Data frames, matrices, & subsetting

Lecture 05

Dr. Colin Rundel

Matrices and Arrays

Matrices

R supports the creation of 2d data structures (rows and columns) of atomic vector types.

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

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

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

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

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

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

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

```r
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 in column order not row order).

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

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

```r
c(m)
```

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

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

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

```r
c(n)
```

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

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

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

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

```r
c(x)
```

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

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

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

```r
c(y)
```

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

Matrix structure

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

```r
typeof(m)
```

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

```r
class(m)
```

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

```r
mode(m)
```

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

```r
attributes(m)
```

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

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

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

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

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

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

Arrays

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

```r
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
```

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

```
, , 1

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

, , 2

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

, , 3

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

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

```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*.

```r
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
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
```

Build your own data.frame

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

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

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

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

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

```r
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
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.).

```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
```

```r
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
```

```r
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 not multiples of each other then there will be an error (other previous forms of length coercion would produce a warning for this case).

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

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

```r
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
```

```r
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 subsetting that can be performed:

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

Positive Integer subsetting

Returns elements at the given location(s)

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

```r
x[1]
```

```
[1] 1
```

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

```
[1] 1 7
```

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

```
[1] 1 1
```

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

```
[1] 1 4
```

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

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

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

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

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

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

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

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

Note - R uses a 1-based indexing scheme

Negative Integer subsetting

Excludes elements at the given location(s)

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

```r
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)
```

```r
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 coerced to be the same as the vector being subsetted.

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

```r
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)
```

```r
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, this is not the same as subsetting with `NULL`

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

```r
x[]
```

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

```r
x[NULL]
```

```
numeric(0)
```

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

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

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

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

```
 list()
```

Zero subsetting

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

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

```r
x[0]
```

```
numeric(0)
```

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

```
list()
```

0s can be mixed with either positive or negative integers for subsetting (they are ignored)

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

```
[1] 1
```

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

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

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

```
[1] 4 7
```

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

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

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

Character subsetting

If the vector has names, selects elements whose names correspond to the values in the name vector.

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

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

```
a 
1 
```

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

```
a a 
1 1 
```

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

```
b c 
4 7 
```

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

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

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

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

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

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

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

Out of bounds

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

```r
x[4]
```

```
[1] NA
```

```r
x[-4]
```

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

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

```
[1] NA
```

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

```
[1]  1 NA
```

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

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

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

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

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

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

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

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

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

Missing values

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

```r
x[NA]
```

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

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

```
[1]  1 NA
```

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

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

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

```r
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)

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

```r
x[NULL]
```

```
numeric(0)
```

```r
x[integer()]
```

```
numeric(0)
```

```r
x[character()]
```

```
numeric(0)
```

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

```r
y[NULL]
```

```
list()
```

```r
y[integer()]
```

```
list()
```

```r
y[character()]
```

```
list()
```

The other subset operators ([[and $)

Atomic vectors - [vs. [[

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

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

```r
x[1]
```

```
a 
1 
```

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

```
[1] 1
```

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

```
[1] 1
```

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

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

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

```
[1] 1
```

Generic Vectors (lists) - [vs. [[

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

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

