Data structures and subsetting

Programming for Statistical Science

Shawn Santo

Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Companion videos

- Git from the command line

Additional resources

- Sections 3.3 - 3.4 Advanced R
- Chapter 4 Advanced R

Recall

Atomic vector creation

We can use functions such as `c()`, `vector()`, and `:` to create atomic vectors.

```
c(5, 10, pi, 0, -sqrt(3))
```

```
#> [1]  5.000000 10.000000  3.141593  0.000000 -1.732051
```

```
vector(mode = "character", length = 4)
```

```
#> [1] "" "" "" ""
```

```
vector(mode = "integer", length = 3)
```

```
#> [1] 0 0 0
```

```
-10:-3
```

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

Generic vector creation

Function `list()` allows us to create a generic vector.

```
x <- list(
    a         = -100:100,
    b         = list(lower = letters, upper = LETTERS),
    cars_data = cars
  )

str(x)
```

```
#> List of 3
#>  $ a         : int [1:201] -100 -99 -98 -97 -96 -95 -94 -93 -92 -91 ...
#>  $ b         :List of 2
#>   ..$ lower: chr [1:26] "a" "b" "c" "d" ...
#>   ..$ upper: chr [1:26] "A" "B" "C" "D" ...
#>  $ cars_data:'data.frame':    50 obs. of  2 variables:
#>   ..$ speed: num [1:50] 4 4 7 7 8 9 10 10 10 11 ...
#>   ..$ dist : num [1:50] 2 10 4 22 16 10 18 26 34 17 ...
```

Attributes

Data structures

You may have heard of factors, matrices, arrays, and date-times. These are just atomic vectors with special attributes.

- Attributes attach metadata to an object.
- Function `attr()` can retrieve and modify a single attribute.

```
attr(x, which) # get attribute
attr(x, which) <- value # set / modify attribute
```

- Function `attributes()` can retrieve and set attributes en masse.

```
attributes(x) # get attributes
attributes(x) <- value # set / modify attributes
```

Attribute: `names`

Get or set the names of an object.

One option:

```
x <- 1:4
attributes(x)
```

```
#> NULL
```

```
attr(x = x, which = "names") <- c("a", "b", "c", "d")
attributes(x)
```

```
#> $names
#> [1] "a" "b" "c" "d"
```

```
x
```

```
#> a b c d
#> 1 2 3 4
```

Another option:

```
a <- 1:4
names(a) <- c("a", "b", "c", "d")
attributes(a)
```

```
#> $names
#> [1] "a" "b" "c" "d"
```

```
a
```

```
#> a b c d
#> 1 2 3 4
```

Either method is okay to use, but stick with using the replacement function.

Attribute: `dim`

Get or set the dimension of an object.

```
z <- 1:9
z
```

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

```
attr(x = z, which = "dim") <- c(3, 3)
attributes(z)
```

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

```
z
```

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

We have a 3 x 3 matrix.

```r
y <- matrix(z, nrow = 3, ncol = 3)
attributes(y)
```

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

```r
y
```

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

Exercise

Create a 3 x 3 x 2 array using the `dim` attribute with the vector below.

```
x <- c(5, 1, 5, 5, 1, 1, 5, 3, 2, 3, 2, 6, 4, 4, 1, 2, 1, 3)
```

Try to create the same array using function `array()`. What do you notice about how the array object is populated?

Factors

Factors are built on top of integer vectors with two attributes: `class` and `levels`. Factors are how R stores and represents categorical data.

