Data Visualisation Week 9: Advanced data tidying

What's happening today?

Tidying untidy data.

> "Happy families are all alike; every unhappy family is unhappy in its own way."
> — Leo Tolstoy

> "Tidy datasets are all alike, but every messy dataset is messy in its own way."
> — Hadley Wickham

So far we have been working with relatively tidy data. By tidy data, we mean data that follows the rules we talked about in the lecture in week 4:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

However, you will often find data that is "non-tidy" — this doesn't necessarily make it bad, data can be non-tidy for a range of important reasons, for example, in order to save storage space or in order to include a header or footer that contains vital meta-information about the dataset. However, those forms of "non-tidy" data do not work very well with our programming languages, and often you will find data that is "untidy" and requires some tidying up before it can be visualised. This is especially important for when you are finding your own data to complete your second assessment.

Included in this R project is a dataset called `wgidataset_2020.xlsx`. This is a Microsoft Excel file. First, let's **open that file and have a look at the data that's included**. We can already see we might have several challenges:

1. The data is spread across multiple sheets.
2. Each of the sheets with data on has a header on with explanations of the columns
3. Each of the columns is technically the observation for that variable for a specific year, **not** a variable itself
4. The way that data has been recorded as missing is different to how it is coded in R (as NA)

Why don't we just tidy up the spreadsheet itself instead of tidying the data in `R`?

That is a good question. The first thing to say is, if you need to manually tidy up a spreadsheet in order to get the data into a format you need for completing your second assessment **that is completely fine**. I'd rather you do that and hand in something with a visualisation than hand in nothing at all. However, in

general, there are a lot of good reasons to not use this as a way of working for the long term. Some of these include:

1. Tidying data manually does not scale very well - it might be feasible for tidying 20 rows and 10 columns, but what about 200, or 2000, or 2 million rows? 100, 200, 2000 variables? Tidying data using `R` code can be a lot *faster* than doing it manually.
2. Tidying data manually leaves no record of what you did — other people will not be able to see exactly what you did to tidy the data and will be unable to replicate it, you'll also not be able to go back over your work to see if you made any mistakes. Tidying data using `R` code means that all of your tidying steps are documented, replicable, and reliable.

Reading Excel data into `R`

The first thing for us to do is to read the Excel data into `R`. We can do this using the `readxl` package. Remember that if this is your first time using the `readxl` package (which it probably is!) you will need to install it first using `install.packages('readxl')`. Let's load our packages.

```
library(tidyverse)
library(readxl)
```

Okay, now let's see what happens if we just try to read our data into `R` using `read_xlsx()` and see what we get.

```
wgi <- read_xlsx("wgidataset_2020.xlsx")
wgi
```

Okay, well, that's not what we wanted at all. That's because we specifically want the data from one of the sheets (the first sheet is just the cover page). Let's see what happens if we try reading the data, but this time we add sheet = 2 in order to read the second sheet.

```
wgi <- read_xlsx("wgidataset_2020.xlsx", sheet = 2)
wgi
```

That's better, now we are at least reading the Voice and Accountability variable data but you'll notice that if we look at our data by using view(), or clicking on it while holding Ctrl/Cmd, you'll see that we have all of the header information that we don't really want but also we have a lot of data that *should* be missing, but is coming up as a string "#N/A".

Something we can do instead is to specify the range of the data we want to use. We can see the data seems to have 229 rows (when we bear in mind that the first row's been read in as variable names); we can also see the first 14 rows seem to be redundant. So we can specify the range of the data so we only include the stuff we want, and exclude the stuff we don't. We can also set the string "#N/A" to be detected as missing. We can do this like so.

```
wb <- read_excel("wgidataset_2020.xlsx",
                 sheet = 2,
                 range = cell_rows(15:229),
                 na = "#N/A")

wb
```

Renaming, but faster

Okay, now we're kind of in the same position we were last week: we at least have a 'rectangular' set of data with consistent missing values now. We're still not in the position of it being 'tidy': really our observations (countries at specific years) are actually on our columns, rather than variables, whereas ideally they would be on our rows (e.g. we would have Andora 1996, Andora 1997, Andora 1998, and so on).

Let's also look at our variable names:

```
names(wb)
```

Okay, that's not ideal either. We don't have any information on the year, and a lot of repeated variable names with numbers that don't really correspond with the years either.

From here, there are a lot of different tools we could use. We could just manually rename everything like we did last week, but as we can see, this week we have 128 variables to rename instead of only 23. We could also generative AI to make some names for us, but this would be unlikely to work very well because there is so little context in our existing variable names: they have too little information now whereas last week they had too much.

