R tidying data and Exploratory Data Analysis

Ghana Workshop TADs

EPIDEMIOLOGY, ECONOMICS AND RISK ASSESSMENT (EERA)

Tidy data

Hopefully if we've entered our own data into something like Excel we can make sure the data are well organised and tidy. By tidy we mean...

1. Each observation is in its own row
2. Each variable is in its own column

Additionally we would like the column names to work well with our statistics software (no crazy symbols or spaces) and to have consistent data in the columns (dates in a standard format etc.)

Here we will use the untidy version of the wedding dataset, **weddingdata_2018b_untidy.xlsx**, to demonstrate some of these principles and how you can use R to tidy a dataset in preparation for analysis. This is made up data from an outbreak investigation at a wedding dinner.

First load the `tidyverse` set of packages (it includes data tidying functions), the `readxl` package (we'll use it to import the data) and other useful packages.

```
library(tidyverse)
library(readxl)
library(here)
library(lubridate)
library(writexl)
library(forcats)
library(janitor)
```

Before you can work on the data you'll need to import it.

```
wedd <- read_excel(here("data", "weddingdata_2018b_untidy.xlsx"))
```

You can have a look at the data by either typing `wedd` in the console to see a small snapshot in the console or by clicking on the `wedd` object name in the environment tab (top right of your screen).

The idea is that every time you have a new dataset you need to first check to make sure there are no obvious errors and that it is in a tidy format to make it more straight forward to analyse.

Let's start with the column names..

Renaming columns

In order to see all the column names type...

```
names(wedd)
```

```
##  [1] "id"                   "name_gender"          "D o b"
##  [4] "how_old_are_you?"     "weight"               "occupation"
##  [7] "not_working"          "at_home_parent"       "chronic_illness"
## [10] "ate_dinner"           "ate_pate"             "ate_prawncocktail"
```

```
## [13] "ate_chicken"           "ate_beef"              "ate_vegetable_tempura"
## [16] "ate_tiramisu"          "postcode"              "place"
## [19] "county"                "region"                "country"
## [22] "recent_travel_country" "recent_travel_region"  "recent_travel"
## [25] "noro"                  "onset_delay_hours"     "onset_of_illness"
## [28] "health_status"         "mode_of_transport"
```

Here we notice that some column names are a bit too long, include spaces and 'special characters'. We really like only letters, numbers and '_' in column names as they work well with the R software.

We can edit the odd column names using the `rename` function. The following code takes the `wedd` dataset, ***then*** pipes it into the `rename` function changing column names (new name on the left old name on the right) and finally puts it back into the `wedd` object. Effectively replacing the old object with the new one with renamed columns. Note that we use back ticks ‘ around the old column names as otherwise R will give an error - they are that bad!

You can also use function `clean_names` from the `janitor` package if you have lot's of columns in strange formats. We have commented it out for now as it stop our code below from working :)

```
wedd <- wedd %>%
  rename(dob = `D o b`,
         age = `how_old_are_you?`)
```

```
#OR
#wedd <- clean_names(wedd)
```

Filling empty columns

We can click on the `wedd` object name in the environment tab again to see if that worked.

Next thing to trouble us is the `region` column. Only the region for the first guest in each region is entered. The blank cells in the Excel spreadsheet are represented as NA in R. This is quite common in datasets, rather lazy and also dangerous as if the rows were jumbled we'd not be able to give guests their correct region. Let's sort it now. The `fill` function will fill down a column until it hits new data. You'll see that in action with this code... Note: Only use this if you are absolutely sure the rows are in the right order!

```
wedd <- wedd %>%
  fill(region)
```

We won't mention it again but do check each time by looking at the `wedd` data after you've run the code. Do note also that these chunks of code are only changing the `wedd` object in R's memory. The Excel spreadsheet is not being changed. That's important and good. It means we are creating a reproducible audit trail of the changes to the untidy data and not destroying the original data. At the end of this session we'll save the cleaned up tidy data with a new name for future use.

