[Creating a Table 2](#_Toc130393647)

[Table Design Principles 3](#_Toc130393648)

[Principle One: Minimize Clutter 3](#_Toc130393649)

[Principle Two: Differentiate the Header From the Body 5](#_Toc130393650)

[Principle Three: Align Appropriately 7](#_Toc130393651)

[Principle Four: Use the Correct Level of Precision 9](#_Toc130393652)

[Principle Five: Use Color Intentionally 10](#_Toc130393653)

[Principle Six: Add a Data Visualization Where Appropriate 11](#_Toc130393654)

[Conclusion 14](#_Toc130393655)

Crafting High-Quality Tables

In his book Fundamentals of Data Visualization, Claus Wilke writes that “tables are an important tool for visualizing data.” This statement might seem odd. Tables are often considered the opposite of data visualizations such as plots: a place to dump numbers for the few nerds who care to read them. But Wilke sees things differently.

Tables should not be data dumps devoid of design. While bars, lines, and points in graphs are visualizations, so too are numbers in a table, and we should care about their appearance. As an example, take a look at the tables made by reputable news sources; data dumps these are not. Media organizations, whose job it is to communicate effectively, pay a lot of attention to table design. But elsewhere, because of tables’ apparent simplicity, Wilke writes, “they may not always receive the attention they need.”

Many people use Microsoft Word to make tables, a strategy that has potential pitfalls. Wilke found that his version of Word included 105 built-in table styles. Of those, around 80 percent violated some key principles of table design:

So if you pick a Microsoft Word table layout at random, you have an 80% chance of picking one that has issues. And if you pick the default, you will end up with a poorly formatted table every time.

The good news is that R is a great tool for making high-quality tables. It has a number of packages for this purpose, and within these packages, several functions designed to make sure your tables follow important design principles. Moreover, if you’re writing reports in R Markdown (which you’ll learn about in Chapter 6), you can include code that will generate a table when you export your document. By working with a single tool to create tables, text, and other visualizations, you won’t have to copy and paste your data, lowering the risk of human error.

This chapter examines these design principles and shows you how to apply them to your tables using R’s gt package, one of the most popular table-making packages (and, as you’ll soon see, one that uses good design principles by default). These principles, and the code in this chapter, are adapted from Tom Mock’s blog post “10+ Guidelines for Better Tables in R.” Mock works at Posit, the company that makes RStudio, and has become something of an R table connoisseur. We’ll walk through examples of Mock’s code to show how small tweaks can make a big difference.

Creating a Data Frame

We’ll begin by creating a data frame that we can use to make tables throughout this chapter. First, let’s load the packages we need. We’ll rely on the tidyverse package for general data manipulation functions, gapminder for the data we’ll use, and gt to make the tables:

library(tidyverse)

library(gapminder)

library(gt)

library(gtExtras)

As we saw in Chapter 2, the gapminder package provides country-level demographic statistics. To make a data frame for our table, let’s use just a few countries (the first four, in alphabetical order: Afghanistan, Albania, Algeria, and Angola) and three years (1952, 1972, and 1992). The gapminder data has many years, but we only need a few to demonstrate table-making principles. Here’s the code to make the data frame called gdp:

gdp <- gapminder %>%

filter(country %in% c("Afghanistan", "Albania", "Algeria", "Angola")) %>%

select(country, year, gdpPercap) %>%

mutate(country = as.character(country)) %>%

pivot\_wider(id\_cols = country,

names\_from = year,

values\_from = gdpPercap) %>%

select(country, `1952`, `1972`, `1992`) %>%

rename(Country = country)

Let’s see what gdp looks like:

#> # A tibble: 4 × 4

#> Country `1952` `1972` `1992`

#> <chr> <dbl> <dbl> <dbl>

#> 1 Afghanistan 779. 740. 649.

#> 2 Albania 1601. 3313. 2497.

