PharmaSUG 2018 - Paper HT-07

R 101: The base Package

Arthur Li, City of Hope National Medical Center, Duarte, CA

ABSTRACT

Statistical computing is employed within a diverse range of industries. In recent decades, an open source project, R, has emerged as the preeminent statistical computing platform. With its unsurpassed library of freely available packages, R is capable of addressing almost every statistical inference and data management problem. In this paper, you will learn the most commonly-used functions and operators from the R `base` Package, which serves as the fundamental tools for performing data management.

INTRODUCTION

There are thousands of R packages that exist on CRAN (Comprehensive R Archive Network), and each package consists of a large number of functions to tackle a wide variety of data science challenges. This might be one of the reasons that intrigue a beginner from mastering the language since he or she doesn't know where to start and what the essential components are that they need to know to grasp in R language. Similar to other programming languages, one doesn't need to know all the functionality in a language in order to perform the daily routine work. The essential and critical path to mastering the R language is understanding the differences between different types of object and knowing how to utilize the basic functions and operators from the `base` package to create objects, extract and add elements to an object, and combining objects. Grasping these basic concepts is fundamental for performing data manipulation tasks.

R SYSTEM & INSTALLATION

The R System consists of two conceptual parts: the "base" R system and the contributed packages. You can download the "base" R system from CRAN (www.r-project.org); this is what you want to install R for the first time. The "base" system contains the base package which is required to run R and contains the most fundamental functions. The main installation will install R and a popular set of add-on libraries. Once you stalled R software on your computer, you can then install hundreds of libraries from CRAN or the Bioconductor.

To install a new package from the CRAN, you can use the `install.packages()`function. For example, to install the `Epi` package, type:

```
> install.packages("Epi")
Installing package into 'C:/Users/Arthur/Documents/R/win-library/3.4'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
```

To install a package from the Bioconductor, you need to run the `biocLite.R` R script first by using the `source()` function. Then use the `biocLite()` function to download the package. For example, to install the `preprocessCore` package, you can type the following:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("preprocessCore")
```

In order to use a function from the newly downloaded package, you need to use the `library()` function. For example,

```
> library(Epi)
```

RStudio® is an integrated development environment (IDE) for R. RStudio is available in open source and commercial editions and runs on the desktop. It has better user-friendly interface. To run RStudio, you need to download R from CRAN first, then download RStudio from www.rstudio.com.

AN OVERVIEW OF R LANGUAGE

EXPRESSIONS AND ASSIGNMENTS

Of the many functions R can accomplish, it can serve as a simple calculator. For example, you can start type the command for calculation after the prompt (>).

```
> log(2)+exp(5)
[1] 149.1063
```

Commands in R are either expressions or assignments. Commands are separated by either a semi-colon or a new line. From the previous example, you can see expression command is evaluated and (normally) printed**.** You can assign the computed value to a variable by using `<-` or = operator. When typing the name of an object, R will print this object or a short summary of the object. This is refers as auto-printing. Alternatively, you can use the `print()` function explicitly to print an object.

```
> a <- 5*pi
> a
[1] 15.70796
> print(a)
[1] 15.70796
```

An R object name or variable name can be made up from upper and/or lower case letters, digits (0 to 9) in any non-initial position, periods and underscores. Object names cannot start with an underscore; and it's case sensitive.

VECTORIZED ARITHMETIC

A data vector is an array of numbers that can be constructed by using the `c()` function. For example, to construct vectors storing the weights and heights for a group of people, you can type the following command:

```
> weight <- c(60, 72, 57, 90, 95, 72)
> weight
[1] 60 72 57 90 95 72
> height <- c(1.75, 1.80, 1.65, 1.90, 1.74, 1.91)
```

All the computations performed between vectors is operated element-by-element between vectors. This is referred to as vectorized arithmetic in R. You can perform the calculations with vectors of the same or different length. For example,

```
> bmi <- weight/height^2
> bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```

When you perform the calculations with vectors of different lengths, the shorter vector is recycled. Generally, when the computation is operated between vectors of different lengths, the length of longer vector is multiple of the length of the shorter vector. In the example above, the number 2 is a vector of length 1, and both lengths of weight and height are multiple of 1. In the situation where the longer vector is not a multiple of the shorter vector, R will issue a warning message: `longer object length is not a multiple of shorter object length`.

To calculate the mean of the `height`, you can use the divide the sum of `height` by the length of `height`, For example:

```
> sum(height)
[1] 10.75
> length(height)
[1] 6
> sum(height)/length(height)
[1] 1.791667
```

Alternatively you can simply use the `mean()` function accomplish the same task.

```
> mean(height)
[1] 1.791667
```

GETTING HELP

The documentation of any R functions can be obtained by using the `help()` function or `?` operator. For example, to see the description of the `var` function, you can type the following:

```
> help(var)
> ?var
```

To see the documentation of operators, such as +, -, [, [[, you need to enclosed them the operator in `` (tick marks), single or double quotes. For example:

```
> help(`+`)
```

Sometimes we know the name of the function but we are not sure about the names of its argument, we can use the the `args()` function to find out the names of a function's argument. For example:

```
> args(var)
function (x, y = NULL, na.rm = FALSE, use)
NULL
```

If we do not know the name of the function, we can use the `help.search()` function to perform searching which is similar to doing Google search. For example, `help.search("linear models")`.

AN OVERVIEW OF R OBJECTS

OBJECT STORAGE MODE AND CLASS

Everything within the R language is an object. R data objects can be organized by their dimensionality or by their storage type. The object that is linear or having one dimension is a vector or a list. The object having two dimensions is a matrix or a data frame. The only object having more than two dimensions or N dimensions is an array. Objects can also be categorized by their storage mode. For example, the vector, matrix, and array objects can only contain one data type. The list and the data frame can contain mixed data type.

The nature of an R object can be described by its own attributes. The number of attributes for each object varies. When managing objects, it is critical and important to distinguish different R objects and to understand the differences between each object.