Adding a prefix

The `id` column is the guest's ID. It's currently just a number but we worry about that as we might, mistakenly, treat it as a number when really it's just a label. One way to make it safer is to add one or more letters at the start. Let's add an "FS" to it. We use two functions here: `mutate` is the tidyverse way to change a column. We can overwrite existing columns or create new ones from existing ones. We use the `str_c` function to add text to text...

```
wedd <- wedd %>%
  mutate(id = str_c("FS", id))
```

This 'mutates' the `id` column by taking the existing `id` column and putting an "FS" (for food safety) in font of each entry.

Removing duplicates

Sometimes datasets contain the same row more than once, either due to technical glitches or human error. We will use the function `distinct` to keep only unique rows in the dataset. We will use function `get_dupes` and `nrow` to check how many rows are duplicated, if any.

```
get_dupes(wedd) %>% nrow()
```

```
## [1] 0
```

```
wedd <- wedd %>%
  distinct()
```

You can also check specifically for duplicated in IDs. Depending on the type of data, it may be ok to have the same person appearing more than once, e.g. multiple sampling points, but it's important to know how to check.

If you do find duplicated, you can use the function **distinct** to remove them. Note that this time you need to use the argument **.keep_all** otherwise R will drop all other columns.

```
get_dupes(wedd, id) %>% nrow()
```

```
## [1] 0
```

```
wedd <- wedd %>%
  distinct(id, .keep_all = TRUE)
```

Separating a column

Now let's have a look at the `name_gender` column. Let's check what the first few entries look like. The column name suggests trouble!

```
wedd %>%
  select(name_gender) %>%
  slice(1:10)
```

```
## # A tibble: 10 x 1
##    name_gender
##    <chr>
##  1 Raneshia_Female
##  2 Christen_Female
##  3 Jasira_F
##  4 Nayda_F
##  5 Amarri_Female
##  6 Monta_F
##  7 Naima_Female
##  8 Marq_Male
##  9 Finley_M
## 10 Antoniette_Female
```

This appears to hold both the name and the gender of the people. This isn't tidy - each variable should have its own column. We can use the `separate` function to split the column. This function needs lots of arguments - the column to split, names for the new columns and what to split on (the separator). Here it is in action (we'll show a section of the data before and after for you)...

Before:

name_gender	dob	age
Raneshia_Female	1985-11-02	32

name_gender	dob	age
Christen_Female	1984-05-05	34
Jasira_F	1975-08-28	42
Nayda_F	1953-10-13	64
Amarri_Female	1948-04-13	-999

```
wedd <- wedd %>%
  separate(col = name_gender, into = c("name", "gender"), sep = "_")
```

name	gender	dob	age
Raneshia	Female	1985-11-02	32
Christen	Female	1984-05-05	34
Jasira	F	1975-08-28	42
Nayda	F	1953-10-13	64
Amarri	Female	1948-04-13	-999

Sorting out categories

In the `gender` column we have several labels for the same gender. We'd spot this either by eye or when we came to tabulate the data. Let's confirm it here. Since it's categorical, we will just make a quick table of gender. Note that we don't write `wedd <- ...` here as we don't want to overwrite the `wedd` object. Just to print a count summary to the console so we simply pipe `wedd` into the `count` function.

```
wedd %>%
  count(gender)
```

```
## # A tibble: 4 x 2
##   gender     n
##   <chr>  <int>
## 1 F         36
## 2 Female    86
## 3 M         56
## 4 Male      29
```

We can see that we have "F" and "Female" and "M" and "Male". There's lots of ways to convert these to two categories but here we'll use a function from the tidyverse `forcats` package that's designed to manage factors - the R word for categorical data. We use the `fct_recode` function giving it the column name and then pairs with the new value on the left and the old value on the right. As we are changing a column's contents we need to put the code inside a `mutate` function...

```
wedd <- wedd %>%
  mutate(gender = fct_recode(gender,
                             "Female" = "F",
                             "Male" = "M"))
```

We pressed our 'enter' key after the commas there to make the code a bit neater. Long lines can be hard to read and it's good style to spread long functions over several lines. RStudio will automatically indent code to keep it looking nice and lined up.

