Data Classes

Introduction to R for Public Health Researchers

Data Types:

- One dimensional types (‘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` types.

```
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 type

```
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 type that only has two possible elements: `TRUE` and `FALSE`

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

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

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

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

```
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_()`).

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

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

```
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))
```

Necessary for the lab: `%in%`

```
x = c(0, 2, 2, 3, 4)
(x == 0 | x == 2)
```

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

Introduce the `%in%` operator:

```
x %in% c(0, 2) # NEVER has NA and returns logical
```

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

reads “return `TRUE` if `x` is in 0 or 2”. (Like `inlist` in Stata).

Lab Part 1

Website

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

Another way to do this once you already have the factor made is with the `relevel()` function.

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

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

Factors

One of the core “tidyverse” packages is `forcats` which offers useful functionality for interacting with factors. For example, there is a function for releveling factors here:

```
fct_relevel(cc, "control")
```

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

Factors

There are other useful functions for dictating the levels of factors, like in the order they appears in the vector, by frequency, or into collapsed groups.

```
levels(fct_inorder(chickwts$feed))
```

```
## [1] "horsebean" "linseed"   "soybean"   "sunflower" "meatmeal"  "casein"
```

```
levels(fct_infreq(chickwts$feed))
```

```
## [1] "soybean"   "casein"    "linseed"   "sunflower" "meatmeal"  "horsebean"
```

```
levels(fct_lump(chickwts$feed, n=1))
```

```
## [1] "soybean" "Other"
```

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" "control"
```

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

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

Creating categorical variables

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

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

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

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

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

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

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

Lab Part 2

Website

Dates

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

```
circ = jhur::read_circulator()
head(sort(circ$date))
```

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

```
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" "2010-01-15"
## [6] "2010-01-16"
```

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

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

Works great - but need to specy the correct format still

See `?ymd` and `?ymd_hms`

```
x = c("2014-02-4 05:02:00", "2016/09/24 14:02:00")
ymd_hms(x)
```

```
## [1] "2014-02-04 05:02:00 UTC" "2016-09-24 14:02:00 UTC"
```

```
ymd_hm(x)
```

```
## Warning: All formats failed to parse. No formats found.
```

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

POSIXct

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

```
x = c("2014-02-4 05:02:00", "2016/09/24 14:02:00")
dates = ymd_hms(x)
class(dates)
```

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

Adding Periods of time

The `as.Period` command is helpful for adding time to a date:

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

```
## [1] "2020-01-05 20:25:23 EST"
```

```
class(theTime)
```

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

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

```
## [1] "2020-01-05 20:45:23 EST"
```

Differences in Times

You can subtract times as well, the `difftime` function is helpful as you can set the units (note it does `time1 - time2`):

```r
the_future = ymd_hms("2020-12-31 11:59:59")
the_future - theTime
```

```
## Time difference of 360.4407 days
```

```r
difftime(the_future, theTime, units = "weeks")
```

```
## Time difference of 51.49153 weeks
```

Lab Part 3

Website

Website

Website

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
```

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
```

Quick Aside: “slicing” data: like _n and _N in Stata

In `dplyr`, there are `first`, `last` and `nth` operators.

If you first sort a data set using `arrange`, you can grab the first or last as so:

```
circ %>%
  mutate(first_date = first(newDate2),
         last_date = last(newDate2),
         third_date = nth(newDate2, 3)) %>%
  select(day, date, first_date, last_date, third_date) %>% head(3)
```

```
## # A tibble: 3 x 5
##   day       date       first_date last_date  third_date
##   <chr>     <chr>      <date>     <date>     <date>    
## 1 Monday    01/11/2010 2010-01-11 2013-03-01 2010-01-13
## 2 Tuesday   01/12/2010 2010-01-11 2013-03-01 2010-01-13
## 3 Wednesday 01/13/2010 2010-01-11 2013-03-01 2010-01-13
```

Quick Aside: “slicing” data

Many times, you need to group first

```
circc %>%
  group_by(day) %>%
  mutate(first_date = first(newDate2),
         last_date = last(newDate2),
         third_date = nth(newDate2, 3)) %>%
  select(day, date, first_date, last_date, third_date) %>% head(3)
```

```
## # A tibble: 3 x 5
## # Groups:   day [3]
##   day       date       first_date last_date  third_date
##   <chr>     <chr>      <date>     <date>     <date>    
## 1 Monday    01/11/2010 2010-01-11 2013-02-25 2010-01-25
## 2 Tuesday   01/12/2010 2010-01-12 2013-02-26 2010-01-26
## 3 Wednesday 01/13/2010 2010-01-13 2013-02-27 2010-01-27
```

Differences in Times

```
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 604800 604800 604800
```

```
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
```

Website

Website