The storage mode and class describe the characteristics of the objects. You can find the storage mode of each object by using the `typeof()` function. For example,

```
> n1 <- c(1, 4, pi, 10)
> n1
[1] 1.000000 4.000000 3.141593 10.000000
> typeof(n1)
[1] "double"
```

The most commonly encountered storage modes for an object are numeric (integer or double), character, and logical. The class attribute describes the data structure of an object. You can find the class of each object by using the `class()` function. For example, the following command creates a matrix, `n2` from `n1`, by using the `matrix()` function:

```
> n2 <- matrix(n1, nrow=2)
> n2
     [,1]     [,2]
[1,]    1 3.141593
[2,]    4 10.000000
> typeof(n2)
[1] "double"
> class(n2)
[1] "matrix"
> class(n1)
[1] "numeric"
```

Notice that the mode of `n2` is still double. The difference between `n1` and `n2` is distinguished by their class; `n1` is a one-dimensional vector, whereas `n2` is a matrix.

The following example creates a list by using the `list()` function. For example:

```
> l1 <- list(n = c(1,3), ch = "a")
> l1
$n
[1] 1 3
```

```
$ch
[1] "a"
> typeof(l1)
[1] "list"
> class(l1)
[1] "list"
```

A data frame is a special case of the list object. To create a data frame, you can use the `data.frame()` function. For example:

```
> d1 <- data.frame(n = 1:26, L = LETTERS)
> typeof(d1)
[1] "list"
> class(d1)
[1] "data.frame"
```

Note: `LETTERS` and `letters` are an R constant vectors that contain upper case letters from A to Z or lower case letters from a to z.

DISPLAYING OBJECTS

An R object can be very large. Instead of printing the entire object, you can use the `str()` function to display the internal structure of an R object compactly. For example:

```
> str(d1)
'data.frame': 26 obs. of 2 variables:
$ n: int 1 2 3 4 5 6 7 8 9 10 ...
$ L: Factor w/ 26 levels "A","B","C","D",..: 1 2 3 4 5 6 7 8 9 10 ...
```

The `head()` or `tail()` function can be used to display the first or last parts of an object. For example:

```
> head(d1)
  n L
1 1 A
2 2 B
3 3 C
4 4 D
5 5 E
6 6 F
```

SESSION MANAGEMENT

WORKING DIRECTORY

The working directory is the directory in which R will, by default, look for and save files. To know the current working directly, you can type:

```
> getwd()
[1] "C:/Users/Arthur/Documents/SAS Talk/Talks/R 101"
```

To see the contents of the working directory, you can use the `dir()` function. For example:

```
> dir()
[1] "HT_07 R 101 The base Package_Copyright.pdf" "R 101.doc"  "R 101.pptx"
```

The default choice of a working directory, usually an R installation directory, is not a good choice for storing data. You can change the directory by using the `setwd()` function. For example:

```
> setwd("C:/Users/Arthur/Documents")
```

WORKSPACE

The workspace is the collection of R objects that are listed upon typing ls() or objects(). All objects created in R are stored in a common workspace.

```
> ls()
[1] "a"      "bmi"    "d1"     "height" "l1"     "n1"     "n2"     "weight"
```

If you decide to remove some of the objects from the workspace, for example, `height` and `weight`, you can use the `rm()` function.

```
> rm(height, weight)
> ls()
[1] "a"   "bmi" "d1"  "l1"  "n1"  "n2"
```

To remove or delete all the objects, you can type `rm(list=ls())`.

SAVING OBJECTS

You can save all the objects in your workspace to a physical file at any time by using the `save.image()` function. For example:

```
> save.image("lect1.RData")
```

If you want to save only a few selected objects, you can use the `save()` function. For example, to save a and `bmi` objects, you can type:

```
> save(a, bmi, file="example.RData")
```

The saved file can also be loaded to your current workspace by using the `load()` function. For example:

```
> load("lect1.RData")
```

SCRIPT FILE

All the R commands we've seen so far are typed after the R prompt (after > symbol). Most of time, we write the R code in a script file. To create a script file, click on File → New Script. Then you can type all your R commands in the script file. You can run a segment of your script file by highlighting the R codes and clicking on the Run icon. To save your script file, click on File → Save or File → Save as ... and use *.R as the file extension.

VECTORS

TYPES OF VECTORS

The three common properties for vector object are type, length, and attributes. The most common type of vectors are integer, double, logical, or character. And a vector cannot contain mixed data type. By default, numbers in R are stored in double precision real numbers. The simplest way to create a vector is to use the `c()` function, and the missing value is represented as NA, which is used for all data type.

```
> mydata <- c(2.9, 3.5, 4.5, NA, 3, 2.4)
> mydata
[1] 2.9 3.5 4.5 NA 3.0 2.4
> typeof(mydata)
[1] "double"
> length(mydata)
[1] 6
> class(mydata)
[1] "numeric"
```

If you want to create an integer vector explicitly, you need to specify the `L` suffix. For example,

```
> int <- c(1L, 3L, 10L)
> int
[1] 1 3 10
> typeof(int)
[1] "integer"
> class(int)
[1] "integer"
```

Character strings can be entered with either double or single quotes, For example,

```
> colors <- c("red", "green", "blue", "yellow", NA, "purple")
> colors
[1] "red" "green" "blue" "yellow" NA "purple"
> typeof(colors)
[1] "character"
```

Logical values are presented as TRUE or FALSE and can be entered as TRUE or FALSE, or simply T or F.

```
> newLogic <- c(TRUE, NA, T, F)
> newLogic
[1] TRUE NA TRUE FALSE
> typeof(newLogic)
[1] "logical"
```

OBJECTS ATTRIBUTES
You can add names to a vector by using the `names()` function. For example:

```
> names(mydata) = letters[1:6]
> mydata
  a   b   c  d   e   f
2.9 3.5 4.5 NA 3.0 2.4
> names(mydata)
[1] "a" "b" "c" "d" "e" "f"
```

Here's an alternative way to create a vector and adding names at the same time:

```
> vec1 <- c(first=1, second=pi, third = sqrt(2))
> vec1
   first   second    third
1.000000 3.141593 1.414214
```

If a vector has names, then this vector has the names attribute. An object can have more than one attributes and different object has different attributes. When we are not sure what attributes an object has, we can use the `attributes()` function to find out.

```
> attributes(vec1)
$names
[1] "first" "second" "third"
```

