Automatically Accessing Online Data

So far, we’ve imported data into projects from CSV files. Many online datasets allow you to export data as a CSV, 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: using the googlesheets4 package to fetch data directly from Google Sheets and using the tidycensus package to access data from the United States Census Bureau. You’ll learn how to connect your R Markdown project to Google so you can automatically download data when a Google Sheet updates. Then you’ll explore working with two large census datasets, 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 can avoid having to manually download data, copy it into your project, and adjust your code so it imports that new data each 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 this package can help you manage complex datasets that update frequently. For example, in her role at the Primary Care Research Institute at the University of Buffalo, Megan 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, she was able to bring all of her data into one place and use R to put them to use. Data that had once been largely unused because accessing it was so complicated came to inform research on opioid use disorder.

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

Connecting to Google

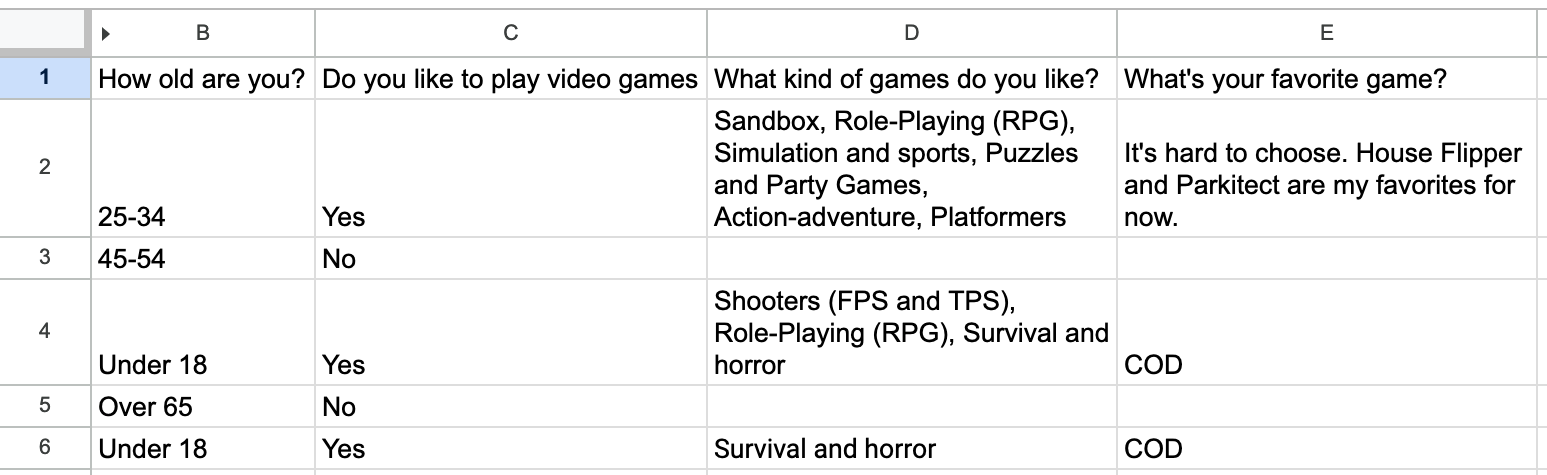
Begin by installing the googlesheets4 package by running install.packages("googlesheets4"). Next, you’ll need to connect to your Google account. To do this, run the gs4\_auth() function in the console. If you have more than one Google account, select the account that has access to the Google Sheet you 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 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 reauthenticate.

Reading Data from a Sheet

Now that we’ve connected R to our Google account, we can import data. We’ll import the fake data that Meghan Harris created about video game preferences (you can access it at https://data.rwithoutstatistics.com/google-sheet). Figure 10-1 shows what it looks like in Google Sheets.

[F10001.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:

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 can make it easier to read:

library(tidyverse)

survey\_data\_raw %>%

glimpse()

The glimpse() function, which creates one output row per variable, shows that we’ve indeed 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 we have the data in R, we can use the same workflow we always do when creating reports with R Markdown.

