Data Manipulation

http://datascience.tntlab.org

Module 5

Today's Agenda

- A couple of base-R notes
  - Advanced data typing
  - Relabeling text
- In depth with *dplyr* (part of tidyverse)
  - tbl class
  - *dplyr* grammar
  - Grouping
  - Joins and set operations
- A warning about *dplyr* (and packages broadly)

Advanced Data Typing

- There is no class that stores a single number. Even a<-1 creates a vector.
- Data frames are lists of vectors that act like matrices.
- Do not try to memorize the coercion rules. Use local testing and use *dplyr*, which will warn you when it coerces. Be aware that coercion occurs and be on the lookout for situations where it looks like it might happen.
- Classes are vectors with special characteristics. You can create your own classes (e.g., tbl_df).
- Factors will cause you nightmares. Try not to allow anything to be a factor until/unless you specifically need a factor.

Relabeling Text (base R)

- Create a "lookup table" that is actually a *named vector*
  - c("A"="Experimental", "B"="Control")
- For each value in a target vector, get value from lookup table
  - c("A"="Experimental", "B"="Control")[c("A","A","B","B","B")]
- Save this back wherever you want it
  - my_df$condition <- lookuptable[my_df$condition]

Heart of Tidyverse Data: tbl class

- Tibbles are type of data frame with extra features in line with the tidy philosophy
  - Does not change types from what they obviously should be (don't need stringsAsFactors=FALSE)
  - Easier to work with lists inside tbl documents than with df
  - Does not arbitrarily change column names (e.g., no "my name" to "my.name")
  - Evaluates arguments sequentially by column
  - Does not allow row names (because rows shouldn't have names)
  - When you display one, gives more useful information
  - Subsetting consistently returns tbls (not true for a data.frame)
  - Extraction requires complete column names

- *as.tbl, as.tibble, as_data_frame* and *is.tbl, is.tibble*
- Use *tibble()* instead of *data.frame()*

Dataframe Pliers (*dplyr*): Verbs

- Data wrangling cheat sheet is *really* handy here

- Five common types
  - Subset columns, use select() w/helper functions (esp., contains(), matches())
  - Subset rows, use filter(), distinct(), and slice()
  - Sort rows by variables, use arrange(), sometimes with desc()
  - Create new columns, use mutate() and transmute() with window functions
  - Create new summary df with summarize() and summary functions

- All use the standard tidy philosophy and tbls
  - Always specify the tbl first, then verb parameters
  - You are discouraged from subsetting the base-R way, e.g., [, 4:5] or [1:2,]
  - Try to maintain your data pipeline

Common Problem at this Stage

- You will need to either remember or check which functions evaluate *values* and which functions evaluate *variables*
  - is.character() evaluates a variable
  - is.na() evaluates a value

- If you forget, dplyr will sometimes fail silently and you will be confused
  - filter(my_tbl, is.numeric(x))
  - What should this be?

- select(), filter(), arrange() and mutate() modify an existing dataset
- summarize() *creates a new dataset;* only variables you specify to be retained will be retained
- Some functions drop referenced variables after use (gather(), spread(), transmute()) and others don't (mutate()) but you can change this.

magrittr: Piped Functions

- *Magrittr* includes many different types of pipes beyond %>%, but %>% itself is included in all core tidyverse packages
- You can use *magrittr* in any code, not just when using *dplyr*, but they must adhere to the format: take the output of the previous function and use it as the first parameter in a second
- %>% is pronounced "then"

- These are equivalent in final output
  - a <- c(1,2,3); mean(a)
  - c(1, 2, 3) %>% mean()

  - "Create a vector, then calculate the mean of that vector"

Best Practices with *magrittr*

- On the first line, include any variable assignment plus the source of the data only (e.g., could be a data frame itself or the result of a join)
  - new_tbl <- old_tbl %>%
- Indent one tab for each additional verb; try not to nest verbs
    - filter(x == 1) %>%
    - summarize(mean(x))

- Remember that you do not need to do variable assignment if you don't need that information later in your code

- Examples of good *magrittr* form:

```
new_tbl <- old_tbl %>%
    filter(x == 1) %>%
    summarize(mean(x))
```

```
old_tbl %>%
    filter(x == 1) %>%
    summarize(mean(x))
```

Grouping

- To create explicit groups, use the group_by() verb
  - Does not subset; does not sort; does nothing but create grouping information which is then used by other verbs

