Data wrangling

Katie Schuler

Acknowledgement

These notes are adapted from Ch 5 Data tidying, Ch 7 Data import and Ch 4 Data transformation in R for Data Science

Tidy

Welcome to the tidyverse

> The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.
> ~ Tidyverse package docs

The `tidyverse` collection of packages includes:

- `ggplot2` - for data visualization
- `dplyr` - for data wrangling
- `readr` - for reading data
- `tibble` - for modern data frames
- `stringr`: for string manipulation
- `forcats`: for dealing with factors
- `tidyr`: for data tidying
- `purrr`: for functional programming
- `lubridate`: for working with dates and times

We load the `tidyverse` like any other package, with `library(tidyverse)`. When we do, we will receive a message with (1) a list packages that were loaded and (2) a warning that there are potential conflicts with base R's `stats` functions

- We can resolve conflicts with the `::` operator, which allows us to specify which package our intended function belongs to as a prefix: `stats::filter()` or `dplyr::filter()`

What is tidy data?

The same underlying data can be represented in a table in many different ways; some easier to work with than others. The tidyverse makes use of tidy data principles to make datasets easier to work with in R. **Tidy data** provides a standard way of structuring datasets:

1. each variable forms a **column**; each column forms a variable
2. each observation forms a **row**; each row forms an observation
3. value is a **cell**; each cell is a single value

Why is tidy data easier to work with?

- Because consistency and uniformity are very helpful when programming
- Variables as columns works well for vectorized languages (R!)

Functional programming with purrr

> purrr enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors. If you've never heard of FP before, the best place to start is the family of map() functions which allow you to replace many for loops with code that is both more succinct and easier to read. ~ purrr docs

Let's illustrate the joy of the tidyverse with one of its packages: `purrr`. The docs say that the best place to start is the family of `map()` functions, so we'll do that.

The `map()` functions:

1. take a vector as input
2. apply a function to each element
3. return a new vector

We say "functions" because there are 5, the generic `map()` function and `map_*()` variants for each type of vector:

- `map()`
- `map_lgl()`
- `map_int()`
- `map_dbl()`
- `map_chr()`

To illustrate, suppose we have a data frame `df` with 3 columns and we want to compute the mean of each column. We could solve this with copy-and-paste (run `mean()` 3 different times) or try to use a `for` loop, but `map()` can do this with just one line:

```
# We use `map_dbl()` because `mean()` returns a *double* value
map_dbl(df, mean)
```

Modern data frames with tibble

> A tibble, or tbl_df, is a modern reimagining of the data.frame, keeping what time has proven to be effective, and throwing out what is not. Tibbles are data.frames that are lazy and surly: they do less and complain more ~ tibble docs

Tibbles do less than data frames, in a good way:

- never changes type of input (never converts strings to factors!)
- never changes the name of variables
- only recycles vectors of length 1
- never creates row names

You can read more in the tibble vignette if you are interested, but understanding these differences is not necessary to be successful in the course. The take-away is that `data.frame` and `tibble` sometimes behave differently. The behavior of `tibble` makes more sense for modern data science, so we should us it instead!

Create a `tibble` with one of the following:

```
# (1) coerce an existing object (e.g., a data frame) to tibble
as_tibble(x)

# (2) construct a tibble from a column of vectors
tibble(x=1:5, y=1)

# (3) define row-by-row, short for transposed tibble
tribble(
    ~x, ~y, ~z,
    "a", 2, 3.6,
    "b", 1, 8.5
)
```

We will encounter two main ways tibbles and data frames differ:

- **printing** - by default, tibbles print the first 10 rows and all columns that fit on screen, making it easier to work with large datasets. Tibbles also report the type of each column (e.g. `<dbl>`, `<chr>`)

- **subsetting** - tibbles are more strict than data frames, which fixes two quirks we encountered last lecture when subsetting with `[[` and `$`: (1) tibbles *never* do partial matching, and (2) they *always* generate a warning if the column you are trying to extract does not exist.

To test if something is a `tibble` or a `data.frame`:

- `is_tibble(x)`
- `is.data.frame(x)`

Import

Often we want to read in some data we've generated or collected outside of R. The most basic and common format is plain-text **rectangular** files. We will "read" these into R with the `readr` package.

