Lists, Attributes, & S3

Lecture 04

Dr. Colin Rundel

Generic Vectors

Lists

Lists are the other vector data structure in R, they differ from atomic vectors in that they can contain a heterogeneous collection of R object (e.g. atomic vectors, functions, other lists, etc.).

```r
list("A", c(TRUE,FALSE), (1:4)/2, list(TRUE, 1), function(x) x^2)
```

```
[[1]]
[1] "A"

[[2]]
[1]  TRUE FALSE

[[3]]
[1] 0.5 1.0 1.5 2.0

[[4]]
[[4]][[1]]
[1] TRUE
```

List Structure

Often we want a more compact representation of a complex object, the `str()` function is useful for this, particularly for lists.

```r
str(c(1,2))
```

```
num [1:2] 1 2
```

```r
str(1:100)
```

```
int [1:100] 1 2 3 4 5 6 7 8 9 10 ...
```

```r
str("A")
```

```
chr "A"
```

```r
str( list(
  "A", c(TRUE,FALSE),
  (1:4)/2, list(TRUE, 1),
  sum
) )
```

```
List of 5
 $ : chr "A"
 $ : logi [1:2] TRUE FALSE
 $ : num [1:4] 0.5 1 1.5 2
 $ :List of 2
  ..$ : logi TRUE
  ..$ : num 1
 $ :function (..., na.rm = FALSE)
```

Recursive lists

Lists can contain other lists, meaning they don’t have to be flat

```r
str( list(1, list(2, list(3, 4), 5)) )
```

```
List of 2
 $ : num 1
 $ :List of 3
  ..$ : num 2
  ..$ :List of 2
  .. ..$ : num 3
  .. ..$ : num 4
  ..$ : num 5
```


Because of this, lists become a natural way of representing tree-like structures within R

List Coercion

By default a vector will be coerced to a list (as a list is more general) if needed

```r
str( c(1, list(4, list(6, 7))) )
```

```
List of 3
 $ : num 1
 $ : num 4
 $ :List of 2
  ..$ : num 6
  ..$ : num 7
```

```r
str( list(1, list(4, list(6, 7))) )
```

```
List of 2
 $ : num 1
 $ :List of 2
  ..$ : num 4
  ..$ :List of 2
  .. ..$ : num 6
  .. ..$ : num 7
```

We can coerce a list into an atomic vector using `unlist()` - type coercion rules then apply to determine the resulting type.

```r
unlist(list(1:3, list(4:5, 6)))
```

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

```r
unlist( list(1, list(2, list(3, "Hello"))) )
```

```
[1] "1"     "2"     "3"     "Hello"
```

Named lists

Because of their more complex structure we often want to name the elements of a list (we can also do this with atomic vectors).

This can make accessing list elements more straight forward (and avoids the use of magic numbers)

```r
str(list(A = 1, B = list(C = 2, D = 3)))
```

```
List of 2
 $ A: num 1
 $ B:List of 2
  ..$ C: num 2
  ..$ D: num 3
```

More complex names (i.e. non-valid object names) are allowed but must be quoted,

```r
list("knock knock" = "who's there?")
```

```
$`knock knock`
[1] "who's there?"
```

Exercise 1

Represent the following JSON data as a list in R.

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address":
  {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": 10021
  },
  "phoneNumber":
  [ {
      "type": "home",
      "number": "212 555-1239"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
  } ]
}
```

05:00

NULL Values

NULLs

`NULL` is a special value within R that represents nothing - it always has length zero and a type and mode of `"NULL"` and cannot have any attributes.

```r
NULL
```

```
NULL
```

```r
typeof(NULL)
```

```
[1] "NULL"
```

```r
mode(NULL)
```

```
[1] "NULL"
```

```r
length(NULL)
```

```
[1] 0
```

```r
c()
```

```
NULL
```

```r
c(NULL)
```

```
NULL
```

```r
c(1, NULL, 2)
```

```
[1] 1 2
```

```r
c(NULL, TRUE, "A")
```

```
[1] "TRUE" "A"
```

Note - If you're familiar with SQL, its `NULL` is more like R's `NA`

0-length coercion

0-length length coercion is a special case of length coercion when one of the arguments has length 0.

In this special case the longer vector will have its length coerced to 0.

