Lists

Bok, Jong Soon
javaexpert@nate.com
www.javaexpert.info

Lists

- A vector, all elements must be of the same mode.
- A list structure can combine objects of different types.
- For those familiar with Python, an R list is similar to a `Python dictionary` or, for that matter, a `Perl hash`.
- C programmers may find it similar to a `C struct`.
- The list plays a central role in R, forming the basis for data frames, object-oriented programming, and so on.

Creating Lists

- Technically, a `list` is a `vector`.
- Ordinary vectors are termed atomic vectors, since their components cannot be broken down into smaller components.
- In contrast, lists are referred to as recursive vectors.
- Let's consider an employee database.
- For each employee, we wish to store the name, salary, and a Boolean indicating union membership.
- Since we have three different modes here : `character`, `numeric`, and `logical`.

Creating Lists (Cont.)

- We could create a list to represent our employee, Joe.

Console C:/R Home/

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

$salary
[1] 55000

$union
[1] TRUE
```

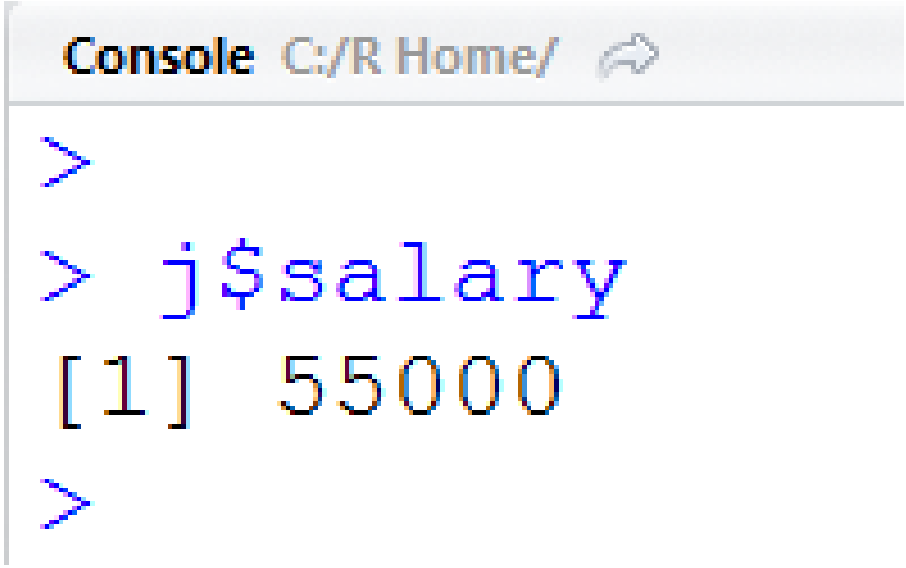
Creating Lists (Cont.)

```
Console C:/R Home/
>
> jalt <- list("Joe", 55000, T)
>
> jalt
[[1]]
[1] "Joe"

[[2]]
[1] 55000

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

- Actually, the component names : called *tags* in the R literature : such as **salary** are optional.

Creating Lists (Cont.)

- However, it is generally considered clearer and less error-prone to use names instead of numeric indices.
- Names of list components can be abbreviated to whatever extent is possible without causing ambiguity.



Creating Lists (Cont.)

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

```
Console C:/R Home/
> z <- vector(mode="list")
> z[["abc"]] <- 3
> z
$abc
[1] 3
```

List Indexing

```
Console C:/R Home/
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

> j$salary
[1] 55000
>
> j[["salary"]]
[1] 55000
>
> j[[2]]
[1] 55000
```

- You can access a list component in several different ways.

List Indexing (Cont.)

- We can refer to list components by their numerical indices, treating the list as a vector.
- However, note that in this case, we use double brackets(`[[ ]]`) instead of single ones.
- So, there are three ways to access an individual component `c` of a list `lst` and return it in the data type of `c`:
  - `lst$c`
  - `lst[["c"]]`
  - `lst[[i]]`, where `i` is the index of `c` within `lst`.

List Indexing (Cont.)

- An alternative techniques listed is to use single brackets rather than double brackets.
  - `lst["c"]`
  - `lst[i]`, where `i` is the index of `c` within `lst`.
- Both single-bracket and double-bracket indexing access list elements in vector-index fashion.
- But there is an important difference from ordinary (atomic) vector indexing.
- If single brackets `[ ]` are used, the result is another `list` : a *sublist* of the original.

List Indexing (Cont.)

```
Console C:/R Home/

> 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
```

- The subsetting operation returned another list consisting of the first two components of the original list `j`.

List Indexing (Cont.)

```
Console C:/R Home/
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

> j[[1:2]]
Error in j[[1:2]] : subscript out of bounds
>
> j2a <- j[[2]]
> j2a
[1] 55000
>
> class(j2a)
[1] "numeric"
```

- By contrast, can use double brackets `[[ ]]` for referencing only a single component, with the result having the type of that component.

