Basic Coding Syntax and Structure in R

Norman Lo

1/10/2023

1.1 - Basic Mathematical Operations

As a programming language, R is built for basic mathematical operations. We begin our exploration of R from these simple operations.

Arithmetic Operators	Operations	Examples
+	Addition	15 + 5 = 20
-	Subtraction	15 - 5 = 10
*	Multiplication	15 * 5 = 75
/	Division	15 / 5 = 3
%/%	Integer Division	16%/% 3 = 5
^	Exponent	15 ^ 3 = 3375
%%	Modulus	15 %% 5 = 0, 17%% 4 = 1

Here are some coding examples:

```
# Addition
1 + 1 + 10
```

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

```
# Subtraction
10 - 5 - 1
```

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

```
# Multipication
3 * 2 * 4
```

```
## [1] 24
```

```
# Division
10 / 5
```

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

```r
# Exponent, returns the power of one variable against the other
2^4
```

```
## [1] 16
```

```r
# Modulus, returns the remainder after the division
17%%4
```

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

The above examples demonstrates the simple mathematical operations in R. It is worth to note that R is similar to other programming language, which follows the **order of operation**. Here are some examples to demonstrate:

```r
# Multiplication before addition
5 + 6 * 4
```

```
## [1] 29
```

```r
# Parenthese always goes first
(5 + 6) * 4
```

```
## [1] 44
```

```r
# Mixed with the parentheses with other operators
5 + 6 * (4 - 2)
```

```
## [1] 17
```

Note: we have been using white space to separate the operators with the numbers in the previous examples, but it is not necessarily to leave a white space in coding. However, it would be a good practice to improve the readability in your code.

1.2 - Variable

Variable Assignment

In any computer programming language, variable assignment is essential and a fundamental construct. An assignment statement sets and/or re-sets the value stored in the storage location(s) denoted by a variable name; in other words, it copies a value into the variable. R actually provides more flexibility to variable assignment. For instance, R does not require a declaration of a variable before assignment. Almost any R object can be assigned to variable, such as function, statistical summary, plot, or any data type object.

In R, we can use either "<-" or "=" for assignment statement (we suggest using the first assignment operator "<-"). Here are a few examples to demonstrate how to assign value to variable in R:

```r
# Assign 2 to x
x <- 2
x
```

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

```r
# Assign 5 to y
y = 5
y
```

```
## [1] 5
```

```r
# The arrow operator can also be used in reverse direction
3 -> z
z
```

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

```r
# The arrow operator can be used to assign multiple variables at a time
a <- b <- 7
a
```

```
## [1] 7
```

```r
b
```

```
## [1] 7
```

```r
# Sometime we can also use a more complex assignment statement, assign()
assign("j", 4)
j
```

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

Variable name can be any character, number, period(.), and underscore(*) combination, but it cannot start with a number nor underscore(*). Also, it would be good practice to use some meaningful word to name the variable, so the code can be easily interpreted.

Remove Variable

Sometime we need to remove a variable in a project, which we can use the remove() or rm() functions.

```r
# Remove a variable
x <- 2
remove(x)
# x  <- return error message

j <- "Hello"
rm(j)
# j  <- return error message
```

R variable is case sensitive like C and Java, so SQL and Visual Basic users many need to adopt to this different.

```r
# Assign 17 to theVariable
theVariable <- 17
theVariable
```

```
## [1] 17
```

```r
# THEVARIABLE  <- return error message
```

1.3 - Data Types

R has a wide variety of data types, which the basic four major data types are numeric, character, date/POSIXct, and logical. An easy way to check the type of data is using class() function.

```r
# Checking the data type using class()
x <- 2
class(x)
```

```
## [1] "numeric"
```

```r
y <- 2.50
class(y)
```

```
## [1] "numeric"
```

