lists in R

Vectors and matrices contain elements of the same class (mode). Conversely, Lists in R can contain structures of different object types. Lists form the basis for data frames in object-oriented programming.

A list is technically a vector. Ordinary vectors (atomic vectors) cannot be broken down into smaller components; therefore, lists are referred to as ***recursive*** vectors.

For example, the following list represents an employee database with three classes—character, numeric, and a logical. Lists can be constructed as lists of lists, or other types of lists like data frames (discussed later):

```
> j <- list( name = "joe", salary = 55000, union = TRUE)          #named components
> j
$name
[1] "joe"

$salary
[1] 55000

$union
[1] TRUE

> jalt <- list("Joe", 55000, TRUE)                         #numerically indexed components
> jalt
[[1]]
[1] "Joe"

[[2]]
[1] 55000

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

As illustrated above, lists can be constructed with names assigned with the **name =** argument; alternatively, the lists can be indexed by numbers. It is best practice to use names for features to support referencing.

Considering the lists are vectors, lists can be created via the **vector( )** function:

```
> z <- vector(mode = "list")
> z[["abc"]] <- 3
> z
$abc
[1] 3
```

general list operations in R

List components can be accessed by treating the list as a vector with numerical indices, using double brackets **[[ ]]**.

```
> j$salary
[1] 55000

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

> j[[2]]
[1] 55000
```

Concretely, the three ways to access a list component **c** of a list **lst** to return the data type of **c**.

… lst$c

… lst[["c"]]

… lst[[i]], where **I** is the index of **c** within **lst**

An important property in accessing components of a list lies in the **data being returned in type c**.
Alternatively, single brackets **[ ]** can be used to access list components, opposed to double brackets **[[ ]]**:
Both options access lists in a vector-index fashion; the difference exists in ***atomic*** (ordinary) vector indexing.

```
> j[1:2]           #subset the first two components of list j with single brackets [ ]
$name
[1] "joe"

$salary
[1] 55000

> j2 <- j[2]       #subset the second component of list j with single brackets [ ]
> j2
$salary
[1] 55000

> class(j2)        #print the class of j2, a subset of list j with single brackets [ ]
[1] "list"

> str(j2)          #print the structure of list j2, confirming the list assignment
List of 1
 $ salary: num 55000
```

The use of single brackets **[ ]** results in another list—a sublist of the original as illustrated above.
Conversely, the above illustration accessed with double brackets **[[ ]]** maintains the **type** of each component:

```
> j[[1:2]]
Error in j[[1:2]] : subscript out of bounds

> j2a <- j[[2]]           #subset of list j2 assigned to j2a returns the component
> j2a
[1] 55000

> class(j2a)              #the class of j2a retains the numeric type
[1] "numeric"

> str(j2a)
 num 55000
```

adding and deleting list elements in R

Adding and deleted elements of a list can be performed in many different contexts. For example…

New components can be added ***after*** a list is created:

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

> z$c <- "sailing"                      #add an additional component c
> z
$a
[1] "abc"
$b
[1] 12
$c
[1] "sailing"
```

Components of a list can also be added with a **vector index**:

```
> 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
```

Components of a list can be deleted 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
```

Noting above that the deletion of **z$b** shifted the indices up by 1. Lists can also be **concatenated**:

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

[[2]]
[1] 55000

[[3]]
[1] TRUE

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

Considering a list is a vector, the number of components in a list can be returned with the **length( )** function:

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

accessing list components and values in R

Assuming list components have tags (assigned names), they can be returned with the **names( )** function. Additionally, the values of the list are returned with the **unlist( )** function:

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

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

> class(ulj)
[1] "character"
```

The **unlist( )** function returns a vector of character strings in the illustration above, with the names originating from the original list **j**.

The same assumption applies if the list is created as numbers, returning a vector of numbers:

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

> y
 a  b  c
 5 12 13
```

Applying the above behavior when returning functions, note the output of **mixed classes**:

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

> wu
    a     b
  "5" "xyz"
```

Applying the common denominator coercion to the output of the **unlist( )** function, R applies the highest type of components to result vectors in the hierarchy :

NULL>raw>logical>integer>real>complex>character>list>expression: pairlists are treated as lists.

Although **wu** is a vector and not a list, vector **wu** was still assigned names; of which can be removed.

```
> names(wu) <- NULL    #remove the names of vector wu by setting names to NULL
> wu
[1] "5"   "xyz"

> wun <- unname(wu)    #remove names of vector wu directly with the unname( ) function
> wun
[1] "5"   "xyz"
```

The above preserves the names in vector **wu** for later use; otherwise **wu** could be assigned instead of **wun**.

applying functions to lists in R

the **lapply( )** and **sapply( )** functions are useful for applying functions to lists in R. Similar to the matrix **apply( )** function, the **lapply( )** calls a specific function on each component of a list (or vector coerced to a list) and returns an additional list.

```
> lapply(list(1:3,25:29),median)  #using lapply to call the median function
[[1]]
[1] 2

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

As shown above, the **lapply( )** function calls the median function on all components of the list. In some cases, the list returned above could be simplified to a vector or matrix through the **sapply( )** function.

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

recursive lists in R

Lists in R can be recursive; in the sense that lists can exists within lists.

```
> b <- list(u = 5, v = 12)                  #a consists of a two-component list
> c <- list(w = 13)                         #with each component also being
> a <- list(b,c)                            #its own separate list
> a
[[1]]
[[1]]$u
[1] 5

[[1]]$v
[1] 12

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

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

The concatenate function **c( )** has an optional argument of ***recursive***, controlling whether ***flattening*** occurs when recursive lists are combined. The second example results in a single list; opposed to a recursive list.

```
> c(list(a=1, b=2, c=list(d=5, e=9)))#concatenate lists, default recursive arg = FALSE
$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 = TRUE) #concatenate lists, arg = TRUE

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