Introduction to R

Will Mackey

15/01/2019

Reading this document

This document was created in `R` using `Rmarkdown`. Text, like this, looks like this.

Code chunks that you can copy-and-paste into your script is in a box with a light-grey background.

The **output** from that code will be displayed underneath in a box with a white background. The numbers in square brackets e.g. `[1]` in the ouput mean only the line of the output, and hold no real meaning.

```
# this is a chunk of code you can run (but this line is a comment that doesn't do anything)

2 + 3
```

```
## [1] 5
```

```
ten <- 10
ten * 16
```

```
## [1] 160
```

```
hello <- "hello"
paste0(hello, " Federica")
```

```
## [1] "hello Federica"
```

```
vector <- c(10, 20, 42)
vector * 2
```

```
## [1] 20 40 84
```

Packages

'Packages' are a collection of functions with—*usually*—clear documentation. The people who make them are great and should feel good about themselves. Like any application, you need to *install* packages and then open them when you want to use it. In `R`, we install (once only) using `install.packages("thePackageName")`:

```
install.packages("tidyverse")
install.packages("zoo")
```

(Note that tidyverse is a collection of packages rather than a single package itself).

We then load packages each time we start a new `R` session with `library(thePackageName)`:

```
library(tidyverse)
library(zoo)
```

Reading messy data

We will jump right into it by reading an ABS dataset generated by 2016 Census TableBuilder. The file is called `vet_participation.csv`. Read it into `R` using the `read_csv` function:

```
vet_participation <- read_csv("data/vet_participation.csv")

head(vet_participation)
```

```
## # A tibble: 6 x 1
##   `Australian Bureau of Statistics`
##   <chr>
## 1 2016 Census - Counting Persons, Place of Usual Residence (MB)
## 2 AGE5P - Age in Five Year Groups, SEXP Sex, TYPP Type of Educational Inst~
## 3 Counting: Persons Place of Usual Residence
## 4 Filters:
## 5 Default Summation
## 6 AGE5P - Age in Five Year Groups
```

Using the `head` function above, we see that there is one variable (column) in our dataset. This doesn't look right. There is, annoyingly, a 10 rows of *cruft* included by the ABS at the top of the csv file (although this number will **vary**). We can skip the first 10 rows by using the `skip` argument:

```
vet_participation <- read_csv("data/vet_participation.csv",
                              skip = 10)

head(vet_participation)
```

```
## # A tibble: 6 x 6
##   `AGE5P - Age in~ `SEXP Sex` `TYPP Type of E~ `IRSAD Deciles ~ `RA (UR)`
##   <chr>            <chr>      <chr>            <chr>            <chr>
## 1 0-4 years        Male       Preschool        Decile 1         Major Ci~
## 2 <NA>             <NA>       <NA>             <NA>             Inner Re~
## 3 <NA>             <NA>       <NA>             <NA>             Outer Re~
## 4 <NA>             <NA>       <NA>             <NA>             Remote A~
## 5 <NA>             <NA>       <NA>             <NA>             Very Rem~
## 6 <NA>             <NA>       <NA>             <NA>             Migrator~
## # ... with 1 more variable: X6 <dbl>
```

Fixed at the top. Use `View(vet_participation)` to check the bottom of the dataset and see that there is also cruft there. We see that rows with `NA` values for X6 are rubbish, so we can filter them out using `filter` combined with `is.na(x)`, which returns a logical if `x` is `NA`, combined with the negator operator `!` (can be read as 'not', turning `TRUE` to `FALSE` and vice-versa). The filter portion of the chunk below reads 'keep all rows in which `X6` is **not** `NA`'.

```
old_number_rows <- nrow(vet_participation)

vet_participation <- read_csv("data/vet_participation.csv",
                              skip = 10) %>%
                     filter(!is.na(X6))

new_number_rows <- nrow(vet_participation)

old_number_rows - new_number_rows
```

```
## [1] 4
```

```
head(vet_participation)
```

```
## # A tibble: 6 x 6
##   `AGE5P - Age in~ `SEXP Sex` `TYPP Type of E~ `IRSAD Deciles ~ `RA (UR)`
##   <chr>            <chr>      <chr>            <chr>            <chr>
## 1 0-4 years        Male       Preschool        Decile 1         Major Ci~
## 2 <NA>             <NA>       <NA>             <NA>             Inner Re~
## 3 <NA>             <NA>       <NA>             <NA>             Outer Re~
## 4 <NA>             <NA>       <NA>             <NA>             Remote A~
## 5 <NA>             <NA>       <NA>             <NA>             Very Rem~
## 6 <NA>             <NA>       <NA>             <NA>             Migrator~
## # ... with 1 more variable: X6 <dbl>
```

Great. We dropped 4 at the bottom.

`head` shows us that the values for the variables aren't filled in all the way down. We get a bunch of `NA`s instead. The very handy `zoo::na.locf` function can come to the rescue here (the notation there is `package::function`, and can be used in your code to explicitly call a function from a particular package. This is sometimes needed if there is more than one function with the same name). We can ask for more details about a function—what it does; what its arguments and output are; some examples—with `?na.locf`.

```
vet_participation <- read_csv("data/vet_participation.csv",
                              skip = 10)  %>%
                     filter(!is.na(X6)) %>%
                     na.locf()

head(vet_participation)
```

```
## # A tibble: 6 x 6
##   `AGE5P - Age in~ `SEXP Sex` `TYPP Type of E~ `IRSAD Deciles ~ `RA (UR)`
##   <chr>            <chr>      <chr>            <chr>            <chr>
## 1 0-4 years        Male       Preschool        Decile 1         Major Ci~
## 2 0-4 years        Male       Preschool        Decile 1         Inner Re~
## 3 0-4 years        Male       Preschool        Decile 1         Outer Re~
## 4 0-4 years        Male       Preschool        Decile 1         Remote A~
## 5 0-4 years        Male       Preschool        Decile 1         Very Rem~
## 6 0-4 years        Male       Preschool        Decile 1         Migrator~
## # ... with 1 more variable: X6 <dbl>
```

Fixed!

