fst") %>%  
    [. $depnais==.dep, "agemere"] %>%  
    as.numeric()  
  if (length(.funcs)==1) .funcs(data)  
  else Map(function (x) x(data), .funcs)  
}
```

```
> calcul("976",fun=list(mean,median))  
[[1]]  
[1] 28.22976
```

```
[[2]]  
[1] 28
```

```
> calcul("976")  
[1] 28.22976
```

Ou Vectorize :
Vectorize(calcul)("976",c(mean,median))

Important

1

Utile

3

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l) != length(n)) stop("Wrong number of arguments.")  
    names(l) <- if (is.null(names(l))) as.character(n)  
                  else ifelse(names(l) == "", as.character(n), names(l))  
    e <- do.call(substitute, list(c$expr, l))  
    eval(e, envir=-2)  
  }  
}
```

Un peu partout !

Séquence F5

Les fonctions génériques

Une fonction peut en cacher une autre !

Les fonctions génériques (1/2)

- Par défaut, R possède un système très simplifié de programmation orientée objet dit S3.
- Certaines fonctions vont avoir un comportement différent suivant la classe de l'objet passé en premier argument, ce sont des **fonctions génériques**.

```
> p <- haven::read_sas("IGoR/data/poplegale_6815.sas7bdat")
```

```
> class(p)
```

```
[1] "tbl df"      "tbl"      "data.frame"
```

$> p$

```
# A tibble: 38,219 x 12
```

DC	NCC	PMUN15	PMUN10	PMUN06
----	-----	--------	--------	--------

```
<chr> <chr> <dbl> <dbl> <dbl>
```

1	01001	L'	A~	767	784	811
---	-------	----	----	-----	-----	-----

2 01002 L' A~ 241 221 198

3 01004 AmbÃ~ 14127 13835 12709

4	01005	AmbÃ~	1619	1616	1436
---	-------	-------	------	------	------

5 01006 Amb1~ 109 116 120

6 01007 Ambr~ 2615 2362 2241

7	01008	Ambu~	747	729	641
---	-------	-------	-----	-----	-----

8	01009	Ande~	342	340	319
---	-------	-------	-----	-----	-----

9 01010 Angl~ 1133 994 900

10	01011	Apr~	390	363	333
----	-------	------	-----	-----	-----

```
# ... with 38,209 more rows, and 1
```

REMARQUE 11. — Soit \mathcal{A} un anneau commutatif et \mathcal{B} un \mathcal{A} -module. Soient \mathcal{C} et \mathcal{D} deux \mathcal{A} -modules. On a :

La fonction `'print'` va ainsi se comporter différemment suivant que l'objet à afficher est de classe `'tbl_df'` ou de classe `'data.frame'` (s'il appartient aux deux classes, c'est la première de la liste qui l'emporte)

- REMARQUE : Il y a d'autres systèmes de programmation objet S4 et surtout R6, non abordés ici. Ils sont plus complets et utilisés dans certains packages nécessitant des structures complexes (cartographie, **shiny**).

Les fonctions génériques (2/2)

- Il serait possible d'avoir une fonction 'print' qui teste le type de son argument et fait des traitements différenciés. Cela ne serait pas compatible avec une prolifération de packages nécessitant cette fonctionnalité.
- Le principe des fonctions génériques est juste d'être marquées comme telles. A l'exécution de la fonction, c'est le moteur de R qui cherchera la fonction adéquate à appliquer, une fonction de nom composé <fonction>.<type>

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x08a789d8>
<environment: namespace:base>

> methods(class="tbl")
[1] $<-          %within%      [[<-          [<-          as.tbl      coerce
[7] coerce<-     count        format         fortify      glimpse    initialize
[13] Ops          print          show          slotsFromS3  tally      tbl_sum
see '?methods' for accessing help and source code
```

Explorer les fonctions génériques

Une fonction cachée d'un package :::

```
> methods("print") %>% {print(length(.)); head(.)}
[1] 303
[1] "print.abbrev"      "print.acf"          "print.all_vars"
[4] "print.anova"       "print.Anova"        "print.anova.loglm"

