**Introduction**

There are often situations when you need to perform repetitive plotting tasks. For example, you’d like to plot the same kind of data (e.g. the same economic indicator) for several states, provinces, or cities. Here are some ways you can address this:

* You can try to fit all the data into the same plot. Often it works just fine, especially if [the data is simple and can easily fit into the plot](https://dataenthusiast.ca/?p=1122#plot-labor).
* Another option is to create a [faceted plot](https://dataenthusiast.ca/?p=1122#plot-work-by-gender), broken down by whatever grouping variable you choose (e.g. by city or region).

But what if the data is too complex to fit into a single plot? Or maybe there are just too many levels in your grouping variable – for example, if you try to plot family income data for all 50 U.S. states, a plot made up of 50 facets would be virtually unreadable. Same goes for a plot with all 50 states on its X axis.

Yet another example of a repetitive plotting task is when you’d like to use your own custom plot theme for your plots.

Both use cases – making multiple plots on the same subject, and using the same theme for multiple plots – require the same R code to run over and over again. Of course, you can simply duplicate your code (with necessary changes), but this is tedious and not optimal, putting it mildly. In case of plotting data for all 50 U.S. states, would you copy and paste the same chunk of code 50 times?

Fortunately, there is a much better way – simply write a function that will iteratively run the code as many times as you need.

**Making Multiple Plots on the Same Subject**

Lets’ start with a more complex use case – making multiple plots on the same subject. To illustrate this, I will be using the ‘education’ dataset that contains education levels of people aged 25 to 64, broken down by gender, according to 2016 Canadian Census.

Let’s take a look at the first 20 lines of the ‘education’ dataset (all data for the ‘Canada’ region):

> head(education, 20)

# A tibble: 20 x 5

region vector count gender level

1 Canada v\_CA16\_5100 1200105 Male None

2 Canada v\_CA16\_5101 969690 Female None

3 Canada v\_CA16\_5103 2247025 Male High school or equivalent

4 Canada v\_CA16\_5104 2247565 Female High school or equivalent

5 Canada v\_CA16\_5109 1377775 Male Apprenticeship or trades

6 Canada v\_CA16\_5110 664655 Female Apprenticeship or trades

7 Canada v\_CA16\_5118 1786060 Male College or equivalent

8 Canada v\_CA16\_5119 2455920 Female College or equivalent

9 Canada v\_CA16\_5121 240035 Male University below bachelor

10 Canada v\_CA16\_5122 340850 Female University below bachelor

11 Canada v\_CA16\_5130 151210 Male Cert. or dipl. above bachelor

12 Canada v\_CA16\_5131 211250 Female Cert. or dipl. above bachelor

13 Canada v\_CA16\_5127 1562155 Male Bachelor's degree

14 Canada v\_CA16\_5128 2027925 Female Bachelor's degree

15 Canada v\_CA16\_5133 74435 Male Degree in health\*\*

16 Canada v\_CA16\_5134 78855 Female Degree in health\*\*

17 Canada v\_CA16\_5136 527335 Male Master's degree

18 Canada v\_CA16\_5137 592850 Female Master's degree

19 Canada v\_CA16\_5139 102415 Male Doctorate\*

20 Canada v\_CA16\_5140 73270 Female Doctorate\*

Our goal is to plot education levels as percentages for both genders, and for all regions. This is a good example of a repetitive plotting task, as we’ll be making one plot for each region. Overall, there are 6 regions, so we’ll be making 6 plots:

> levels(education$region)

[1] "Canada" "Halifax" "Toronto" "Calgary" "Vancouver" "Whitehorse"

Ideally, our plot should also reflect the hierarchy of education levels.

**Preparing the Data**

The data, as retrieved from Statistics Canada is of the Working with Statistics Canada Data in R series, is not yet ready for plotting: it doesn’t have percentages, only counts. Also, education levels are almost, but not quite, in the correct order: the ‘Cert. or dipl. above bachelor’ is before ‘Bachelor’s degree’, while it should of course *follow* the Bachelor’s degree.

So let’s apply some final touches to our dataset, after which it will be ready for plotting. First, lets load **tidyverse**:

library(tidyverse)

Then let’s calculate percentages and re-level the ‘levels’ variable:

# prepare 'education' dataset for plotting

education <- education %>%

group\_by(region) %>%

mutate(percent = round(count/sum(count)\*100, 1)) %>%

mutate(level = factor(level, # put education levels in logical order

levels = c("None",

"High school or equivalent",

"Apprenticeship or trades",

"College or equivalent",

"University below bachelor",

"Bachelor's degree",

"Cert. or dipl. above bachelor",

"Degree in health\*\*",

"Master's degree",

"Doctorate\*")))

Note that we needed to group the data by the ‘region’ variable to make sure our percentages get calculated correctly, i.e. by region. If you are not sure if the dataset has been grouped already, you can check this with the dplyr::is\_grouped\_df() function.

