Hands-on Introduction to R*

Lecture 4: Heterogeneous vectors (Lists & Dataframes) and IO

Wim R.M. Cardoen

Last updated: 10/21/2022 @ 13:21:25

Contents

In the first part of this section, two kinds[1] of **heterogeneous** vectors will be discussed:

- lists
- data frames (& tibbles)

Input-output (IO) in R forms the subject of the latter part.

1 R Lists

A `list` is a vector that **may** contain one or more **components**.
The components can be **heterogeneous** objects (atomic types, functions, lists[2], ...).

Under the hood, the list is implemented as a vector of pointers to its top-level components.
The list's length equals the number of top-level components.

1.1 Creation of a list

An R list can be created in several ways:

- using the **list()** function (most common)
- via the **vector()** function
- via a cast using the **as.list()** function

1.1.1 Examples

- use of the **list()** function

```
# Creating an empty list
x1 <- list()
str(x1)
```

```
 list()
```

```
typeof(x1):list   class(x1):list   length(x1):0
```

```
# A more realistic list
x2 <- list(1:10, c("hello","world"),
           3+4i, matrix(data=1:6,nrow=2,ncol=3,byrow=TRUE))
str(x2)
```

```
List of 4
 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ : chr [1:2] "hello" "world"
 $ : cplx 3+4i
 $ : int [1:2, 1:3] 1 4 2 5 3 6
```

```
typeof(x2):list   class(x2):list   length(x2):4
```

[1] R also has the pairlist. This topic will not be discussed in this section. People interested in this subject, should have a look at R-internals.
[2] Due to this feature, they are also called `recursive` vectors.

```r
# Using existing names
x3 <- list(x=1, y=2, str1="hello", str2="world", vec=1:5)
str(x3)
```

```
List of 5
 $ x   : num 1
 $ y   : num 2
 $ str1: chr "hello"
 $ str2: chr "world"
 $ vec : int [1:5] 1 2 3 4 5
```

```r
# Applying names to a list
x4 <- list(matrix(data=1:4,nrow=2,ncol=2), c(T,F,T,T), "hello")
names(x4) <- c("mymat","mybool","mystr")
str(x4)
```

```
List of 3
 $ mymat : int [1:2, 1:2] 1 2 3 4
 $ mybool: logi [1:4] TRUE FALSE TRUE TRUE
 $ mystr : chr "hello"
```

- use **vector()** function:

  Allows to create/allocate an empty vector of a certain length.

```r
# Allocate a vector of length 5
x5 <- vector(mode="list", length=5)
str(x5)
```

```
List of 5
 $ : NULL
 $ : NULL
 $ : NULL
 $ : NULL
 $ : NULL
```

- using the **as.list()** function

```r
x6 <- as.list(matrix(5:10,nrow=2))
str(x6)
```

```
List of 6
 $ : int 5
 $ : int 6
 $ : int 7
 $ : int 8
 $ : int 9
 $ : int 10
```

Note: The 'inverse' operation is **unlist()**

```
x7 <- unlist(x6)
str(x7)
```

```
 int [1:6] 5 6 7 8 9 10
```

1.2 Accessor operators [ ], [[ ]], $ in R.

1.2.1 General statements

The operator [[i]] selects **only one component** (in cases of lists) or **only one element** in case of homogeneous vectors.

The operator [ ] allows to select **one or more components** (in the case of lists) or **one or more elements** in the case of homogeneous vectors.

The **$** operator can **only** be used for **generic/recursive vectors**.
If you use the **$** operator to other objects you will obtain an **error**.
The **$** operator can **only** be followed by a string or a non-computable index.

1.2.2 Homogenous vectors

In praxi, for homogeneous vectors there is hardly any difference between [[ ]] and [ ] **except** that [[ ]] does **NOT** allow to select more than **one** element.

Note: The operator [[ ]] can be used as a tool of defensive programming.