> tibble:::print.tbl
function (x, ..., n = NULL, width = NULL, n_extra = NULL)
{
  cat_line(format(x, ..., n = n, width = width, n_extra = n_extra))
  invisible(x)
}
<bytecode: 0x04e1fa90>
<environment: namespace:tibble>
```

NB : **'format'** est aussi une fonction générique !

Explorer les fonctions génériques

Obtenir l'accès à une fonction spécialisée

```
> methods(str)
```

```
[1] str.arrow*
[3] str.Date*
[5] str.dendrogram*
[7] str.POSIXt*
[9] str.Rcpp_stack_trace*
[11] str.rlang_envs*
[13] str.vctrs_vctr*

str.data.frame*
str.default*
str.logLik*
str.quosure*
str.rlang_data_pronoun*
str.tbl_df*
```

```
see '?methods' for accessing help and source code
```

Pas exporté, pour y avoir accès, il faut
utils::str.data.frame

ou :

```
getS3method("str","data.frame")
```

```
# - La structure d'un data frame vu comme une simple liste
```

```
# utils::str.default(mtcars)
```

```
# Le package apparaît dans le code de getS3method("str","default")
```

```
> getS3method("str","default")(mtcars)
```

```
$ mpg : num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
$ cyl : num [1:32] 6 6 4 6 8 6 8 4 4 6 ...
$ disp: num [1:32] 160 160 108 258 360 ...
$ hp : num [1:32] 110 110 93 110 175 105 245 62 95 123 ...
$ drat: num [1:32] 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
$ wt : num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
$ qsec: num [1:32] 16.5 17 18.6 19.4 17 ...
$ vs : num [1:32] 0 0 1 1 0 1 0 1 1 1 ...
$ am : num [1:32] 1 1 1 0 0 0 0 0 0 0 ...
$ gear: num [1:32] 4 4 4 3 3 3 3 4 4 4 ...
$ carb: num [1:32] 4 4 1 1 2 1 4 2 2 4 ...
```

Exemple : redéfinir un `||` à la SAS ou SQL

- Dans l'exemple qui suit on cherche à redéfinir l'opérateur `||` de manière à ce qu'il s'applique aussi à des chaînes de caractères. La fonction `||` est une primitive (c.a.d. du code C). On la redéfinit de manière à ce qu'elle devienne une fonction générique...

```
> `||` <- function (e1, e2) UseMethod("||")
```

- ... qui ait le même comportement par défaut que la fonction primitive (un OU non vectorisé qui s'arrête si le premier argument est vrai)

```
> `||.default` <- function (e1, e2) .Primitive("||")(e1, e2)
```

- ... sauf pour les chaînes de caractères pour lesquelles on cherche à faire une concaténation.

```
> `||.character` <- function(e1, e2) paste0(e1, e2)
```

```
> 42||6                # Le comportement par défaut
[1] TRUE
> "42"||"6"            # Notre concaténation
[1] "426"
> "42"||6              # Notre concaténation, après conversion du 2nd argument
[1] "426"
> 42||"6"              # Le comportement par défaut dicté par le 1er argument
[1] TRUE
```

Pour rire

- Les fonctions génériques sont partout. Cela signifie en pratique que beaucoup de fonctions usuelles peuvent très facilement adopter un comportement déroutant.
- L'exemple ci joint n'est que partiellement factice : il est inspiré d'une histoire réelle (la fonction `st_crs` du package `sf`).

```
# Une liste, apparemment...
> machin
$a
[1] 1

$b
[1] 2

attr(,"class")
[1] "truc"

> machin$a
[1] 1

# Un champ qui n'existe pas !
> machin$c
[1] 42

## --- La clé de l'énigme
> `$.truc` <-
  function(x,y)
    if (y %in% names(x)) x[[y]] else 42
> machin <- list(a=1,b=2)
> class(machin) <- "truc"
```

Il y a classe et classe !

- La fonction `class` ne répond pas toujours la même chose que `attr` avec `"class"`. **Pour certains types d'objets usuels de R, la classe n'est pas un attribut mais une propriété interne.**
- Les fonctions génériques, elles, utilisent la classe telle qu'elle est positionnée par `attr` pour trouver la méthode à appliquer.
- Et des fonctions génériques peuvent se cacher à l'intérieur de fonctions de type `"builtin"`.