Let's check it worked...

```
wedd %>%
  count(gender)
```

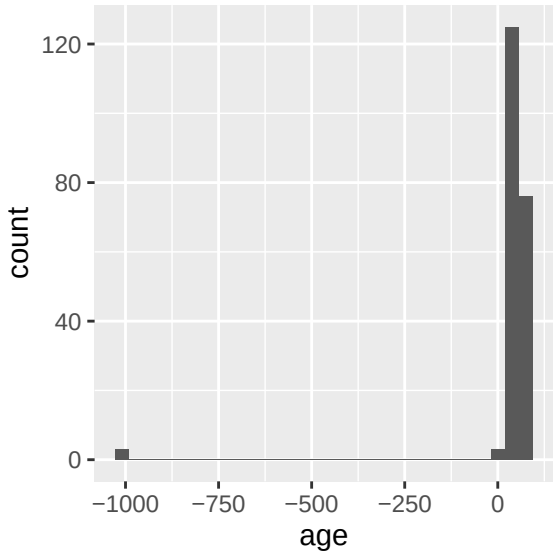
```
## # A tibble: 2 x 2
##   gender     n
##   <fct>  <int>
## 1 Female   122
## 2 Male      85
```

Check numerical variables

Data tidying and cleaning can be hard work. Often you'll spend more time tidying data than analysing it. Especially with data you haven't prepared yourself. But it's a worthwhile investment - tidy data is much easier to plot, analyse and explore than messy data.

Let's have a look at the age column. Since it's numerical, the first thing to do is plot a histogram.

```
ggplot(wedd) +
  geom_histogram(aes(x = age))
```



Hmm.. there seems to be a strange value around -1000. We can use the *filter* function to check what the actual value is.

```
wedd %>%
  filter(age < 0) %>%
  select(id, age)
```

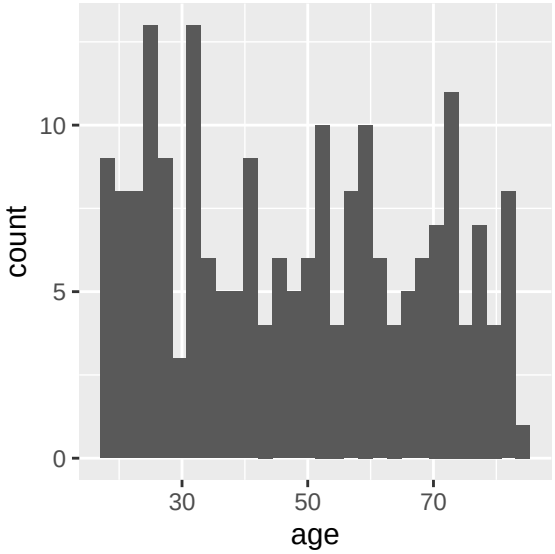
```
## # A tibble: 3 x 2
##   id      age
##   <chr> <dbl>
## 1 FS5    -999
## 2 FS45   -999
## 3 FS100  -999
```

There seem to be three entries of "-999". This doesn't make sense as an age. What investigate this and find out that this is the researcher's code for missing data. We need to turn these values into R's missing data code - NA. That way we can ignore them where appropriate. Otherwise the "-999"s will mess up our averages and plots. We're changing data in a column so we use `mutate`. This time we include the `na_if` function. Given a column name and a value it will replace the entries with that value with NA.

```
wedd <- wedd %>%
  mutate(age = na_if(age, -999))
```

Let's check it worked.

```
ggplot(wedd) +
  geom_histogram(aes(x = age))
```



Don't worry when you try this plot if you get a warning about missing values - that's because there should be some in the `age` column now!

