

# Subsetting

Lecture 04

Dr. Colin Rundel

# Matrices and Arrays

# Matrices

R supports the creation of 2d matrix data structures using atomic vector types.

Generally these are formed via a call to `matrix()`.

```
1 matrix(1:4, nrow=2, ncol=2)
```

```
      [,1] [,2]  
[1,]    1    3  
[2,]    2    4
```

```
1 matrix(LETTERS[1:6], 2)
```

```
      [,1] [,2] [,3]  
[1,] "A"  "C"  "E"  
[2,] "B"  "D"  "F"
```

```
1 matrix(c(TRUE, FALSE), 2, 2)
```

```
      [,1] [,2]  
[1,] TRUE  TRUE  
[2,] FALSE FALSE
```

```
1 matrix(6:1 / 2, ncol = 2)
```

```
      [,1] [,2]  
[1,] 3.0  1.5  
[2,] 2.5  1.0  
[3,] 2.0  0.5
```

# Data ordering

Matrices in R use column major ordering (data is stored by *column*).

```
1 (m = matrix(1:6, nrow=2, ncol=3))
```

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

```
1 c(m)
```

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

```
1 (n = matrix(1:6, nrow=3, ncol=2))
```

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

```
1 c(n)
```

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

We can populate a matrix by row, but the data is still stored by column.

```
1 (x = matrix(1:6, nrow=2, ncol=3, byrow = TRUE))
```

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

```
1 c(x)
```

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

```
1 (y = matrix(1:6, nrow=3, ncol=2, byrow=TRUE))
```

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

```
1 c(y)
```

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

# Matrix structure

```
1 m = matrix(1:4, ncol=2, nrow=2)
```

```
1 typeof(m)
```

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

```
1 mode(m)
```

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

```
1 class(m)
```

```
[1] "matrix" "array"
```

```
1 attributes(m)
```

```
$dim  
[1] 2 2
```

Matrices (and arrays) are just atomic vectors with a `dim` attribute attached. They do not have an explicit class attribute, but have implicit class(es).

```
1 n = letters[1:6]  
2 dim(n) = c(2L, 3L)  
3 n
```

```
      [,1] [,2] [,3]  
[1,] "a"  "c"  "e"  
[2,] "b"  "d"  "f"
```

```
1 o = letters[1:6]  
2 attr(o,"dim") = c(2L, 3L)  
3 o
```

```
      [,1] [,2] [,3]  
[1,] "a"  "c"  "e"  
[2,] "b"  "d"  "f"
```

# Demo - S3 w/ matrices

```
1 report = function(x) {
2   UseMethod("report")
3 }
4 report.default = function(x) {
5   paste0("Class ", class(x), " not supported.")
6 }
7 report.double = function(x) {
8   "I'm a double!"
9 }
10 report.numeric = function(x) {
11   "I'm a numeric!"
12 }
13 report.matrix = function(x) {
14   "I'm a matrix!"
15 }
16 report.array = function(x) {
17   "I'm an array!"
18 }
```

```
1 #rm(report.double)
2 #rm(report.numeric)
3 #rm(report.matrix)
4 #rm(report.array)
5
6 report(matrix(1))
```

# Arrays

Arrays are just an  $n$ -dimensional extension of matrices and are defined by adding the appropriate dimension sizes.

```
1 array(1:8, dim = c(2,2,2))
```

```
, , 1
```

```
      [,1] [,2]  
[1,]    1    3  
[2,]    2    4
```

```
, , 2
```

```
      [,1] [,2]  
[1,]    5    7  
[2,]    6    8
```

```
1 array(letters[1:6], dim = c(1,2,3))
```

```
, , 1
```

```
      [,1] [,2]  
[1,] "a"  "b"
```

```
, , 2
```

```
      [,1] [,2]  
[1,] "c"  "d"
```

```
, , 3
```

```
      [,1] [,2]  
[1,] "e"  "f"
```

# Arrays & class()

A 2d array will have class `c("matrix", "array")` while 1d or >2d will only have class "array"

```
1 class(array(1, c(1,1)))
```

```
[1] "matrix" "array"
```

```
1 class(array(1, c(1)))
```

```
[1] "array"
```

```
1 class(array(1, c(1,1,1)))
```

```
[1] "array"
```

```
1 class(array(1, c(1,1,1,1)))
```

```
[1] "array"
```