#> 3 Algeria 2449. 4183. 5023.

#> 4 Angola 3521. 5473. 2628.

Now that we have some data, let’s use it to make a table.

Table Design Principles

Unsurprisingly, the principles of good table design are similar to those for data visualization more generally. In this section, we cover six of the most important.

Principle One: Minimize Clutter

As with data visualization, one of the most important principles of table design is to minimize clutter. One way we can do this is by removing unnecessary elements. A common source of clutter in tables is gridlines. Often, you see tables that look like Figure 5-1.

[F05001.png]



* + - * 1. A table with gridlines everywhere

Having gridlines around every single cell in our table is unnecessary and creates visual clutter that distracts from the goal of communicating clearly. A table with minimal or even no gridlines (Figure 5-2) is a much more effective communication tool.

[F05002.png]



* + - * 1. A table with only horizontal gridlines

I mentioned that gt uses good table design principles by default, and this is a great example of it. The second table, with minimal gridlines, requires just two lines of code. We pipe our gdp data into the gt() function, which creates a table:

gdp %>%

gt()

To add gridlines to every part of the example, we would have to add additional code. Here, the code that follows gt() adds gridlines:

gdp %>%

gt() %>%

tab\_style(

style = cell\_borders(

side = "all",

color = "black",

weight = px(1),

style = "solid"

),

locations = list(

cells\_body(

everything()

),

cells\_column\_labels(

everything()

)

)

) %>%

opt\_table\_lines(extent = "none")

Since I don’t recommend taking this approach, I won’t walk through this code. However, if we wanted to remove additional gridlines, we could use the following:

gdp %>%

gt() %>%

tab\_style(

style = cell\_borders(color = "transparent"),

locations = cells\_body()

)

The tab\_style() function uses a two-step approach. First, it identifies the style we want to modify (in this case, the borders); next, it tells the function where to apply these styles. Here, we tell tab\_style() that we want to modify the borders using the cell\_borders() function, making our borders transparent. Then, we say that we want this transformation to apply to the cells\_body() location. Other options include cells\_column\_labels() for the first row.

Doing this gives us a table with no gridlines at all in the body (Figure 5-3).

[F05003.png]



* + - * 1. A table with gridlines only on the header row and bottom

Let’s save this table as an object called table\_no\_gridlines so that we can add onto it later.

Principle Two: Differentiate the Header from the Body

While reducing clutter is an important goal, going too far can have negative consequences. A table with no gridlines at all can make it hard to differentiate between the header row and the table body. Take Figure 5-4, for example.

[F05004.png]



* + - * 1. A table with all gridlines removed

We’ve already covered how to use appropriate gridlines. But by making the header row bold, we can make it stand out even more:

table\_no\_gridlines %>%

tab\_style(

style = cell\_text(weight = "bold"),

locations = cells\_column\_labels()

)

We start with the table\_no\_gridlines object (our saved table from earlier). Then, we apply our formatting with the tab\_style() function, using two steps. First, we say that we want to alter the text, using the cell\_text() function, by setting the weight to bold. Second, we say we want this to happen only to the header row using the cells\_column\_labels() function. In Figure 5-5, we can see what our table looks like with headers bolded.

[F05005.png]



* + - * 1. A table with the header row bolded

Let’s save this table as table\_bold\_header in order to add additional formatting.

Principle Three: Align Appropriately

A third principle of high-quality table design is appropriate alignment. Specifically, numbers in tables should be right-aligned. Tom Mock explains why:

Left-alignment or center-alignment of numbers impairs the ability to clearly compare numbers and decimal places. Right-alignment lets you align decimal places and numbers for easy parsing.

Let’s see this principle in action. In Figure 5-6, we’ve left-aligned 1952, center-aligned 1972, and right-aligned 1992. You can see how much easier it is to compare the values in the 1992 column than in the other two columns. In both 1952 and 1972, it is much more difficult to compare the numeric values because the numbers in the same columns (the tens place, for example) are not in the same vertical position. In 1992, however, the number in the tens place in Afghanistan (4) aligns with the number in the tens place in Albania (9) and all other countries. This vertical alignment makes it easier to scan the table.

