Paper ET05

Metaprogramming in R

Sudhir Kedare, Jazz Pharmaceuticals, Palo Alto, California, USA
Raj Chennamaneni, Jazz Pharmaceuticals, Wilmington, NC, USA

ABSTRACT

Metaprogramming techniques facilitate the generation, modification, and reuse of programming syntax through programmatic statements. Although metaprogramming has its roots in early languages like Lisp, its application in R offers robust capabilities for code manipulation. This paper explores R's distinct metaprogramming features, focusing on fundamental concepts such as defusing and injecting expressions. The analysis demonstrates how these R-specific metaprogramming approaches enhance programming efficiency and code reusability. Through a practical implementation involving dynamic table generation, this paper illustrates how R's metaprogramming capabilities can solve complex programming challenges in data analysis and reporting.

INTRODUCTION

The FDA's Statistical Software Clarifying Statement notes that "FDA does not require the use of any specific software for statistical analyses, and statistical software is not explicitly addressed in Title 21 of the Code of Federal Regulations (e.g., 21CFR part 11)." However, the software used for statistical analyses must be thoroughly documented, including version and build details. This flexibility opens up various options for statistical analysis. While SAS® remains the most commonly used software in the pharmaceutical industry, a significant percentage of FDA staff are familiar with R. This paper explores programming techniques applicable to a broad range of R and R-Shiny applications. Specifically, we examine the concepts of quotation-defusion, unquotation-injection, and the map function, along with a practical example. These metaprogramming techniques are implemented using the {rlang} package in R.

DEFUSION

Defusion, also known as quotation, is a process that captures R code for later use in the program. In the example below, R evaluates the statements immediately, producing a result that is subsequently returned.

Example Code 1:

```
1 + 1
#> [1] 2
```

In defusion, we interrupt the immediate evaluation of an R expression, preventing it from being executed right away. This allows us to capture the expression for later use, enabling its evaluation to be deferred until a more appropriate time in the program.

Example Code 2:

```
expr(1 + 1)
#> 1 + 1
```

The broader concept behind this is that by capturing an expression, we gain the ability to modify it at runtime, facilitating the automation of tasks. However, the expr() function only applies to expressions or functions that are explicitly evaluated by the programmer. As shown below, this approach will not work.

Example Code 3:

```
custom_function <- function(x) {
  expr(x)
}

custom_function(a + b + c)
#> x
```

For custom functions, the enexpr() function should be used instead of expr() function. While expr() function captures expressions more generally, enexpr() function preserves the original context of the expression, ensuring it remains unevaluated and can be manipulated or evaluated later.

Example Code 4:

```
custom_function <- function(x) {
  enexpr(x)
}

custom_function(a + b + c)
#> a + b + c
```

ENVIORNMENT AND DATA VARIABLES

R expressions are typically evaluated in the global environment or within user-defined environments. In the example below, the value of y is evaluated and stored in the global environment, where it can be accessed and used throughout the session.

Example Code 5:

```
y <- 1
y
#> [1] 1
```

DATA MASKING

Data masking refers to the binding of variables between the global environment and user-defined function environments. This concept is automatically implemented in the dplyr package. The following code and results demonstrates how this works.

Example Code 6:

```
factor <- 1000

# Can now use `factor` in computations
mean(mtcars$cyl)

# Defining an env-variable
cyl <- 1000

# Referring to a data-variable
dplyr::summarise(mtcars, mean(cyl))
#> # A tibble: 1 x 1
#>   `mean(cyl)`
#>         <dbl>
#> 1      1000
```

However, variable collisions can occur if the same variable name is used both in the global environment and within a function or dataset. For example, in the code below:

Example Code 7:

```
mpg %>%
  select(manufacturer)
```

The select(manufacturer) call selects the manufacturer column from the mpg dataset. But if a variable of the same name, manufacturer, exists in the global environment, it may cause unintended behavior.

In the case of the custom function grouped_mean, which calculates the mean for a given variable within a specified group:

Example Code 8:

```
grouped_mean <- function(var_group, var_summary) {
  mpg %>%
    group_by(var_group) %>%
    summarize(mean = mean(var_summary))
```

```
}
```

If var_group or var_summary matches a global variable (like manufacturer), it can lead to unintended behavior. To avoid this, one way is to ensure variables within the function are distinct from global variables.

UNQUOTATION

Unquotation is the process of evaluating a previously captured expression, essentially the reverse of quotation. There are two main ways to handle unquotation in R:

DEFUSE AND INJECT SEPARATELY

In this approach, you first capture the argument using enquo(), and then unquote it within the function using !! (bang-bang operator). This allows you to evaluate the argument at the appropriate time.

