Data Classes

Introduction to R for Public Health Researchers

Data Classes:

- One dimensional classes (‘vectors’):
  - Character: strings or individual characters, quoted
  - Numeric: any real number(s)
  - Integer: any integer(s)/whole numbers
  - Factor: categorical/qualitative variables
  - Logical: variables composed of TRUE or FALSE
  - Date/POSIXct: represents calendar dates and times

Character and numeric

We have already covered `character` and `numeric` classes.

```
class(c("Andrew", "Jaffe"))
```

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

```
class(c(1, 4, 7))
```

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

Integer

Integer is a special subset of numeric that contains only whole numbers

A sequence of numbers is an example of the integer class

```
x = seq(from = 1, to = 5) # seq() is a function
x
```

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

```
class(x)
```

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

Integer

The colon `:` is a shortcut for making sequences of numbers

It makes consecutive integer sequence from `[num1]` to `[num2]` by 1

```
1:5
```

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

Logical

logical is a class that only has two possible elements: TRUE and FALSE

```r
x = c(TRUE, FALSE, TRUE, TRUE, FALSE)
class(x)
```

```
## [1] "logical"
```

```r
is.numeric(c("Andrew", "Jaffe"))
```

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

```r
is.character(c("Andrew", "Jaffe"))
```

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

Logical

Note that `logical` elements are NOT in quotes.

```
z = c("TRUE", "FALSE", "TRUE", "FALSE")
class(z)
```

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

```
as.logical(z)
```

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

Bonus: `sum()` and `mean()` work on `logical` vectors - they return the total and proportion of `TRUE` elements, respectively.

```
sum(as.logical(z))
```

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

General Class Information

There are two useful functions associated with practically all R classes, which relate to logically checking the underlying class (`is.CLASS_()`) and coercing between classes (`as.CLASS_()`).

```
is.numeric(c("Andrew", "Jaffe"))
```

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

```
is.character(c("Andrew", "Jaffe"))
```

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

General Class Information

There are two useful functions associated with practically all R classes, which relate to logically checking the underlying class (`is.CLASS_()`) and coercing between classes (`as.CLASS_()`).

```r
as.character(c(1, 4, 7))
```

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

```r
as.numeric(c("Andrew", "Jaffe"))
```

```
## Warning: NAs introduced by coercion
```

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

Factors

A `factor` is a special `character` vector where the elements have pre-defined groups or ‘levels’. You can think of these as qualitative or categorical variables:

```
x = factor(c("boy", "girl", "girl", "boy", "girl"))
x
```

```
## [1] boy  girl girl boy  girl
## Levels: boy girl
```

```
class(x)
```

```
## [1] "factor"
```

Note that levels are, by default, in alphanumerical order.

Factors

Factors are used to represent categorical data, and can also be used for ordinal data (ie categories have an intrinsic ordering)

Note that R reads in character strings as factors by default in functions like `read.csv()` (but not `read_csv`)

‘The function factor is used to encode a vector as a factor (the terms 'category' and ‘enumerated type' are also used for factors). If argument ordered is TRUE, the factor levels are assumed to be ordered.'

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x))
```

Factors

Suppose we have a vector of case-control status

```
cc = factor(c("case","case","case",
        "control","control","control"))
cc
```

```
## [1] case    case    case    control control control
## Levels: case control
```

We can reset the levels using the `levels` function, but this is **bad** and can cause problems. You should do this using the `levels` argument in the `factor()`

```
levels(cc) = c("control","case")
cc
```

```
## [1] control control control case    case    case
## Levels: control case
```

Factors

Note that the levels are alphabetically ordered by default. We can also specify the levels within the factor call

```
casecontrol = c("case","case","case","control",
          "control","control")
factor(casecontrol, levels = c("control","case") )
```

```
## [1] case    case    case    control control control
## Levels: control case
```

```
factor(casecontrol, levels = c("control","case"),
       ordered=TRUE)
```

```
## [1] case    case    case    control control control
## Levels: control < case
```

Factors

Factors can be converted to `numeric` or `character` very easily

```
x = factor(casecontrol,
        levels = c("control","case") )
as.character(x)
```

```
## [1] "case"    "case"    "case"    "control" "control" "c
```

```
as.numeric(x)
```

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

Factors

However, you need to be careful modifying the labels of existing factors, as its quite easy to alter the meaning of the underlying data.