**Writing Functions to Generate Multiple Plots**

Now our data is ready to be plotted, so let’s write a function that will sequentially generate our plots – one for each region. Pay attention to the comments in the code:

## plot education data

# a function for sequential graphing of data by region

plot.education <- function(x = education) {

# a vector of names of regions to loop over

regions <- unique(x$region)

# a loop to produce ggplot2 graphs

for (i in seq\_along(regions)) {

# make plots; note data = args in each geom

plot <- x %>%

ggplot(aes(x = level, fill = gender)) +

geom\_col(data = filter(x, region == regions[i],

gender == "Male"),

aes(y = percent)) +

geom\_col(data = filter(x, region == regions[i],

gender == "Female"),

# multiply by -1 to plot data left of 0 on the X axis

aes(y = -1\*percent)) +

geom\_text(data = filter(x, region == regions[i],

gender == "Male"),

aes(y = percent, label = percent),

hjust = -.1) +

geom\_text(data = filter(x, region == regions[i],

gender == "Female"),

aes(y = -1\*percent, label = percent),

hjust = 1.1) +

expand\_limits(y = c(-17, 17)) +

scale\_y\_continuous(breaks = seq(-15, 15, by = 5),

labels = abs) + # axes labels as absolute values

scale\_fill\_manual(name = "Gender",

values = c("Male" = "deepskyblue2",

"Female" = "coral1")) +

coord\_flip() +

theme\_bw() +

theme(plot.title = element\_text(size = 14, face = "bold",

hjust = .5,

margin = margin(t = 5, b = 15)),

plot.caption = element\_text(size = 12, hjust = 0,

margin = margin(t = 15)),

panel.grid.major = element\_line(colour = "grey88"),

panel.grid.minor = element\_blank(),

legend.title = element\_text(size = 13, face = "bold"),

legend.text = element\_text(size = 12),

axis.text = element\_text(size = 12, color = "black"),

axis.title.x = element\_text(margin = margin(t = 10),

size = 13, face = "bold"),

axis.title.y = element\_text(margin = margin(r = 10),

size = 13, face = "bold")) +

labs(x = "Education level",

y = "Percent of population",

fill = "Gender",

title = paste0(regions[i], ": ", "Percentage of Population by Highest Education Level, 2016"),

caption = "\* Doesn’t include honorary doctorates.\n\*\* A degree in medicine, dentistry, veterinary medicine, or optometry.\nData: Statistics Canada 2016 Census.")

# create folder to save the plots to

if (dir.exists("output")) { }

else {dir.create("output")}

# save plots to the 'output' folder

ggsave(filename = paste0("output/",

regions[i],

"\_plot\_education.png"),

plot = plot,

width = 11, height = 8.5, units = "in")

# print each plot to screen

print(plot)

}

}

Let’s now look in detail at the key sections of this code. First, we start with creating a vector of regions’ names for our function to loop over, and then we follow with a simple for-loop: for (i in seq\_along(regions)). We put our plotting code *inside* the loop’s curly brackets { }.

Note the data = argument in each geom: region == regions[i] tells ggplot() to take the data that corresponds to each element of the ‘regions’ vector, for each new iteration of the for-loop.

Since we want our plot to reflect the hierarchy of education levels *and* to show the data by gender, the best approach would be to plot the data as a pyramid, with one gender being to the left of the center line, and the other – to the right. This is why each geom is plotted twice, with the dplyr::filter() function used to subset the data.

The y = -1\*percent argument to the aes() function tells the geom to plot the data to the left of the 0 center line. It has to be accompanied by labels = abs argument to scale\_y\_continuous(), which tells this function to use absolute values for the Y axis labels, since you obviously can’t have a negative percentage of people with a specific education level.

Note also the expand\_limits(y = c(-17, 17)), which ensures that axis limits stay the same in all plots generated by our function. This is one of those rare cases when expand\_limits() is preferable to coord\_flip(), since with expand\_limits() axis limits stay the same in all auto-generated plots.

Next, coord\_flip() converts a bar plot into a pyramid, so that education levels are on the Y axis, and percentages are on the X axis.

Finally, note how our for-loop uses regions[i] inside the labs() function to iteratively add the names of the regions to the plots’ titles, and to correctly name each file when saving our plots with ggsave().

To generate the plots, run:

plot.education()

Here is one of our plots:

If you did everything correctly, there should be five more graphics like this one in your “output” folder – one for each region in our dataset.