Adding and Deleting List Elements

```
Console C:/R Home/
> z <- list(a="abc", b=12)
> z
$a
[1] "abc"

$b
[1] 12

> z$c <- "sailing"     # Add a c component
> z
$a
[1] "abc"

$b
[1] 12

$c
[1] "sailing"
```

- The operations of adding and deleting list elements arise in a surprising number of contexts.
- New components can be added *after* a list is created.

Adding and Deleting List Elements (Cont.)

```
Console C:/R Home/
> z[[4]] <- 28
> z[5:7] <- c(FALSE,TRUE,TRUE)
> z
$a
[1] "abc"

$b
[1] 12

$c
[1] "sailing"

[[4]]
[1] 28

[[5]]
[1] FALSE

[[6]]
[1] TRUE

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

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

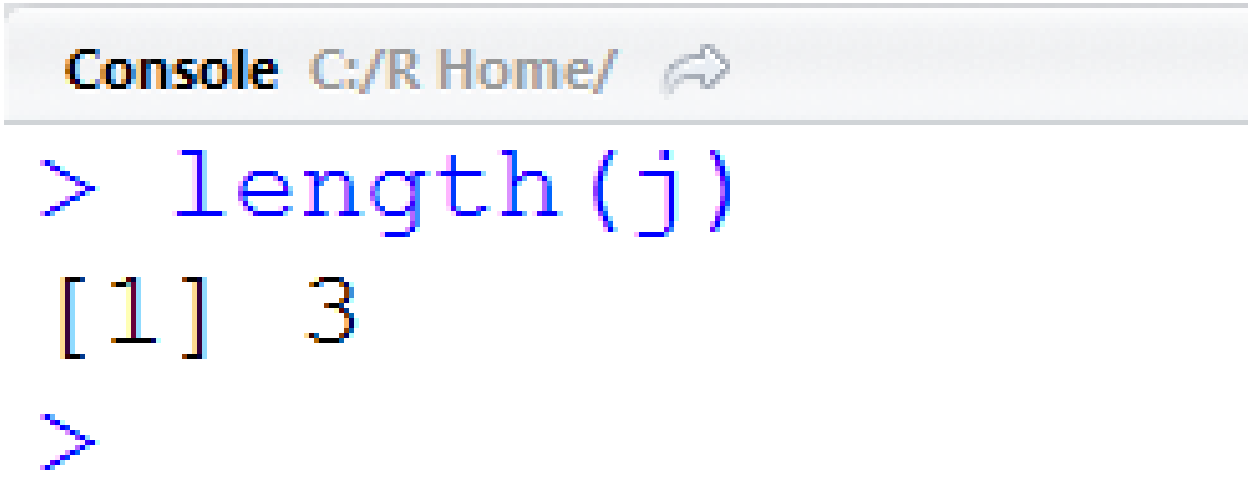
Adding and Deleting List Elements (Cont.)

```
Console C:/R Home/
> z$b <- NULL
> z
$a
[1] "abc"

$c
[1] "sailing"

[[3]]
[1] 28

[[4]]
[1] FALSE

[[5]]
[1] TRUE

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

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

Adding and Deleting List Elements (Cont.)

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



Accessing List Components and Values

```
Console C:/R Home/
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

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

- If the components in a `list` do have *tags*, as is the case with `name`, `salary`, and `union` for `j`, can obtain them via `names()`.

Accessing List Components and Values (Cont.)

```
Console C:/R Home/
> ulj <- unlist(j)
> ulj
   name   salary    union
  "Joe" "55000"  "TRUE"
>
> class(ulj)
[1] "character"
```

- To obtain the values, use **`unlist()`**.
- The return value of **`unlist()`** is a vector, a vector of character strings.
- Note that the element names in this vector come from the components in the original list.

Accessing List Components and Values (Cont.)

- On the other hand, if we were to start with numbers, we would get numbers.

```
Console C:/R Home/
> z <- list(a=5, b=12, c=13)
> y <- unlist(z)
> class(y)
[1] "numeric"
>
> y
 a  b  c
 5 12 13
```

- So the output of **`unlist()`** in this case was a numeric vector.

Accessing List Components and Values (Cont.)

- R chose the least common denominator: character strings.

Console C:/R Home/

