# Attributes, Classes, S3, and Subsetting

Colin Rundel

2019-01-17

# Generic Vectors (Briefly)

# Lists

Lists are *generic vectors*, as such they are 1 dimensional (i.e. have a length) and can contain any type of R object.

```
list("A", c(TRUE,FALSE), (1:4)/2, list(1:2), 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 2
##
##
## [[5]]
## function(x) x^2
```

# structure

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

```
str(1:4)
```

```
##  int [1:4] 1 2 3 4
```

```
str( list("A", c(TRUE,FALSE), (1:4)/2, list(1:2), 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 1
##   ..$ : int [1:2] 1 2
##  $ :function (x)
##   ..- attr(*, "srcref")= 'srcref' int [1:8] 1 51 1 65 51 65 1 1
##   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7fe98ed9cf40>
```

# Lists as "trees"

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

```
str( list(a=1, b=list(c=2, d=list(f=3, g=4), e=5)) )
```

```
## List of 2
##  $ a: num 1
##  $ b:List of 3
##   ..$ c: num 2
##   ..$ d:List of 2
##   .. ..$ f: num 3
##   .. ..$ g: num 4
##   ..$ e: num 5
```

# Lists as "trees"

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

```
str( list(a=1, b=list(c=2, d=list(f=3, g=4), e=5)) )
```

```
## List of 2
##  $ a: num 1
##  $ b:List of 3
##   ..$ c: num 2
##   ..$ d:List of 2
##   .. ..$ f: num 3
##   .. ..$ g: num 4
##   ..$ e: num 5
```

```
json = '{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },{
      "type": "mobile",
      "number": "123 456-7890"
    }
  ]
}'
```

# Lists as "trees"

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

```
str( list(a=1, b=list(c=2, d=list(f=3, g=4), e=5)) )
```

```
## List of 2
##  $ a: num 1
##  $ b:List of 3
##   ..$ c: num 2
##   ..$ d:List of 2
##   .. ..$ f: num 3
##   .. ..$ g: num 4
##   ..$ e: num 5
```

```
json = '{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },{
      "type": "mobile",
      "number": "123 456-7890"
    }
  ]
}'
```

```
str( jsonlite::fromJSON(json, simplifyVector =
```

```
## List of 5
##  $ firstName   : chr "John"
##  $ lastName    : chr "Smith"
##  $ isAlive     : logi TRUE
##  $ age         : int 27
##  $ phoneNumbers:List of 2
##   ..$ :List of 2
##   .. ..$ type  : chr "home"
##   .. ..$ number: chr "212 555-1234"
##   ..$ :List of 2
##   .. ..$ type  : chr "mobile"
##   .. ..$ number: chr "123 456-7890"
```

# List Coercion - concatenation

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

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

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

# List Coercion - concatenation

By default a vector will be coerced to a list (as a list is more generic) 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 atomic vector's type.

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

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

```
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"
```

# Attributes

# Attributes

Attributes are metadata that can be attached to objects in R. Some are special (e.g. `class`, `comment`, `dim`, `dimnames`, `names`, etc.) and change the way in which an object is treated by R.

# Attributes

Attributes are metadata that can be attached to objects in R. Some are special (e.g. `class`, `comment`, `dim`, `dimnames`, `names`, etc.) and change the way in which an object is treated by R.

Attributes are implemented as a named list that are accessed (get and set) individually via the `attr` function and collectively via the `attributes` function.

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

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

# Attributes

Attributes are metadata that can be attached to objects in R. Some are special (e.g. `class`, `comment`, `dim`, `dimnames`, `names`, etc.) and change the way in which an object is treated by R.

Attributes are implemented as a named list that are accessed (get and set) individually via the `attr` function and collectively via the `attributes` function.

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

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

```
str(x)
```

```
##  Named num [1:3] 1 2 3
##  - attr(*, "names")= chr [1:3] "L" "M" "N"
```

# Attributes

Attributes are metadata that can be attached to objects in R. Some are special (e.g. `class`, `comment`, `dim`, `dimnames`, `names`, etc.) and change the way in which an object is treated by R.

Attributes are implemented as a named list that are accessed (get and set) individually via the `attr` function and collectively via the `attributes` function.

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

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

```
str(x)
```

```
##  Named num [1:3] 1 2 3
##  - attr(*, "names")= chr [1:3] "L" "M" "N"
```

```
attributes(x)
```

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

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

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

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

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

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

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

```r
names(x)
```

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

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

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

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

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

```r
names(x)
```

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

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

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

```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"
```

# 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
```

# 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
```

# 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"
```

# Composition

A factor is just an integer vector with two attributes: `class = "factor"` and `levels` a character vector with the possible levels.