1.2.2.1 Examples

```
a <- seq(from=1,to=30,by=3)
a
```

```
 [1]  1  4  7 10 13 16 19 22 25 28
```

```
# Extraction of ONE element
cat(sprintf(" a[[2]] : %d\n", a[[2]]))
```

```
 a[[2]] : 4
```

```
cat(sprintf(" a[2]   : %d\n", a[2]))
```

```
 a[2]   : 4
```

```
# Extraction of MORE than 1 element using [[]] => ERROR
a[[c(2,3)]]
```

```
Error in a[[c(2, 3)]]: attempt to select more than one element in vectorIndex
```

but:

```
# Extraction of MORE than 1 element using [] => OK
a[c(2,3)]
```

```
[1] 4 7
```

1.2.3 Heterogeneous vectors (i.e. lists and derived classes)

We stated earlier that the operator [[ ]] allows to select **only one** component.
It also means that this operator selects **the component as is** (matrix, list, function,...).

The operator [ ] allows to select more than **one** component.
Therefore, in order to return potentially heterogeneous components it **always** returns a `list` even if only one component were to be returned.

1.2.3.1 Examples

```
str(x2)
```

```
List of 4
 $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ : chr [1:2] "hello" "world"
 $ : cplx 3+4i
 $ : int [1:2, 1:3] 1 4 2 5 3 6
```

```
# Selection using [[]]
x24 <- x2[[4]]
x24
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```
class(x24)
```

```
[1] "matrix" "array"
```

```
typeof(x24)
```

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

```
length(x24)
```

```
[1] 6
```

```
# Selection using []
x24 <- x2[4]
x24
```

```
[[1]]
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```
class(x24)
```

```
[1] "list"
```

```
typeof(x24)
```

```
[1] "list"
```

```
length(x24)
```

```
[1] 1
```

```
# Select third el. of the FIRST component
x13 <- x2[[1]][3]
x13
```

```
[1] 3
```

which is the same as:

```
v1 <- x2[[1]]
v1[3]
```

```
[1] 3
```

Heterogeneous vectors are also known as **recursive/generic** vectors,
as can be seen in the following example:

```
# A more advanced 'recursive' example
v <- list(v1=1:4,
          lst1=list(a=3, b=2, c=list(x=5,y=7, v2=seq(from=7,to=12))))
```

```
# Extracting the component as a homogeneous vector
v[[2]][[3]][[3]]
```

```
[1]  7  8  9 10 11 12
```

```
class(v[[2]][[3]][[3]])
```

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

```
# Extracting as a list
v[[2]][[3]][3]
```

```
$v2
[1]  7  8  9 10 11 12
```

```
class(v[[2]][[3]][3])
```

```
[1] "list"
```

We can extract the same data using names if available:

```
v$lst1$c$v2
```

```
[1]  7  8  9 10 11 12
```

```
# List of function objects
lstfunc <- list(cube=function(x){x**3},
                quartic=function(x){x**4})
lstfunc$cube(5)
```

```
[1] 125
```

```
lstfunc$quartic(5)
```

```
[1] 625
```

1.3 Modifying lists

- **Removal/deletion** of components:

  Set the list element which refers to the component to **NULL**.
  The list will be **automatically** re-indexed and its length adjusted.

  ```
  mylst1 <- list(a=1:10, b=seq(1,5), matrix(1:10, nrow=2),
                 "hello", "world", 1:5 )
  str(mylst1)
  ```

  ```
  List of 6
   $ a: int [1:10] 1 2 3 4 5 6 7 8 9 10
   $ b: int [1:5] 1 2 3 4 5
   $  : int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
   $  : chr "hello"
   $  : chr "world"
   $  : int [1:5] 1 2 3 4 5
  ```

  ```
  # Removal of the 5th component
  mylst1[[5]] <- NULL
  str(mylst1)
  ```

  ```
  List of 5
  ```

```
 $ a: int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ b: int [1:5] 1 2 3 4 5
 $  : int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
 $  : chr "hello"
 $  : int [1:5] 1 2 3 4 5
```

- **Appending** an object:

  Assign the object (`obj`) to the list element with index `length(lst)+1`

```
# Creation of a list mylst2
mylst2 <- list( 1:5, 'a' , 'b')
str(mylst2)
```

```
List of 3
 $ : int [1:5] 1 2 3 4 5
 $ : chr "a"
 $ : chr "b"
```

```
# Appending a Boolean vector to the existing list mylst2
mylst2[[length(mylst2)+1]] <- c(T,F,T)
str(mylst2)
```

```
List of 4
 $ : int [1:5] 1 2 3 4 5
 $ : chr "a"
 $ : chr "b"
 $ : logi [1:3] TRUE FALSE TRUE
```