Reading data with readr

> The goal of readr is to provide a fast and friendly way to read rectangular data from delimited files, such as comma-separated values (CSV) and tab-separated values (TSV). It is designed to parse many types of data found in the wild, while providing an informative problem report when parsing leads to unexpected results.
>
> readr docs

We will use `readr`'s `read_*()` functions to read in our rectangular data files.

The `read_*()` functions have two important arguments:

- `file` - the path to the file (that reader will try to parse)
- `col_types` - **column specification**, a description of how each column should be converted from a character vector to a specific data type

There are 7 supported file types, each with their own `read_*()` function:

- `read_csv()`: comma-separated values (CSV)
- `read_tsv()`: tab-separated values (TSV)
- `read_csv2()`: semicolon-separated values
- `read_delim()`: delimited files (CSV and TSV are important special cases)
- `read_fwf()`: fixed-width files
- `read_table()`: whitespace-separated files
- `read_log()`: web log files

To read `.csv` files, include a path and (optionally) a column specification in `col_types`:

```
# (1) pass only the path; readr guesses col_types
read_csv(file='path/to/file.csv')

# (2) include a column specification with col_types
read_csv(
    file='path/to/file.csv',
    col_types = list( x = col_string(), y = col_skip() )
)
```

With no column specification, `readr` uses the the first 1000 rows to guess with a simple heuristic:

- if column contains only T/F, `logical`
- if only numbers, `double`
- if ISO8601 standard, `date` or `date-time`
- otherwise `string`

There are 11 column types that can be specified:

- `col_logical()` - reads as boolean TRUE FALSE values
- `col_integer()` - reads as integer
- `col_double()` - reads as double
- `col_number()` - numeric parser that can ignore non-numbers
- `col_character()` - reads as strings
- `col_factor(levels, ordered = FALSE)` - creates factors
- `col_datetime(format = "")` - creates date-times
- `col_date(format = "")` - creates dates
- `col_time(format = "")` - creates times
- `col_skip()` - skips a column
- `col_guess()` - tries to guess the column

Some useful additional arguments:

- if there is **no header** (the top row containing column names), include `col_names = FALSE`
- to **provide a header**, include `col_names = c("x","y","z")`
- to **skip some lines**, include `skip = n`, where n is number of lines to skip
- to **select which columns** to import, include `col_select(x, y)`
- to **guess** column types **with all rows**, include `guess_max = Inf`

Sometimes weird things happen. The most common problems are:

- **column contains unexpected values** - your dataset has a column that you expected to be `logical` or `double`, but there is a typo somewhere, so R has coerced the column into `character`. Solve by specifying the column type `col_double()` and then using the `problems()` function to see where R failed.
- **missing values are not `NA`** - your dataset has missing values, but they were not coded as `NA` as R expects. Solve by adding an `na` argument (e.g. `na=c("N/A")`)
- **column names have spaces** - your dataset has column names that include spaces, breaking R's naming rules. In these cases, R adds backticks (e.g. `` `brain size` ``); we can use the `rename()` function to fix them. If we have a lot to rename and that gets annoying, see `janitor::clean_names()`.

Reading more complex file types requires functions outside the tidyverse:

- **excel** with `readxl` - see Spreadsheets in R for Data Science
- **google sheets** with `googlesheets4` - see Spreadsheets in R for Data Science
- **databases** with `DBI` - see Databases in R for Data Science
- **json data** with `jsonlite` - see Hierarchical data in R for Data Science

Writing data

We can also write to a csv file with `readr`:

```
write_csv(our_tibble, "name_of_file.csv")
```

We pass the name of our tibble and the name of the file as arguments. In Google Colab, files we write appear in the left side bar file menu.

Data transformation

Why do we need to transform data?

> Visualization is an important tool for generating insight, but it's rare that you get the data in exactly the right form you need to make the graph you want. Often you'll need to create some new variables or summaries to answer your questions with your data, or maybe you just want to rename the variables or reorder the observations to make the data a little easier to work with.

Data transformation with dplyr

All `dplyr` functions (verbs) share a common structure:

- 1st argument is always a data frame
- Subsequent arguments typically describe which columns to operate on (via their names)
- Output is always a new data frame

We can group `dplyr` functions based on what they operate on:

- rows - see section 3 Manipulating rows
- columns - see section 4 Manipulating columns
- groups - see section 5 Grouping and summarizing data frames
- tables - see section 6 Joining data frames

We can easily combine `dplyr` functions to solve complex problems:

- The pipe operator, `|>` takes the output from one function and passes it as input (the first argument) to the next function.
- There is another version of the pipe, `%>%`. See the reading on data transformation if you are curious about the difference.

