

R: Langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

R: Langage

Eric Marcon

25 December 2020

Langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Architecture

R-base

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Fonctions primitives et structures de données de base.

Exemples : fonction `sum` et données de type `matrix`:

```
pryr::otype(sum)
```

```
## Registered S3 method overwritten by 'pryr':  
##   method      from  
##   print.bytes Rcpp
```

```
## [1] "base"
```

```
pryr::otype(matrix(1))
```

```
## [1] "base"
```

Langage orienté objet.

Classes déclaratives.

```
MonPrenom <- "Eric"  
class(MonPrenom) <- "Prenom"
```

S3 - Méthodes

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Les méthodes S3 sont liées aux fonctions, pas aux objets.

```
# Affichage par défaut  
MonPrenom
```

```
## [1] "Eric"  
## attr(,"class")  
## [1] "Prenom"
```

```
print.Prenom <- function(x) cat("Le prénom est", x)  
# Affichage modifié  
MonPrenom
```

```
## Le prénom est Eric
```

S3 - Génériques

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

`print` est une méthode générique (“un générique”) déclaré dans `base`.

```
help(print)  
pryr::otype(print)
```

Son code se résume à une déclaration `UseMethod("print")`:

```
print
```

```
## function (x, ...)  
## UseMethod("print")  
## <bytecode: 0x7fe3e41aaab8>  
## <environment: namespace:base>
```

S3 - print

Il existe beaucoup de méthodes S3 pour print:

```
head(methods("print"))
```

```
## [1] "print.acf"          "print.AES"
## [3] "print.all_vars"     "print.anova"
## [5] "print.ansi_string" "print.ansi_style"
```

Chacune s'applique à une classe. `print.default` est utilisée en dernier ressort et s'appuie sur le type (R de base), pas la classe (S3).

```
typeof(MonPrenom)
```

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

```
pryr::otype(MonPrenom)
```

```
## [1] "S3"
```

S3 - Héritage

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Un objet peut appartenir à plusieurs classes.

```
class(MonPrenom) <- c("PrenomFrancais", "Prenom")  
inherits(MonPrenom, what = "PrenomFrancais")
```

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

```
inherits(MonPrenom, what = "Prenom")
```

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


S3 - Héritage

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Le générique cherche une méthode pour chaque classe, dans l'ordre.

```
print.PrenomFrancais <- function(x) cat("Prénom français:",  
    x)  
MonPrenom
```

```
## Prénom français: Eric
```

S3 - Résumé

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

S3 est le langage courant de R.

Presque tous les packages sont écrits en S3.

Les génériques sont partout mais passent inaperçu:

```
library("entropart")  
.S3methods(class = "SpeciesDistribution")
```

```
## [1] autoplot plot  
## see '?methods' for accessing help and source code
```

```
# help(InternalMethods)
```

S4 structure les classes :

- slots pour les données ;
- constructeur explicite.

```
setClass("Personne", slots = list(Nom = "character",  
  Prenom = "character"))  
Moi <- new("Personne", Nom = "Marcon", Prenom = "Eric")  
pryr::otype(Moi)
```

```
## [1] "S4"
```

S4 - Méthodes

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Les méthodes appartiennent toujours aux fonctions:

```
setMethod("print", signature = "Personne", function(x,  
  ...) {  
  cat("La personne est:", x@Prenom, x@Nom)  
})  
print(Moi)
```

```
## La personne est: Eric Marcon
```

S4 - Résumé

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

S4 est plus rigoureux que S3.

Quelques packages sur CRAN : Matrix, sp, odbc,... et beaucoup sur Bioconductor.

RC a été introduit dans R 2.12 (2010) avec le package *methods*.

