Part III

Automation and Collaboration

Automatically Accessing Online Data

So far, you’ve imported data into your projects from CSV files. Many online data sets allow you to export CSVs, but before you do so, you should look for packages to automate your data access. If you can eliminate the manual steps involved in fetching data, your analysis and reporting will be more accurate. You’ll also be able to efficiently update your report when the data changes.

R offers many ways to automate the process of accessing online data. In this chapter, I’ll discuss two such approaches. First, you’ll use the googlesheets4 package to fetch data directly from Google Sheets. Then, you’ll use the tidycensus package to access data from the United States Census Bureau. You’ll work with two large census data sets, the Decennial Census and the American Community Survey, and practice visualizing them.

Importing Data from Google Sheets with googlesheets4

By using the googlesheets4 package to access data directly from Google Sheets, you avoid having to manually download data, copy it into your project, and adjust your code so it imports that new data every time you want to update a report. This package lets you write code that automatically fetches new data directly from Google Sheets. Whenever you need to update your report, you can simply run your code to refresh the data. In addition, if you work with Google Forms, you can pipe your data into Google Sheets, completely automating the workflow from data collection to data import.

Using the googlesheets4 package can help you manage complex data sets that update frequently. For example, in her role at the Primary Care Research Institute at the University of Buffalo, Meghan Harris used it for a research project about people affected by opioid use disorder. The data came from a variety of surveys, all of which fed into a jumble of Google Sheets. Using googlesheets4, Harris was able to collect all of her data in one place and use R to put it to use. Data that had once been largely unused because accessing it was so complicated could now inform research on opioid use disorder.

This section demonstrates how the googlesheets4 package works using a fake data set about video game preferences that Harris created to replace her opioid survey data (which, for obvious reasons, is confidential).

Connecting to Google

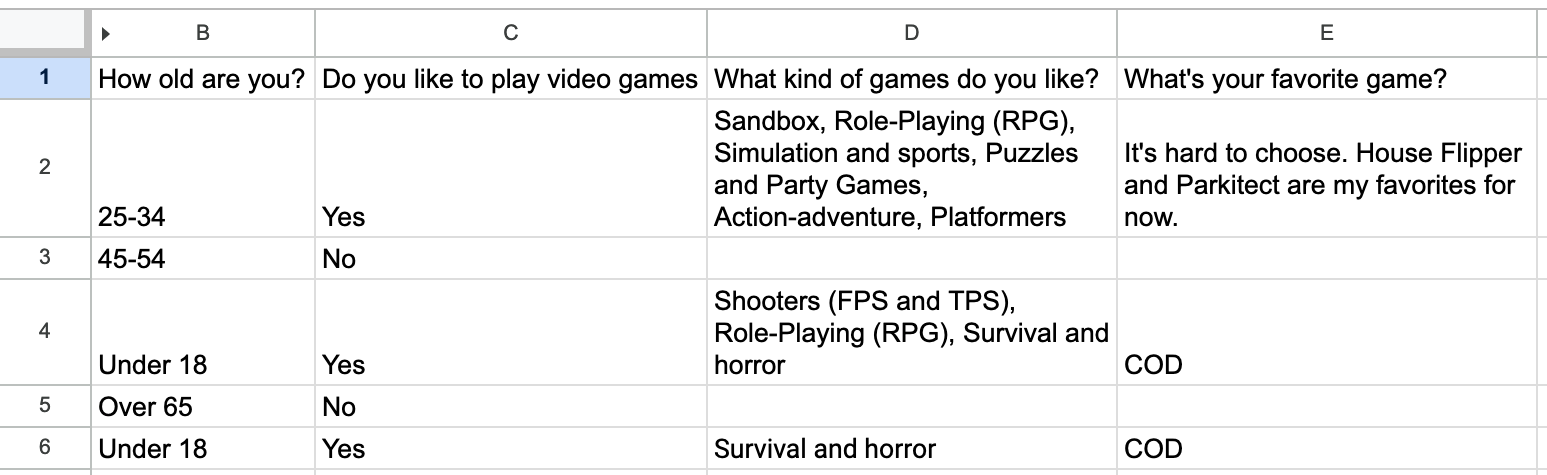
To begin, install the googlesheets4 package by running install.packages("googlesheets4"). Next, connect to your Google account by running the gs4\_auth() function in the console. If you have more than one Google account, select the account that has access to the Google Sheet you want to work with.