```
xCopy = x
levels(xCopy) = c("case", "control") # wrong way
xCopy
```

```
## [1] control control control case    case    case
## Levels: case control
```

```
as.character(xCopy) # labels switched
```

```
## [1] "control" "control" "control" "case"    "case"    "c
```

```
as.numeric(xCopy)
```

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

Creating categorical variables

The `rep()` ["repeat"] function is useful for creating new variables

```r
bg = rep(c("boy","girl"),each=50)
head(bg)
```

```
## [1] "boy" "boy" "boy" "boy" "boy" "boy"
```

```r
bg2 = rep(c("boy","girl"),times=50)
head(bg2)
```

```
## [1] "boy"  "girl" "boy"  "girl" "boy"  "girl"
```

```r
length(bg) == length(bg2)
```

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

Creating categorical variables

One frequently-used tool is creating categorical variables out of continuous variables, like generating quantiles of a specific continuously measured variable.

A general function for creating new variables based on existing variables is the `ifelse()` function, which “returns a value with the same shape as test which is filled with elements selected from either yes or no depending on whether the element of test is `TRUE` or `FALSE`.”

```
ifelse(test, yes, no)

# test: an object which can be coerced
    to logical mode.
# yes: return values for true elements of test.
# no: return values for false elements of test.
```

Charm City Circulator data

Please download the Charm City Circulator data:

http://www.aejaffe.com/winterR_2017/data/Charm_City_Circulator_Ridership.csv

```r
# paste/paste0 combines strings/character
circ = read_csv(
  paste0("http://www.aejaffe.com/winterR_2017/data",
         "/Charm_City_Circulator_Ridership.csv"))
```

```
## Parsed with column specification:
## cols(
##   day = col_character(),
##   date = col_character(),
##   orangeBoardings = col_integer(),
##   orangeAlightings = col_integer(),
##   orangeAverage = col_double(),
##   purpleBoardings = col_integer(),
##   purpleAlightings = col_integer(),
```

Creating categorical variables

For example, we can create a new variable that records whether daily ridership on the Circulator was above 10,000.

```
hi_rider = ifelse(circ$daily > 10000, "high", "low")
hi_rider = factor(hi_rider, levels = c("low","high"))
head(hi_rider)
```

```
## [1] low low low low low low
## Levels: low high
```

```
table(hi_rider)
```

```
## hi_rider
##  low high
##  740  282
```

Creating categorical variables

You can also nest `ifelse()` within itself to create 3 levels of a variable.

```r
riderLevels = ifelse(circ$daily < 10000, "low",
                  ifelse(circ$daily > 20000,
                  "high", "med"))
riderLevels = factor(riderLevels,
              levels = c("low","med","high"))
head(riderLevels)
```

```
## [1] low low low low low low
## Levels: low med high
```

```r
table(riderLevels)
```

```
## riderLevels
##  low  med high
##  740  280    2
```

Creating categorical variables

However, it's much easier to use `cut()` to create categorical variables from continuous variables.

'cut divides the range of x into intervals and codes the values in x according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.'

```
cut(x, breaks, labels = NULL, include.lowest = FALSE,
  right = TRUE, dig.lab = 3,
  ordered_result = FALSE, ...)
```

Creating categorical variables

`x`: a numeric vector which is to be converted to a factor by cutting.

`breaks`: either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which `x` is to be cut.

`labels`: labels for the levels of the resulting category. By default, labels are constructed using "(a,b]" interval notation. If `labels = FALSE`, simple integer codes are returned instead of a factor.

Creating categorical variables

```
riderLevels2 = cut(
  circ$daily,
  breaks = c(min(circ$daily, na.rm = TRUE),
             10000,
             20000,
             max(circ$daily, na.rm = TRUE)),
  labels = c("low", "med", "high"), # one less than breaks
  include.lowest = TRUE)
head(riderLevels2)
```

```
## [1] low low low low low low
## Levels: low med high
```

```
table(riderLevels2, riderLevels)
```

```
##              riderLevels
## riderLevels2 low med high
##         low  740    0    0
```

Cut

Now that we know more about factors, cut() will make more sense:

```r
x = 1:100
cx = cut(x, breaks = c(0,10,25,50,100))
head(cx)
```

```
## [1] (0,10] (0,10] (0,10] (0,10] (0,10] (0,10]
## Levels: (0,10] (10,25] (25,50] (50,100]
```

```r
table(cx)
```

```
## cx
##   (0,10]  (10,25]  (25,50] (50,100]
##       10       15       25       50
```

Cut

We can also leave off the labels

```r
cx = cut(x, breaks = c(0,10,25,50,100), labels = FALSE)
head(cx)
```

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

```r
table(cx)
```

```
## cx
##  1  2  3  4
## 10 15 25 50
```

Cut

Note that you have to specify the endpoints of the data, otherwise some of the categories will not be created

```
cx = cut(x, breaks = c(10,25,50), labels = FALSE)
head(cx)
```

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

```
table(cx)
```

```
## cx
##  1  2
## 15 25
```

```
table(cx, useNA = "ifany")
```

```
## cx
##    1    2 <NA>
##   15   25   60
```

Date

You can convert date-like strings in the Date class (http://www.statmethods.net/input/dates.html for more info)

```
head(sort(circ$date))
```

```
## [1] "01/01/2011" "01/01/2012" "01/01/2013" "01/02/2011"
## [6] "01/02/2013"
```