Renaming and data coding

Now we should manipulate the dataset to create clear, no-space variable names using `rename`. First use `names` to produce a vector of current variable names in the dataset. Then use `rename` to rename the variables.

```
names(vet_participation)
```

```
## [1] "AGE5P - Age in Five Year Groups"
## [2] "SEXP Sex"
## [3] "TYPP Type of Educational Institution Attending"
## [4] "IRSAD Deciles at SA1 Level (Area)"
## [5] "RA (UR)"
## [6] "X6"
```

```
vet_participation <- vet_participation %>%
                     rename(
                       age = "AGE5P - Age in Five Year Groups",
                       sex = "SEXP Sex",
                       institution = "TYPP Type of Educational Institution Attending",
                       irsad_dec = "IRSAD Deciles at SA1 Level (Area)",
                       rural = "RA (UR)",
                       n = X6
                     )

names(vet_participation)
```

```
## [1] "age"         "sex"         "institution" "irsad_dec"   "rural"
## [6] "n"
```

Now we want to look at how things are coded. We can use the `unique` function to retrieve all unique values of a given vector combined with the `pull` function to pull a vector out of the dataset. The first line here reads 'define `age_unique` as all `unique` values in the `age` vector pulled from the `vet_participation` dataset. When we define a variable there is no output, so we call the variable on the next line to show the results. Alternatively, this can be done automatically by surrounding the whole line in brackets:

```
age_unique <- vet_participation %>% pull(age) %>% unique()
age_unique
```

```
##  [1] "0-4 years"          "5-9 years"          "10-14 years"
##  [4] "15-19 years"        "20-24 years"        "25-29 years"
##  [7] "30-34 years"        "35-39 years"        "40-44 years"
## [10] "45-49 years"        "50-54 years"        "55-59 years"
## [13] "60-64 years"        "65-69 years"        "70-74 years"
## [16] "75-79 years"        "80-84 years"        "85-89 years"
## [19] "90-94 years"        "95-99 years"        "100 years and over"
## [22] "Total"
```

```
(sex_unique <- vet_participation %>% pull(sex) %>% unique())
```

```
## [1] "Male"   "Female"
```

```
(institution_unique <- vet_participation %>% pull(institution) %>% unique())
```

```
##  [1] "Preschool"
##  [2] "Infants/Primary - Government"
##  [3] "Infants/Primary - Catholic"
##  [4] "Infants/Primary - Other Non Government"
##  [5] "Secondary - Government"
##  [6] "Secondary - Catholic"
##  [7] "Secondary - Other Non Government"
##  [8] "Technical or Further Educational Institution (including TAFE Colleges)"
##  [9] "University or other Tertiary Institution"
## [10] "Other"
## [11] "Not stated"
## [12] "Not applicable"
```

```
(irsad_dec_unique <- vet_participation %>% pull(irsad_dec) %>% unique())
```

```
##  [1] "Decile 1"       "Decile 2"       "Decile 3"       "Decile 4"
##  [5] "Decile 5"       "Decile 6"       "Decile 7"       "Decile 8"
##  [9] "Decile 9"       "Decile 10"      "Not applicable"
```

```
(rural_unique <- vet_participation %>% pull(rural) %>% unique())
```

```
##  [1] "Major Cities of Australia (NSW)"
##  [2] "Inner Regional Australia (NSW)"
##  [3] "Outer Regional Australia (NSW)"
##  [4] "Remote Australia (NSW)"
##  [5] "Very Remote Australia (NSW)"
##  [6] "Migratory - Offshore - Shipping (NSW)"
##  [7] "No usual address (NSW)"
##  [8] "Major Cities of Australia (Vic.)"
##  [9] "Inner Regional Australia (Vic.)"
## [10] "Outer Regional Australia (Vic.)"
## [11] "Remote Australia (Vic.)"
## [12] "Migratory - Offshore - Shipping (Vic.)"
## [13] "No usual address (Vic.)"
## [14] "Major Cities of Australia (Qld)"
## [15] "Inner Regional Australia (Qld)"
## [16] "Outer Regional Australia (Qld)"
## [17] "Remote Australia (Qld)"
## [18] "Very Remote Australia (Qld)"
## [19] "Migratory - Offshore - Shipping (Qld)"
## [20] "No usual address (Qld)"
## [21] "Major Cities of Australia (SA)"
## [22] "Inner Regional Australia (SA)"
## [23] "Outer Regional Australia (SA)"
## [24] "Remote Australia (SA)"
## [25] "Very Remote Australia (SA)"
## [26] "Migratory - Offshore - Shipping (SA)"
## [27] "No usual address (SA)"
## [28] "Major Cities of Australia (WA)"
## [29] "Inner Regional Australia (WA)"
## [30] "Outer Regional Australia (WA)"
## [31] "Remote Australia (WA)"
## [32] "Very Remote Australia (WA)"
## [33] "Migratory - Offshore - Shipping (WA)"
## [34] "No usual address (WA)"
## [35] "Inner Regional Australia (Tas.)"
## [36] "Outer Regional Australia (Tas.)"
## [37] "Remote Australia (Tas.)"
## [38] "Very Remote Australia (Tas.)"
## [39] "Migratory - Offshore - Shipping (Tas.)"
## [40] "No usual address (Tas.)"
## [41] "Outer Regional Australia (NT)"
## [42] "Remote Australia (NT)"
## [43] "Very Remote Australia (NT)"
## [44] "Migratory - Offshore - Shipping (NT)"
## [45] "No usual address (NT)"
## [46] "Major Cities of Australia (ACT)"
## [47] "Inner Regional Australia (ACT)"
## [48] "Migratory - Offshore - Shipping (ACT)"
## [49] "No usual address (ACT)"
## [50] "Inner Regional Australia (OT)"
## [51] "Very Remote Australia (OT)"
## [52] "Migratory - Offshore - Shipping (OT)"
```

```
## [53] "No usual address (OT)"
```

(Sidenote: hey we repeated ourselves a lot there – maybe we should write a function that pulls out a vector of unique values for a given variable? We'll get to that later.)

