

# 10 raisons pour lesquelles j'adore R en tant que dévelopeur Python

Joseph Barbier<sup>1</sup>

## Résumé

Le langage R possède un grand nombre de particularités dans son fonctionnement qui semblent banales ou sont peu connus, alors qu'elles sont en fait particulièrement agréable à utiliser, notamment du point de vue d'un développeur Python.

Dans cette présentation, j'exposerai une liste de 10 exemples de choses que R permet et qui me font adorer le langage R, et que je souhaite encourager les gens à utiliser davantage.

On parlera de méthodes S3, de la fonction `substitute()`, de l'opérateur pipe, de contrôle de l'état global, de lazy evaluation ou encore de retours invisibles. Pleins de choses qui me paraissaient bizarres en venant de Python, mais qui sont en fait géniales.

**Mots-clefs (3 à 5) :** Python - Syntaxe - Lazy - Environnement - Pipe

## Développement

### Créer ses propres opérateurs

En R, n'importe quelle fonction entourée de backticks et de symboles % devient un opérateur infixé personnalisé. Cette syntaxe permet de créer des DSL (Domain Specific Languages) intuitifs et expressifs.

**Cas d'usage :** Le package `magrittr` utilise `%>%` pour le pipe, `dplyr` propose `%>%` et `%in%`, tandis que `data.table` utilise `%between%` pour tester si une valeur est dans un intervalle. Cette fonctionnalité permet de rendre le code plus lisible en exprimant des opérations métier dans un langage naturel.

```
`%within%` <- function(x, range) {  
  x >= range[1] & x <= range[2]  
}  
  
5 %within% c(1, 10)
```

```
[1] TRUE
```

---

<sup>1</sup>Yellow Sunflower, joseph@ysunflower.com

```
12 %within% c(1, 10)
```

```
[1] FALSE
```

## Surchage d'opérateurs

Le système de méthodes S3 permet de surcharger les opérateurs arithmétiques (+, -, \*, /) en définissant des fonctions nommées `+.ma_classe`. Cela permet de définir un comportement personnalisé pour vos objets tout en conservant une syntaxe naturelle.

**Cas d'usage :** Le package `lubridate` surcharge les opérateurs pour les dates (ajouter des jours, soustraire des heures), `units` gère automatiquement les conversions d'unités physiques, et `vctrs` utilise massivement cette fonctionnalité pour créer des types de vecteurs personnalisés avec des règles de combinaison spécifiques.

```
# Crée un objet avec une unité
weight <- function(x, unit) {
  structure(x, unit = unit, class = "unit_val")
}

convert_to <- function(x, target_unit) {
  if (attr(x, "unit") == target_unit) {
    return(x)
  }

  value <- unclass(x)
  new_value <- switch(
    paste(attr(x, "unit"), target_unit, sep = "->"),
    "g->kg" = value / 1000,
    "kg->g" = value * 1000,
    stop("Conversion non supportée"))
}
weight(new_value, target_unit)
}

# Surchage de l'addition pour unit_val
`+.unit_val` <- function(a, b) {
  if (inherits(b, "unit_val")) {
    # Convert b vers l'unité de a
    b_conv <- convert_to(b, attr(a, "unit"))
    res <- unclass(a) + unclass(b_conv)
    weight(res, attr(a, "unit"))
  } else {
    res <- unclass(a) + b
    weight(res, attr(a, "unit"))
  }
}
```

```
x <- weight(5, "kg")
y <- weight(300, "g")
z <- weight(2, "kg")

x + z
```

```
[1] 7
attr(,"unit")
[1] "kg"
attr(,"class")
[1] "unit_val"
```

```
x + y
```

```
[1] 5.3
attr(,"unit")
[1] "kg"
attr(,"class")
[1] "unit_val"
```

```
x + y + z
```

```
[1] 7.3
attr(,"unit")
[1] "kg"
attr(,"class")
[1] "unit_val"
```

L'opérateur + détecte automatiquement qu'on travaille avec des objets `unit_val` et effectue la conversion nécessaire avant l'addition.

## Retours invisibles

La fonction `invisible()` permet de retourner une valeur qui ne s'affiche pas automatiquement dans la console, mais qui reste assignable. C'est parfait pour les fonctions à effet de bord (sauvegarde de fichier, modification d'environnement) qui retournent leur entrée pour permettre le chaînage.

**Cas d'usage :** Les fonctions `<-`, `print()`, `plot()` retournent toutes invisiblement leur résultat. Le package `ggplot2` utilise cette technique pour que `ggsave()` retourne le graphique sans l'afficher. C'est aussi massivement utilisé dans les packages de validation de données comme `pointblank` pour chaîner les validations tout en conservant les rapports.

```

save_plot <- function(plot, filename) {
  ggsave(filename, plot)
  invisible(plot) # Permet le chaînage sans afficher
}

assign_and_return <- function(name, value) {
  assign(name, value, envir = parent.frame())
  invisible(value)
}

```

```

assign_and_return("other_var", 100)

x <- assign_and_return("my_var", 42)
x

```

```
[1] 42
```

La première ligne ne produit aucune sortie (retour invisible), mais la variable est bien créée. La deuxième assigne la valeur et on peut l'afficher ensuite.

### La fonction substitute()

`substitute()` capture une expression sans l'évaluer, ce qui permet d'inspecter ou de modifier du code avant son exécution. C'est la base de la métaprogrammation en R : on peut analyser ce que l'utilisateur a écrit, le transformer, puis l'évaluer.