```
x
```

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

```
attributes(x)
```

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

# Composition

A factor is just an integer vector with two attributes: `class = "factor"` and `levels` a character vector with the possible 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
```

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

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

```
x+1
```

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

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

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

```
## [1] FALSE
```

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

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

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

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

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

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

# Data Frames

# Data Frames

A data frame is how R handles heterogeneous tabular data (i.e. rows and columns) and is one of the most commonly used data structure in R.

```
(df = data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  z = c(TRUE)
))
```

```
##   x y    z
## 1 1 a TRUE
## 2 2 b TRUE
## 3 3 c TRUE
```

# Data Frames

A data frame is how R handles heterogeneous tabular data (i.e. rows and columns) and is one of the most commonly used data structure in R.

```
(df = data.frame(
  x = 1:3,
  y = c("a", "b", "c"),
  z = c(TRUE)
))
```

```
##   x y    z
## 1 1 a TRUE
## 2 2 b TRUE
## 3 3 c TRUE
```

R represents data frames using a *list* of equal length *vectors* (usually atomic, but they can be generic as well).

```
str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ x: int  1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ z: logi  TRUE TRUE TRUE
```

```r
typeof(df)
```

```
## [1] "list"
```

```r
class(df)
```

```
## [1] "data.frame"
```

```r
attributes(df)
```

```
## $names
## [1] "x" "y" "z"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3
```

```
typeof(df)
```

```
## [1] "list"
```

```
class(df)
```

```
## [1] "data.frame"
```

```
attributes(df)
```

```
## $names
## [1] "x" "y" "z"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3
```

```
str(unclass(df))
```

```
## List of 3
##  $ x: int [1:3] 1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ z: logi [1:3] TRUE TRUE TRUE
##  - attr(*, "row.names")= int [1:3] 1 2 3
```

# Roll your own data.frame

```
df2 = list(x = 1:3, y = factor(c("a", "b", "c")), z = c(TRUE, TRUE, TRUE))
```

# Roll your own data.frame

```
df2 = list(x = 1:3, y = factor(c("a", "b", "c")), z = c(TRUE, TRUE, TRUE))
```

```
attr(df2,"class") = "data.frame"
df2
```

```
## [1] x y z
## <0 rows> (or 0-length row.names)
```

# Roll your own data.frame

```
df2 = list(x = 1:3, y = factor(c("a", "b", "c")), z = c(TRUE, TRUE, TRUE))
```

```
attr(df2,"class") = "data.frame"
df2
```

```
## [1] x y z
## <0 rows> (or 0-length row.names)
```

```
attr(df2,"row.names") = 1:3
df2
```

```
##   x y    z
## 1 1 a TRUE
## 2 2 b TRUE
## 3 3 c TRUE
```

# Roll your own data.frame

```
df2 = list(x = 1:3, y = factor(c("a", "b", "c")), z = c(TRUE, TRUE, TRUE))
```

```
attr(df2,"class") = "data.frame"
df2
```

```
## [1] x y z
## <0 rows> (or 0-length row.names)
```

```
attr(df2,"row.names") = 1:3
df2
```

```
##   x y    z
## 1 1 a TRUE
## 2 2 b TRUE
## 3 3 c TRUE
```

```
str(df2)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ x: int  1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ z: logi  TRUE TRUE TRUE
```

```
identical(df, df2)
```

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

# Strings (Characters) vs Factors

By default character vectors will be convert into factors when they are included in a data frame.

Sometimes this is useful (usually it isn't), either way it is important to know what type/class you are working with. This behavior can be changed using the `stringsAsFactors` argument to `data.frame` and related functions (e.g. `read.csv`, `read.table`, etc.).

```
df = data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = FALSE)
df
```

```
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
```

```
str(df)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: chr  "a" "b" "c"
```

# Length Coercion

For data frames on creation the lengths of the component vectors will be coerced to match, however if they not multiples then there will be an error (previously this produced a warning).

```
data.frame(x = 1:3, y = c("a"))
```

```
##   x y
## 1 1 a
## 2 2 a
## 3 3 a
```

```
data.frame(x = 1:3, y = c("a","b"))
```

```
## Error in data.frame(x = 1:3, y = c("a", "b")): arguments imply differing number of rows: 3, 2
```

```
data.frame(x = 1:3, y = character())
```

```
## Error in data.frame(x = 1:3, y = character()): arguments imply differing number of rows: 3, 0
```