Creating new variables

Cool. Looking at `institution_unique` we see 12 unique values. But we don't need this level of detail. Instead, we can group them into useful categories: `school`, `vet`, `university` and `other/na`. First, define a vector that contains all the detailed `institution_unique` for each group. Create a vector using the `c` function (i.e. combine).

```
university <- c("University or other Tertiary Institution")

vet <- c("Technical or Further Educational Institution (including TAFE Colleges)")
```

Writing out the unique values can be tedious, so we can just refer to elements of the `institution_unique` vector directly:

```
(school <- institution_unique[1:7])
```

```
## [1] "Preschool"
## [2] "Infants/Primary - Government"
## [3] "Infants/Primary - Catholic"
## [4] "Infants/Primary - Other Non Government"
## [5] "Secondary - Government"
## [6] "Secondary - Catholic"
## [7] "Secondary - Other Non Government"
```

```
(other_na <- institution_unique[10:12])
```

```
## [1] "Other"          "Not stated"     "Not applicable"
```

With our groups defined, we can create a VET dummy variable in the `vet_participation` dataset using `mutate`. We will employ the function `%in%` which means 'is contained within' and returns a logical. In this example: for each row, if the `institution` variable is contained within the `vet` vector we defined before, the `is_vet` variable will be `TRUE`; otherwise it will be `FALSE`.

```
vet_participation <- vet_participation %>%
                     mutate(is_vet = institution %in% vet)
```

We also want to add a variable `inst_group` which takes the value of `"vet"`, `"university"`, `"school"` or `"other"`. One way to do this would be nested `if_else` functions, like:

```
vet_participation <- vet_participation %>%
                     mutate(inst_group =
                              if_else(inst_group %in% school, "school",
                                      if_else(inst_group %in% vet, "vet",
                                              if_else.......))) etc
```

Luckily, there is a `case_when` function that will save us from coding as if we're using Excel. The `case_when` function works like this and is a *delight* to read:

```
vet_participation <- vet_participation %>%
                     mutate(inst_group = case_when(
                       institution %in% school     ~ "school",
                       institution %in% vet        ~ "vet",
                       institution %in% university ~ "university",
```

```
                      institution %in% other_na   ~ "other")
                    )

head(vet_participation)
```

```
## # A tibble: 6 x 8
##   age     sex   institution irsad_dec rural             n is_vet inst_group
##   <chr>   <chr> <chr>       <chr>     <chr>         <dbl> <lgl>  <chr>
## 1 0-4 ye~ Male  Preschool   Decile 1  Major Cities~  2371 FALSE  school
## 2 0-4 ye~ Male  Preschool   Decile 1  Inner Region~  1463 FALSE  school
## 3 0-4 ye~ Male  Preschool   Decile 1  Outer Region~   669 FALSE  school
## 4 0-4 ye~ Male  Preschool   Decile 1  Remote Austr~    60 FALSE  school
## 5 0-4 ye~ Male  Preschool   Decile 1  Very Remote ~    24 FALSE  school
## 6 0-4 ye~ Male  Preschool   Decile 1  Migratory - ~     0 FALSE  school
```

Summarising data

Wonderful. Now we have a question: how many young people (15-24yo) are in each broad institution group? We have a bunch of variables we don't need here: `sex`, `institution`, `age`, `irsad_dec`, `rural` and, now, `is_vet`. We can summarise the dataset to answer this question. The `summarise` function collapses a dataset into a single row and returns names summary statistics. For example:

```
vet_participation %>%
  summarise(sum_n = sum(n),
            count_rows = n(),
            mean_n = mean(n),
            you_can_name_it_whatever_you_want = median(n))
```

```
## # A tibble: 1 x 4
##      sum_n count_rows mean_n you_can_name_it_whatever_you_want
##      <dbl>      <int>  <dbl>                              <dbl>
## 1 46794499     307824   152.                                  0
```

That returned four statistics—a count (using `n()`), the `sum`, the `mean` and the `median`—for the dataset. Kind of useful, but far more powerful when combined with `group_by`. The `group_by` variable creates (in the backround) separate datasets for each unique group. The `summarise` function will then return the statistics for each unique group.

Our question is about `inst_group`, so we group by those variables before summarising:

```
inst_young_sum <- vet_participation %>%
  filter(age == "15-19 years" | age == "20-24 years") %>%
  group_by(inst_group) %>%
  summarise(sum_n = sum(n),
            count_rows = n(),
            mean_n = mean(n),
            median_n = median(n))

inst_young_sum
```

```
## # A tibble: 4 x 5
##   inst_group   sum_n count_rows mean_n median_n
##   <chr>        <dbl>      <int>  <dbl>    <dbl>
## 1 other      1353482       6996  193.         0
## 2 school      771368      16324   47.3        0
```

```
## 3 university  683770       2332  293.         0
## 4 vet         178868       2332   76.7        0
```

Beaut.

