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.

```r
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

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

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

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

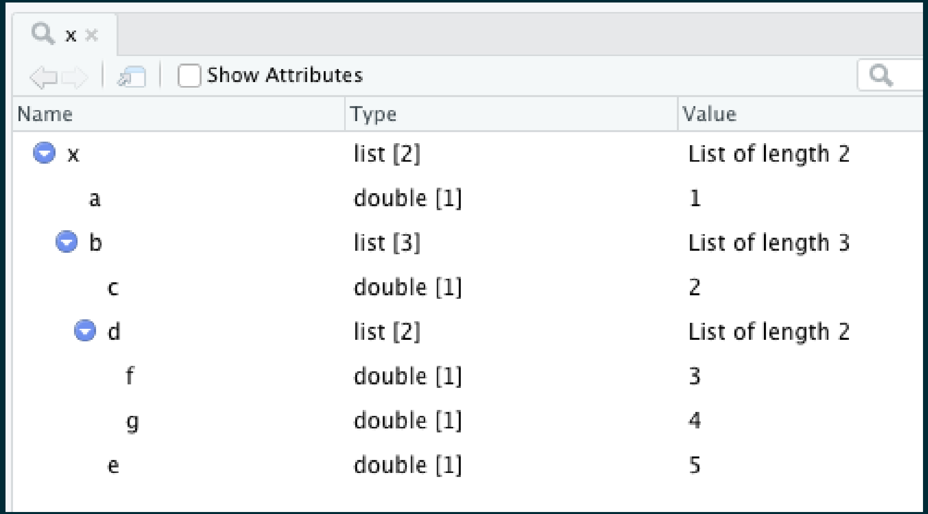
```
## List of 5
##  $ : chr "A"
##  $ : logi [1:2] TRUE FALSE
##  $ : num [1:4] 0.5 1 1.5 2
##  $ :List of 1
##   ..$ : int [1:2] 1 2
##  $ :function (x)
##   ..- attr(*, "srcref")= 'srcref' int [1:8] 1 51 1 65 51 65 1 1
##   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7fee64100310>
```

Lists as "trees"

Lists can contain other lists, meaning they can have a hierarchical (tree-like) structure

```
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
```



□ Show Attributes

Name	Type	Value
x	list [2]	List of length 2
a	double [1]	1
b	list [3]	List of length 3
c	double [1]	2
d	list [2]	List of length 2
f	double [1]	3
g	double [1]	4
e	double [1]	5

Heirarchical Data

```
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 = FA
```

```
## 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"
```

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
```

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

```
str(df)
```

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

```r
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
```

Roll your own data.frame

```
df2 = list(x = 1:3, y = 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: chr  "a" "b" "c"
##  $ z: logi  TRUE TRUE TRUE
```

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

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

Strings (Characters) vs Factors

Previous to R v4.0.0, the default behavior of data frames was to convert character data into factors. Sometimes this was useful, but mostly it wasn'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.).

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

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

```r
str(df)
```

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

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
```

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.

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

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

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
```

```r
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
```

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

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

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

```
## [[1]]
## [1] 4
## 
## [[2]]
## [1] 7
```

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

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

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

```
x[-4]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

Missing and NULL

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

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

```
x[NULL]
```

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

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

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

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

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

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