```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
```

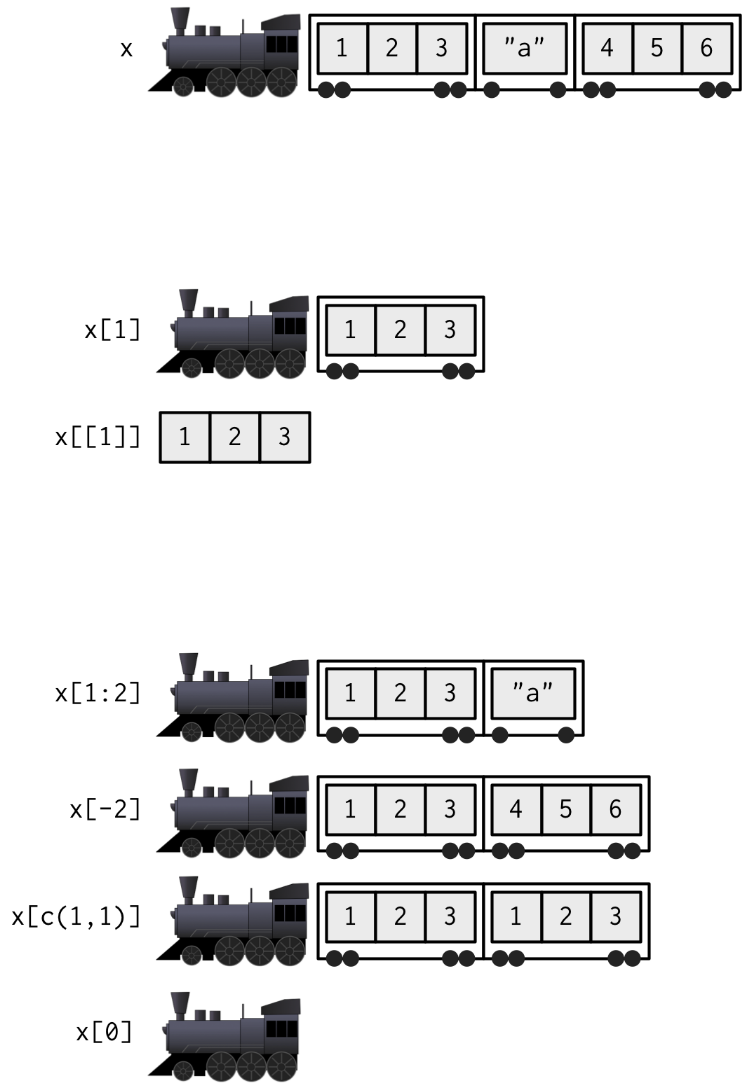
```r
y[[1:2]]
```

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

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

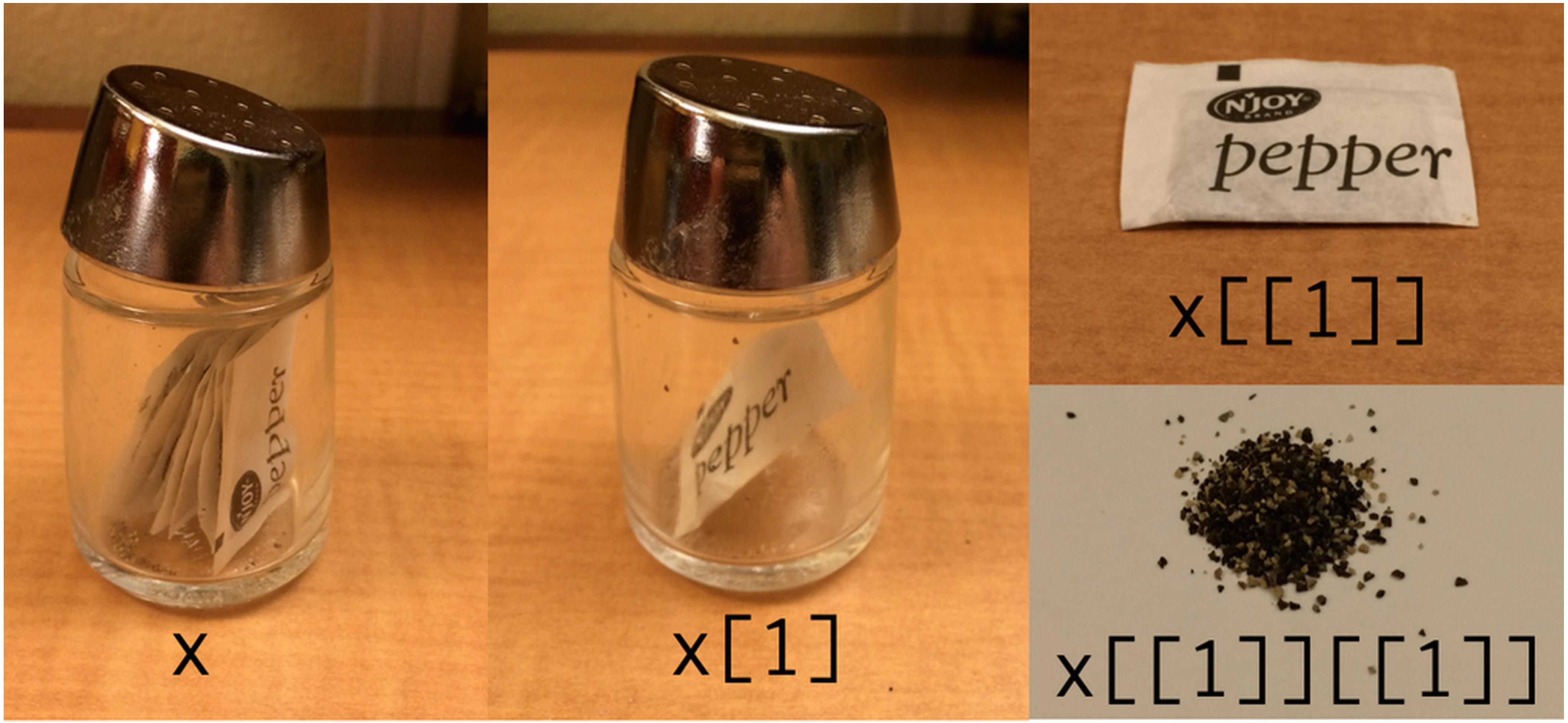
```
[1] 4
```

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 name based subsetting of *named lists* (also it uses partial matching for names)

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

```r
x$abc
```

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

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

```r
y[["abc"]]
```

```
[1] 1
```

```r
y$abc
```

```
[1] 1
```

```r
y$d
```

```
[1] 5
```

A common error

Why does the following code not work?

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

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

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

```r
x$y
```

```
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]]`.

Exercise 1

Below are 100 values,

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

05:00

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.

```r
(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
```

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

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

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

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

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

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

```r
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

```r
df
```

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

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

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

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

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

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

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

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

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

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

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

```r
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.

Drop

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

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

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

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

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

```
$x
[1] 1

$y
[1] "A"

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

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

Exceptions

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

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

```
  x y
1 1 A
2 2 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"
```

Preserving vs Simplifying Subsets

Type	Simplifying	Preserving
Atomic Vector	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`

Matrix / Array	`x[[1]]`
	`x[1, ]`
	`x[, 1]`
		`x[1, , drop=FALSE]`
		`x[, 1, drop=FALSE]`

Factor	`x[1:4, drop=TRUE]`	`x[1:4]`
		`x[[1]]`

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