# S3 Object System

# `class`

Confusingly, `class` adds another level onto R's type hierarchy,

| value | `typeof()` | `mode()` | `class()` |
|---|---|---|---|
| `NULL` | NULL | NULL | NULL |
| `TRUE` | logical | logical | logical |
| `1` | double | numeric | numeric |
| `1L` | integer | numeric | integer |
| `"A"` | character | character | character |

# class

Confusingly, `class` adds another level onto R's type hierarchy,

| value | typeof() | mode() | class() |
|---|---|---|---|
| NULL | NULL | NULL | NULL |
| TRUE | logical | logical | logical |
| 1 | double | numeric | numeric |
| 1L | integer | numeric | integer |
| "A" | character | character | character |

```r
class( matrix(1,2,2) )
```

```
## [1] "matrix"
```

```r
class( factor(c("A","B")) )
```

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

```r
class( data.frame(x=1:3) )
```

```
## [1] "data.frame"
```

```r
class( (function(x) x^2) )
```

```
## [1] "function"
```

# 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"
```

# 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"
```

```
df = data.frame(a=1:3, b=4:6, c=TRUE)
print( df )
```

```
##   a b    c
## 1 1 4 TRUE
## 2 2 5 TRUE
## 3 3 6 TRUE
```

```
print( unclass(df) )
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] 4 5 6
##
## $c
## [1] TRUE TRUE TRUE
##
## attr(,"row.names")
## [1] 1 2 3
```

# 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"
```

```
df = data.frame(a=1:3, b=4:6, c=TRUE)
print( df )
```

```
##   a b    c
## 1 1 4 TRUE
## 2 2 5 TRUE
## 3 3 6 TRUE
```

```
print( unclass(df) )
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] 4 5 6
##
## $c
## [1] TRUE TRUE TRUE
##
## attr(,"row.names")
## [1] 1 2 3
```

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x7fe990cee3f0>
## <environment: namespace:base>
```

# Other examples

```r
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7fe98d37ae18>
## <environment: namespace:base>
```

```r
summary
```

```
## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x7fe993471d38>
## <environment: namespace:base>
```

```r
t.test
```

```
## function (x, ...)
## UseMethod("t.test")
## <bytecode: 0x7fe98d4a84d8>
## <environment: namespace:stats>
```

```r
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7fe98e4ae428>
## <environment: namespace:graphics>
```

Not all base functions are S3,

```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, 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`.

# 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.git_repository* 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
```

```r
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.by
##  [16] print.bytes*
##  [17] print.changedFiles*
##  [18] print.check_code_usage_in_package*
##  [19] print.check_compiled_code*
##  [20] print.check_demo_index*
##  [21] print.check_depdef*
##  [22] print.check_details*
##  [23] print.check_details_changes*
##  [24] print.check_doi_db*
##  [25] print.check_dotInternal*
##  [26] print.check_make_vars*
##  [27] print.check_nonAPI_calls*
##  [28] print.check_package_code_assign_to_globalenv*
##  [29] print.check_package_code_attach*
##  [30] print.check_package_code_data_into_globalenv*
##  [31] print.check_package_code_startup_functions*
##  [32] print.check_package_code_syntax*
##  [33] print.check_package_code_unload_functions*
##  [34] print.check_package_compact_datasets*
##  [35] print.check_package_CRAN_incoming*
##  [36] print.check_package_datasets*
##  [37] print.check_package_depends*
##  [38] print.check_package_description*
##  [39] print.check_package_description_encoding*
##  [40] print.check_package_license*
##  [41] print.check_packages_in_dir*
##  [42] print.check_packages_used*
```

```
print.data.frame
```

```
## function (x, ..., digits = NULL, quote = FALSE, right = TRUE,
##     row.names = TRUE, max = NULL)
## {
##     n <- length(row.names(x))
##     if (length(x) == 0L) {
##         cat(sprintf(ngettext(n, "data frame with 0 columns and %d row",
##             "data frame with 0 columns and %d rows"), n), "\n",
##             sep = "")
##     }
##     else if (n == 0L) {
##         print.default(names(x), quote = FALSE)
##         cat(gettext("<0 rows> (or 0-length row.names)\n"))
##     }
##     else {
##         if (is.null(max))
##             max <- getOption("max.print", 99999L)
##         if (!is.finite(max))
##             stop("invalid 'max' / getOption(\"max.print\"): ",
##                 max)
##         omit <- (n0 <- max%/%length(x)) < n
##         m <- as.matrix(format.data.frame(if (omit)
##             x[seq_len(n0), , drop = FALSE]
##         else x, digits = digits, na.encode = FALSE))
##         if (!isTRUE(row.names))
##             dimnames(m)[[1L]] <- if (isFALSE(row.names))
##                 rep.int("", if (omit)
##                   n0
##                 else n)
##             else row.names
##         print(m, ..., quote = quote, right = right, max = max)
##         if (omit)
##             cat(" [ reached 'max' / getOption(\"max.print\") -- omitted",
##                 n - n0, "rows ]\n")
##     }
```

```
print.integer
```

