Accessing Data

The switch from R Markdown to a multi-tool workflow can improve the quality of your work significantly. But R Markdown on its own does nothing to improve how you access data. In Chapter 6 the data I worked with came from a CSV file. To get my data in CSV format, I still had to do manual work. And manual work is one thing we’re working on avoiding.

Fortunately for us, R has many ways to help you not only automate reporting with R Markdown, but also automate the process of accessing data. In this chapter, I’ll discuss two such approaches. The first uses the googlesheets4 package to bring in data directly from Google Sheets. The second uses the tidycensus package to access data from the United States Census Bureau. If you access data from either of these sources, this chapter will help you learn to do so efficiently. But even if you don’t use Google Sheets or the Census Bureau data, this chapter can still help. R has many packages to automate the process of accessing data from a wide variety of sources (I discuss a few others below). Before you hit the export button to download your data as a CSV, see if you can find a package to automate accessing your data. Your data analysis and reporting will be more efficient and more accurate.

Importing Data from Google Sheets

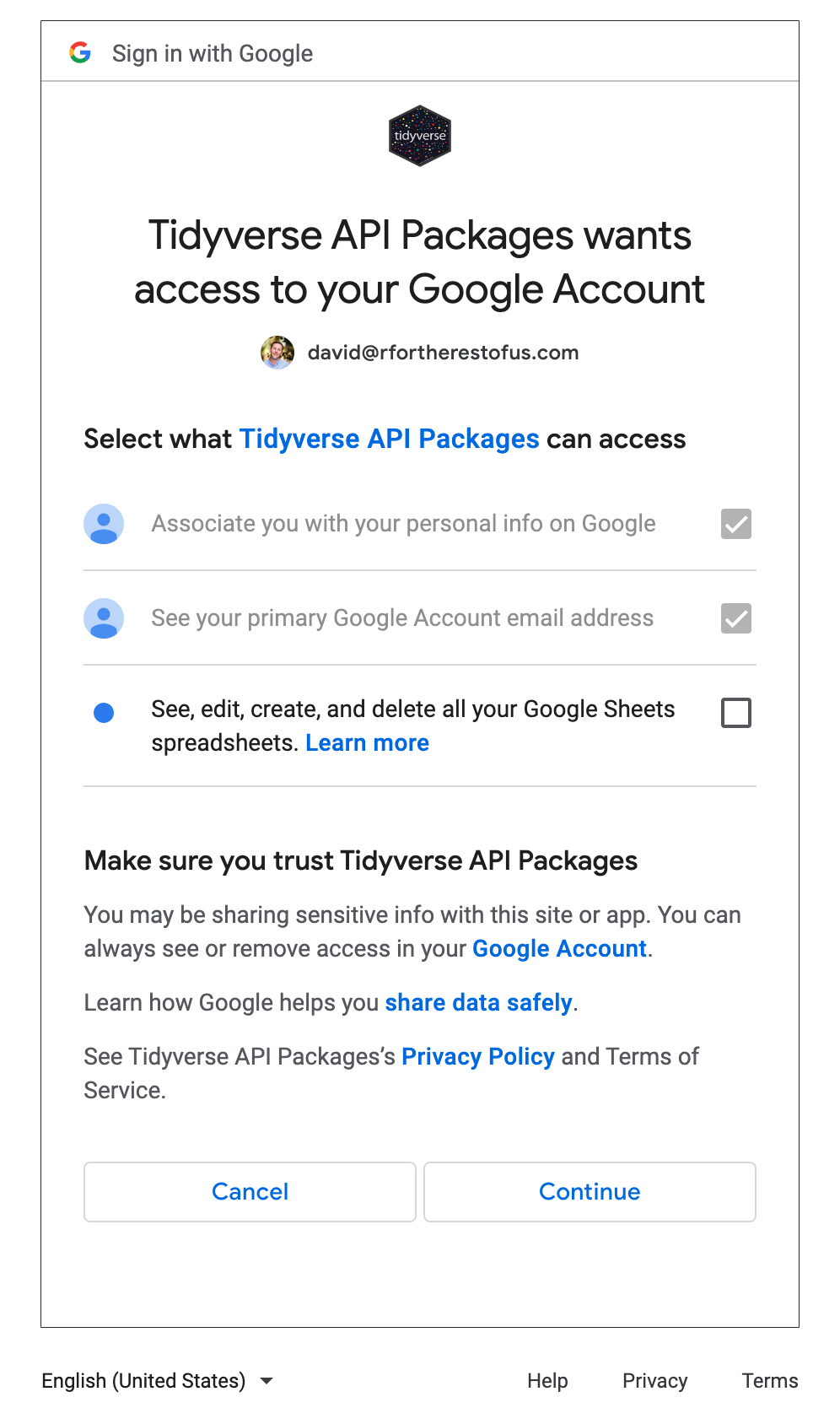
In 2020, Meghan Harris started a job at the Primary Care Research Institute at the University of Buffalo. Her title was Data Integration Specialist, which was both generic and an accurate representation of the work she would do. One of the projects Harris worked on during her time in this job was looking at people affected by opioid use disorder, and data for this project came from a variety of surveys, all of which fed into a series of Google Sheets. She started her new job faced with a jumble of Google Sheets, tasked with helping the organization to make sense of and use its data.