# Data Frames

# Data Frames

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

```
1 (df = data.frame(  
2   x = 1:3,  
3   y = c("a", "b", "c"),  
4   z = c(TRUE)  
5 ))
```

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

```
1 str(df)
```

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

# Data Frame Structure

R represents data frames using a *list* of equal length *vectors*.

```
1 typeof(df)
```

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

```
1 class(df)
```

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

```
1 attributes(df)
```

```
$names
```

```
[1] "x" "y" "z"
```

```
$class
```

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

```
$row.names
```

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

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

```
List of 3
```

```
$ x: int [1:3] 1 2 3
```

```
$ y: chr [1:3] "a" "b" "c"
```

```
$ z: logi [1:3] TRUE TRUE TRUE
```

```
- attr(*, "row.names")= int [1:3] 1 2 3
```

# Build your own data.frame

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

```
1 attr(df,"class") = "data.frame"  
2 df
```

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

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

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

```
1 str(df)
```

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

```
1 is.data.frame(df)
```

```
[1] TRUE
```

# Strings (Characters) vs Factors

Previous to R v4.0, the default behavior of data frames was to convert character data into factors. Sometimes this was useful, but mostly it wasn't.

This behavior is controlled via the `stringsAsFactors` argument to `data.frame` (and related functions like `read.csv`, `read.table`, etc.).

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

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

```
1 str(df)
```

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

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

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

```
1 str(df)
```

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

# Length Coercion

When creating a data frame from different vectors, the lengths of the component vectors will be coerced to match. However, if they are not multiples of each other then there will be an error (other previous forms of length coercion would produce a warning for this case).

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

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

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

```
Error in `data.frame()` :
! arguments imply differing number of rows: 3, 2
```

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

```
Error in `data.frame()` :
! arguments imply differing number of rows: 3, 0
```

# Subsetting

# Subsetting in R

R has three subsetting operators (`[`, `[[`, and `$`). The behavior of these operators depends on the object (class) they are being used with (S3).

In general there are 6 different types of subsetting that can be performed based on the value passed to the operator,

- Positive integer
- Negative integer
- Logical value
- Empty / NULL
- Zero valued
- Character value (names)

# Positive Integer subsetting

Returns elements at the given location(s)

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

```
1 x[1]
```

```
[1] 1
```

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

```
[1] 1 7
```

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

```
[1] 1 1
```

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

```
[1] 1 4
```

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

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

```
List of 1  
 $ : num 1
```

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

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

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

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

```
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)

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

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

```
[1] 4 7
```

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

```
[1] 4
```

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

```
[1] 4 7
```

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

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

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

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

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

```
1 str( y[-1] )
```

```
List of 2
```

```
$ : num 4
```

```
$ : num 7
```

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

```
List of 1
```

```
$ : num 4
```

# Logical Value Subsetting

Returns elements that correspond to `TRUE` in the logical vector. Length of the logical vector is coerced to be the same as the vector being subsetted.

```
1 x = c(1,4,7,12)
```

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

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

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

```
[1] 1 7
```

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

```
[1] 4 12
```

```
1 y = list(1,4,7,12)
```

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

```
List of 3
```

```
$ : num 1
```

```
$ : num 4
```

```
$ : num 12
```

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

```
List of 2
```

```
$ : num 1
```

```
$ : num 7
```

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

```
Error in `y %% 2`:
```

```
! non-numeric argument to binary operator
```

# Empty Subsetting

Returns the original vector, this is not the same as subsetting with `NULL`

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

```
1 x[]
```

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

```
1 x[NULL]
```

```
numeric(0)
```

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

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

```
List of 3
```

```
$ : num 1
```

```
$ : num 4
```

```
$ : num 7
```

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

```
list()
```

# Zero subsetting

Returns an empty vector (of the same type), this is the same as subsetting with `NULL`

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

```
1 x[0]
```

```
numeric(0)
```

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

```
list()
```

`0`s can be mixed with either positive or negative integers for subsetting, and are ignored in both cases.

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

```
[1] 1
```

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

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

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