Numeric Data

Numeric data is one of the most popular data type in any analytic study. The most frequently used numeric data in R is **numeric** object, which is similar to float and double in differen programming languages. Numeric includes positive integers, negative integers, decimal points, and zero. Any numeric value assigned to a variable is defined as numeric automatically. We can use is.numeric() function to check if a variable contains numeric data only.

```r
# Check if variable x contains only numeric data only
is.numeric(x)
```

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

Another numeric data in R is integer, which only include integers but not decimal point. When we want to assign an integer to a variable, we need to add the letter "L" at the end of the number. We can use is.integer() function to check if a variable contains integer data only.

```r
# Assign an integer 5 to variable i
i <- 5L
i
```

```
## [1] 5
```

```r
# Check if variable i contains only integer data
is.integer(i)
```

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

```r
# Note that even though the data type in i is defined as integer, it is also numeric by definition
is.numeric(i)
```

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

In some cases, R automatically transform integer to numeric. For instance, when multiply an **integer** by a **numeric** or divide an **integer** by an **integer**. Here are some examples:

```r
# 4L is an integer
class(4L)
```

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

```r
# 2.8 is a numeric
class(2.8)
```

```
## [1] "numeric"
```

```r
# Multiply 4L by 2.8 returns a numeric
class(4L*2.8)
```

```
## [1] "numeric"
```

```r
# 5L is an integer
class(5L)
```

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

```r
# 2L is an integer
class(2L)
```

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

```r
# Divide 5L by 2L returns a numeric
class(5L/2L)
```

```
## [1] "numeric"
```

Character Data

A character object is used to represent string values in R, which is another populr data type in statistical analysis. R provides two ways to process string values: **character** or **factor**. Most people confused about the application of the two, but we are not going to discuss the different of the two in this section.

```r
# Assign a string to variable x
x <- "data"
x
```

```
## [1] "data"
```

```r
# Assign a string to variable y using factor()
y <- factor("data")
y
```

```
## [1] data
## Levels: data
```

Note that the string "data" from variable x is wrapped by the double quotes, but the string "data" from variable y is not. Also, the return value of y includes a second line, the "levels" detail, which will be discussed later when we introduce **vectors**.

character is case sensitive, so "Data", "data, and"DATA" are different string values. The find the length of a character (or numeric), we can use nchar() function.

```r
x <- "data"

# Find the length of the string in x
nchar(x)
```

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

```r
# Find the length of the string "Hello"
nchar("Hello")
```

```
## [1] 5
```

```r
# Find the length of the numeric 3
nchar(3)
```

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

```r
# Find the length of the numeric 452
nchar(452)
```

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

```r
# nchar does not apply to factor object
# nchar(y)  <- return error message
```

Logicals Data

The logical class can only take on two values, TRUE or FALSE. In fact, the logical values of TRUE always have a numeric value of 1, while logical values of FALSE always have a numeric value of 0. Therefore, when we multiply TRUE by 5, it returns a value of 5, vice versa, multiply FALSE by 5, it returns a value of 0.

```r
# Multiply TRUE by 5
TRUE * 5
```

```
## [1] 5
```

```r
# Multiply FALSE by 5
FALSE * 5
```

```
## [1] 0
```

Similar to other data type, we can use class() or is.logical() functions to identify the data type.

```r
# Assign TRUE to variable k
k <- TRUE

# Check the data type using class()
class(k)
```

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

```r
# Check if k contains only logicals data
is.logical(k)
```

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

R also provides **T** and **F** as the short forms for TRUE and FALSE. However, **T** and **F** are just a variable assigned with the logical value TRUE and FALSE, which means we can also assign other values to them as well. Therefore, we do not recommend beginners to use these short forms to avoid confusion in the code. Here is an examples:

```r
# Compare the return value from TRUE and T
TRUE
```

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

```r
T
```

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

```r
# Check the data type of T
class(T)
```

