R Note

Chih-Tse Liu

21 June, 2024

Table of Contents

Basic R Syntax

Basics

Comments start with a #. Texts are surrounded by quotes **'text'** or double quotes **"text"**. Numbers are recognized directly.

```
# This is a comment.
"Hello World"  # text
5  # number
```

print() is not a must for printing, only when the code is executed within a function or loop. (But it is good practice to specify it.) *cat()* functions as *cout()* in C++, which '\n' is usually necessary.

```
"Hi, my name is Austin."
```

```
## [1] "Hi, my name is Austin."
```

```
100
```

```
## [1] 100
```

```
print("Hi, my name is Austin.")
```

```
## [1] "Hi, my name is Austin."
```

```
cat("Hi, my name is Austin.", 100, "\n")
```

```
## Hi, my name is Austin. 100
```

paste() can concatenate elements and returns a string. ('+' only works for numbers, not for strings!)

```
text1 = paste("Austin", "Good.")
text1
```

```
## [1] "Austin Good."
```



```r
# 'sep' sets the character added between elements, default is a single space
text2 = paste("Austin is ", 20, " years old.", sep = "")
text2
```

```
## [1] "Austin is 20 years old."
```

Variables are created once initialized. '<-' is preferred over '='. Multiple assignments are available.

```r
x <- "Austin"
height <- 40
a <- b <- c <- "happy"
paste(x, "is", a, b, c, "~")
```

```
## [1] "Austin is happy happy happy ~"
```

«- is the **global operator** in R.

```r
var.g <<- 33
```

Assignment operators work both ways.

```r
"Austin" -> y
33 ->> Var
cat(y, Var)
```

```
## Austin 33
```

Variable names can contain characters(**aẑ**), digits(**09**), underscores(_) and periods('). They can only start with characters or '.', while the latter can't be followed directly by a digit. In R, variable names **are case sensitive**.

```r
x    <- 1
x.1  <- 1
x1._ <- 1
._1  <- 1
.x1  <- 1
._.  <- 1
..   <- 1
```


Math

The followings are built-in arithmetic operators.
Addition(+), subtraction(-), multiplication(*), division(/) and exponent(^). The operation is conducted from left to right, with exponent(^) being processed first, multiplication(*) and division(/) the second, and addition(+) and subtraction(-) the last.

```
10 + 6 * 10 - 6 / 10 * 6 ^ 3
```

```
## [1] -59.6
```

The remainder/modulus(%%) and the quotient(%/%) are also built-in methods.

```
10 %% 6
```

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

```
10 %/% 6
```

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

Common mathematical functions.

```
max(1:10)       # returns the maximum number
```

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

```
min(1:10)       # returns the minimum number
```

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

```
sqrt(9)         # returns the square root
```

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

```
sqrt(8.5)       # float input can be calculated as well
```

```
## [1] 2.915476
```



```r
abs(-1)          # returns the absolute value
```

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

```r
ceiling(2.8)     # returns the closest greater integer
```

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

```r
floor(2.8)       # returns the closest smaller integer
```

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

```r
round(2.123, 2) # rounds to some decimal point, default is 0
```

```
## [1] 2.12
```

Conditions

If statements *if()* must be written in brackets. Curly brackets {} are used to define the scope of the code. *else if()*, following *if()*, is for handling multiple conditions, while *else()* catches all other conditions. Nested if statements are legal for R.

```
if (1 == 2) {
  cat("1 is equal to 2.\n")
} else if (1 > 2) {
  cat("1 is greater than 2.\n")
} else {
  if (FALSE) {
    cat("I am right.\n")
  } else {
    cat("Don't be stupid.\n")
  }
}
```

```
## Don't be stupid.
```

```
if (TRUE)
  cat("Curly brackets are unnecessary for one-line statement code.\n")
```

```
## Curly brackets are unnecessary for one-line statement code.
```