Les méthodes appartiennent aux classes, comme en C++.

```
library("methods")
PersonneRC <- setRefClass("PersonneRC",
  fields = list(Nom = "character", Prenom = "character"),
  methods = list(print = function() cat(Prenom, Nom)))
MoiRC <- new("PersonneRC", Nom = "Marcon", Prenom = "Eric")
pryr::otype(MoiRC)
```

```
## [1] "RC"
```

```
MoiRC$print()
```

```
## Eric Marcon
```

R6 perfectionne RC mais n'est pas inclus dans R.

Les attributs et les méthodes peuvent être publics ou privés.

Une méthode `initialize()` est utilisée comme constructeur.

```
library(R6)
PersonneR6 <- R6Class("PersonneR6", public = list(Nom = "character",
  Prenom = "character", initialize = function(Nom = NA,
    Prenom = NA) {
    self$Nom <- Nom
    self$Prenom <- Prenom
  }, print = function() cat(self$Prenom, self$Nom)))
MoiR6 <- PersonneR6$new(Nom = "Marcon", Prenom = "Eric")
MoiR6$print()
```

Eric Marcon

RC et R6 - Résumé

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Très peu utilisés, plutôt considérés comme des exercices de style.

S6 permet de programmer rigoureusement en objet.

Les performances sont inférieures à celles de S3.

Langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Style

Choisir son style de code

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Pas d'obligation mais cohérence à assurer.

Nommage des objets :

- *CamelCase* : les mots sont compactés, les majuscules assurent la lisibilité ;
- *tiret_bas* : les espaces sont remplacés par des `_`, pas de majuscule.

Les points sont interdits : séparateurs des génériques. Hélas : `data.frame`, `t.test()`.

Utiliser des noms clairs, pas de `valeur=12` ou `TarbrInv`.

Affectation

Utiliser impérativement `<-` et réserver `=` aux arguments de fonctions.

```
a <- b <- 12  
a
```

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

```
b
```

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

```
(a <- runif(1)) * (rnorm(1) -> b)
```

```
## [1] 0.2031326
```

Espacement

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Entourer `<-` par des espaces pour éviter la confusion avec `-`.

Respecter l'espacement standard du code autant que possible.
knitr avec l'option `tidy=TRUE` met en forme automatiquement
le code des documents RMarkdown.

Alignement

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Aller à la ligne entre les commandes, éviter ;.

Les accolades commencent sur la ligne de la commande, se terminent sur une ligne seule.

Indenter avec deux espaces. Tabulations interdites.

```
if (a > b) {  
    print("Plus grand")  
} else {  
    print("Plus petit")  
}
```

```
## [1] "Plus petit"
```

Commentaires

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Dans le code, commenter toute ligne non évidente (de l'ordre de 2 sur 3). Commenter le pourquoi, pas le comment sauf extraordinaire.

Le commentaire précède la ligne de code ou va en fin de ligne.

```
# Calcul de la surface
if (is.null(Largeur)) {
  # Longueur seulement: carré
  Longueur^2
} else {
  # Vrai rectangle. Formule de xxx(1920).
  Longueur * Largeur
}
```

Bien choisir la langue des commentaires. Accents interdits dans les packages.

Commentaires

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Commentaires de blocs par :

```
# Première partie ####  
x <- 1  
# Deuxième partie ####  
y <- 2
```

Ces blocs sont repliables dans RStudio (menu *Edit / Folding*).

Appel des fonctions

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Les fonctions du package *base* sont toujours disponibles. Les autres non.

Les packages chargés par défaut peuvent être déchargés: *utils*, *graphics*, *stats*

Bonne pratique :

- usage interactif : taper le nom de la fonction seulement : `fonction()`
- code à conserver : préciser le package (une fonction peut être masquée par un autre package) : `package::fonction()`.

Appel des fonctions

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Principes :

- `library("package")` charge `loadNamespace()` et attache `attachNamespace()` un package
- le package exporte des fonctions : `package::fonction()`
- les fonctions sont accessibles par leur nom `fonction()`
- si `nouveaupackage` exporte `nouveaupackage::fonction()`, la nouvelle fonction masque l'ancienne.