A quick way to create a categorical variable as a factor is with function `factor()`.

```
x <- factor(c("walk", "single", "double", "triple", "home run"))
x
```

```
#> [1] walk     single   double   triple   home run
#> Levels: double home run single triple walk
```

```
typeof(x)
```

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

```
attributes(x)
```

```
#> $levels
#> [1] "double"   "home run" "single"   "triple"   "walk"
#>
#> $class
#> [1] "factor"
```

Ordered factors

To induce an ordering we can use function `ordered()` as opposed to `factor()`.

```
y <- ordered(c("walk", "single", "double", "triple", "home run"),
        levels = c("walk", "single", "double", "triple", "home run"))
y
```

```
#> [1] walk     single   double   triple   home run
#> Levels: walk < single < double < triple < home run
```

```
attributes(y)
```

```
#> $levels
#> [1] "walk"     "single"   "double"   "triple"   "home run"
#>
#> $class
#> [1] "ordered" "factor"
```

```
str(y)
```

```
#>  Ord.factor w/ 5 levels "walk"<"single"<..: 1 2 3 4 5
```

Exercise

Create a factor vector based on the vector of airport codes below. Try to do it without using function `factor()`.

```
airports <- c("RDU", "ABE", "DTW", "GRR", "RDU", "GRR", "GNV",
              "JFK", "JFK", "SFO", "DTW")
```

Assume all the possible levels are

```
c("RDU", "ABE", "DTW", "GRR", "GNV", "JFK", "SFO")
```

Hint: Think about what type of object factors are built on.

What if the possible levels are

```
c("RDU", "ABE", "DTW", "GRR", "GNV", "JFK", "SFO", "GSO", "ORD", "PHL")
```

Matrices and arrays

- Homogeneous in their type.
- Matrices are populated based on column major ordering (use `byrow` argument to change this).
- Arrays can have one, two or more dimensions.

Data frames

Data frames are built on top of lists with attributes: `names`, `row.names`, and `class`. Here the class is `data.frame`.

```
typeof(longley)
```

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

```
attributes(longley)
```

```
#> $names
#> [1] "GNP.deflator" "GNP"          "Unemployed"   "Armed.Forces" "Population"
#> [6] "Year"         "Employed"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#>  [1] 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 196
#> [16] 1962
```

Here `names` refers to variable names.

Data frame characteristics

- Data frames can be heterogeneous across columns.
- Data frames are rectangular in structure (not always tidy).
- They have column names and row names.
- Data frames can be subset by name or position.

Data frame creation by setting attributes

Start with a list

```
x <- list(c("48501", "48507", "48505"),
          c(3, 4, 21),
          c(2, 1, 2))
str(x)
```

```
#> List of 3
#>  $ : chr [1:3] "48501" "48507" "48505"
#>  $ : num [1:3] 3 4 21
#>  $ : num [1:3] 2 1 2
```

Add attributes

```
attributes(x) <- list(class     = "data.frame",
                      names     = c("zip", "lead_value", "time"),
                      row.names = 1:3)
```

Then we have a data frame

```
x
```

```
#>     zip lead_value time
#> 1 48501          3    2
#> 2 48507          4    1
#> 3 48505         21    2
```

```
str(x)
```

```
#> 'data.frame':    3 obs. of  3 variables:
#>  $ zip       : chr  "48501" "48507" "48505"
#>  $ lead_value: num  3 4 21
#>  $ time      : num  2 1 2
```

Of course, we could have used function `data.frame()` to create our data frame object. There is also function `tidyverse::tibble()` - it creates a tibble object. Similar to a data frame but with two addition class components.

Length coercion

Coercion is slightly different for data frames.

```
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
```

If a shorter vector is not a multiple of the longest vector an error will occur.

What do you think will happen here?

```
data.frame(num       = 1:6,
           treatment = c(0, 10, 20),
           type      = c("a", "b"))
```

Summary

Data Structure	Built On	Attribute(s)	Quick creation
Matrix, Array	Atomic vector	`dim`	`matrix()`, `array()`
Factor	Atomic integer vector	`class`, `levels`	`factor()`, `ordered()`
Date	Atomic double vector	`class`	`as.Date()`
Date-times	Atomic double vector	`class`	`as.POSIXct()`, `as.POSIXlt()`
Data frame	List	`class`, `names`, `row.names`	`data.frame()`

Subsetting

Subsetting techniques

R has three operators (functions) for subsetting:

1. `[`
2. `[[`
3. `$`

Which one you use will depend on the object you are working with, its attributes, and what you want as a result.

