Day Two: Data Cleaning

Dillon Niederhut
Shinhye Choi

04 February, 2016

Review

Inspecting objects

we'll start by using some data that is already in R

```r
data(state)
str(state.x77)
```

Inspecting variables

We should see 50 levels in this division variable

```r
state.division
```

```
##  [1] East South Central Pacific            Mountain
##  [4] West South Central Pacific            Mountain
##  [7] New England        South Atlantic     South Atlantic
## [10] South Atlantic     Pacific            Mountain
## [13] East North Central East North Central West North Central
## [16] West North Central East South Central West South Central
## [19] New England        South Atlantic     New England
## [22] East North Central West North Central East South Central
## [25] West North Central Mountain           West North Central
## [28] Mountain           New England        Middle Atlantic
## [31] Mountain           Middle Atlantic    South Atlantic
## [34] West North Central East North Central West South Central
## [37] Pacific            Middle Atlantic    New England
## [40] South Atlantic     West North Central East South Central
## [43] West South Central Mountain           New England
## [46] South Atlantic     Pacific            South Atlantic
## [49] East North Central Mountain
## 9 Levels: New England Middle Atlantic ... Pacific
```

```r
length(state.division)
```

```
## [1] 50
```

```r
levels(state.division)
```

```
## [1] "New England"        "Middle Atlantic"    "South Atlantic"
## [4] "East South Central" "West South Central" "East North Central"
## [7] "West North Central" "Mountain"           "Pacific"
```

Inspecting data frames

recall, a dataframe is a list of vectors, where each vector is one variable with all of its measurements

R expects dataframes to be rectangular

```
state <- state.x77
rm(state.x77)
```

```
## Warning in rm(state.x77): object 'state.x77' not found
```

```
state <- as.data.frame(state)
head(state)
```

```
##            Population Income Illiteracy Life Exp Murder HS Grad Frost
## Alabama          3615   3624        2.1    69.05   15.1    41.3    20
## Alaska            365   6315        1.5    69.31   11.3    66.7   152
## Arizona          2212   4530        1.8    70.55    7.8    58.1    15
## Arkansas         2110   3378        1.9    70.66   10.1    39.9    65
## California      21198   5114        1.1    71.71   10.3    62.6    20
## Colorado         2541   4884        0.7    72.06    6.8    63.9   166
##              Area
## Alabama     50708
## Alaska     566432
## Arizona    113417
## Arkansas    51945
## California 156361
## Colorado   103766
```

Introduction

Today's class will be essentially be split into two components: CRUD operations in R and TIDY data. For more on tidiness in data, see Hadley Wickham's paper. We will also touch on missingness - for an accessible introduction, you can read this very old and no longer state-of-the-art paper.

yesterday we saw how to create dataframes in R

```
my.data <- data.frame(n = c(1, 2, 3),
                      c=c('one', 'two', 'three'),
                      b=c(TRUE, TRUE, FALSE),
                      d=c(as.Date("2015-07-27"),
                          as.Date("2015-07-27")+7,
                          as.Date("2015-07-27")-7),
                      really.long.and.complicated.variable.name=999)
```

remember, you can learn about dataframes with

```
str(my.data)
```

```
## 'data.frame':    3 obs. of  5 variables:
##  $ n                                        : num  1 2 3
##  $ c                                        : Factor w/ 3 levels "one","three",..: 1 3 2
```

```
##  $ b                                      : logi  TRUE TRUE FALSE
##  $ d                                      : Date, format: "2015-07-27" "2015-08-03" ...
##  $ really.long.and.complicated.variable.name: num  999 999 999
```

in practice, you will only rarely create dataframes by hand, because creating tables in a text editor is both boring and prone to error

Readibility

we've broken up the previous command across multiple lines to make it easier to read this is a stylistic choice, and one that should be encouraged: however, it won't be obvious to most of the students that it is necessary to either highlight the whole command and run, or hit run for every line, starting from the first one, in order

often, students will just run the second line, and be confused when nothing runs correctly in the console anymore - the way to get out of this is by hitting `ESC`

Reading dataframes from file

why read data from text files?