  If you set the index to a number which is **larger** than `length(lst) +1`
  all the new intermittent list elements will be set to `NULL`.
  You can get rid of these additional `NULL` values by **subsequently** deleting them.

```
# Insert a component at an index > length(mylst2)+1
# -> we will get some intermittent NULL values.
mylst2[[7]] <- "value"
str(mylst2)
```

```
List of 7
 $ : int [1:5] 1 2 3 4 5
 $ : chr "a"
 $ : chr "b"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : NULL
 $ : NULL
 $ : chr "value"
```

```
# Delete the NULL values! Start from the end!
mylst2[[6]] <- NULL
```

```
mylst2[[5]] <- NULL
str(mylst2)
```

```
List of 5
 $ : int [1:5] 1 2 3 4 5
 $ : chr "a"
 $ : chr "b"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : chr "value"
```

- **Insertion** of new components

  Create a new vector containing three parts:

  - the 'left' sublist
  - the new components
  - the 'right' sublist

```
str(mylst2)
```

```
List of 5
 $ : int [1:5] 1 2 3 4 5
 $ : chr "a"
 $ : chr "b"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : chr "value"
```

```
# Add new component at index 4
newlst2 <- c(mylst2[1:3], "NEW", mylst2[4:length(mylst2)])
str(newlst2)
```

```
List of 6
 $ : int [1:5] 1 2 3 4 5
 $ : chr "a"
 $ : chr "b"
 $ : chr "NEW"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : chr "value"
```

1.4 Functions: return multiple objects

If a function needs to return **multiple objects** a list **must** be used.

1.4.1 Example

```
func01 <- function(n)
{
   x <- n*(n+1)/2
   y <- cbind(1:n,(1:n)^2)
```

```
   return(list('x'=x,'y'=y))
}
n <- 8
res <- func01(n)
```

```
res$x
```

```
[1] 36
```

```
res$y
```

```
     [,1] [,2]
[1,]    1    1
[2,]    2    4
[3,]    3    9
[4,]    4   16
[5,]    5   25
[6,]    6   36
[7,]    7   49
[8,]    8   64
```

1.5 Exercises

- Let's consider the following list:

```
lstex1 <- list(
               list(
                    seq(from=4,to=40,by=5),
                    "hello world",
                    list(
                         matrix(100:119,nrow=5),
                         "bye",
                         c(7,13,17),
                         list(451,-1,-17)
                        )
                   )
              )
```

  Extract the following data from `mylistex1`:

  - the elements 7 13 as a vector.
  - the second column of `matrix(100:119,nrow=5)` as a matrix.
  - -1 as a scalar.
  - -1 as a list.
  - **all** numerical values into a vector. (Hint:**unlist()**)

- Create the following list:

```
lstex2 <- list(1:10,
               matrix(seq(from=1,to=20), nrow=4),
               5+3i,
               list('a','d','b'),
               "UoU",
               c(T,F,T,T,T),
               "Hello"
              )
```

  Perform some operations (deletions, insertions, modifications) on `lstex2` such that `lstex2` takes on the following form:

```
lstex2 <- list( matrix(seq(from=1,to=20), nrow=4),
               "Hello UoU",
               list('a','b','c','d'),
               c(T,F,T,T,T)
              )
```

- Write the function `countOcc(content)` which returns

  - the occurrence of each letter in the string `content`.
  - the occurrence of each non-letter character in the string `content`.

