Data manipulation and cleaning

Tad Dallas

Reading for this week:

> https://paldhous.github.io/ucb/2016/dataviz/week7.html

Data

Data is life. We would likely not be using a statistical programming language like `R` if we were not thinking of applying the tools we learn to some data. I sincerely hope this is the case, for the sake of your final projects. We have covered basic `R` commands that allow us to work with data. Here, we will go over specific subsetting and data manipulation operations.

> A note on data cleaning: Best practices are to never directly edit your raw data files. Ideally, any pre-processing steps necessary before manipulation and analysis would be completed programmatically.

Data manipulation

Here, we differentiate "data cleaning" from "data manipulation", which is perhaps an arbitrary distinction. "Data cleaning" typically refers to altering variable class information, fixing mistakes that could have arisen in the data (e.g., an extra '.' symbol in a numeric value), and things of this nature. "Data manipulation", in my mind, refers to altering the structure of the data in a way that changes the functional structure the data (e.g., an addition of a column, deletion of rows, long/wide formatting change).

We briefly touched on `R` packages previously. Packages are incredibly useful, as they can make complicated analyses or issues quite simple (i.e., somebody else has already done the heavy-lifting). However, we also must bear in mind that each package we use adds a dependency to our code. That package you use might be available now, but an update to `R` might easily break it. The ease of package creation in `R` has created a situation where creation occurs but maintenance does not, resulting in lots of link rot and deprecated packages. For all of the faults of CRAN (The Comprehensive R Archive Network), they recognize this as an issue, and try to archive and standardize package structures. But wow, Brian Ripley can be a bit abrasive.

gapminder data

The gapminder data are commonly used to explore concepts of data exploration and manipulation, maybe because of the combination of character and numeric variables, nested structure in terms of country and year, or maybe it is just out of ease in copying notes from other people.

> some of the material presented here has been adapted from the great work of Jenny Bryan (https://jennybryan.org/).

```
dat <- read.delim(file = "http://www.stat.ubc.ca/~jenny/notOcto/STAT545A/examples/gapminder/data/gapmind
```

```
head(dat)
```

```
##       country year      pop continent lifeExp gdpPercap
## 1 Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2 Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3 Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4 Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5 Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6 Afghanistan 1977 14880372      Asia  38.438  786.1134
```

```
str(dat)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ gdpPercap: num  779 821 853 836 740 ...
```

We can use what we learned before in terms of base `R` functions to calculate summary statistics.

```
# mean life expectancy
mean(dat$lifeExp)
```

```
## [1] 59.47444
```

But what does mean life expectancy really tell us, when we also have information on space (`country`) and time (`year`)? So we may wish to subset the data to a specific country or time period. We can do this using `which` statements.

```
dat[which(dat$country=='Afghanistan'), ]
dat[which(dat$year < 1960), ]
```

Recall that `which` evaluates a condition, and then determines the index of each TRUE value. So for the first example, the `which` tells us the indices where the vector `dat$country` is equal to "Afghanistan". Putting this result vector of indices within the square brackets allows us to subset the data.frame based on these indices (specifically, we are subsetting specific rows of data).

In the second example, we want to see all data that was recorded prior to 1960. As you will quickly realize, there are always multiple ways to do the same thing when programming. For instance, this second statement could be done in base `R` using the `subset` function.

```
subset(dat, dat$year < 1960)
```

The `subset` function also allows you to 'select' specific columns in the output.

```
subset(dat, dat$year < 1955, select=c(lifeExp,gdpPercap))
```

However, this is the same as

```
dat[which(dat$year < 1960), c("lifeExp","gdpPercap")]
```

To refresh your memory and clarify the use of conditionals, the list below provides a bit more information.

- ==: equals exactly
- <, <=: is smaller than, is smaller than or equal to
- >, >=: is bigger than, is bigger than or equal to
- !=: not equal to

And some that we did not go into before, but will go into a bit more detail on now:

- !: NOT operator, to specify things that should be omitted
- &: AND operator, allows you to chain two conditions which must both be met
- |: OR operator, to chains two conditions when at least one should be met
- %in%: belongs to one of the following (usually followed by a vector of possible values)

The NOT operator is super useful, as it is always better to index existing cases than to remove cases. An example would be if we wanted to ignore all cases in the gapminder data with lifeExp value that is NA.

```
dat[!is.na(dat$lifeExp),]
dat[-is.na(dat$lifeExp),]
```

These two are essentially the same statement, so why do they display such different results?