Once you do so, a screen should appear. Check the box next to See, Edit, Create, and Delete All Your Google Sheets Spreadsheets. This will ensure that R can access data from your Google Sheets account. Click Continue, and you should see 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 reauthenticate.

Reading Data from a Sheet

Now that you’ve connected R to your Google account, you can import the fake data that Harris created about video game preferences (access it at https://data.rwithoutstatistics.com/google-sheet). Figure 11-1 shows what it looks like in Google Sheets.

[F11001.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. Import the data by passing the spreadsheet’s URL to the function like so:

library(googlesheets4)

survey\_data\_raw <- read\_sheet("https://docs.google.com/spreadsheets/d/

1AR0\_RcFBg8wdiY4Cj-k8vRypp\_txh27MyZuiRdqScog/edit?usp=sharing")

Take a look at the survey\_data\_raw object to confirm that the data was imported. Using the glimpse() function from the dplyr package makes it easier to read:

library(tidyverse)

survey\_data\_raw %>%

glimpse()

The glimpse() function, which creates one output row per variable, shows that you’ve successfully imported the data directly from Google Sheets:

#> Rows: 5

#> Columns: 5

#> $ Timestamp <dttm> 05-16 15:20:50

#> $ `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 you have the data in R, you can use the same workflow you’ve been using to create reports with R Markdown.

Using the Data in R Markdown

The following code is taken from an R Markdown report that Harris made to summarize the video games data. You can see the YAML, the setup code chunk, a code chunk that loads packages, and the code to import data from Google Sheets:

---

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

1 survey\_data\_raw <- read\_sheet("https://docs.google.com/spreadsheets/d/

1AR0\_RcFBg8wdiY4Cj-k8vRypp\_txh27MyZuiRdqScog/edit?usp=sharing")

```

This R Markdown document resembles those discussed in previous chapters, except for the way you import the data 1. Because you’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 next code chunk cleans the survey\_data\_raw object, saving the result as survey\_data\_clean:

```{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")))

```

Here, the clean\_names() function from the janitor package makes the variable names easier to work with. Defining a participant\_id variable using the row\_number() function then adds a consecutively increasing number to each row, and the as.character() function makes the number a character. Next, the code changes several variable names with the rename() function. The mutate() function then transforms the age variable into a data structure known as a factor, which ensures that age will show up in the right order in your chart. Finally, the relocate() function positions participant\_id before the age variable.

Now you can use the glimpse() function again to view your updated survey\_data\_clean data frame, which looks like this:

#> Rows: 5

#> Columns: 6

#> $ timestamp <dttm> 2024-05-16 15:20:50, 2024-05-16 15:21:28, 2024-05...

#> $ participant\_id <chr> "1", "2", "3", "4", "5"

#> $ age <fct> 25-34, 45-54, Under 18, Over 65, Un...

#> $ like\_games <chr> "Yes", "No", "Yes", "No", "Yes"

#> $ game\_types <chr> "Sandbox, Role-Playing (RPG), Simul...

#> $ favorite\_game <chr> "It's hard to choose. House Flipper...

The rest of the report uses this data to highlight various statistics:

# Respondent Demographics

```{r}

# Calculate number of respondents

number\_of\_respondents <- nrow(survey\_data\_clean) 1

```

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

```{r}

survey\_data\_clean %>%

select(participant\_id, age) %>%

gt() %>% 2

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, 3

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

)

