Generic Vectors

Lists

Lists are the final 1 dimensional (i.e. have a length) data structure in R, the different from atomic vectors in that they can contain a heterogeneous collection of R object (e.g. atomic vectors, functions, other lists, etc.).

```
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
##
## [[4]][[2]]
## [1] 1
##
##
```

List Structure

Often we want a more compact representation of a complex object, the `str` function is useful for this particular task

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

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

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

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

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

```
##  chr "A"
```

```
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
```

Recursive lists

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

```
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 the most 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

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

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

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

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

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

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

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

`as.intger` and similar functions can be used, but only if the list is flat (i.e. no nested lists)

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.

```
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 need to be quoted,

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

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

NULL Values

NULLs

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

```
NULL
```

```
## NULL
```

```
typeof(NULL)
```

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

```
mode(NULL)
```

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

```
length(NULL)
```

```
## [1] 0
```

```
c()
```

```
## NULL
```

```
c(NULL)
```

```
## NULL
```

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

```
## [1] 1 2
```

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

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

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

0-length coercion

0-length length coercion is a special case of length coercion when one of the arugments has length 0. In this case the longer vector's length is not used and result will have length 0.

```
integer() + 1
```

```
## numeric(0)
```

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

```
## numeric(0)
```

```
logical() | TRUE
```

```
## logical(0)
```

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

```
## logical(0)
```

As a `NULL` values always have length 0, this coercion rule will apply (note type coercion is also occurring here)

```
NULL + 1
```

```
## numeric(0)
```

```
NULL | TRUE
```

```
## logical(0)
```

```
NULL > "M"
```

Attributes

Attributes

Attributes are metadata that can be attached to objects in R. Some are special, e.g. `class`, `comment`, `dim`, `dimnames`, `names`, etc., as they change the way in which the object behaves.

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

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

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

```
attributes(x)
```

```
## $names
## [1] "L" "M" "N"
```

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

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

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

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

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

```
## NULL
```

Assigning attributes

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

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

```
names(x)
```

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

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

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

```
names(x)
```

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

Helpers functions vs attr

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

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

```
attributes(x)
```

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

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

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

```
attributes(x)
```

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

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

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

```
attributes(x)
```

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

Factors

Factor objects are how R represents categorical data (e.g. a variable where there are a fixed # 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"
```

```
mode(x)
```

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

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
```

```
attributes(x)
```

```
## $levels
## [1] "Cloudy" "Rainy"  "Sunny"
##
## $class
## [1] "factor"
```

We can build our own factor from scratch using,

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

Building objects

The approach we just used is a bit clunky - generally the prefered 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"
)

y
```

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

```
class(y)
```

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

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

```
## [1] TRUE
```

Named arguments become attributes

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

S3 class specialization

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

```
print( x )
```

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

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

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

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

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

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

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

```
print
```

Other examples

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x558b9d5f13c0>
## <environment: namespace:base>
```

```
summary
```

```
## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x558b9c3e1418>
## <environment: namespace:base>
```

```
t.test
```

```
## function (x, ...)
## UseMethod("t.test")
## <bytecode: 0x558b9b77d460>
## <environment: namespace:stats>
```

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x558b9bf79960>
## <environment: namespace:base>
```

Not all base functions use this approach,

```
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, and R6*.

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.

```
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.nls*
## [22] plot.R6*            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
```

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

```
##   [1] print.acf*
##   [2] print.AES*
##   [3] print.anova*
##   [4] print.aov*
##   [5] print.aovlist*
##   [6] print.ar*
##   [7] print.Arima*
##   [8] print.arima0*
##   [9] print.AsIs
##  [10] print.aspell*
##  [11] print.aspell_inspect_context*
##  [12] print.bibentry*
##  [13] print.Bibtex*
##  [14] print.browseVignettes*
##  [15] print.bslib_fragment*
##  [16] print.bslib_page*
##  [17] print.by
##  [18] print.changedFiles*
##  [19] print.check_bogus_return*
##  [20] print.check_code_usage_in_package*
##  [21] print.check_compiled_code*
##  [22] print.check_demo_index*
##  [23] print.check_depdef*
##  [24] print.check_details*
##  [25] print.check_details_changes*
##  [26] print.check_doi_db*
##  [27] print.check_dotInternal*
##  [28] print.check_make_vars*
##  [29] print.check_nonAPI_calls*
##  [30] print.check_package_code_assign_to_globalenv*
##  [31] print.check_package_code_attach*
##  [32] print.check_package_code_data_into_globalenv*
##  [33] print.check_package_code_startup_functions*
```

```
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", "(0)\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]
##         print(xx, quote = quote, ...)
##     }
##     maxl <- if (is.null(max.levels))
##         TRUE
##     else max.levels
##     if (maxl) {
##         n <- length(lev <- encodeString(levels(x), quote = ifelse(quote,
##             "\"", "")))
##         colsep <- if (ord)
##             " < "
##         else " "
```

```
print.integer
```

```
## Error in eval(expr, envir, enclos): object 'print.integer' not found
```

```
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: 0x558b9bb6dd88>
## <environment: namespace:base>
```

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.

```
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")
x
```

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

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

x
```

```
## Class A!
## [1] 1 2 3
```

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

```
## Class B!
## [1] 1 2 3
```

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

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

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

y
```

```
## Class B!
## [1] 6 5 4
```

```
class(y) = "class_A"
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.\n", call. = FALSE)
}
```

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

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

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

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

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

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

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

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

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.

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

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

```
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?

Conclusion?

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?