Lecture 5. Lists

R and Data Visualization

BIG2006, Hanyang University, Fall 2022

Lists

In contrast to a vector, in which all elements must be of the same mode, R's `list` structure can combine objects of different types.

The list plays a central role in R, forming the basis for data frames, object-oriented programming, and so on.

Note: An R list is similar to a Python dictionary or a C struct.

Creating Lists

Ordinary vectors, those of the type we've been using so far

- `atomic` vectors
- their components cannot be broken down into smaller components

List (also a vector): `recursive` vectors.

Employee database

- We want to store the name (character), salary (numeric), and a Boolean indicating union membership (logical).

```
j <- list(name="June", salary=76500, union=T)
j # the component names (tags) are optional
```

```
## $name
## [1] "June"
##
## $salary
## [1] 76500
##
## $union
## [1] TRUE
```

Since lists are vectors, they can be created via `vector()`,

```
z <- vector(mode="list")
z[["salary"]] <- 100000
z
```

```
## $salary
## [1] 1e+05
```

General List Operations

List Indexing

- To access a list component

```
j$salary
```

```
## [1] 76500
```

```
j[["salary"]]  # double bracket!
```

```
## [1] 76500
```

```
j[[2]]
```

```
## [1] 76500
```

- Single-bracket and double-bracket indexing
    - Both access list elements in vector-index fashion
    - Single brackets [ ]: refer a sublist of the original
    - Double brackets [[ ]]: refer a single component

```r
j[1:2]
```

```
## $name
## [1] "June"
##
## $salary
## [1] 76500
```

```r
c(j[[1]], j[[2]])
```

```
## [1] "June"  "76500"
```

```
class(j[2])
```

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

```
str(j[2])
```

```
## List of 1
##  $ salary: num 76500
```

```
class(j[[2]])
```

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

```
str(j[[2]])
```

```
##  num 76500
```

Adding and Deleting List Elements

- New components can be added *after* a list is created.

```
z <- list(a="abc", b=12)
z$c <- "sailing" # add a c component
z
```

```
## $a
## [1] "abc"
##
## $b
## [1] 12
##
## $c
## [1] "sailing"
```

- Adding components can also be done via a vector index.

```r
z[[4]] <- 35
z[5:6] <- c(TRUE, FALSE)
z[3:6]
```

```
## $c
## [1] "sailing"
##
## [[2]]
## [1] 35
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] FALSE
```

- You can delete a list component by setting it to NULL.

```r
z$b <- NULL
z[5] <- NULL
z # the indices of the elements after it moved up by 1
```

```
## $a
## [1] "abc"
##
## $c
## [1] "sailing"
##
## [[3]]
## [1] 35
##
## [[4]]
## [1] TRUE
```

- You can also concatenate lists.

```
c(list("Jeong", 500, T), list(30))
```

```
## [[1]]
## [1] "Jeong"
##
## [[2]]
## [1] 500
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 30
```

Getting the Size of a List

- Since a list is a vector, the number of component can be obtained via `length`().

```
length(j)
```

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

Example: Text Concordance

- Web search and other types of textual data mining are of great interest today.
- We will write a `findwords`() function that will determine which words are in a text file and compile a list of the locations of each word's occurrences in the text.

- Suppose our input file, `testconcord.txt`, has the following contents.

```
the here means that the first item in this line of output is
item in this case our output consists of only one line and one
item so this is redundant but this notation helps to read
voluminous output that consists of many items spread over many
lines for example if there were two rows of output with six
items per row the second row would be labeled
```

Note: The word *item* has locations (word positions) 7, 14, and 27 in the file.

```r
findwords <- function(tf) {
  # read in the words from the file,
  #into a vector of mode character
  txt <- scan(tf,"")
  wl <- list()
  for (i in 1:length(txt)) {
    wrd <- txt[i] # ith word in input file
    wl[[wrd]] <- c(wl[[wrd]],i)
  }
  return(wl)
}
```

Note: We read in the words of the file by calling `scan()`. `txt` will now be a vector of string: one string per instance of a word in the file.