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

```r
print.integer
```

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

```r
print.default
```

```
## function (x, digits = NULL, quote = TRUE, na.print = NULL, print.gap = NULL,
##     right = FALSE, max = NULL, useSource = TRUE, ...)
## {
##     args <- pairlist(digits = digits, quote = quote, na.print = na.print,
##         print.gap = print.gap, right = right, max = max, useSource = useSource,
##         ...)
##     missings <- c(missing(digits), missing(quote), missing(na.print),
##         missing(print.gap), missing(right), missing(max), missing(useSource))
##     .Internal(print.default(x, args, missings))
## }
## <bytecode: 0x7fe98eab7410>
## <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 - this time with the `class` argument.

```
methods(class="data.frame")
```

```
##  [1] [             [[            [[<-          [<-           $<-          
##  [6] aggregate     anyDuplicated as.data.frame as.list       as.matrix    
## [11] by            cbind         coerce        dim           dimnames     
## [16] dimnames<-    droplevels    duplicated    edit          format       
## [21] formula       head          initialize    is.na         Math         
## [26] merge         na.exclude    na.omit       Ops           plot         
## [31] print         prompt        rbind         row.names     row.names<-  
## [36] rowsum        show          slotsFromS3   split         split<-      
## [41] stack         str           subset        summary       Summary      
## [46] t             tail          transform     type.convert  unique       
## [51] unstack       within       
## see '?methods' for accessing help and source code
```

```
`is.na.data.frame`
```

```
## function (x)
## {
##     y <- if (length(x)) {
##         do.call("cbind", lapply(x, "is.na"))
##     }
##     else matrix(FALSE, length(row.names(x)), 0)
##     if (.row_names_info(x) > 0L)
##         rownames(y) <- row.names(x)
##     y
## }
## <bytecode: 0x7fe98e5d3988>
## <environment: namespace:base>
```

```r
`is.na.data.frame`
```

```
## function (x)
## {
##     y <- if (length(x)) {
##         do.call("cbind", lapply(x, "is.na"))
##     }
##     else matrix(FALSE, length(row.names(x)), 0)
##     if (.row_names_info(x) > 0L)
##         rownames(y) <- row.names(x)
##     y
## }
## <bytecode: 0x7fe98e5d3988>
## <environment: namespace:base>
```

```r
df = data.frame(x = c(1,NA,3), y = c(TRUE, FALSE, NA))
is.na(df)
```

```
##          x     y
## [1,] FALSE FALSE
## [2,]  TRUE FALSE
## [3,] FALSE  TRUE
```

# Adding methods

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

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

```r
y = structure(c(1,2,3), class="class_B")
y
```

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

# Adding methods

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

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

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

x
```

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

```r
y = structure(c(1,2,3), class="class_B")
y
```

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

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

y
```

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

# Adding methods

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

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

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

x
```

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

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

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

```r
y = structure(c(1,2,3), class="class_B")
y
```

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

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

y
```

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

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

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

# 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.data.frame = function(df) {
  sample(df)
}

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

# 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.data.frame = function(df) {
  sample(df)
}

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

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

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

```
shuffle(
  data.frame(a=1:4, b=5:8, c=9:12)
)
```

```
##    c a b
## 1  9 1 5
## 2 10 2 6
## 3 11 3 7
## 4 12 4 8
```

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

```
## Error: Class character is not supported by shuf
```

# Subsetting

# Subsetting in General

R has three subsetting operators (`[`, `[[`, and `$`).

The behavior of these operators will depend on the object (class) they are being used with.

# Subsetting in General

R has three subsetting operators (`[`, `[[`, and `$`).

The behavior of these operators will depend on the object (class) they are being used with.

In general there are 6 different types of subseting that can be performed:

- Positive integers
- Negative integers
- Logical values
- Empty / NULL
- Zero
- Character values (names)

The exact behavior of each of these depends on the type / class being subset.

# Positive Integer subsetting

Returns elements at the given location(s) (Note - R uses a 1-based indexing scheme).

