Data types and structures

Previously, we used str to inform on the **structure** of an object. Knowing the type and the structure of the data you are manipulating is critical before any further analysis. Yet, this step is often skipped whch may cause serious issue in subsequent steps. Probably boring and for sure lenghty, this section introduces some common data type and structure of data often met during this class.

# Core data types

These data types, or **modes**, define how the values are stored in the computer. You can get an object’s mode using the typeof function. Note that R also has a built-in mode function that will serve the same purpose with the one exception in that it will not distinguish **integers** from **doubles** (see meanings below).

## Numeric

The **numeric** data type is probably the simplest. It consists of numbers such as **integers** (whole-valued positive or negative number or 0, e.g. 1 ,-3 ,33 ,0) or **doubles** (e.g. 0.3, 12.4, -0.04, 1.0). For example, to create a numeric (double) vector we can write:

x <- c(1.0, -3.4, 2, 140.1) # numeric and double  
typeof(x)

## [1] "double"

mode(x)

## [1] "numeric"

Note that removing the fraction part of a number when creating a numeric object does not necessarily create an integer. For example, creating what seems to be an integer object returns double using typeof:

x <- 4  
typeof(x)

## [1] "double"

To force R to recognize a value as an integer add an upper case L to the number.

x <- 4L  
typeof(x)

## [1] "integer"

## Character

The **character** data type consists of letters or words such as “a”, “f”, “project”, “house value”.

x <- c("bubul", "magpie", "spoonbill", "barbet")  
typeof(x)

## [1] "character"

Characters can also consist of numbers represented as characters. The distinction between a character representation of a number and a numeric one is important. For example, if we have two numeric vectors x and y such as

x <- 3  
y <- 5.3  
x + y

## [1] 8.3

If we repeat these steps but instead of representing x and y as **numeric** and **doubles** we input ‘3’ and ‘5.3’ as **character**. We will logically get an error when attempting to sum them:

x <- "3"  
y <- "5.3"  
  
# not run: x+ y  
  
###########################################################  
# Error in x + y: non-numeric argument to binary operator #  
###########################################################

Note the use of quotation marks to force **double** to be recognized as **character**. Be careful some function such as plot will automatically convert your **character** as **double**, try plot(x,y)

## Logical

Logical values can take one of two values/conditions: TRUE or FALSE. These can also be represented as 1 or 0, later assigned as **logical** values. For example, to create a logical vector of 4 elements, you can type

x <- c(TRUE, FALSE, FALSE, TRUE)

or

x1<-c(1,0,0,1)  
x2 <- as.logical(c(1,0,0,1))  
# OR: x3 <- as.logical(c(1,0,0,1))

Note that in both cases, typeof returns **logical**. Also note that the 1’s and 0’s in the last example are converted to TRUE’s and FALSE’s internally using the function as.logical (= create a logical vector).

# Derived data types

These data types are stored as either numeric, character or logical but have an additional *attribute* information that allows these objects to be treated in special ways by certain functions. These attributes define an object’s **class** and can be extracted from an object using the function class.

## Factor

**Factor** is used to group variables into a fixed number of unique categories or **levels**. For example, a data set may be grouped by *gender* or *month* of the year. Such data are usually loaded into R as a **numeric** or **character** data type, requiring then to be converted as a factor using the as.factor function (if it was not automatically done at the importation of the data).

a <- c("M", "F", "F", "U", "F", "M", "M", "M", "F", "U")  
typeof(a) # mode character

## [1] "character"

class(a)# class character

## [1] "character"

a.fact <- as.factor(a)  
class(a.fact)# class factor

## [1] "factor"

**Note**: a did not change of mode. Data are still of the type character.

However, the derived object a.fact is now stored as an integer!

mode(a.fact)

## [1] "numeric"

typeof(a.fact)

## [1] "integer"

Yet, when displaying the contents of a.fact we see character values.

