Data types

Daniel Anderson

Week 1, Class 2

Agenda

- Finishing up on coercion
- Attributes
- Missing values
- Intro to lists
- Subsetting

Learning objectives

- Understand the fundamental difference between lists and atomic vectors
- Understand how atomic vectors are coerced, implicitly or explicitly
- Understand various ways to subset vectors, and how subsetting differs for lists
- Understand what an attribute is, and how to set and modify attributes

Pop quiz

Without actually running the code, predict which type each of the following will coerce to.

```
c(TRUE, 1L, 0L, "False")
c(1L, FALSE)
c(7L, 6.23, "eight")
c(1.25, TRUE, 4L)
```

01:00

Answers

```r
typeof(c(TRUE, 1L, 0L, "False"))
```

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

```r
typeof(c(1L, FALSE))
```

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

```r
typeof(c(7L, 6.23, "eight"))
```

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

```r
typeof(c(1.25, TRUE, 4L))
```

```
## [1] "double"
```

Challenge

Work with a partner

One of you share your screen:

- Create four atomic vectors, one for each of the fundamental types
- Combine two or more of the vectors. Predict the implicit coercion of each.
- Apply explicit coercions, and predict the output for each.

(basically quiz each other)

08:00

Attributes

Attributes

- What are attributes?
    - metadata... what's metadata?
    - Data about the data

Other data types

Atomic vectors by themselves make up only a small fraction of the total number of data types in R

What are some other data types?

- Data frames
- Matrices & arrays
- Factors
- Dates

Remember, atomic vectors are the atoms of R. Many other data structures are built from atomic vectors.

- We use attributes to create other data types from atomic vectors

Attributes

Common

- Names
- Dimensions

Less common

- Arbitrary metadata

Examples

Please follow along!

- See **all** attributes associated with a give object with `attributes`

```r
library(palmerpenguins)
attributes(penguins[1:50, ]) # limiting rows just for slides
```

```
## $names
## [1] "species"           "island"            "bill_length_mm"    "bill_de
## [6] "body_mass_g"       "sex"               "year"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 2
## [35] 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
```

```r
head(penguins)
```

```
## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_
##   <fct>   <fct>              <dbl>         <dbl>             <int>
## 1 Big one Torgersen         39.1          18.7                 181
## 2 Big one Torgersen         39.5          17.400               186
## 3 Big one Torgersen         40.300        18                   195
## 4 Big one Torgersen         NA            NA                    NA
## 5 Big one Torgersen         36.7          19.3                 193
## 6 Big one Torgersen         39.300        20.6                 190
```

Get specific attribute

- Access just a single attribute by naming it within `attr`

```
attr(penguins, "class")
```

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

```
attr(penguins, "names")
```

```
## [1] "species"           "island"            "bill_length_mm"    "bill_de
## [6] "body_mass_g"       "sex"               "year"
```

Note – this is not generally how you would pull the names attribute. Rather, you would use `names()`.

Be specific

- Note in the prior slides, I'm asking for attributes on the entire data frame.

- Is that what I want?... maybe. But the individual vectors may have attributes as well

```
attributes(penguins$species)
```

```
## $levels
## [1] "Big one"    "Little one" "Funny one"
##
## $class
## [1] "factor"
```

```
attributes(penguins$bill_length_mm)
```

```
## NULL
```

Set attributes

- Just redefine them within `attr`

```
attr(penguins$species, "levels") <- c("Big one",
                                      "Little one",
                                      "Funny one")

head(penguins)
```

```
## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_
##   <fct>   <fct>              <dbl>         <dbl>             <int>
## 1 Big one Torgersen         39.1          18.7                 181
## 2 Big one Torgersen         39.5          17.400               186
## 3 Big one Torgersen         40.300        18                   195
## 4 Big one Torgersen         NA            NA                    NA
## 5 Big one Torgersen         36.7          19.3                 193
## 6 Big one Torgersen         39.300        20.6                 190
```

Note – you would generally not define levels this way, but it is a general method for modifying attributes.