Sorting

We can sort a dataset on one or more columns using the `arrange` function. So to sort the `wedd` dataset on the age of guest we'd use...

```
wedd %>%
  arrange(age)
```

```
## # A tibble: 207 x 30
##    id    name     gender dob                  age weight occupation not_working
##    <chr> <chr>    <fct>  <dttm>              <dbl>  <dbl> <chr>      <chr>
##  1 FS48  Graciano Male   1999-06-09 00:00:00    18   55.8 health pr~ no
##  2 FS56  Kayonna  Female 1999-06-01 00:00:00    18   46.5 care work~ no
##  3 FS119 Namon    Male   1999-12-07 00:00:00    18   54.5 telecommu~ no
##  4 FS35  Rayniel  Male   1999-04-13 00:00:00    19   49.0 hairdress~ yes
##  5 FS91  Auron    Male   1999-01-10 00:00:00    19   55.3 health ca~ no
##  6 FS92  Khyree   Male   1998-08-06 00:00:00    19   55.6 health ca~ no
##  7 FS144 Lindra   Female 1998-06-29 00:00:00    19   50.8 science, ~ no
##  8 FS145 Michael~ Female 1998-10-12 00:00:00    19   46.1 health se~ yes
##  9 FS184 Josalyn  Female 1998-06-17 00:00:00    19   51.9 engineeri~ no
## 10 FS17  Avel     Male   1998-02-27 00:00:00    20   54.7 aircraft ~ no
## # i 197 more rows
## # i 22 more variables: at_home_parent <chr>, chronic_illness <chr>,
## #   ate_dinner <chr>, ate_pate <chr>, ate_prawncocktail <chr>,
## #   ate_chicken <chr>, ate_beef <chr>, ate_vegetable_tempura <chr>,
## #   ate_tiramisu <chr>, postcode <chr>, place <chr>, county <chr>,
## #   region <chr>, country <chr>, recent_travel_country <chr>,
## #   recent_travel_region <chr>, recent_travel <chr>, noro <chr>, ...
```

Have a look at the data to see if it worked.

To sort on descending age use...

```
wedd_sorted <- wedd %>%
  arrange(desc(age))
```

I've put the sorted dataset into a new object (`wedd_sorted`) as I don't want to mess with my original!

You can also sort alphabetically or by date. Try sorting by `name` and by `dob`

Filtering

Sometimes we want to exclude rows. Perhaps we want to make a dataset that only includes old male guests. We can use the `filter` function to do this...

```
wedd_male_60 <- wedd %>%
  filter(age >= 60 & gender == "Male")
```

We're taking the `wedd` data and piping it into the `filter` function. The result gets put into a new object called `wedd_old_male` - I don't want to overwrite the full dataset. You give filter a set of conditions - in this case older or equal to 60 and male. If you want to be really clever you can use R's logical operators (techy stuff). See the filtering section of R 4 Data science for lots of information on this - this is very optional!

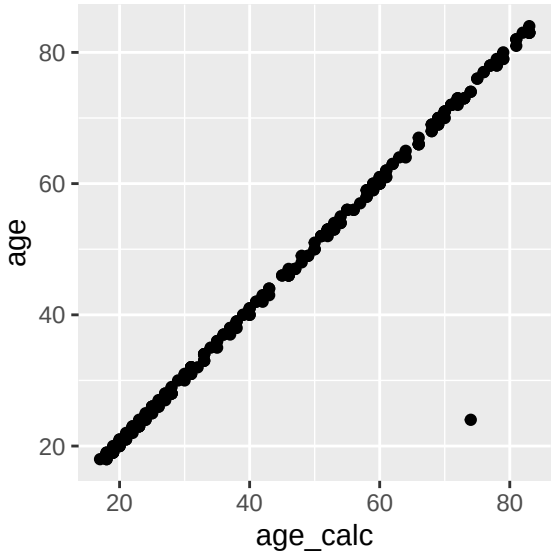
Checking for errors using multiple columns

We have data on both date of birth and age so we can check if there are any typos in ages, by calculating the age. The dataset was collected on the 27th October 2017. We will use function `ymd` to create date objects and function `interval` to calculate the interval between their dob and the date of the questionnaire. We will then divide that by years to get their age! We will save the output into a new column called `age_cal`. All these handy functions come from the `lubridate` package. Check it out if you want to learn more about how to manipulate date data in R.

```
wedd <- wedd %>%
  mutate(age_calc = interval(ymd(dob), ymd("2017/10/27")) %/% years(1))
```

A quick way to check for any obvious mistakes is by plotting the two new columns

```
ggplot(wedd) +
  geom_point(aes(x = age_calc, y = age))
```



Hmmmm there is an age of around 75 that has been entered as ~25. Let's find which ID it is and correct it.

```
wedd %>%
  filter(age_calc-age >2) %>%
  select(id, age, age_calc)
```

```
## # A tibble: 1 x 3
##   id      age age_calc
##   <chr> <dbl>    <dbl>
## 1 FS182    24       74
```

So id FS182, had the age swapped from 74 to 24! This looks like a typo! It's tempting to go back to Excel and make the change but there's no record of that so it's not best practice. Much better is to import the data as-is and then write lines of R code to edit the data. Let's do that!