a.fact

## [1] M F F U F M M M F U  
## Levels: F M U

**為什麼**, how can it be ? Well, a.fact is a more complicated object than the simple objects created so far. In fact, the **factor** is storing additional information not seen in its output. This hidden information is stored in **attributes**. To view these hidden attributes, use the attributes function.

attributes(a.fact)

## $levels  
## [1] "F" "M" "U"  
##   
## $class  
## [1] "factor"

There are two attributes: levels and class. The levels attribute lists the three unique values in a.fact. The order in which these levels are listed reflect their *numeric* representation [check str (a.fact) and numeric sequence created: 2 1 1 3 ...]. So in essence, a.fact is storing each value as an integer that points to one of the three unique levels. The unique levels of a factor, and the order in which they are stored can be extracted using the levels function.

levels(a.fact)

## [1] "F" "M" "U"

Remember, the order in which the levels are displayed match their integer representation. If we want them in a different order (i.e. U first followed by F then M) we need to modify them by recreating the factor object as follows:

factor(a, levels=c("U","F","M"))

## [1] M F F U F M M M F U  
## Levels: U F M

This detail is critically important when you are trying to rearrange elements in a plot, so yu have to remember it.

**Practice 3.1** Understanding the factors and their levels is critical, especially when you subset or make plots. Let’s explore together why using our iris data set. Subset species setosa and virginica into an new data set iris.sel. Check the levels in the variable iris.sel$Species. Conclude on the possible consequence?

iris.sel<- subset(iris, Species == "setosa" | Species == "virginica")  
levels(iris.sel$Species) # 3 species are still there  
# boxplot(Petal.Width ~ Species, iris.sel, horizontal = TRUE)

To drop a level after a subset, you can use the function droplevels on a targeted variable. Try it on the variable iris.sel$Species and check the previous plot.

It is now time to change the row names in our subset:

rownames(iris.sel) = seq(length=nrow(iris.sel))

**ALWAYS**, **ALWAYS**, CHECK THE DATA TYPE AND STRUCTURE OF YOUR DATA SET!

## Date

Date values are stored as numbers. But to be properly interpreted as a date object in R, their attribute must be explicitly defined as a **date**. base provides many facilities to convert and manipulate dates and times, but a package called lubridate [see library (lubridate)] makes working with dates/times much easier. Working with dates could be the object of a specific topic, and it will no be covered here.

## NAs and NULLs

Often you will find missing or unknown values in data set.It may be tempting to assign these missing or unknown values to **0**. However, doing so can be very wrong and can lead to many undesirable results when analyzing the data. R has two placeholders for such elements: NA and NULL

x <- c(23, NA, 1.2, 5)

NA (Not Available) is a missing value indicator. It suggests that a value should be present but is unknown.

The NULL object also represents missing values but its interpretation is slightly different in that it suggests that the value does not exist or that it’s not measurable.

y <- c(23, NULL, 1.2, 5)

The difference between NA and NULL may seem subtle, but their interpretation in some functions can lead to different outcomes. For example, when computing the mean of x, R returns an NA value:

mean(x)

## [1] NA

This serves as a check to remind the user that one of the elements is missing. This can be overcome by setting the na.rm parameter to TRUE as in mean(x, na.rm=T) in which case R ignores the missing value.

A NULL object, on the other hand, is treated differently. Since NULL implies that a value should not be present, R no longer feels the need to treat such element as questionable and allows the mean value to be computed:

mean(y)

## [1] 9.733333

It’s more common to find data tables with missing elements populated with NA’s than NULL’s so unless you have a specific reason to use NULL as a missing value placeholder, use NA instead.

# Data structures

Most data sets we work with consist of batches of values such as a **table** (ex: temperature values) or a list of results (ex: annual survey). These batches of data are stored in R in one of several data structures. These include **(atomic) vector**, **dataframe**, **matrix** and **lists**.