Le Langage

Eric Marcon

Architecture

Style

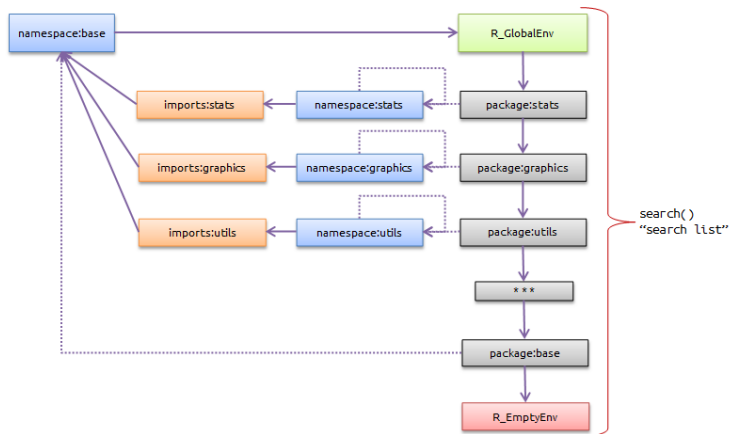
Environnements
et recherche

Eléments du
langage

Vectoriser

Environnements et recherche

Hiérarchie des environnements



Référence

Hiérarchie des environnements

R démarre dans l'environnement vide.

Chaque package chargé crée un environnement fils.

La console se trouve dans l'environnement global, fils du dernier package chargé.

```
search()
```

```
## [1] ".GlobalEnv"           "package:R6"  
## [3] "package:entropart"    "package:devEMF"  
## [5] "package:forcats"      "package:stringr"  
## [7] "package:dplyr"        "package:purrr"  
## [9] "package:readr"        "package:tidyr"  
## [11] "package:tibble"       "package:ggplot2"  
## [13] "package:tidyverse"    "package:formatR"  
## [15] "package:kableExtra"   "package:stats"  
## [17] "package:graphics"     "package:grDevices"  
## [19] "package:utils"        "package:datasets"  
## [21] "package:methods"      "Autoloads"  
## [23] "package:base"
```

Environnements fils de *.GlobalEnv*

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Le code d'une fonction appelée de la console s'exécute dans un environnement fils de *.GlobalEnv*

```
environment()
```

```
## <environment: R_GlobalEnv>
```

```
f <- function() environment()  
f()
```

```
## <environment: 0x7fe3e44520d0>
```

```
parent.env(f())
```

```
## <environment: R_GlobalEnv>
```

Recherche d'un objet

Eric Marcon

Eric Marcon

Architecture

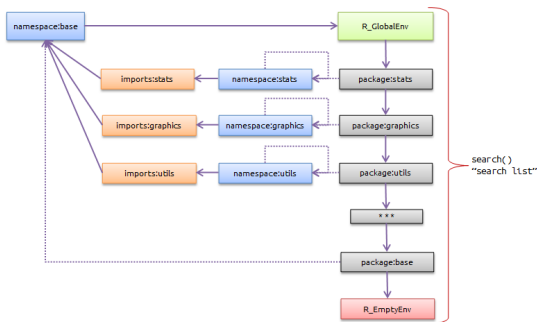
Style

Environnements
et recherche

Éléments du
langage

Vectoriser

La recherche part de l'environnement global (ou de celui d'une fonction appelée) et descend la colonne de droite.



Les packages doivent être attachés pour y être.