```
x = c(1,4,7)
y = list(1,4,7)
```

```
x[c(1,3)]
```

```
## [1] 1 7
```

```
x[c(1,1)]
```

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

```
x[c(1.9,2.1)]
```

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

```
str( y[c(1,3)] )
```

```
## List of 2
##  $ : num 1
##  $ : num 7
```

```
str( y[c(1,1)] )
```

```
## List of 2
##  $ : num 1
##  $ : num 1
```

```
str( y[c(1.9,2.1)] )
```

```
## List of 2
##  $ : num 1
##  $ : num 4
```

# Negative Integer subsetting

Excludes elements at the given location(s)

```r
x = c(1,4,7)
x[-1]
```

```
## [1] 4 7
```

```r
x[-c(1,3)]
```

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

```r
x[c(-1,-1)]
```

```
## [1] 4 7
```

```r
y = list(1,4,7)
str( y[-1] )
```

```
## List of 2
##  $ : num 4
##  $ : num 7
```

```r
str( y[-c(1,3)] )
```

```
## List of 1
##  $ : num 4
```

```r
x[c(-1,2)]
```

```
## Error in x[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

```r
y[c(-1,2)]
```

```
## Error in y[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

# Logical Value Subsetting

Returns elements that correspond to `TRUE` in the logical vector. Length of the logical vector is expanded to be the same of the vector being subsetted (length coercion).

```r
x = c(1,4,7,12)
x[c(TRUE,TRUE,FALSE,TRUE)]
```

```
## [1]  1  4 12
```

```r
x[c(TRUE,FALSE)]
```

```
## [1] 1 7
```

```r
x[x %% 2 == 0]
```

```
## [1]  4 12
```

```r
y = list(1,4,7,12)
str( y[c(TRUE,TRUE,FALSE,TRUE)] )
```

```
## List of 3
##  $ : num 1
##  $ : num 4
##  $ : num 12
```

```r
str( y[c(TRUE,FALSE)] )
```

```
## List of 2
##  $ : num 1
##  $ : num 7
```

# Logical Value Subsetting

Returns elements that correspond to `TRUE` in the logical vector. Length of the logical vector is expanded to be the same of the vector being subsetted (length coercion).

```
x = c(1,4,7,12)
x[c(TRUE,TRUE,FALSE,TRUE)]
```

```
## [1]  1  4 12
```

```
x[c(TRUE,FALSE)]
```

```
## [1] 1 7
```

```
x[x %% 2 == 0]
```

```
## [1]  4 12
```

```
y = list(1,4,7,12)
str( y[c(TRUE,TRUE,FALSE,TRUE)] )
```

```
## List of 3
##  $ : num 1
##  $ : num 4
##  $ : num 12
```

```
str( y[c(TRUE,FALSE)] )
```

```
## List of 2
##  $ : num 1
##  $ : num 7
```

```
str( y[y %% 2 == 0] )
```

```
## Error in y%%2: non-numeric argument to binary operator
```

# Empty Subsetting

Returns the original vector.

```
x = c(1,4,7)
x[]
```

```
## [1] 1 4 7
```

```
y = list(1,4,7)
str(y[])
```

```
## List of 3
##  $ : num 1
##  $ : num 4
##  $ : num 7
```

# Zero subsetting

Returns an empty vector (of the same type)

```
x = c(1,4,7)
x[0]
```

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

```
y = list(1,4,7)
str(y[0])
```

```
##  list()
```

```
x[c(0,1)]
```

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

```
y[c(0,1)]
```

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

# Character subsetting

If the vector has names, select elements whose names correspond to the values in the character vector.