The AND (&) and the OR (|) operators are also super useful when you want to separate data based on multiple conditions.

```
dat[which(dat$country=='Afghanistan' & dat$year==1977),]
dat[which(dat$lifeExp < 40 | dat$gdpPercap < 500), ]
```

Finally, the %in% operator is super useful when you want to subset data based on multiple conditions

```
#fails
dat[which(dat$country == c('Afghanistan', 'Turkey')), ]
```

```
##          country year      pop continent lifeExp gdpPercap
## 1    Afghanistan 1952  8425333      Asia  28.801  779.4453
## 3    Afghanistan 1962 10267083      Asia  31.997  853.1007
## 5    Afghanistan 1972 13079460      Asia  36.088  739.9811
## 7    Afghanistan 1982 12881816      Asia  39.854  978.0114
## 9    Afghanistan 1992 16317921      Asia  41.674  649.3414
## 11   Afghanistan 2002 25268405      Asia  42.129  726.7341
## 1574      Turkey 1957 25670939    Europe  48.079 2218.7543
## 1576      Turkey 1967 33411317    Europe  54.336 2826.3564
## 1578      Turkey 1977 42404033    Europe  59.507 4269.1223
## 1580      Turkey 1987 52881328    Europe  63.108 5089.0437
## 1582      Turkey 1997 63047647    Europe  68.835 6601.4299
## 1584      Turkey 2007 71158647    Europe  71.777 8458.2764
```

```
#does not fail
dat[which(dat$country %in% c('Afghanistan', 'Turkey')), ]
```

```
##          country year      pop continent lifeExp gdpPercap
## 1    Afghanistan 1952  8425333      Asia  28.801  779.4453
## 2    Afghanistan 1957  9240934      Asia  30.332  820.8530
## 3    Afghanistan 1962 10267083      Asia  31.997  853.1007
## 4    Afghanistan 1967 11537966      Asia  34.020  836.1971
## 5    Afghanistan 1972 13079460      Asia  36.088  739.9811
## 6    Afghanistan 1977 14880372      Asia  38.438  786.1134
## 7    Afghanistan 1982 12881816      Asia  39.854  978.0114
## 8    Afghanistan 1987 13867957      Asia  40.822  852.3959
## 9    Afghanistan 1992 16317921      Asia  41.674  649.3414
## 10   Afghanistan 1997 22227415      Asia  41.763  635.3414
## 11   Afghanistan 2002 25268405      Asia  42.129  726.7341
## 12   Afghanistan 2007 31889923      Asia  43.828  974.5803
## 1573      Turkey 1952 22235677    Europe  43.585 1969.1010
## 1574      Turkey 1957 25670939    Europe  48.079 2218.7543
```

```
## 1575      Turkey 1962 29788695    Europe  52.098 2322.8699
## 1576      Turkey 1967 33411317    Europe  54.336 2826.3564
## 1577      Turkey 1972 37492953    Europe  57.005 3450.6964
## 1578      Turkey 1977 42404033    Europe  59.507 4269.1223
## 1579      Turkey 1982 47328791    Europe  61.036 4241.3563
## 1580      Turkey 1987 52881328    Europe  63.108 5089.0437
## 1581      Turkey 1992 58179144    Europe  66.146 5678.3483
## 1582      Turkey 1997 63047647    Europe  68.835 6601.4299
## 1583      Turkey 2002 67308928    Europe  70.845 6508.0857
## 1584      Turkey 2007 71158647    Europe  71.777 8458.2764
```

Related to %in%, is `match`. `match` is best for identifying the index of single types in a vector of unique values. For instance,

```
dat[match(c('Afghanistan', 'Turkey'), dat$country),]
```

```
##          country year      pop continent lifeExp gdpPercap
## 1    Afghanistan 1952  8425333      Asia  28.801  779.4453
## 1573      Turkey 1952 22235677    Europe  43.585 1969.1010
```

only returns two rows, because it only matches the first instance of both countries in the data. We can use match to get the index associated with a single value (useful when writing functions).

```
match('dog', c('dog', 'cat', 'snake'))
```

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

```
#not ideal behavior
match('dog', c('dog', 'cat', 'snake', 'dog'))
```

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

or it can be used to identify multiple instances of a single value across a vector of values.

```
match(c('dog', 'cat', 'snake', 'dog'), 'dog')
```

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

```
match(c('dog', 'cat', 'snake', 'dog'), c('dog', 'cat'))
```

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

While a bit opaque, these functions are pretty useful in a variety of situations. Speaking of data manipulation functions that are useful but a bit conceptually difficult, `do.call` and `Reduce` are solid base `R` functions.