Using the Data in R Markdown

The following code 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 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")

```

The R Markdown document resembles those discussed in previous chapters except for the way we’re importing the data 1. 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 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, we use the clean\_names() function from the janitor package to make the variable names easier to work with. We then add a participant\_id variable using the row\_number() function, which adds a consecutively increasing number to each row. (we also make it a character using the as.character() function.) Next, we change several variable names with rename() before finally using mutate() to make the age variable into a data structure known as a factor; doing so will make sure the age variable shows up in the right order in our charts. I then use relocate() to put participant\_id before the age variable. We can now take a look at our survey\_data\_clean data frame using the glimpse() function again:

#> 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 a variety of 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 in the text using inline R code; create a table that shows the respondents broken down by age group 2; and generate a graph that shows how many respondents like video games 3. Figure 10-2 shows the resulting report.

[F10002.png]



* + - * 1. The rendered video game report

We can re-run the code at any point to bring in updated data. Our survey had five responses today, but if we 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, we read the data of the entire Google Sheet. It is, however, possible to import only a section of a Sheet. For example, the survey data we imported includes a timestamp column. This variable is added automatically whenever someone submits a Google Form that pipes data into a Google Sheet, but we don’t use it in our analysis, so we could get rid of it.

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

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

txh27MyZuiRdqScog/edit?usp=sharing",

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

glimpse()

This argument lets us specify a range of data to import. It uses the syntax you may have used to select columns in Google Sheets. In this example, we use range = "Sheet1!B:E" to import columns B through E (but not A, which contains the timestamp) before adding glimpse().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 functions in the googlesheets4 package that you can use. 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.

Accessing Census Data with tidycensus

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

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

This section shows you how the tidycensus package works using examples from two datasets to which it provides access: the Decennial Census administered every 10 years and the annual American Community Survey. We’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 allow online services to determine whether you are authorized to access data.

You can obtain this key, which is free, by going to https://api.census.gov/data/key\_signup.html and entering 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 will save your API key in your .Renviron file, which is designed for storing confidential information like API keys. 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 data. The most common of these are the Decennial Census and the American Community Survey. You can find a discussion of other datasets you can access in Chapter 2 of Kyle Walker’s book Analyzing US Census Data: Methods, Maps, and Models in R.

Working with Decennial Census Data

The Census Bureau puts out many datasets, several of which you can access using dedicated tidycensus functions. Let’s access data from the 2020 Decennial Census about the Asian population in each state using the get\_decennial() function with three arguments:

get\_decennial(geography = "state",

variables = "P1\_006N",