Example Code 9:

```
my_summarise <- function(data, arg) {
  arg <- enquo(arg)
  data |> dplyr::summarise(mean = mean(!!arg, na.rm = TRUE))
}
```

DEFUSE AND INJECT IN ONE STEP

An alternative is to use the {{ }} syntax, which captures and evaluates the argument in a single step, offering a more streamlined and concise approach.

Example Code 10:

```
my_summarise <- function(data, arg) {
  data |> dplyr::summarise(mean = mean({{ arg }}, na.rm = TRUE))
}
```

Using {{ arg }}, the argument is captured and unquoted in a single operation, streamlining the code and improving readability.

Both unquotation techniques enable you to pass variables or expressions to functions and evaluate them correctly within the function's context, offering flexibility and clarity.

MAP FUNCTION

Typically, to perform repetitive tasks like processing multiple files, a loop can be used. For example, when reading a single file, programmer might use a function like this:

Example Code 11:

```
import_csv <- function(file) {
  data <- read.csv(file, header = TRUE, stringsAsFactors = FALSE)
  return(data)
}
```

But, when dealing with multiple files, the map_df() function from the purrr package provides a more efficient alternative. It allows you to apply a function to each element of a list and combines the results into a single data frame:

Example Code 12:

```
import_csv <- map_df(file_list, extract_data_from_file)
```

This approach eliminates the need for explicit loops, streamlining the code and improving readability, while still performing the same task of reading and processing multiple files.

PRACTICAL EXAMPLE USING METAPROGRAMMING IN R

In clinical trial analysis, descriptive statistics are often calculated to summarize key variables across different treatment groups. The following example demonstrates how Metaprogramming in R can be used to compute these statistics dynamically, based on user-specified variables.

Example Code 13:

```
df |>
  group_by(!!group_var) |>
  summarize(
    n        = sum(!is.na(!!sum_var)),
    mean     = mean(!!sum_var, na.rm = TRUE),
    sd       = sd(!!sum_var, na.rm = TRUE),
    median   = median(!!sum_var, na.rm = TRUE),
    min      = min(!!sum_var, na.rm = TRUE),
    max      = max(!!sum_var, na.rm = TRUE),
    min_max  = NA,
    .groups  = "drop"
  )
```

In above example, metaprogramming concepts like quotation and unquotation enable dynamic evaluation by passing column names as expressions. The group_by(!!group_var) groups the data by the specified group_var, and !!sum_var unquotes the expression to calculate summary statistics for the dynamically specified variable (sum_var). The statistics include count (n), mean, standard deviation (sd), median, minimum (min), and maximum (max), with missing values handled using na.rm = TRUE. The .groups = "drop" removes the grouping structure after computation.

To calculate descriptive statistics for multiple variables (e.g., AGE, BMI) across treatment groups, we use metaprogramming techniques to pass multiple expressions and apply the summary function dynamically:

Example Code 14:

```
adsl_summary <-
  (\(data) purrr::map_df(
    .x = dplyr::vars(AGE, BMI),
    .f = \(x) custom_summary(df = data, group_var = TRTA, sum_var = !!x)
  ))
```

In above example, purrr::map_df() iterates over a list of variables (AGE, BMI), applying the custom_summary function for each. The !! operator unquotes the variable name, enabling dynamic computation of statistics. Leveraging metaprogramming techniques like quotation and unquotation enhances flexibility and efficiency in generating clinical trial analysis outputs.

Metaprogramming automates and generalizes the calculation of descriptive statistics for multiple variables, reducing repetitive code and supporting scalable analysis in clinical trials.

CONCLUSION

In conclusion, Metaprogramming in R significantly improves programming efficiency by enabling dynamic code generation, evaluation, and manipulation. Through techniques like quotation and unquotation, R allows for greater flexibility, reducing redundancy and enhancing scalability. The concepts explored in this paper highlight just a portion of R's metaprogramming capabilities, which can be leveraged to automate complex tasks and integrate seamlessly with evolving automated solutions in data analysis.

REFERENCES

Wickham H. Advanced R: Metaprogramming. https://adv-r.hadley.nz/metaprogramming.html.

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to the IDASP Analytics and Programming team for their invaluable support and assistance in preparing this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name: Sudhir Kedare
Company: Jazz Pharmaceuticals
Address: Palo Alto, CA, USA
Email: Sudhir.Kedare@jazzpharma.com

Author Name: Raj Chennamaneni
Company: Jazz Pharmaceuticals
Address: Wilmington, NC, USA
Email: Rajprakash.Chennamaneni@jazzpharma.com