While Loops

While loops, *while()*, are used for repeated statements as long as the condition in the parentheses () is satisfied. Curly brackets {} are used to define the scope of the code. The *next* statement starts the next iteration, while the *break* statement is for exiting the loop. Nested while loops are legal in R.

```
i = 1
while (i > 0) {
  print("i is always greater than 0, but this is not an infinite loop.")
  if (i == 3) {
    print(paste("i equals to ", i, " now. Exit loop.", sep=""))
    break
  }
  if (i >= 1) {
    i = i + 1
    print(paste("i equals to ", i, " now.", sep=""))
```



```
    next
  }
}
```

```
## [1] "i is always greater than 0, but this is not an infinite loop."
## [1] "i equals to 2 now."
## [1] "i is always greater than 0, but this is not an infinite loop."
## [1] "i equals to 3 now."
## [1] "i is always greater than 0, but this is not an infinite loop."
## [1] "i equals to 3 now. Exit loop."
```

```
x = 0
while (x == 0)  # curly brackets are unnecessary in a one-line statement
  x = x + 1
```

For Loops

For loops, *for()*, iterates through a sequence. Parentheses, (), are a must. Curly brackets {} are used to define the scope of the code. The *next* statement is used to jump to the next iteration, and the *break* statement is used to exit the loop. Nested for loops are legal in R.

```
for (i in 1:10) {
  if (i == 4)
    next
  else if (i > 4)
    break
  print(paste("i equals to ", i, " now.", sep=""))
}
```

```
## [1] "i equals to 1 now."
## [1] "i equals to 2 now."
## [1] "i equals to 3 now."
```

```
for (x in 1:3)
  print("Curly brackets are unnecessary in a one-line statement.")
```

```
## [1] "Curly brackets are unnecessary in a one-line statement."
## [1] "Curly brackets are unnecessary in a one-line statement."
## [1] "Curly brackets are unnecessary in a one-line statement."
```



```
for (i in 1:3) {
  for (j in 1:3)
    cat(3*(i-1)+j)
  cat("\n")
}
```

```
## 123
## 456
## 789
```

Functions

Functions can be defined by keyword *function()*. Curly brackets {} define the scope of the code. Parameters can be specified in the parentheses (), which those with default values must be placed last. Values in *return()* are returned. **It is inappropriate to rely on implicit return. Always specify *return()* whenever needed**.

```
myfunc = function(x, y=FALSE) {
  print(paste("x equals to ", x, ".", sep=""))
  print(paste("The default value of y is ", y, ".", sep=""))
  if (y != FALSE) {
    print("x, the variable, is a parameter.")
    print(paste(x, ", the value been passed, is an argument.", sep=""))
    print(paste("The current argument of parameter y is ", y, ".", sep=""))
  }
  return(paste("myfunc() returns the value of y, ", y, ".", sep=""))
}
myfunc(100, TRUE)
```

```
## [1] "x equals to 100."
## [1] "The default value of y is TRUE."
## [1] "x, the variable, is a parameter."
## [1] "100, the value been passed, is an argument."
## [1] "The current argument of parameter y is TRUE."

## [1] "myfunc() returns the value of y, TRUE."
```

```
oneLineFun = function(x)
  print(x)
oneLineFun("One-line statement functions without curly brackets are legal.")
```

```
## [1] "One-line statement functions without curly brackets are legal."
```



```r
oneLineFun("But they are mostly useless.")
```

```
## [1] "But they are mostly useless."
```

Data Type

class() is used to check the data type of a variable.
It is legal to change a data type of a variable.

```
var <- 11.11
class(var)
```

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

```
var <- "11.11"
class(var)
```

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

Data types can also be converted via *as.numeric()*, *as.integer()*, *as.character()*, *as.logical()* and *as.complex()*.

Numeric

Numbers are recognized as 'numeric' by default, either with decimals or not.

```
x.num <- 11
class(x.num)
```

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

```
x.num <- 11.11
class(x.num)
```

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

```
as.integer(x.num)
```

```
## [1] 11
```

```
as.character(x.num)
```

```
## [1] "11.11"
```

```
as.logical(x.num)
```

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

```
as.complex(x.num)  # the imaginary part is 0
```

```
## [1] 11.11+0i
```

Integer

Numbers can be specified as 'integer' by following the digits with **'L'**.

```
x.int <- 11L
class(x.int)
```

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

```
as.numeric(x.int)
```

```
## [1] 11
```

```
as.character(x.int)
```

```
## [1] "11"
```

```
as.logical(x.int)
```

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

```
as.complex(x.int)  # the imaginary part is 0
```

```
## [1] 11+0i
```