```
x = c(a=1,b=4,c=7)
x["a"]
```

```
## a
## 1
```

```
x[c("a","a")]
```

```
## a a
## 1 1
```

```
x[c("b","c")]
```

```
## b c
## 4 7
```

```
y = list(a=1,b=4,c=7)
str(y["a"])
```

```
## List of 1
##  $ a: num 1
```

```
str(y[c("a","a")])
```

```
## List of 2
##  $ a: num 1
##  $ a: num 1
```

```
str(y[c("b","c")])
```

```
## List of 2
##  $ b: num 4
##  $ c: num 7
```

# Out of bounds

```r
x = c(1,4,7)
x[4]
```

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

```r
x["a"]
```

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

```r
x[c(1,4)]
```

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

```r
y = list(1,4,7)
str(y[4])
```

```
## List of 1
##  $ : NULL
```

```r
str(y["a"])
```

```
## List of 1
##  $ : NULL
```

```r
str(y[c(1,4)])
```

```
## List of 2
##  $ : num 1
##  $ : NULL
```

# Missing and NULL

```r
x = c(1,4,7)
x[NA]
```

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

```r
x[NULL]
```

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

```r
x[c(1,NA)]
```

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

```r
y = list(1,4,7)
str(y[NA])
```

```
## List of 3
##  $ : NULL
##  $ : NULL
##  $ : NULL
```

```r
str(y[NULL])
```

```
##  list()
```

```r
str(y[c(1,NA)])
```

```
## List of 2
##  $ : num 1
##  $ : NULL
```

# Atomic vectors - [ vs. [[

`[[` subsets like `[` except it can only subset for a *single* value or position.

```
x = c(a=1,b=4,c=7)
```

# Atomic vectors - [ vs. [[

`[[` subsets like `[` except it can only subset for a *single* value or position.

```
x = c(a=1,b=4,c=7)
```

```
x[1]
```

```
## a
## 1
```

# Atomic vectors - [ vs. [[

`[[` subsets like `[` except it can only subset for a *single* value or position.

```
x = c(a=1,b=4,c=7)
```

```
x[1]
```

```
## a
## 1
```

```
x[[1]]
```

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

```
x[["a"]]
```

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

```
x[[1:2]]
```

```
## Error in x[[1:2]]: attempt to select more than one element in vectorIndex
```

```
x[[TRUE]]
```

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

# Generic Vectors - [ vs. [[

Subsets a single value, but returns the value - not a list containing that value.

```
y = list(a=1,b=4,c=7)
```

```
y[2]
```

```
## $b
## [1] 4
```

```
str( y[2] )
```

```
## List of 1
##  $ b: num 4
```

# Generic Vectors - [ vs. [[

Subsets a single value, but returns the value - not a list containing that value.

```r
y = list(a=1,b=4,c=7)
```

```r
y[2]
```

```
## $b
## [1] 4
```

```r
str( y[2] )
```

```
## List of 1
##  $ b: num 4
```

```r
y[[2]]
```

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

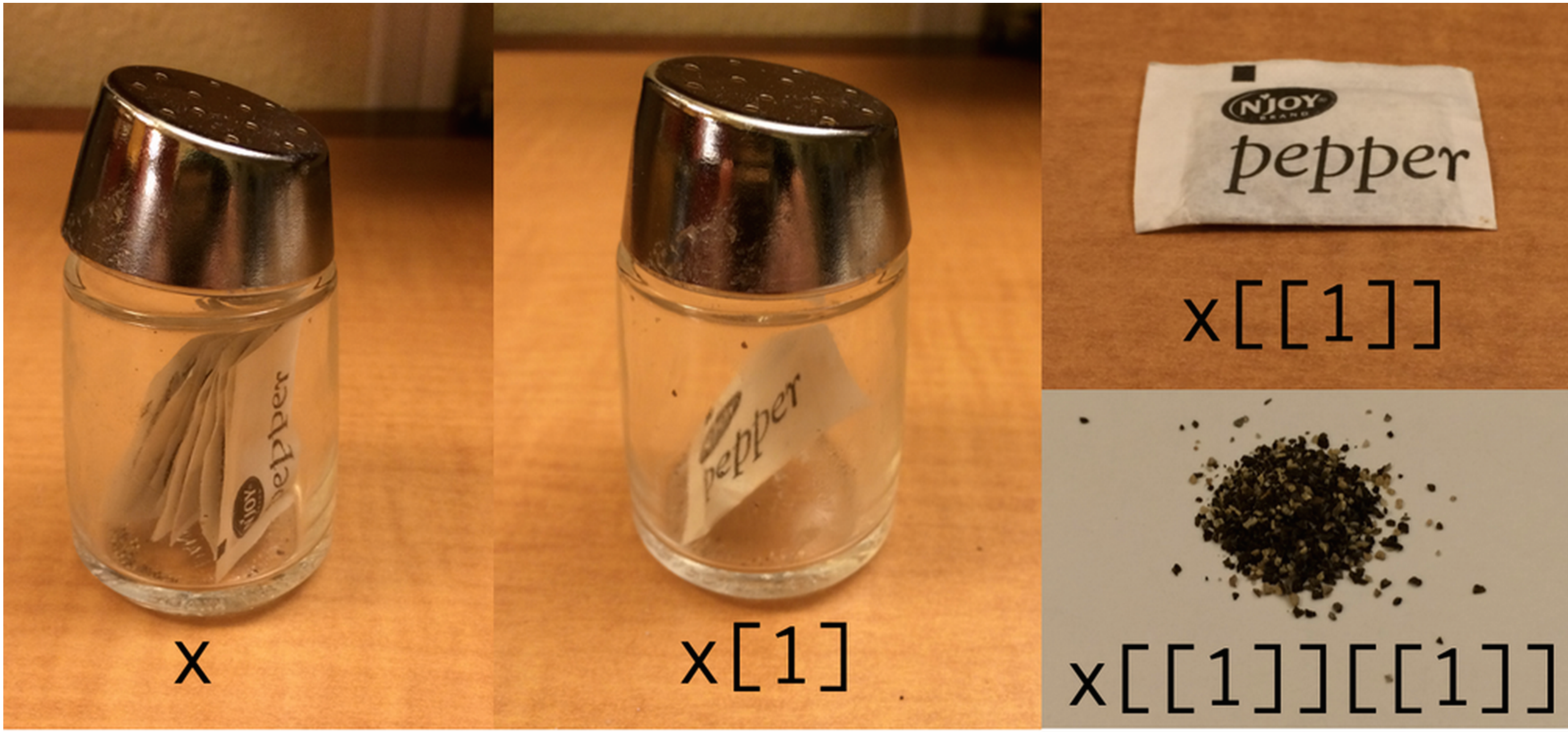
```r
y[["b"]]
```

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