[F05006.png]



* + - * 1. A table with year columns aligned to the left, center, and right

As with other tables, we actually have to override the defaults to get the gt package to misalign the columns, as you can see in the following code. By default, gt will right-align numeric values. Don’t change anything and you’ll be golden!

table\_bold\_header %>%

cols\_align(align = "left",

columns = 2) %>%

cols\_align(align = "center",

columns = 3) %>%

cols\_align(align = "right",

columns = 4)

Right alignment is best practice for numeric columns, but for text columns, use left alignment. As Jon Schwabish points out, it’s much easier to read longer text cells when they are left aligned. To illustrate the benefit of left-aligning, let’s add a country with a long name to the table. I’ve added Bosnia and Herzegovina and saved this as a data frame called gdp\_with\_bosnia. You’ll see that I’m using nearly the same code as I used to create the gdp data frame above.

gdp\_with\_bosnia <- gapminder %>%

filter(country %in% c("Afghanistan", "Albania", "Algeria", "Angola", "Bosnia and Herzegovina")) %>%

select(country, year, gdpPercap) %>%

mutate(country = as.character(country)) %>%

pivot\_wider(id\_cols = country,

names\_from = year,

values\_from = gdpPercap) %>%

select(country, `1952`, `1972`, `1992`) %>%

rename(Country = country)

Here’s what the gdp\_with\_bosnia data frame looks like:

#> # A tibble: 5 × 4

#> Country `1952` `1972` `1992`

#> <chr> <dbl> <dbl> <dbl>

#> 1 Afghanistan 779. 740. 649.

#> 2 Albania 1601. 3313. 2497.

#> 3 Algeria 2449. 4183. 5023.

#> 4 Angola 3521. 5473. 2628.

#> 5 Bosnia and Herzegovina 974. 2860. 2547.

Now take the gdp\_with\_bosnia data frame and create a table with the country column center aligned. In the table in Figure 5-7, it’s hard to scan the country names, and that center-aligned column just looks a bit weird.

[F05007.png]



* + - * 1. A table with country column center aligned

This is another example where we’ve had to change the gt defaults to mess things up. In addition to right-aligning numeric columns by default, gt left-aligns character columns. So, if we don’t touch anything, it will give us the alignment we’re looking for (Figure 5-8).

[F05008.png]



* + - * 1. A table with country column left aligned

If you ever do want to override the default alignments, you can use the cols\_align() function. For example, here is how to make that table with the country names center aligned:

gdp\_with\_bosnia %>%

gt() %>%

tab\_style(

style = cell\_borders(color = "transparent"),

locations = cells\_body()

) %>%

tab\_style(

style = cell\_text(weight = "bold"),

locations = cells\_column\_labels()

) %>%

cols\_align(columns = "Country",

align = "center")

Within this function, we use the columns argument to tell gt which columns to align, and the align argument to select our alignment.

Principle Four: Use the Correct Level of Precision

In all of the tables we’ve made so far, we’ve used the data exactly as it came to us. The numeric columns, for example, extend their data to four decimal places. This is almost certainly too many. Having more decimal places makes a table harder to read, so you should always strike a balance between what Jon Schwabish describes as “necessary precision and a clean, spare table.”

Here is another way I’ve heard this principle described: If adding additional decimal places would change some action, keep them; otherwise, take them out. In my experience, people tend to leave too many decimal places in, putting too much importance on a very high degree of accuracy (and, in the process, reducing the legibility of their tables).