```
> w <- list(a=5, b="xyz")
> wu <- unlist(w)
> class(wu)
[1] "character"
>
> wu
  a   b
"5" "xyz"
```

Accessing List Components and Values (Cont.)

- We can remove them by setting their names to **NULL**.

Console C:/R Home/

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

Accessing List Components and Values (Cont.)

- We can also remove the elements' names directly with **unname()**, as follows.

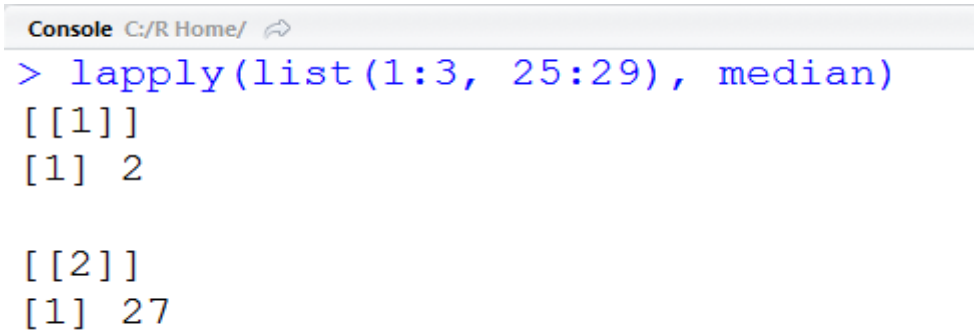
```
> w <- list(a=5, b="xyz")
>
> wu <- unlist(w)
>
> wu
   a      b
 "5"  "xyz"
>
> wun <- unname(wu)
> wun
[1] "5"   "xyz"
```

Applying Functions to Lists

- Two functions are handy for applying functions to lists: `lapply` and `sapply`.

Using the lapply() and sapply() Functions

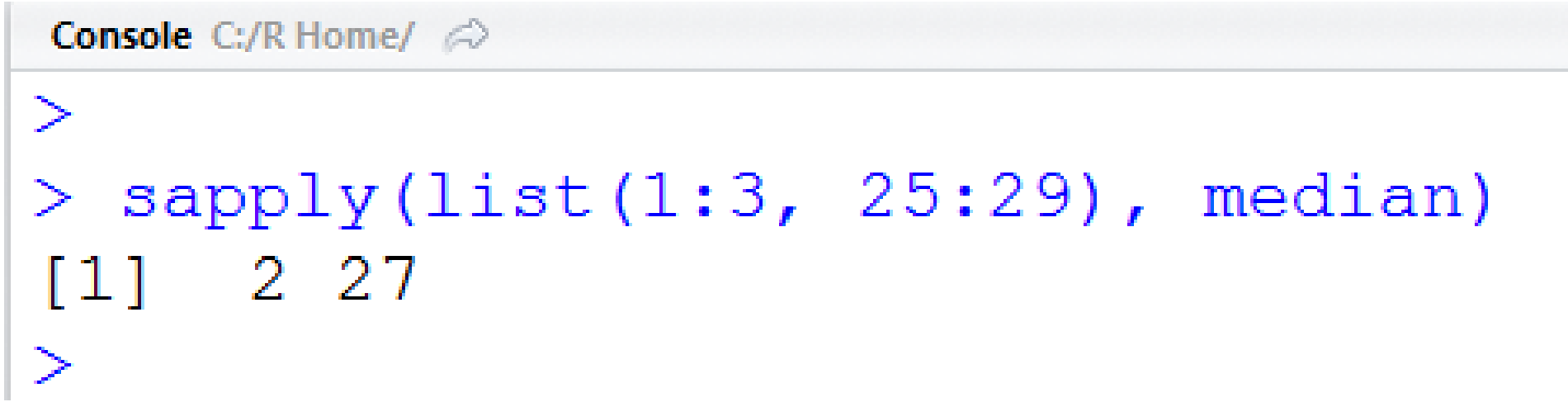
- The function **`lapply()`** (for *list apply*) works like the matrix **`apply()`** function.
- Calls the specified function on each component of a list (or vector coerced to a list) and returning another list.



- R applied **`median()`** to 1:3 and to 25:29, returning a list consisting of 2 and 27.

Using the lapply() and sapply() Functions (Cont.)

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



Recursive Lists

```
Console C:/R Home/
> b <- list(u=5, v=12)
> c <- list(w=13)
> a <- list(b,c)
> a
[[1]]
[[1]]$u
[1] 5

[[1]]$v
[1] 12


[[2]]
[[2]]$w
[1] 13


> length(a)
[1] 2
```

- Lists can be recursive, meaning that you can have lists within lists.
- This code makes a into a two-component list, with each component itself also being a list.

Recursive Lists (Cont.)

```
Console C:/R Home/
> 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


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

- The concatenate function `c()` has an optional argument recursive, which controls whether *flattening* occurs when recursive lists are combined.

Recursive Lists (Cont.)

- In the first case, we accepted the default value of recursive.
- That is **FALSE**.
- Which is obtained a recursive list, with the c component of the main list itself being another list.
- In the second call, with recursive set to **TRUE**.
- That is a single list as a result.
- Which is only the names look recursive.
- It's odd that setting recursive to **TRUE** gives a *nonrecursive* list.