```
# `+`, une fonction interne et ne fait
# apparemment pas référence à UseMethod
> `+`
function (e1, e2)  .Primitive("+")

# Pourtant il y a des méthodes !
> methods("+")
[1] +.Date      +.POSIXt
see '?methods' for accessing help and so

> `+`.character`<- paste0
> z <- "abc"
> class(z)
[1] "character"
> z + z
Erreur dans z + z : argument non
numérique pour un opérateur binaire

> attr(z,"class")
NULL

> attr(z,"class") <- "character"
> z + z
[1] "abcabc"
```

Important

3

Utile

4

Rare

2

```
dm <- function(...,expr) {  
  c <- as.list(match.call())  
  n <- c[2:(length(c)-1)]  
  t <- unlist(Map(Negate(is.name),n))  
  if (any(t)) stop("Bad parameter name.")  
  function(...) {  
    l <- list(...)  
    if (length(l) != length(n)) stop("Wrong number of arguments.")  
    names(l) <- if (is.null(names(l))) as.character(n)  
                  else ifelse(names(l) == "", as.character(n), names(l))  
    e <- do.call(substitute, list(c$expr, l))  
    eval(e, envir = -2)  
  }  
}
```

Séquence F6

L'évaluation retardée

R est paresseux !
Le 7ème principe
Les macros

L'évaluation des arguments

L'évaluation retardée

- Dans nombre de langages de programmation classiques, le **passage d'arguments** à une fonction se fait généralement **par valeur** : les arguments sont d'abord évalués, puis la fonction est invoquée pour travailler sur les valeurs obtenues.
- **En R les valeurs des arguments ne sont calculées qu'au dernier moment, lorsque la fonction en a vraiment besoin : c'est l'évaluation retardée (« lazy evaluation »).**

```
# Classique en apparence...
> f <- function(x,y) x*y
> f(print(6),print(7))
[1] 6      # par 'print' qui restitue 6
[1] 7      # par 'print' qui restitue 7
[1] 42     # Le résultat (par la REPL)

# L'évaluation au dernier moment
> f <- function(x,y) {
      message("ok")
      x*y }
> f(print(6),print(7))
ok      # ici on est dans 'f' !
[1] 6      # PUIS on évalue x...
[1] 7
[1] 42

# Eventuellement pas d'évaluation !
> f <- function(x,y) 42
> f(print(6),print(7))
[1] 42     # où sont les 'print' ???
```


L'évaluation des arguments

L'évaluation retardée : `match.call`

- Attention: si l'évaluation des arguments se fait quand la fonction a commencé son travail, cette évaluation se fait néanmoins dans l'environnement d'appel pas dans celui de la fonction.
- **L'intérêt du mécanisme est que la fonction peut disposer des conditions de son appel.**

La fonction `match.call` permet de connaître précisément les expressions qui ont été passées en argument, avant qu'ils ne soient évalués.

```
# Classique en apparence...
> y <- 6
> f <- function(x,y) x*y
> f(print(y),print(7))
[1] 6   # le 'y' global pas le parametre
[1] 7
[1] 42

# L'évaluation au dernier moment
> f <- function(x,y) {
  z <- match.call()
  str(z)
  x*y
}
> f(print(6),print(7))
language f(x = print(6), y = print(7))
[1] 6
[1] 7
[1] 42
```

Le septième sceau

Une fonction fait ce qu'elle veut !

- L'expression restituée par `match.call` correspond à la représentation interne de l'appel. Néanmoins, avec `as.list` elle peut être convertie en liste pour pouvoir être retravaillée.
- La fonction peut donc décider si l'évaluation des arguments doit se faire ou non et donc interpréter les arguments comme elle veut. On parle d'« **évaluation non standard** ».