they are human-readable and highly interoperable

```r
read.table("data/mydata.csv", sep=',', header = TRUE)
```

```
##   n     c     b          d really.long.and.complicated.variable.name
## 1 1   one  TRUE 2015-07-27                                        999
## 2 2   two  TRUE 2015-08-03                                        999
## 3 3 three FALSE 2015-07-20                                        999
```

> side note - anyone who is 100% new to computing will have a hard time understanding the concept of a working directory, and will try to run this code from their home directory (spoiler alert - it doesn't work)

R has convenience wrappers for reading in tables

```r
read.csv("data/mydata.csv")
```

```
##   n     c     b          d really.long.and.complicated.variable.name
## 1 1   one  TRUE 2015-07-27                                        999
## 2 2   two  TRUE 2015-08-03                                        999
## 3 3 three FALSE 2015-07-20                                        999
```

note that we are only reading the files by doing this

R lets you read in part of a table

you'll sometimes find that you want to work with a smaller part of a dataset - maybe because the data is too large to fit into memory, or maybe because you want to test out some code on a small piece of the data so it runs faster

```
read.csv("data/mydata.csv", nrows=2)
```

```
##   n   c    b          d really.long.and.complicated.variable.name
## 1 1 one TRUE 2015-07-27                                       999
## 2 2 two TRUE 2015-08-03                                       999
```

note that `nrows` is **not** equal to the number of lines in the file, because it does not include the file header

R also has its own kind of data file

```
load("data/mydata.Rda")
```

the `load` function does actually put the file into memory, and with the name you originally gave it when you saved it

this is typically a bad thing, and there is currently no easy workaround

to read in tables from excel, use the `xlsx` package

if you are exporting data from excel, be sure to export datetimes as strings, as excel does not store dates internally the same way Unix does

```
# WARNING! xlsx package install crashed current version of RStudio
install.packages("xlsx")
library(xlsx)
read.xlsx("data/cpds_excel_new.xlsx")
```

But it may be better to save your .xlsx file as a csv. format in Excel first, and then read the csv file into R.

you can also use R to read in data from proprietary software

```
# examples of these?
install.packages("foreign")
library(foreign)
read.dta("data/cpds_stata.dta")
read.spss()
read.octave()
```

Cleaning data

there are two major steps to data cleaning, which we will call 'sanitizing' and 'tidying'

in sanitizing, our goal is to take each variable and force its values to be honest representations of its levels

in tidying, we are arranging our data structurally such that each row contains exactly one observation, and each column contains exactly one kind of data about that observation (this is sometimes expressed in SQL terms as "An attribute must tell something about the key, the whole key, and nothing but the key, so help me Codd")

exporting data from other software can do weird things to numbers and factors

```r
dirty <- read.csv('data/dirty.csv')
str(dirty)
```

```
## 'data.frame':    5 obs. of  5 variables:
##  $ Timestamp                    : Factor w/ 5 levels "7/25/2015 10:08:41",..: 1 2 3 4 5
##  $ How.tall.are.you.            : Factor w/ 5 levels "156","2.1","5'9",..: 5 4 3 2 1
##  $ What.department.are.you.in.: Factor w/ 5 levels "  geology","999",..: 4 2 1 5 3
##  $ Are.you.currently.enrolled.: Factor w/ 3 levels "999","No","Yes": 3 3 1 2 1
##  $ What.is.your.birth.order.  : Factor w/ 3 levels "1","2","9,000": 1 1 2 3 2
```

it's usually better to DISABLE R's intuition about data types

unless you already know the data is clean and has no non-factor strings in it (i.e. you are the one who created it)

```r
dirty <- read.csv('data/dirty.csv',stringsAsFactors = FALSE)
str(dirty)
```

```
## 'data.frame':    5 obs. of  5 variables:
##  $ Timestamp                    : chr  "7/25/2015 10:08:41" "7/25/2015 10:10:56" "7/25/2015 10:11:20"
##  $ How.tall.are.you.            : chr  "very" "70" "5'9" "2.1" ...
##  $ What.department.are.you.in.: chr  "Geology  " "999" "  geology" "goelogy" ...
##  $ Are.you.currently.enrolled.: chr  "Yes" "Yes" "999" "No" ...
##  $ What.is.your.birth.order.  : chr  "1" "1" "2" "9,000" ...
```

let's start by removing the empty rows and columns

```r
tail(dirty)
```

```
##            Timestamp How.tall.are.you. What.department.are.you.in.
## 1 7/25/2015 10:08:41              very                  Geology  
## 2 7/25/2015 10:10:56                70                         999
## 3 7/25/2015 10:11:20               5'9                     geology
## 4 7/25/2015 10:11:25               2.1                     goelogy
## 5 7/25/2015 10:11:29               156                      anthro
```

```
##   Are.you.currently.enrolled. What.is.your.birth.order.
## 1                         Yes                        1
## 2                         Yes                        1
## 3                         999                        2
## 4                          No                    9,000
## 5                         999                        2
```

```
dirty <- dirty[1:5,-6]
dim(dirty)
```

