SM-2302 Software for Mathematicians

R2: Matrices and data frames

Dr. Haziq Jamil
Mathematical Sciences, Faculty of Science, UBD
`https://haziqj.ml`

Semester I 2023/24

Lists

Lists are the other 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
##
##
## [[5]]
## function(x) x ^ 2
```

List structure

Often we want a more compact representation of a complex object, the `str()` function is useful for this, particularly for lists.

```
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
##   ..$ : num 1
##  $ :function (x)
##   ..- attr(*, "srcref")= 'srcref' int [1:8] 1 55 1 69 55 69 1 1
##   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7fab9f132ef0>
```

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 lists inside your base list).

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?"
```

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"
```

0-length coercion

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

```r
integer() + 1
```

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

```r
logical() | TRUE
```

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

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

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

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

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

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

```r
NULL + 1
```

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

```r
NULL | TRUE
```

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

```r
NUL '
```

```r
log(NULL)
```

NULLs and comparison

Given the previous issue, comparisons and conditional with NULLs can be problematic.

```r
x <- NULL
if (x > 0) print("Hello")
```

```
## Error in if (x > 0) print("Hello"): argument is of length zero
```

```r
if (!is.null(x) & (x > 0)) print("Hello")
```

```
## Error in if (!is.null(x) & (x > 0)) print("Hello"): argument is of length zero
```

```r
if (!is.null(x) && (x > 0)) print("Hello")
```

This is due to short circuit evaluation which occurs with && and || but not & or |.

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

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

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

```r
names(x)
```

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

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

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

```r
names(x)
```

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

Helpers functions vs `attr()`

```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"
```

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

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

```r
attributes(x)
```

```
## $names
## [illegible]" "3"
```

Factors

Factor objects are how R represents categorical data (e.g. a variable where there are a fixed # of possible outcomes).

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

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

```r
str(x)
```

```
##  Factor w/ 3 levels "Cloudy","Rainy",..: 3 1 2 1 1
```

```r
typeof(x)
```

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

```r
mode(x)
```

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

Composition

A factor is just an integer vector with two attributes: `class` and `levels`.

```
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
```

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 sorted in column order not row order).

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

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

```
c(x)
```

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

When creating the matrix we can populate the matrix via row, but that data will still be stored in column major order.

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

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

```
c(y)
```

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

Matrix structure

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

```
x <- letters[1:6]
dim(x) <- c(2L, 3L)

x
```

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

```
typeof(x)
```

```
## [1] "character"
```

```
mode(x)
```

```
##          cter"
```

```
class(x)
```

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

```
attributes(x)
```

```
## $dim
## [1] 2 3
```

Arrays

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

```
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"
```

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

```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
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
attributes(df)
```

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

```r
class(df)
```

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

```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

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

```
attr(df, "class") <- "data.frame"
df
```

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

```
attr(df, "row.names") <- 1:3
df
```

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

```
str(df)
```

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

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

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

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,
```

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

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

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)]
```

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

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

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

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

Negative integer subsetting

Excludes elements at the given location(s)

```
x[-1]
```

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

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

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

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

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

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

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

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

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

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

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

```
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.

```
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] )
```

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`.

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

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

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

```
##  list()
```

0s can be mixed with either positive or negative integers for subsetting, in which case they are ignored.

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

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

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

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

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

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

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

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

Character subsetting

If the vector has names, selects 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
```

Out of bounds

```r
x <- c(1, 4, 7)
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)
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
```

Missing

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

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

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

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

Atomic vectors–[vs. [[

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

```
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
```

Generic Vectors–[vs. [[

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

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

```
y[2]
```

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

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

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

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

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

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

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

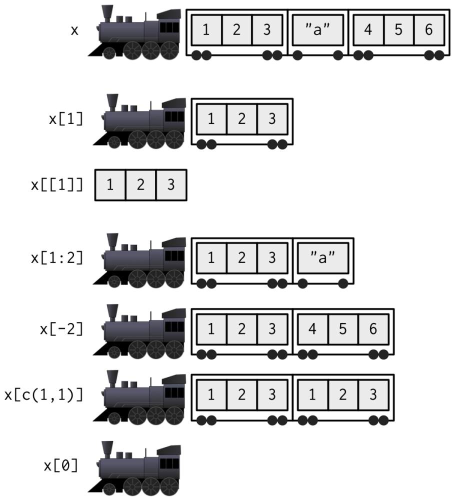
```
y[[1:2]]
```

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

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

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

Hadley's Analogy



From Advanced R, Chapter 4.3.

[[vs. $

$ is equivalent to [[but it only works for named *lists* and it uses partial matching for names.

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

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

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

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

```r
y$abc
```

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

```r
y$d
```

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

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[NULL, ]
```

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

```
df[NA, ]
```

```
##       x    y  z
## NA   NA <NA> NA
## NA.1 NA <NA> NA
```

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
df[, 1:2]
```

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

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

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

```r
df[, NULL]
```

```
## data frame with 0 columns and 3 rows
```

```r
df[, NA]
```

```
## Error in `[.data.frame`(df, , NA): undefined
```

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

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

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

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

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

```
## integer(0)
```

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

```
## data frame with 0 columns and 1 row
```

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.

```
df[1, ]
```

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

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

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

```
df[, 1]
```

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

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

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

Preserving vs simplifying subsets

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

Subsetting and assignment

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

```
x <- c(1, 4, 7, 9, 10, 15)
x[2] <- 2
x
```

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

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

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

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

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

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

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

Subsetting and assignment (cont.)

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

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

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

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

```r
x[c(TRUE, NA)] <- 1
x
```

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

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

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