- Here is what txt looks like after the read:

```
> txt
[1] "the" "here" "means" "that" "the"
[6] "first" "item" "in" "this" "line"
[11] "of" "output" "is" "item" "in"
[16] "this" "case" "our" "output" "consists"
[21] "of" "only" "one" "line" "and"
[26] "one" "item" "so" "this" "is"
[31] "redundant" "but" "this" "notation" "helps"
[36] "to" "read" "voluminous" "output" "that"
[41] "consists" "of" "many" "items" "spread"
[46] "over" "many" "lines" "for" "example"
[51] "if" "there" "were" "two" "rows"
[56] "of" "output" "with" "six" "items"
[61] "per" "row" "the" "second" "row"
[66] "would" "be" "labeled"
```

- Here is an excerpt from the list that is returned when our function `findwords()` is called on this file:

```
> findwords("testconcorda.txt")
Read 68 items
$the
[1] 1 5 63
$here
[1] 2
$means
[1] 3
$that
[1] 4 40
...
```

Accessing List Components and Values

names(): to obtain the component names

unlist(): to obtain the values (character strings)

```
names(j)
```

```
## [1] "name"   "salary" "union"
```

```
ulj <- unlist(j)
ulj # the return value is a vector of character strings
```

```
##    name  salary   union
##  "June" "76500"  "TRUE"
```

```
class(ulj)
```

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

If we were to start with numbers, we would get numbers.

```
z <- list(a=5, b=12, c=13)
y <- unlist(z)
class(y)
```

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

```
y
```

```
##  a  b  c
##  5 12 13
```

What about a mixed case?

```r
w <- list(a=5, b="xyz")
wu <- unlist(w)
class(wu)
```

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

```r
wu
```

```
##     a     b
##   "5" "xyz"
```

R chose the least common denominator: character strings. R's help for `unlist()` states:

```
Where possible the list components are coerced to a
common mode during the unlisting, and so the result
often ends up as a character vector. Vectors will be
coerced to the highest type of the components in the
hierarchy NULL < raw < logical < integer < real <
complex < character < list < expression: pairlists
are treated as lists.
```

`wu` is a vector (not a list), but R did give each of the elements a name. We can remove them by setting their name to NULL and

using `unname()`.

```r
names(wu) <- NULL
wu
```

```
## [1] "5"   "xyz"
```

```r
wun <- unname(wu)
wun
```

```
## [1] "5"   "xyz"
```

Applying Functions to Lists

Using the `lapply()` and `sapply()` Functions

- `lapply()` works like the matrix `apply()`
  - calling the specified function on each component of a list (or vector coerced to a list)
  - returning another list

```
lapply(list(1:3, 25:29), median)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 27
```

- sapply(): the list returned by lapply() could be simplified to a vector or matrix.

```
sapply(list(1:3, 25:29), median)
```

```
## [1]  2 27
```

Example: Abalone Data

- We want to know the indices of the observations that were male, female, and infant.

```r
g <- c("M","F","F","I","M","M","F")
lapply(c("M","F","I"), function(gender) which(g==gender))
```

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

Recursive Lists

Lists can be recursive: you can have lists within lists.

```r
b <- list(u = 5)
c <- list(w = 13)
a <- list(b,c)
a
```

```
## [[1]]
## [[1]]$u
## [1] 5
##
##
## [[2]]
## [[2]]$w
## [1] 13
```

c() has an optional argument `recursive` that controls whether *flattening* occurs when recursive lists are combined.

```
c(list(a=1,b=2,c=list(d=5,e=9)))
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## $c$d
## [1] 5
##
## $c$e
## [1] 9
```

In the previous case, we obtain a recursive list with the c component of the main list itself being another list.

In the following call, with `recursive` = `TRUE`, we get a single list as a result.

```
c(list(a=1,b=2,c=list(d=5,e=9)), recursive=T)
```

```
##    a    b c.d c.e
##    1    2   5   9
```

```
# only the names look recursive
```

Reference

- Matloff, N. The Art of R Programming: A Tour of Statistical Software Design. No Starch Press. Chapter 4.