A Short Programming Guide for Economics Research

César Landín

2023-03-08

Contents

Chapter 1

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 2

Tips for using ggplot to generate publication-quality graphs

2.1 Plot margins

2.1.1 Removing white space around the plot

To remove the white space around the plot, set the plot margins equal to 0. The order is [top, right, bottom, left].

```
figure %>%
  theme(plot.margin = margin(0, 0, 0, 0))
```

This setting is useful when working with package `cowplot` to generate multi-panel figures. `cowplot::plot_grid` often overlays panel labels on top of the figures, so you can add space to the top of the figure:

```
plot_grid(g1 + theme(plot.margin = margin(t = 15)),
          g2 + theme(plot.margin = margin(t = 15)),
          nrow = 2)
```

2.1.2 Removing white space between axes and plot

Sometimes it's useful to reduce the distance between the plot and axis text. You can do this by reducing the top margin of the x-axis text and the right margin of the y-axis text:

```
figure %>%
  theme(axis.text.x = element_text(margin = margin(t = -5, r = 0, b = 0, l = 0)),
```

```
        axis.text.y = element_text(margin = margin(t = 0, r = -5, b = 0, l = 0)))
```

2.2 Axis labels

2.2.1 Removing axis labels

When removing axis labels (for example, when dealing with dates in the x-axis), use `labs(x = NULL)` rather than `labs(x = "")`, as this eliminates the extra white space.

```
figure %>%
  labs(x = NULL)
```

2.3 Legends

2.3.1 Removing the legend title

Legend titles are always redundant: if the figure is included in the paper or report, then the title and axis label give enough information; if the figure is in a presentation, the slide title and description along with axis labels provide enough information to understand what is being plotted.

```
figure %>%
  theme(legend.title = element_blank())
```

2.3.2 Removing black boxes around legend keys

Legend keys look much better without the black border surrounding them. This should be a standard for any figure.

```
figure %>%
  theme(legend.key = element_blank())
```

2.3.3 Putting the legend inside the figure

Useful for when there is a lot of blank space in the figure.

```
figure %>%
  theme(legend.position = c(0.75, 0.85))
```

2.4 Geographic place matching

In applied work, we often have to deal with observations that have an associated address or coordinates but no geographic codes. For cases in which no coordinates are available, one option is to match addresses to coordinates using Google

Place API searches, and then merge the resulting coordinates with shapefiles to obtain geographic codes.

2.4.1 Google Places API searches

R package `googleway` makes it easy to perform Google Place API searches. `googleway::google_find_place` generates a Find Place request, taking a text input and returning an array of place candidates, along with their corresponding search status. From this result, we can extract the address and coordinates.

Currently, you get $200 of Google Maps Platform usage every month for free. Each request costs $0.017. While this may seem like little, generating 12,000 requests will already exceed the monthly free usage quota ($200 = 11,764.7 requests). It's easy to exceed this number of requests when you're running loops for large query vectors repeatedly. Therefore, the best practice is to start out with a small sample, ensure that searches are returning valid results, and then extend the method to the full sample. You can set a maximum quota of 375 requests per day (375 x 31 = 11,625) to ensure you don't exceed the monthly free usage limit.

To start using `googleway` to conduct Google Place API searches, you first need to create a Google Cloud project and set up an API key. Once you have set this up, you can load `googleway`, define the API key and start conducting searches. Here is a simple example of an individual query.

```
pacman::p_load(here, tidyverse, googleway)

# (1.1): Set Google Place search API key.
key <- "KJzaLyCLI-nXPsHqVwz-jna1HYg2jKpBueSsTWs" # insert API key here
set_key(key)

# (1.2): Define tibble with addresses to look up.
missing_locs <- tribble(~id, ~address,
                        1, "Av. Álvaro Obregón 225, Roma Norte, Cuauhtémoc, CDMX, Mexico",
                        2, "Río Hondo #1, Col. Progreso Tizapán, Álvaro Obregón, CDMX, México")

# (1.3): Loop over missing addresses.
loc_coords <- tibble()
for (i in 1:nrow(missing_locs)) {
  # Get results from Google Place search
  results <- google_find_place(missing_locs$address[i], inputtype = "textquery", language = "es")

  # Print results
  search_status <- ifelse(results[["status"]] == "OK",
                          "search successful",
                          "search returned no results")
  print(str_c("Working on address ", i, " out of ", nrow(missing_locs), ", ", search_status))
```

```
  # Extract formatted address and coordinates results
  clean_results <- tibble(id = missing_locs$id[i],
                          address_clean = results[["candidates"]][["formatted_address"]
                          loc_lat = results[["candidates"]][["geometry"]][["location"]]
                          loc_lon = results[["candidates"]][["geometry"]][["location"]]

  # Append to full results dataframe.
  loc_coords %<>% bind_rows(clean_results)
}
rm(results, clean_results, i, search_status)

# (1.4): Keep first result for each address.
clean_results %<>%
  group_by(id) %>%
  slice(1)
```