)

```

These sections calculate the number of survey respondents 1, then put this value in the text using inline R code; create a table that breaks down the respondents by age group 2; and generate a graph displaying how many respondents like video games 3. Figure 11-2 shows the resulting report.

[F11002.png]



* + - * 1. The rendered video game report

You can rerun the code at any point to fetch updated data. The survey had five responses today, but if you run it again tomorrow and it has additional responses, they will be included in the import. If you used Google Forms to run your survey and saved the results to a Google Sheet, you could produce this up-to-date report simply by clicking the Knit button in RStudio.

Importing Only Certain Columns

In the previous sections, you read the data of the entire Google Sheet, but you also have the option to import only a section of a sheet. For example, the survey data includes a timestamp column. This variable is added automatically whenever someone submits a Google Form that pipes data into a Google Sheet, but you don’t use it in your analysis, so you could get rid of it.

To do so, use the range argument in the read\_sheet() function when importing the data like so:

read\_sheet("https://docs.google.com/spreadsheets/d/1AR0\_RcFBg8wdiY4Cj-k8vRypp\_

txh27MyZuiRdqScog/edit?usp=sharing",

range = "Sheet1!B:E") %>%

glimpse()

This argument lets you specify a range of data to import. It uses the same syntax you may have used to select columns in Google Sheets. In this example, range = "Sheet1!B:E" imports columns B through E (but not A, which contains the timestamp). Adding glimpse() and then running this code produces output without the timestamp variable:

#> Rows: 5

#> Columns: 4

#> $ `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...

There are a number of other useful functions in the googlesheets4 package. For example, if you ever need to write your output back to a Google Sheet, the write\_sheet() function is there to help. To explore other functions in the package, check out its documentation website at https://googlesheets4.tidyverse.org/index.html.

Now we’ll turn our attention to another R package that allows you to automatically fetch data, this time from the United States Census Bureau.

Accessing Census Data with tidycensus

If you’ve ever worked with data from the US Census Bureau, you know what a hassle it can be. Usually, the process involves visiting the Census Bureau website, searching for the data you need, downloading it, and then analyzing it in your tool of choice. This pointing and clicking gets very tedious after a while.

Kyle Walker, a geographer at Texas Christian University, and Matt Herman (creator of the Westchester COVID-19 website discussed in Chapter 9) developed the tidycensus package to automate the process of importing Census Bureau data into R. With tidycensus, you can write just a few lines of code to get data about, say, the median income in all counties in the United States.

In this section, I’ll show you how the tidycensus package works using examples from two data sets to which it provides access: the Decennial Census, administered every 10 years, and the annual American Community Survey. I’ll also show you how to use the data from these two sources to perform additional analysis and make maps by accessing geospatial and demographic data simultaneously.

Connecting to the Census Bureau with an API Key

Begin by installing tidycensus using install.packages("tidycensus"). To use tidycensus, you must get an application programming interface (API) key from the Census Bureau. API keys are like passwords that online services use to determine whether you’re authorized to access data.