```
[1] 4 7
```

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

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

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

# Character subsetting

If the vector has names, selects 1st element whose names correspond to the value in the `names` attribute.

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

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

```
a  
1
```

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

```
a a  
1 1
```

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

```
b c  
4 7
```

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

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

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

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

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

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

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

# Out of bounds

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

```
1 x[4]
```

```
[1] NA
```

```
1 x[-4]
```

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

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

```
[1] NA
```

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

```
[1] 1 NA
```

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

```
1 str(y[4])
```

```
List of 1
```

```
$ : NULL
```

```
1 str(y[-4])
```

```
List of 3
```

```
$ : num 1
```

```
$ : num 4
```

```
$ : num 7
```

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

```
List of 1
```

```
$ : NULL
```

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

```
List of 2
```

```
$ : num 1
```

```
$ : NULL
```

# Missing values

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

```
1 x[NA]
```

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

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

```
[1] 1 NA
```

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

```
1 str(y[NA])
```

```
List of 3
```

```
$ : NULL
```

```
$ : NULL
```

```
$ : NULL
```

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

```
List of 2
```

```
$ : num 1
```

```
$ : NULL
```

# NULL and empty vectors (length 0)

This final type of subsetting follows the rules for length coercion with a 0-length vector (i.e. the vector being subset gets coerced to having length 0 if the subsetting vector has length 0)

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

```
1 x[NULL]
```

```
numeric(0)
```

```
1 x[integer()]
```

```
numeric(0)
```

```
1 x[character()]
```

```
numeric(0)
```

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

```
1 y[NULL]
```

```
list()
```

```
1 y[integer()]
```

```
list()
```

```
1 y[character()]
```

```
list()
```

# Subsetting and assignment

Subsets can also be used with assignment to update specific values within an object (in-place).

```
1 x = c(1, 4, 7, 9, 10, 15)
```

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

```
[1] 1 2 7 9 10 15
```

```
1 x %% 2 != 0
```

```
[1] TRUE FALSE TRUE TRUE FALSE TRUE
```

```
1 x[x %% 2 != 0] = (x[x %% 2 != 0] + 1) / 2  
2 x
```

```
[1] 1 2 4 5 10 8
```

# More assignment

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

```
[1] 3 2 4 5 10 8
```

```
1 x = 1:6
```

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

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

```
1 x = 1:6
```

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

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

```
1 x = 1:6
```

```
1 x[c(TRUE,NA)] = 1
2 x
```

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

```
1 x = 1:6
```

```
1 x[] = 1:3
2 x
```

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

# The other subset operators

## [ [ and \$

# Atomic vectors - [ vs. [[

[[ subsets like [ except it can only subset for a *single* value

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

```
1 x[1]
```

```
a  
1
```

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

```
[1] 1
```

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

```
[1] 1
```

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

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

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

```
[1] 1
```

# Generic Vectors (lists) - [ vs. [[

Subsets a single value, but returns the value - not a list containing that value.  
Multiple values are interpreted as nested subsetting.

```
1 y = list(a=1, b=4, c=7:9)
```

```
1 y[2]
```

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

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

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