  If we use the text of the First Amendment of the Bill of Rights[3] as argument for `countOcc`,

```
# Split the text to make it readable
l1 <- "Congress shall make no law respecting an establishment of religion,"
l2 <- "or prohibiting the free exercise thereof;"
l3 <- "or abridging the freedom of speech, or of the press;"
```

[3]The text was obtained from *https://www.law.cornell.edu/constitution/first_amendment*

```
  l4 <- "or the right of the people peaceably to assemble,"
  l5 <- "and to petition the government for a redress of grievances."
  # Glue the chunks {l1,l2,l3,l4,l5} together
  firstamend <- paste(l1, l2, l3, l4, l5, sep=" ")
```

we obtain the following output:

```
res <- countOccurrence(firstamend)
res
```

```
$countAlpha
 a  b  c  d  e  f  g  h  i  k  l  m  n  o  p  r  s  t  v  w  x  y
12  5  6  4 38  9  9 12 14  1  8  5 13 21  8 20 15 17  2  1  1  1

$countNonAlpha
    ,  ;  .
44  3  2  1
```

Note/hints:

- We don't distinguish lowercase letters from uppercase letters.
- **strsplit()** : splits a string into its characters.
- **tolower()** : converts letters to their lower case counterparts.
- **unique()** : extracts the unique elements of a vector.
- R has a built-in vector **letters**.

2 R Dataframes

A `date frame` is a `list` with three `attributes`:

- **names** : component names
- **row.names** : row names
- **class**: **data.frame**

From the above, we can infer that a data frame has the the **same** row names for each component (columns). The components of a data frame can be vectors, factors, numerical matrices, lists or other data frames.

In praxi, a **data frame** can be conceptualized as a **generalized (i.e. heterogeneous) matrix/table where each column has its own type but where each column has the same number of rows**.

2.1 Creating a data frame

- use of the **data.frame** function.
- some IO functions generate a data frame when they read a file. e.g. **read.table()**, **read.csv()**.

2.1.1 Examples

```
# Creation of a data frame using the data.frame function:
vec1 <- c("Smith", "Jensen", "McFall", "Johnson",
          "Brown", "Williams", "Wilson", "Roberts")
vec2 <- c(4, 3, 0 , 6, 2, 0, 5, 1)
vec3 <- c(100000, 80000, 140000, 120000, 60000, 30000, 170000, 100000)
vec4 <- c("Salt Lake", "Provo", "Park City", "Provo",
          "Logan", "Ogden", "Provo", "Salt Lake")
df1  <- data.frame(family=vec1, nchildren=vec2, income=vec3, location=vec4)
df1
```

```
# A tibble: 8 x 4
  family   nchildren income location
  <chr>        <dbl>  <dbl> <chr>
1 Smith            4 100000 Salt Lake
2 Jensen           3  80000 Provo
3 McFall           0 140000 Park City
4 Johnson          6 120000 Provo
5 Brown            2  60000 Logan
6 Williams         0  30000 Ogden
7 Wilson           5 170000 Provo
8 Roberts          1 100000 Salt Lake
```

Note:

- Since `R 4.0.0`, the default value for the argument `stringsAsFactors` in the function **data.frame()** has been set to `FALSE`.
- The functions **head()** and **tail()** display the first n, respectively last n lines (default: 6) of a data frame.

```r
# Creation of a data frame after reading a data file
df2 <- read.table(file="./datafiles/seaice.txt",header=TRUE)
cat(sprintf("Length(df2) :%d\n", length(df2)))
```

```
Length(df2) :2
```

```r
cat(sprintf("Dim. of df2 :\n"))
```

```
Dim. of df2 :
```

```r
dim(df2)
```

```
[1] 37  2
```

2.2 Accessing elements of the data frame

As we saw earlier, a data frame is a list with some extra attributes.
It has also features of matrix i.e. a set of columns with the same number of rows for each column.

Therefore, the elements of a data frame can be accessed in **two** different ways:

- like a list
- like a matrix

2.3 Modifying the data frame

2.4 attach and detach

3 Input-Output (IO)

3.1 Functionality in `Base R`

3.2 Other options:

- library `readr`
  - supports a lot of formats (csv, tcsv, delim, ...)
  - allows column specification
  - faster than `Base R`'s read/write operations
  - uses a `tibble` instead of a `data frame`.
  - for more info: `R for Data Science` - Chapter 11.Data import
- library `data.table`
  - very fast IO: optimal for large read (**fread()**) and write (**fwrite()**) operations
  - memory efficient
  - low-level parallelism (use of multiple CPU threads)