To obtain this key, which is free, go to https://api.census.gov/data/key\_signup.html and enter your details. Once you receive the 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, so after loading the tidycensus package, run the function as follows, replacing 123456789 with your actual API key:

library(tidycensus)

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

The install = TRUE argument saves your API key in your .Renviron file, which is designed for storing confidential information. The package will look for your API key there in the future so that you don’t have to reenter it every time you use the package.

Now you can use tidycensus to access Census Bureau data sets. While the Decennial Census and the American Community Survey are the most common, Chapter 2 of Kyle Walker’s book Analyzing US Census Data: Methods, Maps, and Models in R discusses others you can access.

Working with Decennial Census Data

The tidycensus packages includes several functions dedicated to specific Census Bureau data sets, such as get\_decennial() for Decennial Census data. To access data from the 2020 Decennial Census about the Asian population in each state, use the get\_decennial() function with three arguments as follows:

get\_decennial(geography = "state",

variables = "P1\_006N",

year = 2020)

Setting the geography argument to "state" tells get\_decennial() to access data at the state level. In addition to the 50 states, it will return data for the District of Columbia and Puerto Rico. The variables argument specifies the variable or variables you want to access. Here, P1\_006N is the variable name for the total Asian population. I’ll discuss how to identify other variables you may want to use in the next section. Finally, year specifies the year for which you want to access data—in this case, 2020.

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

--snip--

The resulting data frame has four variables. GEOID is the geographic identifier assigned to the state by the Census Bureau. Each state has a geographic identifier, as do all counties, census tracts, and other geographies. NAME is the name of each state, and variable is the name of the variable you passed to the get\_decennial() function. Finally, value is the numeric value for the state and variable in each row. In this case, it represents the total Asian population in each state.

Identifying Census Variable Values

You’ve just seen how to retrieve the total number of Asian residents of each state, but say you want to calculate that number instead as a percentage of all the state’s residents. To do that, first you need to retrieve the variable for the state’s total population.

The tidycensus package has a function called load\_variables() that shows all of the variables from a Decennial Census. Run it with the year argument set to 2020 and dataset set to pl as follows:

load\_variables(year = 2020,

dataset = "pl")

Running tcodes (CBureau ) and returns the name, label (description), and concept (category) of all available variables:

#> # A tibble: 301 × 3

#> name label concept

#> <chr> <chr> <chr>

#> 1 H1\_001N " !!Total:" OCCU...

#> 2 H1\_002N " !!Total:!!Occupied" OCCU...

#> 3 H1\_003N " !!Total:!!Vacant" OCCU...

#> 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

--snip--

By looking at this list, you can see that the variable P1\_001N returns the total population.

Using Multiple Census Variables

Now that you know which variables you need, you can use the get\_decennial() function again with two variables at once:

get\_decennial(geography = "state",

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

year = 2020) %>%

arrange(NAME)