In our GDP table, we can use the fmt\_currency() function to format our numeric values. (The gt package has a whole series of functions for formatting values in tables, all of which start with fmt\_.) In the following code, we apply fmt\_currency()to the 1952, 1972, and 1992 columns, then use the decimals argument to tell fmt\_currency() to format the values with zero decimal places. After all, the difference between a GDP of $799.4453 and $779 is unlikely to lead to different decisions, so I’m comfortable with sacrificing precision for legibility:

table\_bold\_header %>%

fmt\_currency(

columns = c(`1952`, `1972`, `1992`),

decimals = 0

)

We end up with values formatted as dollars. The fmt\_currency() function automatically adds a thousands-place comma to make the values even easier to read (Figure 5-9).

[F05009.png]



* + - * 1. A table with numbers rounded to whole numbers and dollar signs added

Now save your table for reuse.

Principle Five: Use Color Intentionally

So far, our table hasn’t used any color. We’ll add some now to highlight outlier values. Especially for readers who want to scan your table, highlighting outliers with color can help significantly. Let’s make the highest value in the year 1952 a different color. To do this, we again use the tab\_style() function:

table\_whole\_numbers %>%

tab\_style(style = cell\_text(color = "orange",

weight = "bold"),

locations = cells\_body(

columns = `1952`,

rows = `1952` == max(`1952`)

))

This function uses cell\_text() to both change the color of the text to orange and make it bold. Within the cells\_body() function, we use the locations() function specify the columns and rows to which we want to apply our change. You can see that we’ve simply set the columns argument to the year whose values we’re changing. To set the rows, we need a more complicated formula. The code rows = `1952` == max(`1952`) causes the text transformation to occur in rows whose value is equal to the maximum value in that year.

If we repeat this code for the 1972 and 1992 columns, we generate the result shown in Figure 5-10.

[F05010.png]



* + - * 1. A table with color added to show the highest value in each year

As always, save this table to avoid having to repeat all of the formatting code we’ve created so far.

Principle Six: Add a Data Visualization Where Appropriate

Adding color to highlight outliers is one way to help guide the reader’s attention. Another way is to incorporate graphs into tables. Tom Mock developed an add-on package for gt called gtExtras that makes it possible to do just this. For example, in our table, we might want to show how the GDP of each country changes over time. To do that, we’ll add a new column that visualizes this trend using a sparkline (essentially, a simple line chart):

gdp\_with\_trend <- gdp %>%

group\_by(Country) %>%

mutate(Trend = list(c(`1952`, `1972`, `1992`))) %>%

ungroup()

The gt\_plt\_sparkline() function that we use to do this requires us to have all of the values needed to make the sparkline available in a single column. To accomplish this, we create a variable called Trend, using group\_by() and mutate(), to hold a list of the values for each country. For Afghanistan, for example, Trend would contain 779.4453145, 739.9811058, and 649.3413952. We save this data as an object called gdp\_with\_trend.

Now we create our table as before, but at the end of our code, we add the gt\_plt\_sparkline() function. Within this function, we specify which column to use to create the sparkline (Trend):

gdp\_with\_trend %>%

gt() %>%

tab\_style(

style = cell\_borders(color = "transparent"),

locations = cells\_body()

) %>%

tab\_style(

style = cell\_text(weight = "bold"),

locations = cells\_column\_labels()

) %>%

fmt\_currency(

columns = c(`1952`, `1972`, `1992`),

decimals = 0

) %>%

tab\_style(style = cell\_text(color = "orange",

weight = "bold"),

locations = cells\_body(

columns = `1952`,

rows = `1952` == max(`1952`)

)) %>%

tab\_style(style = cell\_text(color = "orange",

weight = "bold"),

locations = cells\_body(

columns = `1972`,

rows = `1972` == max(`1972`)

)) %>%

tab\_style(style = cell\_text(color = "orange",

weight = "bold"),

locations = cells\_body(

columns = `1992`,

rows = `1992` == max(`1992`)

)) %>%

gt\_plt\_sparkline(column = Trend,

label = FALSE,

palette = c("black", "transparent", "transparent", "transparent", "transparent"))

We set label = FALSE to remove text labels that gt\_plt\_sparkline() adds by default, then add a palette argument to make the sparkline black and all other elements of it transparent (by default, the function will make different parts of the sparkline different colors). The stripped-down sparkline in Figure 5-11 allows the reader to see the trend for each country at a glance.