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

you can replace variable names

and you should, if they are uninformative or long

```
names(dirty)
```

```
## [1] "Timestamp"                   "How.tall.are.you."
## [3] "What.department.are.you.in." "Are.you.currently.enrolled."
## [5] "What.is.your.birth.order."
```

```
names(dirty) <- c("time", "height", "dept", "enroll", "birth.order")
```

it's common for hand-coded data to have a signifier for subject-missingness

(to help differentiate it from your hand-coder forgetting to do something)

```
dirty$enrollment
```

```
## NULL
```

you should replace all of these values in your dataframe with R's missingness signifier, `NA`

```
table(dirty$enroll)
```

```
##
## 999  No Yes
##   2   1   2
```

```
dirty$enroll[dirty$enroll=="999"] <- NA
table(dirty$enroll, useNA = "ifany")
```

```
##
##    No  Yes <NA>
##     1    2    2
```

> side note - read.table() has an option to specify field values as `NA` as soon as you import the data, but this is a BAAAAD idea because R automatically encodes blank fields as missing too, and thus you lose the ability to distinguish between user-missing and experimenter-missing

that timestamp variable is not in a format R likes

base R doesn’t handle time well, so we need to get rid of the time part of the timestamp

```
dirty$time
```

```
## [1] "7/25/2015 10:08:41" "7/25/2015 10:10:56" "7/25/2015 10:11:20"
## [4] "7/25/2015 10:11:25" "7/25/2015 10:11:29"
```

```
dirty$time <- sub(' [0-9]+:[0-9]+:[0-9]+','',dirty$time)
dirty$time
```

```
## [1] "7/25/2015" "7/25/2015" "7/25/2015" "7/25/2015" "7/25/2015"
```

the height variable is in four different units

we can fix this with a somewhat complicated loop (since R started as a functional language, there are not easy ways to conditionally modify structures in place)

OR

we can do the same task line-by-line, since the number of observations is small

```
class(dirty$height)
```

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

```
as.numeric(dirty$height)
```

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

```
## [1]    NA  70.0    NA   2.1 156.0
```

because there are apostrophes and quotation marks, R thinks these are strings

```
dirty$height[grep("’", dirty$height, perl=TRUE)] <- 5*30.48 + 9*2.54
dirty$height[2] <- 70*2.54
dirty$height[3] <- 2.1*100
```

let’s fix some of those department spellings

first, let’s make this all lowercase

```
dirty$dept
```

```
## [1] "Geology  " "999"       "  geology" "goelogy"   "anthro"
```

```r
dirty$dept <- tolower(dirty$dept)
dirty$dept <- gsub(' ', '', dirty$dept)  # what did we just do?
dirty$dept[4] <- "geology"
dirty[dirty == "999"] <- NA
```

then, you can coerce the data into the types they should be

```r
dirty$time <- as.Date(dirty$time,'%m/%d/%Y')
dirty$height <- as.numeric(dirty$height)
```

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

```r
dirty$dept <- as.factor(dirty$dept)
dirty$enroll <- as.factor(dirty$enroll)
dirty$birth.order <- as.numeric(dirty$birth.order)
```

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

```r
str(dirty)
```

```
## 'data.frame':    5 obs. of  5 variables:
##  $ time       : Date, format: "2015-07-25" "2015-07-25" ...
##  $ height     : num  NA 177.8 210 2.1 156
##  $ dept       : Factor w/ 2 levels "anthro","geology": 2 NA 2 2 1
##  $ enroll     : Factor w/ 2 levels "No","Yes": 2 2 NA 1 NA
##  $ birth.order: num  1 1 2 NA 2
```