```r
integer() + 1
```

```
numeric(0)
```

```r
log(numeric())
```

```
numeric(0)
```

```r
logical() | TRUE
```

```
logical(0)
```

```r
character() > "M"
```

```
logical(0)
```

As a `NULL` values always have length 0, this rule will apply (note the types)

```r
NULL + 1
```

```
numeric(0)
```

```r
NULL | TRUE
```

```
logical(0)
```

```r
NULL > "M"
```

```
logical(0)
```

```r
log(NULL)
```

```
Error in log(NULL): non-numeric argument to
mathematical function
```

NULLs and comparison

Given the previous issue, comparisons and conditionals with NULLs can be problematic.

```r
x = NULL
```

```r
if (x > 0)
  print("Hello")
```

```
Error in if (x > 0) print("Hello"): argument is of length zero
```

```r
if (!is.null(x) & (x > 0))
  print("Hello")
```

```
Error in if (!is.null(x) & (x > 0)) print("Hello"): argument is of length zero
```

```r
if (!is.null(x) && (x > 0))
  print("Hello")
```

The last example works due to short circuit evaluation which occurs with && and || but not & or |.

Attributes

Attributes

Attributes are metadata that can be attached to objects in R. Some are special (e.g. `class`, `comment`, `dim`, `dimnames`, `names`, …) and they change the behavior of the object(s).

Attributes are implemented as a *named list* that is attached to an object. They can be interacted with via the `attr()` and `attributes()` functions.

```r
(x = c(L=1,M=2,N=3))
```

```
L M N 
1 2 3 
```

```r
str(attributes(x))
```

```
List of 1
 $ names: chr [1:3] "L" "M" "N"
```

```r
attr(x, "names")
```

```
[1] "L" "M" "N"
```

```r
attr(x, "something")
```

```
NULL
```

Assigning attributes

The most commonly used / important attributes will usually have helper functions for getting and setting the attribute,

```r
x
```

```
L M N
1 2 3
```

```r
names(x) = c("Z","Y","X")
x
```

```
Z Y X
1 2 3
```

```r
names(x)
```

```
[1] "Z" "Y" "X"
```

```r
attr(x, "names") = c("A","B","C")
x
```

```
A B C
1 2 3
```

```r
names(x)
```

```
[1] "A" "B" "C"
```

Helpers functions vs attr

```r
names(x) = 1:3
x
```

```
1 2 3
1 2 3
```

```r
attributes(x)
```

```
$names
[1] "1" "2" "3"
```

```r
names(x) = c(TRUE, FALSE, TRUE)
x
```

```
 TRUE FALSE  TRUE
    1     2     3
```

```r
attributes(x)
```

```
$names
[1] "TRUE"  "FALSE" "TRUE"
```

```r
attr(x, "names") = 1:3
x
```

```
1 2 3
1 2 3
```

```r
attributes(x)
```

```
$names
[1] "1" "2" "3"
```

Factors

Factor objects are how R represents categorical data (e.g. a variable where there is a discrete set of possible outcomes).

```r
(x = factor(c("Sunny", "Cloudy", "Rainy", "Cloudy", "Cloudy")))
```

```
[1] Sunny  Cloudy Rainy  Cloudy Cloudy
Levels: Cloudy Rainy Sunny
```

```r
str(x)
```

```
 Factor w/ 3 levels "Cloudy","Rainy",..: 3 1 2 1 1
```

```r
typeof(x)
```

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

```r
mode(x)
```

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

```r
class(x)
```

```
[1] "factor"
```

Composition

A factor is just an integer vector with two attributes: `class` and `levels`.

```
x
```

```
[1] Sunny  Cloudy Rainy  Cloudy Cloudy
Levels: Cloudy Rainy Sunny
```

```
str(attributes(x))
```

```
List of 2
 $ levels: chr [1:3] "Cloudy" "Rainy" "Sunny"
 $ class : chr "factor"
```

We can build our own factor from scratch using `attr()`,

```
y = c(3L, 1L, 2L, 1L, 1L)
attr(y, "levels") = c("Cloudy", "Rainy", "Sunny")
attr(y, "class") = "factor"
y
```

```
[1] Sunny  Cloudy Rainy  Cloudy Cloudy
Levels: Cloudy Rainy Sunny
```

Building objects

The approach we just used is a bit clunky - generally the preferred method for construction an object with attributes from scratch is to use the `structure()` function.