```
# creating a date for sorting
circ$newDate <- as.Date(circ$date, "%m/%d/%Y")
head(circ$newDate)
```

```
## [1] "2010-01-11" "2010-01-12" "2010-01-13" "2010-01-14"
## [6] "2010-01-16"
```

```
range(circ$newDate)
```

```
## [1] "2010-01-11" "2013-03-01"
```

Date

However, the `lubridate` package is much easier for generating explicit dates:

```
library(lubridate) # great for dates!
circ = mutate(circ, newDate2 = mdy(date))
head(circ$newDate2)
```

```
## [1] "2010-01-11" "2010-01-12" "2010-01-13" "2010-01-14"
## [6] "2010-01-16"
```

```
range(circ$newDate2) # gives you the range of the data
```

```
## [1] "2010-01-11" "2013-03-01"
```

POSIXct

The `POSIXct` class is like a more general date format (with hours, minutes, seconds).

```
theTime = Sys.time()
theTime
```

```
## [1] "2017-01-09 23:10:52 EST"
```

```
class(theTime)
```

```
## [1] "POSIXct" "POSIXt"
```

```
theTime + as.period(20, unit = "minutes") # the future
```

```
## [1] "2017-01-09 23:30:52 EST"
```

Date

However, the `lubridate` package is much easier for generating explicit dates:

```
circ = circ %>%
  group_by(day) %>%
  mutate(first_date = first(newDate2),
         diff_from_first = difftime( # time1 - time2
           time1 = newDate2, time2 = first_date))
head(circ$diff_from_first, 10)
```

```
## Time differences in secs
##  [1]      0      0      0      0      0      0      0 60
```

```
units(circ$diff_from_first) = "days"
head(circ$diff_from_first, 10)
```

```
## Time differences in days
##  [1] 0 0 0 0 0 0 0 7 7 7
```

Data Classes:

- Two dimensional classes:
  - `data.frame`: traditional ‘Excel’ spreadsheets
    - Each column can have a different class, from above
  - Matrix: two-dimensional data, composed of rows and columns. Unlike data frames, the entire matrix is composed of one R class, e.g. all numeric or all characters.

Matrices

```
n = 1:9
n
```

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

```
mat = matrix(n, nrow = 3)
mat
```

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

Matrix (and Data frame) Functions

These are in addition to the previous useful vector functions:

- `nrow()` displays the number of rows of a matrix or data frame
- `ncol()` displays the number of columns
- `dim()` displays a vector of length 2: # rows, # columns
- `colnames()` displays the column names (if any) and `rownames()` displays the row names (if any)

Data Selection

Matrices have two “slots” you can use to select data, which represent rows and columns, that are separated by a comma, so the syntax is `matrix[row,column]`. Note you cannot use `dplyr` functions on matrices.

```
mat[1, 1] # individual entry: row 1, column 1
```

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

```
mat[1, ] # first row
```

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

```
mat[, 1] # first columns
```

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

Data Selection

Note that the class of the returned object is no longer a matrix

```
class(mat[1, ])
```

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

```
class(mat[, 1])
```

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

Data Frames

To review, the `data.frame/tbl_df` are the other two dimensional variable classes.

Again, data frames are like matrices, but each column is a vector that can have its own class. So some columns might be `character` and others might be `numeric`, while others maybe a `factor`.

Lists

- One other data type that is the most generic are `lists`.
- Can be created using list()
- Can hold vectors, strings, matrices, models, list of other list, lists upon lists!
- Can reference data using $ (if the elements are named), or using [], or [[]]

```
> mylist <- list(letters=c("A", "b", "c"),
+          numbers=1:3, matrix(1:25, ncol=5))
```

List Structure

```
> head(mylist)
```

```
$letters
[1] "A" "b" "c"

$numbers
[1] 1 2 3

[[3]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

List referencing

```
> mylist[1] # returns a list
```

```
$letters
[1] "A" "b" "c"
```

```
> mylist["letters"] # returns a list
```

```
$letters
[1] "A" "b" "c"
```

List referencing

```
> mylist[[1]] # returns the vector 'letters'
```

```
[1] "A" "b" "c"
```

```
> mylist$letters # returns vector
```

```
[1] "A" "b" "c"
```

```
> mylist[["letters"]] # returns the vector 'letters'
```

```
[1] "A" "b" "c"
```

List referencing

You can also select multiple lists with the single brackets.

```
> mylist[1:2] # returns a list
```

```
$letters
[1] "A" "b" "c"

$numbers
[1] 1 2 3
```

List referencing

You can also select down several levels of a list at once

```
> mylist$letters[1]
```

```
[1] "A"
```

```
> mylist[[2]][1]
```

```
[1] 1
```

```
> mylist[[3]][1:2,1:2]
```

```
     [,1] [,2]
[1,]    1    6
[2,]    2    7
```