One solution is to start **generalising** our code to rename variables using the same rules, across all of our cases. This means writing one single function to apply to all of our variable names, rather than doing every single one manually.

Let's start with a fairly straightforward one: the first thing we can do is focus on just the variables that contain our country, their code, and then just all of our "Estimate" columns (the average assessment of weak to strong voice and accountability of government). We can do this using the following code:

```
wb <- wb %>%
  select(country = `Country/Territory`,
         code = Code,
         contains("Estimate")
         )

# Take a look at the result
wb

names(wb)
```

So, what is this code doing?

- We are `select`ing the variables, meaning we are choosing just some variables to keep, rather than all of them.

- We are choosing to keep `Country/Territory` and `Code`, but at the same time as choosing to keep them, we are also going to rename them to the much tidier names `country` and `code`.
- Lastly, we are *choosing to select our variables based on a rule, rather than explicitly selecting them*. We are using the `contains` function in order to keep all variables that include the character string "Estimate" in the name.

Okay, so now we have 21 different versions of estimate that all correspond with a specific year in the dataset. If we look at the Excel version of the data we can see that it starts in 1996, and then goes up in two year intervals until 2002 (so, 1996, 1998, 2000, 2002), and then it goes up in single year intervals from 2002 to 2019 (2002, 2003, 2004…).

We could manually rename all of our variables in order to match this, but we could also create a vector (using `c()`) of values that follow that progression of years. If we've done it correctly, we know that it should be 21 values long. To do this, we could use the `seq()` function:

```
first_years <- seq(1996, 2002, 2)
second_years <- seq(2003, 2019, 1)

years <- c(first_years, second_years)

years
length(years)
```

What is this code doing?

- The first part of the code is creating a vector called "first_years", which is a sequence of numbers from 1996 to 2002 that increases by two years at a time.
- The second part is creating a vector called "second_years", which is a sequence of numbers from 2003 to 2019 which increases by one year at a time.
- The third part is combining both of those vectors together into a single vector, called years.
- The last bit of the code prints out the result so that we can check that it looks accurate, and the `length(years)` function tells us how many values are in our vector (21, which is what we wanted).

Now, let's use the `set_names` function that we used last week to set our variable names using this vector:

```
wb <- wb %>%
  set_names(c("country", "code", years))

wb
```

If you followed everything correctly you shouldn't get an error. Now, you might have noticed that we also added some more names to our vector in the end: country and code, because we wanted to keep those names the same. Notice that country and code have "-marks around them, because they need to be specified as strings, but years does not because it was an object we created.

Pivoting

The next stage is to take our observations for each year and put them in a single column; ideally we want to end up with two things:

- one column that states the year of the estimate for accountability
- one column that holds the value for the estimate for accountability

To do this, we could use the `pivot_longer` function:

```
wb <- wb %>%
  pivot_longer(
    cols = c(-country, -code),
    names_to = "year",
    values_to = "accountability"
  )

wb
```

How has our dataset changed? Now, rather than having many columns (making it 'wide'), we have many more rows (4,494, whereas before we only had 213), with repeated entries for the country that, together with the year, make up a unique key.

What does the code do?

- The first argument, cols, tells R which variables we are wanting to transition from wide to long. In this case, it's actually easier for us to tell R which ones we **don't** want to make long — country and code.
- The second argument `names_to`, is us telling R what the variable we are moving our original variables names to should be called.
- The last argument, `values_to`, is telling R what we should call the variable we are adding our values to.

Incidentially, if you wanted to reverse this, you could use the complementary `pivot_wider` function (here is some code if you want to try it, but you don't need to).

```
wb %>%
  pivot_wider(names_from = year,
              values_from = accountability)
```

We have one last thing we might want to do before we start plotting our data. Note that our "year" variable is not a numeric type (continuous) like it should be. Because it was previously a bunch of variable names, R has assumed that it should be a character variable. We might want to change that.

```
wb <- wb %>%
  mutate(
    year = as.numeric(year)
  )
```

Okay, let's draw a graph (finally!)

Drawing a graph of government accountability

Now we've tidied (some) of our data, we can try plotting it. Let's focus on comparing a few countries rather than all of them.

```
wb %>%
  filter(country %in% c("United Kingdom", "United States", "France", "Canada")) %
>%
  ggplot() +
  aes(x = year, y = accountability) +
  geom_line() +
  facet_wrap(~country)
```