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

```r
# Assign a new value to T
T <- 10
T
```

```
## [1] 10
```

```r
# Check the data type of T again
class(T)
```

```
## [1] "numeric"
```

logicals can be generated by comparing two numeric values (or string values).

```r
# 2 equals to 3?
2 == 3
```

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

```r
# 2 not equals to 3?
2 != 3
```

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

```r
# 2 is smaller than 3?
2 < 3
```

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

```r
# 2 is smaller than or equal to 3?
2 <= 3
```

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

```r
# 2 is greater than 3?
2 > 3
```

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

```r
# 2 is greater than or equal to 3?
2 >= 3
```

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

```r
# "data" is the same as "stats"?
"data"  == "stats"
```

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

```r
# "data" is less than "stats"?
"data" < "stats"
```

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

1.4 - Factor Vector

Vector is a basic data structure in R. It is an ordered collection of elements of the same type. The data type can be logical, integer, double, character, complex or raw. Vector are generally created using the c() function (means "combine"), for instance, c(1, 3, 2, 1, 5) is a vector with numeric elements. Similarly, c("R", "Excel", "SAS", "Python") is a vector with character elements. R does not allow different data type to be assigned into a vector.

Vector is a very important data structure in R. In many years, vector has been used to develop the vectorized functions, which allows mathematical operations can be automatically applied to each of the elements in a vector without looping through it. It could be a new concept to people with other programming language background, but it actually increase the computation efficiency in R significantly.

```
# Create a vector with 10 numeric elements
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
x
```

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

```
# Create a vector with 5 string elements
y <- c("red", "blue", "green", "yellow", "purple")
y
```

```
## [1] "red"    "blue"   "green"  "yellow" "purple"
```

Vectorized Operations

Many operations in R are vectorized, meaning that operations occur in parallel in certain R objects. This allow us to write code that is efficient, concise, and easier to read than in non-vectorized languages. Here are some examples to demonstrate:

```
# Create a vector with 10 numeric elements
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# Multiply each element in the vector by 3
x * 3
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

```
# Add 2 to each element in the vector
x + 2
```

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

```
# Subtract 3 from each element in the vector
x - 3
```

```
##  [1] -2 -1  0  1  2  3  4  5  6  7
```

```r
# Divide each element in the vector by 4
x / 4
```

```
##  [1] 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50
```

```r
# Square each element in the vector
x^2
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```r
# Take square-root of each element in the vector
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

In the previous example, we created a vector with 10 numeric element from 1 to 10. We can also use the **:** operator to create a sequence of numbers in R.

```r
# Sequence from 1 to 10
1:10
```

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

```r
# Sequence from 10 to 1
10:1
```

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

```r
# Sequence from -2 to 3
-2:3
```

```
## [1] -2 -1  0  1  2  3
```

```r
# Sequence from 5 to -7
5:-7
```

```
##  [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5 -6 -7
```

Vectorized operation is widely applicable. Suppose we have two vectors with equal length, the vectorized operation allows each element in a vector to operate with the corresponding element in another vector.

```r
# Create some vectors with the same length
x <- 1:10
y <- -5:4
q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
       "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")

# Check the length of each vector
length(x)
```

```
## [1] 10
```

```
length(y)
```

```
## [1] 10
```

```
# The nchar() function also acts on each element of a vector
nchar(q)
```

```
##  [1]  6  8  8  7  5  8 10  6  7  6
```

```
nchar(y)
```

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

```
# Add two vectors together
x + y
```

```
##  [1] -4 -2  0  2  4  6  8 10 12 14
```

```
# Subtract a vector to another one
x - y
```