Packages chargés

Eric Marcon

Eric Marcon

Architecture

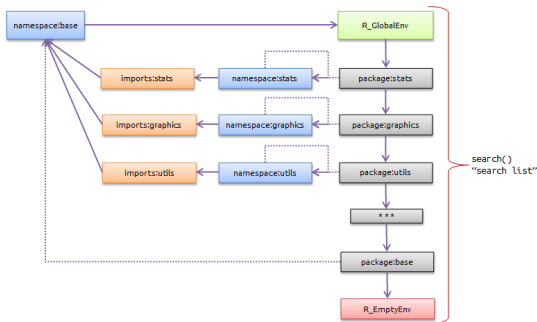
Style

Environnements
 et recherche

Éléments du
 langage

Vectoriser

Un package chargé est dans la colonne centrale: son espace de noms est accessible mais ses objets ne sont pas inclus dans la recherche.



Packages chargés non attachés

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

pryr peut être chargé sans être attaché :

```
unloadNamespace("pryr")  
isNamespaceLoaded("pryr")
```

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

```
loadNamespace("pryr")
```

```
## <environment: namespace:pryr>
```


Packages chargés non attachés

Eric Marcon

Architecture

Style

Environnements
 et recherche

Éléments du
 langage

Vectoriser

```
search()
```

```
## [1] ".GlobalEnv"          "package:R6"
## [3] "package:entropart"    "package:devEMF"
## [5] "package:forcats"      "package:stringr"
## [7] "package:dplyr"        "package:purrr"
## [9] "package:readr"        "package:tidyr"
## [11] "package:tibble"       "package:ggplot2"
## [13] "package:tidyverse"    "package:formatR"
## [15] "package:kableExtra"   "package:stats"
## [17] "package:graphics"     "package:grDevices"
## [19] "package:utils"        "package:datasets"
## [21] "package:methods"      "Autoloads"
## [23] "package:base"
```

```
isNamespaceLoaded("pryr")
```

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

Appel explicite d'un objet

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

La fonction `otype()` ne peut pas être trouvée mais elle peut être utilisée :

```
tryCatch(otype(1), error = function(e) print(e))
```

```
## <simpleError in otype(1): could not find function "otype">
```

```
pryr::otype(1)
```

```
## [1] "base"
```

Chargement implicite

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

`loadNamespace()` n'est jamais utilisé :

- Appeler `package::fonction()` charge le package,
- Attacher un package le charge.

Objets non exportés

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Les objets non exportés par un package sont accessible dans son espace de nom avec trois :

```
package::fonction()
```

Les autres aussi, mais c'est inutile :

```
stats::sd(rnorm(100))
```

```
## [1] 1.017519
```

Objets non exportés

Les méthodes S3 ne sont normalement pas exportées, seul le générique l'est.

```
names(formals(plot))
```

```
## [1] "x" "y" "..."
```

```
tryCatch(names(formals(entropart::plot.SpeciesDistribution)),  
  error = function(e) print(e))
```

```
## <simpleError: 'plot.SpeciesDistribution' is not an exported object from 'entropart'>
```

```
names(formals(entropart::plot.SpeciesDistribution))
```

```
## [1] "x"          "..."  
## [4] "type"       "log"  
## [7] "xlab"       "ylab"
```

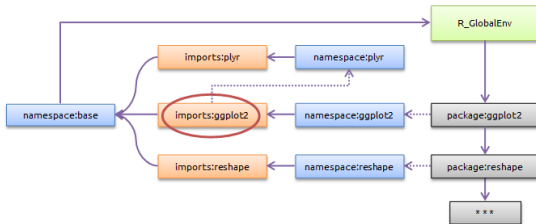
Packages importés

Eric Marcon

Eric Marcon

Les packages s'appuient sur d'autres packages.

Ils peuvent les importer : ggplot2 importe plyr.



Ou en dépendre : ggplot2 dépend de reshape.

Un package qui exporte une méthode S3 dépend forcément du package contenant le générique.

Langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Eléments du langage

Type de données

Langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Réel

```
typeof(1.1)
```

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

Entier : forcer le type en précisant L

```
typeof(1L)
```

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


Type de données

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Logique

```
typeof(TRUE)
```

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

Complexe

```
# help(complex)  
typeof(sqrt(-1 + (0+0i)))
```

```
## [1] "complex"
```

Type de données

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Caractère

```
typeof("Bonjour")
```

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

Brut

```
# help(raw)  
typeof(raw(1))
```

```
## [1] "raw"
```

Test du type de données

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

```
is.character("Bonjour")
```

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

```
is.double(1.2)
```

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

```
is.logical(1 > 0)
```

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

Test du type de données

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Attention à `is.integer()` :

```
is.integer(1)
```

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

```
typeof(1)
```

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

`is.numeric()` est vrai pour les réels et les entiers.

Structures de données

5 structures de données :

Atomique

Récuratif

Unidimensionnel

vector

list

Bidimensionnel

matrix

data.frame

n-dimensionnel

array

`is.atomic()` teste la structure d'une variable.

Vecteurs

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

```
(MonVecteur <- 1:3)
```

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

Tous les éléments sont du même type :

```
c(1, TRUE, "a")
```

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

Matrices

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

```
(MaMatrice <- matrix(1:6, nrow = 2))
```

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

La multiplication matricielle est très performante

```
MaMatrice %*% matrix(1:3, ncol = 1)
```

```
##      [,1]
## [1,]   22
## [2,]   28
```

Tableaux

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
 et recherche

Eléments du
 langage

Vectoriser

```
(MonTableau <- array(1:12, dim = c(2, 3, 2)))
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```


Listes

Langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

```
(MaListe <- list(Premier = 1:2, Deuxieme = "a"))
```

```
## $Premier  
## [1] 1 2  
##  
## $Deuxieme  
## [1] "a"
```

```
identical(MaListe[[2]], MaListe$Deuxieme)
```

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

Data frames

Les types de données sont uniques par colonne.

```
(Mondf <- data.frame(Article = c("Pommes", "Poires"),  
  Prix = c(2, 3)))
```

```
##   Article Prix  
## 1  Pommes    2  
## 2  Poires    3
```

```
identical(Mondf[, 1], Mondf$Article)
```

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

Fonctions

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Elements fondamentaux du langage.

Toute opération repose sur des fonctions y compris + :

```
(`+`)
```

```
## function (e1, e2) .Primitive("+")
```

Fonctions internes et primitives

Langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Les fonctions *primitives* sont écrites en C : ce sont les plus rapides.

Les fonctions *internes* aussi, mais doivent être appelées par un mécanisme spécial, moins efficace :

`cbind`

```
## function (... , deparse.level = 1)
## .Internal(cbind(deparse.level, ...))
## <bytecode: 0x7fe3e46c75c0>
## <environment: namespace:base>
```

Référence

Fonctions standard : *closure*

La majorité des fonctions est écrite en R.

Leur type est *closure* par opposition à *primitive*.

apply

```
## function (X, MARGIN, FUN, ...)  
## {  
##     FUN <- match.fun(FUN)  
##     dl <- length(dim(X))  
##     if (!dl)  
##         stop("dim(X) must have a positive length")  
##     if (is.object(X))  
##         X <- if (dl == 2L)  
##             as.matrix(X)  
##             else as.array(X)  
##     d <- dim(X)  
##     dn <- dimnames(X)  
##     ds <- seq_len(dl)  
##     if (is.character(MARGIN)) {  
##         if (is.null(dnn <- names(dn)))  
##             stop("'X' must have named dimnames")  
##         MARGIN <- match(MARGIN, dnn)
```

Eléments d'une fonction

Eric Marcon

Arguments : passés à la fonction.

```
args(apply)
```

```
## function (X, MARGIN, FUN, ...)  
## NULL
```

Corps : le code de la fonction

```
deparse(body(apply))[1:3]
```

```
## [1] "{"  
## [2] "    FUN <- match.fun(FUN)"  
## [3] "    dl <- length(dim(X))"
```

Eléments d'une fonction

Environnement : l'ensemble des objets déclarés et un pointeur vers l'environnement parent.