```
> f <- function(x,y) {
  z <- as.list(match.call())
  print(z)
  zx <- z$x
  zx <- eval(substitute(
    substitute(zx,
      list(print=quote(message))))))
  x <- eval(zx)
  x*y
}
```

```
> f(1+print(5),print(7))
[[1]]
f
```

```
$x
1 + print(5)

$y
print(7)

5
[1] 7
[1] 42
```

Le résultat de 'match.call'

Tout 'print' dans le 1^{er} argument est désactivé au profit de 'message'.

Exemple : paramétrer les noms des colonnes “à la dplyr”

La fonction suivante permet de paramétrer la colonne qui sert au filtre :

```
> lireV1 <- function(.col,.val)
  fst("IGoR/data/nais2017.fst") %>%
  .[[.col]]==.val, ]

> lireV1("depnais","86") %>% nrow()
[1] 4271
```

*L'inconvénient est que le nom de la colonne doit apparaître entre quotes, ce qui n'est généralement pas le cas quand on utilise **dplyr**. Pour jouer dans la même cour, on peut exploiter l'évaluation retardée :*

```
> lireV2 <- function(.col,.val){
  call <- as.list(match.call())
  col <- as.character(call$.col)
  fst("IGoR/data/nais2017.fst") %>%
  .[[col]]==.val, ]
}

> lireV2(depnaiss,"86") %>% nrow()
[1] 4271
```

Un exemple réel

Une fonctionnalité utile : un « mouchard »

```
> `:=` <- function(.table,value) {
  call = as.list(match.call())
  expr.function <- as.list(call$value)[[1]]
  table.name <- toString(call$.table)
  time <- system.time(assign(table.name,eval(value),envir=parent.frame()))
  message(sprintf("NOTE: The %s '%s' has %d row(s) and %d column(s)."
                  ,head(class(.table),n=1) ,
                  table.name, nrow(.table),ncol(.table))
  )
  message(sprintf(
    "NOTE: Function '%s' used:\n
      elapsed time %6d seconds\n          system time %6d seconds",
      toString(expr.function) ,
      as.integer(time[3]) ,
      as.integer(time[2]))
  )
  invisible(.table) # Retourne .table, mais ne l'imprime pas comme avec <-
}

> a := data.frame(x=1:length(LETTERS),y=LETTERS)
NOTE: The data.frame 'a' has 26 row(s) and 2 column(s)
NOTE: Function 'data.frame' used:
  elapsed time      0 seconds
  system time      0 seconds
```

Pour cette fonctionnalité,
voir aussi le package *tidylog*

L'évaluation retardée

Les « promesses »

- L'évaluation retardée repose sur la notion de « promesse », un type d'objet qui ne calcule sa valeur que lors qu'on l'utilise.
- Les « promesses » ne sont pas présentes que par les paramètres des fonctions :

On en trouve dans les tables de données associées à certains package « lazy loading » : la table n'est mise en mémoire que si on s'en sert.

On peut (exceptionnellement) en créer soi-même.

- *Du fait de leur comportement, les promesses ne peuvent pas être visualisées avant évaluation, il faut passer par des fonctions du package **pryr** pour savoir si quelque chose est une « promesse » ou en avoir une image sous forme de liste.*

```
> pryr::is_promise(cars)
[1] TRUE # 'cars' pas encore utilisée
```

```
> delayedAssign("x", y*print(6))
> y <- 7
> x
[1] 6      # Evaluation déclenchée !
[1] 42     # Valeur de x
> x
[1] 42     # Pas besoin d'un autre calcul
```

```
> f <- function(x) pryr::promise_info(x)
> f(print(1))
$code      # Le code à évaluer
print(1)