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

```
# Multiply two vectors
x * y
```

```
##  [1] -5 -8 -9 -8 -5  0  7 16 27 40
```

```
# Divide a vector by another one
x / y
```

```
##  [1] -0.2 -0.5 -1.0 -2.0 -5.0  Inf  7.0  4.0  3.0  2.5
```

```
# Take the exponential of one vector to the another one
x^y
```

```
##  [1] 1.000000e+00 6.250000e-02 3.703704e-02 6.250000e-02 2.000000e-01
##  [6] 1.000000e+00 7.000000e+00 6.400000e+01 7.290000e+02 1.000000e+04
```

```
# Check the length of the sum of two vectors
length(x+y)
```

```
## [1] 10
```

Note that when we try to perform an operation on two unequal length vectors, the shorter one will be recycled until all the elements in the long one is matched. When the length of the long vector is not a multiple to the length of the short vector, the result will return with a warning message.

```r
# Create a vector with 10 numeric elements
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# Add a short vector to x
x + c(1, 2)
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

```r
# Add a short vector to x (10 is not multiple of 3)
x + c(1, 2, 3)  # warning message will return
```

```
## Warning in x + c(1, 2, 3): longer object length is not a multiple of shorter
## object length
```

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

We can also perform the logical operation with vector.

```r
# Create two vectors with the same length
x <- 1:10
y <- -5:4

# Check of each element in vector x is smaller than or equals to 5
x <= 5
```

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

```r
# Check if the elments in x is less than the corresponding element in y
x < y
```

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

```r
# Check if the element in x is larger than the corresponding elelment in y
x > y
```

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

To return a single logical value for the conditions in the previous vector examples, we can use the all() and any() function.

```r
# Create two vectors with the same length
x <- 1:10
y <- -5:4

# Check if all condition meet
all(x < y)
```

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

```r
# Check if any condition meet
any(x < y)
```

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

Vector elements are accessed using **indexing vectors**, which can be numeric, character, or logical vectors. An individual element of a vector can be assessed by its position or **index**, indicated using square brackets. In R, the first element has an index of 1, which could be very different to other programming language, such as Python starting at 0 position. Here are some examples to demonstrate:

```r
# Vector x
x <- 10:1

# Access the first element in vector x
x[1]
```

```
## [1] 10
```

```r
# Access the first three elements in vector x
x[1:3]
```

```
## [1] 10  9  8
```

```r
# Access the 2nd and 4th elements in vector x
x[c(2,4)]
```

```
## [1] 9 7
```

In R, we can give names to the elements of a vector, which allows us to refer to the elements by name. We can assign names to vector members when we are creating the vector or even after the vector is created.

```r
# Assign names with name values
x <- c(one="a", two="y", last="r")
x
```

```
##  one  two last
##  "a"  "y"  "r"
```

```r
# Assign names with names() function
w <- 1:3
names(w) <- c("a", "b", "c")
w
```

```
## a b c
## 1 2 3
```

Factor Vector

In R, **factors** is an important data object, which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in the columns which have a limited number of unique values, such as "Male", "Female" and TRUE, FALSE etc.They are useful in data analysis for statistical modeling, since categorical variables enter into statistical models differently than continuous (numeric) variables. Here is an example:

```
# Create a character vector
q <- c("Hockey", "Football", "Baseball", "Curling", "Rugby",
       "Lacrosse", "Basketball", "Tennis", "Cricket", "Soccer")

# Create another character vector with some duplicate values plus the original vector q
q2 <- c(q, "Hockey", "Lacrosse", "Hockey", "Water Polo",
        "Hockey", "Lacrosse")

# Transform the character values to factors
q2Factor <- as.factor(q2)
q2Factor
```

```
##  [1] Hockey     Football   Baseball   Curling    Rugby      Lacrosse
##  [7] Basketball Tennis     Cricket    Soccer     Hockey     Lacrosse
## [13] Hockey     Water Polo Hockey     Lacrosse
## 11 Levels: Baseball Basketball Cricket Curling Football Hockey ... Water Polo
```

```
# Transform the q2Factor to numeric data
as.numeric(q2Factor)
```

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

When we execute the above code, q2Factor returns a list of character values and R lists out all the levels (or categories) in the vector. R assign each level (unique element) an integer and use character value to represent these integers. To check the integer value for each level (unique element), we can use as.numeric() function to check.

In some cases, such **nominal variable** that does not have an intrinsic order, the order of the level may not be important to define. However, when the category variable is **ordinal**, for instance, education level: High School, College, Master, Doctorate, we can set the **ordered** argument equals TRUE to modify the levels order.

