Ce que j'adore dans R : 5 fonctionnalités qui séduisent un développeur Python

Joseph Barbier[1]

Résumé

R possède de nombreuses particularités qui peuvent sembler étranges pour un développeur Python, mais qui rendent le langage particulièrement puissant et expressif. Dans cette présentation, je montrerai 5 fonctionnalités que j'apprécie particulièrement et qui peuvent transformer votre manière de coder en R. Vous verrez comment R combine concision, expressivité et flexibilité, souvent avec moins de code qu'en Python.

Vous découvrirez comment créer des opérateurs personnalisés pour rendre votre code plus lisible, surcharger des fonctions pour des objets spécialisés, utiliser la métaprogrammation pour analyser et transformer des expressions, tirer parti du système S3 pour une approche simple de la programmation orientée objet et profiter de la lazy evaluation pour définir des arguments par défaut intelligents et dynamiques.

Mots-clés (3 à 5) : R - Python - S3 - Métaprogrammation - Lazy Evaluation

Développement

1. Créer ses propres opérateurs

En R, toute fonction entourée de backticks et de `%` devient un opérateur infixe. Cette fonctionnalité permet de créer des DSL (Domain Specific Languages) pour exprimer des opérations complexes de manière intuitive. Cela rend le code plus lisible et plus proche du langage naturel, particulièrement utile dans les analyses de données ou les transformations de vecteurs.

```
`%within%` <- function(x, range) x >= range[1] & x <= range[2]

5 %within% c(1, 10)
```

```
[1] TRUE
```

[1]Yellow Sunflower, joseph@ysunflower.com

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

```
[1] FALSE
```

Dans cet exemple, `%within%` teste si une valeur est dans un intervalle, de manière très lisible. C'est beaucoup plus clair que d'écrire la condition `x >= 1 & x <= 10` à chaque fois.

2. Surcharge d'opérateurs

Grâce aux méthodes S3, vous pouvez redéfinir le comportement des opérateurs pour vos objets personnalisés. Cela permet de travailler avec vos types de données de façon naturelle, comme si les objets avaient un comportement « intégré » au langage. C'est particulièrement pratique pour des unités physiques, des dates ou tout autre type spécialisé.

```r
weight <- function(x, unit) structure(x, unit = unit, class = "unit_val")

`+.unit_val` <- function(a, b) {
    if (inherits(b, "unit_val")) {
        b <- weight(unclass(b), attr(a, "unit"))
    }
    weight(unclass(a) + unclass(b), attr(a, "unit"))
}

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

```
[1] 305
attr(,"unit")
[1] "kg"
attr(,"class")
[1] "unit_val"
```

```r
x + z
```

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

```
x + y + z
```

```
[1] 307
attr(,"unit")
[1] "kg"
attr(,"class")
[1] "unit_val"
```

L'opérateur + détecte automatiquement que les objets sont des `unit_val` et effectue la conversion si nécessaire. Cela simplifie énormément le code et évite les erreurs de manipulation.

3. Métaprogrammation avec `substitute()`

La fonction `substitute()` capture une expression sans l'évaluer, ce qui permet d'analyser, modifier ou transformer le code avant son exécution. Cette capacité est à la base de la métaprogrammation en R et est utilisée dans de nombreux packages pour simplifier la syntaxe et rendre les fonctions plus flexibles.

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

log_call(1 + 3)
```

```
Expression évaluée: 1 + 3
Résultat: 1.386294
```

Dans cet exemple, `substitute()` capture l'expression `1 + 3`, permet de l'afficher, puis de calculer son logarithme. C'est très utile lorsque l'on veut enregistrer ou transformer des calculs sans les exécuter immédiatement.

4. Méthodes S3 et fonctions génériques

Le système S3 est un mécanisme simple et puissant de programmation orientée objet. Il permet de définir des méthodes spécifiques pour des classes personnalisées et le dispatch se fait automatiquement selon la classe de l'objet. Cela permet de créer des objets riches sans la complexité des classes en Python ou des systèmes S4 de R.

```r
individual <- function(name, age) {
    structure(list(name = name, age = age), class = "individual")
}

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

p <- individual("Justine", 24)
print(p)
```

```
Justine a 24 ans
```

Ici, `print.individual` est appelée automatiquement lorsque l'on affiche un objet `individual`. Vous pouvez ajouter autant de méthodes que nécessaire pour enrichir vos objets sans modifier la syntaxe de base.

5. Arguments par défaut dépendants

Grâce à la lazy evaluation, les valeurs par défaut des arguments peuvent dépendre d'autres arguments. Cela permet de créer des fonctions plus intelligentes et concises, sans avoir besoin de vérifier ou recalculer manuellement des valeurs intermédiaires comme on le ferait en Python.

```r
# Exemple 1 : dimensions d'un graphique avec ratio automatique
save_plot <- function(filename, width = 800, height = width * 0.618) {
    cat("Dimensions:", width, "x", height, "\n")
}

save_plot("plot.png")
```

```
Dimensions: 800 x 494.4
```

```r
save_plot("plot.png", width = 1200)
```

```
Dimensions: 1200 x 741.6
```

```r
# Exemple 2 : lecture de fichier avec séparateur auto-détecté
read_data <- function(file, sep = if (grepl("\\.csv$", file)) "," else "\t") {
    cat("Fichier:", file, "- Séparateur:", sep, "\\n\n")
}
```

```
read_data("data.csv")
```

```
Fichier: data.csv - Séparateur: , \n
```

```
read_data("data.tsv")
```

```
Fichier: data.tsv - Séparateur:      \n
```

Dans le premier exemple, `height` s'adapte automatiquement à `width` pour respecter le ratio d'or. Dans le second, `sep` est détecté selon l'extension du fichier. En Python, il faudrait utiliser `None` et tester dans le corps de la fonction.