```
##  list()
```

```
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)
```

```
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
```

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

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

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

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

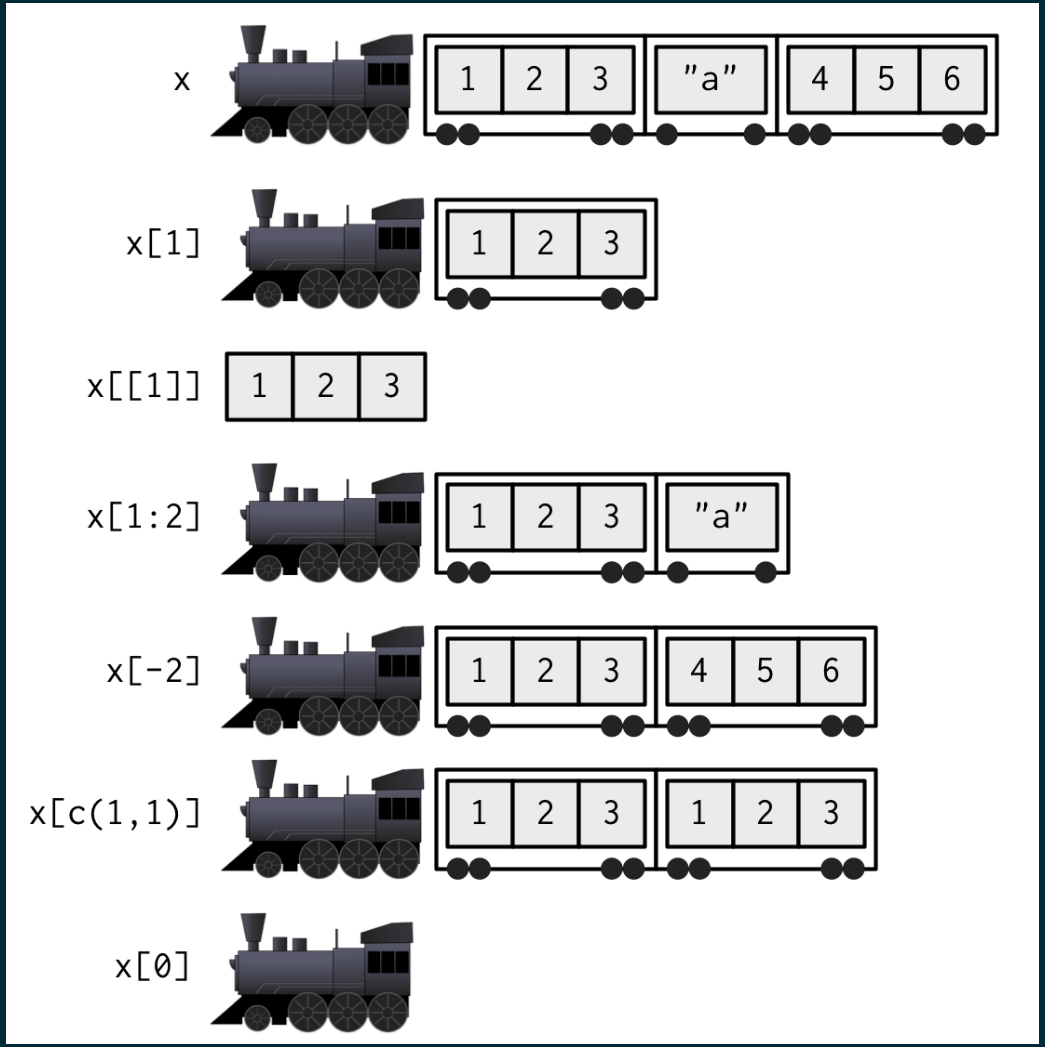
```
y[[1:2]]
```

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

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

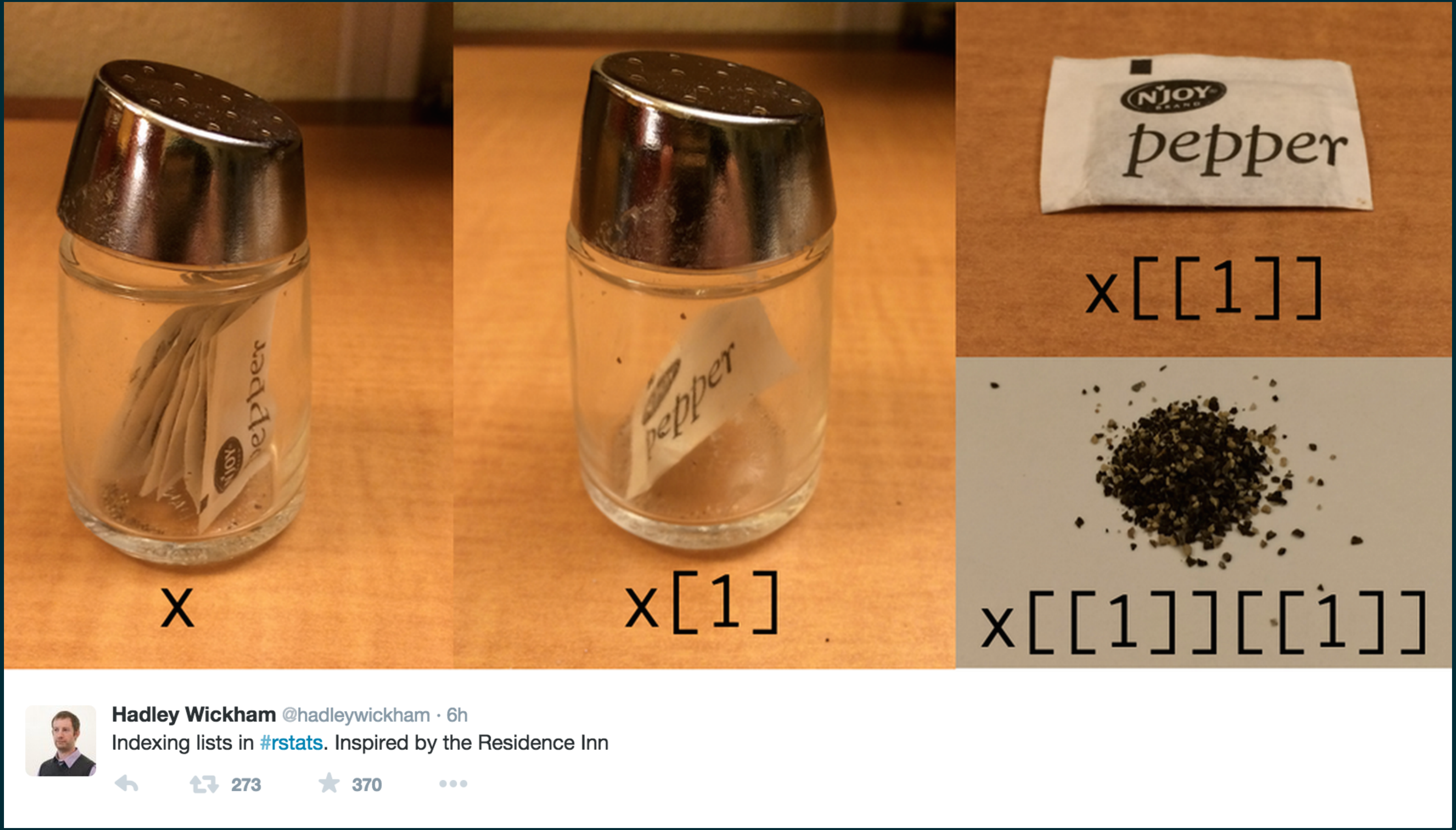
```
## [1] 4
```

Hadley's Analogy (1)



From Advanced R - Chapter 4.3

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 named *lists* and it has a terrible default behavior where it uses partial matching for names.

```
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 error

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
```

Subsetting Data Frames

Basic subsetting

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

```
(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
```

```
df[1, ]
```

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

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

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

```
df[, 1]
```

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

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

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

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

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

Subsetting Rows

```
df
```

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

```
df[, 1]
```

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

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

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

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

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

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

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

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

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

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

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

Subsetting both

```r
df
```

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

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

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

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

```
##  int 1
```

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

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

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

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

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

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

```r
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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 %% 2 != 0
```

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

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

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

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

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

```r
x = 1:6
x[] = 1:3
x
```

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

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
```

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