Missingness

there are many reasons why you might have missing data

AS LONG AS MISSINGNESS IS NOT CAUSED BY YOUR INDEPENDENT VARIABLE this is fine

deleting those observations is wasteful, but easy (listwise deletion)

ignoring the individual missing data points is typical (casewise deletion)

imputing mean values for missing data is possibly the worst thing you can do

imputing via MI + error is currently the best option

listwise deletion is wasteful

```r
na.omit(dirty)
```

```
## [1] time        height      dept        enroll      birth.order
## <0 rows> (or 0-length row.names)
```

casewise deletion is what R does internally

```r
nrow(dirty)
```

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

```r
sum(is.na(dirty$height))
```

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

```r
sum(is.na(dirty$birth.order))
```

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

```r
length(lm(height ~ birth.order,data=dirty)$fitted.values)
```

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

this is usually the default strategy

remember how we talked about the extensibility of R?

amelia is a package that makes a complicated MI approach work without you knowing anything about its implementation

```r
library(Amelia)
```

let's use this large dataset as an example

```r
large <- read.csv('data/large.csv')
summary(large)
```

```
##        a                  b                c
##  Min.   :-33.98426   Min.   :-13.4   Min.   :-249998.64
##  1st Qu.: -6.71903   1st Qu.:128.6   1st Qu.:-141005.65
##  Median :  0.41681   Median :256.9   Median : -63498.56
##  Mean   :  0.00176   Mean   :252.2   Mean   : -83954.09
##  3rd Qu.:  7.00630   3rd Qu.:377.5   3rd Qu.: -15748.98
##  Max.   : 35.33306   Max.   :513.3   Max.   :     11.77
##  NA's   :45          NA's   :45      NA's   :45
```

```r
nrow(na.omit(large))
```

```
## [1] 871
```

for it to work you need low missingness and large N

```
a <- amelia(large,m = 1)
```

```
## -- Imputation 1 --
##
##   1  2  3
```

```
print(a)
```

```
##
## Amelia output with 1 imputed datasets.
## Return code:  1
## Message:  Normal EM convergence.
##
## Chain Lengths:
## --------------
## Imputation 1:  3
```

amelia returns a list, where the first item is a list of your imputations

we only did one, so here it is

```
large.imputed <- a[[1]][[1]]
summary(large.imputed)
```

```
##        a                    b                c
##  Min.   :-33.98426   Min.   :-13.4   Min.   :-249999
##  1st Qu.: -6.56630   1st Qu.:126.5   1st Qu.:-140641
##  Median :  0.46497   Median :252.0   Median : -63010
##  Mean   :  0.02079   Mean   :249.9   Mean   : -83310
##  3rd Qu.:  6.99412   3rd Qu.:375.2   3rd Qu.: -15626
##  Max.   : 35.33306   Max.   :568.8   Max.   :  41249
```

if you give it a tiny dataset, it will fuss at you

```
a <- amelia(large[990:1000,],m = 1)
```

```
## Warning in amelia.prep(x = x, m = m, idvars = idvars, empri = empri, ts =
## ts, : You have a small number of observations, relative to the number, of
## variables in the imputation model. Consider removing some variables, or
## reducing the order of time polynomials to reduce the number of parameters.
```

```
## -- Imputation 1 --
##
##  No missing data in bootstrapped sample:  EM chain unnecessary
```

```r
print(a)
```

```
##
## Amelia output with 1 imputed datasets.
## Return code:  1
## Message:  Normal EM convergence.
##
## Chain Lengths:
## --------------
## Imputation 1:
```

Reshaping

now that our data is clean, it's time to put it in a tidy format. this is a way of storing data that makes it easy to:

1. make graphs
2. run tests
3. summarize
4. transform into other formats

we are basically trying to organize ourselves such that:

1. any grouping is made on rows
2. any testing is done between columns

an aside on testing

in R, you use double symbols for testing

```r
1 == 2
```

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

```r
1 != 1
```

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

```r
1 >= 1
```

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

(you've already seen a couple of these)

tests return boolean vectors

```
1 >= c(0,1,2)
```

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

recall that boolean vectors need to be the same length or a divisor

if your vectors are not multiples of each other, R will fuss at you

```
c(1,2) >= c(1,2,3)
```

```
## Warning in c(1, 2) >= c(1, 2, 3): longer object length is not a multiple of
## shorter object length
```