- Once you group_by(), you can summarize() and then apply additional verbs
  - Useful if you need within-group summary statistics
  - Most useful for us in the context of exploring multi-level datasets

Databases

- Database logic is everywhere in data science, so it's best if you learn the language
  - Keys are identifiers in datasets
  - *Primary keys* are unique identifiers, e.g., participant numbers
  - *Secondary keys* are non-unique identifiers, e.g., condition numbers
  - All tables should have primary keys, but these primary keys are not always useful

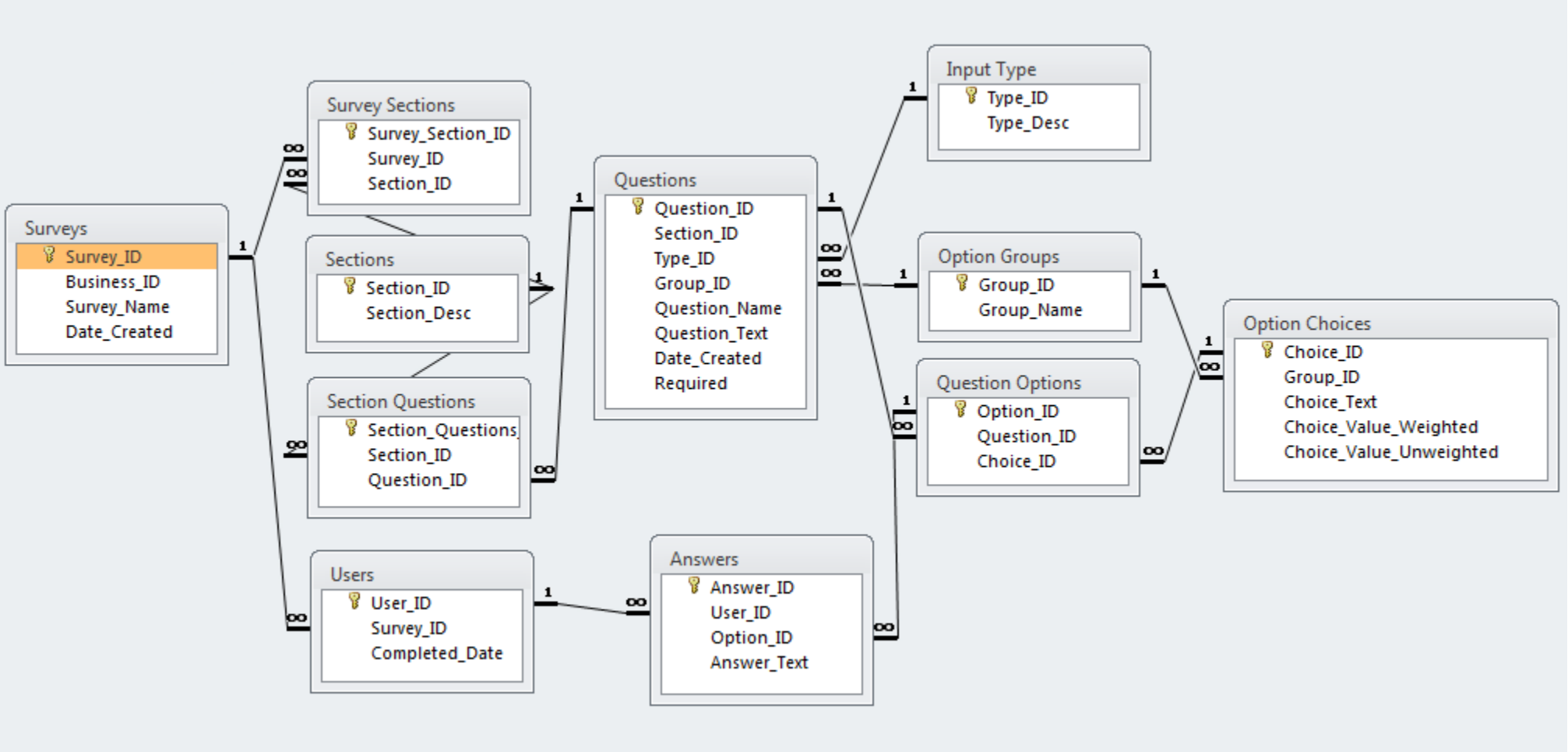
  - respondent_df
    - rID
    - rName

    questions_df
    - qID
    - qText
    - qResponses

    answers_df
    - qID
    - rID
    - response

- The above specifications are called a *schema*

Schema Diagram



(borrowed from stackoverflow.com)