Adding arrange(NAME) after get\_decennial() sorts the results by state name, allowing you to easily see that the output includes 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

--snip--

When you’re working with multiple census variables like this, you might have trouble remembering what names like P1\_001N and P1\_006N mean. Fortunately, you can adjust the code in the call to get\_decennial() to give these 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)

Within the variables argument, this code specifies the new names for the variables, followed by the equal sign and the original variable names. The c() function allows you to rename multiple variables at one time.

Now it’s much easier to see which variables you’re 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, the variables appear as total\_population and asian\_population. Much better!

Analyzing Census Data

Now you have the data you need to calculate the Asian population in each state as a percentage of the total. There are just a few functions to add to the code from the previous section:

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")

The group\_by(NAME) function creates one group for each state because you want to calculate the Asian population percentage in each state (not for the entire United States). Then mutate() calculates each percentage, taking the value in each row and dividing it by the total\_population and asian\_population rows for each state. The ungroup() function removes the state-level grouping, and filter() shows only the Asian population percentage.

When you run this code, you should see both the total 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 0.133390

#> 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

--snip--

This is a reasonable way to calculate the Asian population as a percentage of the total population in each state—but it’s not the only way.

Using a Summary Variable

Kyle Walker knew that calculating summaries like you’ve just done would be a common use case for tidycensus. To calculate, say, the Asian population as a percentage of the whole, you need to have a numerator (the Asian population) and denominator (the total population). So, to simplify things, Walker included the summary\_var argument, which can be used within get\_decennial() to import the total population as a separate variable. Instead of putting P1\_001N (total population) in the variables argument and renaming it, you can assign it to 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 you just got, 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...

#> <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

--snip--

#> # summary\_value

With the data in this new format, now you can calculate the Asian population as a percentage of the whole by dividing the value variable by the summary\_value variable. Then you drop the summary\_value variable because you no longer need it after doing this calculation:

get\_decennial(

geography = "state",

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

summary\_var = "P1\_001N",

year = 2020

) %>%

arrange(NAME) %>%

mutate(pct = value / summary\_value) %>%

select(-summary\_value)

The resulting output is identical to the output of the previous section:

#> # A tibble: 52 × 5

#> GEOID NAME variable value pct

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

#> 1 01 Alabama asian\_popula... 76660 0.015258

#> 2 02 Alaska asian\_popula... 44032 0.060039

#> 3 04 Arizona asian\_popula... 257430 0.035997

#> 4 05 Arkansas asian\_popula... 51839 0.017214

#> 5 06 California asian\_popula... 6085947 0.153930

#> 6 08 Colorado asian\_popula... 199827 0.034610

#> 7 09 Connecticut asian\_popula... 172455 0.047825

#> 8 10 Delaware asian\_popula... 42699 0.043133

