

3

Visualization Basics

Your visuals should be part of your communication plan to share your findings, but sometimes even the best crafted figure can not avoid that the reading audience seeks and sees something different. Then, the meaning of “best crafted” may depend more on how well you know your audience than on how good you are at using the plotting functionalities of **R** and **Python** (or any other software for that matter). I have come to the conclusion that the simpler the better, and that is what I am planing to share next.

In this chapter, I will give some general recommendations on making your visuals, while helping you become familiar with **R** or **Python** plotting functions and philosophy. I will introduce all of this using data present in tables (data frames), just focusing on the data types mentioned in Section 2.1. When dealing with tabular data, you can suspect that you might produce a visualization for each column, and then for a couple of them simultaneously, and then for three or more. In this chapter, the examples will simply use univariate exploration; which is common for searching for problems or verifying outcomes; not for giving explanations.

3.1 Elements of Information Visualization

3.1.1 Components

The components of most visuals are well known, but I just want to make sure you have this clear, as often you may find visuals that do not include what should be included. Also, consider the kind of product you are making: unless there is such a thing as “free style”, you should follow a particular academic style.

Title

Important academic styles, such as the one from the American Psychological Association (2010), or APA, recommend that titles go below the image (while *tables* should have them on top). For APA, you are expected to explain the image shown, as well as assign a number to it.

However, if you were to put a title on the top space within the plotting area you may consider using this to make a point:

- a *question* answered by the plot
- a *guide* for the reader to understand the purpose of your plot
- It can *suggest* a possible conclusion.

Titles that achieve those goals are not that easy to produce. You need to rewrite them many times, until you find a good combination of words that can be read and understood fast enough before the audience loses connection. It is also good to keep in mind that you must never give your audience a cacophonous version (a “tongue twister”). If you need, make use of subtitles with a smaller font size, to explain period and location, or similar information worth stating. Let me use the *Democracy Index for 2019* (The Economist Intelligence Unit, 2019). Let’s call the data:

```
> linkRepo="https://github.com/resourcesbookvisual/data/"
> linkDemo="raw/master/demo.rda" # "R data" file!
> load(url(paste0(linkRepo,linkDemo)))
```

These examples will use *ggplot2* functions(Wickham, 2016), or simply *ggplot*, whose approach is to build layers of information to improve a visual layer after layer:

```
> # call plotting library
> library(ggplot2)
> # produce info: inform dataset, and variables to use
> info=ggplot(data=demo,aes(x=Continent))
```

That code just called the library which needs to be previously installed. Then, I created the **info** object. This object represents the input: the data frame, and the variables to use. The variables to plot are defined in **aesthetics**. Each variable will later be represented in some way. The next step is to add *geometry object* being used:

```
> # add a particular geometry object with the info and create plot
> titles1=info + geom_bar()
```

```
> # show it:
> titles1
```

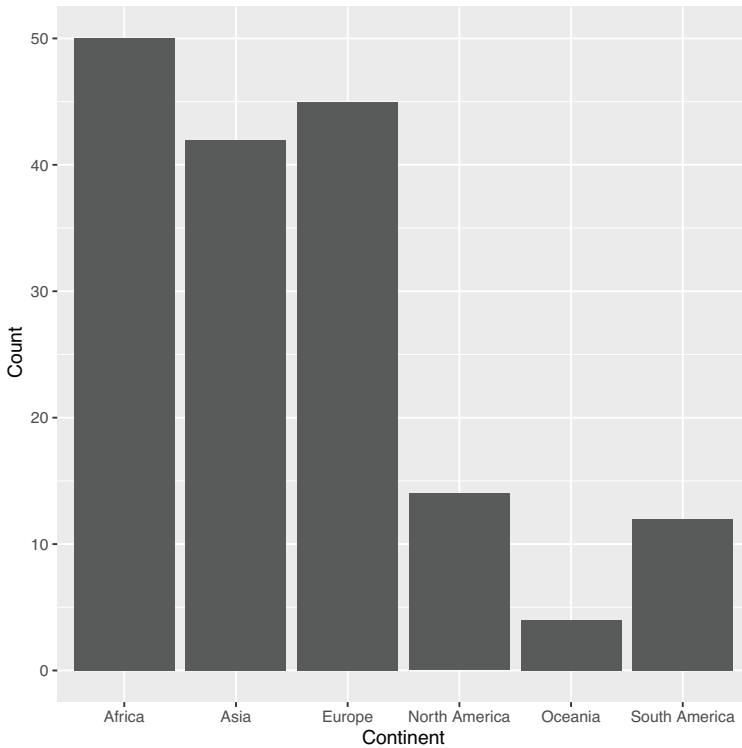


Figure 3.1 Default plot

Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

I have added a layer to the object `info`. Generally, the first layer represents the type of plot, in this case the variable in `aesthetics` from the `info` object will be represented by bars. The resulting plot is stored into the `titles1` object.

The default plot, as shown in Figure 3.1, does not have a title as *ggplot* does not produce one by default (a barplot using basic plotting functions in **R**, without using *ggplot*, puts the name of the variable as the default title). Let's write a title and a subtitle:

```
> # Titles to be used:
> the_Title="A NICE TITLE"
> the_SubTitle="A nice subtitle"
> # adding the titles:
> titles2=titles1 + ggtitle(label = the_Title,
+                           subtitle = the_SubTitle)
```

```
> # result  
> titles2
```

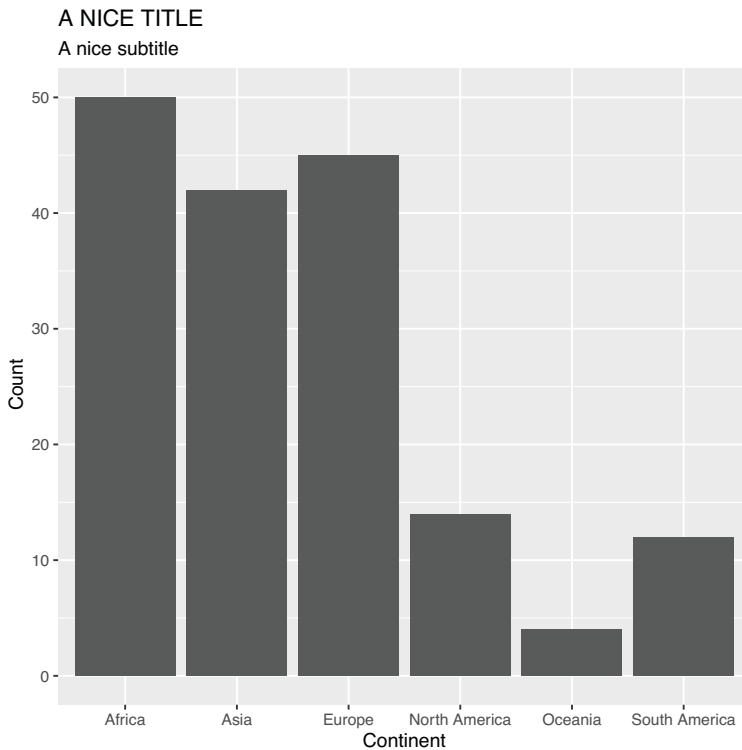


Figure 3.2 Adding titles

Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

Notice that I prefer writing the text outside the code. I consider that this practice enhances readability and reusability: you can use this code again with other variables, just changing the text. Besides, the changes you will make to the text will be not be close to the functions, which will reduce the possibility of your erasing something important by mistake. You can see the result in Figure 3.2.

Figure 3.2 offers you default axes titles, which most of the time you need to change. Make sure they are clearly stated, specially if they represent some unit of measurement. If you are using a plot in a meeting, you do not want the decision maker or any one in the audience constantly interrupting your presentation with questions that reveal they found basic imperfections in your work. Let's add the axes titles:

```
> # Axes to be used:
> horizontalTitle="Continents present in the study"
> verticalTitle="Number of countries studied"
> # adding the axes titles:
> titles3=titles2 + xlab(horizontalTitle) + ylab(verticalTitle)
```

Notice that I am again adding layers to what I had in `titles2`; I am also writing the texts outside *ggplot* functions. Let's see the newest version in Figure 3.3:

Source

The source of your image should be clearly stated. You should include it below the plot. You need to cite the data source, or any other author of the image, following the same style you are using for all your citations. You should state clearly if the source comes from your own work. Again, a particular academic style may offer some specific requirement for this. Let me add another layer on top of `titles3`:

```
> # result
> titles3
```

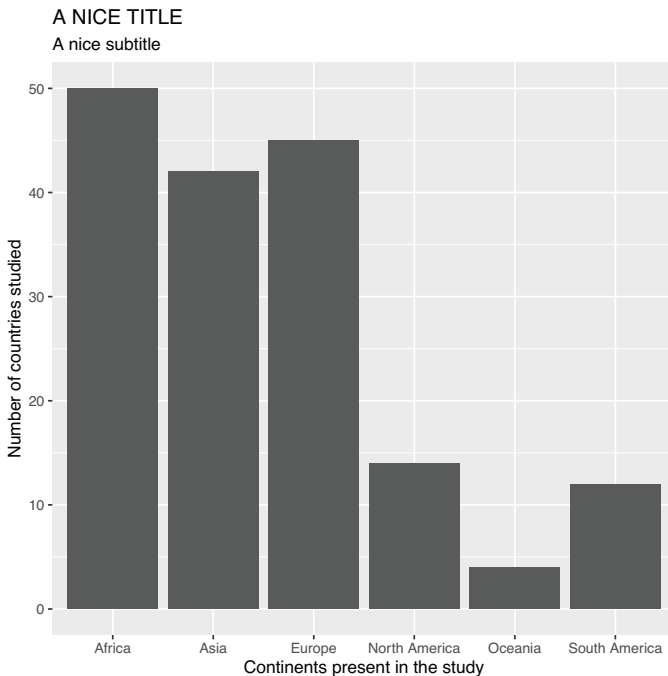


Figure 3.3 Axes titles

Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

```
> # Source to be used:  
> theSource="Source: Democracy Index at Wikipedia"  
> # adding the source:  
> source=titles3 + labs(caption = theSource)
```

Now, the `source` object will have a plot with its source, as shown in Figure 3.4.

Annotations

Annotations help readers focus on some section of your plot. There can be one or more, and they can be a combination of text messages, reference lines, and reference polygons.

Annotations need location information: identify the coordinates from the previous plot. You have to manually locate an annotation using the coordinate system of your current plot (the range of values of your data). Notice that the horizontal is showing the levels of the nominal variable, but not numbers.

```
> # result  
> source
```

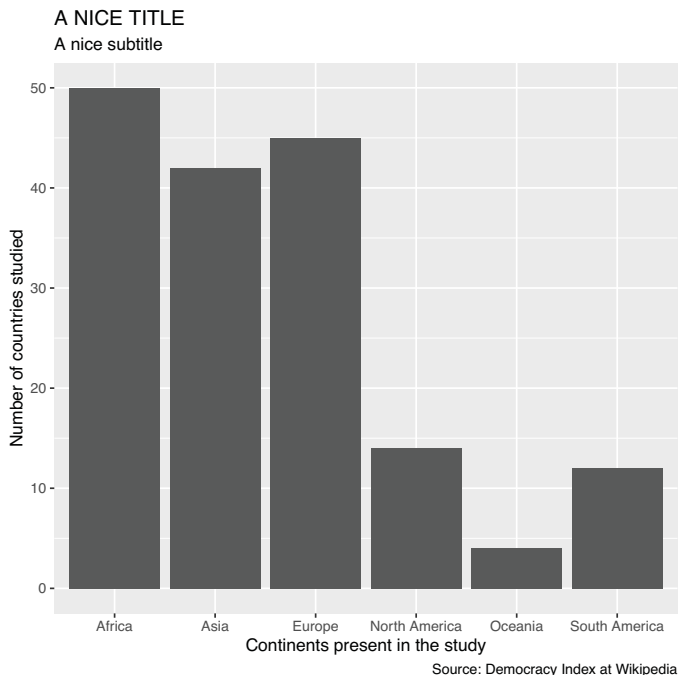


Figure 3.4 Adding source to plot

Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

Then the possible values in the horizontal represent a number based on how many levels your nominal variable has. Let me prepare my extra layer for an annotation:

```
> # data to input to \emph{ggplot} layer:
> theCoordinates=list(X=5,Y=10)
> theMessage="So few?!"
> # adding annotation layer
> annot=source + annotate("text",
+                           x = theCoordinates$X,
+                           y = theCoordinates$Y,
+                           label = theMessage)
```