Adding more variables - Reusing our code

In practice, most of these steps would be combined together into a single chain of data tidying commands. For example, let's try stringing together all of the above steps into a single, very long, set of `R` code that repeats the process for the "Lower" and "Upper" variables for the estimate (you can, **and should**, copy and paste the relevant bits of code you have been writing and only need to change "Estimate" to "Lower" and "accountability" to "acc_lower"):

```
wb_lower <- read_excel("wgidataset_2020.xlsx",
                       sheet = 2,
                       range = cell_rows(15:229),
                       na = "#N/A") %>%
              select(country = `Country/Territory`,
                     code = Code,
                     contains("Lower")
                     ) %>%
              set_names(c("country", "code", years)) %>%
              pivot_longer(
                cols = c(-country, -code),
                names_to = "year",
                values_to = "acc_lower"
              ) %>%
              mutate(
                year = as.numeric(year)
              )

wb_lower
```

Providing we know that the data is originally in exactly the same shape, once we build up a **workflow** we can duplicate it for adding even more variables quickly, by either copy and pasting the above code and changing only the parts we need to change (the value within `contains()` and the value within `values_to`).

Adding more variables - Generalising our code as a function

If we want to be **even more efficient** we can use a **custom function** for our tidying script. A function is a custom set of R code and commands that act as a template, where you can pass arguments to the function to fill in any gaps that need filling in. So far we've been using other peoples' functions, let's try creating one of our own (don't worry if you don't understand this! Copy-and-pasting multiple times is just fine!)

```
tidy_wb <- function (variable, new_name) {

  tidied_data <- read_excel("wgidataset_2020.xlsx",
                        sheet = 2,
                        range = cell_rows(15:229),
                        na = "#N/A") %>%
              select(country = `Country/Territory`,
                     code = Code,
                     contains(variable)
                     ) %>%
              set_names(c("country", "code", years)) %>%
              pivot_longer(
                cols = c(-country, -code),
                names_to = "year",
                values_to = new_name
              ) %>%
              mutate(
                year = as.numeric(year)
              )

  return(tidied_data)

}
```

If we run this code, nothing will happen except a new function will appear in our environment. However, now we can use this function to tidy our data without having to copy and paste and find which bits of our code we need to change. A few things:

1. We gave our function two arguments, called variable and new_name
2. If we look in our code (within the curly brackets, which state it belongs to the function), we can see that we have changed “Lower” and “lower” to instead be references to our two arguments: variable and new_name.

Now, we can use our function and it will automatically put whatever we write for our arguments into our template code. For example:

```
wb_upper <- tidy_wb(variable = "Upper", new_name = "acc_upper")
wb_rank <- tidy_wb(variable = "Rank", new_name = "acc_rank")
wb_upper
wb_rank
```

Did it work? Have we successfully generalised our data tidying code?

Joining our tidied datasets together

Now we have a few tidied datasets that share a key - the combination of country, code, and year creates a unique key that will help us add the columns from all of our datasets into one single large dataset.

Here is how we can use `left_join` to do that, chaining the result into each dataset with a pipe:

```
wb_combined <- left_join(wb, wb_lower, by = c("country", "code", "year")) %>%
                left_join(wb_upper, by = c("country", "code", "year")) %>%
                left_join(wb_rank, by = c("country", "code", "year"))

wb_combined
```

Note here that because our unique key is made up of the combination of three variables that all have the same name, we put them together separated by commas in the `by` argument.

One last plot, one last geom

Now let's make a plot. One of the last geoms I'm going to show you is `geom_ribbon`, this can be used to show uncertainty through shading. Let's change what we created before slightly:

```
wb_combined %>%
  filter(country %in% c("United Kingdom", "United States", "France", "Canada")) %
>%
  ggplot() +
  aes(x = year, y = acc_rank,
      ymin = acc_lower, ymax = acc_upper) +
  geom_ribbon(fill = "seagreen", alpha = 0.8) +
  geom_line(colour = "white") +
  facet_wrap(~country)
```

Task

Data tidying is a real skill that it takes many many hours of practive to become proficient at. One of the reasons it's so difficult is that every dataset is untidy in its own way, and that means the way to make it tidy won't always be the same for every occassion. However, with a good toolbox of skills — the ability to read data from different sources, the ability to use joins, the ability to reshape data into longer and wider formats, the ability to filter and select data — you can figure out how to tidy up any dataset programmatically with a bit of time. Eventually, tidying untidy data becomes a fun challenge, like a little puzzle to solve. But if you've ever picked up puzzles as a hobby — sudoku, crosswords, and so on — you'll know that it can feel frustrating at the start.

- Draw a graph of the relationship between the rule of law and government effectiveness, faceted by year. Make points more faint if the total number of sources across both measures is lower, and make points more opaque if the total number of sources across both measures is higher.
- Find a dataset online of your choice, and identify a graph you'd ideally like to be able to draw using the data that it contains. What's it like? Can you draw the graph you were hoping to straight away? If not, can you get it into a format that would allow you to do so?