[F05011.png]



* + - * 1. A table with sparklines added to show GDP trends over time

The gtExtras package can do way more than merely create sparklines. Its set of theme functions allow you to make your tables look like those published by FiveThirtyEight, The New York Times, The Guardian, and other news outlets. As an example, try removing the formatting we’ve applied so far and instead use the gt\_theme\_538() function to style the table:

gdp %>%

group\_by(Country) %>%

mutate(Trend = list(c(`1952`, `1972`, `1992`))) %>%

ungroup() %>%

gt() %>%

tab\_style(style = cell\_text(color = "orange",

weight = "bold"),

locations = cells\_body(

columns = `1952`,

rows = `1952` == max(`1952`)

)) %>%

tab\_style(style = cell\_text(color = "orange",

weight = "bold"),

locations = cells\_body(

columns = `1972`,

rows = `1972` == max(`1972`)

)) %>%

tab\_style(style = cell\_text(color = "orange",

weight = "bold"),

locations = cells\_body(

columns = `1992`,

rows = `1992` == max(`1992`)

)) %>%

fmt\_currency(

columns = c(`1952`, `1972`, `1992`),

decimals = 0

) %>%

gt\_plt\_sparkline(column = Trend,

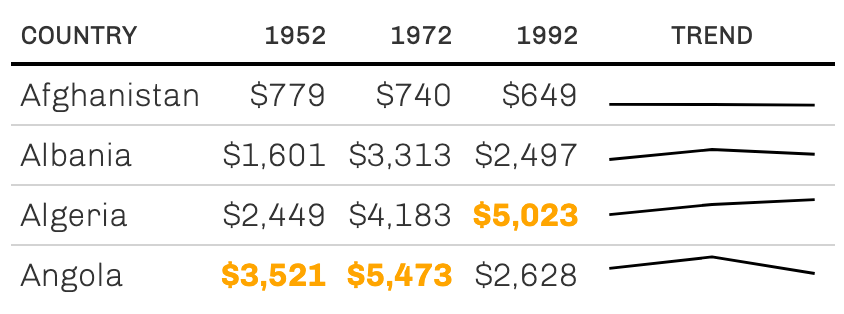
label = FALSE,

palette = c("black", "transparent", "transparent", "transparent", "transparent")) %>%

gt\_theme\_538()

Take a look at tables on the FiveThirtyEight website, and you’ll see similarities to the one in Figure 5-12.

[F05012.png]



* + - * 1. A table redone in FiveThirtyEight style

Add-on packages like gtExtras are common in the table-making landscape. If you are working with the reactable package to make interactive tables, for example, you can also use the reactablefmtr to add interactive sparklines, themes, and more (you’ll learn a bit more about making interactive tables in Chapter 9). The functionality that you get from these packages is enough to never make you go back to making tables in Word!

Conclusion

Many of the tweaks we made to our table are quite subtle. Changes like removing excess gridlines, bolding header text, right aligning numeric values, and adjusting the level of precision can often go unnoticed, but if you skip them, your table will be far less effective. Our final product isn’t flashy, but it does communicate clearly.

We used the gt package to make our high-quality table, and as we’ve repeatedly seen, this package has good defaults built in. Often, you don’t need to change much in your code to make effective tables. But no matter which package you use, it’s essential to treat tables as worthy of just as much thought as other kinds of data visualization.

In Chapter 6, you’ll learn how to create reports using R Markdown, which you can use to integrate your tables directly into the final document. What’s better than using just a few lines of code to make publication-ready tables?