We're changing a column so we use `mutate` and we also use the `case_when` function of the tidyverse. It lets us set a condition, in this case the guest ID being "FS182" and return a different value for a column in that instance. Here's how...

```
wedd <- wedd %>%
  mutate(age = case_when(id == "FS182" ~ 74,
                         TRUE ~ age))
```

We're overwriting age with the output of the `case_when` function. If the test `id == "FS182"` is TRUE it will give us 74 otherwise it gives us what's in `age` anyway. The code is a bit complex but if you need to make changes in your data you just modify the column names and the condition test. Ask if you need to do this with your data and we can show you the way!

Let's check it worked!

```
wedd %>%
  filter(id == "FS182") %>%
  select(id:age) %>%
  knitr::kable()
```

id	name	gender	dob	age
FS182	Rudolfo	Male	1943-05-19	74

Pivot data from wide to long

Data can be presented in a wide or long format.

Wide data: Each row in a wide dataset contains all the data for a single observation. The values in the ID column do not repeat. The wedding dataset is in a wide data format.

Long data: Each ID in long data can have multiple rows. You may need this if you want to perform analysis or visualisation with tools that expect long-format data. For example ggplot2 or dplyr.

We will use the function `pivot_longer` to pivot our dataset from wide to long. We want to do this to calculate and plot percentages of people who consumed each different food. We will tell R that the columns we want to collate are all the columns referring to food types and we want that information to go to new column `Food_type`, while the values of each column should go to column `Consumed`.

```
wedd_long <- wedd %>%
  filter(ate_dinner == "yes") %>%
  pivot_longer(cols = ate_pate:ate_tiramisu, names_to = "Food_type", values_to = "Consumed")
```

Let's see...

```
wedd_long %>%
  select(id, age, Food_type, Consumed) %>%
  slice(1:12)
```

```
## # A tibble: 12 x 4
##    id      age Food_type             Consumed
##    <chr> <dbl> <chr>                 <chr>
##  1 FS1      32 ate_pate              yes
##  2 FS1      32 ate_prawncocktail     no
##  3 FS1      32 ate_chicken           no
##  4 FS1      32 ate_beef              no
##  5 FS1      32 ate_vegetable_tempura yes
##  6 FS1      32 ate_tiramisu          no
##  7 FS2      34 ate_pate              yes
##  8 FS2      34 ate_prawncocktail     no
##  9 FS2      34 ate_chicken           yes
## 10 FS2      34 ate_beef              no
## 11 FS2      34 ate_vegetable_tempura no
## 12 FS2      34 ate_tiramisu          yes
```

As you see in the long format of our data, each id appears as many times as there are different types of food. Let's calculate the percentage of people who consumed each food. We will count how many people consumed each food, then we will group the data by food type, to be able to calculate percentages for each food type. Lastly, we will round the Percentages so that they look good on the table!

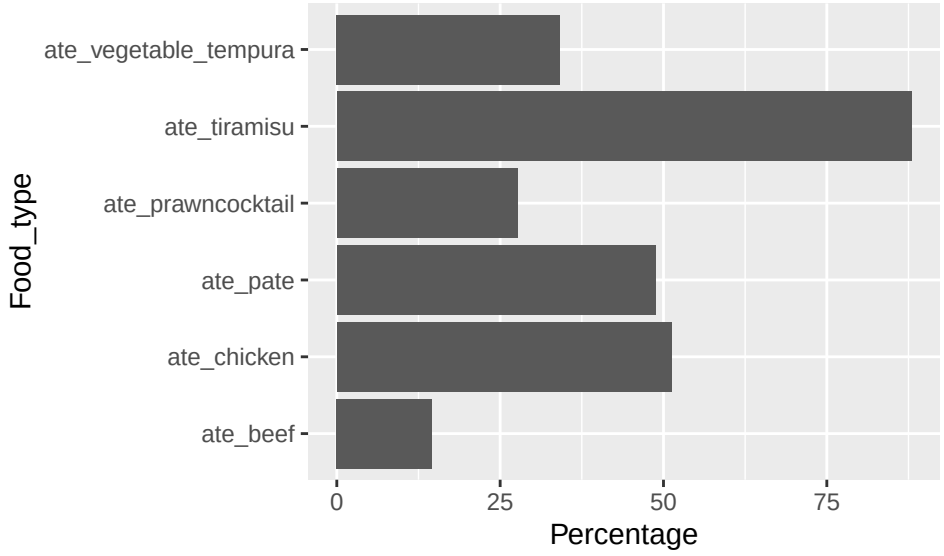
```
wedd_long %>%
  count(Food_type, Consumed) %>%
  group_by(Food_type) %>%
  mutate(Percentage = n / sum(n) * 100) %>%
  mutate(Percentage = round(Percentage))
```

```
## # A tibble: 12 x 4
## # Groups:   Food_type [6]
##    Food_type             Consumed     n Percentage
##    <chr>                 <chr>    <int>      <dbl>
##  1 ate_beef              no         170         85
##  2 ate_beef              yes         29         15
##  3 ate_chicken           no          97         49
##  4 ate_chicken           yes        102         51
##  5 ate_pate              no         102         51
##  6 ate_pate              yes         97         49
##  7 ate_prawncocktail     no         144         72
##  8 ate_prawncocktail     yes         55         28
##  9 ate_tiramisu          no          24         12
## 10 ate_tiramisu          yes        175         88
## 11 ate_vegetable_tempura no         131         66
## 12 ate_vegetable_tempura yes         68         34
```