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

```
c(1,2) >= c(1,2,3,4)      # why no warning this time? R recycles!
```

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

the combination of the length requirement, the lack of support in R for proper indexing, and missingness in your data will cause many headaches later on

subsetting data frames

subsetting your data is where you will use this regularly

```
my.data$numeric == 2
```

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

```
my.data[my.data$numeric == 2,]
```

```
## [1] n
## [2] c
## [3] b
## [4] d
## [5] really.long.and.complicated.variable.name
## <0 rows> (or 0-length row.names)
```

boolean variables can act as filters right out of the box

```
my.data[my.data$b,]
```

```
##   n   c    b          d really.long.and.complicated.variable.name
## 1 1 one TRUE 2015-07-27                                       999
## 2 2 two TRUE 2015-08-03                                       999
```

you see the empty space after the comma? that tells R to grab all the columns

you can also select columns

```r
my.data[,'d']
```

```
## [1] "2015-07-27" "2015-08-03" "2015-07-20"
```

that empy space **before** the comma? that tells R to grab all the rows

you can also match elements from a vector

```r
good.things <- c("three", "four", "five")
my.data[my.data$character %in% good.things, ]
```

```
## [1] n
## [2] c
## [3] b
## [4] d
## [5] really.long.and.complicated.variable.name
## <0 rows> (or 0-length row.names)
```

most subsetting operations on dataframes also return a dataframe

```r
str(my.data[!(my.data$character %in% good.things), ])
```

```
## 'data.frame':    0 obs. of  5 variables:
##  $ n                                       : num
##  $ c                                       : Factor w/ 3 levels "one","three",..:
##  $ b                                       : logi
##  $ d                                       :Class 'Date'  num(0)
##  $ really.long.and.complicated.variable.name: num
```

subsets that are a single column return a vector

```r
str(my.data$numeric)
```

```
##  NULL
```

most tidying can be done with two R packages

(plus a wrapper around the base string functions)

```r
install.packages('reshape2')
install.packages('stringr')
install.packages('plyr')
```

```
library(reshape2)
library(stringr)
library(plyr)
```

reshaping

our goal here is to arrange our data such that each table is about one kind of thing: whether it is everything about a measurement, everything about a person, or everything about a group of people

```
abnormal <- data.frame(name = c('Alice','Bob','Eve'),
                       time1 = c(90,90,150),
                       time2 = c(100,95,100))
```

this table is not tidy - why not?

the table is about measurements, but each measurement does not have its own row, and each type of measurement value is represented by more than one column

```
normal <- melt(data = abnormal, id.vars = 'name')
normal
```

```
##    name variable value
## 1 Alice    time1    90
## 2   Bob    time1    90
## 3   Eve    time1   150
## 4 Alice    time2   100
## 5   Bob    time2    95
## 6   Eve    time2   100
```

we can `melt` this dataframe down into a long format, which makes each row a unique observation, and then clean up the dataframe a bit

```
normal$id <- seq(1:nrow(normal))
names(normal) <- c('name','time','value','id')
normal$time <- str_replace(normal$time,'time','')
```

now that we are in a tidy format, see how easy it is to subset

```
normal[normal$time == 1,]
```

```
##    name time value id
## 1 Alice    1    90  1
## 2   Bob    1    90  2
## 3   Eve    1   150  3
```

```
normal[normal$name == 'Alice',]
```

```
##    name time value id
## 1 Alice    1    90  1
## 4 Alice    2   100  4
```

and test

> side note - don't worry about how this works yet - we'll talk about it tomorrow

```
t.test(value ~ time, data=normal)
```

```
##
##  Welch Two Sample t-test
##
## data:  value by time
## t = 0.58132, df = 2.0278, p-value = 0.6191
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -73.56101  96.89434
## sample estimates:
## mean in group 1 mean in group 2
##       110.00000        98.33333
```

it's easy to combine tidy tables to compare different levels of information simultaneously