```
( y = structure(
    c(3L, 1L, 2L, 1L, 1L),
    levels = c("Cloudy", "Rainy", "Sunny"),
    class = "factor"
) )
```

```
[1] Sunny  Cloudy Rainy  Cloudy Cloudy
Levels: Cloudy Rainy Sunny
```

```
class(y)
```

```
[1] "factor"
```

```
is.factor(y)
```

```
[1] TRUE
```

Factors are integer vectors?

Knowing factors are stored as integers help explain some of their more interesting behaviors:

```r
x+1
```

```
[1] NA NA NA NA NA
```

```r
is.integer(x)
```

```
[1] FALSE
```

```r
as.integer(x)
```

```
[1] 3 1 2 1 1
```

```r
as.character(x)
```

```
[1] "Sunny"  "Cloudy" "Rainy"  "Cloudy" "Cloudy"
```

```r
as.logical(x)
```

```
[1] NA NA NA NA NA
```

S3 Object System

`class`

The `class` attribute is an additional layer to R's type hierarchy,

value	`typeof()`	`mode()`	`class()`
`TRUE`	logical	logical	logical
`1`	double	numeric	numeric
`1L`	integer	numeric	integer
`"A"`	character	character	character
`NULL`	NULL	NULL	NULL
`list(1, "A")`	list	list	list
`factor("A")`	integer	numeric	factor
`function(x) x^2`	closure	function	function
`+`	builtin	function	function
`[`	special	function	function

S3 class specialization

```r
x = c("A","B","A","C")
```

```r
print( x )
```

```
[1] "A" "B" "A" "C"
```

```r
print( factor(x) )
```

```
[1] A B A C
Levels: A B C
```

```r
print( unclass( factor(x) ) )
```

```
[1] 1 2 1 3
attr(,"levels")
[1] "A" "B" "C"
```

```r
print.default( factor(x) )
```

```
[1] 1 2 1 3
```

What’s up with print?

```r
print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x13a8d8510>
<environment: namespace:base>
```

```r
print.default
```

```
function (x, digits = NULL, quote = TRUE, na.print = NULL, print.gap = NULL,
    right = FALSE, max = NULL, width = NULL, useSource = TRUE,
    ...)
{
    args <- pairlist(digits = digits, quote = quote, na.print = na.print,
        print.gap = print.gap, right = right, max = max, width = width,
        useSource = useSource, ...)
    missings <- c(missing(digits), missing(quote), missing(na.print),
        missing(print.gap), missing(right), missing(max), missing(width),
        missing(useSource))
    .Internal(print.default(x, args, missings))
}
<bytecode: 0x10d3b9e40>
<environment: namespace:base>
```

Other examples

```r
mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x139ee8e98>
<environment: namespace:base>
```

```r
summary
```

```
function (object, ...)
UseMethod("summary")
<bytecode: 0x1499bf8c0>
<environment: namespace:base>
```

```r
t.test
```

```
function (x, ...)
UseMethod("t.test")
<bytecode: 0x14bbf2790>
<environment: namespace:stats>
```

```r
plot
```

```
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x14a9f9408>
<environment: namespace:base>
```

Not all base functions use this approach,

```r
sum
```

```
function (..., na.rm = FALSE)  .Primitive("sum")
```

What is S3?

> S3 is R's first and simplest OO system. It is the only OO system used in the base and stats packages, and it's the most commonly used system in CRAN packages. S3 is informal and ad hoc, but it has a certain elegance in its minimalism: you can't take away any part of it and still have a useful OO system.
>
> — Hadley Wickham, Advanced R

S3 should not be confused with R's other object oriented systems: S4, Reference classes, R6, and soon R7.

What’s going on?

S3 objects and their related functions work using a very simple dispatch mechanism - a generic function is created whose sole job is to call the `UseMethod` function which then calls a class specialized function using the naming convention: `<generic>.<class>`

We can see all of the specialized versions of the generic using the `methods` function.

```r
methods("plot")
```

```
 [1] plot.acf*           plot.colors*        plot.data.frame*   
 [4] plot.decomposed.ts* plot.default        plot.dendrogram*   
 [7] plot.density*       plot.ecdf           plot.factor*       
[10] plot.formula*       plot.function       plot.hclust*       
[13] plot.histogram*     plot.HoltWinters*   plot.isoreg*       
[16] plot.lm*            plot.medpolish*     plot.mlm*          
[19] plot.ppr*           plot.prcomp*        plot.princomp*     
[22] plot.profile.nls*   plot.raster*        plot.spec*         
[25] plot.stepfun        plot.stl*           plot.table*        
[28] plot.ts             plot.tskernel*      plot.TukeyHSD*     
see '?methods' for accessing help and source code
```

Other examples