$env        # ... où l'évaluer ?
<environment: R_GlobalEnv>

$evaluated # ... a-t-il été évalué?
[1] FALSE

$value      # pour n'évaluer qu'une fois
NULL
```

Contourner l'évaluation retardée

Les macros

- A son appel une fonction réalise une certaine tâche. **La finalité d'une macro est la même avec une étape intermédiaire de plus : une macro commence par construire le programme qui va effectuer la tâche puis elle l'exécute (l'évalue).**
- L'intérêt des macros est de pouvoir contourner le comportement de nombreuses fonctions qui, s'appuyant sur l'évaluation retardée, n'évaluent pas un de leurs arguments. Et si un argument n'est pas évalué on ne peut pas le paramétrer.

Si on essaie de paramétrer la variable sur laquelle on fait la moyenne, cela ne marche pas :

```
> f <- function(.dep, .var) {
  var <- as.name(.var)      # conversion caractères vers symbole
  fst("IGoR/data/nais2017.fst") %>%
    .[$depnais==.dep, c("indnatm", .var)] %>%
    group_by(indnatm) %>%
    summarise(n = n(),
              age = mean(as.numeric(var)))
}
```

as.numeric cherche à transformer le symbole 'var' non ce qu'il indique ('agemere')

```
> f("976", "agemere")
Error in as.numeric(var) :
  cannot coerce type 'symbol' to vector of type 'double'
```

Des macros avec le package **gtools**

Les macros sont présentes dans divers langages de programmation avec des fonctionnalités plus ou moins évoluées. Souvent le programme à exécuter au final se présente sous forme d'une chaîne de caractères (exemple SAS) qui est construite par substitution puis ensuite compilée.

En R, le package **gtools** apporte la notion de macro sous une forme un peu plus restrictive, on donne :

- un **squelette de programme** qui doit être syntaxiquement correct,
- des **symboles** qui seront substitués ('**substitute**') dans le squelette par les expressions non évaluées passées au moment de l'appel.

```
> f <- defmacro(.dep, .var, expr= {
  var=as.character(quote(.var)) # agemere -> "agemere" sans évaluer
  fst("IGoR/data/nais2017.fst") %>%
    [. $depnais==.dep, c("indnatm",var)] %>%
    group_by(indnatm) %>%
    summarise(n = n(),
              age = mean(as.numeric(.var)))
})
> f("976",agemere)
# A tibble: 2 x 3
  indnatm      n    age
  <chr>    <int> <dbl>
1 1      2314  29.4
2 2      7161  27.9
```

Exercice rapide

Une première macro

Le code suivant permet, sur la table des naissances, de calculer des statistiques sur l'âge de la mère selon les différentes modalités de l'indicateur de nationalité.

```
> naissances %>%  
  group_by(indnatm) %>%  
  summarise(n = n(), age = mean(as.numeric(agemere)))
```

Rajouter une condition en utilisant *dplyr* (fonction *filter* pour ne calculer les statistiques que sur une partie de la table. Construire pour cela une fonction qui acceptera n'importe quelle expression en paramètre.

Tester sur la condition `depnais== "976"`

Corrigé de l'exercice

```
> f <- defmacro(.test,expr=
  naissances %>%
  filter(.test) %>%
  group_by(indnatm) %>%
  summarise(
    age=mean(as.numeric(agemere)),
    n=n())
)
```

```
> f(depnaiss=="976")
# A tibble: 2 x 3
  indnatm   age     n
  <chr>   <dbl> <int>
1 1      29.4  2314
2 2      27.9  7161
```

Le code cible du code

La fonction `substitute`

- La fonction `substitute` est une fonction de base qui permet de **modifier un morceau de code** (c.a.d. une expression non évaluée) en remplaçant les occurrences de certains symboles par d'autres expressions.
- `defmacro` n'est qu'un enrobage autour de `substitute`.
- `substitute` est un instrument puissant pour réaliser de l'évaluation non standard mais quelque peu incommode car il n'évalue pas son argument et demande donc souvent un deuxième passage de substitution.