I chose **5** for the horizontal, as the limit of this axis are one and six (six bars). I chose **10** for the vertical, which is a possible value in that axis. The last version can be seen in Figure 3.5.

```
> # result
> annot
```

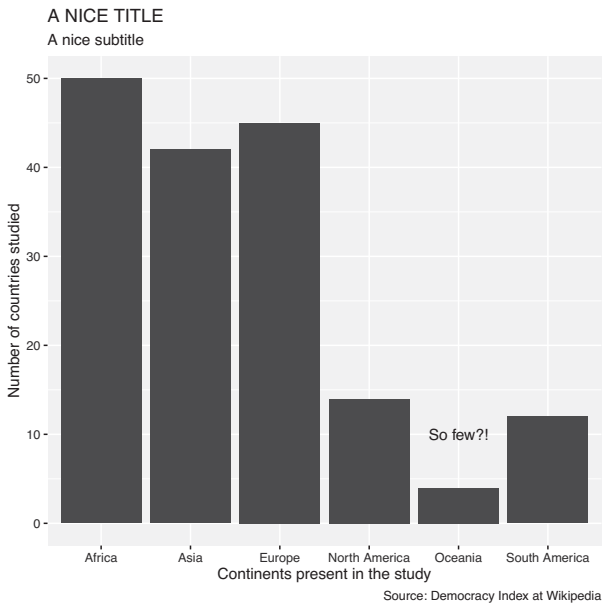


Figure 3.5 Adding annotations to plot

Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

You must avoid cluttering your plot with annotations. If you believe several annotations are needed, then it might be possible you chose the wrong plot in the first place.

Legend

Legends are a traditional element that are generally used to explain the symbols used in the plot. Legends are part of the plotting area. Let me illustrate the relationship between a couple of numerical variables:

```
> info=ggplot(demo, aes(x=Culture, y=Functioning, shape=Continent))
> leyenda=info + geom_point()
> # result
> leyenda
```

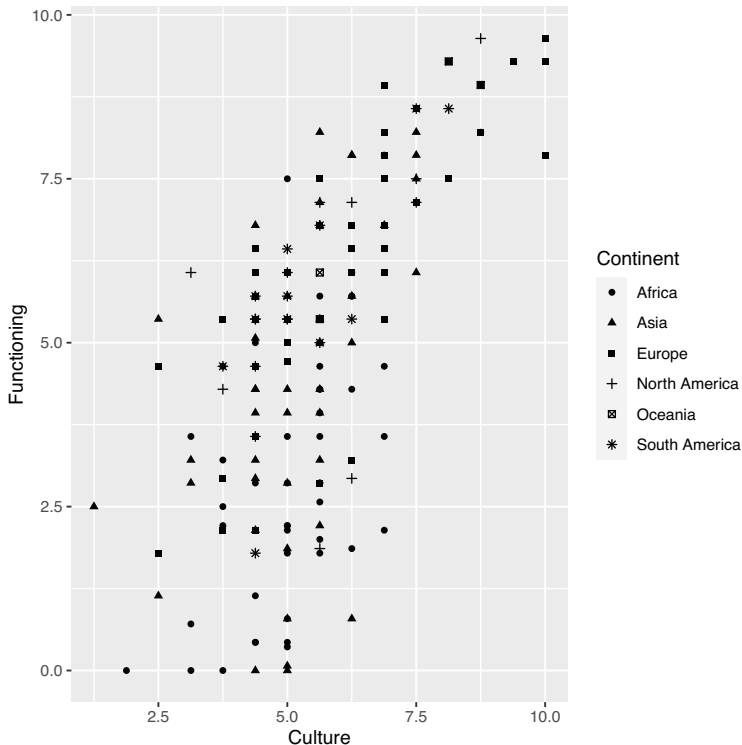


Figure 3.6 Plotting with legend

Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

From Figure 3.6, you can notice the following:

- The **aes** defined the coordinates of the points with **x** and **y** values, but **shape** caused the presence of the legend (you did not request a legend in the code).
- The plot and the legend share the same area; now you have less room for your main message.
- The shape of the dot is not a square anymore, so the legend affected the quality of the plot, by default.

If your aesthetics included **color** and **size** besides **shape**, you will get multiple legends. You can decide which legend stays by using the command **guides** (see Figure 3.7):

```
> # "none" or FALSE for 'no legend'
> leyenda + guides(shape = 'none', color=FALSE, size="none")
```

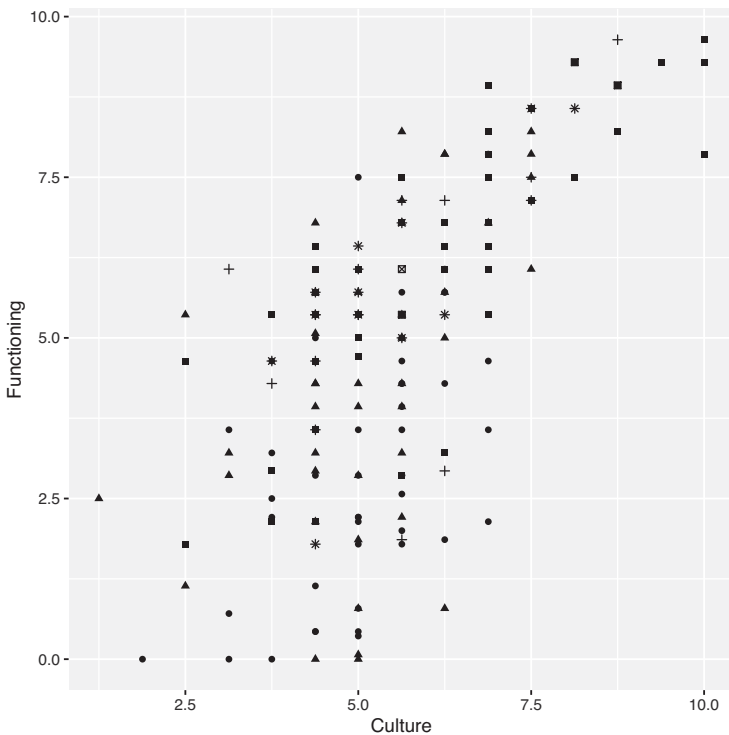


Figure 3.7 Plotting with no legend using **guides**

Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

3.1.2 Objects

Every visual is composed of a set of objects that will encode the information you are trying to share. In *ggplot* jargon, they are known as the *geoms*. Now, I can classify objects by their dimensionality:

- **Dots** are theoretically an adimensional object (no width, no height), so they can represent **location**. In Figure 3.6, each dot location was determined by the value of the axes: *culture* on the horizontal, and *government functioning* on the vertical. You can change the default shape using Figure 3.8.

You can then use a value from Figure 3.8 in `geom_point`:

```
> geom_point(shape=18)
```

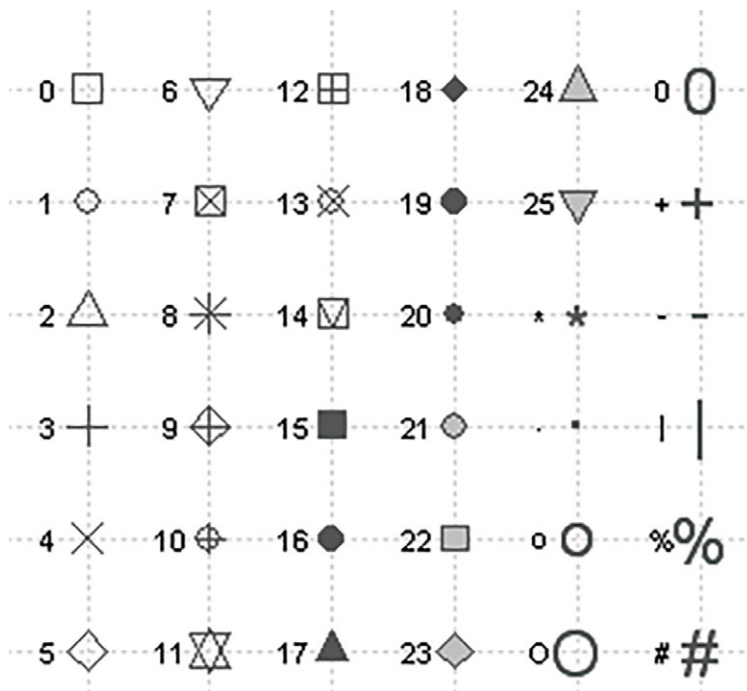


Figure 3.8 The different shapes of points in **R**

The number or symbol to the left determines the shape to be used. The image source is from the Quick-R website (Kabacoff, 2017), Retrieved May 1, 2019.

In this case, `shape` is not an aesthetics, now the shape is fixed and not depending on a variable. Keep in mind that dots are good for location, a property suitable for numerical and categorical data which is fastly interpreted by average humans (Mackinlay, 1986).

- **Lines** are unidimensional objects. They mainly represent **distance**, so they are a clear option for **numerical** values; if they are to be used for categorical values, make sure the audience will not interpret them as distance values. They can also serve to:
 - _ represent direction (with or without arrow tips) and slope
 - _ represent linear relationships
 - _ represent angles (two lines needed).

Let me add a line to Figure 3.6 with `geom_smooth`.

I have used a line in Figure 3.9 to encode a linear relationship captured from the cloud of points. The slope informs a positive relationship when read from left to right. Notice you can change thickness to improve default visibility (using `size`), but avoid doing so for encoding variable values. I have also set the `se` argument to `FALSE`, in order to avoid the display of confidence intervals around the line.

A particular line type is a curve: while a line represents a collection of points following a straight pattern, the curve represents dots moving more “freely” (see Figure 3.10).

I have used `geom_smooth` because it needs few parameters to produce a line, and it needs the original data frame. There are other “geoms” you may be interested in, some of which are shown later:

- _ Lines: `geom_line`, `geom_path`, `geom_segment`.
- _ Curves: `geom_curve`, `geom_smooth`.
- **Polygons** are two-dimensional objects. They are made out of combining at least three line segments. You will mostly deal with some common quadrilaterals (such as squares or rectangles), or common closed curves (i.e. a circle or an ellipse); but when plotting maps you will run into very irregular polygons. In general, you will need polygons to represent **area** or **size**. Size is a continuous value, so it is a possible option for **numerical** values instead of categorical ones. However, **areas are not easily interpreted**, or much less easy to decode than location, unless there are huge differences.

```

> info=ggplot(demo, aes(x=Culture, y=Functioning))
> dots1=info + geom_point(shape="*", size=4)
> lines1=dots1 + geom_smooth(method = lm, se=FALSE, colour="black")
> # result
> lines1

```

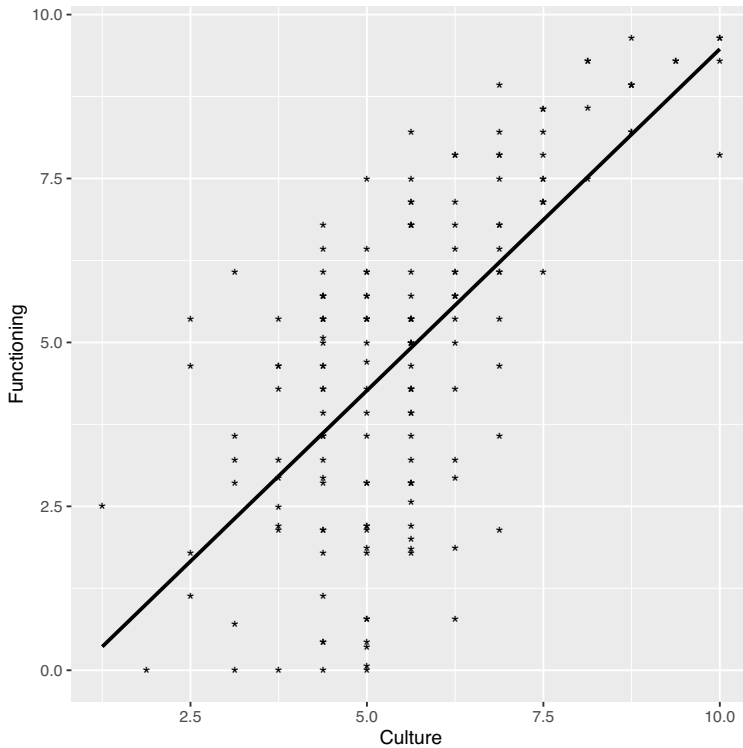


Figure 3.9 Using lines

The line is computed using the simple regression (lm method). Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

Figure 3.5 uses bars, a polygon whose height represents counts, so strictly speaking the area property has not been used. Let me modify Figure 3.6 to represent a size variable, as shown in Figure 3.11.