If R Markdown is an improvement on the typical multitool workflow discussed in Chapter 6, using the googlesheets4 package to access data directly from Google Sheets represents a similar improvement compared to downloading data each time you want to update a report. Rather than going through multiple steps (downloading data, copying it into your project, adjusting your code so it imports the new data), you can write code so that it automatically brings in new data directly from Google Sheets. Whenever you need to update your report, simply run your code and the report, generated with the latest data, will be created. I’ll use a simple example below to demonstrate how the googlesheets4 package works. This example, using fake data on video game preferences, is one that Meghan Harris created to mirror her work with opioid survey data (which, for obvious reasons, is confidential).

Using the googlesheets4 Package

After installing the googlesheets4 package with the standard install.packages("googlesheets4"), you are ready to use it. Before you access data in a Google Sheet, you will need to connect your Google account. To do this, run the gs4\_auth() function in the console. If you have more than one Google account, you will need to select the account that has access to the Google Sheet you want to work with. Once you do so, you’ll see a screen that looks like Figure 10-1.

[F10001.png]

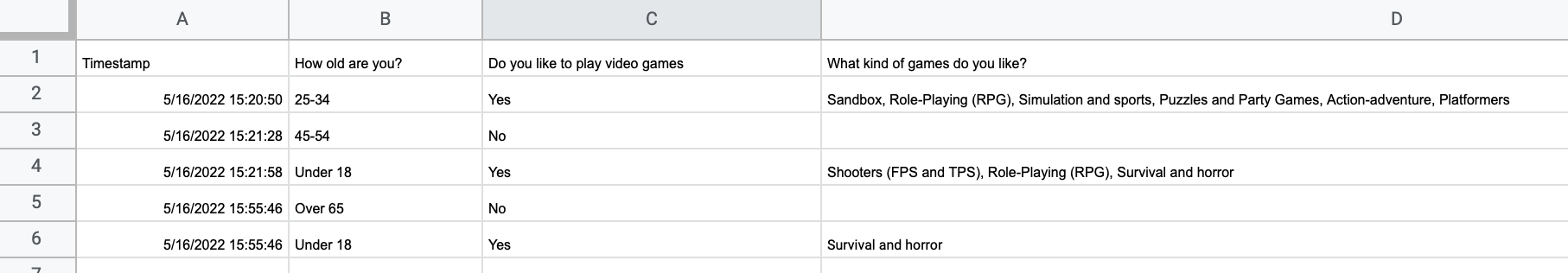


* + - * 1. The screen asking for authorization to access your Google Sheets data

The most important thing is to check the box for “See, edit, create, and delete all your Google Sheets spreadsheets”. This will ensure that R is able to access data from your Google Sheets account. Hit Continue and you’ll be given the message “Authentication complete. Please close this page and return to R.” The googlesheets4 package will now save your credentials so that you can use them in the future without having to authenticate each time.

Now that we’ve connected R to our Google account, we can import data. We’ll import fake data that Meghan Harris created on video preferences. You can see in Figure 10-2 what it looks like in Google Sheets.

[F10002.png]



* + - * 1. The video game data in Google Sheets

The googlesheets4 package has a function called read\_sheet() that allows you to pull in data directly from a Google Sheet. We can import this data with this function in the following way:

library(googlesheets4)

survey\_data\_raw <- read\_sheet("https://docs.google.com/spreadsheets/d/1AR0\_RcFBg8wdiY4Cj-k8vRypp\_txh27MyZuiRdqScog/edit?usp=sharing")

We can take a look at the survey\_data\_raw object to confirm that our data was imported. I’m using the glimpse() function from the dplyr package in order to make it easier to read.

library(tidyverse)

survey\_data\_raw %>%

glimpse()

The output shows that we have indeed imported the data directly from Google Sheets:

#> Rows: 5

#> Columns: 5

#> $ Timestamp <dttm> 2022-05-16 15:2…

#> $ `How old are you?` <chr> "25-34", "45-54"…

#> $ `Do you like to play video games` <chr> "Yes", "No", "Ye…

#> $ `What kind of games do you like?` <chr> "Sandbox, Role-P…

#> $ `What's your favorite game?` <chr> "It's hard to ch…

Once we have the data in R, we can now use the same workflow as always when creating reports with R Markdown. The code below is taken from an R Markdown report that Meghan Harris made to summarize the video games data. You can see the YAML, the setup code chunk, a chunk to load packages, followed by the code to read in data from Google Sheets. The next code chunk cleans the survey\_data\_raw object, saving the result as survey\_data\_clean. We then use this data to:

Calculate the number of respondents and put this in the text using inline R code

Create a table that shows the respondents broken down by age group

Create a graph that shows how many respondents like video games

The code used to generate this report is below.