#> 9 11 District of Columbia asian\_popula... 33545 0.048648

#> 10 12 Florida asian\_popula... 643682 0.029886

#> # 42 more rows

How you choose to calculate summary statistics is up to you; tidycensus makes it easy to do either way.

Visualizing American Community Survey Data

Once you’ve accessed data using the tidycensus package, you can do whatever you want with it. In this section, you’ll practice analyzing and visualizing survey data using the American Community Survey. This survey, which is conducted every year, differs from the Decennial Census in two major ways: it is given to a sample of people rather than the entire population, and it includes a wider range of questions.

Despite these differences, you can access data from the American Community Survey nearly identically to how you access Decennial Census data. Instead of get\_decennial(), you use the get\_acs() function, but the arguments you pass to it are the same:

get\_acs(

geography = "state",

variables = "B01002\_001",

year = 2020

)

This code uses the B01002\_001 variable to get median age data from 2020 for each state. 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 0.2

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

#> 10 12 Florida B01002\_001 42.2 0.2

--snip--

You should notice two differences in the output from get\_acs() compared to that from get\_decennial(). First, instead of the value column, get\_acs() produces a column called estimate. Second, it adds a column called moe, for the margin of error. These changes are the result of American Community Survey being given only to a sample of the population, since extrapolating values from that sample to produce an estimate for the population as a whole introduces a margin of error.

In the state-level data, the margins of error are relatively low, but in smaller geographies, they tend to be higher. Cases where your margins of error are high relative to your estimates indicate a greater level of uncertainty about how well the data represents the population as a whole, so you should interpret such results with caution.

Making Charts

To pipe your data on median age into ggplot to create a bar chart, add the following lines:

get\_acs(

geography = "state",

variables = "B01002\_001",

year = 2020

) %>%

ggplot(aes(

x = estimate,

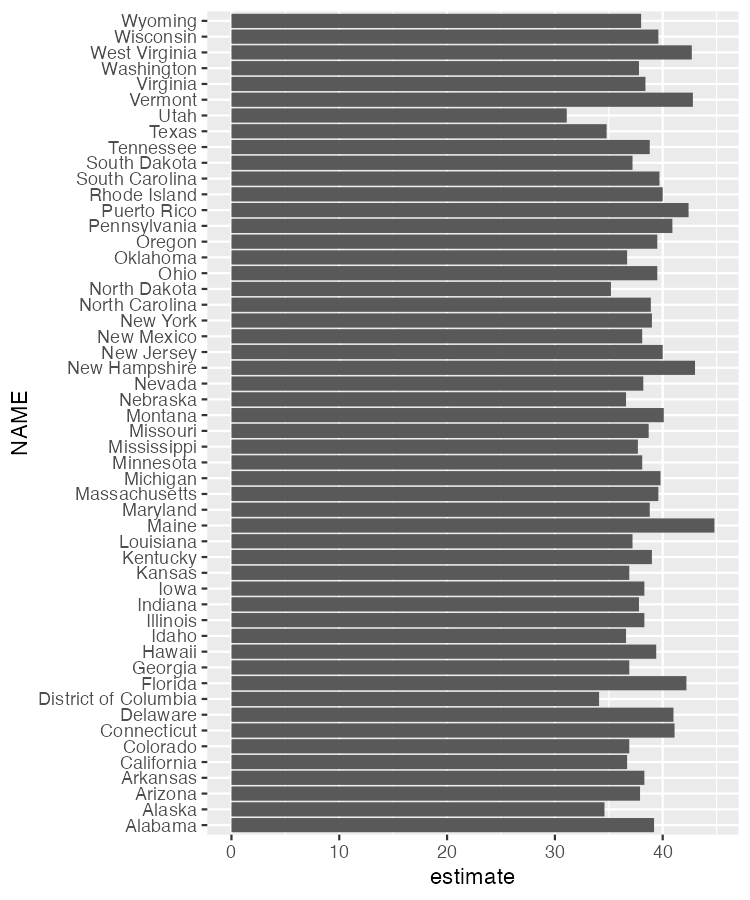
y = NAME

)) +

geom\_col()

After importing the data with the get\_acs() function, the ggplot() function pipes it directly into ggplot. States (which use the variable NAME) will go on the y-axis and median age (estimate) will go on the x-axis. A simple geom\_col() creates the bar chart, shown in Figure 11-3.

[F11003.pdf]



* + - * 1. A bar chart generated using data acquired with the get\_asc() function

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

Making Population Maps with the geometry Argument

In addition to co-creating tidycensus, Kyle Walker also created the tigris package for working with geospatial data. As a result, these packages are tightly integrated. Within the get\_acs() function, you can set the geometry argument to TRUE to receive both demographic data from the Census Bureau and geospatial data from tigris:

get\_acs(

geography = "state",

variables = "B01002\_001",

year = 2020,

geometry = TRUE

)

In the resulting data, you can see that it has the metadata and geometry column of the simple features objects that you 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...

The geometry type is MULTIPOLYGON, which you learned about in Chapter 4. You can pipe this data into ggplot to make a map by adding the following code:

get\_acs(

geography = "state",

variables = "B01002\_001",

year = 2020,

geometry = TRUE

) %>%

ggplot(aes(fill = estimate)) +