You can see that the default sizes offered are not that discernible with the exception of the extreme values.

```
> lines2=dots1 + geom_smooth(se=FALSE,size=3,colour="black")
> # result
> lines2
```

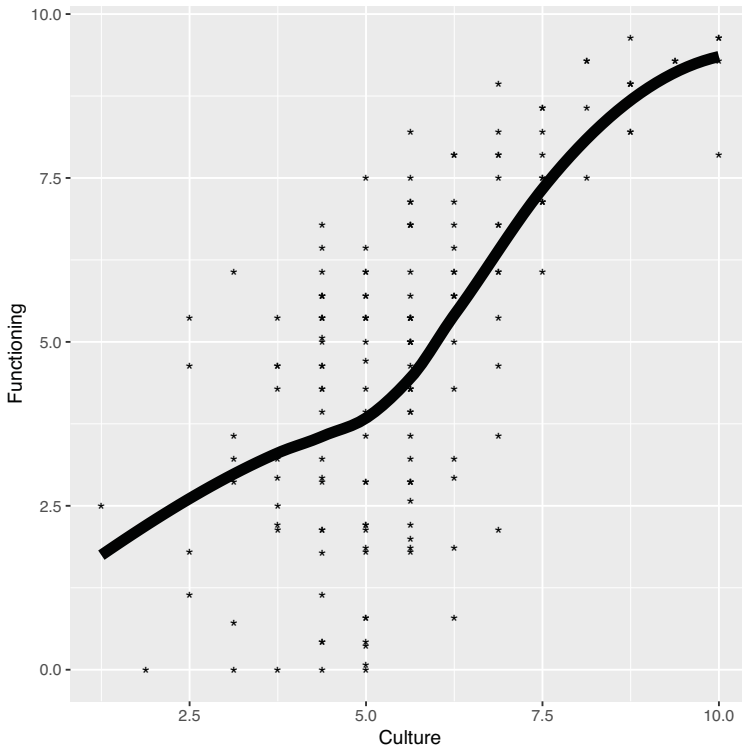


Figure 3.10 Using curves

The line is computed using a local polynomial regression (Fox and Weisberg, 2019), the default method for `geom_smooth`. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

3.1.3 Color

My first advice will be to think in black and white, or gray scale. Considering the possibility of a multicultural audience, you need to keep calm and use color wisely, as color may be the first cause to confusion due to cultural factors (Kroulek, 2016). Then, consider the following when using colors:

- Nominal values. In this case, you use colors to differentiate elements. Avoid any color combination that may induce the reader to see ordering or

```

> info=ggplot(demo, aes(x=Culture, y=Functioning, size=Regime))
> polygl=info + geom_point(shape=23)
> # result
> polygl

```

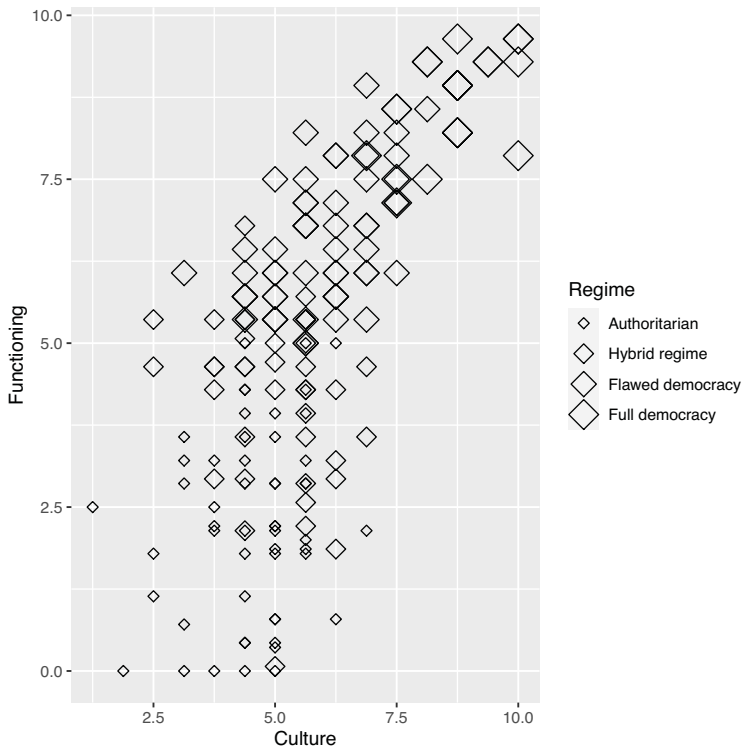


Figure 3.11 Using polygons

Here, the bigger the rhombi sizes the more democratic the country. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

intensity. Let me use the `color` aesthetics to produce object `colorNom1` (see the output in Figure 3.12):

Using basic colors, or *hues*, (such as blue, red, green) is the best option; you can also use any other set of hues or ‘qualitative schemes’ (Brewer, 1999), which differentiate nominal values. Notice that there are schemes available in case of colorblindness for the nominal case, as offered by Brewer (2009)¹.

¹ <http://colorbrewer2.org>

```
> info=ggplot(demo, aes(x=Culture, y=Functioning, colour=Continent))
> colorNom1=info+geom_point(size=3)
> colorNom1
```

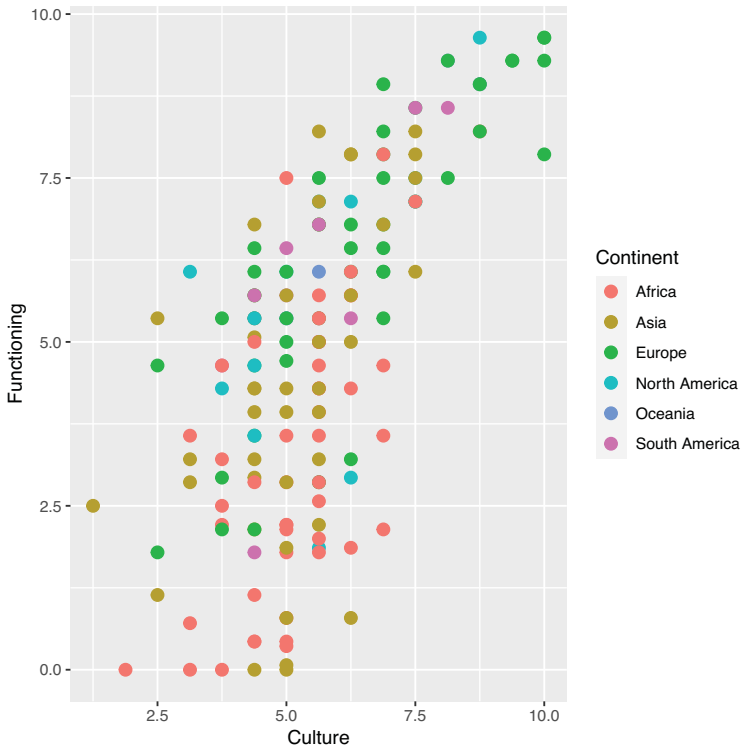


Figure 3.12 Color and nominal data

The colors must not reflect order. Color chosen by default. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

- **Ordinal values.** In this case you do need to show some ordering. In general, you can use one *hue*, and play with different levels of lightness or illumination, for example, from light orange to dark orange. This is also called a sequential scheme in Brewer (1999). If using just black and white for polygons, you can opt for different gray levels. For sure, if you have many ordinal levels, the color of each level may become difficult to differentiate. Let me use values in *Regime* as ordinal values in the *color* aesthetic to create the object `colorOrd1`; as I am not customizing the color, *ggplot* will use a multi-hue sequence (see the result in Figure 3.13).

```

> info=ggplot(demo, aes(x=Culture, y=Functioning,
+                       color=as.ordered(Regime)))
> colorOrd1=info+geom_point()
> colorOrd1

```

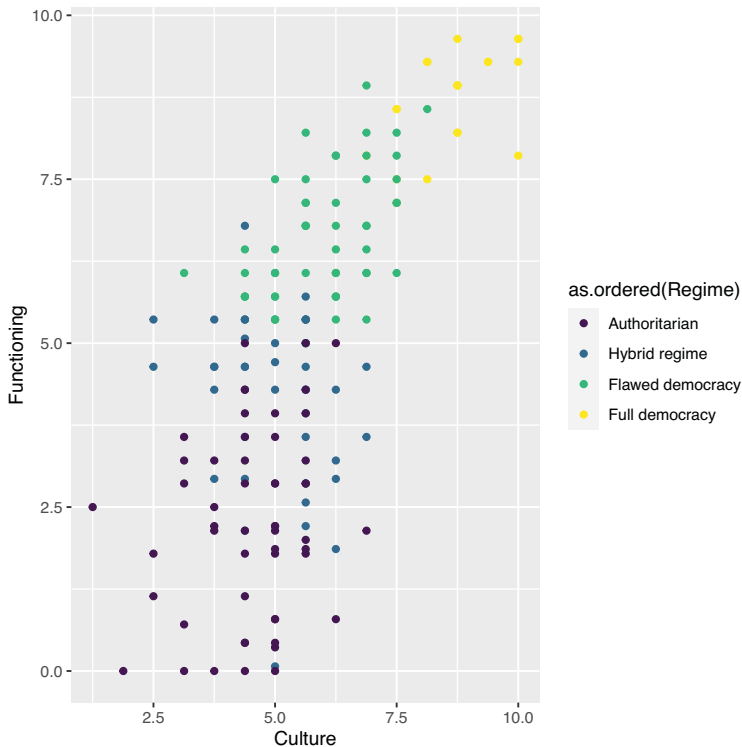


Figure 3.13 Color and ordinal data

The argument `colour` needed a numeric value to show ordering. A multi-hue sequential palette is chosen by default. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

- **Numeric values.** Let me use the numerical variable `Electoral` from the data we have been using in this chapter (The Economist Intelligence Unit, 2019) to create object `colorNum1`. When you use numerical values in the color aesthetic you will get Figure 3.13, where you see a continuous shade of a particular hue, whose lightness depends on the maximum and minimum value of the variable.

Sometimes you may need to organize your numeric values into intervals which are in fact interpreted as ordinal values, and, in that situation, you can make use


```
> info=ggplot(demo, aes(x=Culture, y=Functioning, colour=Electoral))
> colorNum1=info+geom_point(size=3)
> colorNum1
```

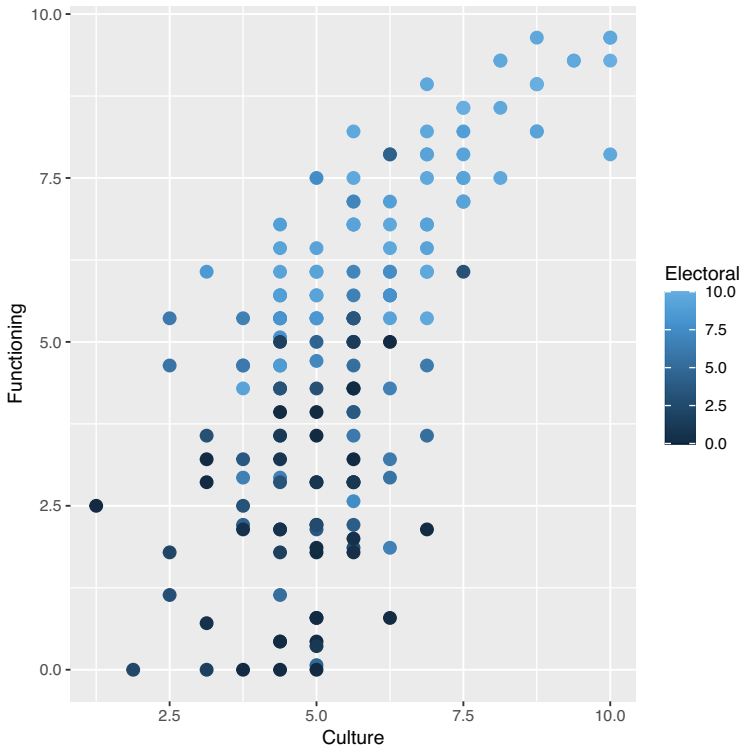


Figure 3.14 Color and numerical data

Color chosen by default. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

of the sequential scheme. Also, as numeric values can have zero and negative values, the intervals can combine two hues into a diverging scheme, as noted by Brewer (1999) (see Figure 3.19).