---

title: "Video Game Survey"

output: html\_document

---

```{r setup, include=FALSE}

knitr::opts\_chunk$set(echo = FALSE,

warning = FALSE,

message = FALSE)

```

```{r}

library(tidyverse)

library(janitor)

library(googlesheets4)

library(gt)

```

```{r}

# Import data from Google Sheets

survey\_data\_raw <- read\_sheet("https://docs.google.com/spreadsheets/d/1AR0\_RcFBg8wdiY4Cj-k8vRypp\_txh27MyZuiRdqScog/edit?usp=sharing")

```

```{r}

# Clean data

survey\_data\_clean <- survey\_data\_raw %>%

clean\_names() %>%

mutate("participant\_id" = as.character(row\_number())) %>%

rename("age" = "how\_old\_are\_you",

"like\_games" = "do\_you\_like\_to\_play\_video\_games",

"game\_types" = "what\_kind\_of\_games\_do\_you\_like",

"favorite\_game" = "whats\_your\_favorite\_game") %>%

relocate(participant\_id, .before = "age") %>%

mutate(age = factor(age, levels = c("Under 18", "18-24", "25-34", "35-44", "45-54", "55-64", "Over 65")))

```

# Respondent Demographics

```{r}

# Calculate number of respondents

number\_of\_respondents <- nrow(survey\_data\_clean)

```

We received responses from `r number\_of\_respondents` respondents. Their ages are below.

```{r}

survey\_data\_clean %>%

select(participant\_id, age) %>%

gt() %>%

cols\_label(

participant\_id = "Participant ID",

age = "Age"

) %>%

tab\_style(

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

locations = cells\_column\_labels()

) %>%

cols\_align(

align = "left",

columns = everything()

) %>%

cols\_width(

participant\_id ~ px(200),

age ~ px(700)

)

```

# Video Games

We asked if respondents liked video games. Their responses are below.

```{r}

survey\_data\_clean %>%

count(like\_games) %>%

ggplot(aes(x = like\_games,

y = n,

fill = like\_games)) +

geom\_col() +

scale\_fill\_manual(values = c(

"No" = "#6cabdd",

"Yes" = "#ff7400"

)) +

labs(title = "How Many People Like Video Games?",

x = NULL,

y = "Number of Participants") +

theme\_minimal(base\_size = 16) +

theme(legend.position = "none",

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

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

axis.title.y = element\_blank(),

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

hjust = 0.5))

```

The resulting report can be seen in Figure 10-3.

[F10003.png]



* + - * 1. The rendered video game report

The R Markdown document here isn’t revolutionary (it has the same type of things we saw in Chapter 6). What is different is the way we’re importing our data. Because we’re bringing it in directly from Google Sheets, there’s no risk of, say, accidentally reading in the wrong CSV. Automating this step reduces the risk of error.

The best part is that we can re-run our code at any point to bring in updated data. The read\_sheet() function will look for all data on the Google Sheet we specify. Our survey had five responses today, but if we run it again tomorrow and it has additional responses, they will all be included in the import. If you use Google Forms to run your survey and have the results go to a Google Sheet, you can have an always up-to-date summary report simply by clicking the Knit button in RStudio. That workflow is one that helped Meghan Harris to collect surveys and manage a wide range of data on opiod use disorder.

Importing data directly from Google Sheets takes our reproducibility one step further, making it possible not only to generate reports automatically, but also automating the process of bringing in the latest data. This process of bringing in data directly from the source applies beyond Google Sheets. There are packages to bring in data directly from Excel365 (Microsoft365R), Qualtrics (qualtRics), Survey Monkey (surveymonkey), and other sources. Before hitting the “Download Data” button in your data collection tool of choice, it’s worth looking into whether a package exists to import data directly into R.

For Meghan Harris, working directly with data in Google Sheets was a game-changer. She used googlesheets4 to bring in data in multiple Google Sheets. From there, she was able to streamline analysis and reporting, which ultimately had a big impact on her organization’s work. Data that had once been largely unused because accessing it was so complicated came to inform research on opioid use disorder. Bringing in data from Google Sheets with a few lines of code may seem small at first, but it can have a big impact.

Accessing Census Data with the tidycensus Package

If you’ve ever worked with data from the United States Census Bureau, you know what a hassle it can be. You’ve got to go to the Census Bureau website, find the data you need, download it, and then analyze it in your tool of choice. Working with Census Bureau data in this way involves a lot of pointing and clicking, and gets very tedious over time.

This tedium is what drove Texas Christian University geographer Kyle Walker to develop a package to automate the process of bringing Census Bureau data into R. Walker had previously created a package called tigris (introduced in Chapter 4) to automatically bring in shape files from the Census Bureau. As he told me, “I was using tigris pretty heavily in my own work to bring in the spatial data, but I didn’t have a seamless way to get the demographic data as well.” Drawing on his experience developing tigris, Walker, along with co-author Matt Herman (yes, he of the Westchester COVID-19 website discussed in Chapter 9), would develop the tidycensus package, which allows R users to bring in data directly from various Census Bureau datasets. With tidycensus, a user can write just a few lines of code and get data on, say, the median income in all 3,000 plus counties in the United States.