CONCATENATING VECTORS
To add a or more components to a vector, we can use the `c()` function. For example:

```
> newdata <- c(mydata, 3.6)
> newdata
  a   b   c  d   e   f
2.9 3.5 4.5 NA 3.0 2.4 3.6
> newdata2 <- c(mydata, newdata)
> newdata2
  a   b   c  d   e   f   a   b   c  d   e   f
2.9 3.5 4.5 NA 3.0 2.4 2.9 3.5 4.5 NA 3.0 2.4 3.6
```

GENERATING NUMERIC VECTORS
Instead of using the `c()` function, you can generate a sequence of number by using the `seq()` Function or colon (:) operator. For example:

```
> 2:15
[1] 2 3 4 5 6 7 8 9 10 11 12 13 14 15
> 10:1
[1] 10 9 8 7 6 5 4 3 2 1
```

The more common way to generate a sequence of numbers is to use the `seq()` function The first two arguments, `from` and `to`, specify the beginning and the end of the sequence. For example:

```
> seq(2, 10)
[1] 2 3 4 5 6 7 8 9 10
```

The third and fourth arguments, `by` and `length.out`, are used to specify a step size and the length of the sequence. These two arguments cannot be used at the same time. If the `by` argument is not given, the default is 1. The following two commands generate the same sequence.

```
> seq(2, 20, by=3)
[1] 2 5 8 11 14 17 20
> seq(2, 20, length.out=7)
[1] 2 5 8 11 14 17 20
```

Another useful function to generate a numeric vector is to use the `rep()` function. The `rep()` function is used to repeat an object in different ways. The most common arguments for this function are `x` (a vector) and `times` (number of times to repeat). If the `times` argument consists of a single integer, the result will consist of the whole vector `x` this number of times. If `times` is a vector of the same length of `x`, the result will consist of `x[1]` repeated `times[1]` times, `x[2]` repeated `times[2]` times and so on. The following examples repeats 2 four times and repeats the `x` vector twice.

```
> i <- rep(2, 4)
> i
[1] 2 2 2 2
> x <- 1:4
> y <- rep(x, 2)
> y
[1] 1 2 3 4 1 2 3 4
```

The next example repeats each element in `x` vector twice:

```
> z <- rep(x, i)
> z
[1] 1 1 2 2 3 3 4 4
```

Since `x` is a vector from 1 to 4, the statement below repeats the first element in `x` once, the second element twice, and so on.

```
> w <- rep(x, x)
> w
[1] 1 2 2 3 3 3 4 4 4 4
```

GENERATING LOGICAL VECTORS

Most of the time, logical vectors are created by specifying a condition which can be constructed by using a relational operator. The relational operators are <, <=, >, >=, == (exact equality), and != (exact inequality). For example:

```
> a <- c(seq(2, 4), NA)
> a
[1] 2 3 4 NA
> b1 <- a > 2
> b1
[1] FALSE TRUE TRUE NA
```

Notice that operations, such as by using a relational operator, on missing values result in missing values. To check whether a given vector contains missing values, you need to use the `is.na()` function. For example,

```
> is.na(a)
[1] FALSE FALSE FALSE TRUE
```

The command `a == NA` is not the same as using the `is.na()` function to check missing values. Since operations by using a relational operator on a missing value (`NA`) results a missing value, `a == NA` generates a vector of the same length as the vector `a`, whose values are all `NA`.

```
> a == NA
[1] NA NA NA NA
```

When using == and `!=` operators and the arguments have a length that is greater than 1, we will not get a single TRUE/FALSE value. In order to obtain a single TRUE/FALSE result from a comparison, you need to use the `identical()` function. For example:

```
> identical (3, c(3,4))
[1] FALSE
> 3 == c(3,4)
[1] TRUE FALSE
```

A logical vector can also be created by combining conditions or negate a condition by using logical operators such as `&`, `|`, and `!`. For example,

```
> !is.na(a)
[1]  TRUE  TRUE  TRUE FALSE
> a>2 & !is.na(a)
[1] FALSE  TRUE  TRUE FALSE
```

The `is.element()` function, or the `%in%` operator also generate a logical vector indicating whether any elements in its first argument is in the second argument. The length of the resulting logical vector is the same as the function's first argument. For example:

```
> is.element(c("a", "c", "d", "f"), c("a", "d"))
[1]  TRUE FALSE  TRUE FALSE
> c("a", "c", "d", "f") %in% c("a", "d")
[1]  TRUE FALSE  TRUE FALSE
```

A logical vector is often used for calculation purposes. During the computation, the logical vectors are coerced into numeric vectors. In this case, the TRUE value becomes 1 and FALSE becomes 0. The following example counts the number of elements in a given vector greater than its mean. It doesn't seem obvious to identify the logical vector in the following example. The one that is being used is `g > mean (g)`, which is enclosed within the `sum()` function.

```
> g <- c(seq(1,6, by=0.5), 10)
> g
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 10.0
> sum(g > mean(g))
[1] 5
```

FACTORS

R has a factor class to store categorical data. A factor is created on top of integer vector with two additional attributes: `class` and `levels`. To create a factor, you can use the `factor()` function. For example,

```
> countyVector <- c("la", "sb", "la", "oc", "oc", "sb")
> countyVector
[1] "la" "sb" "la" "oc" "oc" "sb"
> county <- factor (countyVector)
> county
[1] la sb la oc oc sb
Levels: la oc sb
> attributes(county)
$levels
[1] "la" "oc" "sb"
$class
[1] "factor"
> typeof(county)
[1] "integer"
```

```
> class(county)
[1] "factor"
> levels(county)
[1] "la" "oc" "sb"
```

MATRICES AND ARRAYS

CREATING A MATRIX

A vector becomes an array when we add a `dim` attribute. The storage mode of an array is simply the storage mode of its element. A matrix is a special case of an array with two dimensions. We can create a matrix by giving a vector a `dim` attribute. For example,

```
> mydata
  a   b   c  d   e   f
2.9 3.5 4.5 NA 3.0 2.4
> dim(mydata) <- c(2,3)
> mydata
     [,1] [,2] [,3]
[1,]  2.9  4.5  3.0
[2,]  3.5   NA  2.4
```

Another way to create a matrix is to use the `matrix()` function:

```
> mydata1 <- matrix(mydata, nrow=2, ncol=3)
> mydata1
     [,1] [,2] [,3]
[1,]  2.9  4.5  3.0
[2,]  3.5   NA  2.4
```