```
> e1 <- substitute(x*7,
                    list(x=quote(2*3)))
```

```
> e1
2 * 3 * 7
> eval(e1)
[1] 42
```

```
> f <- function (e) {
  e1 <- substitute(e)
  print(e1)
}
> f(x*7)
x * 7
```

```
dm <- function(...,expr) {
  c <- as.list(match.call())
  n <- c[2:(length(c)-1)]
  t <- unlist(Map(Negate(is.name),n))
  if (any(t)) stop("Bad parameter name.")
  function(...) {
    l <- list(...)
    if (length(l) != length(n)) stop("Wrong number of arguments")
    names(l) <- if (is.null(names(l))) as.character(n)
    else ifelse(names(l) == "", as.character(n), names(l))
    e <- do.call(substitute, list(c$expr, l))
    eval(e, envir = -2)
  }
}
```

Contourner l'évaluation retardée

parse et eval

- La fonction `defmacro` de **gtools** impose que le squelette de programme soit syntaxiquement correct, car les substitutions réalisées sont directement faites sur la forme interne du programme (voir la fonction `substitute`).
- Si certains symboles utilisés par le programme doivent être construits, il faut passer par la forme externe : construire la chaîne de caractères représentant le programme final, puis l'évaluer.

Exemple : le nom de la colonne cible doit être construit :

```
> f <- function(.dep, .var)
  eval(parse(text=paste0(
    'fst("IGoR/data/nais2017.fst") %>%
      [. $depnais==.dep, c("indnatm", "age", .var, ")] %>%
      group_by(indnatm) %>%
      summarise(n = n(),
                age = mean(as.numeric(age', .var, ')))'
  )))
> f("976", "mere")
# A tibble: 2 x 3
  indnatm     n  age
  <chr>   <int> <dbl>
1 1      2314  29.4
2 2      7161  27.9
```

Le chaînon manquant

Des fonctions en forme de variables

- Avec la fonction `makeActiveBinding`, R propose un objet étrange qui se comporte comme une fonction “réversible” sans argument dont la forme de l’appel est syntaxiquement celle de la référence à une variable. **C’est un peu ce qui se passe avec les paramètres des fonctions : leur référence déclenche leur évaluation.**
- Cette fonctionnalité permet d’implémenter des variables où l’opération d’assignation d’une valeur est contrôlée pour ne pas permettre l’assignation de n’importe quelle valeur.
- Une utilisation possible est la définition de constantes : des variables où on interdit la réassignation :

```
> constante <- defmacro(NOM,VALEUR,
  expr=makeActiveBinding(
    as.character(quote(NOM)),
    function(v)
      if (missing(v)) VALEUR
      else stop("Tentative de modifier une constante!"),
    .GlobalEnv))
```

```
> constante(DEUX,2) ————— génère —————>
```

```
> DEUX
```

```
[1] 2
```

```
> DEUX <- 1
```

```
Error in (function (v) : Tentative de modifier une constante!
```

```
MakeActiveBinding(
  "DEUX",
  function (v)
    if missing(v) 2
    else stop("Tentative... !"),
  .GlobalEnv)
```

That's all !

Mesurer les performances (1/2)

Le package *microbenchmark*

- La fonction `microbenchmark` du package de même nom répète l'exécution d'expressions et collecte les temps d'exécution. Elle permet de comparer les vitesses de solutions alternatives.
- *Jeu d'essai* :

```
n <- 10000
n2 <- n*2
data1 <- data.frame(id=1:n,a=runif(n), b=rnorm(n))
data2 <- data.frame(id=1:n2,c=runif(n2), d=rnorm(n2))
# --- data.table travaille sur des tables spécialement formatées
data1bis <- data.table::data.table(data1)
data2bis <- data.table::data.table(data2)
# --- mise en base de données (package RSQLite et DBI)
driver <- dbDriver("SQLite")
base <- dbConnect(driver, "truc.sqlite")
copy_to(base,data1,"data1")
copy_to(base,data2,"data2")
# --- les scenarios
f0 <- function() merge(data1,data2,by="id",all.x=TRUE,all.y=FALSE)
f1 <- function() dplyr::left_join(data1,data2, by="id")
f2 <- function() merge(data1bis, data2bis, by="id", all.x=TRUE, all.y=FALSE)
f3 <- function() dbGetQuery(base,"SELECT * FROM data1 LEFT JOIN data2 ON data1.id=data2.id")
```

NB : `merge` est une fonction générique