Below, we’ll show how the tidycensus package works. We’ll do this using examples from two datasets that tidyverse makes it possible to work with: the every-ten-year (decennial) Census and the American Community Survey. We’ll also show how we can use the data from these two sources for additional analysis and to make maps by accessing geospatial and demographic data simultaneously. While this chapter focuses on data from the United States Census Bureau, I also list other R packages that access analogous data from other countries.

Using tidycensus

The tidycensus package is available on CRAN so you can install it as you would most packages using install.packages("tidycensus"). In order to use tidycensus you must also get an API (application programming interface) key from the Census Bureau. This key, which is free, can be obtained by going to <https://api.census.gov/data/key_signup.html> and entering your details. Once you receive your API key by email, you need to put it in a place where tidycensus can find it. The census\_api\_key() function does this for you. Your best bet, after loading the tidycensus package, is to run the function as follows (replacing 123456789 with your actual API key):

library(tidycensus)

census\_api\_key("123456789", install = TRUE)

The install = TRUE argument will save your API key in your .Renviron file (a file designed to keep confidential information like API keys). The tidycensus will look for your API key there in the future so that you don’t have to enter it every time you want to use the package.

Having obtained and saved our API key, we’re now ready to use tidycensus to access data. The Census Bureau puts out many datasets, several of which can be accessed using tidycensus. The most common datasets to access with tidycensus are the decennial Census and the American Community Survey (other datasets that can be accessed are discussed in Chapter 2 of Kyle Walker’s book Analyzing US Census Data: Methods, Maps, and Models in R).

Working with Decennial Census Data

We’ll start out by accessing data from the 2020 Census on the Asian population in each state. To do this, we use the get\_decennial() function with three arguments:

get\_decennial(geography = "state",

variables = "P1\_006N",

year = 2020)

The arguments we’re using here are:

geography, which tells get\_decennial() to access data at the state level. There are many other geographies, including county, census tract, and more.

variables is where we choose the variable or variables we want to access. I know that P2\_002N is the variable name for the total Asian, but below I’ll demonstrate how to identify other variables you may want to use.

year is where we select the year from which we want to access data. We’re using data from the 2020 Census.

Running this code returns the following:

#> # A tibble: 52 × 4

#> GEOID NAME variable value

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

#> 1 42 Pennsylvania P1\_006N 510501

#> 2 06 California P1\_006N 6085947

#> 3 54 West Virginia P1\_006N 15109

#> 4 49 Utah P1\_006N 80438

#> 5 36 New York P1\_006N 1933127

#> 6 11 District of Columbia P1\_006N 33545

#> 7 02 Alaska P1\_006N 44032

#> 8 12 Florida P1\_006N 643682

#> 9 45 South Carolina P1\_006N 90466

#> 10 38 North Dakota P1\_006N 13213

#> # … with 42 more rows

The resulting data frame has four variables:

1. GEOID is the geographic identifier given by the Census Bureau for the state. Each state has a geographic identifier, as do all counties, census tracts, and all other geographies.
2. NAME is the name of each state.
3. variable is the name of the variable we passed to the get\_decennial() function.
4. value is the numeric value for the state and variable in each row. In our case, it represents the total Asian population in each state.

Let’s say we want to calculate the Asian population as a percentage of all people in each state. To do that, we’d need both the Asian population as well as the total population. How would we do this?

Identifying Variables

First, we’d need to know the variable names. I looked up the variable name for Asian population (P1\_006N) without showing you how I did it. Let’s backtrack so I can show you how to identify variable names. The tidycensus package has a function called load\_variables() that shows us all of the variables from the decennial Census. If we run it with the argument year set to 2020 and dataset set to “pl” (pl refers to public law 94-171, which requires the Census to produce so-called redistricting summary data files every ten years).

load\_variables(year = 2020,

dataset = "pl")

Running this code returns the name, label (description), and concept (category) of all variables available to us. Looking at this, we can see variable P1\_006N (it’s cut off here, but in RStudio you’d see the full description). We can also see that variable P1\_001N gives us the total population.

#> # A tibble: 301 × 3

#> name label concept

#> <chr> <chr> <chr>

#> 1 H1\_001N " !!Total:" OCCUPA…

#> 2 H1\_002N " !!Total:!!Occupied" OCCUPA…

#> 3 H1\_003N " !!Total:!!Vacant" OCCUPA…

#> 4 P1\_001N " !!Total:" RACE

#> 5 P1\_002N " !!Total:!!Population of one race:" RACE

#> 6 P1\_003N " !!Total:!!Population of one race:!!Whi… RACE

#> 7 P1\_004N " !!Total:!!Population of one race:!!Bla… RACE

#> 8 P1\_005N " !!Total:!!Population of one race:!!Ame… RACE

#> 9 P1\_006N " !!Total:!!Population of one race:!!Asi… RACE

#> 10 P1\_007N " !!Total:!!Population of one race:!!Nat… RACE

#> # … with 291 more rows

Using Multiple Variables

Now that we know which variables we need, we can use the get\_decennial() function again. We used just one variable above, but we can run our code again with two variables.

get\_decennial(geography = "state",

variables = c("P1\_001N", "P1\_006N"),

year = 2020) %>%

arrange(NAME)

I’ve added arrange(NAME) after get\_decennial() so that the results are sorted by state name, allowing us to see that we have both variables for each state.