By default, the matrix is filled downward via columns rather than across via rows. To fill the matrix by row, you can use the `byrow` argument.

```
> mydata2 <- matrix(mydata, 2, 3, byrow = TRUE)
> mydata2
     [,1] [,2] [,3]
[1,]  2.9  3.5  4.5
[2,]   NA  3.0  2.4
```

To transpose a matrix, you can use the `t()` function:

```
> t(mydata)
     [,1] [,2]
[1,]  2.9  3.5
[2,]  4.5   NA
[3,]  3.0  2.4
```

COMBINING MATRICES

The `cbind()` function combines matrices by columns, and the `rbind()` function combines matrices by rows. For example:

```
> newdata1 <- matrix(1:4, 2)
> newdata1
    [,1] [,2]
[1,]   1   3
[2,]   2   4
> newdata2 <- matrix(5:10, 2)
> newdata2
    [,1] [,2] [,3]
[1,]   5   7    9
[2,]   6   8   10
```

```
> cbind(mydata, newdata1)
    [,1] [,2] [,3] [,4] [,5]
[1,] 2.9  4.5  3.0    1   3
[2,] 3.5   NA  2.4    2   4
> rbind(mydata, newdata2)
    [,1] [,2] [,3]
[1,] 2.9  4.5  3.0
[2,] 3.5  NA   2.4
[3,] 5.0  7.0  9.0
[4,] 6.0  8.0 10.0
```

MATRIX ATTRIBUTES

Once `mydata` has changed to a matrix, its `names` attribute has been removed. The `names` attribute generalizes to `rownames` and `colnames` attributes in a matrix. You can use `rownames()` and `colnames()` functions to assign and extract column and row names of a matrix.

```
> rownames(mydata) <- c("r1", "r2")
> colnames(mydata) <- c("cl", "c2", "c3")
> mydata
    cl  c2  c3
r1 2.9 4.5 3.0
r2 3.5  NA 2.4
> rownames(mydata)
[1] "r1" "r2"
> colnames(mydata)
[1] "cl" "c2" "c3"
```

You can use the `nrow()` and `ncol()` function to access the number of rows and columns respectively. The `dim()` function can be used to find the dimensions of a matrix.

```
> nrow(mydata)
[1] 2
> ncol(mydata)
[1] 3
> attributes(mydata)
$dim
[1] 2 3
$dimnames
$dimnames[[1]]
[1] "r1" "r2"
$dimnames[[2]]
[1] "cl" "c2" "c3"
```

LISTS

CREATING A LIST

A list can contain multiple R objects with different storage modes and classes. You can create a list by using the `list()` function and assign each component with a name. For example,

```
> student <- list(name = "John", year = 2, classTaken = c("PM510", "PM511A",
+ "PM511B"), GPA = 3.85)
> student
$name
[1] "John"
$year
[1] 2
$classTaken
[1] "PM510" "PM511A" "PM511B"
$GPA
[1] 3.85
```

Notice that the elements of a list do not have to be of the same length and storage mode. A list can contain a list as well. For example:

```
> nestedList <- list(c1 = 1, letters, list(c1 = 2,c2 = LETTERS))
> str(nestedList)
List of 3
$ c1: num 1
$ : chr [1:26] "a" "b" "c" "d" ...
$ :List of 2
..$ c1: num 2
..$ c2: chr [1:26] "A" "B" "C" "D" ...
```

LIST ATTRIBUTES
Since a list is linear, which is the same as a vector, it can have a `names` attribute.

```
> names(student)
 [1] "name" "year" "classTaken" "GPA"
> attributes(student)
$names
[1] "name" "year" "classTaken" "GPA"
> length(student)
[1] 4
```

Suppose we would like to change the names of `student` to letters a to d. We can use the `names` function:

```
> names(student) <- letters[1:4]
> student
$a
[1] "John"
$b
[1] 2
$c
[1] "PM510" "PM511A" "PM511B"
$d
[1] 3.85
```

CONCATENATING LISTS
We can use the `c()` function to concatenate lists. For example:

```
> list1 <- c(list(letters[1:3], 2:4), list(c(1,3.5)))
> str(list1)
List of 3
$ : chr [1:3] "a" "b" "c"
$ : int [1:3] 2 3 4
$ : num [1:2] 1 3.5
```

DATA FRAMES

CREATING A DATA FRAME FROM EXISTING VECTORS
A data frame is used to store a data matrix, which consists of a list of variables of the same length, but possibly of different types. Data frames are normally imported by reading a text file or from a spreadsheet. We can also create a data frame from pre-existing vectors. For example:

```
> sex <- c("M", "F", "F", "M", "M")
> height <- c(65, 63, 60, 62, 57)
> weight <- c(150, 140, 135, 165, 175)
> liveOnCampus <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
> d <- data.frame(sex, height, weight, liveOnCampus)
> d
  sex height weight liveOnCampus
1   M     65    150         TRUE
2   F     63    140         TRUE
3   F     60    135        FALSE
4   M     62    165        FALSE
5   M     57    175        FALSE
```

All the character columns are converted to factors unless their names are enclosed in the I() function.

```
> d1 <- data.frame(I(sex), height, weight, liveOnCampus)
```

DATA FRAME ATTRIBUTES

A data frame is a special case of the list that contains a list of equal-length vectors. Since a data frame has a two-dimensional structure, it shares the common properties for both a matrix and a list.

```
> typeof(d)
[1] "list"
> class(d)
[1] "data.frame"
> attributes(d)
$names
[1] "sex" "height" "weight" "liveOnCampus"
$row.names
[1] 1 2 3 4 5
$class
[1] "data.frame"
```

Instead of using the default row names generated by R, we can assign new row names to the data frame. For example, ID can be used as a meaningful row name.

```
> id <- c(2345, 1236, 2986, 6543, 6544)
> rownames(d) <- id
> d
     sex height weight liveOnCampus
2345   M     65    150         TRUE
1236   F     63    140         TRUE
2986   F     60    135        FALSE
6543   M     62    165        FALSE
6544   M     57    175        FALSE
```

To extract the column names of a data frame, you can use either the `colnames()` or `names()` function. To extract the row names, you can use the `rownames()` function.

```
> colnames(d)
[1] "sex" "height" "weight" "liveOnCampus"
> names(d)
[1] "sex" "height" "weight" "liveOnCampus"
> rownames(d)
[1] "2345" "1236" "2986" "6543" "6544"
```

