8. Data Retrieval

Self-Paced Workshop #8

Prof. Jack Reilly

Fall 2026

Asking for the Data Instead of Downloading It

Every dataset so far this semester arrived as a file: you downloaded it, put it in a folder, and pointed a file read command (`read_csv()` or `read_dta()`) at it. This read it from your file system into working memory so you could manipulate, analyze, and visualize it.

This week, we looked at a second way to get data into working memory. A package can *fetch* the data for you, straight from the organization that publishes it, so the script itself says exactly where the data came from and anyone who runs it gets the same thing. No filesystem to worry about, no data to download manually, no file to email, no "which version did I download and where did I put it?" This process is both more reproducible and easier - win/win.

This workshop works through the two examples from lecture:

- **The General Social Survey**, with Kieran Healy's `gssr` package (and its companion `gssrdoc`), which packages the whole cumulative GSS file — the same one you used for Problem Set 6 — inside R.
- **The US Census**, with Kyle Walker's `tidycensus` package, which sends your request to the Census Bureau's servers and hands back a tidy table.

  Part 1 runs here; Part 2 runs in your own R

  Part 1, the GSS, runs in the browser, with one exception at the end. Part 2, the Census, does not: `tidycensus` has to reach the Census Bureau over the internet with your own key, which the in-browser R cannot do. For Part 2, open RStudio or Positron, start a new `.R` script, and copy each block into it as you go (the copy icon is in the top-right corner of each block). **You must get your Census key in order to do this** (see below).

  Before you start

  - **Get a Census API key now** if you have not already: sign up here. It arrives by email, usually within a few minutes, and has to be activated from the link in that email. You need it for Part 2.

- **Give the page a moment.** `gssr` is a 30 MB download, because the data is inside the package, so the first box takes several seconds longer than usual (and longer if you are on a slow connection).
- As usual, the boxes on this page share one R session; run them top to bottom or you'll run into errors.

Part 1. The General Social Survey with `gssr`

Step 1. Load the packages

```
library(dplyr)
library(ggplot2)
library(gssr)
```

`gssr` is not on CRAN, so installing it at home takes one extra argument; Step 6 has the line. Here in the browser it is already installed.

Step 2. The GSS, already inside R

`gssr` ships the GSS *as data inside the package*. `data()` loads a dataset from a package into your session by name. Start with the smaller of the two cumulative files, `gss_sub`:

```
data(gss_sub)
gss_sub
```

Every GSS respondent from 1972 to 2024 — about 75,000 rows — but only 20 of the survey's variables, chosen by the package author as an example. Its big sibling, `gss_all`, is the same rows with all of the nearly 7,000 variables, and it is too large for a browser tab; you will meet it in Step 6.

Look at the column types in the printout. They are `<dbl+lbl>`: a number with its label attached, exactly what `read_dta()` gave you from the Stata file in Problem Set 6. Now compare this process to that of Problem Set 6, where getting this far without the course drive would have meant finding the right file on the NORC website, downloading it, unzipping it, and reading it in. (And even with the dataset already on the course drive for you, it still meant moving the file around your filesystem.) Here it is one line, and the line documents itself.

Step 3. One year

A single survey year is a `filter()` away:

```
gss18 <- gss_sub |>
  filter(year == 2018)
gss18
```

2,348 respondents. The labels are what make the codebook unnecessary for a quick look; to *use* them in a table or a graph, turn the column into a factor with `as_factor()` from `haven` (installed alongside `gssr`):

```
gss18 |>
  mutate(degree = haven::as_factor(degree)) |>
  count(degree)
```

Try it yourself: pick a different year and count `polviews` instead. The GSS ran most years from 1972 to 1994 and every other year from 1996 to 2018; the 2020 survey was fielded in 2021 because of the pandemic, and was followed by surveys in 2022 and 2024.

```
gss_sub |>
  filter(year == 2018) |>
  mutate(degree = haven::as_factor(degree)) |>
  count(degree)
```

Step 4. Which years was it asked?

The catch with a fifty-year survey is that most questions were not asked every year. Before you build an analysis around a variable, check when it exists. `gss_which_years()` does that for any set of variables:

```
gss_sub |>
  gss_which_years(c(fefam, polviews, degree, padeg)) |>
  print(n = Inf)
```

One row per survey year, `TRUE` where the question was asked (`print(n = Inf)` prints every row instead of the first ten). `degree` and `padeg` are there from the start; `polviews` begins in 1974; and `fefam` — whether it is better for the man to work and the woman to tend the home — was asked once in 1977, then intermittently in the 1980s, then every time since 1988. Notice what this would have taken by hand: a search through the codebook for each variable, one at a time.