#> # A tibble: 104 × 4

#> GEOID NAME variable value

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

#> 1 01 Alabama P1\_001N 5024279

#> 2 01 Alabama P1\_006N 76660

#> 3 02 Alaska P1\_001N 733391

#> 4 02 Alaska P1\_006N 44032

#> 5 04 Arizona P1\_001N 7151502

#> 6 04 Arizona P1\_006N 257430

#> 7 05 Arkansas P1\_001N 3011524

#> 8 05 Arkansas P1\_006N 51839

#> 9 06 California P1\_001N 39538223

#> 10 06 California P1\_006N 6085947

#> # … with 94 more rows

Giving Variables More Descriptive Names

I often have trouble remembering what variable names like P1\_001N and P1\_006N mean. Fortunately, we can adjust our code in get\_decennial() to give our variables more meaningful names using the following syntax:

get\_decennial(geography = "state",

variables = c(total\_population = "P1\_001N",

asian\_population = "P1\_006N"),

year = 2020) %>%

arrange(NAME)

When we run this code, it is now much easier to remember which variables we are working with.

#> # A tibble: 104 × 4

#> GEOID NAME variable value

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

#> 1 01 Alabama total\_population 5024279

#> 2 01 Alabama asian\_population 76660

#> 3 02 Alaska total\_population 733391

#> 4 02 Alaska asian\_population 44032

#> 5 04 Arizona total\_population 7151502

#> 6 04 Arizona asian\_population 257430

#> 7 05 Arkansas total\_population 3011524

#> 8 05 Arkansas asian\_population 51839

#> 9 06 California total\_population 39538223

#> 10 06 California asian\_population 6085947

#> # … with 94 more rows

Instead of “P1\_001N” and “P1\_006N”, we have “total\_population” and “asian\_population.” Much better!

Analyzing Census Data

Let’s now return to what started us down this path: calculating the Asian population in each state as a percentage of the total. To do this, we use the code from above and add a few things to it:

1. We use group\_by(NAME) to create one group for each state because we want to calculate the Asian population percentage in each state.
2. We use mutate(pct = value / sum(value)) to calculate the percentage. This line takes the value in each row and divides it by the total\_population and asian\_population rows for each state.
3. We use ungroup() to remove the state-level grouping.
4. We use filter(variable == "asian\_population") to only show the Asian population percentage.

get\_decennial(geography = "state",

variables = c(total\_population = "P1\_001N",

asian\_population = "P1\_006N"),

year = 2020) %>%

arrange(NAME) %>%

group\_by(NAME) %>%

mutate(pct = value / sum(value)) %>%

ungroup() %>%

filter(variable == "asian\_population")

When we run this code, we see the Asian population and the Asian population as a percentage of the total population in each state.

#> # A tibble: 52 × 5

#> GEOID NAME variable value pct

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

#> 1 01 Alabama asian\_popula… 76660 0.015029

#> 2 02 Alaska asian\_popula… 44032 0.056638

#> 3 04 Arizona asian\_popula… 257430 0.034746

#> 4 05 Arkansas asian\_popula… 51839 0.016922

#> 5 06 California asian\_popula… 6085947 0.13339

#> 6 08 Colorado asian\_popula… 199827 0.033452

#> 7 09 Connecticut asian\_popula… 172455 0.045642

#> 8 10 Delaware asian\_popula… 42699 0.041349

#> 9 11 District of Columbia asian\_popula… 33545 0.046391

#> 10 12 Florida asian\_popula… 643682 0.029018

#> # … with 42 more rows

Using a Summary Variable

Kyle Walker knew that calculating summaries like this would be a common use case for tidycensus. So, to simplify things, he gives us the summary\_var argument that we can use within get\_decennial(). Instead of putting “P1\_001N” (total population) in the variables argument, we can instead use it with the summary\_var argument as follows.

get\_decennial(geography = "state",

variables = c(asian\_population = "P1\_006N"),

summary\_var = "P1\_001N",

year = 2020) %>%

arrange(NAME)

This returns a nearly identical data frame to what we got above, except that the total population is now a separate variable, rather than additional rows for each state.

#> # A tibble: 52 × 5

#> GEOID NAME variable value summar…¹

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

#> 1 01 Alabama asian\_popula… 76660 5024279

#> 2 02 Alaska asian\_popula… 44032 733391

#> 3 04 Arizona asian\_popula… 257430 7151502

#> 4 05 Arkansas asian\_popula… 51839 3011524

#> 5 06 California asian\_popula… 6085947 39538223

#> 6 08 Colorado asian\_popula… 199827 5773714

#> 7 09 Connecticut asian\_popula… 172455 3605944

#> 8 10 Delaware asian\_popula… 42699 989948

#> 9 11 District of Columbia asian\_popula… 33545 689545

#> 10 12 Florida asian\_popula… 643682 21538187

#> # … with 42 more rows, and abbreviated variable name

#> # ¹​summary\_value

With our data in this new format, we can calculate the Asian population as a percentage of the whole by dividing the value variable by the summary\_value variable.

get\_decennial(geography = "state",

variables = c(asian\_population = "P1\_006N"),

summary\_var = "P1\_001N",

year = 2020) %>%

arrange(NAME) %>%

mutate(pct = value / summary\_value)

The resulting output is nearly identical.