```
environment(apply)
```

```
## <environment: namespace:base>
```

```
ls(environment(apply))[1:2]
```

```
## [1] "-"      "-.Date"
```

```
parent.env(environment(apply))
```

```
## <environment: R_GlobalEnv>
```

Environnement d'une fonction

Une fonction est exécutée dans son propre environnement :

```
environment()
```

```
## <environment: R_GlobalEnv>
```

```
f <- function() environment()  
f()
```

```
## <environment: 0x7fe3eba93dc0>
```

Son environnement parent est celui du code qui l'a appelé :

```
parent.env(f())
```

```
## <environment: R_GlobalEnv>
```


Corps d'une fonction

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Code R standard.

```
Surface <- function(Longueur, Largeur) {  
  return(Longueur * Largeur)  
}  
Surface(Longueur = 2, Largeur = 3)
```

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

Retourne un résultat avec `return()` ou la dernière valeur calculée.

Portée des variables

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

```
Volume <- function(Longueur, Largeur) {  
  return(Longueur * Largeur * Hauteur)  
}  
Longueur <- 5  
Hauteur <- 10  
Volume(Longueur = 2, Largeur = 3)
```

```
## [1] 60
```

Variables locales (définies dans l'environnement de la fonction)
: Longueur et Largeur.

Variables manquantes recherchées dans les environnements
parents : Hauteur.

Evaluation tardive (*lazy*) de Hauteur.

Arguments d'une fonction

Nommés. Appel par leur nom ou dans leur ordre :

```
c(Surface(Largeur = 3, Longueur = 2), Surface(Longueur = 2,  
Largeur = 3))
```

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

```
c(Surface(Longueur = 2, 3), Surface(2, 3))
```

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

Et même (mais illisible) :

```
Surface(3, Longueur = 2)
```

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

Bonnes pratiques

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Donner des noms explicites aux arguments. Le premier s'appelle souvent `x` dans les génériques.

Donner autant que possible des valeurs par défaut aux arguments.

```
Surface <- function(Longueur, Largeur = Longueur) {  
  return(Longueur * Largeur)  
}  
Surface(Largeur = 2)
```

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

Bonnes pratiques

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Nommer tous les arguments à partir du deuxième lors de l'appel :

```
x <- runif(10, min = 5, max = 10)
mean(x, na.rm = FALSE)
```

```
## [1] 7.598087
```

Ne jamais abrégier les noms des arguments ou T pour TRUE.

Argument ...

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Les génériques prévoient tous des arguments libres avec ... :

```
names(formals(plot))
```

```
## [1] "x" "y" "..."
```

Les méthodes ont la même signature que les génériques :

```
names(formals(entropart:::plot.SpeciesDistribution))
```

```
## [1] "x" "..." "Distribution"
## [4] "type" "log" "main"
## [7] "xlab" "ylab"
```

Argument ...

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

La méthode `plot` pour la classe `SpeciesDistribution` accepte tous arguments à la place de `...` et les utilise dans une de ses lignes de code :

```
deparse(entropart:::plot.SpeciesDistribution)[15:16]
```

```
## [1] "    graphics::plot(Ns, type = type, log = log, main = main, xlab=  
## [2] "        ylab = ylab, axes = FALSE, ...)"
```

Tous les arguments non reconnus par `plot.SpeciesDistribution` sont passés à `plot()`.

Argument ...

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Les ... ne sont pas réservés aux génériques :

```
f <- function(x) x
g <- function(y, ...) f(...)
g("Rien", x = "Argument x passé à f par g")
```

```
## [1] "Argument x passé à f par g"
```

Mais il faut que tout argument soit reconnu par une fonction :

```
tryCatch(g("Rien", z = 2), error = function(e) print(e))
```

```
## <simpleError in f(...): unused argument (z = 2)>
```


Fonctions opérateurs (*infix functions*)

Les opérateurs de R sont en fait des fonctions:

```
identical(2 + 2, `+`(2, 2))
```

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

Les opérateurs définis par l'utilisateur sont obligatoirement entre % :

```
`%+%` <- function(a, b) paste(a, b)  
"Nouvelle" +% "chaîne"
```

```
## [1] "Nouvelle chaîne"
```

Référence

Le Langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Vectoriser

Fonctions vectorielles

Eric Marcon

Architecture

Style

Environnements

et recherche

Eléments du
langage

Vectoriser

La plupart des fonctions de R sont vectorielles :

```
x1 <- runif(3)
x2 <- runif(3)
sqrt(x1)
```

```
## [1] 0.8462332 0.5122658 0.7404687
```

```
x1 + x2
```

```
## [1] 0.9918549 0.8924083 0.7398485
```

Raisonner en termes de vecteurs plutôt que de scalaires.

Fonctions vectorielles

Eric Marcon

Ecrire des fonctions vectorielles sur leur premier argument :

```
entropart::lnq
```

```
## function (x, q)
## {
##     if (q == 1) {
##         return(log(x))
##     }
##     else {
##         Log <- (x^(1 - q) - 1)/(1 - q)
##         Log[x < 0] <- NA
##         return(Log)
##     }
## }
## <bytecode: 0x7fe3e8bc5990>
## <environment: namespace:entropart>
```

Fonctions de résumé

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

Exceptions à la règle : fonctions d'un vecteur, résultat scalaire.

```
sum(x1)
```

```
## [1] 1.526821
```

Fonctions non vectorielles

`sapply()` applique une fonction à chaque élément d'un vecteur ou d'une liste.

```
x1 <- runif(1000)
identical(sqrt(x1), sapply(x1, FUN = sqrt))
```

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

```
library("microbenchmark")
mb <- microbenchmark(sqrt(x1), sapply(x1, FUN = sqrt),
  lapply(x1, sqrt), vapply(x1, sqrt, FUN.VALUE = 0))
summary(mb)[, c("expr", "median")]
```

```
##              expr      median
## 1              sqrt(x1)  4.8220
## 2      sapply(x1, FUN = sqrt) 384.5185
## 3      lapply(x1, sqrt)  312.9765
## 4 vapply(x1, sqrt, FUN.VALUE = 0) 330.0930
```

Infiniment plus lent qu'une fonction vectorielle.

Exemple : `sqrt(x1)` vs `sapply(x1, sqrt)` vs `lapply(x1, sqrt)` vs `vapply(x1, sqrt, FUN.VALUE = 0)`

Boucles

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
 et recherche

Eléments du
 langage

Vectoriser

Les boucles sont plus rapides !

```
Boucle <- function(x) {
  Racine <- vector("numeric", length = length(x))
  for (i in 1:length(x)) Racine[i] <- sqrt(x[i])
  return(Racine)
}

Vapply <- function(x) vapply(x, FUN = sqrt, 0)
mb <- microbenchmark(Vapply(x1), Boucle(x1))
summary(mb)[, c("expr", "median")]
```

```
##          expr  median
## 1 Vapply(x1) 296.2230
## 2 Boucle(x1)  79.6225
```

Boucles

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
 et recherche

Éléments du
 langage

Vectoriser

Les boucles longues permettent un suivi

```
Boucle <- function(x) {
  pgb <- txtProgressBar(min = 0, max = length(x))
  Racine <- vector("numeric", length = length(x))
  for (i in 1:length(x)) {
    Racine[i] <- sqrt(x[i])
    setTxtProgressBar(pgb, i)
  }
  return(Racine)
}
RacineX <- Boucle(x1)
```

=====

replicate et vectorize

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

`replicate()` répète une instruction.