3.2 Beyond Default

I have kept the defaults on the previous plots, only using some function arguments when strictly needed; I did that to save me time and make your learning easier. However, as you suspect, I could have done a better job, but

that demands more work too. In this section, I will show you how to change some defaults while giving you more work.

3.2.1 The Brewer Palettes

Since finding an effective set of colors can be time-consuming, Geographer Cynthia Brewer organized a portal to help us choose palettes (Brewer, 2009).

Figure 3.15 shows you how to choose a qualitative palette: *Set1*. Let's **add** the suggested palette to change Figure 3.12 into Figure 3.16.

Figure 3.16 just added `scale_colour_brewer` to Figure 3.12 to change colors that represent variable *Continent*; however, Figure 3.15 also tells you that this palette may not be colorblind safe (notice the question mark on the eye icon), and that it is not photocopy safe (notice the X-mark); but, if you needed less colors this may change. From Figure 3.15, you should first select how many colors you need; then the nature of the palette (qualitative for nominal, and the others for ordinal or numerical). Alternatively, you can request the palette covers some properties (colorblind safe, etc.).

Figure 3.13 shows a sequence of colors, but I had to turn the ordinal variable into a number. If I use a sequential Brewer palette, I can keep the original data type.

Figure 3.17 follows the same strategy as Figure 3.16; however, the lighter colors are difficult to see. The smart move here will be to color the border of the dots. That little change requires:



- In `geom_point`:
 - Use a `shape` greater than **20** (these have borders).
- In `aes`:
 - Change the aesthetics from `colour` to `fill`.
- Use `scale_fill_brewer` instead of `scale_color_brewer`.


See the result of those changes in Figure 3.18.

Finally, let me redo Figure 3.14. If we use a diverging scheme, the middle value of the variable will be the lightest color (see Figure 3.19).

Figure 3.19 and Figure 3.17 color the objects using a similar strategy, using the `fill` argument (not `color`); however, to have more control over the numerical data, I used a different function from **ggplot**: `scale_fill_gradient2` instead. This function is specific for divergent schemes².

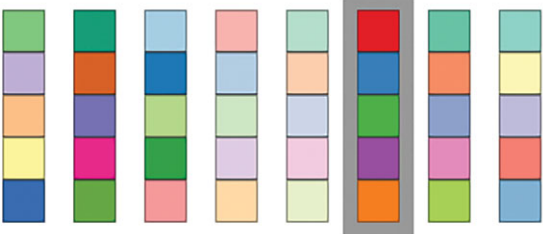
² Do not confuse this function with `scale_fill_gradient`, which helps produce sequential color schemes, and which can actually be an alternative to the function `scale_fill_brewer` that I used to make Figure 3.18.


Number of data classes: 6  

Nature of your data: 

☐ sequential ☐ diverging ☒ qualitative

Pick a color scheme:





Only show: 


☐ colorblind safe


☐ print friendly

☐ photocopy safe


Context: 

☐ roads 


☐ cities 





☐ borders 


Background:

☒ solid color 

☐ terrain

6-class Set1 

HEX 







	#e41a1c
	#377eb8
	#4daf4a
	#984ea3
	#ff7f00
	#ffff33

Figure 3.15 Brewer color selection

The arrow signals palette name. Image captured from ColorBrewer website (Brewer, 2009).

```

> colorNom2=colorNom1 + scale_colour_brewer(palette = "Set1")
> #result:
> colorNom2

```

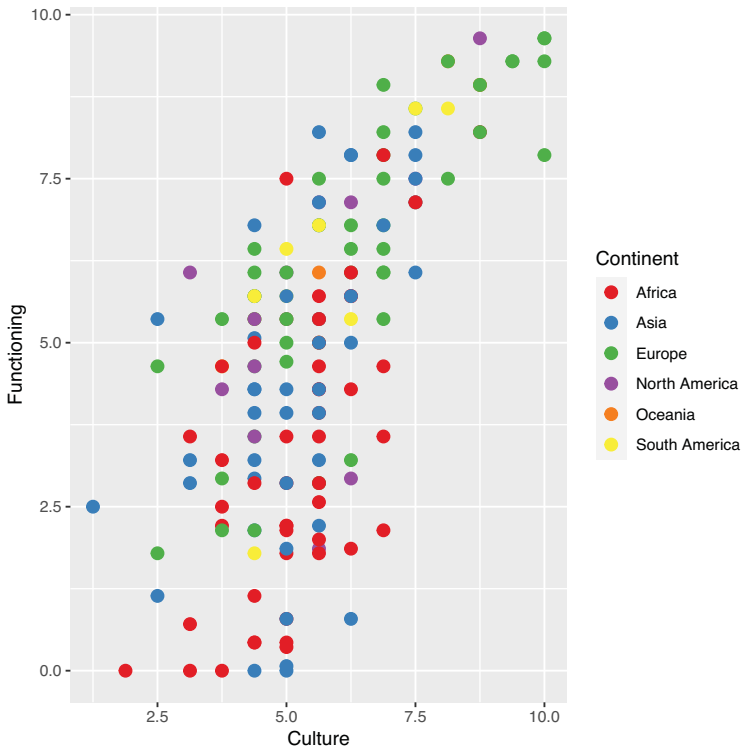


Figure 3.16 Brewer palette for nominal data

Color chosen is **Set1** (Brewer, 2009). Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

There is this function `scale_fill_distiller`, which I could have used to accomplished something similar, as it accepts the scheme **PuOr** as an argument for the **palette** argument. You can try this code:

```

> info4=ggplot(demo, aes(x=Culture, y=Functioning, fill=Electoral))
> colorNum=info4+geom_point(size=3, shape=21)
> # no midpoint
> colorNum + scale_fill_distiller(palette = "PuOr", direction = -1)

```

After runing the code, you will get a similar plot to Figure 3.19; however, `scale_fill_distiller` does not allow for the selection of a middle value, it just picks the actual value in the middle. On the other hand,

```

> info2=ggplot(demo, aes(x=Culture, y=Functioning, colour=Regime))
> colorOrd2=info2+geom_point(size=3)
> # direction -1 will show the scheme in the inverse order.
> colorOrd2 + scale_color_brewer(palette = "OrRd", direction = 1)

```

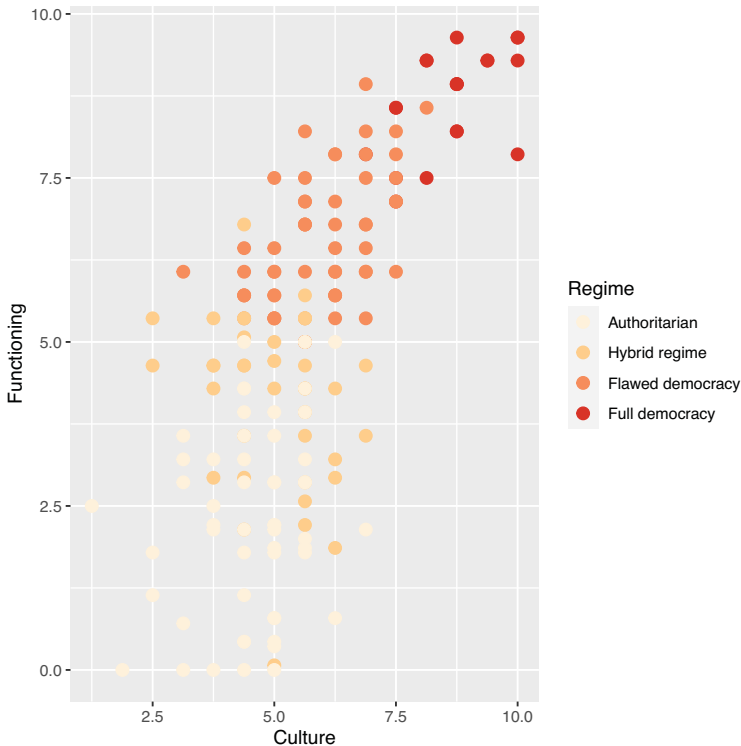


Figure 3.17 Brewer palette for ordinal data

The colors must reflect order. Brewer palette chosen is **OrRd**. Notice this scheme is colorblind safe, print friendly, and photocopy safe (Brewer, 2009). Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

even though `scale_fill_gradient2` does not have the argument for palette, it does allow you to select the middle value of the palette (and its color using `mid`). The downside of `scale_fill_gradient2`, however, is that you need to manually provide the *hues* for the low and high arguments. In this case, since I wanted to use the **PuOr** palette, I had to visit Brewer's webpage (Brewer, 2009) and select a diverging scheme, so that I could see the first and last *hues* from the palette; in this case, I just copied the hexadecimal

```

> # fill in aes
> info3=ggplot(demo, aes(x=Culture, y=Functioning, fill=Regime))
> # shape > 20
> colorOrd3=info3+geom_point(size=3, shape=21)
> # different function
> colorOrd3 + scale_fill_brewer(palette = "OrRd")

```

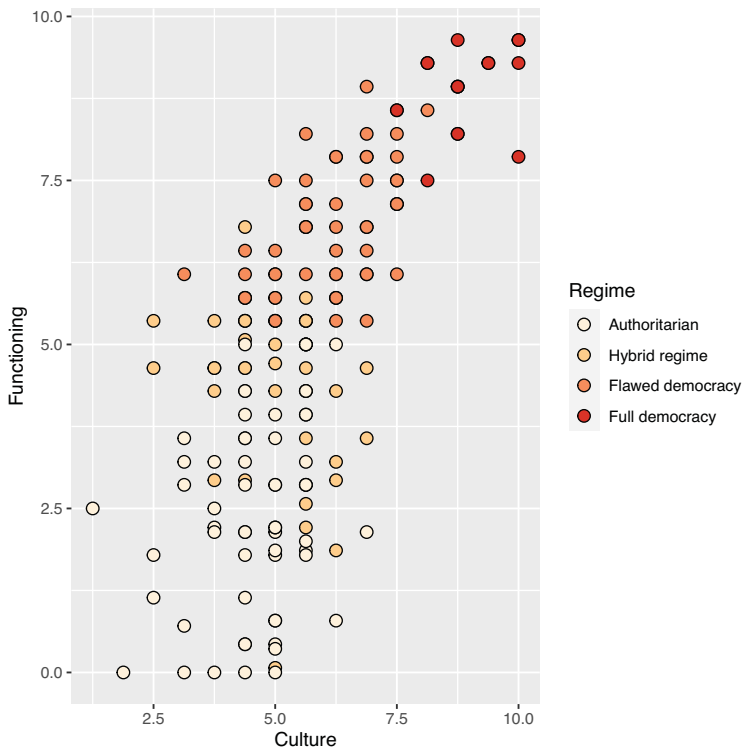


Figure 3.18 Fixing color for ordinal data

Code for Figure 3.17 was changed to improve the visibility of objects. Color chosen from Brewer **OrRd** (Brewer, 2009). Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

values and paste them into the code. In `scale_fill_distiller` you can also set the direction of the colors, but in `scale_fill_gradient2` you simply assign the colors of the limits.