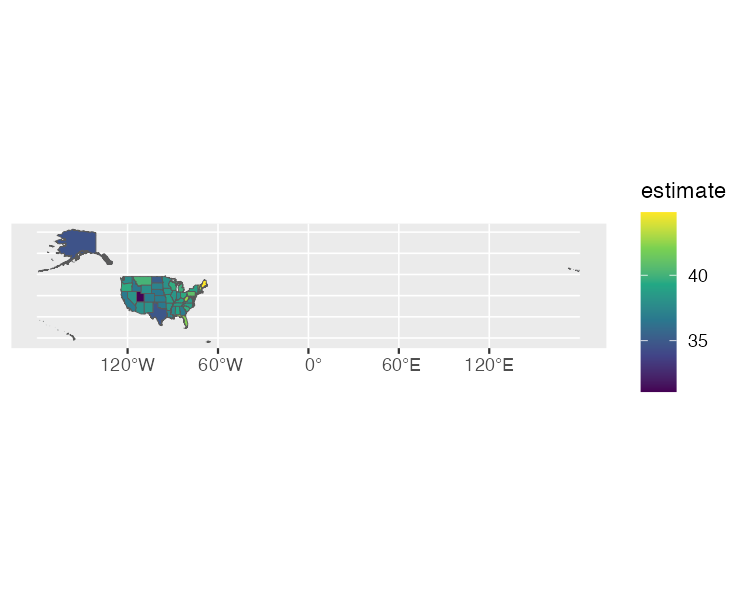
geom\_sf() +

scale\_fill\_viridis\_c()

After importing the data with get\_acs() and piping it into the ggplot() function, this code sets the estimate variable to use for the fill aesthetic property; that is, the fill color of each state will vary depending on the median age of its residents. Then the geom\_sf() draws the map, and the scale\_fill\_viridis\_c() function gives it a colorblind-friendly palette.

The resulting map, seen in Figure 11-4, is less than ideal because the Aleutian Islands in Alaska cross the 180-degree line of longitude, or the International Date Line. As a result, most of Alaska appears on one side of the map and a small part appears on the other side. What’s more, both Hawaii and Puerto Rico are hard to see.

[F11004.pdf]



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

To fix these problems, load the tigris package, then use the shift\_geometry() function to move Alaska, Hawaii, and Puerto Rico into places where they’ll be more easily visible:

library(tigris)

get\_acs(

geography = "state",

variables = "B01002\_001",

year = 2020,

geometry = TRUE

) %>%

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

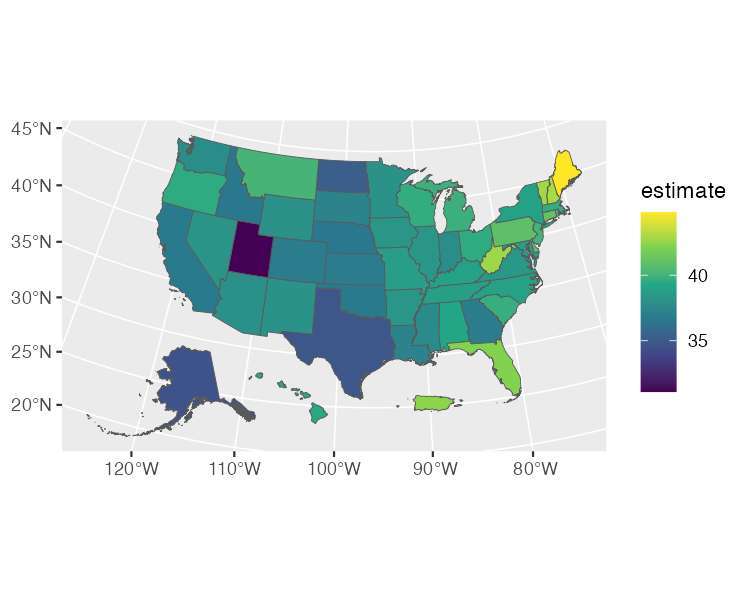
ggplot(aes(fill = estimate)) +

geom\_sf() +

scale\_fill\_viridis\_c()

Setting the preserve\_area argument to FALSE shrinks the giant state of Alaska and makes Hawaii and Puerto Rico larger. Although the state sizes in the map won’t be precise, the map will be easier to read, as you can see in Figure 11-5.

[F11005.pdf]



* + - * 1. An easier-to-read map tweaked using tigris functions

Now try making the same map for all 3,000 counties by changing the geography argument to "county". Other geographies include region, tract (for census tracts), place (for census-designated places, more commonly known as towns and cities), and congressional district. There are also many more arguments in both the get\_decennial() and get\_acs() functions; I’ve shown you only a few of the most common. If you want to learn more, Walker’s book Analyzing US Census Data: Methods, Maps, and Models in R is a great resource.

Summary

This chapter explored two packages that use APIs to access data directly from its source. The googlesheets4 package lets you import data from a Google Sheet. It’s particularly useful when you’re working with survey data, as it makes it easy to update your reports when new results come in. If you don’t work with Google Sheets, you could use similar packages to fetch data from Excel365 (Microsoft365R), Qualtrics (qualtRics), Survey Monkey (surveymonkey), and other sources.

If you work with US Census Bureau data, the tidycensus package is a huge timesaver. Rather than having to manually download data from the Census Bureau website, you can use tidycensus to write R code that accesses the data automatically, making it ready for analysis and reporting. Because of the package’s integration with tigris, you can also easily map this demographic data.

If you’re looking for census data from other countries, there are also R packages to bring census data from Canada (cancensus), Kenya (rKenyaCensus), Mexico (mxmaps and inegiR), Europe (eurostat), and other regions. Before hitting that download button in your data collection tool to get a CSV, it’s worth looking for a package that can import that data directly into R.

Further Resources

Isabella Velásquez and Curtis Kephart, “Automated Survey Reporting with googlesheets4, pins, and R Markdown,” Posit, June 15, 2022, https://posit.co/blog/automated-survey-reporting/.

Kyle Walker, Analyzing US Census Data: Methods, Maps, and Models in R (Boca Raton, FL: CRC Press, 2023), https://walker-data.com/census-r/.