```
factor(x=c("High School", "College", "Masters", "Doctorate"),
       levels=c("High School", "College", "Masters", "Doctorate"),
       ordered=TRUE)
```

```
## [1] High School College     Masters     Doctorate
## Levels: High School < College < Masters < Doctorate
```

Note: If you still wondering when to use character vector or factor vector, it may be helpful to keep the following conditions in mind:

1. when given a categorical data (either ordinal or nominal), we should always use factor vector.
2. when given a textual file or collection of words, we should stick with character vector.

1.5 - R Basic Functions

In the previous sections, we have demonstrated some basic functions in R, such as nchar(), length(), as.Date(). Function is important and commonly-used in any programming language because it allows us to repeat the same task without writing the same code over and over again. In R, almost every process involve the use of function, so we should at least understand how to use them properly.

We begin with the most basic function in R, mean(). This function will take a vector of numeric values and return the average value. The object inside the parentheses of a function is called "argument".

```
# Assign a vector of numeric values to x
x <- c(2, 4, 6, 8, 10)

# Use mean() function to calcuate the average of x
mean(x)
```

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

If the function requires multiple arguments, the order of the arguments are based on the setting of the function or using the argument names with "=" sign.

```
# Create two numeric vectors
x <- c(2, 4, 6, 8, 10)
y <- c(1, 3, 5, 7, 9)

# Use identical() function to check if the two vectors are exactly equal
identical(x,y)
```

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

```
# Using log() function with default base
log(2)
```

```
## [1] 0.6931472
```

```
# Using log() function with base=5
log(2, base=5)
```

```
## [1] 0.4306766
```

Note: It is often time we write our own function for a repeated task in our program, so it is important to understand the concept of argument.

List of Basic Functions

Function	Explanation	Application
abs()	The absolute value of a numeric object	abs(x)

Function	Explanation	Application
c()	A generic function which combines its arguments	c(x)
identical()	Test if 2 objects are **exactly** equal	identical(x,y)
length()	Return number of elements in a vector	length(x)
mean()	Return the mean values of a vector	mean(x)
plot()	Generic function for plotting of R objects	plot(x)
range()	Return the minimum and maximum	range(x)
sort()	Sort the elements in vector	sort(x)
unique()	Remove duplicate entries from a vector	unique(x)

R Functions Documentation

R has the documentation pages for all R functions, data sets, and other objects, both for packages in the standard R distribution and for contributed packages. To access these documentations, we can use the help operator "**?**". For instance, we can check the documentation for the mathematical operators like "+" and "*" or logical operator "==" by the help operator.

```
# Using the help operator for operational objects
?'+'
```

```
## starting httpd help server ... done
```

```
?'*'
?'=='

# Using the help operator for function
?mean

# Using apropos() to find the function name
apropos("mea")
```

```
##  [1] ".colMeans"          ".rowMeans"          "colMeans"
##  [4] "influence.measures" "kmeans"             "mean"
##  [7] "mean.Date"          "mean.default"       "mean.difftime"
## [10] "mean.POSIXct"       "mean.POSIXlt"       "rowMeans"
## [13] "weighted.mean"
```

1.6 - Missing Values

In data processing, it is often the case that we may have some **missing value** in our data set. In R, there are genearlly two ways to represent the missing values or undefined value, 1) is coded by the logical constant NA and 2) is to code by NULL object. It could be confusing to beginner identifying the different between the two, but we should be careful when using them.

NA

In some statistical softwares, a missing values are presented by either a dash '-', period '.', or a numeric 99. R uses NA to represent a missing value and it is treated as an element in a vector. We can check if a vector contain any NA element by using is.na() function. When NA values is passing into a function such as mean(), it returns NA values because NA value cannot be computed. One way to take care of the issue is to use the argument na.rm=TRUE, which also applies to other statistical functions such as sum(), min(), max(), var(), and sd().