The length of a data frame equals to the number of columns; it is also the length of the underlying list. Since a data frame has the same property of a matrix, we can use `ncol()` and `nrow()` to access the number of columns and rows respectively.

```
> length(d)
[1] 4
> ncol(d)
[1] 4
> nrow(d)
[1] 5
```

COMBINING DATA FRAMES

Similar to combining matrices, we can use the `cbind()` function to combine data frames column-wise. The number of rows must be the same. If the row names of the data frame are different, R will use the row names of the first data frame in the `cbind()` function.

```
> d2 <- data.frame(midterm=c(80, NA, 90, 40, 95), final=c(88, 100, 94, 88, 99))
> rownames(d2) <- c("John", "Helen", "Mary", "Joe", "Art")
```

```
> d2
      midterm final
John       80    88
Helen      NA   100
Mary       90    94
Joe        40    88
Art        95    99
> cbind(d, d2)
     sex height weight liveOnCampus midterm final
2345   M     65    150         TRUE      80    88
1236   F     63    140         TRUE      NA   100
2986   F     60    135        FALSE      90    94
6543   M     62    165        FALSE      40    88
6544   M     57    175        FALSE      95    99
```

To combine data frame row-wise, we use the `rbind()` function. The number of columns must be the same and the names of the columns must match.

```
> d3 <- data.frame(sex=c("M", "F"), height=c(63, 60), weight=c(160, 146),
+ liveOnCampus=c(FALSE, FALSE))
> rbind(d, d3)
     sex height weight liveOnCampus
2345   M     65    150         TRUE
1236   F     63    140         TRUE
2986   F     60    135        FALSE
6543   M     62    165        FALSE
6544   M     57    175        FALSE
1      M     63    160        FALSE
2      F     60    146        FALSE
```

READING DATA

Most of time, a data frame is created by reading an external text file. The `read.table()` function is used to read an external text file in which each field is separated by one or more separators. The result from the `read.table()` function is a data frame. The `read.table()` function has a large number of arguments that you can use to read a file with different formats. When reading external files by using `read.table()`, you should consider the following format criteria of the input file:

- Separators: The default separators are spaces, tabs, newlines, or carriage returns. To specify another type of separator, you need to use the `sep` argument.
- Header: If the first row of the external file contains the variable name, you need to specify the `header = TRUE` option. By default, this argument is set to FALSE and `read.table()` uses V followed by the column number as the variable names.
- Row names: If the header line is one column shorter than the body of the file, the first column in the file is taken as row names. In this case, the `header` option is automatically set to TRUE.
- Missing values: The `read.table()` function can recognize NA as a missing value for any data type and treat `NaN`, `Inf`, and `-Inf` as missing for numeric data. You can modify the default option by using the `na.strings` argument.
- Comments: By default, the `read.table()` function treats any text after the # sign as comments. To change the default option, you can use the `comment.char` argument. For example, `comment.char ="%"`.
- Skip lines: You can skip a number of lines from the input file by using the `skip` option.
- Number of rows: You can control the maximum number of lines to read by using the `nrow` option.

Suppose we want to read a text file, example1.txt, into R. Each field of example1.txt is separated by a tab. The first two lines are comments after #. The first row after the comments has the variable names. The missing values are represented as periods(.).

```
> setwd("C:/Users/Arthur/Documents/SAS Talk/Talks/R 101")
> example1 = read.table(file="example1.txt", header= T, na.strings = ".")
```

```
> head(example1)
     Fname    Lname race age preg income
1    KAREN    ARIAS    H  26    0  35000
2 Caroline   Embrey    W  26    1  48000
3      GEN ERECKSON    W  32    1  30000
4     JOAN   RIVERA    W  17    0  59000
5   ANDREA    Jones    B  29    1 120000
6  BEVERLY    ROELL    W  26    1 113000
```

The `setwd()` is used because example1.txt is located in C:/Users/Arthur/Documents/SAS Talk/Talks/R 101 directory. Instead of using the `setwd()` function, you can also write-out the full name after the file option. For example, `file=" C:/Users/Arthur/Documents/SAS Talk/Talks/R 101/example1.txt"`. In the `read.table()` function above, the `header` argument is set to TRUE since the variable names are in the input file. The `comment.char` argument is not used since # is the default value for indicating comments. The `na.strings` argument is set to a period (.) which is the missing value for the input data. Since each field is separated by tabs, the `sep` option is not used.

There are other similar functions to read raw data. These functions are the wrapper functions for the `read.table()` function. For example, the `read.csv()` function reads the comma separated value files. It sets the `header` argument to TRUE and the `sep` argument to a comma (,) for the `read.table()` function. The `read.delim()` function reads delimited files, defaulting to the TAB character for the delimiter and TRUE for the `header`. There are many ways to read an EXCEL file. For example, you can use the `read.xlsx()` function from the `xlsx` package.

SUBSETTING OBJECTS

INDEX VECTORS

To subset one or more components from an object, we need to create an index vector. The index vector is used to identify which elements to choose. When selecting an object, we need to insert an index vector within the square brackets of an object. There are five types of index vectors and you cannot mix them when performing object subsetting.

Type 1: A logical vector

When a logical vector is used as an index vector to select an object, it should be the same length as the vector from which the elements are to be selected. Values corresponding to TRUE in the index vector are being selected, and values corresponding to FALSE are being excluded. Values corresponding to a missing value returns NA. For example,

```
> a <- c(1, 3, 5, NA, 7)
> is.na(a)
[1] FALSE FALSE FALSE TRUE FALSE
> !is.na(a)
[1] TRUE TRUE TRUE FALSE TRUE
> a[!is.na(a)]
[1] 1 3 5 7
> a > 3
[1] FALSE FALSE TRUE NA TRUE
> a [a > 3]
[1] 5 NA 7
```

If the index vector is shorter, then its elements will be recycled. If the length of the vector being subsetted is not a multiple of the length of the index vector, no warning message is generated.