Dimensions

- Let's create a matrix (please do it with me)

```
m <- matrix(1:6, ncol = 2)
m
```

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

- Notice how the matrix fills
- Check out the attributes

```
attributes(m)
```

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

Modify the attributes

- Let's change it to a 2 x 3 matrix, instead of 3 x 2 (you try first)

```
attr(m, "dim") <- c(2, 3)
m
```

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

- is this the result you expected?

Alternative creation

- Create an atomic vector, assign a dimension attribute

```r
v <- 1:6
v
```

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

```r
attr(v, "dim") <- c(3, 2)
v
```

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

Aside

- What if we wanted it to fill by row?

```r
matrix(6:13,
       ncol = 2,
       byrow = TRUE)
```

```
##      [,1] [,2]
## [1,]    6    7
## [2,]    8    9
## [3,]   10   11
## [4,]   12   13
```

```r
vect <- 6:13
dim(vect) <- c(2, 4)
vect
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    6    8   10   12
## [2,]    7    9   11   13
```

```r
t(vect)
```

```
##      [,1] [,2]
## [1,]    6    7
## [2,]    8    9
## [3,]   10   11
## [4,]   12   13
```

Names

- The following (this slide and the next) are equivalent

```r
attr(v, "dimnames") <- list(c("the first", "second", "III"),
                            c("index", "value"))
v
```

```
##            index value
## the first      1     4
## second         2     5
## III            3     6
```

Names

```r
v2 <- 1:6
attr(v2, "dim") <- c(3, 2)
rownames(v2) <- c("the first", "second", "III")
colnames(v2) <- c("index", "value")
v2
```

```
##           index value
## the first     1     4
## second        2     5
## III           3     6
```

Arbitrary metadata

- I don't use this often (wouldn't recommend you do either)

```
attr(v, "matrix_mean") <- mean(v)
v
```

```
##              index value
## the first        1     4
## second           2     5
## III              3     6
## attr(,"matrix_mean")
## [1] 3.5
```

```
attr(v, "matrix_mean")
```

```
## [1] 3.5
```

- Note that *anything* can be stored as an attribute (including matrices or data frames, etc.)

Matrices vs Data frames

Usually we want to work with data frames because they represent our data better.

Sometimes a matrix is more efficient because you can operat on the **entire** matrix at once.

```
set.seed(42)
m <- matrix(rnorm(100, 200, 10), ncol = 10)
m
```

```
##             [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]
##  [1,] 213.7096 213.0487 196.9336 204.5545 202.0600 203.2193 196.3277 18
##  [2,] 194.3530 222.8665 182.1869 207.0484 196.3894 192.1616 201.8523 19
##  [3,] 203.6313 186.1114 198.2808 210.3510 207.5816 215.7573 205.8182 20
##  [4,] 206.3286 197.2121 212.1467 193.9107 192.7330 206.4290 213.9974 19
##  [5,] 204.0427 198.6668 218.9519 205.0496 186.3172 200.8976 192.7271 19
##  [6,] 198.9388 206.3595 195.6953 182.8299 204.3282 202.7655 213.0254 20
##  [7,] 215.1152 197.1575 197.4273 192.1554 191.8861 206.7929 203.3585 20
##  [8,] 199.0534 173.4354 182.3684 191.4909 214.4410 200.8983 210.3851 20
##  [9,] 220.1842 175.5953 204.6010 175.8579 195.6855 170.0691 209.2073 19
## [10,] 199.3729 213.2011 193.6001 200.3612 206.5565 202.8488 207.2088 18
```

```r
sum(m)
```

```
## [1] 20032.51
```

```r
mean(m)
```

```
## [1] 200.3251
```

```r
rowSums(m)
```

```
##  [1] 2048.470 1993.774 2041.155 2025.924 1978.173 2007.265 1998.086 1960
```

```r
colSums(m)
```

```
##  [1] 2054.730 1983.654 1982.192 1963.610 1997.978 2001.839 2053.908 1978
```

```r
# standardize the matrix
z <- (m - mean(m)) / sd(m)
```

```r
z
```

```
##               [,1]       [,2]       [,3]        [,4]       [,5]         [,
##  [1,]  1.28528802  1.2218239 -0.3256841  0.40613865  0.1665940  0.277916
##  [2,] -0.57349498  2.1646089 -1.7417882  0.64562157 -0.3779416 -0.783932
##  [3,]  0.31748345 -1.3649263 -0.1963133  0.96277141  0.6968297  1.481924
##  [4,]  0.57650528 -0.2989403  1.1352110 -0.61596668 -0.7290676  0.586143
##  [5,]  0.35698951 -0.1592501  1.7887033  0.45367758 -1.3451640  0.054972
##  [6,] -0.13313334  0.5794704 -0.4445968 -1.68004206  0.3844054  0.234344
##  [7,]  1.42026916 -0.3041875 -0.2782756 -0.78452812 -0.8103926  0.621087
##  [8,] -0.12212321 -2.5821792 -1.7243635 -0.84833774  1.3555260  0.055041
##  [9,]  1.90703954 -2.3747685  0.4106013 -2.34955213 -0.4455350 -2.905444
## [10,] -0.09144695  1.2364622 -0.6458013  0.00346451  0.5983857  0.242345
##               [,9]       [,10]
##  [1,]  1.42140711  1.30560568
##  [2,]  0.21645471 -0.48848642
##  [3,]  0.05370436  0.59329679
##  [4,] -0.14731870  1.30463971
##  [5,] -1.17812024 -1.09789797
##  [6,]  0.55646825 -0.85783015
##  [7,] -0.23973975 -1.11801576
##  [8,] -0.20672212 -1.43248556
##  [9,]  0.86505545  0.04558258
## [10,]  0.75791330  0.59603916
```

