List

February 2, 2020

List

- List structure in R can combine objects of **different types**.
- The R list is similar to a Python dictionary or Perl hash.
- C programmers may find list similar to a C struct.

An example: list

Let's consider an employee database.

- For each employee, we want to store the **name**, **salary**, and a **Boolean** indicating union membership
- We have three different modes here: **character**, **numeric**, and **logical**
- **List** is a good choice for storing data
- The entire database might then be a list of lists

Creating list

We could create a list to represent our employee, Joe, this way:

```
> j <- list(name="Joe", salary=55000, union=T)
```

The components in the list j

```
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE
```

Creating list

- The component names, called *tags* in the R literature, such as *salary* are optional
- **Tip**: Names of list components can be abbreviated to whatever extent is possible without causing ambiguity

```
> j$sal
[1] 55000
```

Creating List

Alternatively, you can do this

```
> jalt <- list("Joe", 55000, T)
> jalt
[[1]]
[1] "Joe"

[[2]]
[1] 55000

[[3]]
[1] TRUE
```

- However, it is generally considered clearer and less error-prone to use names instead of numeric indices.

Creating List

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

```
> z <- vector(mode="list")
> z[["abc"]] <- 3

> z
$abc
[1] 3
```

List Operations

You can access a list component in several different ways:

```
> j$salary
[1] 55000

> j[["salary"]]
[1] 55000

#You can refer to list components by their numerical indices,
    treatingthe list as a vector.
#However, note that in this case, you need to use double brackets
     instead of single ones.
> j[[2]]
[1] 55000
```

List

- Both single-bracket and double-bracket indexing access list elements in vector-index fashion.
- But there is an important difference from ordinary vector indexing.
- If single brackets [ ] are used, the result is another list, which is a sublist of the original.

```
> j[1:2]
$name
[1] "Joe"
$salary
[1] 55000

> j2 <- j[2]
> j2
$salary
[1] 55000
> class(j2)
[1] "list"
> str(j2)
List of 1
$ salary: num 55000
```

Adding List Elements

New components can be added after a list is created

```
> z <- list(a="abc",b=12)
> z
$a
[1] "abc"

$b
[1] 12
```

```
# add a c component
> z$c <- "sailing"
```

Question: what is z now?

Adding List Elements

New components has been added to the list

```
> z
$a
[1] "abc"

$b
[1] 12

$c
[1] "sailing"
```

Adding List Elements

Adding components can also be done via a vector index

```
> z[[4]] <- 28
> z[5:7] <- c(FALSE,TRUE,TRUE)
```

Question: What is z now?

Adding List Elements

z is updated with new components

```
> z
$a
[1] "abc"

$b
[1] 12

$c
[1] "sailing"

[[4]]
[1] 28

[[5]]
[1] FALSE

[[6]]
[1] TRUE

[[7]]
[1] TRUE
```

Delete List Elements

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

```
> z$b <- NULL

> z
$a
[1] "abc"

$c
[1] "sailing"

[[3]]
[1] 28

[[4]]
[1] FALSE

[[5]]
[1] TRUE

[[6]]
[1] TRUE
```

Concatenate lists

You can also concatenate lists.

```
> c(list("Joe", 55000, T),list(5))
[[1]]
[1] "Joe"

[[2]]
[1] 55000

[[3]]
[1] TRUE

[[4]]
[1] 5
```

Getting the Size of a List

Since a list is a vector, you can obtain the number of components in a list via **length()**.

```
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

> length(j)
[1] 3
```

Accessing List Components and Values

- If the components in a list do have tags, such as *name*, *salary*, and *union* for j , you can obtain them via R function **names()**:

```
> names(j)
[1] "name" "salary" "union"
```

- To obtain the values, use function **unlist()**:

```
> ulj <- unlist(j)
> ulj
name salary union
"Joe" "55000" "TRUE"

#Data type of ulj a vector of character strings.
> class(ulj)
[1] "character"
```

Note that the element names in this vector come from the components in the original list

Accessing List Components and Values

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
```

How about a mixed case?

Accessing List Components and Values

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

> class(wu)
[1] "character"

# 5 in the list now is a character
> wu
a b
"5" "xyz"
```

Accessing List Components and Values

- The list components are coerced to a common mode during the unlisting
- Vectors will be coerced to the highest type of the components in the hierarchy NULL $<$ raw $<$ logical $<$ integer $<$ real $<$ complex $<$ character $<$ list

Remove names

Function **names()** give each of the elements a name. We can remove them by setting their names to NULL,

```
> names (wu)
[1] "a" "b"

> names(wu) <- NULL
> wu
[1] "5" "xyz"
```

You can also remove the elements ' names directly using function **unname()**, as follows:

```
> wun <- unname(wu)
> wun
[1] "5" "xyz"
```

Applying Functions to Lists: lapply()

- Two functions are handy for applying functions to lists: **lapply()** and **sapply()**.
- The function **lapply()** (for **list** apply) works like the **matrix apply()** function.
- It calls the specified function on each component of a list and returning another list. Here is an example:

```
#R applied median() to 1:3 and to 25:29, returning a list
    consisting of 2 and 27.
> lapply(list(1:3,25:29),median)
[[1]]
[1] 2

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

Applying Functions to Lists: sapply()

- In some cases, the list returned by lapply() could be simplified to a vector or matrix.
- This is exactly what **sapply()** (for *simplified [l]apply*) does.

```
#sapply returns a vector
> sapply(list(1:3,25:29),median)
[1] 2 27
```

An example

Let's use the **lapply()** function in our abalone gender example

```
# g is a vector of character
g <- c("M","F","F","I","M","M","F")
```

Question: Find the index of M, F and I in the vector using function **lapply**.

An example

Let's use the **lapply()** function in our abalone gender example

```
> lapply(c("M","F","I"),function(gender) which(g==gender))
[[1]]
[1] 1 5 6

[[2]]
[1] 2 3 7

[[3]]
[1] 4
```

- The **lapply()** function expects its **first argument** to be a list.
- Here we have a vector, **lapply()** will coerce that vector to a list form.
- Also, **lapply()** expects its **second argument** to be a function. This could be the name of a function, or the actual code, as we have here.