Creating visuals

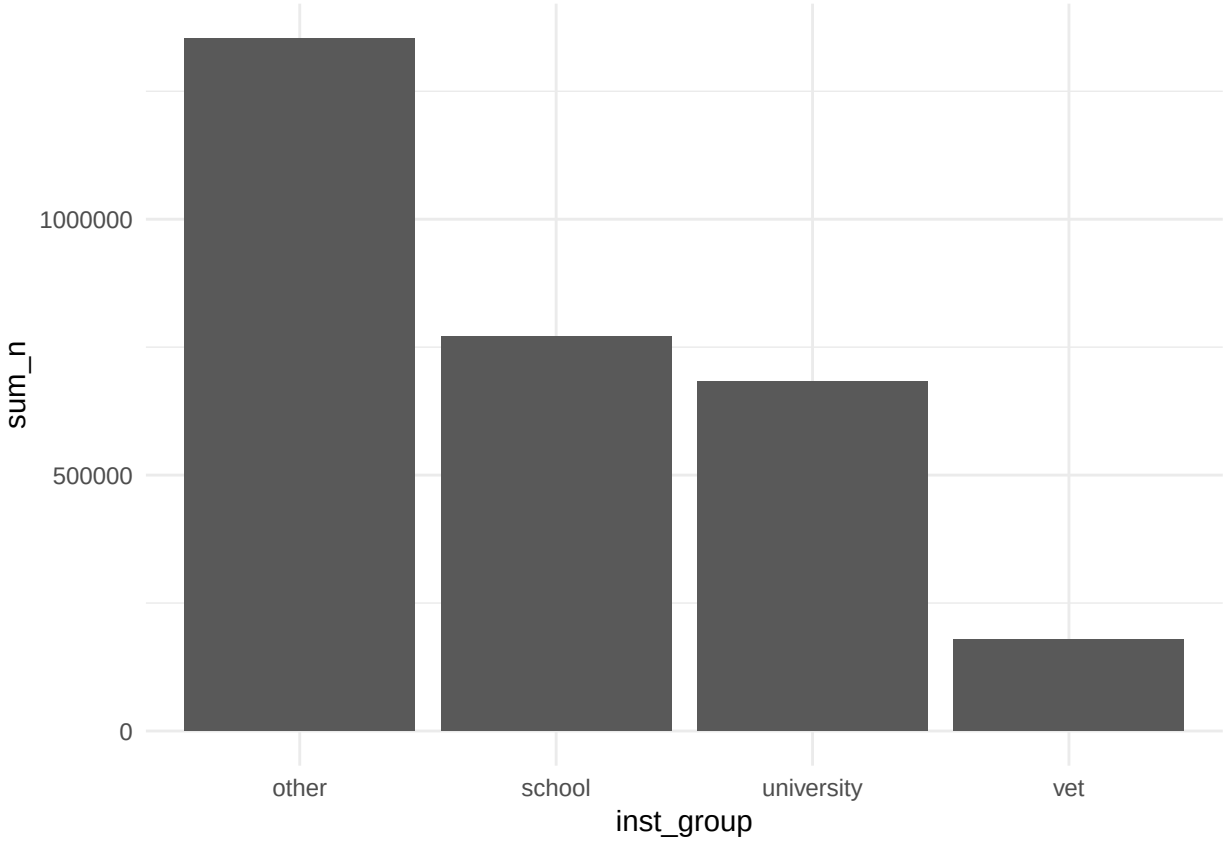
There's too many unique groups in `inst_young_sum` now to quickly evaluate. Let's ~data viz~ it using `ggplot`. `ggplot` is based on the idea that most visualisations can be created using data, aesthetics and geoms. I'll jump right in here, but Chapter 3 of Hadley Wickham's *R for Data Science* explains it all really well.

So:

- data: we'll use our summary dataset `age_inst`.
- aesthetics: we want to plot the total number of people in each institutition group by age. Our `y` variable is `sum_n`; our `x` variable is `inst_group`.
- geom: let's plot a bar chart using `geom_bar`. Note that we have calculated our statistic `sum_n` already. So we'll need to tell `geom_bar` that using the argument `stat = "identity"` (aka "just plot the number and don't do anything else to it").

This all looks like:

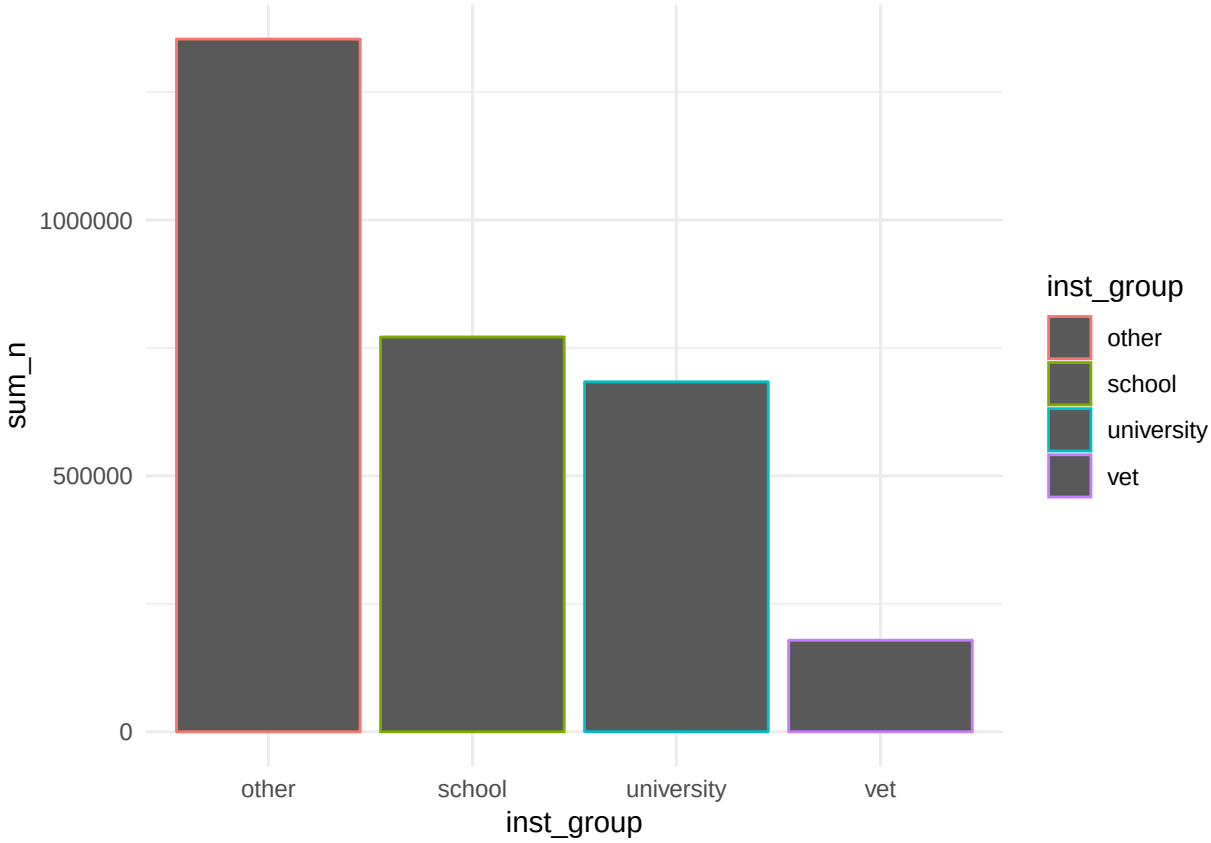
```
inst_young_sum %>%
  ggplot(aes(x = inst_group,
             y = sum_n)) +
  geom_bar(stat = "identity")
```