## (Atomic) vectors

The **atomic vector** (or **vector** for short) is the simplest data structure in R which consists of an ordered set of values of the same type and or class (e.g. numeric, character, date, etc…). As we did it before, a vector can be sinply created using the combine function c() as in:

x <- c(674 , 4186 , 5308 , 5083 , 6140 , 6381)  
x

## [1] 674 4186 5308 5083 6140 6381

As we seen before, a vector object is an **indexable** collection of values which allows one to access a specific index number (subscript). Remimber to access the third element of x, type:

x[3]

## [1] 5308

Also already seen it before, but you can select a subset of elements by index values using the combine function c().

x[c(1,3,4)]

## [1] 674 5308 5083

Or, if you are interested in a range of indexed values such as index 2 through 4, use the : operator.

x[2:4]

## [1] 4186 5308 5083

You can also assign new values to a specific index. For example, we can replace the second value in vector x with 0.

x[2] <- 0  
x

## [1] 674 0 5308 5083 6140 6381

Note that a vector can store any data type such as characters.

x <- c("all", "b", "olive")  
x

## [1] "all" "b" "olive"

However, **a vector can only be of one type**. For example, you cannot mix numeric and character types as follows:

x <- c( 1.2, 5, "Rt", "2000")

In such a situation, R will convert the element types to the **highest common mode** following the order **NULL** < **logical** < **integer** < **double** < **character**. In our working example, the elements are coerced to character:

typeof(x)

## [1] "character"

## Matrices and arrays

Matrices in R can be thought of as vectors indexed using two indices instead of one. For example, the following line of code creates a 3 by 3 matrix of randomly generated values. The parameters nrow and ncol define the matrix dimension and the function runif generates nine random numbers that populate this matrix.

m <- matrix(runif(9,0,10), nrow = 3, ncol = 3)  
m

## [,1] [,2] [,3]  
## [1,] 0.1884348 0.3239877 3.560227  
## [2,] 4.4045853 7.6155979 6.583111  
## [3,] 0.8016988 4.2329608 3.441703

If a higher dimension vector is desired, then use the array function to generate the n-dimensional object. A 3x3x3 array can be created as follows:

m <- array(runif(27,0,10), c(3,3,3))  
m

## , , 1  
##   
## [,1] [,2] [,3]  
## [1,] 1.590723 5.017405 7.7110079  
## [2,] 6.589408 8.872715 0.4909224  
## [3,] 4.378727 6.135730 1.3366331  
##   
## , , 2  
##   
## [,1] [,2] [,3]  
## [1,] 7.814215 0.8830889 2.5657942  
## [2,] 1.837691 1.5734359 0.2465858  
## [3,] 8.953766 6.1665381 6.3142486  
##   
## , , 3  
##   
## [,1] [,2] [,3]  
## [1,] 3.287069 1.823886 3.244362  
## [2,] 5.829482 8.639950 1.822876  
## [3,] 5.837465 5.281307 5.279875

Matrices and arrays can store numeric or character data types, but they cannot store both. They derived directly form vector structure but have a higher number of dimensions.

## Data frames

A **data frame** is what comes closest to our perception of a data table. It’s an extension of the matrix object in that, unlike a matrix, a data frame can *mix* data types across columns (e.g. both **numeric** and **character** columns can coexist in a data frame) but data type remains the same across rows.

name <- c("a1", "a2", "b3")  
value1 <- c(23, 4, 12)  
value2 <- c(1, 45, 5)  
dat <- data.frame(name, value1, value2)  
dat

## name value1 value2  
## 1 a1 23 1  
## 2 a2 4 45  
## 3 b3 12 5

str(dat) # provide structure

## 'data.frame': 3 obs. of 3 variables:  
## $ name : chr "a1" "a2" "b3"  
## $ value1: num 23 4 12  
## $ value2: num 1 45 5

attributes(dat) # provide attributes

## $names  
## [1] "name" "value1" "value2"  
##   
## $class  
## [1] "data.frame"  
##   
## $row.names  
## [1] 1 2 3

names(dat) # extract colum names

## [1] "name" "value1" "value2"

rownames(dat) # extract row names

## [1] "1" "2" "3"

## Lists

A **list** is an ordered set of components stored in a 1D vector. In fact, it’s another kind of vector called a **recursive vector** where each vector element can be of different data type and structure. This implies that **each element of a list** can hold complex objects such as **matrix**, **dataframe** and even in some cases other **list** objects imbricated within the **lis**! Think of a list as a single column spreadsheet where each cell stores anything from a single number to a three paragraph sentence or even another five column table.

