pick values that are on the same scale as the variable you are mapping.

- Density estimates can also be drawn in two dimensions. The `geom_density_2d()` function draws contour lines estimating the joint distribution of two variables. Try it with the `midwest` data, for example, plotting percent below the poverty line (`percbelowpoverty`) against percent college-educated (`percollege`). Try it with and without a `geom_point()` layer.

5 Graph Tables, Add Labels, Make Notes

This chapter builds on the foundation we have laid down. Things will get a little more sophisticated in three ways. First, we will learn about how to transform data *before* we send it to ggplot to be turned into a figure. As we saw in chapter 4, ggplot's geoms will often summarize data for us. While convenient, this can sometimes be awkward or even a little opaque. Often it's better to get things into the right shape before we send anything to ggplot. This is a job for another tidyverse component, the `dplyr` library. We will learn how to use some of its "action verbs" to select, group, summarize, and transform our data.

Second, we will expand the number of geoms we know about and learn more about how to choose between them. The more we learn about ggplot's geoms, the easier it will be to pick the right one given the data we have and the visualization we want. As we learn about new geoms, we will also get more adventurous and depart from some of ggplot's default arguments and settings. We will learn how to reorder the variables displayed in our figures, and how to subset the data we use before we display it.

Third, this process of gradual customization will give us the opportunity to learn more about the scale, guide, and theme functions that we have mostly taken for granted until now. These will give us even more control over the content and appearance of our graphs. Together, these functions can be used to make plots much more legible to readers. They allow us to present our data in a more structured and easily comprehensible way, and to pick out the elements of it that are of particular interest. We will begin to use these methods to layer geoms on top of one another, a technique that will allow us to produce sophisticated graphs in a systematic, comprehensible way.

Our basic approach will not change. No matter how complex our plots get, or how many individual steps we take to layer and tweak their features, underneath we will always be doing the same thing. We want a table of tidy data, a mapping of variables to

aesthetic elements, and a particular type of graph. If you can keep sight of this, it will make it easier to confidently approach the job of getting any particular graph to look just right.

5.1 Use Pipes to Summarize Data

In chapter 4 we began making plots of the distributions and relative frequencies of variables. Cross-classifying one measure by another is one of the basic descriptive tasks in data analysis. Tables 5.1 and 5.2 show two common ways of summarizing our GSS data on the distribution of religious affiliation and region. Table 5.1 shows the column marginals, where the numbers sum to a hundred by column and show, e.g., the distribution of Protestants across regions. Meanwhile in table 5.2 the numbers sum to a hundred across the rows, showing, for example, the distribution of religious affiliations within any particular region.

We saw in chapter 4 that `geom_bar()` can plot both counts and relative frequencies depending on what we asked of it. In practice, though, letting the geoms (and their `stat_` functions) do the work can sometimes get a little confusing. It is too easy to lose track of whether one has calculated row margins, column margins, or

TABLE 5.1
Column marginals. (Numbers in columns sum to 100.)

	Protestant	Catholic	Jewish	None	Other	NA
Northeast	12	25	53	18	18	6
Midwest	24	27	6	25	21	28
South	47	25	22	27	31	61
West	17	24	20	29	30	6

TABLE 5.2
Row marginals. (Numbers in rows sum to 100.)

	Protestant	Catholic	Jewish	None	Other	NA
Northeast	32	33	6	23	6	0
Midwest	47	25	0	23	5	1
South	62	15	1	16	5	1
West	38	25	2	28	8	0

overall relative frequencies. The code to do the calculations on the fly ends up stuffed into the mapping function and can become hard to read. A better strategy is to calculate the frequency table you want first and then plot that table. This has the benefit of allowing you do to some quick sanity checks on your tables, to make sure you haven't made any errors.

Let's say we want a plot of the row marginals for religion within region. We will take the opportunity to do a little bit of data munging in order to get from our underlying table of GSS data to the summary tabulation that we want to plot. To do this we will use the tools provided by `dplyr`, a component of the tidyverse that provides functions for manipulating and reshaping tables of data on the fly. We start from our individual-level `gss_sm` data frame with its `bigregion` and `religion` variables. Our goal is a summary table with percentages of religious preferences grouped within region.