Type 2: A vector of positive integer

The values in the positive integer index vector lie in the set 1, 2, ..., `length(x)` most of the time. The corresponding elements of the vectors are selected and concatenated.

```
> a[c(1:3, 2)]
[1] 1 3 5 3
```

Using the `which()` function to generate a vector of integer is better approach to select elements than using a logical vector. The `which()` function returns the TRUE integer indices of a logical object and it doesn't select the missing values (NA). For example,

```
> a[a > 3]
[1] 5 NA 7
> which(a>3)
[1] 3 5
> a[which(a>3)]
[1] 5 7
```

If the element of the index vector exceeds `length(x)`, then the corresponding selection is NA. Elements of the index vector that are NA generate a NA.

Type 3: A vector of negative integer
The negative integer index vector is use to specify the elements to be excluded. The following example drops the first and the fourth elements of the vector a.

```
> a[-c(1,4)]
[1] 3 5 7
```

NA is not allowed to be included in the index vector. You cannot mix positive and negative index together.

Type 4: A vector of character strings
The index vector that contains character strings only applies to an object that has names. If the target vector has no names, it will result in an NA

```
> a[c("a","c")]
[1] NA NA
> names(a)<- letters[1:5]
> a
 a  b  c  d  e
 1  3  5 NA  7
> a[c("a","c")]
a c
1 5
```

Type 5: The index position may be empty
When the index vector is left empty, all the components of the vector are selected. An empty index is not very useful for applying to vectors; but it will be useful for selecting components from matrices, arrays, and data frames.

```
> a[]
 a  b  c  d  e
 1  3  5 NA  7
```

SUBSETTING MATRICES
To select components from a matrix, we need to specify the index vector(s) at either row and/or column position of a matrix. Any of the five indices can be used to select elements from a matrix.

```
> x= matrix(1:12, nrow=3)
> rownames(x)<-letters[1:3]
> colnames(x)<-LETTERS[1:4]
> x
  A B C  D
a 1 4 7 10
b 2 5 8 11
c 3 6 9 12
```

The following example selects the first and the third rows, and columns with column names equaling to “A” or “C” from the matrix `x`.

```
> x[1:3, c("A", "C")]
  A C
a 1 7
b 2 8
c 3 9
```

You can also select only rows or columns of a matrix. Not specifying an index vector in the row position means all the rows are selected. Similarly, to select all the column components, we need to leave the column position empty as well.

```
> x[c("a", "c"),]
  A B C  D
a 1 4 7 10
c 3 6 9 12
> x[,-c(2,4)]
  A C
a 1 7
b 2 8
c 3 9
```

SUBSETTING LISTS

To extract components from a list, you can use single brackets ([), double brackets([[), or $ operator. Using single brackets allows you to extract one or more components. Any of the five forms of indexing can be used within the single bracket. Using single brackets to extract components from a list always returns a list object. For example:

```
> student[c(1,3)]
$name
[1] "John"

$classTaken
[1] "PM510"  "PM511A" "PM511B"

> student[c("year", "GPA")]
$year
[1] 2

$GPA
[1] 3.85
```

You can also use double brackets to extract one single component from a list. The object type that is being returned is the same as the object type of selected component.

```
> student[[3]]
[1] "PM510"  "PM511A" "PM511B"
> student[["GPA"]]
[1] 3.85
```

Since the third component from `student` is a vector, `student[[3]][2]` selects the second entry of `student[[3]]`.

```
> student[[3]][2]
[1] "PM511A"
```

You can also use the $ operator to select one single component from a list by specifying the component name. For example,

```
> student$classTaken
[1] "PM510"  "PM511A" "PM511B"
> student$classTaken[2]
[1] "PM511A"
```

SUBSETTING DATA FRAMES

Data frames share the common properties of matrices and lists. If you subset with two vectors at both positions, it behaves like subsetting a matrix. For example,

```
> d[c(1,3),]
     sex height weight liveOnCampus
2345   M     65    150         TRUE
2986   F     60    135        FALSE
```

```
> d[,c(2,4)]
     height liveOnCampus
2345     65         TRUE
1236     63         TRUE
2986     60        FALSE
6543     62        FALSE
6544     57        FALSE
> d[d$sex=="M",]
     sex height weight liveOnCampus
2345   M     65    150         TRUE
6543   M     62    165        FALSE
6544   M     57    175        FALSE

> d[-c(1,3), c("sex", "height")]
     sex height
1236   F     63
6543   M     62
6544   M     57
```

If you subset with a single vector, it behaves like subsetting a list.

```
> d[c("sex", "liveOnCampus")]
     sex liveOnCampus
2345   M         TRUE
1236   F         TRUE
2986   F        FALSE
6543   M        FALSE
6544   M        FALSE
> d[-c(1,3)]
     height liveOnCampus
2345     65         TRUE
1236     63         TRUE
2986     60        FALSE
6543     62        FALSE
6544     57        FALSE
```

DATA MANIPULATIONS

CREATING AND RE-CODING VARIABLES

We will use the data set, `painters`, from the MASS library to illustrate some example.

```
> library(MASS)
> head(painters)
             Composition Drawing Colour Expression School
Da Udine              10       8     16          3      A
Da Vinci              15      16      4         14      A
Del Piombo             8      13     16          7      A
Del Sarto             12      16      9          8      A
Fr. Penni              0      15      8          0      A
Guilio Romano         15      16      4         14      A
```

We can create an indicator variable based on existing continuous variables or categorical variables by using relational operators. The indicator variable can be either in the form of a logical vector with values TRUE or FALSE or an integer vector with values 1 or 0. To create an indicator variable in the form of integers, you can use the `as.integer()` function which can convert an logical vector to an integer vector. Suppose we would like to create a variable indicating whether each element of the `Drawing` variable is above or below its mean value. To add a new variable to the existing data set, you can use the $ operator.

```
> painters$DrawingInd <- painters$Drawing >= mean(painters$Drawing)
```

```
> head(painters)
             Composition Drawing Colour Expression School DrawingInd
Da Udine              10       8     16          3      A      FALSE
Da Vinci              15      16      4         14      A       TRUE
Del Piombo             8      13     16          7      A       TRUE
Del Sarto             12      16      9          8      A       TRUE
Fr. Penni              0      15      8          0      A       TRUE
Guilio Romano         15      16      4         14      A       TRUE
```