#> # A tibble: 52 × 6

#> GEOID NAME varia…¹ value summar…² pct

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

#> 1 01 Alabama asian\_… 76660 5024279 0.015258

#> 2 02 Alaska asian\_… 44032 733391 0.060039

#> 3 04 Arizona asian\_… 257430 7151502 0.035997

#> 4 05 Arkansas asian\_… 51839 3011524 0.017214

#> 5 06 California asian\_… 6085947 39538223 0.15393

#> 6 08 Colorado asian\_… 199827 5773714 0.034610

#> 7 09 Connecticut asian\_… 172455 3605944 0.047825

#> 8 10 Delaware asian\_… 42699 989948 0.043133

#> 9 11 District of Colu… asian\_… 33545 689545 0.048648

#> 10 12 Florida asian\_… 643682 21538187 0.029886

#> # … with 42 more rows, and abbreviated variable names

#> # ¹​variable, ²​summary\_value

How you choose to calculate summary statistics is up to you. The good thing is that tidycensus makes it easy to do either way!

Working with American Community Survey Data

Let’s switch now to accessing data from the American Community Survey (ACS). This survey, which is conducted every year, differs from the decennial Census in two major ways:

1. It is given to a sample of people rather than the entire population.
2. It includes a wider range of questions.

Despite these differences, accessing data from the ACS is nearly identical to how we access Census data. Instead of get\_decennial(), we use the function get\_acs(), but the arguments are the same. Here I’ve identified a variable I’m interested in (B01002\_001, which shows median age) and am using it to get the data for each state.

get\_acs(geography = "state",

variables = "B01002\_001",

year = 2020)

Here’s what the output looks like:

#> # A tibble: 52 × 5

#> GEOID NAME variable estimate moe

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

#> 1 01 Alabama B01002\_001 39.2 0.1

#> 2 02 Alaska B01002\_001 34.6 0.2

#> 3 04 Arizona B01002\_001 37.9 0.2

#> 4 05 Arkansas B01002\_001 38.3 0.2

#> 5 06 California B01002\_001 36.7 0.1

#> 6 08 Colorado B01002\_001 36.9 0.1

#> 7 09 Connecticut B01002\_001 41.1 0.2

#> 8 10 Delaware B01002\_001 41 0.2

#> 9 11 District of Columbia B01002\_001 34.1 0.1

#> 10 12 Florida B01002\_001 42.2 0.2

#> # … with 42 more rows

There are two differences we can see in the get\_acs() output compared to that from get\_decennial():

1. The value column in get\_decennial() is called estimate with get\_acs().
2. We have an additional column called moe for margin of error.

Both of these changes are because the ACS is given to a sample of the population. As a result, we don’t have precise values, but rather estimates, which are extrapolations from the sample to the population as a whole. And with an estimate comes a margin of error. In our state-level data, the margins of error are relatively low, but if you get data from smaller geographies, they tend to be higher. In cases where your margins of error are high relative to your estimates, you should interpret results with caution, as there is greater uncertainty about how well the data represents the population as a whole.

Using ACS Data to Make Charts

As we saw with Census data on the Asian population in the United States, once you access data using the tidycensus package, you can do whatever else you want with it. We calculated the Asian population as a percentage of the total above. Here we could take the data on median age and pipe it into ggplot in order to create a bar chart.