As shown schematically in figure 5.1, we will start with our individual-level table of about 2,500 GSS respondents. Then we want to summarize them into a new table that shows a count of each religious preference, grouped by region. Finally we will turn these within-region counts into percentages, where the denominator is the total number of respondents within each region. The `dplyr` library provides a few tools to make this easy and clear to read. We will use a special operator, `%>%`, to do our work. This is the *pipe* operator. It plays the role of the yellow triangle in figure 5.1, in that it helps us perform the actions that get us from one table to the next.

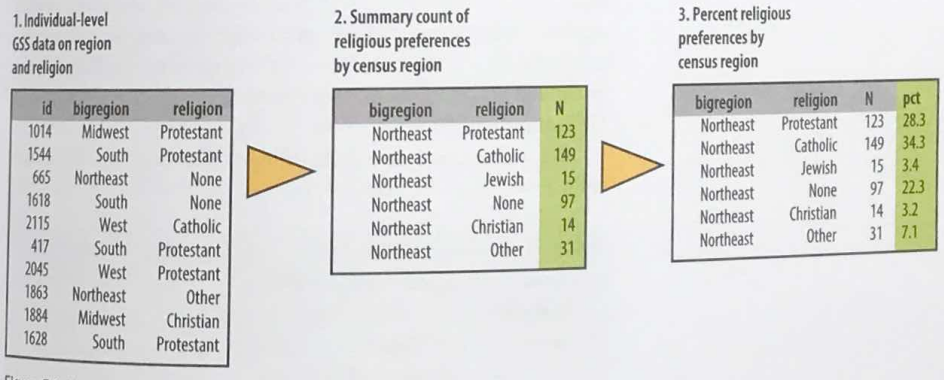

Figure 5.1: How we want to transform the individual-level data.

We have been building our plots in an *additive* fashion, starting with a `ggplot` object and layering on new elements. By analogy, think of the `%>%` operator as allowing us to start with a data frame and perform a *sequence* or *pipeline* of operations to turn it into another, usually smaller and more aggregated, table. Data goes in one side of the pipe, actions are performed via functions, and results come out the other. A pipeline is typically a series of operations that do one or more of four things:

`group_by()`

`filter()` rows; `select()` columns

`mutate()`

`summarize()`

- *Group* the data into the nested structure we want for our summary, such as "Religion by Region" or "Authors by Publications by Year."
- *Filter* or *select* pieces of the data by row, column, or both. This gets us the piece of the table we want to work on.
- *Mutate* the data by creating new variables at the *current* level of grouping. This adds new columns to the table without aggregating it.
- *Summarize* or aggregate the grouped data. This creates new variables at a *higher* level of grouping. For example we might calculate means with `mean()` or counts with `n()`. This results in a smaller, summary table, which we might further summarize or mutate if we want.

We use the `dplyr` functions `group_by()`, `filter()`, `select()`, `mutate()`, and `summarize()` to carry out these tasks within our pipeline. They are written in a way that allows them to be easily piped. That is, they understand how to take inputs from the left side of a pipe operator and pass results along through the right side of one. The dplyr documentation has some useful vignettes that introduce these grouping, filtering, selection, and transformation functions. There is also a more detailed discussion of these tools, along with many more examples, in Wickham & Grolemund (2016).

We will create a new table called `rel_by_region`. Here's the code:

```
rel_by_region ← gss_sm %>%
    group_by(bigregion, religion) %>%
    summarize(N = n()) %>%
    mutate(freq = N / sum(N),
           pct = round((freq*100), 0))
```


What are these lines doing? First, we are creating an object as usual, with the familiar assignment operator, `<-`. Next comes the pipeline. Read the objects and functions from left to right, with the pipe operator "`%>%`" connecting them together meaning "and then …." Objects on the left side "pass through" the pipe, and whatever is specified on the right of the pipe gets done to that object. The resulting object then passes through to the right again, and so on down to the end of the pipeline.

Reading from the left, the code says this:

- Create a new object, `rel_by_region`. It will get the result of the following sequence of actions: Start with the `gss_sm` data, and then — `rel_by_region ← gss_sm %>%`
- Group the rows by `bigregion` and, within that, by `religion`. — `group_by(bigregion, religion) %>%`
- Summarize this table to create a new, much smaller table, with three columns: `bigregion`, `religion`, and a new summary variable, `N`, that is a count of the number of observations within each religious group for each region. — `summarize(N = n()) %>%`
- With this new table, use the `N` variable to calculate two new columns: the relative proportion (`freq`) and percentage (`pct`) for each religious category, still grouped by region. Round the results to the nearest percentage point. — `mutate(freq = N / sum(N), pct = round((freq*100), 0))`

In this way of doing things, objects passed along the pipeline and the functions acting on them carry some assumptions about their context. For one thing, you don't have to keep specifying the name of the underlying data frame object you are working from. Everything is implicitly carried forward from `gss_sm`. Within the pipeline, the transient or implicit objects created from your summaries and other transformations are carried through, too.