year = 2020)

The geography argument tells get\_decennial() to access data at the state level. In addition to the 50 states, it will return for the District of Columbia and Puerto Rico. There are many other geographies, including county, census tract, and more. The variables argument specifies the variable or variables we want to access. Here, P2\_002N is the variable name for the total Asian population. We’ll discuss how to identify other variables you may want to use in the next section. Lastly, year specifies 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

--snip--

The resulting data frame has four variables. 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 other geographies. NAME is the name of each state, and variable is the name of the variable we passed to the get\_decennial() function. Lastly, 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

To pass a specific variable to get\_decennial(), you have to first look it up. Let’s say we want to calculate the Asian population as a percentage of all people in each state. To do that, we’d first 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. This should pull data from so-called redistricting summary data files, which Public Law 94-171 requires the census to produce every 10 years:

load\_variables(year = 2020,

dataset = "pl")

Running this code returns the name, label (a description), and concept (a category) of all variables available to us:

#> # 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

--snip--

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

Using Multiple Census Variables

Now that we know which variables we need, we 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)

We add 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

--snip--

If you’re working with multiple census variables, however, you might have trouble remembering what names like P1\_001N and P1\_006N mean. Fortunately, we 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, we give the name we want our variables to have, followed by the equal sign and the original variable name. We can rename multiple variables by putting them within the c() function, as I’ve done here. It should now be much easier to see which variables we’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, we have total\_population and asian\_population. Much better!

Analyzing Census Data

Now we have the data we need to calculate the Asian population in each state as a percentage of the total. To do this, we add a few things 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")

We use group\_by(NAME) to create one group for each state because we want to calculate the Asian population percentage in each state (not for the entire United States). We then use mutate() to calculate each percentage, taking the value in each row and dividing it by the total\_population and asian\_population rows for each state. We use ungroup() to remove the state-level grouping. We use filter() to show only the Asian population percentage.

When we run this code, we see both 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

--snip--

This is one 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 we’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, he gives us the summary\_var argument that we can use within get\_decennial()to import the total population as a separate variable. Instead of putting P1\_001N (total population) in the variables argument, we 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 we got earlier, 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 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. Then we drop the summary\_value variable because we no longer need it after doing our 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 0.15393

#> 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