You must understand that the use of color is not for decorating your images, it is for information. If you are not under control, your readers may get a message different from the one you intend for them.

```

> info4=ggplot(demo, aes(x=Culture, y=Functioning, fill=Electoral))
> colorNum=info4+geom_point(size=3, shape=21)
> # new function with midpoint
> colorNum2=colorNum + scale_fill_gradient2(midpoint = 5,
+                                           mid= 'white',
+                                           low = '#e66101',
+                                           high = '#5e3c99')
> # result
> colorNum2

```

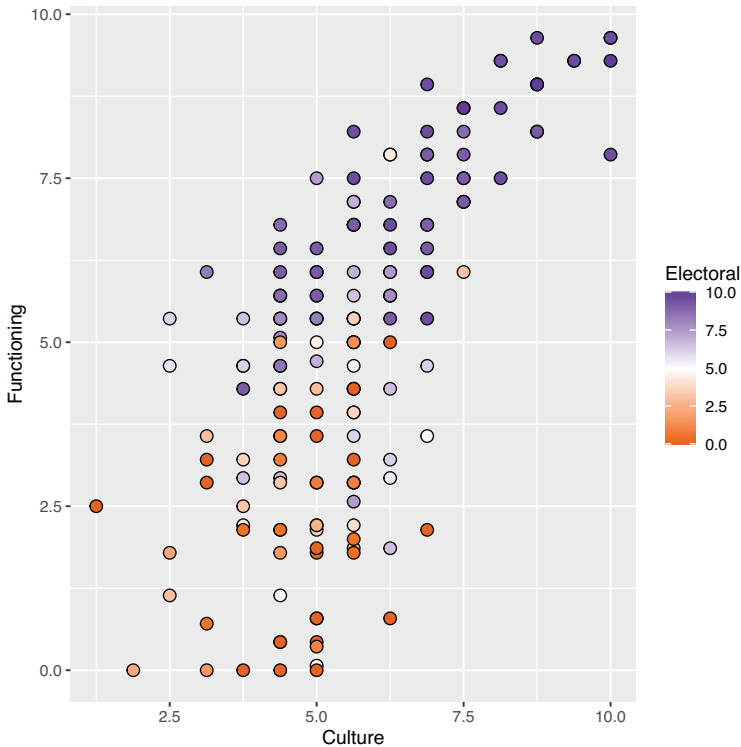


Figure 3.19 Color and numerical data

The colors must reflect order. The low and high colors are chosen from Brewer palette **PuOr** (Brewer, 2009). Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

3.2.2 The Background

Most plots have default backgrounds. In the previous examples, you have seen the default background for **ggplot**, which has a light grey plotting zone and a grid. The option for background should maximize “data–ink ratio”,

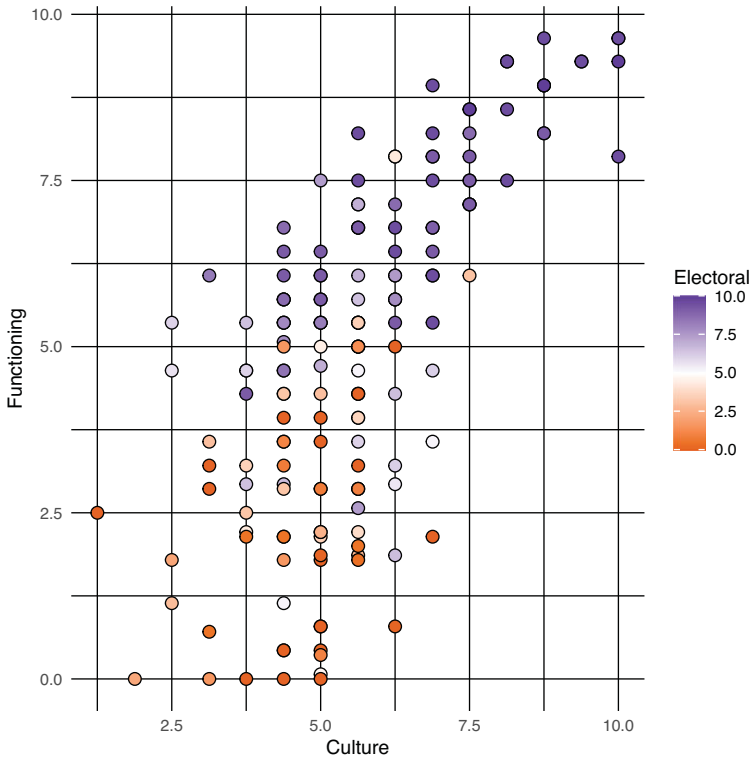


Figure 3.20 Minimal theme

Using **ggplot**'s `theme_minimal` (Wickham et al., 2019a) to try maximizing data-ink ratio.
Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

according to Tufte (2001). Now, this code can be an option for the Figure 3.19 background:

```
> colorBack1a=colorNum2 + theme_minimal()
```

The object `colorNum2` was holding Figure 3.19, so I just added `theme_minimal` to get Figure 3.20. Do you consider that respects Tufte's advice?

Using `theme_minimal` turned the gray background from Figure 3.19 into a white one, while turning the white grid lines into black ones. It also got rid of the axes lines. Notice that the grid lines are of two types: the primary, or the major, and the secondary, or the minor. Also notice that major grid lines are thicker than minor ones. Will Tufte be happy with this version? The presence

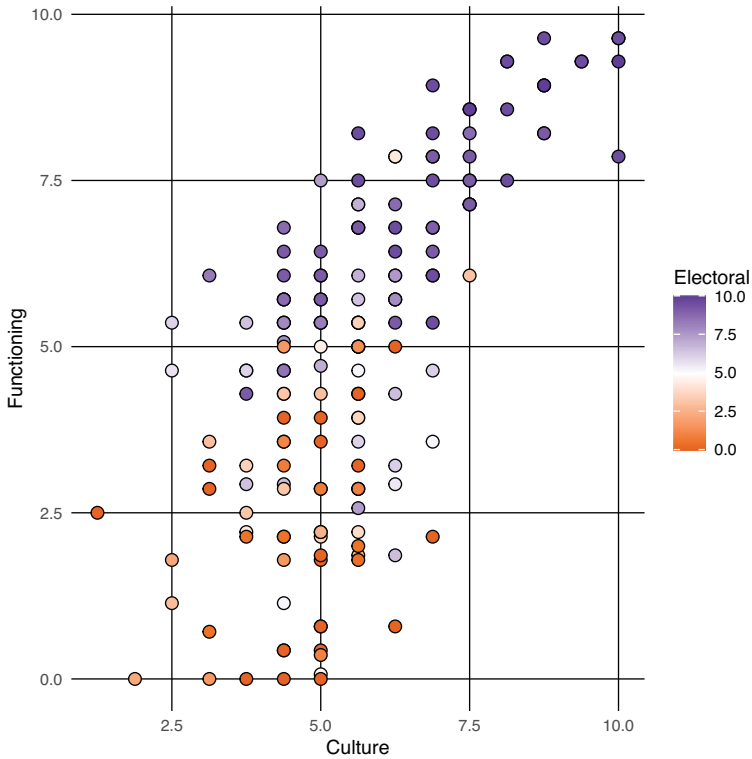


Figure 3.21 Minimal theme without minor grid

Using **ggplot**'s `theme_minimal` (Wickham et al., 2019a) to try maximizing data-ink ratio. This plot gets rid of minor grid lines by adding a theme layer and modifying the `panel.grid.minor` argument with the value `element_blank()`. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

of grid lines might disturb Tufte's followers, as in this case they are more an annotation than actual information. Let me show you how to get rid of the minor grid lines:

```
> colorBack1a + theme(panel.grid.minor = element_blank())
```

That code will produce Figure 3.21. The use of an extra theme layer is recommended when you want to change details of the theme at work. In this case, you are changing the properties of the minor grid lines.

If you still want no grid lines at all, you can keep using theme, but a better option will be to use `theme_classic`. This theme offers no grid lines, and

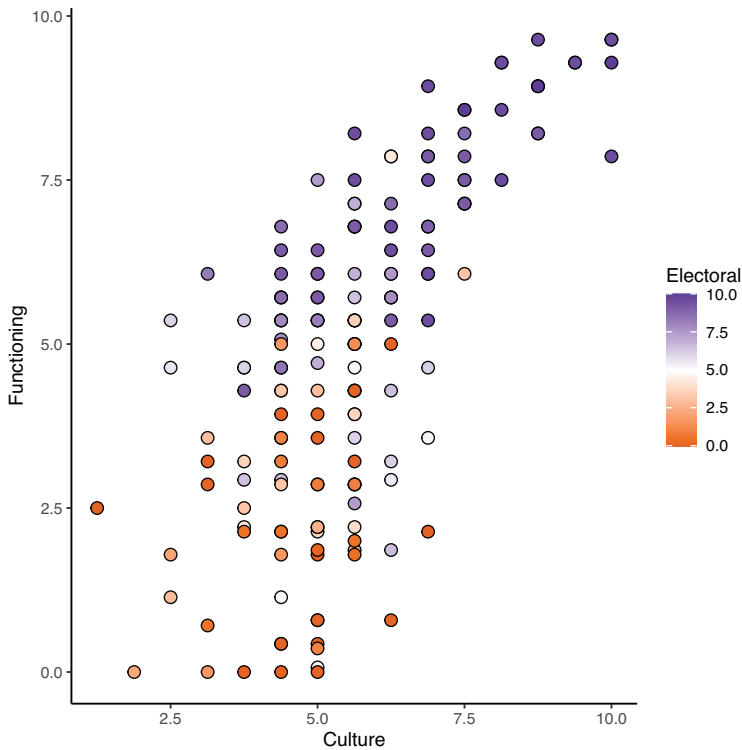


Figure 3.22 Classic theme

Using **ggplot**'s `theme_classic` (Wickham et al., 2019a) to try maximizing data-ink ratio.
Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

no grey background; however, it will make axes lines come back. The code is simply:

```
> colorBack1b=colorNum2 + theme_classic()
```

The new object `colorBack1b` is holding the latest plot, which you can see in Figure 3.22.

If you want a plot using `theme_classic` but without the axes lines you again use `theme`. You just need to set its `axis.line` argument with the property `element_blank()`, like this:

```
> colorBack1b + theme(axis.line = element_blank())
```

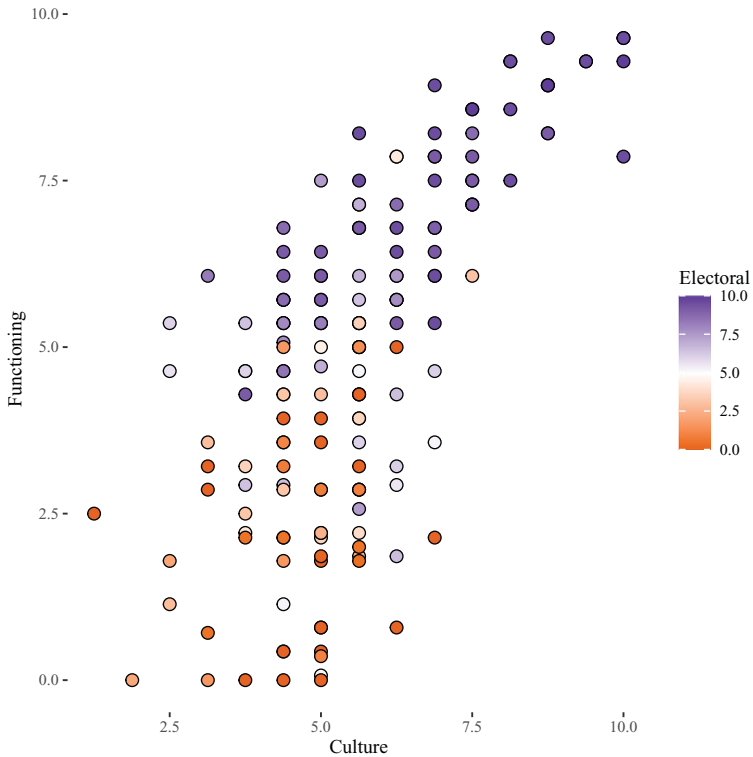


Figure 3.23 Tufte's theme

Using `ggthemes`'s `theme_tufte` (Arnold et al., 2019) to try maximizing "data-ink ratio".
Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

You may consider installing a particular package named **ggthemes** (Arnold et al., 2019) which has several themes that can be used in any *ggplot* object just by adding it like any other **ggplot** function. This package offers a theme named **tufte**:

```
> library(ggthemes)
> colorBack2a = colorNum2 + theme_tufte()
```

Now, the object `colorBack2a` has the looks Tufte required, according to **ggthemes** (Arnold et al., 2019). This new plot object is shown in Figure 3.23.