Second, the `group_by()` function sets up how the grouped or nested data will be processed within the `summarize()` step. Any function used to create a new variable within `summarize()`, such as `mean()` or `sd()` or `n()`, will be applied to the *innermost* grouping level first. Grouping levels are named from left to right within `group_by()` from outermost to innermost. So the function call `summarize(N = n())` counts up the number of observations for each value of `religion` within `bigregion` and puts them in a new variable named `N`. As dplyr's functions see things, summarizing actions peel off one grouping level at a time, so that the resulting summaries are at the next level up. In this case, we start with

individual-level observations and group them by religion within region. The `summarize()` operation aggregates the individual observations to counts of the number of people affiliated with each religion, for each region.

Third, the `mutate()` step takes the `N` variable and uses it to create `freq`, the relative frequency for each subgroup within region, and finally `pct`, the relative frequency turned into a rounded percentage. These `mutate()` operations add or remove columns from tables but do not change the grouping level.

Inside both `mutate()` and `summarize()`, we are able to create new variables in a way that we have not seen before. Usually, when we see something like `name = value` inside a function, the `name` is a general, named argument and the function is expecting information from us about the specific value it should take. Normally if we give a function a named argument it doesn't know about (`aes(chuckles = year)`), it will ignore it, complain, or break. With `summarize()` and `mutate()`, however, we can invent named arguments. We are still assigning specific values to `N`, `freq`, and `pct`, but we pick the names, too. They are the names that the newly created variables in the summary table will have. The `summarize()` and `mutate()` functions do not need to know what they will be in advance.

As in the case of `aes(x = gdpPercap, y = lifeExp)`, for example.

Finally, when we use `mutate()` to create the `freq` variable, not only can we make up that name within the function, `mutate()` is also clever enough to let us *use* that name right away, on the next line of the same function call, when we create the `pct` variable. This means we do not have to repeatedly write separate `mutate()` calls for every new variable we want to create.

Our pipeline takes the `gss_sm` data frame, which has 2,867 rows and 32 columns, and transforms it into `rel_by_region`, a summary table with 24 rows and 5 columns that looks like this, in part:

```
rel_by_region
```

```
## # A tibble: 24 x 5
## # Groups:   bigregion [4]
##    bigregion religion       N    freq   pct
##    <fct>     <fct>      <int>   <dbl> <dbl>
##  1 Northeast Protestant   158 0.324     32.
##  2 Northeast Catholic     162 0.332     33.
##  3 Northeast Jewish        27 0.0553     6.
##  4 Northeast None         112 0.230     23.
##  5 Northeast Other         28 0.0574     6.
##  6 Northeast <NA>           1 0.00205    0.
##  7 Midwest   Protestant   325 0.468     47.
##  8 Midwest   Catholic     172 0.247     25.
##  9 Midwest   Jewish         3 0.00432    0.
## 10 Midwest   None         157 0.226     23.
## # ... with 14 more rows
```


The variables specified in `group_by()` are retained in the new summary table; the variables created with `summarize()` and `mutate()` are added, and all the other variables in the original dataset are dropped.

We said before that, when trying to grasp what each additive step in a `ggplot()` sequence does, it can be helpful to work backward, removing one piece at a time to see what the plot looks like when that step is not included. In the same way, when looking at pipelined code it can be helpful to start from the end of the line and then remove one "%>%" step at a time to see what the resulting intermediate object looks like. For instance, what if we remove the `mutate()` step from the code above? What does `rel_by_region` look like then? What if we remove the `summarize()` step? How big is the table returned at each step? What level of grouping is it at? What variables have been added or removed?