Stripping attributes

- Many operations will strip attributes (generally why it's not a good idea to store important things in them)

```r
v
```

```
##              index value
## the first        1     4
## second           2     5
## III              3     6
## attr(,"matrix_mean")
## [1] 3.5
```

```r
rowSums(v)
```

```
## the first     second        III
##         5          7          9
```

```r
attributes(rowSums(v))
```

```
## $names
## [1] "the first" "second"    "III"
```

- Generally `names` are maintained
- Sometimes, `dim` is maintained, sometimes not
- All else is stripped

More on `names`

- The `names` attribute corresponds to the individual elements within a vector

```
names(v)
```

```
## NULL
```

```
names(v) <- letters[1:6]
v
```

```
##            index value
## the first      1     4
## second         2     5
## III            3     6
## attr(,"matrix_mean")
## [1] 3.5
## attr(,"names")
## [1] "a" "b" "c" "d" "e" "f"
```

- Perhaps more straightforward

```r
v3a <- c(a = 5, b = 7, c = 12)
v3a
```

```
##  a  b  c
##  5  7 12
```

```r
names(v3a)
```

```
## [1] "a" "b" "c"
```

```r
attributes(v3a)
```

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

Alternatives

```
v3b <- c(5, 7, 12)
names(v3b) <- c("a", "b", "c")
v3b
```

```
##  a  b  c
##  5  7 12
```

```
v3c <- setNames(c(5, 7, 12), c("a", "b", "c"))
v3c
```

```
##  a  b  c
##  5  7 12
```

- Note that `names` is **not** the same thing as `colnames`, but, somewhat confusingly, both work to rename the variables (columns) of a data frame. We'll talk more about why this is momentarily.

Why names might be helpful

```r
v
```

```
##              index value
## the first        1     4
## second           2     5
## III              3     6
## attr(,"matrix_mean")
## [1] 3.5
## attr(,"names")
## [1] "a" "b" "c" "d" "e" "f"
```

```r
v["b"]
```

```
## b
## 2
```

```r
v["e"]
```

```
## e
## 5
```

Implementation of factors

Quickly

```r
fct <- factor(c("a", "a", "b", "c"))
typeof(fct)
```

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

```r
attributes(fct)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

```r
str(fct)
```

```
##  Factor w/ 3 levels "a","b","c": 1 1 2 3
```

Implementation of dates

Quickly

```r
date <- Sys.Date()
typeof(date)
```

```
## [1] "double"
```

```r
attributes(date)
```

```
## $class
## [1] "Date"
```

```r
attributes(date) <- NULL
date
```

```
## [1] 18708
```

- This number represents the days passed since January 1, 1970, known as the Unix epoch.

Missing values

- Missing values breed missing values

```
NA > 5
```

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

```
NA * 7
```

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

- What about this one?