get\_acs(geography = "state",

variables = "B01002\_001",

year = 2020) %>%

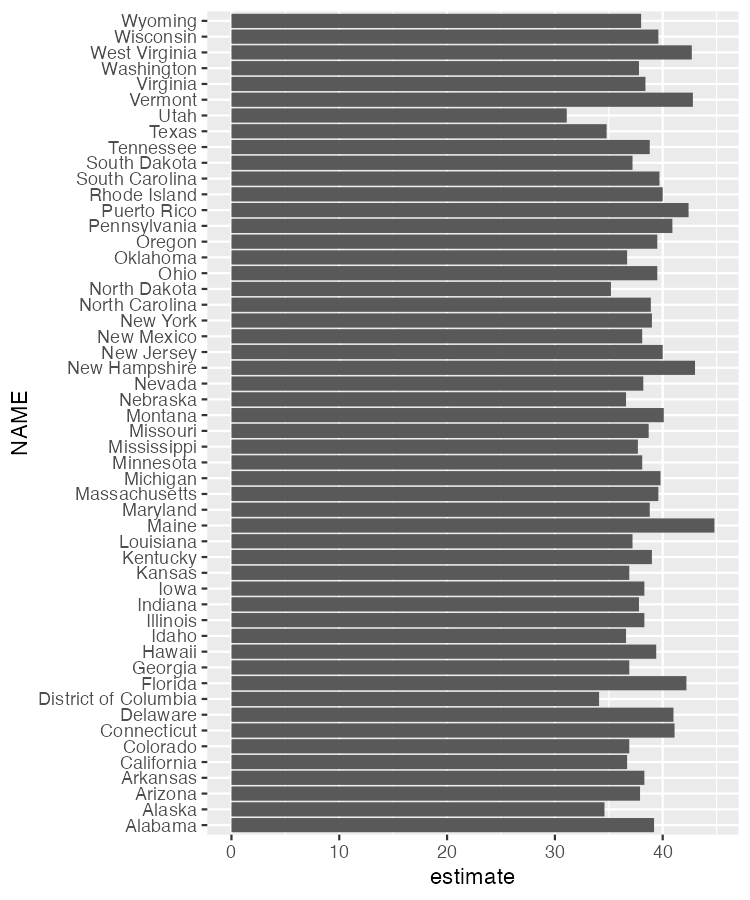
ggplot(aes(x = estimate,

y = NAME)) +

geom\_col()

Figure 10-4 shows our bar chart.

[F10004.pdf]



* + - * 1. A bar chart showing the median age in each state

This chart is nothing special, but the fact that it takes just six lines of code to create most definitely is.

Using ACS Data to Make Maps

Kyle Walker’s original motivation to build tidycensus came from wanting to make it easy to access demographic data, just as he had done with geospatial data in the tigris package. He succeeded. And one additional benefit of Walker working on both packages is that there is a tight integration between them. Using the get\_acs() function, you can set the geometry argument to TRUE and you will get both demographic and geospatial data (which, under the hood, actually comes from the tigris package).

get\_acs(geography = "state",

variables = "B01002\_001",

year = 2020,

geometry = TRUE)

If we take a look at the resulting data, we can see that it has the metadata and geometry column of simple features objects that we saw in Chapter 4.

#> Simple feature collection with 52 features and 5 fields

#> Geometry type: MULTIPOLYGON

#> Dimension: XY

#> Bounding box: xmin: -179.1489 ymin: 17.88328 xmax: 179.7785 ymax: 71.36516

#> Geodetic CRS: NAD83

#> First 10 features:

#> GEOID NAME variable estimate moe

#> 1 35 New Mexico B01002\_001 38.1 0.1

#> 2 72 Puerto Rico B01002\_001 42.4 0.2

#> 3 06 California B01002\_001 36.7 0.1

#> 4 01 Alabama B01002\_001 39.2 0.1

#> 5 13 Georgia B01002\_001 36.9 0.1

#> 6 05 Arkansas B01002\_001 38.3 0.2

#> 7 41 Oregon B01002\_001 39.5 0.1

#> 8 28 Mississippi B01002\_001 37.7 0.2

#> 9 08 Colorado B01002\_001 36.9 0.1

#> 10 49 Utah B01002\_001 31.1 0.1

#> geometry

#> 1 MULTIPOLYGON (((-109.0502 3...

#> 2 MULTIPOLYGON (((-65.23805 1...

#> 3 MULTIPOLYGON (((-118.6044 3...

#> 4 MULTIPOLYGON (((-88.05338 3...

#> 5 MULTIPOLYGON (((-81.27939 3...

#> 6 MULTIPOLYGON (((-94.61792 3...

#> 7 MULTIPOLYGON (((-123.6647 4...

#> 8 MULTIPOLYGON (((-88.50297 3...

#> 9 MULTIPOLYGON (((-109.0603 3...

#> 10 MULTIPOLYGON (((-114.053 37...

We can pipe this data into ggplot to make a map with the following code.

get\_acs(geography = "state",

variables = "B01002\_001",

year = 2020,

geometry = TRUE) %>%

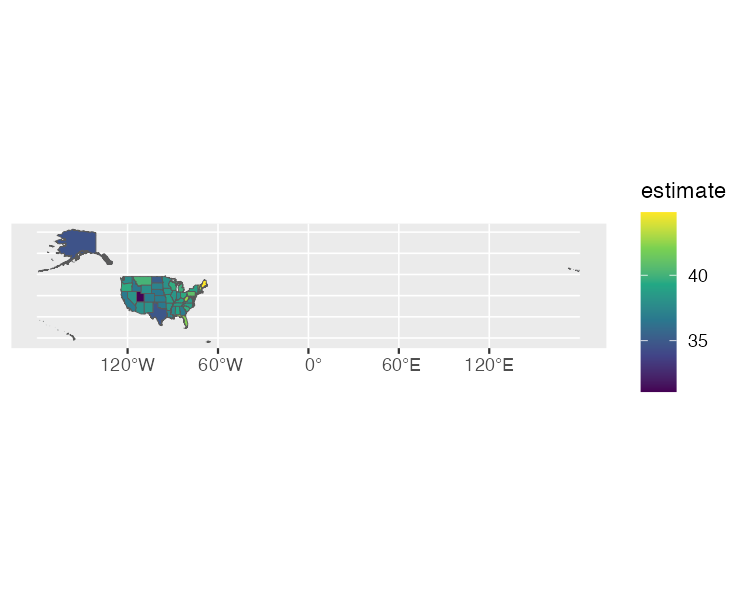
ggplot(aes(fill = estimate)) +

geom\_sf() +

scale\_fill\_viridis\_c()