A list is constructed using the list function. For example, the following list consists of 3 components: a two-column data frame (tagged as component A), a two element logical vector (tagged as component B) and a three element character vector (tagged as component C).

A <- data.frame(  
 x = c(7.3, 29.4, 29.4, 2.9, 12.3, 7.5, 36.0, 4.8, 18.8, 4.2),  
 y = c(5.2, 26.6, 31.2, 2.2, 13.8, 7.8, 35.2, 8.6, 20.3, 1.1) )  
B <- c(TRUE, FALSE)  
C <- c("apples", "oranges", "round")  
my.lst <- list(A = A, B = B, C = C)

You can view each component’s structure using the str function.

str(my.lst)

## List of 3  
## $ A:'data.frame': 10 obs. of 2 variables:  
## ..$ x: num [1:10] 7.3 29.4 29.4 2.9 12.3 7.5 36 4.8 18.8 4.2  
## ..$ y: num [1:10] 5.2 26.6 31.2 2.2 13.8 7.8 35.2 8.6 20.3 1.1  
## $ B: logi [1:2] TRUE FALSE  
## $ C: chr [1:3] "apples" "oranges" "round"

names(my.lst)

## [1] "A" "B" "C"

Each component of a list can be extracted using the $ symbol followed by that component’s name. For example, to access component A from list lst, type:

my.lst$A

## x y  
## 1 7.3 5.2  
## 2 29.4 26.6  
## 3 29.4 31.2  
## 4 2.9 2.2  
## 5 12.3 13.8  
## 6 7.5 7.8  
## 7 36.0 35.2  
## 8 4.8 8.6  
## 9 18.8 20.3  
## 10 4.2 1.1

You can also access that same component using its numerical index. Since A is the first component in lst, its numerical index is 1.

my.lst[[1]]

## x y  
## 1 7.3 5.2  
## 2 29.4 26.6  
## 3 29.4 31.2  
## 4 2.9 2.2  
## 5 12.3 13.8  
## 6 7.5 7.8  
## 7 36.0 35.2  
## 8 4.8 8.6  
## 9 18.8 20.3  
## 10 4.2 1.1

class(my.lst[[1]])

## [1] "data.frame"

Note that components do not require names. For example, we could have created a list as follows (note the omission of A=, B=, etc…):

lst.notags <- list(A, B, D)  
lst.notags

## [[1]]  
## x y  
## 1 7.3 5.2  
## 2 29.4 26.6  
## 3 29.4 31.2  
## 4 2.9 2.2  
## 5 12.3 13.8  
## 6 7.5 7.8  
## 7 36.0 35.2  
## 8 4.8 8.6  
## 9 18.8 20.3  
## 10 4.2 1.1  
##   
## [[2]]  
## [1] TRUE FALSE  
##   
## [[3]]  
## function (expr, name)   
## .External(C\_doD, expr, name)  
## <bytecode: 0x000001d911884678>  
## <environment: namespace:stats>

When lists do not have component names, the names function will return NULL.

names(lst.notags)

## NULL

You’ll find that many functions in R return list objects such as the linear regression model function lm. For example, when we run a regression analysis for vector elements x and y (in data frame A) and save the output of the regression analysis to an object called M:

M <- lm( y ~ x, A)  
str(M)