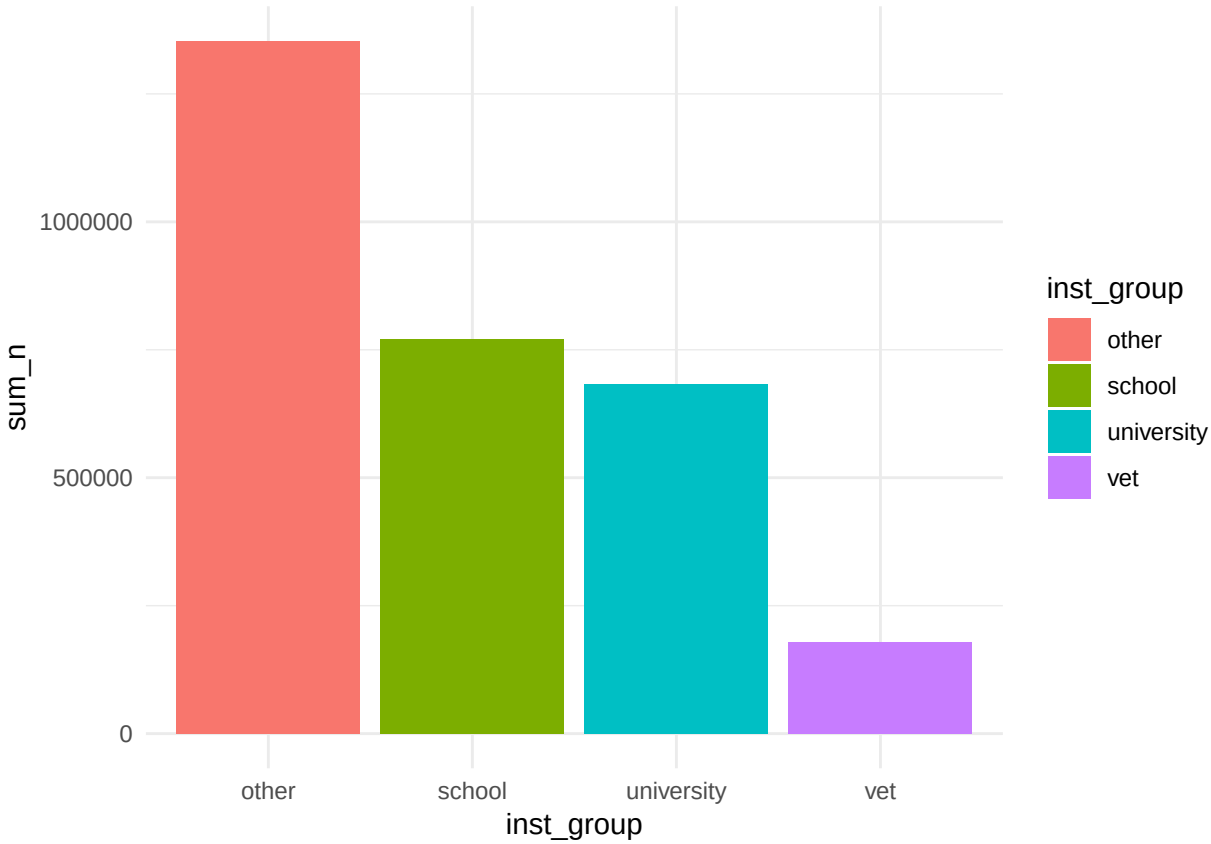
Nice. But it's all a bit **bland**. Let's liven it up with some COLOUR.

We can map `inst_group` to the `colour` aesthetic (note the difference between `colour` and `fill`). **ALSO** note that because Hadley is from NZ, we're allowed to spell colour correctly (. . . `color` also works).

```
# 1
inst_young_sum %>%
  ggplot(aes(x = inst_group,
             y = sum_n,
             colour = inst_group)) +
  geom_bar(stat = "identity")
```

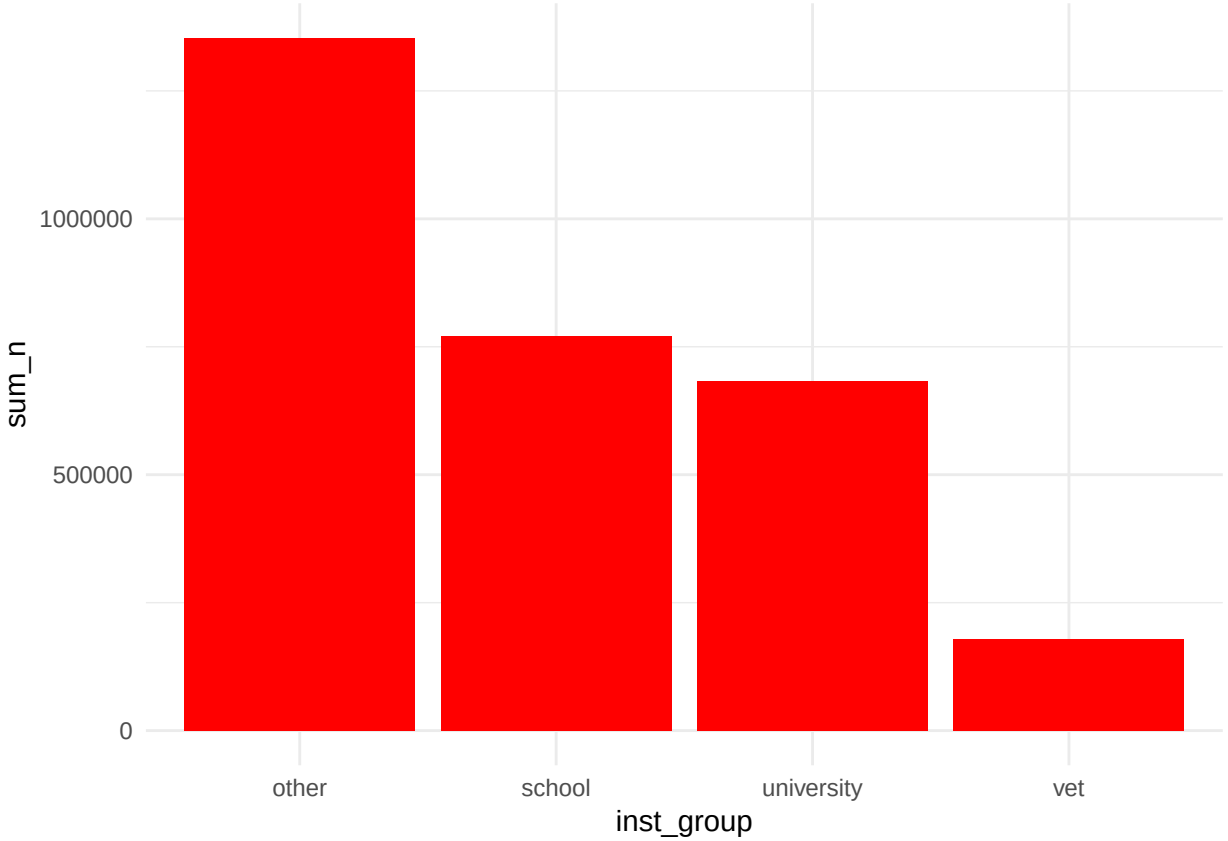



```
# 2
inst_young_sum %>%
  ggplot(aes(x = inst_group,
             y = sum_n,
             fill = inst_group)) +
  geom_bar(stat = "identity")
```