Plots that do not require sequential aggregation and transformation of the data before they are displayed are usually easy to write directly in ggplot, as the details of the layout are handled by a combination of mapping variables and layering geoms. One-step filtering or aggregation of the data (such as calculating a proportion, or a specific subset of observations) is also straightforward. But when the result we want to display is several steps removed from the data, and in particular when we want to group or aggregate a table and do some more calculations on the result before drawing anything, then it can make sense to use dplyr's tools to produce these summary tables first. This is true even if it would be possible to do it within a `ggplot()` call. In addition to making our code easier to read, pipes let us more easily perform sanity checks on our results, so that we are sure we have grouped and summarized things in the right order. For instance, if we have

done things properly with `rel_by_region`, the `pct` values associated with `religion` should sum to 100 within each region, perhaps with a bit of rounding error. We can quickly check this using a very short pipeline:

```
rel_by_region %>% group_by(bigregion) %>% summarize(total = sum(pct))
```

```
## # A tibble: 4 x 2
##   bigregion total
##   <fct>     <dbl>
## 1 Northeast  100.
## 2 Midwest    101.
## 3 South      100.
## 4 West       101.
```

This looks good. As before, now that we are working directly with percentage values in a summary table, we can use `geom_col()` instead of `geom_bar()`.

```
p <- ggplot(rel_by_region, aes(x = bigregion, y = pct, fill = religion))
p + geom_col(position = "dodge2") +
    labs(x = "Region",y = "Percent", fill = "Religion") +
    theme(legend.position = "top")
```

Try going back to the code for figure 4.13, in chapter 4, and using this "dodge2" argument instead of the "dodge" argument there.

We use a different `position` argument here, `dodge2` instead of `dodge`. This puts the bars side by side. When dealing with precomputed values in `geom_col()`, the default `position` is to make a proportionally stacked column chart. If you use `dodge` they will be stacked within columns, but the result will read incorrectly. Using `dodge2` puts the subcategories (religious affiliations) side-by-side within groups (regions).

The values in the bar chart in figure 5.2 are the percentage equivalents to the stacked counts in figure 4.10. Religious affiliations sum to 100 percent within region. The trouble is, although we now know how to cleanly produce frequency tables, this is still a bad figure! It is too crowded, with too many bars side by side. We can do better.

As a rule, dodged charts can be more cleanly expressed as faceted plots. Faceting removes the need for a legend and thus makes the chart simpler to read. We also introduce a new function. If we map religion to the x-axis, the labels will overlap and become

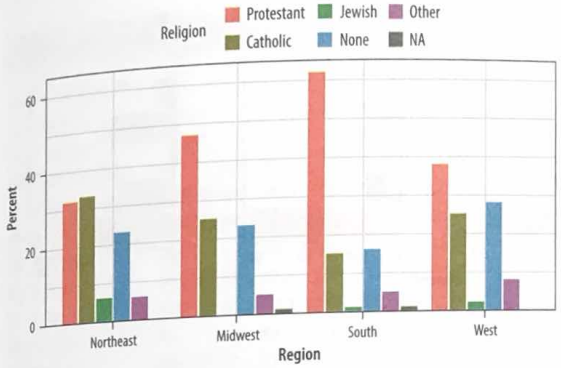

Figure 5.2: Religious preferences by region.

illegible. It's possible to manually adjust the tick-mark labels so that they are printed at an angle, but that isn't so easy to read, either. It makes more sense to put the religions on the y-axis and the percent scores on the x-axis. Because of the way `geom_bar()` works internally, simply swapping the `x` and `y` mapping will not work. (Try it and see what happens.) What we do instead is to transform the *coordinate system* that the results are plotted in, so that the x and y axes are flipped. We do this with `coord_flip()`.

```
p <- ggplot(rel_by_region, aes(x = religion, y = pct, fill = religion))
p + geom_col(position = "dodge2") +
    labs(x = NULL, y = "Percent", fill = "Religion") +
    guides(fill = FALSE) +
    coord_flip() +
    facet_grid(~ bigregion)
```

For most plots the coordinate system is Cartesian, showing plots on a plane defined by an x-axis and a y-axis. The `coord_cartesian()` function manages this, but we don't need to call it. The `coord_flip()` function switches the x and y axes after the plot is made. It does not remap variables to aesthetics. In this case, `religion` is still mapped to `x` and `pct` to `y`. Because the religion names do not need an axis label to be understood, we set `x = NULL` in the `labs()` call. (See fig. 5.3.)

We will see more of what dplyr's grouping and filtering operations can do later. It is a flexible and powerful framework. For now, think of it as a way to quickly summarize tables of data