## List of 12  
## $ coefficients : Named num [1:2] 0.0779 0.991  
## ..- attr(\*, "names")= chr [1:2] "(Intercept)" "x"  
## $ residuals : Named num [1:10] -2.112 -2.612 1.988 -0.752 1.533 ...  
## ..- attr(\*, "names")= chr [1:10] "1" "2" "3" "4" ...  
## $ effects : Named num [1:10] -48.07 -36.54 3.13 -0.66 2 ...  
## ..- attr(\*, "names")= chr [1:10] "(Intercept)" "x" "" "" ...  
## $ rank : int 2  
## $ fitted.values: Named num [1:10] 7.31 29.21 29.21 2.95 12.27 ...  
## ..- attr(\*, "names")= chr [1:10] "1" "2" "3" "4" ...  
## $ assign : int [1:2] 0 1  
## $ qr :List of 5  
## ..$ qr : num [1:10, 1:2] -3.162 0.316 0.316 0.316 0.316 ...  
## .. ..- attr(\*, "dimnames")=List of 2  
## .. .. ..$ : chr [1:10] "1" "2" "3" "4" ...  
## .. .. ..$ : chr [1:2] "(Intercept)" "x"  
## .. ..- attr(\*, "assign")= int [1:2] 0 1  
## ..$ qraux: num [1:2] 1.32 1.44  
## ..$ pivot: int [1:2] 1 2  
## ..$ tol : num 1e-07  
## ..$ rank : int 2  
## ..- attr(\*, "class")= chr "qr"  
## $ df.residual : int 8  
## $ xlevels : Named list()  
## $ call : language lm(formula = y ~ x, data = A)  
## $ terms :Classes 'terms', 'formula' language y ~ x  
## .. ..- attr(\*, "variables")= language list(y, x)  
## .. ..- attr(\*, "factors")= int [1:2, 1] 0 1  
## .. .. ..- attr(\*, "dimnames")=List of 2  
## .. .. .. ..$ : chr [1:2] "y" "x"  
## .. .. .. ..$ : chr "x"  
## .. ..- attr(\*, "term.labels")= chr "x"  
## .. ..- attr(\*, "order")= int 1  
## .. ..- attr(\*, "intercept")= int 1  
## .. ..- attr(\*, "response")= int 1  
## .. ..- attr(\*, ".Environment")=<environment: R\_GlobalEnv>   
## .. ..- attr(\*, "predvars")= language list(y, x)  
## .. ..- attr(\*, "dataClasses")= Named chr [1:2] "numeric" "numeric"  
## .. .. ..- attr(\*, "names")= chr [1:2] "y" "x"  
## $ model :'data.frame': 10 obs. of 2 variables:  
## ..$ y: num [1:10] 5.2 26.6 31.2 2.2 13.8 7.8 35.2 8.6 20.3 1.1  
## ..$ x: num [1:10] 7.3 29.4 29.4 2.9 12.3 7.5 36 4.8 18.8 4.2  
## ..- attr(\*, "terms")=Classes 'terms', 'formula' language y ~ x  
## .. .. ..- attr(\*, "variables")= language list(y, x)  
## .. .. ..- attr(\*, "factors")= int [1:2, 1] 0 1  
## .. .. .. ..- attr(\*, "dimnames")=List of 2  
## .. .. .. .. ..$ : chr [1:2] "y" "x"  
## .. .. .. .. ..$ : chr "x"  
## .. .. ..- attr(\*, "term.labels")= chr "x"  
## .. .. ..- attr(\*, "order")= int 1  
## .. .. ..- attr(\*, "intercept")= int 1  
## .. .. ..- attr(\*, "response")= int 1  
## .. .. ..- attr(\*, ".Environment")=<environment: R\_GlobalEnv>   
## .. .. ..- attr(\*, "predvars")= language list(y, x)  
## .. .. ..- attr(\*, "dataClasses")= Named chr [1:2] "numeric" "numeric"  
## .. .. .. ..- attr(\*, "names")= chr [1:2] "y" "x"  
## - attr(\*, "class")= chr "lm"

names(M)