```r
y[[1:2]]
```

```
## Error in y[[1:2]]: subscript out of bounds
```

# Hadley's Analogy




**Hadley Wickham** @hadleywickham · 6h

Indexing lists in #rstats. Inspired by the Residence Inn

273 370

# [[ vs. $

`$` is equivalent to `[[` but it only works for named *lists* and it has a terrible default where it uses partial matching (`exact=FALSE`) to access the underlying value.

```
x = c("abc"=1, "def"=5)
x$abc
```

```
## Error in x$abc: $ operator is invalid for atomic vectors
```

```
y = list("abc"=1, "def"=5)
y[["abc"]]
```

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

```
y$abc
```

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

```
y$d
```

```
## [1] 5
```

# A common gotcha

Why does the following code not work?

```
x = list(abc = 1:10, def = 10:1)
y = "abc"

x$y
```

```
## NULL
```

# A common gotcha

Why does the following code not work?

```
x = list(abc = 1:10, def = 10:1)
y = "abc"

x$y
```

```
## NULL
```

The expression `x$y` gets directly interpreted as `x[["y"]]` by R, not the include of the `"`s, this is not the same as the expression `x[[y]]`.

```
x[[y]]
```

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

# (After Class) Exercise 1

Below are 100 values,

```
x = c(56, 3, 17, 2, 4, 9, 6, 5, 19, 5, 2, 3, 5, 0, 13, 12, 6, 31, 10, 21, 8, 4, 1, 1, 2, 5, 16,
      3, 4, 8, 5, 2, 8, 6, 18, 40, 10, 20, 1, 27, 2, 11, 14, 5, 7, 0, 3, 0, 7, 0, 8, 10, 10, 12
      21, 3, 34, 55, 18, 2, 9, 29, 1, 4, 7, 14, 7, 1, 2, 7, 4, 74, 5, 0, 3, 13, 2, 8, 1, 6, 13,
      5, 2, 4, 4, 14, 15, 4, 17, 1, 9)
```

write down how you would create a subset to accomplish each of the following:

- Select every third value starting at position 2 in `x`.
- Remove all values with an odd index (e.g. 1, 3, etc.)
- Remove every 4th value, but only if it is odd.

# Subsetting Data Frames

# Basic subsetting

```
df = data.frame(x = 1:3, y=c("A","B","C"))
```

```
df[1, ]
```

```
##   x y
## 1 1 A
```

```
df[, 1]
```

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

```
df[1]
```

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

```
df[[1]]
```

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

```
df$x
```

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

```
str( df[1, ] )
```

```
## 'data.frame':	1 obs. of  2 variables:
##  $ x: int 1
##  $ y: Factor w/ 3 levels "A","B","C": 1
```