Merging data frames

flexibly join dataframes with `merge`

imagine you have two datasets that you want to merge

```
data.1 <- read.csv('data/merge_practice_1.csv')
data.2 <- read.csv('data/merge_practice_2.csv')
```

```
## Warning in read.table(file = file, header = header, sep = sep, quote
## = quote, : incomplete final line found by readTableHeader on 'data/
## merge_practice_2.csv'
```

```
str(data.1)
```

```
## 'data.frame':    5 obs. of  4 variables:
##  $ id      : int  1 2 3 4 5
##  $ name    : Factor w/ 5 levels "Alice","Bob",..: 1 2 3 4 5
##  $ job     : Factor w/ 3 levels "communications",..: 1 1 2 1 3
##  $ location: Factor w/ 3 levels "Berkeley","Cambridge",..: 3 2 3 1 2
```

```
str(data.2)
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ id      : int  1 4 5 6
##  $ name    : Factor w/ 4 levels "Alice","Dave",..: 1 2 3 4
##  $ job     : Factor w/ 3 levels "hacker","handler",..: 1 3 2 1
##  $ location: Factor w/ 4 levels "berkeley","cambridge",..: 2 4 3 1
```

sometimes the same people have differet jobs in different locations

you can do an *inner* join using merge

```
merge(data.1, data.2, by = 'id')
```

```
##   id name.x          job.x location.x name.y   job.y location.y
## 1  1  Alice communications   New York  Alice  hacker  cambridge
## 2  4   Dave communications   Berkeley   Dave    tree  palo alto
## 3  5    Eve            spy  Cambridge    Eve handler   new york
```

that's no good - we lost half of our people!

inner joins are mostly used when you **only** want records that appear in both tables

if you want the union, you can use an outer join

```
merge(data.1, data.2, by = 'id', all = TRUE)
```

```
##   id name.x          job.x location.x name.y   job.y location.y
## 1  1  Alice communications   New York  Alice  hacker  cambridge
## 2  2    Bob communications  Cambridge   <NA>    <NA>       <NA>
## 3  3  Chuck         hacker   New York   <NA>    <NA>       <NA>
## 4  4   Dave communications   Berkeley   Dave    tree  palo alto
## 5  5    Eve            spy  Cambridge    Eve handler   new york
## 6  6   <NA>           <NA>       <NA>  Faith  hacker   berkeley
```

this works basically the same as `join` in SQL

running merges is particularly useful when:

a. your data is tidy; and,
b. you want to add information with a lookup table

in this case, you can store your lookup table as a dataframe, then merge it

```
lookup <- read.csv('data/merge_practice_3.csv')
str(lookup)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ location  : Factor w/ 5 levels "Berkeley","Cambridge",..: 2 3 1 4 5
##  $ population: int  107289 8406000 116768 66642 233294
```

this lookup table gives us the population for each location

we can add this to our people table with

```
merge(data.1, lookup, by = "location")
```

```
##    location id  name            job population
## 1  Berkeley  4  Dave communications     116768
## 2 Cambridge  2   Bob communications     107289
## 3 Cambridge  5   Eve            spy     107289
## 4  New York  1 Alice communications    8406000
## 5  New York  3 Chuck         hacker    8406000
```

note that Reno was in our lookup table

```
lookup[lookup$location == 'Reno', ]
```

```
##   location population
## 5     Reno     233294
```

but doesn't show up when we merge - why do you think this is?

Transforming data

introduction

because R started out as a functional language, it can be hard to modify data, especially in place

in practice, if you want 100% control over how your frames are being modified, you'll be writing lots of `for` loops, which is messy

luckily, there is a package that handles the common tasks for you

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
##
## The following objects are masked from 'package:plyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize
##
## The following objects are masked from 'package:stats':
##
##     filter, lag
##
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

sort data with `arranage`

base R syntax for sorting is a bit of a pain in that you have to create a sorting vector based on the values in a column, then subset the same dataframe and apply the sorting vector to the rows slice

to demonstrate this, let's start by making a toy data frame

```
toy <- data.frame(
  id = c(1,1,1,2,2,2,3,3,3),
  score.1 = c(90,94,40,80,80,80,76,80,82)
)
arrange(toy, score.1)
```

```
##   id score.1
## 1  1      40
## 2  3      76
## 3  2      80
## 4  2      80
## 5  2      80
## 6  3      80
## 7  3      82
## 8  1      90
## 9  1      94
```