As mentioned, the package *ggthemes* offers several other themes that resemble the default themes in Stata, Tableau and others (*The Economist*, *Wall Street*, etc.).

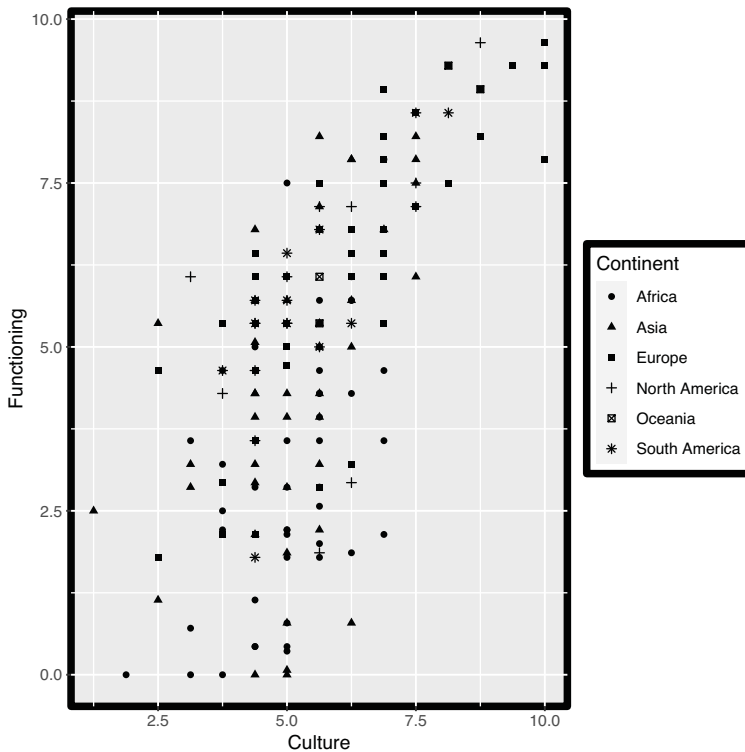


Figure 3.24 Default margins highlighted

Borders have been thickened. Data from Index of Democracy, in The Economist Intelligence Unit, 2019.

3.2.3 Margins

The plot and the legend have margins, and they are within other margins. Let me use this code to alter the object `leyenda` to highlight the margins in Figure 3.6:

```
> # changes to border lines:
> newBorders=element_rect(colour = "black",size = 2)
> # highlighting borders :
> margins1= leyenda + theme(panel.background = newBorders,
+                             plot.background = newBorders,
+                             legend.background = newBorders)
```

In general, you should respect default margin setups. However, **ggplot**'s theme gave you the power to change these in Figure 3.24.

Next, let me alter the default margin values using:

```
> # new info:
> newMargins=margin(3, 3, 3, 3, "cm") #top, right,bottom, left
> #changing:
> margins2= margins1 + theme(plot.margin = newMargins)
```

The plot in object `margins2` is shown in Figure 3.25.

3.2.4 Erasing

Keeping our loyalty to data-ink maximization requires that we get rid of some redundant elements. Candidates to be erased are axes ticks, axes titles, and legend titles. Let me use the following code to get rid of those elements in Figure 3.4:

```
> erasel= source + theme_classic() +
+       theme(axis.title.x = element_blank(),
+             axis.line.x = element_blank(),
+             axis.ticks.x = element_blank())
```

```
> margins2
```

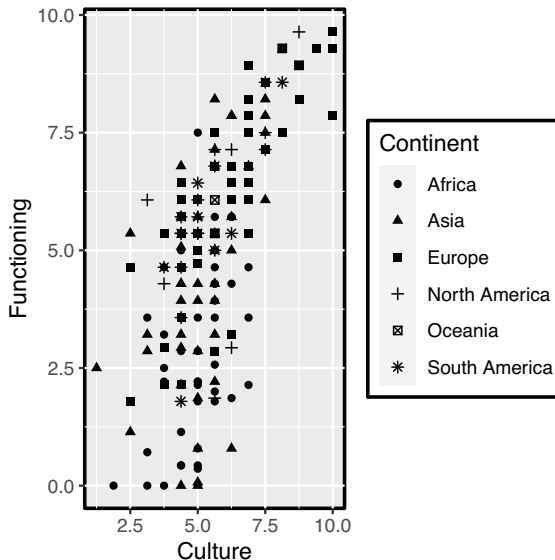


Figure 3.25 Altering margins

The main plot suffers more the consequences of altering margins, but not the legend. Consider that altering margins in one plot may considerably differentiate this from the other ones in your work. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

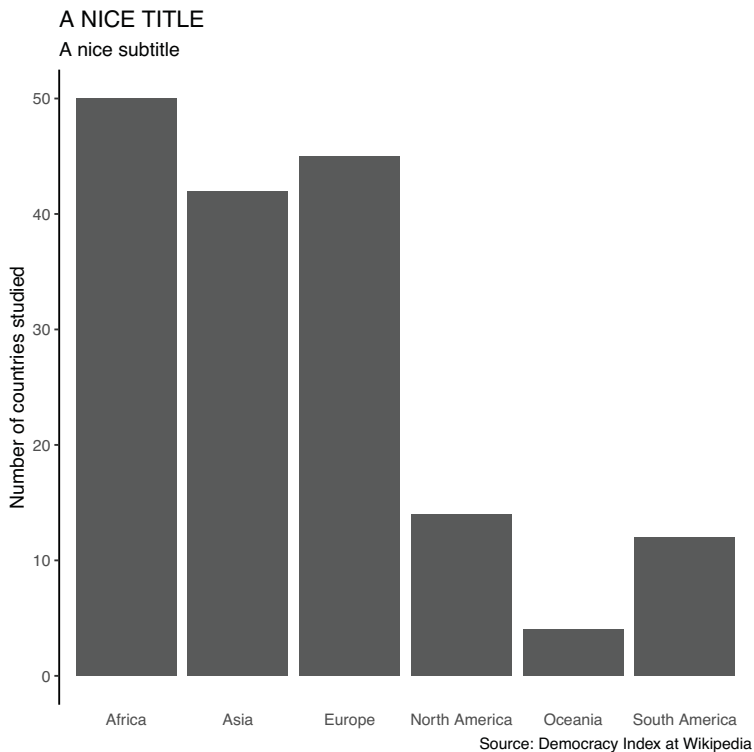


Figure 3.26 Erasing text elements in plot

The plot is erasing several elements using `element_blank`. The theme used is `theme_classic`. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

You can see object `erase1` in Figure 3.26.

The categorical nature of the data helps you support your decision to erase the axis line (it is not a continuity), and erase the tick marks (it is redundant if a label is present to clearly differentiate one bar from the other). Also, since the bars represent a small and complete set of elements (the continents), the axis title becomes redundant.

The theme element can help you get rid of legend titles as well. Let me do that in Figure 3.6:

```
> erase2 = leyenda + theme_classic() +
+           theme(legend.title = element_blank())
```

Object `erase2` is shown in Figure 3.27

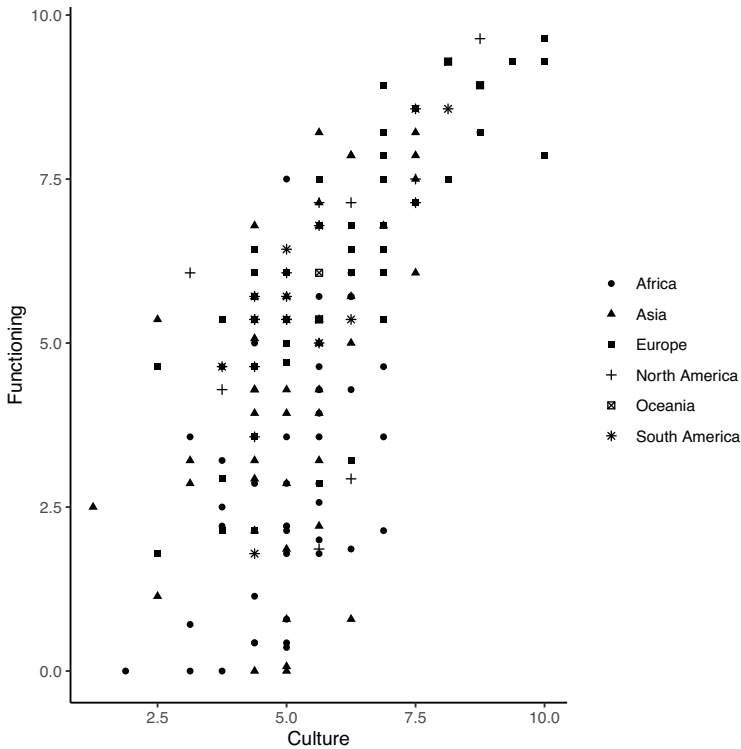


Figure 3.27 Erasing text elements in legend

The plot is erasing the legend title using `element_blank`. The theme used is `theme_classic`. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

Notice the fact I put `theme_classic` before the theme element in Figures 3.26 and 3.27. If it goes after, the changes in theme will have no effect.

3.2.5 Alignment

Let's modify the title, subtitle, source caption and axis titles alignment from Figure 3.26. This necessitates writing a value from zero to one in the argument `hjust` (zero is on the far left in the horizontal or the bottom on the vertical).

```
> ali1= erasel + theme(plot.title = element_text(hjust = 0.5),
+                       plot.subtitle = element_text(hjust = 0.5),
+                       plot.caption = element_text(hjust = 0),
+                       axis.title.y = element_text(hjust = 1))
```

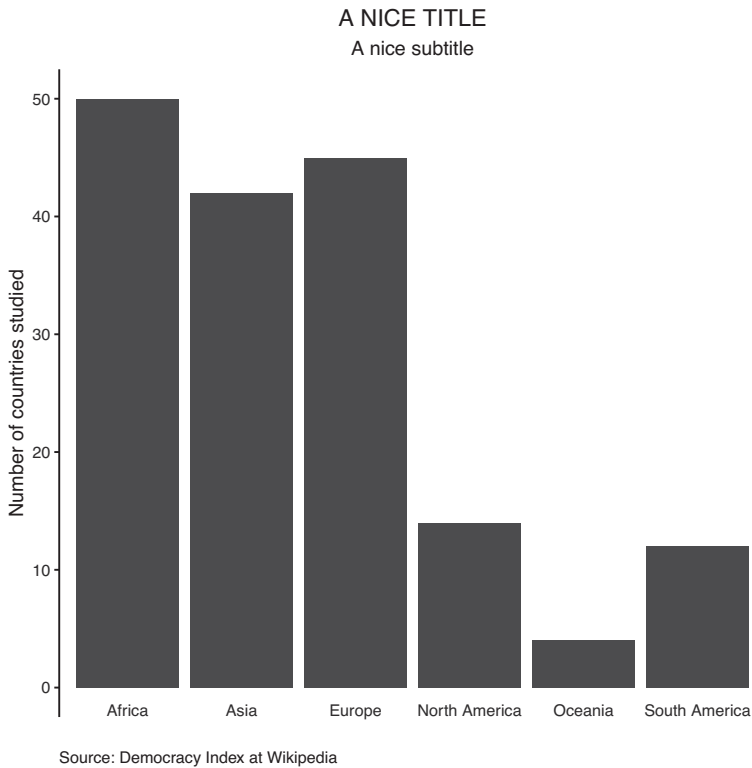


Figure 3.28 Altering text alignment

The theme used is `theme_classic`. This plot corrected the categorical labels position from Figure 3.26. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

```
> # categorical labels alignment correction!!
> alil= alil + theme(axis.text.x = element_text(vjust = 7))
```

Notice I used `vjust` to realign vertically the categorical labels, which remained far from their bars in Figure 3.26 after erasing (see Figure 3.28).