#> # ℹ 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. Let’s 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, we can access data from the American Community Survey in a manner nearly identical to how we access Decennial Census data. Instead of get\_decennial(), we use the function get\_acs(), but the arguments we pass to these functions are the same. In the following example, I’ve identified a variable I’m interested in (B01002\_001, which shows the median age) and use it to get this data for each state:

get\_acs(geography = "state",

variables = "B01002\_001",

year = 2020)

Here is 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

--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. It also produces an additional column called moe, for the margin of error. We see these changes because the American Community Survey is given to a sample of the population. As a result, we must extrapolate values from that sample to the population as a whole, and with such an estimate comes a margin of error.

In the 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.

Making Charts

Here is how we could take the data on median age and pipe it into ggplot to create a bar chart:

get\_acs(geography = "state",

variables = "B01002\_001",

year = 2020) %>%

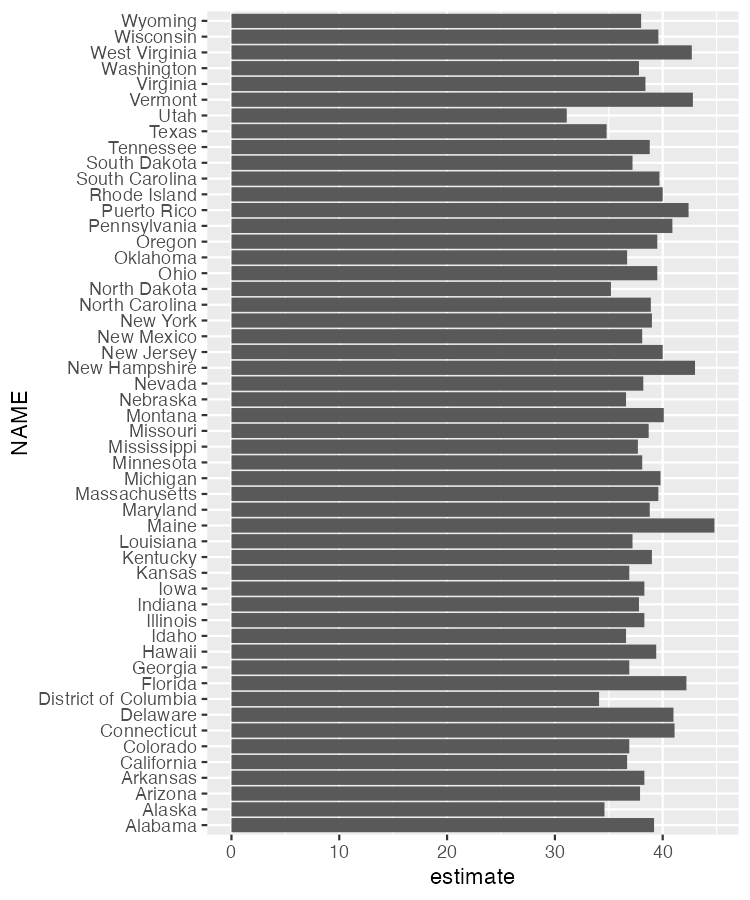
ggplot(aes(x = estimate,

y = NAME)) +

geom\_col()

We import data with the get\_acs() function and then pipe this directly into ggplot. We put states (which use the variable NAME) on the y axis and median age (estimate) on the x axis. A simple geom\_col() creates the bar chart, shown in Figure 10-3.

[F10003.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

Kyle Walker, the creator of tidycensus, 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)

If we take a look at the resulting data, we can see that it has the metadata and geometry column of the 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 see that the geometry type is MULTIPOLYGON, which you learned about in Chapter 4. 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)) +

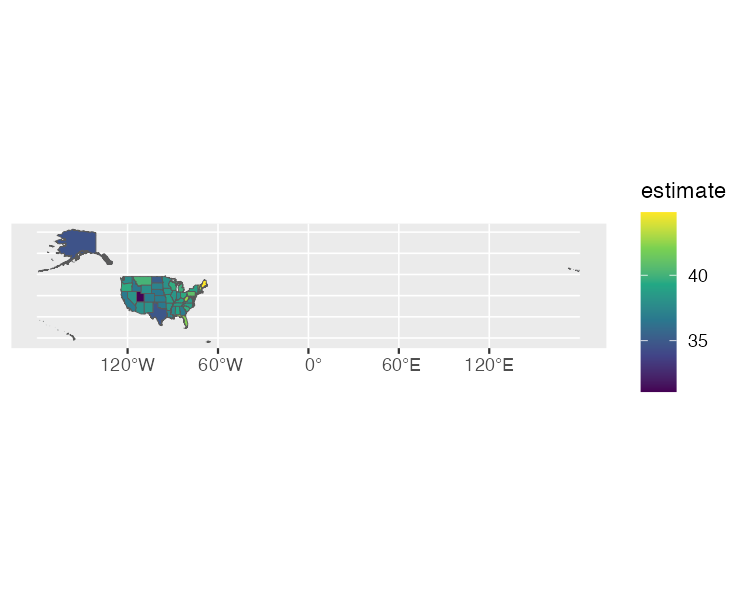
geom\_sf() +

scale\_fill\_viridis\_c()

Here, we import the data with the get\_acs() function before piping it into the ggplot() function. We set 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 we use geom\_sf() to draw the map. The scale\_fill\_viridis\_c() function gives us a colorblind-friendly palette.

The resulting map, seen in Figure 10-4, is less than ideal because 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 appears on one side of the map while a small part appears on the other side. What’s more, both Hawaii and Puerto Rico are hard to see.

[F10004.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) %>%

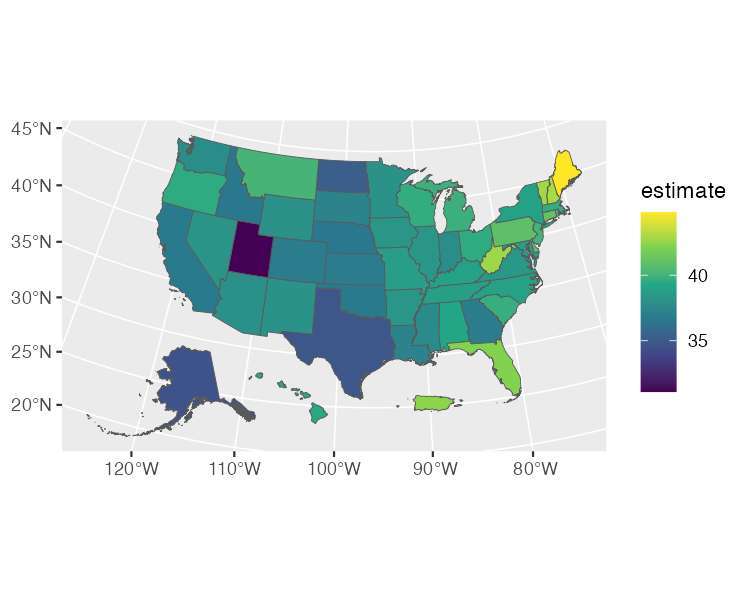
ggplot(aes(fill = estimate)) +

geom\_sf() +

scale\_fill\_viridis\_c()

We set the preserve\_area argument to FALSE to shrink the giant state of Alaska and make 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 10-5.

[F10005.pdf]



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

We’ve made a map that shows the median age by state. As an exercise, 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), congressional district, and more. 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 shown only a few of the most common. If you want to learn more, Walker’s book is a great resource.

Conclusion

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

If you’re looking for census data from other countries, Walker’s Analyzing US Census Data book gives examples of packages that can help. There are R packages to bring census data from Canada (cancensus), Kenya (rKenyaCensus), Mexico (mxmaps and inegiR), Europe (eurostat), and other countries. Before hitting the download button in your data collection tool of choice, it’s worth looking for a package that can import that data directly into R.

Learn More

Consult the following resources to learn how to access data from Google Sheets using the googlesheets4 package and from the Census Bureau using the tidycensus package:

Automated survey reporting with googlesheets4, pins, and R Markdown by Isabella Velásquez and Curtis Kephart (2022), https://posit.co/blog/automated-survey-reporting/

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