```
replicate(3, runif(1))
```

```
## [1] 0.8879289 0.6643613 0.4888854
```

est équivalent à `runif(3)`.

`vectorize()` rend vectorielle une fonction qui ne l'est pas par des boucles. Ecrire plutôt les boucles.

Vectoriser un problème

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Données : inventaire d'une parcelle de Paracou, 4 carrés distincts.

Objectif : calculer le nombre d'arbres par espèce, le nombre d'arbres par carré, la biodiversité par carré.

Technique : utiliser les fonctions vectorielles, les fonctions de type `apply`, éventuellement des boucles.

Lecture des données

Installer le package EcoFoG à partir de GitHub

```
devtools::install_github("EcoFoG/EcoFoG")
```

Extraire les données

```
library("EcoFoG")
```

```
## Loading required package: magrittr
```

```
##
```

```
## Attaching package: 'magrittr'
```

```
## The following object is masked from 'package:purrr':
```

```
##
```

```
##      set_names
```

```
## The following object is masked from 'package:tidyr':
```

```
##
```

```
##      extract
```

```
Paracou6 <- Paracou2df(WHERE = "Plot='6' AND CensusYear=2016")
```

Organisation des données

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Création d'un tableau croisé :

```
Paracou6X <- as.data.frame.matrix(xtabs(~paste(Family,
        Genus, Species) + SubPlot, data = Paracou6))
Paracou6X[1:2, ]
```

```
##                1 2 3 4
## Anacardiaceae Anacardium spruceanum 0 1 0 2
## Anacardiaceae Tapirira guianensis  1 2 0 0
```

`as.data.frame.matrix` est la méthode de conversion des matrices en dataframes...

Statistiques marginales

`apply()` applique une fonction aux lignes ou colonnes d'un objet 2D.

`colSums()` et semblables (`colMeans()`, `rowMeans()`) sont optimisées.

```
mb <- microbenchmark(apply(Paracou6X, 2, sum), colSums(Paracou6X))
summary(mb)[, c("expr", "median")]
```

```
##                expr    median
## 1 apply(Paracou6X, 2, sum) 134.0545
## 2      colSums(Paracou6X)  77.6890
```

```
colSums(Paracou6X)
```

```
##      1      2      3      4
## 976 885 948 811
```

Comptage du nombre d'espèces

Eric Marcon

Eric Marcon

Architecture

Style

Environnements
et recherche

Eléments du
langage

Vectoriser

apply() ou préparation des données

```
mb <- microbenchmark(apply(Paracou6X, 2, function(x) x >
  0), colSums(Paracou6X > 0))
summary(mb)[, c("expr", "median")]
```

```
##                               expr    median
## 1 apply(Paracou6X, 2, function(x) x > 0) 203.0935
## 2                               colSums(Paracou6X > 0) 99.9365
```

```
colSums(Paracou6X > 0)
```

```
##    1    2    3    4
## 192 200 199 177
```

Fonctions complexes

Le langage

Eric Marcon

Architecture

Style

Environnements
et recherche

Éléments du
langage

Vectoriser

Estimation de la richesse spécifique avec entropart

```
library("entropart")  
apply(Paracou6X, 2, Richness)
```

```
##      1      2      3      4  
## 358 346 319 295
```

Performance de apply()

Eric Marcon

Comparaison avec une boucle

```
Boucle <- function(Cmnt) {
  Richesse <- vector("numeric", length = ncol(Cmnt))
  for (i in 1:ncol(Cmnt)) Richesse[i] <- Richness(Cmnt[,
    i])
  return(Richesse)
}
Apply <- function(Cmnt) apply(Cmnt, 2, Richness)
mb <- microbenchmark(Boucle(Paracou6X), Apply(Paracou6X))
summary(mb)[, c("expr", "median")]
```

```
##              expr    median
## 1 Boucle(Paracou6X) 15.14594
## 2  Apply(Paracou6X) 15.22656
```

apply() clarifie (vectorise) le traitement mais ne l'accélère pas.