The resulting map, seen in Figure 10-5 below, is less than ideal. The problem with it is that the Aleutian Islands in Alaska cross the 180-degree line of longitude, also known as the international date line. As a result, most of Alaska is on one side of the map while a small part is on the other side. What’s more, both Hawaii and Puerto Rico, both being decently far from the United States mainland and relatively small, are hard to see.

[F10005.pdf]



* + - * 1. A hard-to-read map showing median age by state

Fortunately for us, Kyle Walker has a solution. If we load the tigris package, we can then use the shift\_geometry() function to move Alaska, Hawaii, and Puerto Rico into places where they are more easily visible. We set the argument preserve\_area to FALSE so that the giant state of Alaska is shrunk while Hawaii and Puerto Rico are made larger.

library(tigris)

get\_acs(geography = "state",

variables = "B01002\_001",

year = 2020,

geometry = TRUE) %>%

shift\_geometry(preserve\_area = FALSE) %>%

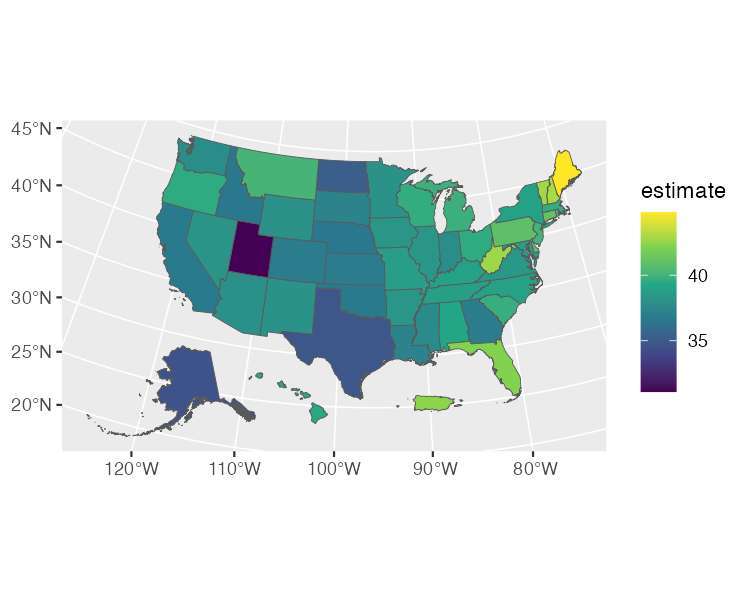
ggplot(aes(fill = estimate)) +

geom\_sf() +

scale\_fill\_viridis\_c()

This lack of precision in the exact sizes of the states is more than made up for by having an easier to read map, which we can see in Figure 10-6.

[F10006.pdf]



* + - * 1. An easier-to-read map showing median age by state

We’ve made a map that shows median age by state. But there’s nothing to stop us from making the same map by county. Just change the geography argument to “county” and you’ll get a map for all 3,000 plus counties. Chapter 2 of Kyle Walker’s book Analyzing US Census Data: Methods, Maps, and Models in R discusses the various geographies available. There are also many more arguments in both the get\_decennial() and get\_acs() functions. We’ve only shown a few of the most common arguments. If you want to learn more, Walker’s book is a great resource.

Packages like googlesheets4 and tidycensus Make it Easy to Access Data

If you work with Census data, the tidycensus package is a huge timesaver. Rather than having to manually download data from the Census Bureau website, you can write R code that brings it in automatically, making it ready for analysis and reporting. If you’re looking for Census data from other countries, Chapter 12 of Walker’s Analyzing US Census Data book gives examples of packages that can help. There are R packages to bring Census data from Canada, Kenya, Mexico, Brazil, and other countries.

What all of these packages (and the googlesheets4 package) have in common is that they use application programming interfaces (APIs) to access data directly from its source. These packages are often referred to as “wrapper packages” because they wrap R code around the code needed to access data through APIs. You don’t have to figure out how to access data through APIs yourself; you can just write some simple R code and the wrapper packages convert your code into the complex code needed to bring in the data.

In talking with Kyle Walker, he nicely summarized the benefit of tidycensus, saying it does “all of the tedious aspects of getting census data so that you can focus on the fun aspects.” He continued: “making maps is fun, analyzing data and finding out insights about your community is fun and interesting. But setting up a connector to an API or figuring out how to align columns [is] more tedious.”

This is the benefit of working with an open-source tool like R. Because R is extensible, others can create packages to do things that would take you extraordinary amounts of time to do on your own. You don’t need to figure on your own out how to access data from Google Sheets or the Census Bureau API by yourself. You can build on the hours of work put in by the R community to make your life easier.