The following example shows how to create an ordinal variable, `CompositionOrd`, based on `Composition` variable. If `Composition` is less than 8, `CompositionOrd` will be set to 1. `CompositionOrd` is set to 2 for `Composition` greater and equal to 8 but less than 12. `CompositionOrd` is set to 3 for `Composition` greater and equal to 12.

```
> painters$CompositionOrd <- 1 + (painters$Composition >=8) +
(painters$Composition >=12)
> head(painters[c("Composition", "CompositionOrd")])
             Composition CompositionOrd
Da Udine              10              2
Da Vinci              15              3
Del Piombo             8              2
Del Sarto             12              3
Fr. Penni              0              1
Guilio Romano         15              3
```

The `ifelse()` function is a useful function to create indicator, ordinal, or categorical variables. The `ifelse()` function has the following form: **`ifelse (test, yes, no).`** The function returns a value with the same shape as `test`. The result is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is TRUE or FALSE. For example, the `School` variable in the `painters` data frame has the following levels:

```
> table(painters$School)

 A  B  C  D  E  F  G  H
10  6  6 10  7  4  7  4
```

To create a variable that equals to 1 if the School variable equals to A, B or C, and equals to 2 otherwise, we can write the following code:

```
> School2 <- ifelse(is.element(painters$School, c("A", "B", "C")), 1, 2)
> table(School2, painters$School)

School2  A  B  C  D  E  F  G  H
      1 10  6  6  0  0  0  0  0
      2  0  0  0 10  7  4  7  4
```

The `cut` function is often used to create categorical variables based on existing continuous variables. The `cut` function has five of the following arguments:

- `x` is the variable that needs to be divided.
- `breaks` is a numeric vector of two or more cut points. For example, providing `breaks = c(n1, n2, n3, n4)` divides `x` into three intervals: (n1, n2], (n2, n3], and (n3, n4]. Notice that n1 is not included in any of these three intervals.
- The default value for the `labels` argument is NULL. It means that the resulting categorical variable will be labeled as (n1, n2], (n2, n3], and (n3, n4]. You can provide a vector of character values for labeling purposes.
- You can set the `include.lowest` argument to TRUE to include the smallest number in the lowest interval. Therefore, the resulting intervals will be [n1, n2], (n2, n3], and (n3, n4].
- To set the `right` argument to FALSE, the resulting intervals will be [n1, n2), [n2, n3), and [n3, n4).

Suppose that we would like to create a categorical variable of four levels based on the `Colour` variable. Each level corresponds to each quartile of `Colour`.

```
> qt <- quantile(painters$Colour, c(0, 0.25, 0.5, 0.75, 1))
```

```
> qt
   0%   25%   50%   75%  100%
 0.00  7.25 10.00 16.00 18.00
> painters$ColourCat <- cut(painters$Colour, qt, labels= c("first", "second", "third",
+ "fourth"), include.lowest = T)
> table(painters$ColourCat)

 first second  third fourth
  14     15     18      7
```

SORTING AN OBJECT

When sorting a data frame, you need to use the `order()` function, which produces an index vector. Then you use this index vector to sort the data frame. For illustration purposes, we will create a smaller data frame, `painterABC`.

```
> painterABC <- painters[is.element(painters$School, c("A", "B", "C")),]
> nameIndex <- order (row.names(painterABC))
> nameIndex
 [1] 17 18  1  2  3  4 11  5 12  6 19 20  7 13  8  9 14 10 15 21 22 16
> painterABC[nameIndex,]
                Composition Drawing Colour Expression School
Barocci                  14      15      6         10      C
Cortona                  16      14     12          6      C
Da Udine                 10       8     16          3      A
Da Vinci                 15      16      4         14      A
Del Piombo                8      13     16          7      A
Del Sarto                12      16      9          8      A
F. Zucarro               10      13      8          8      B
Fr. Penni                 0      15      8          0      A
Fr. Salviata             13      15      8          8      B
Guilio Romano            15      16      4         14      A
Josepin                  10      10      6          2      C
L. Jordaens              13      12      9          6      C
Michelangelo              8      17      4          8      A
Parmigiano               10      15      6          6      B
Perino del Vaga          15      16      7          6      A
Perugino                  4      12     10          4      A
Primaticcio              15      14      7         10      B
Raphael                  17      18     12         18      A
T. Zucarro               13      14     10          9      B
Testa                    11      15      0          6      C
Vanius                   15      15     12         13      C
Volterra                 12      15      5          8      B
```

In the example above, an index vector is created first by using the `order()` function, then using the index to rearrange the data frame. If you want to sort the data frame by the variable `School` in decreasing order, we need to set the `decreasing` argument to TRUE.

```
> painterABC[order(painterABC$School, decreasing=TRUE),]
                Composition Drawing Colour Expression School
Barocci                  14      15      6         10      C
Cortona                  16      14     12          6      C
Josepin                  10      10      6          2      C
L. Jordaens              13      12      9          6      C
Testa                    11      15      0          6      C
Vanius                   15      15     12         13      C
F. Zucarro               10      13      8          8      B
Fr. Salviata             13      15      8          8      B
Parmigiano               10      15      6          6      B
Primaticcio              15      14      7         10      B
T. Zucarro               13      14     10          9      B
Volterra                 12      15      5          8      B
Da Udine                 10       8     16          3      A
Da Vinci                 15      16      4         14      A
```

```
Del Piombo                8      13     16          7      A
Del Sarto                12      16      9          8      A
Fr. Penni                 0      15      8          0      A
Guilio Romano            15      16      4         14      A
Michelangelo              8      17      4          8      A
Perino del Vaga          15      16      7          6      A
Perugino                  4      12     10          4      A
Raphael                  17      18     12         18      A
```

When sorting the data by using more than one variable, the `decreasing` argument applies to all variables. For example:

```
> painterABC[order (painterABC$School, painterABC$Colour, painterABC$Drawing,
+ decreasing=T),]
               Composition Drawing Colour Expression School
Vanius                  15      15     12         13      C
Cortona                 16      14     12          6      C
L. Jordaens             13      12      9          6      C
Barocci                 14      15      6         10      C
Josepin                 10      10      6          2      C
Testa                   11      15      0          6      C
T. Zucarro              13      14     10          9      B
Fr. Salviata            13      15      8          8      B
F. Zucarro              10      13      8          8      B
Primaticcio             15      14      7         10      B
Parmigiano              10      15      6          6      B
Volterra                12      15      5          8      B
Del Piombo               8      13     16          7      A
Da Udine                10       8     16          3      A
Raphael                 17      18     12         18      A
Perugino                 4      12     10          4      A
Del Sarto               12      16      9          8      A
Fr. Penni                0      15      8          0      A
Perino del Vaga         15      16      7          6      A
Michelangelo             8      17      4          8      A
Da Vinci                15      16      4         14      A
Guilio Romano           15      16      4         14      A
```