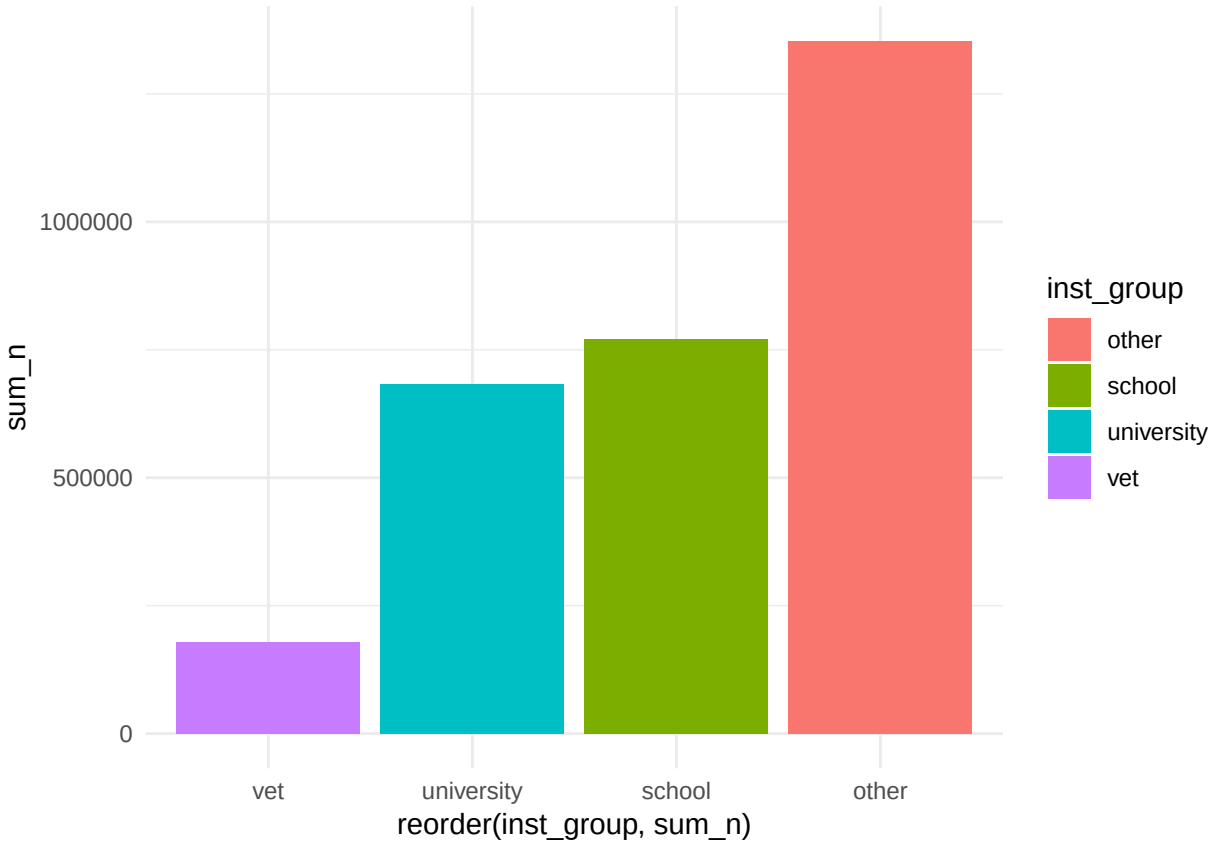
Above we have 'mapped' colour to a variable, so it will change according to the variable. We can alternatively just set a colour for the `geom` by using `fill` or `colour` *outside of the* `aes` *function*:

```
inst_young_sum %>%
  ggplot(aes(x = inst_group,
             y = sum_n)) +
  geom_bar(stat = "identity",
           fill = "red")
```