3.2.6 Repositioning

Changing the default position of some elements may improve understanding or take more advantage of the plotting space. Let me first prepare a **frequency table** as a data frame for the variable *Continent* we have previously used:

```
> ValuesAndCounts=table(demo$Continent)
> values=names(ValuesAndCounts)
```



```
> counts=as.vector(ValuesAndCounts)
> FT=data.frame(Values=values,Counts=counts)
> # ordering FreqTable by ascending counts
> FT=FT[order(FT$Counts),]
> row.names(FT)=NULL # resetting row names
> # here it is:
> print(FT, row.names = F)
```

Values	Counts
Oceania	4
South America	12
North America	14
Asia	42
Europe	45
Africa	50

I am ready to plot the FT object. Let's start by using the basic code:

```
> info=ggplot(data=FT, aes(x=Values,y=Counts))
> barFT1= info + geom_bar(stat = 'identity')
> barFT2= barFT1 + theme_classic()
```

Notice that I produced the barplot in Figure 3.1 using only the **x** argument in **aes** element; you can do that with data frames that do not represent *summaries*. Our current FT data is a frequency table (a summary of the data), so the **info** object used two arguments in its **aes** for the categories and their counts. When that is the case, you need to tell **geom_bar** to use the **y** “as it is”; so I set **stat** to “**identity**”. The object **barFT1** will produce the same as Figure 3.1. Then, I added the **theme_classic** element we have been using, and saved it all in **barFT2**.

Next, I will do some modifications to the **theme_classic**. Based on previous default changes (remember that the last line is moving the continent label upwards) I produced the object **barFT3**, which will look like Figure 3.28.

```
> barFT3 = barFT2 + theme(axis.title.x = element_blank(),
+                          axis.line.x = element_blank(),
+                          axis.ticks.x = element_blank(),
+                          axis.ticks.y = element_blank(),
+                          axis.text.x = element_text(vjust = 5))
```

After object **barFT3** is created, I can try some repositioning. My first try will be changing the column order in Figure 3.28. For that, I will use **scale_x_discrete**, whose **limits** argument will be set using the Continents ordered according to the our FT data frame:

```
> barFT4 = barFT3 + scale_x_discrete(limits=FT$Values)
```

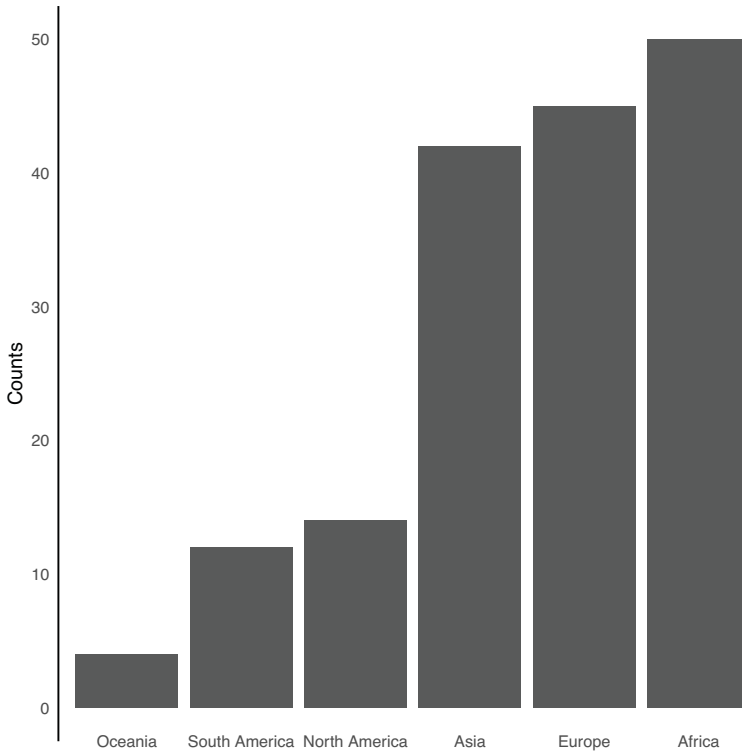


Figure 3.29 Changing order of bars

The theme used is `theme_classic`. This plot uses an ordered frequency table as data input. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

The modified barplot proposal is saved in object `barFT4`, and is shown in Figure 3.29.

What about annotating the bars in Figure 3.29, while dropping the y-axis values? We could use the `Counts` in the data frame as an `aes` in `geom_text`:

```
> barFT4 + geom_text(aes(label=Counts),
+                     vjust=1.6, # manual alignment
+                     color="white", size=3.5) +
+   theme(axis.text.y = element_blank())
```

Let's see the result from the code above in Figure 3.30.

My second try will be an alternative to Figure 3.30, using a vertical grid for the count values:

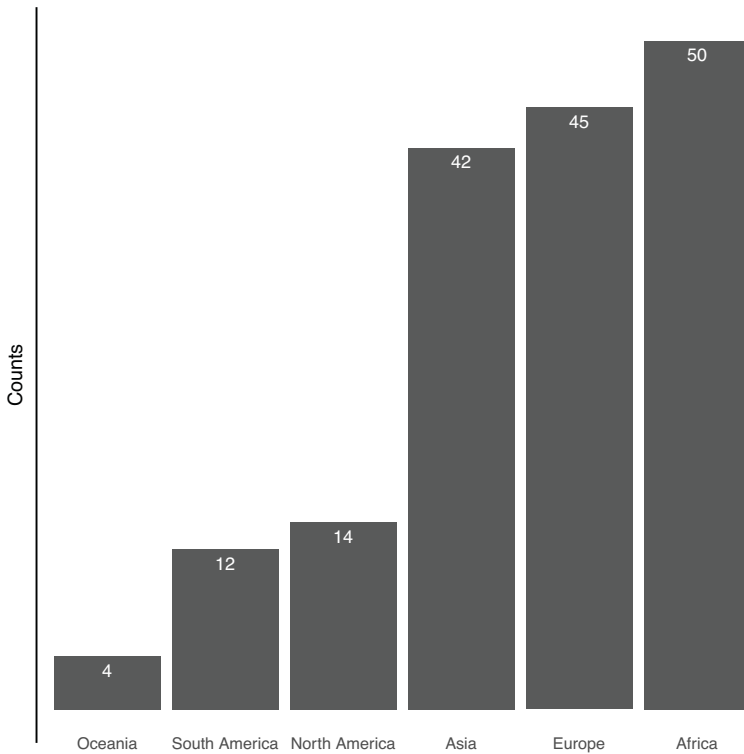


Figure 3.30 Position of horizontal bar with annotations

The theme used is `theme_classic`. Each bar was annotated, so redundant y-values were erased. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

```
> barFT4 + theme(panel.grid.major.y = element_line(color = "grey60")) +
+   scale_y_discrete(limits=FT$Counts)
```

Using `theme`, I requested a vertical grid with a dark grey color; and then, I instructed where the grid lines had to be, this time using `scale_y_discrete`. This alternative version can be seen in Figure 3.31.

Moving elements around gets easy as you become more and more familiar with the **ggplot** functions. However, that does not solve the problem. The more possibilities you have, the less clear becomes what the right combination to choose is; and even worse, it could complicate things by solving an unintended negative effect after you change a default³.

³ Remember that this happened in Section 3.2.4 when I erased the x-axis line, which caused the categorical labels be too far from the bars, and which was corrected later in Section 3.2.5.

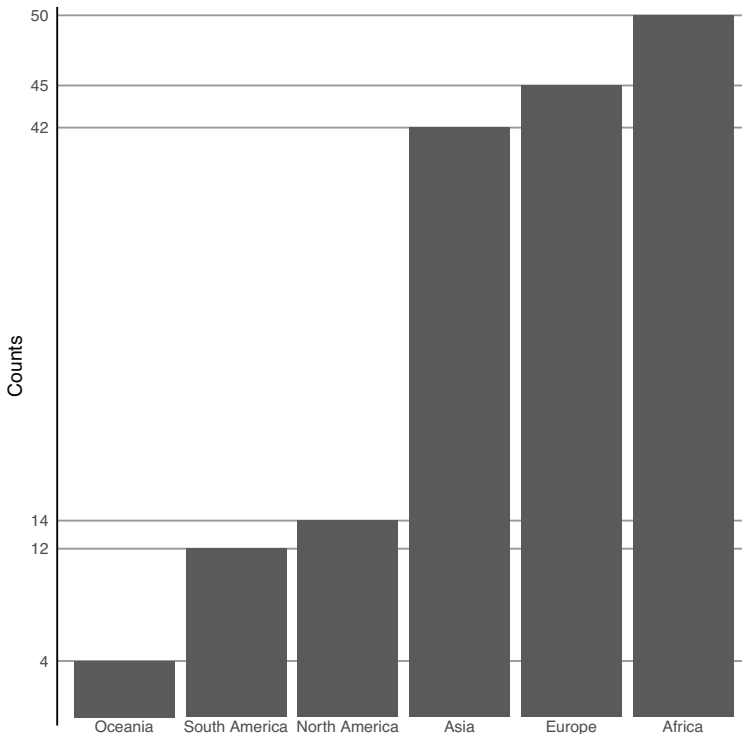


Figure 3.31 Position of vertical grid

The theme used is `theme_classic`. The grid serves to annotate frequency table plot. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

Let me show you a similar situation when repositioning the legend. This should not be hard at all:

```
> repol = erase2 + theme(legend.position="top")
> # you can try "bottom", too.
```

Object `repol` is a new version of Figure 3.27, which is shown in Figure 3.32, but if you want to make changes, you still have some further steps to follow.

For instance, in this (and all the previous plots), you may require that:

- One **measurement unit** in either axis has the same length.
- The legend should occupy less space.

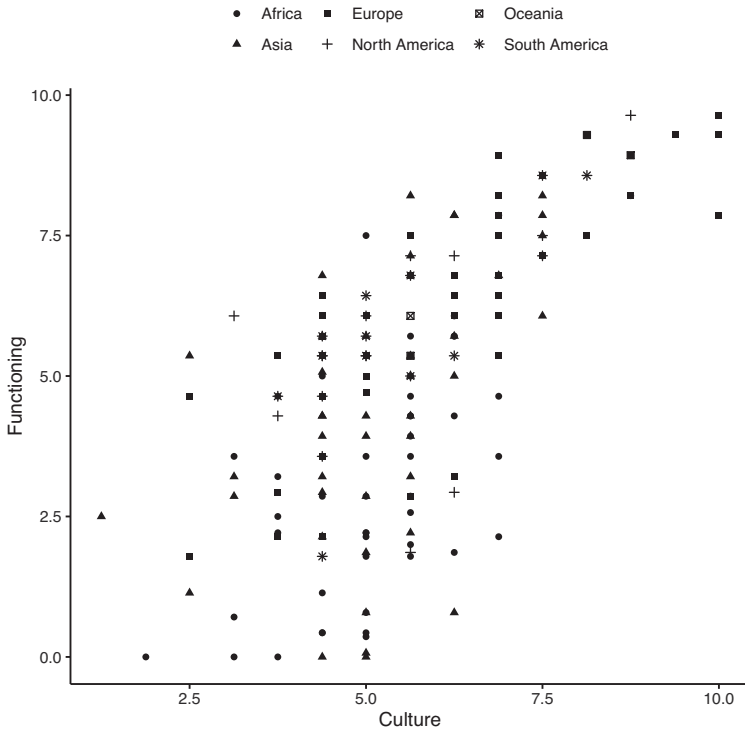


Figure 3.32 Position of legend

The theme used is `theme_classic` from Figure 3.27. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

Let's test this code, and see the result in Figure 3.33. Notice that the changes needed functions other than `theme` this time; and realize I used the argument `shape` because the legend represents shapes in the input `aes` from Figure 3.6.

```
> repo2 = repo1 +
+   # One unit same length in both axes
+   coord_fixed(ratio=1) +
+   # less space
+   guides(shape=guide_legend(nrow = 1))
```

When you see the output, you may consider that the legend occupies a smaller space, but you do not like that the legend elements are too far from each other; then, you can plan another change with this code:

```
> repo2 + theme(#legend elements no too far from each other
+   legend.key.width = unit(0, 'lines'),
+   # frame the legend
```

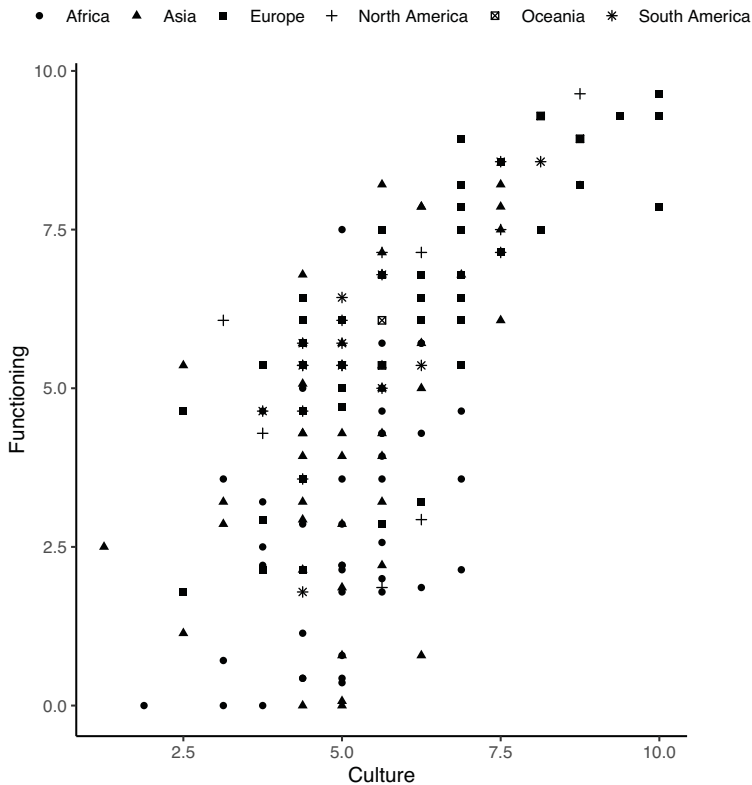


Figure 3.33 Position of legend (II)

Legend occupies just one row. By default, legend elements are a little far from each other in this case. The theme used is `theme_classic` from Figure 3.27. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

```
+ legend.background = element_rect(size=0.5,
+                                 linetype="solid",
+                                 colour="grey"))
```

Figure 3.34 is the resulting plot after using the previous code.