```
str( df[, 1] )
```

```
##  int [1:3] 1 2 3
```

```
str( df[1] )
```

```
## 'data.frame':	3 obs. of  1 variable:
##  $ x: int  1 2 3
```

```
str( df[[1]] )
```

```
##  int [1:3] 1 2 3
```

```
str( df$x )
```

```
##  int [1:3] 1 2 3
```

# Preserving vs Simplifying

Most of the time, R's `[` subset operator is a *preserving* operator, in that the returned object will have the same type/class as the parent. Confusingly, when used with some classes (e.g. data frame, matrix or array) `[` becomes a *simplifying* operator (does not preserve type) - this behavior is controlled by the `drop` argument.

```
x = data.frame(x = 1:3, y=c("A","B","C"))
```

```
x[1, ]
```

```
##   x y
## 1 1 A
```

```
x[1, , drop=TRUE]
```

```
## $x
## [1] 1
##
## $y
## [1] A
## Levels: A B C
```

```
x[1, , drop=FALSE]
```

```
##   x y
## 1 1 A
```

```
str(x[1, ])
```

```
## 'data.frame':    1 obs. of  2 variables:
##  $ x: int 1
##  $ y: Factor w/ 3 levels "A","B","C": 1
```

```
str(x[1, , drop=TRUE])
```

```
## List of 2
##  $ x: int 1
##  $ y: Factor w/ 3 levels "A","B","C": 1
```

```
str(x[1, , drop=FALSE])
```

```
## 'data.frame':    1 obs. of  2 variables:
##  $ x: int 1
##  $ y: Factor w/ 3 levels "A","B","C": 1
```

# Aside - Factor Subsetting

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

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

```r
x[1:2]
```

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

```r
x[1:3]
```

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

```r
x[1:2, drop=TRUE]
```

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

```r
x[1:3, drop=TRUE]
```

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

# Preserving vs Simplifying Subsets

| Type | Simplifying | Preserving |
|---|---|---|
| Atomic Vector | | `x[[1]]`<br>`x[1]` |
| List | `x[[1]]` | `x[1]` |
| Matrix / Array | `x[[1]]`<br>`x[1, ]`<br>`x[, 1]` | `x[1, , drop=FALSE]`<br>`x[, 1, drop=FALSE]` |
| Factor | `x[1:4, drop=TRUE]` | `x[1:4]`<br>`x[[1]]` |
| Data frame | `x[, 1]`<br>`x[[1]]` | `x[, 1, drop=FALSE]`<br>`x[1]` |

# Subsetting and assignment

# Subsetting and assignment

Subsets can also be used with assignment to update specific values within an object.

```
x = c(1, 4, 7)
```

```
x[2] = 2
x
```

```
## [1] 1 2 7
```

```
x[x %% 2 != 0] = x[x %% 2 != 0] + 1
x
```

```
## [1] 2 2 8
```

```
x[c(1,1)] = c(2,3)
x
```

```
## [1] 3 2 8
```

```r
x = 1:6
x[c(2,NA)] = 1
x
```

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

```r
x = 1:6
x[c(TRUE,NA)] = 1
x
```

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

```r
x = 1:6
x[c(-1,-3)] = 3
x
```

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

```r
x = 1:6
x[] = 6:1
x
```

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

# Subsets of Subsets

```
df = data.frame(a = c(5,1,NA,3))
```

```
df$a[df$a == 5] = 0
df
```

```
##    a
## 1  0
## 2  1
## 3 NA
## 4  3
```

```
df[1][df[1] == 3] = 0
df
```

```
##    a
## 1  0
## 2  1
## 3 NA
## 4  0
```

# (After Class) Exercise 2

Some data providers choose to encode missing values using values like `-999`. Below is a sample data frame with missing values encoded in this way.

```
d = data.frame(
  patient_id = c(1, 2, 3, 4, 5),
  age = c(32, 27, 56, 19, 65),
  bp = c(110, 100, 125, -999, -999),
  o2 = c(97, 95, -999, -999, 99)
)
```

- *Task 1* - using the subsetting tools we've discussed come up with code that will replace the `-999` values in the `bp` and `o2` column with actual `NA` values. Save this as `d_na`.
- *Task 2* - Once you have created `d_na` come up with code that translate it back into the original data frame `d`, i.e. replace the `NA`s with `-999`.

# Acknowledgments

Above materials are derived in part from the following sources:

- Hadley Wickham - Advanced R
- R Language Definition