Now let's plot this! We will pipe the result straight into ggplot, use the `geom_col` function to make a bar chart and use the `coord_flip()` function to make it horizontal to make it easier to read!

```
wedd_long %>%
  count(Food_type, Consumed) %>%
  group_by(Food_type) %>%
  mutate(Percentage = n / sum(n) * 100) %>%
  filter(Consumed == "yes") %>%
```

```
ggplot() +
geom_col(aes(x = Food_type, y = Percentage)) +
coord_flip()
```



Saving your tidy data

Last job!

After all this work we'd like to save a tidy and clean version of the data. We can save it in several formats but comma-separated-values make sense (csv). csv files are simple, can be read by R, Excel and lots of other programs and are also human-readable - if you open one in a simple text editor you can see what it contains. To save the tidied up `wedd` data we can use the `write_csv` function. It's part of the tidyverse set of functions and better than R's built in csv writing function. You simply give it the name of the R data frame you want to save and where to save it...

First create a new folder called `data_clean` where you can store any new, clean and tidy versions of your dataset..

```
write_csv(wedd, here("data_clean", "weddingdata_todaysdate_tidy.csv"))
```

If you prefer .xlxs here is the code, although we recommend using a csv file lilke in the previous command! (CSV = comma separated values, the advantage of a csv file is that it is a plain text file and can be read by any computer program!)

```
write_xlsx(wedd, here("data_clean", "weddingdata_todaysdate_tidy.xlsx"))
```

You should see two new files in your `data_clean` folder.

To import this data if you wanted to work on it without having to run all the cleaning code you can read it in using the import tools in the environment tab (top right-ish of RStudio). You'd choose 'import dataset' then 'From text (readr)'. Alternatively the following code would read in the file and put it into an object called `tidied_wedd`...

```
tidied_wedd <- read_csv(here("data_clean", "weddingdata_todaysdate_tidy.csv"))
```

OR..

```
tidied_wedd <- read_excel(here("data_clean", "weddingdata_todaysdate_tidy.xlsx"))
```

That's it. For way more information on tidying data (and more) have a look at R for Data Science by Hadley Wickham and Garrett Grolemund. It's very well written and it has a lot of information!

Exercises

Ex 1.

Make a dataset that contains only female guests who ate pate.

Ex 2.

Correct the weight of guest with ID FS135 to 65.55 kg.

Ex 3.

Round up the weight column to 1 decimal place and save results in a new column called weight_round.

Ex 4.

Change the name of the variable noro to norovirus.

Ex 5.

Recode the gender column to female and male.

For future reference: A lot of the pain of having to tidy data in R can be avoided if you start by recording your data in a well organised format. If you are lucky enough to be the person designing the data entry sheet you can control this. If you have someone else's data you just need to go ahead and sort it in R (don't try and tidy it in Excel!). For a great reference on how to store data in Excel read Data Organisation in Spreadsheets by Broman and Woo.