Character

Single line and Multiline string. Note that indentations are recognized as parts of a multiline string.

```
str <- "string"
str <- "This is a
String."
print(str)    # a '\n' will be placed at each line break
```

```
## [1] "This is a \nString."
```

```
cat(str)      # '\n' won't appear, but there will still be line breaks
```

```
## This is a
## String.
```

Calculate the number of characters in a string. Note that *length()* returns the length of a vector, not the number of characters.

```
str <- "Hello World"
nchar(str)
```

```
## [1] 11
```

```
length(str)
```

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

Check if a string contains a specific character.

```
grepl("H", str)
```

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

```
grepl("A", str)
```

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

Combine two strings.


```
str <- paste("Hello", "World")
```

Escape characters. Note that using *print()* or auto-printing will print out the backslash. Use *cat()* to show the intended string.

```
text <- "I love \"you\""
print(text)
```

```
## [1] "I love \"you\""
```

```
cat(text)
```

```
## I love "you"
```

```
cat("I love \\ you", "\n")  # '\\' a backslash
```

```
## I love \ you
```

```
cat("I love \r you", "\n")  # '\r' covers the previous output
```

```
## I love  you
```

```
cat("I love \t you", "\n")  # '\t' add a tab
```

```
## I love    you
```

```
#cat("I love \b you", "\n")  # '\b' backspace
```

```
x.chr <- "11.11"
class(x.chr)
```

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

```
as.numeric(x.chr)
```

```
## [1] 11.11
```



```r
as.integer(x.chr)
```

```
## [1] 11
```

```r
as.logical(x.chr)  # not TRUE or FALSE
```

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

```r
as.complex(x.chr)
```

```
## [1] 11.11+0i
```

If characters are included, the conversion leads to 'NA'.

```r
x.chr <- "11.11+1i"
as.numeric(x.chr)
```

```
## Warning: NAs introduced by coercion
```

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

```r
as.integer(x.chr)
```

```
## Warning: NAs introduced by coercion
```

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

```r
as.logical(x.chr)
```

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

```r
as.complex(x.chr)      # conversion will work only in this form
```

```
## [1] 11.11+1i
```

```r
x.chr <- "11.11+ 1i"
as.complex(x.chr)      # space is not ignored
```

```
## Warning: NAs introduced by coercion
```

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

Logical

TRUE is stored as '1' and **FALSE** is stored as '0'. For numeric, integer and complex, it is recognized as **TRUE** unless the value is 0. Characters can't be converted into logical.

```
x.log <- TRUE
class(x.log)
```

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

```
as.numeric(x.log)
```

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

```
as.integer(x.log)
```

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

```
as.character(x.log)
```

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

```
as.complex(x.log)  # imaginary part is 0
```

```
## [1] 1+0i
```

TRUE and **FALSE** are returned by comparisons done by comparison operators. Note that multiple comparison (eg. $x == y == z$) is **illegal** in R.

```
1 == 2   # equal
```

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

```
1 != 2   # not equal
```

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



```
1 >  2   # greater than
```

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

```
1 <  2   # less than
```

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

```
1 >= 2   # not less than
```

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

```
1 <= 2   # not greater than
```

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

Logical operators can combine conditional statements. & and | are **element-wise operators**, which they can compare each element in an object (such as vector, array, matrix, etc.). Note that logical comparison are processed **from right to left**.

```
TRUE &  FALSE | FALSE
```

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

```
TRUE |  FALSE & FALSE
```

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

```
TRUE || FALSE           # or
```

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

```
TRUE && FALSE           # and
```

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

```
!1                          # not
```

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

Complex

```
x.cpx <- 1i+2  # 'i' is the imaginary part
class(x.cpx)
```

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

```
x <- 2+1i
class(x.cpx)
```

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

```
as.integer(x.cpx)  # the imaginary part is dropped
```

```
## Warning: imaginary parts discarded in coercion
```

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

```
as.numeric(x.cpx)  # the imaginary part is dropped
```

```
## Warning: imaginary parts discarded in coercion
```

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

```
as.character(x.cpx)
```

```
## [1] "2+1i"
```

```
as.logical(x.cpx)
```

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

Data Structure

```
1:10
```

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

```
1 %in% 1:10
```

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

```
# %*% matrix multiplication
```

Figure

Statistical Analysis

String Manipulation

Important Packages

Misc