**Cas d'usage :** Le package `rlang` utilise massivement `substitute()` et ses variantes pour implémenter la non-standard evaluation. `ggplot2` l'utilise pour les aesthetic mappings (`aes(x = var)`), et `data.table` s'en sert pour permettre la syntaxe `dt[, .(mean_x = mean(x))]` sans guillemets autour des noms de colonnes. C'est aussi la base de `bquote()` pour le templating de code.

```

log_call <- function(expr) {
  call_expr <- substitute(expr)
  result <- log(eval(expr, envir = parent.frame()))
  cat("Expression évaluée: log(", deparse(call_expr), ")\n", sep = "")
  cat("Résultat:", result, "\n")
}

log_call(1 + 3)

```

```

Expression évaluée: log(1 + 3)
Résultat: 1.386294

```

La fonction capture l'expression `1 + 3` sans l'évaluer immédiatement, l'affiche, puis calcule son logarithme.

## Méthodes S3 et génériques

Le système S3 est un système de programmation orientée objet simple et élégant. En créant des méthodes nommées `fonction.classe`, vous définissez le comportement de fonctions génériques (`print`, `summary`, `plot`, etc.) pour vos objets personnalisés. Le dispatch est automatique basé sur la classe de l'objet.

**Cas d'usage :** Presque tous les packages R utilisent S3 ! `lm()` retourne un objet de classe `lm` avec des méthodes `print.lm`, `summary.lm`, `plot.lm`, `predict.lm`. Le package `sf` définit des méthodes géométriques pour les objets géospatiaux, et `tibble` étend les `data.frames` avec des méthodes d'affichage améliorées. C'est beaucoup plus simple que les classes en Python tout en étant très puissant.

```
# Créer une classe personnalisée
individual <- function(name, age) {
  structure(
    list(name = name, age = age),
    class = "individual"
  )
}

# Méthode print pour individual
print.individual <- function(x, ...) {
  cat(sprintf("%s a %d ans\n", x$name, x$age))
  invisible(x)
}

# Méthode summary pour individual
summary.individual <- function(object, ...) {
  cat("Résumé de la personne:\n")
  cat("  Nom:", object$name, "\n")
  cat("  Age:", object$age, "ans\n")
  cat("  Statut:", ifelse(object$age >= 18, "Adulte", "Mineur"), "\n")
}

p <- individual("Alice", 25)
print(p)
```

```
Alice a 25 ans
```

```
summary(p)
```

```
Résumé de la personne:
  Nom: Alice
  Age: 25 ans
  Statut: Adulte
```

R détecte automatiquement la classe de `p` et appelle les bonnes méthodes `print.individual` et `summary.individual`.

## Tout est un vecteur

En R, il n'existe pas de scalaires : tout est un vecteur, même les « nombres simples » sont des vecteurs de longueur 1. Les opérations sont vectorisées par défaut, ce qui permet d'écrire du code concis et performant. Le recyclage automatique étend les vecteurs plus courts pour correspondre aux plus longs.

**Cas d'usage :** Cette philosophie est au cœur de la performance de R. Les packages comme `dplyr` et `data.table` exploitent la vectorisation pour des opérations ultra-rapides sur des colonnes entières. `purrr::map_*` généralise cette idée aux fonctions. En NumPy (Python), il faut explicitement utiliser des array et apprendre les broadcasting rules, alors qu'en R c'est le comportement par défaut.

```
x <- 5
y <- c(1, 2, 3)
y + x
```

```
[1] 6 7 8
```

```
x <- c(-1, 0, 1, 5, -3)
ifelse(x > 0, "positif", "négatif ou nul")
```

```
[1] "négatif ou nul" "négatif ou nul" "positif"      "positif"
[5] "négatif ou nul"
```

Dans le premier exemple, le scalaire 5 est recyclé trois fois pour s'adapter à la longueur de `y`. Dans le second, `ifelse()` applique la condition à chaque élément du vecteur, ce qui serait une boucle en Python.

## Arguments par défaut dépendants

En R, grâce à la lazy evaluation, les arguments par défaut d'une fonction peuvent référencez d'autres arguments de la même fonction. Ces valeurs sont évaluées au moment de l'appel, dans le contexte de la fonction, ce qui permet de créer des valeurs par défaut intelligentes et contextuelles.

**Cas d'usage :** C'est massivement utilisé dans l'écosystème R ! Dans `ggplot2, geom_point(size = 2, stroke = size/2)` définit l'épaisseur du contour en fonction de la taille. Le package `dplyr` permet avec `mutate(x = a + 1, y = x * 2)` de référencer les colonnes créées précédemment. La fonction native `matrix(data, nrow, ncol = length(data)/nrow)` calcule automatiquement le nombre de colonnes. En Python, il faudrait utiliser `None` puis tester et calculer dans le corps de la fonction.

```
# Exemple : créer un intervalle de confiance
create_interval <- function(
  mean,
  sd,
  confidence = 0.95,
  z_score = qnorm((1 + confidence) / 2),
  margin = z_score * sd
) {
  list(
    lower = mean - margin,
    upper = mean + margin,
    confidence = confidence
  )
}

create_interval(mean = 100, sd = 15)
```

```
$lower
[1] 70.60054

$upper
[1] 129.3995

$confidence
[1] 0.95
```

```
create_interval(mean = 100, sd = 15, confidence = 0.99)
```

```
$lower
[1] 61.36256

$upper
[1] 138.6374

$confidence
[1] 0.99
```

Ici, `z_score` dépend de `confidence`, et `margin` dépend à la fois de `z_score` et `sd`. Tout est calculé automatiquement ! En Python, il faudrait écrire :

```
def create_interval(mean, sd, confidence=0.95,
                    z_score=None, margin=None):
    if z_score is None:
        z_score = norm.ppf((1 + confidence) / 2)
    if margin is None:
```

```
margin = z_score * sd  
# ...
```

La version R est beaucoup plus concise et expressive.