```
NA == NA
```

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

It is correct because there's no reason to presume that one missing value is or is not equal to another missing value.

When missing values don't propagate

```r
NA | TRUE
```

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

```r
x <- c(NA, 3, NA, 5)
any(x > 4)
```

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

How to test missingness?

- We've already seen the following doesn't work

```
x == NA
```

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

- Instead, use `is.na`

```
is.na(x)
```

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

- When does this regularly come into play?

Lists

Lists

- Lists are vectors, but not *atomic* vectors

- Fundamental difference – each element can be a different type

```r
list("a", 7L, 3.25, TRUE)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 7
##
## [[3]]
## [1] 3.25
##
## [[4]]
## [1] TRUE
```

Lists

- Technically, each element of the list is a vector, possibly atomic
- The prior example included all *scalars*, which are vectors of length 1.
- Lists do not require all elements to be the same length

```r
l <- list(
  c("a", "b", "c"),
  rnorm(5),
  c(7L, 2L),
  c(TRUE, TRUE, FALSE, TRUE)
)
l
```

```
## [[1]]
## [1] "a" "b" "c"
##
## [[2]]
## [1]  1.2009654  1.0447511 -1.0032086
##
## [[3]]
## [1] 7 2
##
## [[4]]
## [1]  TRUE  TRUE FALSE  TRUE
```

Check the list

```r
typeof(l)
```

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

```r
attributes(l)
```

```
## NULL
```

```r
str(l)
```

```
## List of 4
##  $ : chr [1:3] "a" "b" "c"
##  $ : num [1:5] 1.201 1.045 -1.003 1.848 -0.667
##  $ : int [1:2] 7 2
##  $ : logi [1:4] TRUE TRUE FALSE TRUE
```

Data frames as lists

- A data frame is just a special case of a list, where all the elements are of the same length.

```r
l_df <- list(
  a = c("red", "blue"),
  b = rnorm(2),
  c = c(7L, 2L),
  d = c(TRUE, FALSE)
)
l_df
```

```
## $a
## [1] "red"  "blue"
##
## $b
## [1]  0.1055138 -0.4222559
##
## $c
## [1] 7 2
##
## $d
## [1]  TRUE FALSE
```

```r
data.frame(l_df)
```

```
##      a          b c     d
## 1  red  0.1055138 7  TRUE
## 2 blue -0.4222559 2 FALSE
```

Subsetting Lists

A nested list

Lists are often complicated objects. Let's create a somewhat complicated one

```
x <- c(a = 3, b = 5, c = 7)
l <- list(
  x = x,
  x2 = c(x, x),
  x3 = list(
    vect = x,
    squared = x^2,
    cubed = x^3)
)
```

Subsetting lists

Multiple methods

- Most common: `$`, `[`, and `[[`

```
l[1]
```

```
## $x
## a b c
## 3 5 7
```

```
typeof(l[1])
```

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

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

```
## a b c
## 3 5 7
```

```
typeof(l[[1]])
```

```
## [1] "double"
```

```
l[[1]]["c"]
```

```
## c
## 7
```

Named list

- Because the elements of the list are named, we can use `$`

```r
l$x2
```

```
## a b c a b c
## 3 5 7 3 5 7
```

```r
l$x3
```

```
## $vect
## a b c
## 3 5 7
##
## $squared
##  a  b  c
##  9 25 49
##
## $cubed
##   a   b   c
##  27 125 343
```

Subsetting nested lists

- Multiple `$` if all named

```
l$x3$squared
```

```
##  a  b  c
##  9 25 49
```

- Note this doesn't work on named elements of an atomic vector, just the named elements of a list

```
l$x3$squared$b
```

```
## Error in l$x3$squared$b: $ operator is invalid for atomic vectors
```

But we could do something like...