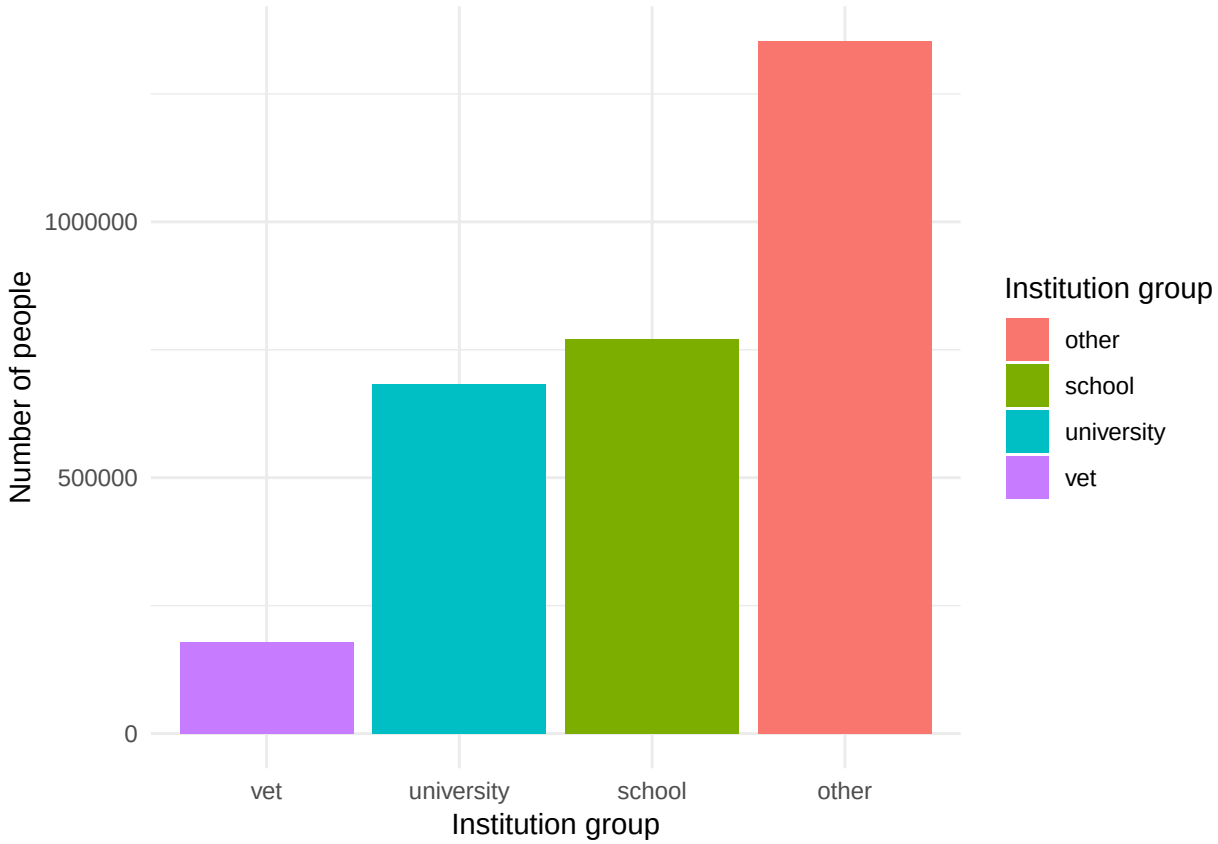
Woah that red is gross. And the order of our `x`-axis variables are all out-of-whack. `ggplot` has been supplied a set of `character` values in our `inst_group` variable that it has ordered alphabetically. We could `reorder` it to go from lowest `sum_n` to highest:

```
inst_young_sum %>%
  ggplot(aes(x = reorder(inst_group, sum_n),
             y = sum_n,
             fill = inst_group)) +
  geom_bar(stat = "identity")
```



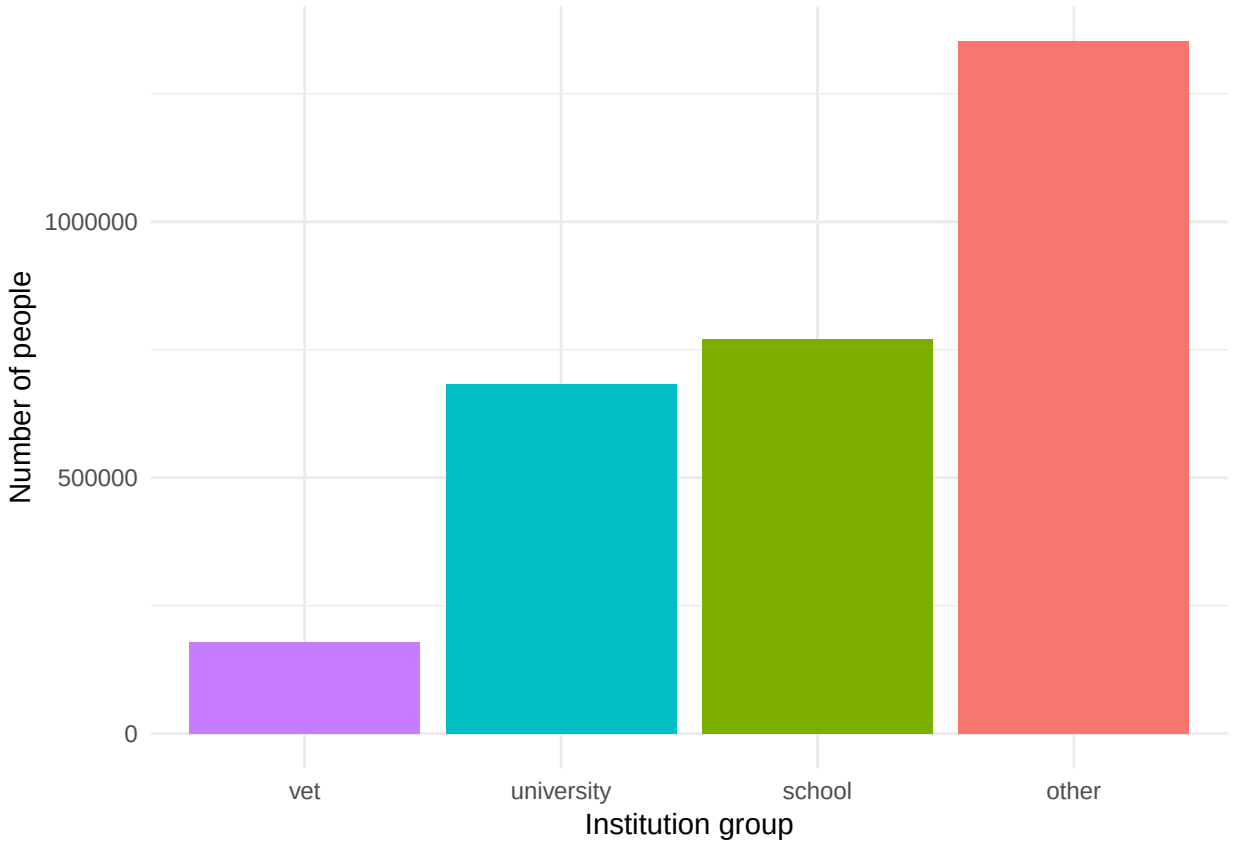
That has reordered things, but the label is a bit silly. We can adjust the labels:

```
inst_young_sum %>%
  ggplot(aes(x = reorder(inst_group, sum_n),
             y = sum_n,
             fill = inst_group)) +
  geom_bar(stat = "identity") +
  labs(x = "Institution group",
       y = "Number of people",
       fill = "Institution group")
```



Nicer. We don't really need the legend though, so we can turn it off:

```
inst_young_sum %>%
  ggplot(aes(x = reorder(inst_group, sum_n),
             y = sum_n,
             fill = inst_group)) +
  geom_bar(stat = "identity") +
  labs(x = "Institution group",
       y = "Number of people") +
  theme(legend.position = "off")
```



Sick. But our `inst_group` order still doesn't really make sense. We would actually like to specify the order ourselves so it runs: `school`, `vet`, `university`, `other`. To do this we need to use `factors`.

