Lists, Attributes, & S3

Lecture 03

Dr. Colin Rundel

Generic Vectors

Lists

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

```
1 list("A", c(TRUE,FALSE), (1:4)/2, list(1L), 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] 1


[[5]]
function (x)
x^2
```

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),
  function(x) x^2
) )
```

```
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 (x)
```

Recursive lists

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

```
1 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 an atomic vector concatenated with a list will be coerced to part of the list (as the list is more generic)

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

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

```
1 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()` - typical atomic type coercion rules then apply to determine the final type.

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

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

```
1 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 straightforward and avoids the use of magic numbers.

```
1 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) must be quoted,

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

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

Aside - Variable names vs. value names

We have seen how to assign a name to an R object (via `=` or `<-`). The general rule for these names is that it must start with a letter (upper or lower) or a `.` and then be followed by additional letters, numbers, `.` or `_`.

These names are unambiguous to the interpreter / parser and so do not need any additional decoration. However if you want to use a name that does not follow these rules, then you must quote it using backticks.

```
1 a b = 1
2 a b
```

```
Error in parse(text = input)
1: a b
      ^
```

```
1 "a b" = 1
2 "a b"
```

```
[1] "a b"
```

```
1 `a b` = 1
2 `a b`
```

```
[1] 1
```

Vector (atomic or generic) names can be any valid R character vector values (as this is how they are stored) but there are a number of circumstances where we use them like a variable name (e.g. `mtcars$mpg`), and so it is a good idea to avoid using names that violate the object naming rules to avoid having to use backticks all the time (e.g. `x$`knock knock``).

Exercise 1

Represent the following JSON data as a list in R.

```
{
  "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"
  } ]
}
```

NULL Values

NULLs

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

```
1 NULL
```

```
NULL
```

```
1 typeof(NULL)
```

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

```
1 mode(NULL)
```

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

```
1 length(NULL)
```

```
[1] 0
```

```
1 c()
```

```
NULL
```

```
1 c(NULL)
```

```
NULL
```

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

```
[1] 1 2
```

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

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

```
1 double()
```

```
numeric(0)
```

```
1 length(double())
```

```
[1] 0
```

```
1 typeof(double())
```

```
[1] "double"
```

```
1 mode(double())
```

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

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 resulting in a 0-length vector being returned.

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

```
numeric(0)
```

```
1 logical() | TRUE
```

```
logical(0)
```

```
1 log(numeric())
```

```
numeric(0)
```

```
1 character() > "M"
```

```
logical(0)
```

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

```
1 NULL + 1
```

```
numeric(0)
```

```
1 NULL > "M"
```

```
logical(0)
```

```
1 NULL | TRUE
```

```
logical(0)
```

```
1 log(NULL)
```

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

NULLs and comparison

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

```
x = NULL
```

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

```
Error in `if (x > 0) ...`:
! argument is of length zero
```

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

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

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

The null coalescing operator (`%||%`)

The `%||%` operator is a useful infix operator that returns the left hand side if it is not `NULL` and the right hand side otherwise.

```
`%||%`
```

```
function (x, y)
if (is.null(x)) y else x
<bytecode: 0x119924d90>
<environment: namespace:base>
```

```
if (x %||% 0 > 0)
  print("Hello")
```

```
if (x %||% 1 > 0)
  print("Hello")
```

```
[1] "Hello"
```

This operator is a relatively recent addition to base R (added in v4.4) but was popularized in the tidyverse and other packages.

Attributes

Attributes

Attributes are metadata that can be attached to objects in R.

Certain attributes are special (e.g. `class`, `comment`, `dim`, `dimnames`, `names`, ...) because they change the behavior of the object(s).

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

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

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

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

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

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

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

```
1 attr(x, "other")
```

```
NULL
```

Assigning attributes

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

```
1 x
```

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

```
1 names(x)
```

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

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

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

```
1 names(x)
```

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

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

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

```
1 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"
```

Note that there are some guardrails in place to ensure that special attributes make sense (e.g. `names` must be a character

Factors

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

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

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

```
str(x)
```

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

```
typeof(x)
```

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

```
class(x)
```

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

```
mode(x)
```

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

Composition

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

```
1 x
```

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

```
1 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()`,

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

```
1 x+1
```

```
Warning in Ops.factor(x, 1): '+' not meaningful for factors
```

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

```
1 is.integer(x)
```

```
[1] FALSE
```

```
1 as.integer(x)
```

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

```
1 as.character(x)
```

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

```
1 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

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

```
1 print( x )
```

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

```
1 print( factor(x) )
```

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

```
1 print( unclass( factor(x) ) )
```

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

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

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

What's up with print?

```
1 print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x1075a8a38>
<environment: namespace:base>
```

```
1 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: 0x107a78158>
<environment: namespace:base>
```

Other examples

```
1 mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x109a227d0>
<environment: namespace:base>
```

```
1 summary
```

```
function (object, ...)
UseMethod("summary")
<bytecode: 0x107aa8bd8>
<environment: namespace:base>
```

```
1 t.test
```

```
function (x, ...)
UseMethod("t.test")
<bytecode: 0x12acd4480>
<environment: namespace:stats>
```

```
1 plot
```

```
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x108185b48>
<environment: namespace:base>
```

A lot but not all base functions use this approach,

```
1 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

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.

```
1 methods("plot")
```

```
 [1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
 [4] plot.default        plot.dendrogram*    plot.density*
 [7] plot.ecdf           plot.factor*        plot.formula*
[10] plot.function       plot.hclust*        plot.histogram*
[13] plot.HoltWinters*   plot.isoreg*        plot.lm*
[16] plot.medpolish*     plot.mlm*           plot.ppr*
[19] plot.prcomp*        plot.princomp*      plot.profile*
[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

```
1 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*
[18] print.check_bogus_return*
[19] print.check_code_usage_in_package*
[20] print.check_compiled_code*
```

```
1 print.factor
```

```
function (x, quote = FALSE, max.levels = N
    ...)
{
    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(attribu
            "class"))
        attributes(xx)[keepAttrs] <- attri
        print(xx, quote = quote, ...)
    }
    maxl <- max.levels %||% TRUE
    if (maxl) {
        n <- length(lev <- encodeString(le
            "\"", "")))
```

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.

```
1 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

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

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

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

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

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

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

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

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

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

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

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

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

Defining a new S3 Generic

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

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

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

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

Shuffle results

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

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

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

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

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

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

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

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

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

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

Demo - classes, modes, and types

```r
report = function(x) {
  UseMethod("report")
}
report.default = function(x) {
  paste0("Class ", class(x)," does not have a method
}
report.integer = function(x) {
 "I'm an integer!"
}
report.double = function(x) {
  "I'm a double!"
}
report.numeric = function(x) {
  "I'm a numeric!"
}

#rm(report.integer)
#rm(report.double)
#rm(report.numeric)

report(1)
report(1L)
report("1")
```

On the left 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.

- Try running the `report` function with different input types, what happens?
- Now uncomment out the first `rm()` line and try rerunning the code, what has changed?
- What does this tell us about S3, types, modes, and classes?
- What if we also uncomment the code the other `rm()` lines?

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?

"For compatibility with S"