We can subset with

- integers
- logicals
- `NULL`, `NA`
- character values

Numeric (positive) subsetting

Indexing begins at 1, not 0.

```
x <- c("NC", "SC", "VA", "TN")
y <- list(states  = x, rank = 1:4, message = "")
```

Atomic vector

```
x[1]
```

```
#> [1] "NC"
```

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

```
#> [1] "NC" "VA"
```

```
x[c(1:5)]
```

```
#> [1] "NC" "SC" "VA" "TN" NA
```

```
x[c(2.2, 3.9)]
```

```
#> [1] "SC" "VA"
```

List

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

```
#> List of 1
#>  $ states: chr [1:4] "NC" "SC" "VA" "TN"
```

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

```
#> List of 2
#>  $ states : chr [1:4] "NC" "SC" "VA" "TN"
#>  $ message: chr ""
```

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

```
#> List of 4
#>  $ states : chr [1:4] "NC" "SC" "VA" "TN"
#>  $ rank   : int [1:4] 1 2 3 4
#>  $ message: chr ""
#>  $ NA     : NULL
```

Numeric (negative) subsetting

```
x <- c("NC", "SC", "VA", "TN")
y <- list(states = x, rank = 1:4, message = "")
```

Atomic vector

```
x[-1]
```

```
#> [1] "SC" "VA" "TN"
```

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

```
#> [1] "SC" "TN"
```

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

```
#> Error in x[c(-1, 3)]: only 0's may be mixed with
```

```
x[-c(2.2, 3.9)]
```

```
#> [1] "NC" "TN"
```

List

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

```
#> List of 2
#>  $ rank   : int [1:4] 1 2 3 4
#>  $ message: chr ""
```

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

```
#> List of 1
#>  $ rank: int [1:4] 1 2 3 4
```

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

```
#> Error in y[c(-1, 3)]: only 0's may be mixed with nega
```

```
str(y[-c(2.2, 3.9)])
```

```
#> List of 2
#>  $ states : chr [1:4] "NC" "SC" "VA" "TN"
#>  $ message: chr ""
```

Logical subsetting

It returns elements that correspond to `TRUE` in the logical vector. The length of the logical vector is expected to be of the same length as the vector being subset.

Atomic vector

```
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
```

List

```
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

It 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 as the vector being subset.

```
x <- c(1,4,7)
y <- list(1,4,7)
```

```
x[0]
```

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

```
str(y[0])
```

```
#>  list()
```

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

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

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

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

Character subsetting

If a vector has names, you can select elements whose names correspond to the character vector.

Atomic vector

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

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

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

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

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

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

List

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

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

Missing and NULL subsetting

Atomic vector

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

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

```
x[NULL]
```

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

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

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

List

```
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
```

Exercise

Consider the vectors `x` and `y` below.

```
x <- letters[1:5]
y <- list(i = 1:5, j = -3:3, k = rep(0, 4))
```

What is difference between subsetting with `[` and `[[` using integers? Try various indices.

Understanding [vs. [[with lists

How do you get a shopping cart with only the cheese and bananas?

How do you get the bananas out of the cart?

Using `$` for subsetting lists

The `$` operator only works with named lists and works similar to `[[`.

```
x <- list(a   = 1:3,
          ab  = 4:6,
          abc = 7:9)
x
```

```
#> $a
#> [1] 1 2 3
#>
#> $ab
#> [1] 4 5 6
#>
#> $abc
#> [1] 7 8 9
```

```
x$a
```

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

```
x$ab
```

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

```
y <- list(a    = 1:3,
          abc  = 4:6,
          abde = 7:9)
y
```

```
#> $a
#> [1] 1 2 3
#>
#> $abc
#> [1] 4 5 6
#>
#> $abde
#> [1] 7 8 9
```

```
y$a
```

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

```
y$abd
```

```
#> [1] 7 8 9
```

References

1. Wickham, H. (2020). Advanced R. https://adv-r.hadley.nz/