select rows by pattern with `select`

it's common for variables that measure similar things to have similar names, but selecting columns this was in base R requires running `grepl` on column names, then subsetting the dataframe and applying the logical vector to the column field

```
toy$score.2 <- 100
select(toy, score.1, score.2)
```

```
##   score.1 score.2
## 1      90     100
## 2      94     100
## 3      40     100
## 4      80     100
## 5      80     100
## 6      80     100
## 7      76     100
## 8      80     100
## 9      82     100
```

```
select(toy, contains('score'))
```

```
##   score.1 score.2
## 1      90     100
## 2      94     100
## 3      40     100
## 4      80     100
## 5      80     100
## 6      80     100
## 7      76     100
## 8      80     100
## 9      82     100
```

apply summary fucntions with `summarise`

dplyr includes most of the base R summary statistics, along with:

- `n()`
- `n_distinct()`
- `first()`

- `last()`

```
summarise(toy, n(), n_distinct(score.1), last(score.1))
```

```
##   n() n_distinct(score.1) last(score.1)
## 1   9                   6            82
```

dplyr allows you to apply functions to groups

so far, these have taken base R functions and made them faster (with C++ calls behind the scenes), easier to use, or both

dplyr's real utility is in its grouped dataframes, which apply dplyr functions groupwise

```
group_by(toy, id)
```

```
## Source: local data frame [9 x 3]
## Groups: id
##
##   id score.1 score.2
## 1  1      90     100
## 2  1      94     100
## 3  1      40     100
## 4  2      80     100
## 5  2      80     100
## 6  2      80     100
## 7  3      76     100
## 8  3      80     100
## 9  3      82     100
```

```
summarise(group_by(toy, id), n(), n_distinct(score.1))
```

```
## Source: local data frame [3 x 3]
##
##   id n() n_distinct(score.1)
## 1  1   3                   3
## 2  2   3                   1
## 3  3   3                   3
```

you can add as many functions as you want inbetween, but wrapping function call around function call can be hard to read (and write!)

you can pipe functions with the %>% operator

this will look very familiar if you are used to working in bash

```
toy %>% group_by(id) %>% summarise(n(), n_distinct(score.1))
```

```
## Source: local data frame [3 x 3]
##
##   id n() n_distinct(score.1)
## 1  1   3                   3
## 2  2   3                   1
## 3  3   3                   3
```

Practice

Grab some data from Pew

and sanitize/tidy it

this will be hard

```
library(foreign)
pew <- as.data.frame(read.spss("data/pew.sav"))
```

```
## re-encoding from CP1252
```

```
## Warning in `levels<-`(`*tmp*`, value = if (nl == nL) as.character(labels)
## else paste0(labels, : duplicated levels in factors are deprecated
```

```
religion <- pew[c("q16", "reltrad", "income")]
rm(pew)
```

we'll start by cleaning up the factor variables

```
religion$reltrad <- as.character(religion$reltrad)
religion$reltrad <- str_replace(religion$reltrad, " Churches", "")
religion$reltrad <- str_replace(religion$reltrad, " Protestant", " Prot")
religion$reltrad[religion$q16 == " Atheist (do not believe in God) "] <- "Atheist"
religion$reltrad[religion$q16 == " Agnostic (not sure if there is a God) "] <- "Agnostic"
religion$reltrad <- str_trim(religion$reltrad)
religion$reltrad <- str_replace_all(religion$reltrad, " \\(.*?\\)", "")

religion$income <- c("Less than $10,000" = "<$10k",
  "10 to under $20,000" = "$10-20k",
  "20 to under $30,000" = "$20-30k",
  "30 to under $40,000" = "$30-40k",
  "40 to under $50,000" = "$40-50k",
  "50 to under $75,000" = "$50-75k",
  "75 to under $100,000" = "$75-100k",
  "100 to under $150,000" = "$100-150k",
  "$150,000 or more" = ">150k",
  "Don't know/Refused (VOL)" = "Don't know/refused")[religion$income]

religion$income <- factor(religion$income, levels = c("<$10k", "$10-20k", "$20-30k", "$30-40k", "$40-50k
  "$75-100k", "$100-150k", ">150k", "Don't know/refused"))
```

now we can reduce this down to three columns for three variables

```
religion <- count(religion, reltrad, income)
names(religion)[1] <- "religion"
```

Acknowledgements

Materials taken from:

Chris Krogslund Hadley Wickham