`do.call` is a way of calling the same function recursively on multiple objects, and may have similar output to `Reduce`, which is also a way to recursively apply a function.

```
lst <- list(1:10)
lst
```

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

```
#this makes a single rbind call with each element of the list as an argument
do.call(rbind, lst)
```

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

```
#this does it iteratively (so makes n-1 rbind calls)
Reduce(rbind, lst)
```

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

```
# Reduce isn't specifically designed to work on list objects, while do.call is
do.call(sum, lst)
```

```
## [1] 55
```

```
Reduce(sum, lst)
```

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

```
Reduce(sum, unlist(lst))
```

```
## [1] 55
```

The tidyverse

The general goal of the `tidyverse` is to create a set of interconnected packages with the same overarching goal, which is to promote so-called 'tidy' data. This corresponds to each row being an observation of a specific set of conditions or treatments. This is perhaps best shown by looking back at the gapfinder data we read in above. There, the variables of interest that vary across levels are population size (`pop`), life expectancy (`lifeExp`), and GDP per capita (`gdpPercap`). The other variables serve as nesting columns, corresponding to information on country, year, and continent. These values are repeated throughout the data, while the other variables are not. Sometimes this structure of data is referred to as "long". Long data are arguably more conducive to analysis, due to some stuff about key-value pairing of data structures that I will not go into. "wide" data, on the other hand, would have one of the nesting variables (e.g., `year`) as a series of columns, with rows corresponding to another one of the nesting variables (e.g., `country`), and entries corresponding to the continuous variables. For the sake of this class, we will strive, or even sometimes just assume, that data are in the "long" format.

There are many `R` libraries designed to manipulate data and work with specific data structures (e.g., `purrr` for list objects, `lubridate` for dates, etc.). For the sake of brevity and generality, we will examine two main useful packages for data manipulation: `plyr` and `dplyr`. These are two of the near-original `tidyverse` packages developed by Hadley Wickham. They are solid. We will also use many base `R` functions for data manipulation.

```
install.packages('plyr')
install.packages('dplyr')

library(plyr)
library(dplyr)
```

plyr-specific functions

If there is one thing that tidyverse packages love to do, it is to reinvent the wheel and claim to have invented it. That is, much of the functionality you will see in the tidyverse packages is in base `R`, but the tidyverse just makes it easier, providing a nice wrapper for existing functionality. One such implementation of this is the use of the `XYply` statements as nice wrappers to more classic `apply` statements. Here, `X` and `Y` can take values of 'a', 'l', or 'd', depending on the input or output data structure desired. For instance, if we have a list that we would like to apply over and return a data.frame, we would use `ldply`, where the `l` is claiming that the input is a list object, and the `d` is claiming that the output should be formatted as a data.frame. Other examples of this syntax would be `adply`, `ddply`, `laply`, `aaply`, etc. etc.

Below, I provide an example of the aXply syntax (e.g., adply, alply, aaply).

```
my.array = array(1:27, c(3,3,3))
rownames(my.array) = c("Curly", "Larry", "Moe")
colnames(my.array) = c("Groucho", "Harpo", "Zeppo")
dimnames(my.array)[[3]] = c("Bart", "Lisa", "Maggie")
my.array
```

```
## , , Bart
##
##       Groucho Harpo Zeppo
## Curly       1     4     7
## Larry       2     5     8
## Moe         3     6     9
##
## , , Lisa
##
##       Groucho Harpo Zeppo
## Curly      10    13    16
## Larry      11    14    17
## Moe        12    15    18
##
## , , Maggie
##
##       Groucho Harpo Zeppo
## Curly      19    22    25
## Larry      20    23    26
## Moe        21    24    27
```

Arrays are something that we did not introduce when we talked about `R` basics, and that is because they really are not used *too* often. Think of matrix. It has two dimensions (x and y), so it can be viewed as a rectangle of data. Arrays simply add more dimensions. In the example above, there is another dimension, forming a data cube (in the rectangle analogy).