Factors

A factor is a special data type that is kind-of-like a character but with some sometimes-useful properties. They're designed to be used with categorical variables, like `inst_group`. If you want to get right into factors-world, check out the first few subsections of Chapter 15 of R4DS.

But, best to a quick example: we define a character vector of sizes.

- Looking at `size` shows the list of character elements.
- We then define a hierarchy in `size_order`, and
- use it to generate `size_factor` which contains the elements of `size` and the ability to order according to `levels = size_order`. Note the output of `size_factor`.
- What happened to our "HELLO" element? It wasn't part of the defined levels, so was wiped. This *can be useful* if you are ensuring that your data fit within a set of values. But because the resulting value is `NA`, it *can also be annoying*. Just be wary.
- We can also call for the levels of `size_factor`.

```
size <- c("big", "big", "largest", "big", "small", "small", "tiny", "medium", "small", "HELLO")

size
```

```
##  [1] "big"     "big"     "largest" "big"     "small"   "small"   "tiny"
##  [8] "medium"  "small"   "HELLO"
```

```
size_order <- c("tiny", "small", "medium", "big", "largest")

size_factor <- factor(size,
                      levels = size_order)


size_factor
```

```
##  [1] big     big     largest big     small   small   tiny    medium
##  [9] small   <NA>
## Levels: tiny small medium big largest
```

```
levels(size_factor)
```

```
## [1] "tiny"    "small"   "medium"  "big"     "largest"
```

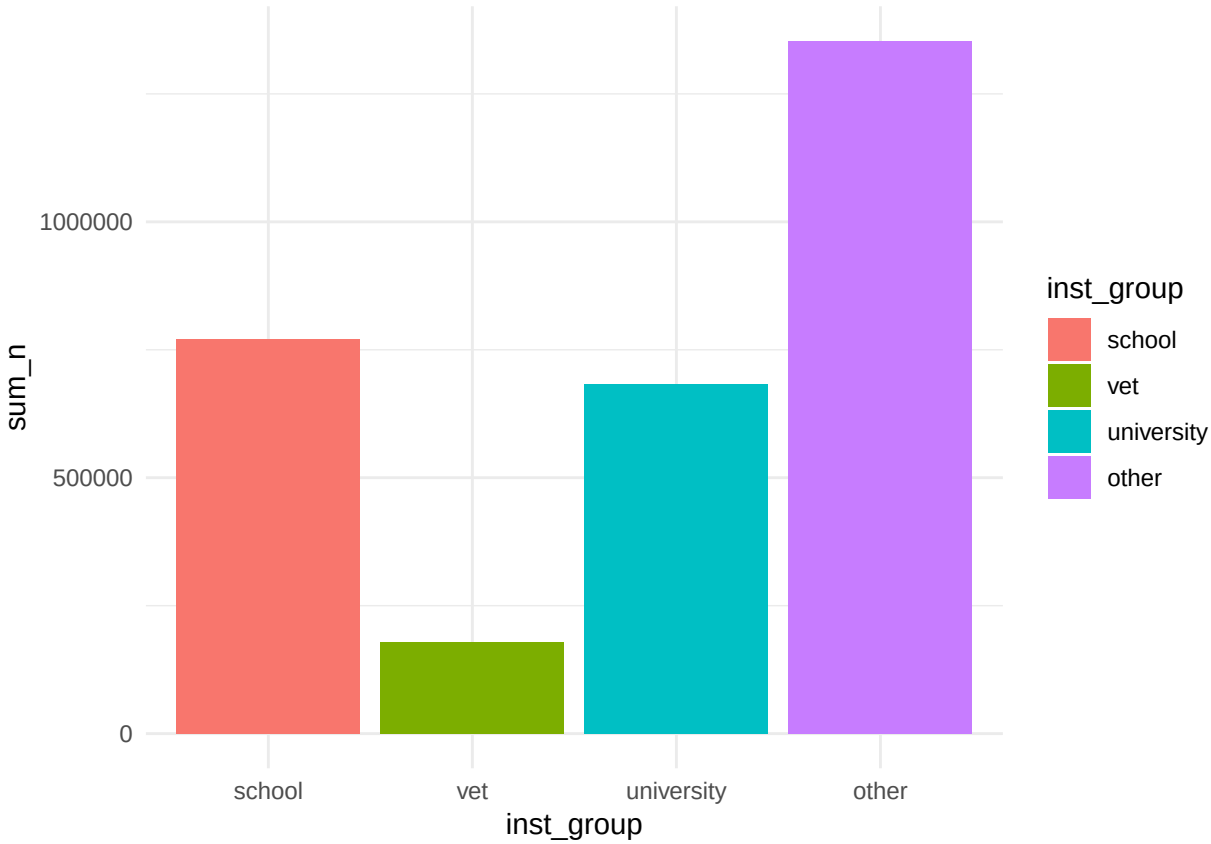
This comes in handy when we want to order unique elements of a vector. We can do the same thing for `inst_group`. Define the levels, then convert the `inst_group` variable to a factor with those levels:

```
inst_group_levels <- c("school", "vet",
                       "university", "other")

inst_young_sum <- inst_young_sum %>%
  mutate(inst_group = factor(inst_group,
                             levels = inst_group_levels))
```

Now we can plot with some sort of control over the order:

```
inst_young_sum %>%
  ggplot(aes(x = inst_group,
             y = sum_n,
             fill = inst_group)) +
  geom_bar(stat = "identity")
```


1000000
500000
0
sum_n
school
vet
university
other
inst_group
inst_group
school
vet
university
other