```r
l$x3$squared["b"]
```

```
##  b
## 25
```

Alternatives

- You can always use logical
- Indexing works too

```
l[c(TRUE, FALSE, TRUE)]
```

```
## $x
## a b c
## 3 5 7
##
## $x3
## $x3$vect
## a b c
## 3 5 7
##
## $x3$squared
##  a  b  c
##  9 25 49
##
## $x3$cubed
##   a   b   c
##  27 125 343
```

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

```
## $x
## a b c
## 3 5 7
##
## $x3
## $x3$vect
## a b c
## 3 5 7
##
## $x3$squared
##  a  b  c
##  9 25 49
##
## $x3$cubed
##   a   b   c
##  27 125 343
```

Careful with your brackets

```r
l[[c(TRUE, FALSE, FALSE)]]
```

```
## Error in l[[c(TRUE, FALSE, FALSE)]]: recursive indexing failed at level
```

- Why doesn't the above work?

Subsetting in multiple dimensions

- Generally we deal with 2d data frames
- If there are two dimensions, we separate the `[` subsetting with a comma

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
mtcars[3, 4]
```

```
## [1] 93
```

Empty indicators

- An empty indicator implies "all"

Select the entire fourth column

```
mtcars[ ,4]
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
## [27]  91 113 264 175 335 109
```

Select the entire 4th row

```
mtcars[4, ]
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

Data types returned

- By default, each of the prior will return a vector, which itself can be subset

The following are equivalent

```
mtcars[4, c("mpg", "hp")]
```

```
##                 mpg  hp
## Hornet 4 Drive 21.4 110
```

```
mtcars[4, ][c("mpg", "hp")]
```

```
##                 mpg  hp
## Hornet 4 Drive 21.4 110
```

Return a data frame

- Often, you don't want the vector returned, but rather the modified data frame.
- Specify `drop = FALSE`

```
mtcars[ ,4]
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
## [27]  91 113 264 175 335 109
```

```
mtcars[ ,4, drop = FALSE]
```

```
##                      hp
## Mazda RX4           110
## Mazda RX4 Wag       110
## Datsun 710           93
## Hornet 4 Drive      110
## Hornet Sportabout   175
## Valiant             105
## Duster 360          245
## Merc 240D            62
## Merc 230             95
```

tibbles

- Note dropping the data frame attribute is the default for a `data.frame` but NOT a `tibble`.

```
mtcars_tbl <- tibble::as_tibble(mtcars)
mtcars_tbl[ ,4]
```

```
## # A tibble: 32 x 1
##        hp
##     <dbl>
##  1    110
##  2    110
##  3     93
##  4    110
##  5    175
##  6    105
##  7    245
##  8     62
##  9     95
## 10    123
## # … with 22 more rows
```

You can override this

```
mtcars_tbl[ ,4, drop = TRUE]
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
## [27]  91 113 264 175 335 109
```

More than two dimensions

- Depending on your applications, you may not run into this much

```r
array <- 1:12
dim(array) <- c(2, 3, 2)
array
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

Subset array

Select just the second matrix

```
array[ , ,2]
```

```
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

Select first column of each matrix

```
array[ ,1, ]
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
```

Back to lists

Why are they so useful?

- Fairly obviously, they're much more flexible
- Often returned by functions, for example, `lm`

```
m <- lm(mpg ~ hp, mtcars)
str(m)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 30.0989 -0.0682
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "hp"
##  $ residuals    : Named num [1:32] -1.594 -1.594 -0.954 -1.194 0.541 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 7
##  $ effects      : Named num [1:32] -113.65 -26.046 -0.556 -0.852 0.67 ..
##   ..- attr(*, "names")= chr [1:32] "(Intercept)" "hp" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:32] 22.6 22.6 23.8 22.6 18.2 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 7
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ...
```

Summary

- Atomic vectors must all be the same type
  - implicit coercion occurs if not (and you haven't specified the coercion explicitly)
- Lists are also vectors, but not atomic vectors
  - Each element can be of a different type and length
  - Incredibly flexible, but often a little more difficult to get the hang of, particularly with subsetting

Next time

Loops with base R