```
# Assign a vector with missing values to z
z <- c(1, 2, NA, 4, 5, NA, 7)
z
```

```
## [1]  1  2 NA  4  5 NA  7
```

```
# Check if vector z contains any NA
is.na(z)
```

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

```
# If we calculate the average value of z, it returns NA value
mean(z)
```

```
## [1] NA
```

```
# We can remove the NA values in the mean() function by using the argument na.rm=TRUE
mean(z, na.rm=TRUE)
```

```
## [1] 3.8
```

We can also represent a missing value in a character vector by NA without any quotation marks.

```
# Create a character vector with NA
zChar <- c("Hockey", NA, "Lacrosse", "Basketball", NA)
zChar
```

```
## [1] "Hockey"     NA           "Lacrosse"   "Basketball" NA
```

```
# Check if vector zChar contains any NA
is.na(zChar)
```

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

NULL

NULL, on the other hand, is a R reserved word meaning non-existing value. Unlike NA, NULL actually means the value does not exist (no missing). NULL is often returned by expressions and functions whose values are undefined. Since NULL represents an non-existing value, it does not exist in a vector. If we assign it to a vector, it will automatically disappear.

```r
# Assign a NULL value to the vector z
z <- c(1, NULL, 3)
z
```

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

```r
# Check the length of vector z
length(z)
```

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

To check if a NULL value exists, we can use the is.null() function. Since NULL cannot be part of a vector, so we should not consider using is.null() function on a vector.

```r
# Assign a NULL value to d
d <- NULL

# Check if d is a NULL value
is.null(d)
```

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

```r
# Check if 7 is a NULL value
is.null(7)
```

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

1.7 - Pipe

In the recent years, users have adopted to a new norm for programming in R by using the principle function provided by the **magrittr** package, or what's called **pipe** operator, %>%. This operator will forward a value, or the result of an expression, into the next function call/expression. We can demonstrate the use of the pipe operator with a simple example below:

```r
# Import the library magrittr
library(magrittr)

# Create a vector x
x <- 1:10

# Using the mean() function to calculate the average of x
mean(x)
```

```
## [1] 5.5
```

```r
# Using pipe operator to calculate the average of x
x %>% mean
```

```
## [1] 5.5
```

Even thought the syntax looks different, but the return object is the same. Pipe operators is explicitly powerful when we need to perform multiple functions, for example, if we want to filter some data, group it by categories, summarize it, and then order the summarized results. Using the basic R code, it may require several lines of code or nest several functions in a very long code, but pipe operator can put together a clean and readable code in one line. Below simple example demonstrates the advantage of using pipe operator:

```
# Create a vector with NAs.
z <- c(1, 2, NA, 4, 5, NA, NA, 7)

# Find the total number of NAs with nested function method
sum(is.na(z))
```

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

```
# Find the total number of NAs with pipe operator
z %>% is.na %>% sum
```

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

```
# Calculate the mean of z using the mean() function
mean(z, na.rm=TRUE)
```

```
## [1] 3.8
```

```
# Calculate the mean of z using the pipe operator
z %>% mean(na.rm=TRUE)
```

```
## [1] 3.8
```

Note: The last example demonstrates how we define the second argument na.rm=TRUE with the mean function using pipe operator.

Pipe operator processes from the left to right, which makes the code easy to understand and increase readability. In fact, pipe is not as computational efficient as the nested functions, but the marginal difference is hardly noticeable. Hadley Wickham, Chief Scientist at RStudio, stated that pipe should not create efficient issue in R program.