We can use `plyr` functionality to operate on this array and return different forms. For instance, `aaply` takes an array and returns a simplified array (here a vector).

```
plyr::aaply(my.array, 1, sum)
```

```
## Curly Larry   Moe
##   117   126   135
```

We can change one letter and now return a data.frame containing two columns. This is also a good time to point out the flexibility of the XYply statements to different margins. Margins (denoted as `.margins` argument in `R`, asks along which axis you would like to operate on the array. If we set .margins=1, this corresponds to a row-wise operation, so we calculate the sum across the array for Curly, Larry, and Moe. If we change this to .margins=2, we operate on columns, and will return sums for Groucho, Harpo, and Zeppo. And if we use .margins=3, we will return sums for Bart, Lisa, and Maggie.

```
plyr::adply(.data=my.array, .margins=1, .fun=sum)
```

```
##      X1  V1
## 1 Curly 117
## 2 Larry 126
## 3   Moe 135
```

```
plyr::adply(.data=my.array, .margins=2, .fun=sum)
```

```
##        X1  V1
```

```
## 1 Groucho  99
## 2   Harpo 126
## 3   Zeppo 153
```

```
plyr::adply(.data=my.array, .margins=3, .fun=sum)
```

```
##       X1  V1
## 1   Bart  45
## 2   Lisa 126
## 3 Maggie 207
```

Finally, we can return a list object. In this use case, this is not super helpful, but in other use cases the list output is pretty helpful.

```
plyr::alply(.data=my.array, .margins=1, .fun=sum)
```

```
## $`1`
## [1] 117
##
## $`2`
## [1] 126
##
## $`3`
## [1] 135
##
## attr(,"split_type")
## [1] "array"
## attr(,"split_labels")
##      X1
## 1 Curly
## 2 Larry
## 3   Moe
```

A pitch for `plyr::ldply`. I really like this function, as I often find myself with lists of similar structures that I want to operate on and get a single clean object back. I will not go into an example, but this is a pretty useful function (though all the utility is basically contained in `vapply`).

Finally, you may wonder why am I pushing apply statements so hard. It has nothing to do with speed, and only a bit to do with code clarity. The main advantage is understanding the programmatic nature of apply statements (which will be similar but less chronological than a for loop), and many parallel computing packages have their own little versions of apply statements ready to go (e.g., `parallel::mclapply`, `parallel::parLapply`, `parallel::clusterApplyLB`).

dplyr-specific functions

rename

```
df <- data.frame(A=runif(100), B=runif(100), D=rnorm(100,1,1))
df2 <- dplyr::rename(df, a=A, b=B, d=D)
```

This is the same functionality as the base `R` function `colnames` (or `names` for a data.frame)

```
names(df2) <- c('a', 'b', 'd')
#or
names(df2) <- tolower(names(df))
```

The nice part about `dplyr::rename()` is that we specify the old and new column names, meaning that there is little risk of an indexing error as with using the `colnames()` or `names()` functions.

select

Many of the next functions are directly analogs of functions from another programming language used to query databases (SQL). This makes it really nice to learn, as you can essentially learn two languages while learning one. SQL is pretty powerful when working with relational data. I will not go into what I mean by this, unless there is time during lecture and interest among you all.

We use `dplyr::select` when we want to…select…columns.

```
dplyr::select(df2, a)
dplyr::select(df2, obs = starts_with('a'))
```

filter

`dplyr::filter` is another one of those useful functions that we already know how to use in base `R`. Previously, we have used `which` statements or the `subset` function. `dplyr::filter` is used to filter down a data.frame by some condition applied to rows.

```
dplyr::filter(df2, a < 0.5)
```

mutate

`dplyr::mutate` is used when we wish to create a new covariate based on our existing covariates. For instance, if we wanted to create a column `e` on `df2` that was the sum of `a+b` divided by `d`…

```
df2 <- dplyr::mutate(df2, e=(a+b)/d)
head(df2,5)
```

```
##           a         b          d          e
## 1 0.2933658 0.4865453  1.5870110  0.4914340
## 2 0.4369177 0.1743441 -0.4417302 -1.3837900
## 3 0.3084441 0.2230921  0.7636438  0.6960525
## 4 0.0612698 0.2452349  2.0129651  0.1522653
## 5 0.8831706 0.8917788  0.3535045  5.0210093
```