In lecture, we will demonstrate with the 3 most common ***`dplyr`*** *functions for manipulating rows, manipulating columns, and grouping. But you should feel comfortable reading the docs/resources to use others to solve unique problems.*

Manipulating rows

`filter()` filters rows, allowing you to keep only some rows based on the values of the columns.

- the first argument is a data frame (all dplyr verbs)
- subsequent arguments are the conditions that must be true to keep the row (using R's logical and comparison operators we learned in R basics!), e.g. `filter(age > 18)`
- a common filtering mistake is to use `=` instead of the logical operator `==`!

`arrange()` arranges the rows in the order you specify based on column values (does not change the number of rows, just changes their order)

- the first argument is a data frame (all dplyr verbs)
- subsequent arguments are a set of column names to order by
- note that the default order is ascending, but you can specify descending by wrapping the column in the `desc()` function

`distinct()` finds unique rows in a dataset, but you can also provide column names

- the first argument is a data frame
- optionally subsequent arguments provides column names to find the distinct combination of some variables
- note that if you provide column names, distinct will only return those columns unless you add the argument `.keep_all=TRUE`

Manipulating columns

`mutate()` adds new columns that are calculated from existing columns

- first argument is a data frame (all dplyr verbs)
- subsequent arguments are the new column name, an equals sign, followed by an expression you want to use to calculate the new value, e.g. `difference=age_end - age_start`
- by default new columns are added to the right, but the `.before` and `.after` arguments allows you to add them before/after specific positions (by position number, e.g. `.before=1` or by column name, e.g. `before=age`)

`select()` selects columns based on their names

- first argument is a data frame (all dplyr verbs)
- subsequent arguments can be the names of the columns you want to keep
- use the `:` operator to select everything from one column to another, e.g. `age:height`
- you can also use logical operators like `&` (and) or `!` (not) to identify the subset of columns you want to select, e.g. `!age:height`
- you can also rename columns within select by putting the name of the column and an equals sign before the column you want to select, e.g. `new_name=selected_column`

`rename()` we've already seen this function when importing data. Rename is used when want to keep all of our columns but rename one or more.

- first argument is a data frame (all dplyr verbs)
- subsequent arguments are the columns we would like to rename, e.g. `new_colname=old_colname`

Group and summarise

In addition to manipulating rows and columns in your dataset, `dplyr` also allows you to work with groups

`group_by()` is used to divide your dataset into groups that are meaningful for your analysis.

- `group_by()` doesn't change the data, but adds a groups attribute, which tells R that subsequent operations will be performed by group

- you can tell if a data frame is grouped by the first line of the output (or with `attributes()`!)

`summarise()` is often used after `group_by()` to calculate summary statistics on grouped data, which returns a data frame with a single row for each group

- you can add any number of summary stats; usually you want to name them something that makes sense for your analysis

- `n()` is a particularly useful summary stat to add to our list that returns a count
- use the argument `na.rm=TRUE` to compute the summary statistics with NAs removed (remember they are contageous!)
- note that the returned data frame is iteself grouped, but in a quirky way, with one fewer group (you may get a warning about this). You can add the argument **`.groups="drop"`** to drop all groups or **`.groups="keep"`** to keep them all
- to avoid this quirk, `summarise()` also has a cool new **`.by`** argument that can be used **instead of calling `group_by()`**, which always returns an ungrouped data frame.

`ungroup()` is used to remove the grouping attribute from a data frame

More advanced

There are a few more advanced techniques for transforming and tidying data that we won't cover now, but might be useful to you in your own research.

- **joins** - sometimes you have more than one dataset that you want to join into one. `dplyr` also has functions for handling that. Learn more about joins
- **pivots** - sometimes your data doesn't arrive in the tidy data form. The `tidyr` package can help with `pivot_longer()` and `pivot_wider()`. Learn more about pivots

Further reading and references

Recommended further reading:

- Data tidying in R for Data Science
- Tibbles in R for Data Science
- Data import in R for Data Science
- readr cheatsheet
- Ch 4 Data transformation in R for Data Science textbook

Other references:

- Ch 20 Joins in R for Data Science textbook

- Ch 6 Data tidying in R for Data Science textbook