## [1] "coefficients" "residuals" "effects" "rank"   
## [5] "fitted.values" "assign" "qr" "df.residual"   
## [9] "xlevels" "call" "terms" "model"

The M list is more complex than the simple list lst created earlier. In addition to having more components, it stores a *wider* range of data types and structures. For example, element qr is itself a list of five elements!

str(M$qr)

## List of 5  
## $ qr : num [1:10, 1:2] -3.162 0.316 0.316 0.316 0.316 ...  
## ..- attr(\*, "dimnames")=List of 2  
## .. ..$ : chr [1:10] "1" "2" "3" "4" ...  
## .. ..$ : chr [1:2] "(Intercept)" "x"  
## ..- attr(\*, "assign")= int [1:2] 0 1  
## $ qraux: num [1:2] 1.32 1.44  
## $ pivot: int [1:2] 1 2  
## $ tol : num 1e-07  
## $ rank : int 2  
## - attr(\*, "class")= chr "qr"

So if we want to access the element rank in the component qr of list M, we can use:

M$qr$rank

## [1] 2

This structure of output from a function is common. Using lm it is specified in the help of the function (?lm): *“object of class”lm” is a* ***list*** *containing at least the following components”*

# Coercing data

Data can be coerced (forced, ~ converted) from one type to another. For example, to coerce the following vector object from character to numeric, use the as.numeric function.

y <- c("23.8", "6", "100.01","6")  
y.c <- as.numeric(y)  
y.c

## [1] 23.80 6.00 100.01 6.00

The as.numeric function forces the vector to a double (you could have also used the as.double function). If you convert y to an integer, R will remove all fractional parts of the number.

as.integer(y)

## [1] 23 6 100 6

To convert a number to a character use as.character

numchar <- as.character(y.c)  
numchar

## [1] "23.8" "6" "100.01" "6"

You can also coerce a number or a character into a factor.

numfac <- as.factor(y)  
numfac

## [1] 23.8 6 100.01 6   
## Levels: 100.01 23.8 6

charfac <- as.factor(y.c)  
charfac

## [1] 23.8 6 100.01 6   
## Levels: 6 23.8 100.01

There are many other coercion functions in R, a summary of some the most common ones we’ll be using in this course and others:

as.numeric # Coerce to numeric  
as.double # Coerce to double  
as.integer # Coerce to integer  
as.character # Coerce to character  
as.logical # Coerce to Boolean (logical: TRUE | FALSE)  
as.factor # Coerce to factor  
as.Date # Coerce to date  
as.data.frame # Coerce to data frame  
as.list # Coerce to list

**⚠ Practice 3.2** Create the following data frame from scratch. This short experiment aims at testing the efficiency of a diet made of bubble teas: three cups a day for a week. Weight of subjects is provided before and after this ‘bubble tea’ diet (**the data are fake**). Reformat this data frame to obtain the weight as a **double** into one column and the time of measurement as a factor with two levels before\_diet and after\_diet. You will store this **data frame** as the first element of a list called **BUBBLE\_DIET**. The second element of this list will be another list called **WEIGHT\_LOSS**, storing three elements: [1] a **vector of character** extracting the row names of the table previously created; [2] a **numeric vector** (double) indicating the weight loss (in %) of each subject (can be positive or negative); [3] a combination of these two elements in a **table with two columns**: subject and weight\_loss. The third element of the list **BUBBLE\_DIET** will be any message saying how much you enjoy manipulating data in R.You will push both your *.Rmd* and *.html* files into a public repository available from your Github account.You will share with me be email [[vianneydenis@g.ntu.edu.tw](mailto:vianneydenis@g.ntu.edu.tw)] the address (URL) of this repository (such as <https://github.com/vianneydenis/OCEAN5098.git>) **before next Monday** in order for me to check your work. The **title of your email**  should be `Practice 3.2 (your name: your student no.). ENJOY ;)