Notice that the function creates a new column and appends it to the existing data.frame, but does not "write in place". That is, the `df2` object is not modified unless it is stored (which we do above).

group_by

`dplyr::group_by` is really useful as an intermediate step to getting at summary statistics which take into account grouping by a character or factor variable. For instance, if we wanted to calculate the mean life expectancy (`lifeExp`) for every `country` in the gapminder data (`dat`), we would first have to group by `country`.

```
datG <- dplyr::group_by(dat, country)
```

This is a bit like a non-function, since `dat` and `datG` are essentially the same….but they are not for the purposes of computing group-wise statistics. This is done using the `dplyr::summarise` function.

summarise

So if we wanted to calculate mean life expectancy (`lifeExp`) per country, we could use the grouped data.frame `datG` and the `dplyr::summarise` function to do so.

```
dplyr::summarise(datG, mnLife=mean(lifeExp))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 142 x 2
##    country     mnLife
##    <fct>        <dbl>
##  1 Afghanistan   37.5
##  2 Albania       68.4
##  3 Algeria       59.0
##  4 Angola        37.9
##  5 Argentina     69.1
##  6 Australia     74.7
##  7 Austria       73.1
##  8 Bahrain       65.6
##  9 Bangladesh    49.8
## 10 Belgium       73.6
## # ... with 132 more rows
```

joins

joins are something taken directly from SQL. Table joins are ways of combining relational data by some index variable. That is, we often have situations where our data are inherently multi-dimensional. If we have a data.frame containing rows corresponding to observations of a species at a given location, we could have another data.frame containing species-level morphometric or trait data. While we could mash this into a single data.frame, it would repeat many values, which is not ideal for data clarity or memory management.

```
df$species <- sample(c('dog', 'cat', 'bird'),100, replace=TRUE)

info <- data.frame(species=c('dog', 'cat', 'bird', 'snake'),
    annoying=c(10, 2, 100, 1),
    meanBodySize=c(20, 5, 1, 2))
```

Now we can join some stuff together, combining data on mean species-level characteristics with individual-level observations.

```
# maintains the structure of df (the "left" data structure)
left_join(df, info, by='species')
# maintains the structure of info (the "right" data structure)
right_join(df,info, by='species')

# return things that are in info but not in df
anti_join(info, df, by='species')
```

There are other forms of joins (`full_join`, `inner_join`, etc.), but I find that I mostly use the `left` or `right` variations of the joins, as it specifically allows me to control the output (i.e., using `dplyr::left_join`, I know that the resulting data.frame will have the same number of rows as the left hand data.frame).

piping

Alright. So before we discussed joins, we were describing the different main verbs of `dplyr`. We discussed `rename`, `select`, `mutate`, `group_by`, and `summarise`. A final point, and something `tidyverse` folks really love, is the use of these functions in nested statements through the use of piping.

Pipes in bash scripting look like |, pipes in `R` syntax look like %>%. It does not matter what it looks like though, it matter what it does. Here is a simple example of the use of piping. We can go back to the example of calculating the mean life expectancy per country from the gapminder data.

The usual way

```
tmp <- dplyr::group_by(dat, country)
tmp2 <- dplyr::summarise(tmp, mnLifeExp=mean(lifeExp))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

The piped way

```
tmp3 <- dat %>%
    dplyr::group_by(country) %>%
    dplyr::summarise(mnLifeExp=mean(lifeExp))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

The results of these two are identical (`all(tmp3==tmp2)` returns TRUE).

This is useful, as commands can be chained together, including the creation of new variables, subsetting and summarising of existing variables, etc. One thing to keep in mind is to check intermediate results – instead of just piping all the way through – as data manipulation errors can be introduced mid-statement and go unnoticed. That is, in some situations, piping does not help reproducibility. Many proponents argue that it helps with code readability, while many others say that actively makes code less human readable. It definitely does require adopting a certain syntax and the assumption that every end user is on the `tidyverse` train, which is not ideal when reproducibility involves everyone, not just the cool tidy/Hadley/RStudio crowd.