```
1 y[[2]]
```

```
[1] 4
```

```
1 y[["b"]]
```

```
[1] 4
```

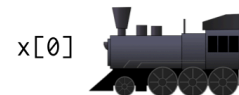
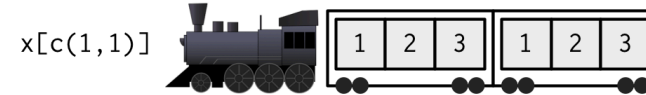
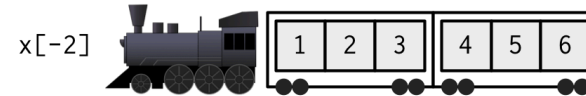
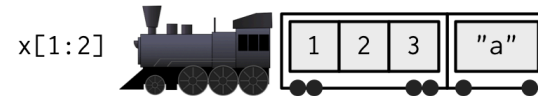
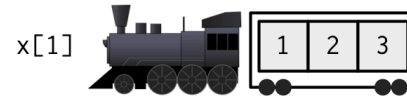
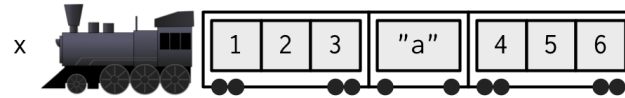
```
1 y[[1:2]]
```

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

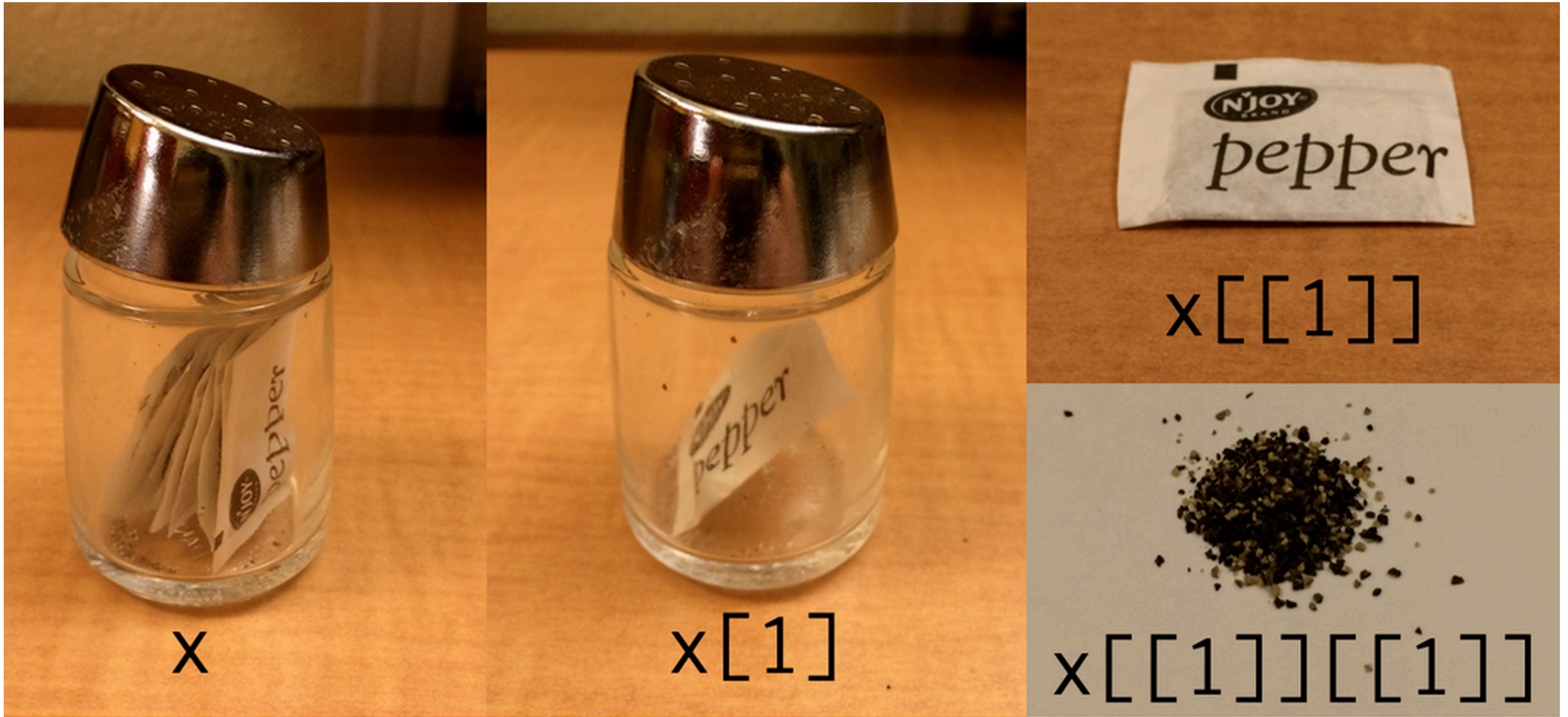
```
1 y[[2:1]]
```

```
[1] 4
```

# Hadley's Analogy (1)



# Hadley's Analogy (2)



**Hadley Wickham** @hadleywickham · 6h  
Indexing lists in [#rstats](#). Inspired by the Residence Inn

← ↻ 273 ★ 370 ⋮

# [[ vs. \$

\$ is equivalent to [[ but it only works for name based subsetting of *lists* (it also uses partial matching for names)

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