Joins

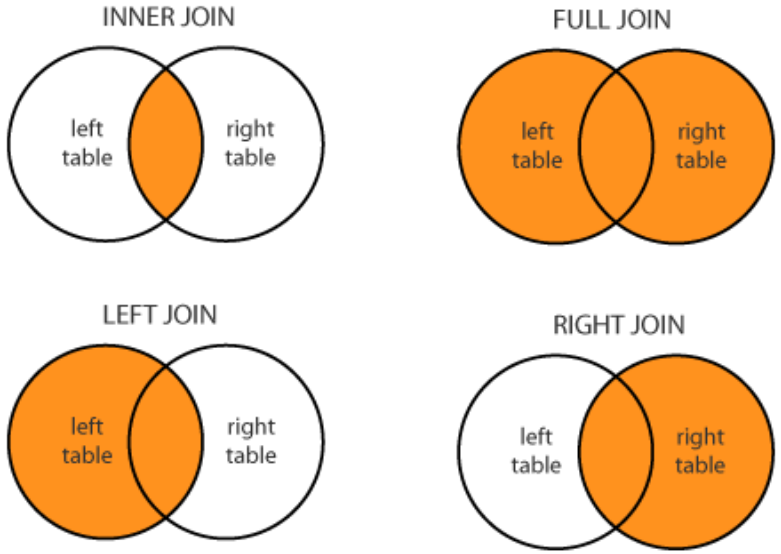
- The two differences between all joins are:
  - what is treated as a key
  - what is dropped

- Always think about joins as if there are literally two data files side by side: a left-hand side dataset (LHS) and a right-hand-side datasets (RHS).
  - All join functions take them in that order: *join*(LHS, RHS, …)

- Common mistake with joins!
  - Although we're in *tidyverse*, the by statement takes a character vector, not variable quasi-notation

Mutating Joins

- **Left join**: Take the LHS, and add columns to it from the RHS.
  For rows on the RHS where the LHS key is missing, drop those rows.
- **Right join**: Take the RHS, and add columns to it from the LHS.
  For rows on the LHS where the RHS key is missing, drop those rows.
- **Inner join**: Take the LHS, and add columns to it from the RHS.
  For rows on either side where the key is missing in the other, drop those rows.
- **Full join** (aka **outer join**): Take the LHS, and add columns to it from the RHS.
  Retain all rows.



Filtering Joins

- **Semi join**: Perform a left join, but run only rows and columns **from LHS** that match rows in RHS
- **Anti join**: Perform a left join, but return only the rows **from LHS** that did not match a row in RHS

- You can recreate all of these using different joins plus other *dplyr* functions

Set Operations

- **union():** Useful for combining observations across multiple identically formatted datasets
  - Particularly useful if you have multiple data collection efforts (e.g., two identical or Qualtrics surveys)
  - Also useful in combination with other dplyr functions if slightly different
  - Similar to rbind (or bind_rows), but **only returns unique rows**

- **intersect()**: Identify rows identical between datasets; less useful for us
- **setdiff()**: Identify rows different between datasets; also less useful for us

- **setequal():** Determine if two datasets contain the same data
- **identical():** Determine if two datasets contain the same data in the same order

Raw Data Manipluation

- bind_rows() instead of rbind(), bind_cols() instead of cbind()
  - Can bind within lists

- Very useful when binding rows: Data source indicator variables
  - bind_rows(name1 = one_df, name2 = two_df, .id = “identifier”)

- **You generally want to join instead of bind_cols**
  - Column binding is only useful when you are 100% certain two data files are formatted the same way
  - This is most valuable as the middle step of several different operations

Important Warning

- *dplyr* is the first package we've dived into deeply that is under *active development* with *frequent updates*.
  - New versions can be released any time.
  - Existing functions may be *deprecated*; this function works for now but may disappear from the package in the future. Deprecation is generally used to provide temporary *backwards compatibility*.
  - New functions may be added that make things easier than they were before.

  - The data wrangling cheat sheet we've been looking at is *already out-of-date*.
    - Example: *summarize_each()* is deprecated and should be replaced with *summarize_all*(), *summarize_at*(), or *summarize_if*()

  - You can stay up-to-date on packages by reading *changelogs* or *news*
    - R updates are accessible by calling *news()*
    - You can also use *news(package="packagename")* but a lot of package maintainers don't keep their news updated
    - Google is your friend