3.3 Python's Grammar of Graphics

Python has its own grammar of graphics thanks to the library **plotnine** created by Kibirige (2019). Let me call a data file following the same approach I used in the code on page 25:

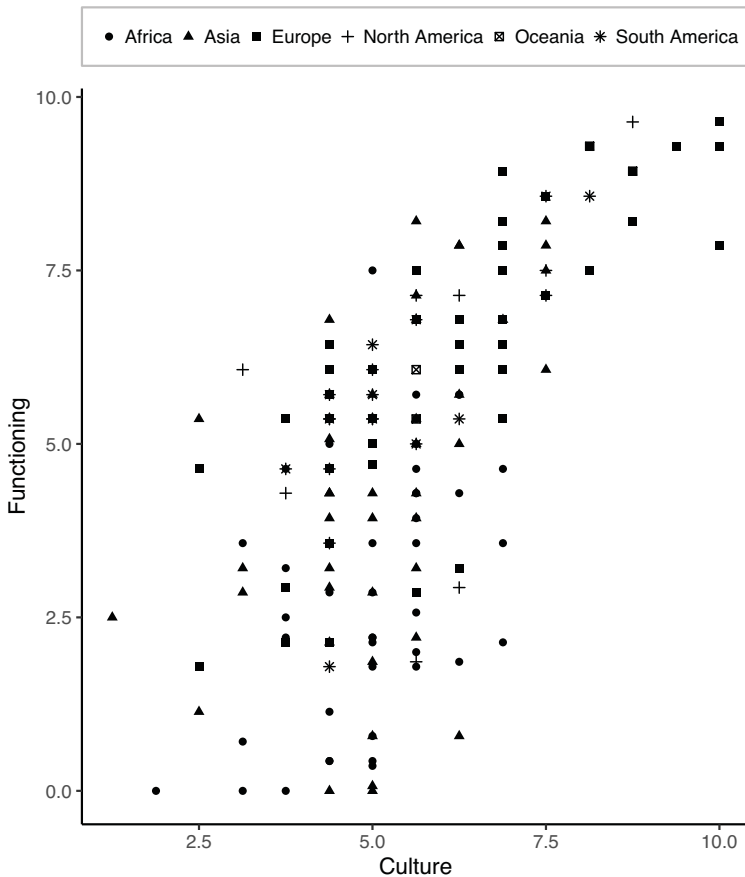


Figure 3.34 Position of legend (III)

Legend is in one row, elements are closer to each other, and the whole legend is in a frame. The theme used is `theme_classic` from Figure 3.27. Data from Index of Democracy, in Wikipedia, n.d., Retrieved June 1, 2019.

```
import pickle
from urllib.request import urlopen
linkRepo='https://github.com/resourcesbookvisual/data/'
linkDemo="raw/master/demo.pkl" # 'pickle' file!

demo = pickle.load(urlopen(linkRepo+linkDemo))
```

This time I opened a **pickle** file format. Like my example in **R**, a pickle file can keep the information on the data types (numeric, nominal, ordinal). The load function from pickle returns a **Pandas** data frame. Remember that

data frames are not a native structure in **Python**, so you need to have **Pandas** pre-installed.

Let me recreate the plots I showed you in this chapter using the *plotnine* library in **Python**, in all cases, I am using `theme_classic`:

- The next code recreates Figure 3.9 from page 35. Notice that *plotnine* is extremely similar to **R**'s **ggplot**. The main difference is that, in the `aes` section, you must write the name of the variables in quotations. Also, the argument for `method` has to be in quotations. Pay attention to the capitalization of `False` (or `True`), as **R** uses all letters in capitals (or just the first letter capitalized). Keep in mind that the symbols needed for dots depend on another set of values⁴.

```
from plotnine import *

info=ggplot(demo, aes(x='Culture', y='Functioning'))
dots1=info + geom_point(shape="*", size=2)
lines1=dots1 + geom_smooth(method = "lm",se=False,colour="black")
lines1 + theme_classic()
```

- Making some basic changes to the previous code, I recreate Figure 3.10 from page 36. Notice that the code is the same as in **R**.

```
lines2=dots1 + geom_smooth(se=False,colour="black")
lines2 + theme_classic()
```

- This time, I recreate Figure 3.11 from page 37. Notice that the code is the same as in **R**, just using a valid symbol for dots.

```
info=ggplot(demo,aes(x='Culture',y='Functioning',size='Regime'))
polyg1 = info + geom_point(shape = "x")
polyg1 + theme_classic()
```

- This code will produce Figure 3.12 from page 38. Besides the use of quotations, notice that `scale_colour_brewer` requires that you specify the `type` argument⁵. Pay attention to the line breaks. **R** will keep reading code even after a line break, as long as the plus symbol (+) is at the end of the line; that is not the case in **Python**: you need to add the backslash symbol (\) after +. Of course, as you can see, you do not need to do that when you have a line break inside a function (first two lines in the code below).

```
info=ggplot(demo,aes(x='Culture',y='Functioning',
                    colour='Continent'))
colorNom1=info+geom_point(size=3)
colorNom1 + scale_colour_brewer(type='qual',palette = "Set1") + \
    theme_classic()
```

⁴ Visit https://matplotlib.org/3.1.1/api/markers_api.html to see the available symbols.

⁵ The options are 'seq', 'div' and 'qual', being the first one the default.

- This code will produce Figure 3.18 from page 45.

```
info2=ggplot(demo,aes(x='Culture',y='Functioning',fill='Regime'))
colorOrd3 = info2 + geom_point(size=3,shape='o')
colorOrd3 + scale_fill_brewer(palette = "OrRd") + theme_classic()
```

- This code will produce Figure 3.19 from page 46.

```
info4=ggplot(demo,aes(x='Culture',y='Functioning',
                      fill='Electoral'))
colorNum=info4 + geom_point(size=3, shape='o')
colorNum + scale_fill_gradient2(midpoint = 5,
                                mid= 'white',
                                low = '#e66101',
                                high = '#5e3c99')
```

- I worked very hard on a bar plot to show you how to make changes, let me show you the main steps I will follow in **Python**:

1. Create a frequency table:

```
import pandas as pd

FT = pd.value_counts(demo.Continent,ascending=True).reset_index()
FT.columns = ['Values','Counts']
```

2. Create text for titles and annotations:

```
# Titles to be used:
the_Title="A NICE TITLE"
the_SubTitle="A nice subtitle"
TheTopTitles=the_Title+'\n'+the_SubTitle # adaptation
horizontalTitle="Continents present in the study"
verticalTitle="Number of countries studied"
# data for annotation
theCoordinates={'X':1,'Y':7} #dict instead of list
theMessage="So few?!"
```

Plotnine does not have functionality to show a subtitle, yet; so, I combined both in the title using line break (\n). Notice that the object `theCoordinates` uses a *dictionary* to store the coordinates of the annotation (I used a list in **R**).

3. Set up information for bar plot while changing defaults:

```
titles2= titles1 + xlab(horizontalTitle) + ylab(verticalTitle)

annot1= titles2 + annotate("text",
                          x = theCoordinates['X'], #reading dict
                          y = theCoordinates['Y'],
                          label = theMessage)

align1= annot1 + theme(plot_title = element_text(ha = "center"),
                       axis_title_y = element_text(ha = "top"))

barFT1= align1 + geom_bar(stat = 'identity')
barFT2= barFT1 + theme_classic()
barFT3= barFT2 + theme(axis_title_x = element_blank(),
                       axis_title_y = element_text(ha = "center"),
                       axis_ticks_major_x = element_blank(),
```

```
axis_ticks_major_y = element_blank(),
axis_text_x = element_text(va = 'top',
                           size=6),
plot_title = element_text(ha = "center"),
axis_line_x = element_blank())

barFT4 = barFT3 + scale_x_discrete(limits=FT.Values)
```

Notice some difference from R's **ggplot** in the code above:

- While some arguments in **ggplot** use a dot (i.e. `axis.title.x`), *plotnine* will use an underscore (`_`) instead.
- The arguments that control horizontal or vertical alignment are written as `hjust` or `vjust` in *ggplot*, while they are `ha` or `va`, respectively, in *plotnine*; besides, the arguments allowed in **ggplot** are numeric, while you need to write a specific location in *plotnine*⁶.

4. Now, producing Figure 3.30 from page 58.

```
barFT4 + geom_text(aes(label='Counts'),
                  va='top',
                  color="white", size=8) + \
  theme(axis_text_y = element_blank())
```

5. Now, producing Figure 3.31 from page 59.

```
barFT4 + theme(panel_grid_major_y=element_line(color="grey")) + \
  scale_y_continuous(breaks=FT.Counts)
```

- Let's code the **Python** version of Figure 3.34 on 62:

```
info=ggplot(demo, aes(x='Culture', y='Functioning',
                    shape='Continent'))
leyenda=info + geom_point()
erase2=leyenda + theme_classic() + \
  theme(legend_title=element_blank())
repo1=erase2 + theme(legend_position="top")
repo2=repo1 + coord_fixed(ratio=1) + \
  guides(shape=guide_legend(nrow=1))
repo2 + theme(legend_key_width = 0,
              legend_background = element_rect(size=0.5,
                                                  linetype="solid",
                                                  colour = "grey"))
```

Calling *matplotlib*

As *plotnine* does not have a subtitle and caption, let me finally invoke the main plotting library in **Python** : *matplotlib* (Caswell et al., 2019). Since *plotnine* uses *matplotlib*, we simply turn our *plotnine* visual into a *matplotlib* object:

⁶ These can be 'center', 'left', or 'right' for `ha`; or 'center', 'top', 'bottom', or 'baseline' in `va`.

```

## plotnine

info=ggplot(demo, aes(x='Culture', y='Functioning',
                      shape='Continent'))
leyenda=info + geom_point()
erase2=leyenda + theme_classic() + \
        theme(legend_title=element_blank())
repo2=erase2 + coord_fixed(ratio=1)
repo2=repo2 + theme(legend_key_width = 0,
                  legend_background = element_rect(size=0.5,
                                                    linetype="solid",
                                                    colour = "grey"))

# HELP from matplotlib
import matplotlib.pyplot as plt
fig = repo2.draw()
# x=0,y=0 is lower left corner; x=1,y=1 is upper right
fig.text(x=0.7,y=0.01,s="The Caption with matplotlib")
# supitle for TITLE
plt.suptitle(the_Title, y=0.95, fontsize=18)
# title for subTITLE
plt.title(the_SubTitle, fontsize=10)

```

Matplotlib is a huge plotting library, and all you needed here, as you just saw, is to find a way to get a *matplotlib* fig. After that, you can apply the missing functionalities.