```
microbenchmark("base"=f0(),"dplyr"=f1(),"data.table"=f2(),"sqlLite"=f3())
```

Unit: milliseconds

expr	min	lq	mean	median	uq	max	neval
base	11.5430	12.52540	14.293097	13.23960	14.62885	84.1831	100
dplyr	3.8764	4.21135	4.951043	4.70685	5.23605	8.9652	100
data.table	3.6683	6.58905	6.866870	6.94160	7.24665	8.0829	100
sqlLite	26.8950	29.16910	30.896952	30.56830	32.61310	36.9469	100

Mesurer les performances (2/2)

Le package *bench*

- La fonction `mark` du package *bench* donne accès à d'autres statistiques que celles sur la vitesse, en particulier elle fournit l'occupation mémoire et le nombre d'appels au « ramasse miettes » (Garbage Collector, GC)

NB : Les différents scénarios doivent restituer exactement le même résultat.

```
bm <- bench::mark("base"=f0(), "dplyr"=f1(),
                  "data.table"=as.data.frame(f2()), "sqlLite"=f3()[, -4])
summary(bm)
# A tibble: 4 x 13
  expression      min  median `itr/sec` mem_alloc `gc/sec` n_itr n_gc total_time
  <bch:expr> <bch:tm> <bch:t>    <dbl> <bch:byt>    <dbl> <int> <dbl>    <bch:tm>
1 base         12.5ms 13.96ms   65.0    2.82MB     4.33     30      2     462ms
2 dplyr         4.57ms  5.04ms  188.     1.55MB     8.56     88      4     467ms
3 data.table    4.13ms  4.94ms  198.     1.04MB     4.13     96      2     484ms
4 sqlLite      33.98ms 37.2ms   27.2   898.47KB     2.09     13      1     479ms
```

Lire un fichier Ecrire un fichier	rio	<pre>df <- import("truc.dbf") export(df, "truc.dbf")</pre>	Formats supportés : csv, dbf, ods, xlsx, (xls) sas7bdat, RDS, Rdata, fst...
Visualiser une table Visualiser sa structure		<pre>View(df) str(df)</pre>	Rstudio : click dans le nom click dans le bouton bleu
Compter les lignes	dplyr	<pre>df1 <- count(mtcars) df1 <- count(mtcars, cyl)</pre>	
Faire une statistique		<pre>df1 <- summarise(mtcars, M = mean(hp), N = n())</pre>	min(hp), max(hp), median(hp) quantile(hp,0.25)
Calculer une colonne		<pre>df1 <- mutate(mtcars, cyl=as.factor(cyl), m=mean(hp))</pre>	as.numeric as.character
Conserver des colonnes Conserver des lignes		<pre>df1 <- select(mtcars, hp, cyl) df1 <- filter(mtcars, hp>100)</pre>	-hp supprime hp == teste l'égalité
Calculer par modalités d'autres colonnes		<pre>df2 <- group_by(df1, mtcars) df3 <- summarise(df2, m=mean(hp)) df2 <- mutate(df2, m=mean(hp)) df2 <- filter(df2, hp>mean(hp))</pre>	
Appeler une fonction	dplyr	<pre>df <- "truc.dbf" %>% import() "truc.dbf" %>% export(df, .)</pre>	

Modalités distinctes	dplyr	<code>df2 <- distinct(mtcars, cyl)</code>	
Renommer des colonnes		<code>df2 <- rename(mtcars, c=cyl)</code>	
Fusionner des tables		<code>dfc <- inner_join(dfa, dfb, by="id"</code>	left_join, anti_join, semi_join
Empiler des tables		<code>dfc <- bind_rows(dfa, dfb)</code>	
Transposer une table	tidyr	<code>df1 <- gather(df0, k, v, -id) df0 <- spread(df1, k, v)</code>	Large vers long Long vers large