```r
methods("print")
```

```
  [1] print.acf*
  [2] print.activeConcordance*
  [3] print.AES*
  [4] print.anova*
  [5] print.aov*
  [6] print.aovlist*
  [7] print.ar*
  [8] print.Arima*
  [9] print.arima0*
 [10] print.AsIs
 [11] print.aspell*
 [12] print.aspell_inspect_context*
 [13] print.bibentry*
 [14] print.Bibtex*
 [15] print.browseVignettes*
 [16] print.by
 [17] print.changedFiles*
```

```r
print.factor
```

```
function (x, quote = FALSE, max.levels = NULL,
width = getOption("width"),
    ...)
{
    ord <- is.ordered(x)
    if (length(x) == 0L)
        cat(if (ord)
            "ordered"
        else "factor", "()\n", sep = "")
    else {
        xx <- character(length(x))
        xx[] <- as.character(x)
        keepAttrs <- setdiff(names(attributes(x)),
c("levels",
            "class"))
        attributes(xx)[keepAttrs] <- attributes(x)
[keepAttrs]
```

The other way

If instead we have a class and want to know what specialized functions exist for that class, then we can again use the `methods` function with the `class` argument.

```r
methods(class="factor")
```

```
 [1] [              [[             [[<-           [<-            all.equal
 [6] as.character   as.data.frame  as.Date        as.list        as.logical
[11] as.POSIXlt     as.vector      c              coerce         droplevels
[16] format         initialize     is.na<-        length<-       levels<-
[21] Math           Ops            plot           print          relevel
[26] relist         rep            show           slotsFromS3    summary
[31] Summary        xtfrm
see '?methods' for accessing help and source code
```

Adding methods

```r
( x = structure(
    c(1,2,3),
    class="class_A") )
```

```
[1] 1 2 3
attr(,"class")
[1] "class_A"
```

```r
( y = structure(
    c(6,5,4),
    class="class_B") )
```

```
[1] 6 5 4
attr(,"class")
[1] "class_B"
```

```r
print.class_A = function(x) {
  cat("(Class A) ")
  print.default(unclass(x))
}
print(x)
```

```
(Class A) [1] 1 2 3
```

```r
print.class_B = function(x) {
  cat("(Class B) ")
  print.default(unclass(x))
}
print(y)
```

```
(Class B) [1] 6 5 4
```

```r
class(x) = "class_B"
print(x)
```

```
(Class B) [1] 1 2 3
```

```r
class(y) = "class_A"
print(y)
```

```
(Class A) [1] 6 5 4
```

Defining a new S3 Generic

```r
shuffle = function(x) {
  UseMethod("shuffle")
}
```

```r
shuffle.default = function(x) {
  stop("Class ", class(x), " is not supported by shuffle.", call. = FALSE)
}
```

```r
shuffle.factor = function(f) {
  factor( sample(as.character(f)), levels = sample(levels(f)) )
}
```

```r
shuffle.integer = function(x) {
  sample(x)
}
```

Shuffle results

```r
shuffle( 1:10 )
```

```
 [1]  1  6  5  3  8 10  9  4  2  7
```

```r
shuffle( factor(c("A","B","C","A")) )
```

```
[1] C A A B
Levels: B A C
```

```r
shuffle( c(1, 2, 3, 4, 5) )
```

```
Error: Class numeric is not supported by shuffle.
```

```r
shuffle( letters[1:5] )
```

```
Error: Class character is not supported by shuffle.
```

```r
shuffle( factor(letters[1:5]) )
```

```
[1] a c e b d
Levels: d c b a e
```

Exercise 2 - classes, modes, and types

Below we have defined an S3 method called `report`, it is designed to return a message about the type/mode/class of an object passed to it.

```r
report = function(x) {
  UseMethod("report")
}

report.default = function(x) {
  "This class does not have a method defined."
}
```

```r
report.integer = function(x) {
 "I'm an integer!"
}

report.double = function(x) {
  "I'm a double!"
}

report.numeric = function(x) {
  "I'm a numeric!"
}
```

- Try running the `report` function with different input types, what happens?
- Now run `rm("report.integer")` in your Console and try using the `report` function again, what has changed?
- What does this tell us about S3, types, modes, and classes?
- What if we also `rm("report.double")`?

Conclusions?

From `UseMethod`s R documentation:

> If the object does not have a class attribute, it has an implicit class. Matrices and arrays have class “matrix” or “array” followed by the class of the underlying vector. Most vectors have class the result of `mode(x)`, except that integer vectors have class `c("integer", "numeric")` and real vectors have class `c("double", "numeric")`.

From Advanced R:

> How does UseMethod() work? It basically creates a vector of method names, paste0(“generic”, “.”, c(class(x), “default”)), and then looks for each potential method in turn.

Why?

See @WhyDoesR