```
1 x$abc
```

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

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

```
1 y[["abc"]]
```

```
[1] 1
```

```
1 y$abc
```

```
[1] 1
```

```
1 y$d
```

```
[1] 5
```

# A common error

Why does the following code not work?

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

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

```
1 x$y
```

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

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

# Subsetting Data Frames

# Subsetting rows

As data frames have 2 dimensions, we can subset on either the rows or the columns  
- the subsetting values are separated by a comma.

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

```
  x y    z
1 1 A TRUE
2 2 B TRUE
3 3 C TRUE
```

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

```
  x y    z
1 1 A TRUE
```

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

```
'data.frame':  1 obs. of  3 variables:
 $ x: int 1
 $ y: chr "A"
 $ z: logi TRUE
```

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

```
  x y    z
1 1 A TRUE
3 3 C TRUE
```

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

```
'data.frame':  2 obs. of  3 variables:
 $ x: int  1 3
 $ y: chr  "A" "C"
 $ z: logi  TRUE TRUE
```

# Subsetting Columns

```
1 df
```

```
  x y   z
1 1 A TRUE
2 2 B TRUE
3 3 C TRUE
```

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

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

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

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

```
1 df[, 1:2]
```

```
  x y
1 1 A
2 2 B
3 3 C
```

```
1 str( df[, 1:2] )
```

```
'data.frame':  3 obs. of  2 variables:
 $ x: int  1 2 3
 $ y: chr  "A" "B" "C"
```

```
1 df[, -3]
```

```
  x y
1 1 A
2 2 B
3 3 C
```

```
1 str( df[, -3] )
```

```
'data.frame':  3 obs. of  2 variables:
 $ x: int  1 2 3
 $ y: chr  "A" "B" "C"
```

# Subsetting both

```
1 df
```

```
  x y   z
1 1 A TRUE
2 2 B TRUE
3 3 C TRUE
```

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

```
[1] 1
```

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

```
int 1
```

```
1 df[1:2, 1:2]
```

```
  x y
1 1 A
2 2 B
```

```
1 str( df[1:2, 1:2] )
```

```
'data.frame':  2 obs. of  2 variables:
 $ x: int  1 2
 $ y: chr  "A" "B"
```

```
1 df[-1, 2:3]
```

```
  y   z
2 B TRUE
3 C TRUE
```

```
1 str( df[-1, 2:3] )
```

```
'data.frame':  2 obs. of  2 variables:
 $ y: chr  "B" "C"
 $ z: logi TRUE TRUE
```

# Preserving vs Simplifying

Most of the time, R's `[]` subset operator is a *preserving* operator, in that the returned object will always have the same type/class as the object being subset.

Confusingly, when used with some classes (e.g. data frame, matrix or array) `[]` becomes a *simplifying* operator (does not preserve type) - this behavior is instead controlled by the `drop` argument.

# Drop w/ row subset

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

```
  x y   z  
1 1 A TRUE
```

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

```
$x  
[1] 1  
  
$y  
[1] "A"  
  
$z  
[1] TRUE
```

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

```
'data.frame':  1 obs. of  3 variables:  
 $ x: int 1  
 $ y: chr "A"  
 $ z: logi TRUE
```

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

```
List of 3  
 $ x: int 1  
 $ y: chr "A"  
 $ z: logi TRUE
```

# Drop w/ column subset

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

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

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

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

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

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

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

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

# Exceptions

`drop` only works when the resulting value can be represented as a 1d vector (either a list or atomic).

```
1 df[1:2, 1:2]
```

```
  x y  
1 1 A  
2 2 B
```

```
1 str(df[1:2, 1:2])
```

```
'data.frame':  2 obs. of  2 var  
 $ x: int  1 2  
 $ y: chr  "A" "B"
```

```
1 df[1:2, 1:2, drop=TRUE]
```

```
  x y  
1 1 A  
2 2 B
```

```
1 str(df[1:2, 1:2, drop=TRUE])
```

```
'data.frame':  2 obs. of  2 var  
 $ x: int  1 2  
 $ y: chr  "A" "B"
```

# Preserving vs Simplifying Subsets

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