**Making Custom Plot Themes**

The other way how you can simplify repetitive plotting tasks, is by making your own custom plot themes. Since every plot theme in **ggplot2** is a function, you can easily save your favorite theme settings as a custom-made function. Making a theme is easier than writing functions to generate multiple plots, as you won’t have to write any loops.

Suppose, you’d like to save the theme of our education plots, and to use it in other plots. To do this, simply wrap theme settings in function():

## Save custom theme as a function ##

theme\_custom <- function() {

theme\_bw() + # note ggplot2 theme is used as a basis

theme(plot.title = element\_text(size = 14, face = "bold",

hjust = .5,

margin = margin(t = 5, b = 15)),

plot.caption = element\_text(size = 11, hjust = 0,

margin = margin(t = 15)),

panel.grid.major = element\_line(colour = "grey88"),

panel.grid.minor = element\_blank(),

legend.title = element\_text(size = 13, face = "bold"),

legend.text = element\_text(size = 12),

axis.text = element\_text(size = 12),

axis.title.x = element\_text(margin = margin(t = 10),

size = 13, face = "bold"),

axis.title.y = element\_text(margin = margin(r = 10),

size = 13, face = "bold"))

}

Note that this code takes one of **ggplot2** themes as a basis, and then alters some of its elements to our liking.

Let’s now use the saved theme in a plot. Usually it doesn’t matter what kind of data we are going to visualize, as themes tend to be rather universal. Note however, that sometimes the data and the type of visualization do matter. For example, our theme\_custom() won’t work for a pie chart, because our theme has grid lines and labelled X and Y axes.

To illustrate how this theme fits an entirely different kind of data, let’s plot some data about penguins. Why penguins? Because I love [Linux](https://en.wikipedia.org/wiki/Tux_(mascot))!

The data was recently released as the palmerpenguins package containing various measurements of 3 species of penguins. The package is quite educational:

**palmerpenguins** is not yet on CRAN, so you’ll need to install it from GitHub:

devtools::install\_github("allisonhorst/palmerpenguins")

library(palmerpenguins)

Let’s now make a scatterplot showing the relationship between the bill length and body mass in the three species of penguins from **palmerpenguins**. Let’s also add regression lines with 95% confidence intervals to our plot, and apply our custom-made theme:

## Plot penguins data with a custom theme

plot\_penguins <-

penguins %>%

group\_by(species) %>%

ggplot(aes(x = bill\_length\_mm,

y = body\_mass\_g,

color = species)) +

geom\_point(size = 2, na.rm = TRUE) +

geom\_smooth(aes(fill = species),

formula = y ~ x, # optional: removes message

method = "lm",

alpha = .3, # alpha level for conf. interval

na.rm = TRUE) +

# Note that you need identical name, labels, and values

# in both manual scales to avoid legend duplication:

# this merges two legends into one.

scale\_color\_manual(name = "Species",

values = c("Adelie" = "orange2",

"Chinstrap" = "dodgerblue",

"Gentoo" = "orchid")) +

scale\_fill\_manual(name = "Species",

values = c("Adelie" = "orange2",

"Chinstrap" = "dodgerblue",

"Gentoo" = "orchid")) +

theme\_custom() + # here is our custom theme

labs(x = "Bill length, mm",

y = "Body mass, grams",

title = "Body Mass to Bill Length in Adelie, Chinstrap, and Gentoo Penguins",

caption = "Data: Gorman, Williams, and Fraser 2014")

As usual, let’s save the plot to the ‘output’ folder and print it to screen:

# save plot to 'output' folder

ggsave("output/plot\_penguins.png",

plot\_penguins,

width = 11, height = 8.5, units = "in")

# print plot to screen

print(plot\_penguins)

Here it is:

**Updating Plot Themes**

Now, suppose your organization uses a green-colored theme for their website and reports, so your penguin data plot needs to fit the overall style. Fortunately, updating a custom theme is very easy: you re-assign those theme elements you’d like to change, e.g. to use a different color:

# further change some elements of our custom theme

theme\_custom\_green <- function() {

theme\_custom() +

theme(plot.title = element\_text(color = "darkgreen"),

plot.caption = element\_text(color = "darkgreen"),

panel.border = element\_rect(color = "darkgreen"),

axis.title = element\_text(color = "darkgreen"),

axis.text = element\_text(color = "darkgreen"),

axis.ticks = element\_line(color = "darkgreen"),

legend.title = element\_text(color = "darkgreen"),

legend.text = element\_text(color = "darkgreen"),

panel.grid.major = element\_blank())

}

Then simply replace theme\_custom() in the code above with theme\_custom\_green(). No other changes needed!