Once you finish your set of Google Place API requests, you should save the results to a CSV file for later use. The search process is not perfectly replicable as identical searches can produce different results over time, so you should only run your full search loop once.

2.5 Number formatting functions

You can save these functions in a script called number_functions.R and import them in each script where they're needed, e.g.:

```
source(here("scripts", "programs", "number_functions.R"))
```

2.5.1 Calculating the mean, median and standard deviation of a variable

```
# Mean
num_mean <- function(df, variable, dig = 1) {
  df %>%
    pull(eval(as.name(variable))) %>%
    mean(na.rm = TRUE) %>%
    round(digits = dig)
}

# Median
num_median <- function(df, variable, dig = 1) {
  df %>%
    pull(eval(as.name(variable))) %>%
    median(na.rm = TRUE) %>%
```

```
    round(digits = dig)
}

# Standard deviation
num_sd <- function(df, variable, dig = 1) {
  df %>%
    pull(eval(as.name(variable))) %>%
    sd(na.rm = TRUE) %>%
    round(digits = dig)
}
```

You can call these functions the following way:

```
df %>% num_mean("number_employees")
```

2.5.2 Checking if a number is an integer.

This is used in the functions that print numbers to .tex files, since no decimals should be added after integers.

```
num_int <- function(x) {
  x == round(x)
}
```

2.5.3 Number formatting function

This function formats numbers with a standard number of digits and commas to present large and small numbers in a more readable format.

How does this function work? First, the function calculates the number of digits the number should have to the right of the decimal point. For numbers from 1-9, three digits are assigned, two digits are assigned for numbers 10-99, one for 100-999, and no right digits for numbers equal or larger than 1,000. This function sets the maximum number of right digits as 3. Therefore, 0.0001 will display as 0.000. Once the number of right digits is defined, the number is formatted. Numbers smaller than 1 are padded if necessary to ensure that there are 3 right digits (e.g., 0.25 is formatted as 0.250). The default number of right digits can be overridden with the option `override_right_digits`.

This function has the same name as `scales::comma_format`. However, this function has been superseded by `scales::label_comma`, so there are no issues with this user-defined function taking priority over `scales::comma_format`.

```
comma_format <- function(x, override_right_digits = NA) {
  # Calculate number of right digits
  if (x <= 0) {num <- 1} else {num <- x}
  right_digits <- 3 - floor(log10(abs(num)))
  if (right_digits < 0) {right_digits <- 0}
```

```
  if (right_digits > 3) {right_digits <- 3}
  if (!is.na(override_right_digits)) {right_digits <- override_right_digits}
  # Calculate number of left digits
  left_digits <- 4 + floor(log10(abs(num)))
  if (left_digits <= 0) {left_digits <- 1}
  # Format number
  proc_num <- format(round(x, right_digits), nsmall = right_digits, digits =  left_digi
  if (proc_num != "0" & as.numeric(str_replace(proc_num, fixed(","), "")) < 1) {
    proc_num <- str_pad(proc_num, right_digits + 2, "right", "0")
  }
  return(proc_num)
}
```

Here are a few examples of the output of this function:

```
pacman::p_load(tidyverse)

lapply(c(0.098, 0.11, 3.1233, 45.968, 1949), comma_format) %>% unlist()
```

```
## [1] "0.098" "0.110" "3.123" "45.97" "1,949"
```