For numerical vectors in the data frame, you can use the "-" sign to control which vector needs to be sorted in decreasing order. For example:

```
> painterABC[order(painterABC$School, -painterABC$Colour, painterABC$Drawing),]
               Composition Drawing Colour Expression School
Da Udine                10       8     16          3      A
Del Piombo               8      13     16          7      A
Raphael                 17      18     12         18      A
Perugino                 4      12     10          4      A
Del Sarto               12      16      9          8      A
Fr. Penni                0      15      8          0      A
Perino del Vaga         15      16      7          6      A
Da Vinci                15      16      4         14      A
Guilio Romano           15      16      4         14      A
Michelangelo             8      17      4          8      A
T. Zucarro              13      14     10          9      B
F. Zucarro              10      13      8          8      B
Fr. Salviata            13      15      8          8      B
Primaticcio             15      14      7         10      B
Parmigiano              10      15      6          6      B
Volterra                12      15      5          8      B
Cortona                 16      14     12          6      C
Vanius                  15      15     12         13      C
L. Jordaens             13      12      9          6      C
Josepin                 10      10      6          2      C
```

```
Barocci                  14      15      6         10      C
Testa                    11      15      0          6      C
```

In order to use the negative sign to sort a character column in a data frame, you need to use the `xtfrm()` function. The `xtfrm()` function creates a numeric vector based on its argument, which will sort in the same order as its argument. For example,

```
> xtfrm(c("A", "D", "B", "C"))
[1] 1 4 2 3
```

Suppose that you would like to sort the `painterABC` data frame by `School` in descending order and `Drawing` in ascending order. You can write the following:

```
> painterABC[order(-xtfrm(painterABC$School), painterABC$Drawing),]
                Composition Drawing Colour Expression School
Josepin                  10      10      6          2      C
L. Jordaens              13      12      9          6      C
Cortona                  16      14     12          6      C
Barocci                  14      15      6         10      C
Testa                    11      15      0          6      C
Vanius                   15      15     12         13      C
F. Zucarro               10      13      8          8      B
Primaticcio              15      14      7         10      B
T. Zucarro               13      14     10          9      B
Fr. Salviata             13      15      8          8      B
Parmigiano               10      15      6          6      B
Volterra                 12      15      5          8      B
Da Udine                 10       8     16          3      A
Perugino                  4      12     10          4      A
Del Piombo                8      13     16          7      A
Fr. Penni                 0      15      8          0      A
Da Vinci                 15      16      4         14      A
Del Sarto                12      16      9          8      A
Guilio Romano            15      16      4         14      A
Perino del Vaga          15      16      7          6      A
Michelangelo              8      17      4          8      A
Raphael                  17      18     12         18      A
```

The `order()` function has an argument, `na.last`, that determines the way of placing the missing values in the result. The default value for this argument is TRUE; it means the missing values are placed at the end. You can set this argument to FALSE if you want to place the missing values at the top of sorted data. To remove all the missing values from the result, you can set this argument to NA.

MERGING DATA FRAMES

The `merge()` function is to used to merge two data sets by a common variable. Suppose we would like to merge the following two data sets by `id`.

```
> data1 <- data.frame(id= c(2,1,3), var1=c("A","B","C"))
> data2 <- data.frame(id= c(3,4,1), var2=c(1,3,9))
> merge(data1, data2)
  id var1 var2
1  1    B    9
2  3    C    1
```

Notice that the resulting data frame only contains observations that are matched by the `id` variable. Suppose we want the resulting data frame to contain all the observations from `data1`, we need to use the `all.x` argument.

```
> merge(data1, data2, all.x=TRUE)
  id var1 var2
1  1    B    9
2  2    A   NA
3  3    C    1
```

Similarly, we can set the `all.y` argument to TRUE to allow the resulting data frame to contain all the observations from `data2`. You can set the `all` argument to TRUE to contain all the observations. By default, R automatically merges data sets by columns with common names. However, you can specify the variable explicitly by using the `by` argument:

```
> merge(data1, data2, all=TRUE, by="id")
  id var1 var2
1  1    B    9
2  2    A   NA
3  3    C    1
4  4 <NA>    3
```

Suppose that common columns that we want to merge have different names. You can use the `by.x` and `by.y` arguments to indicate which variables to use for merging. For example, you would like to merge the following two data frames by `ID` (or `id`). Notice that in `data3`, `ID` is in upper case. Also, both data frames have `var2` as the common variable name, but we don't want to merge the data by `var2`.

```
> data3 <- data.frame(ID= c(1,3,4), var2=c(1, 8,9))
> merge(data2, data3, by.x="id", by.y="ID")
  id var2.x var2.y
1  1      9      1
2  3      1      8
3  4      3      9
```

Notice that `.x` and `.y` are added to `var2` as suffixes. We can use the `suffixes` argument to specify our own suffixes. For example,

```
> merge(data2, data3, by.x="id", by.y="ID", suffixes=c(".data2", ".data3"))
  id var2.data2 var2.data3
1  1          9          1
2  3          1          8
3  4          3          9
```

CONCLUSION

R is getting more and more popular among programmers, analysts, and researchers. But being proficient in the R language is not an easy task, and one can easily be overwhelmed with the tremendous language elements in R. One of the key components for learning this language is understanding the characteristics of different objects and knowing the basic functions or operators for creating, combining, and extracting elements from them. Grasping these basics will simplify the complexity for learning more advanced materials in R.

REFERENCES

Spector, Phil. *Data Manipulation with R*. Springer, 2008.
Wickham, Hadley. *Advanced r*. Taylor & Francis Ltd, 2017.
Venables, William N., and Brian D. Ripley. *S Programming*. Springer, 2000.

CONTACT INFORMATION

Arthur Li
City of Hope National Medical Center
Department of Information Science
1500 East Duarte Road
Duarte, CA 91010 - 3000
Work Phone: (626) 256-4673 ext. 65121
E-mail: arthurli@coh.org