Try it yourself: `names(gss_sub)` lists the 20 variables. Check the years for `relig`, `race`, and `wtssall` — one of those is a weight, and it is not there in every year.

```
names(gss_sub)
```

Step 5. From the survey to a graph

Put Steps 3 and 4 together. Take `fefam`, turn it into a share per year, and draw the line:

```
gss_sub |>
  select(year, fefam) |>
  filter(!is.na(fefam)) |>
  group_by(year) |>
  summarise(agree = mean(fefam <= 2)) |>
  ggplot(aes(x = year, y = agree)) +
```

```
  geom_line() +
  geom_point() +
  scale_y_continuous(labels = scales::label_percent(), limits = c(0, NA)) +
  labs(title = "Fewer Americans say a woman's place is in the home",
       subtitle = "Agree it is better for the man to work and the woman to tend the home",
       x = NULL, y = NULL,
       caption = "Source: General Social Survey 1977-2024, via gssr.")
```

`fefam` runs from 1 (strongly agree) to 4 (strongly disagree), so `fefam <= 2` is `TRUE` for anyone who agrees, and the mean of a `TRUE/FALSE` column is the share that is `TRUE`.[1] The `geom_point()` matters here: it shows the years the question was actually asked, so the long gap after 1977 is not mistaken for data. The `select()` on the second line is the important habit with big survey files: pull the columns you need out of the giant table first, then do everything else on the small one.

Try it yourself: draw the same line for the share with a bachelor's degree or more (`degree >= 3`). Check the labels first with `haven::as_factor()`, as in Step 3, so you know what `3` means.

```
gss_sub |>
  select(year, fefam) |>
  filter(!is.na(fefam)) |>
  group_by(year) |>
  summarise(agree = mean(fefam <= 2)) |>
  ggplot(aes(x = year, y = agree)) +
  geom_line() +
  geom_point()
```

This idea — pick the rows and columns you need from a central store *before* the data reaches your analysis — is the same one behind a SQL query against a database. `gssr` just makes the store a package.

Step 6. In your own R: the full GSS

Everything above works the same at home, plus three things the browser cannot do. Install the packages in the console rather than in a script — installing is a one-time change to your computer, while the script is the analysis itself and gets run again every time.[2] They are not on CRAN, the usual place `install.packages()` looks, but on Healy's own *R-universe* repository, so the install names that repository, with CRAN second for anything they depend on:

[1] This is an unweighted share, which is fine for a first look. For a published estimate you would weight it, as in Workshop 5; the GSS weight for all years is `wtssps`, which is in `gss_sub`. Try `summarise(agree = weighted.mean(fefam <= 2, wtssps))`.

[2] This is the same rule as `install.packages("tidyverse")` back in Week 1: a script that reinstalls packages every time it runs is slow, and rude to anyone else who runs it. Put the `library()` lines in the script, and the installs nowhere.

```
# Install 'gssr' from the author's R-universe
install.packages("gssr", repos =
  c("https://kjhealy.r-universe.dev",
    "https://cloud.r-project.org"))

# Also recommended: install 'gssrdoc'
install.packages("gssrdoc", repos =
  c("https://kjhealy.r-universe.dev",
    "https://cloud.r-project.org"))
```

One year, every variable. `gss_get_yr()` downloads a single year's complete file from NORC, the survey's home, as it runs:

```
library(tidyverse)
library(gssr)

gss18 <- gss_get_yr(2018)
gss18
```

That is the entire 2018 GSS: the same 2,348 respondents as Step 3, but with all 1,146 of that year's variables. It cannot run here only because NORC's website refuses downloads requested from inside a web page. `gss_get_yr()` takes one year at a time.

Every year, every variable. `gss_all` is the full cumulative file:

```
data(gss_all)

gss_all |>
  gss_which_years(c(industry, indus80, wrkgovt, commute)) |>
  print(n = Inf)
```

About 75,000 rows by nearly 7,000 columns. It takes a few seconds to load and about **4 GB of memory** once it has, so you may want to close anything else you have open, especially on a laptop with 8 GB.[3] The lecture's example then shows `industry` running from 1972 to 1990, `indus80` taking over from 1988, and `commute` asked in exactly one year.

The codebook inside R. `gssrdoc` puts the GSS codebook into R's help system, so every GSS variable has a help page:

```
library(gssrdoc)
?fefam
```

The help pane shows the question wording, the response categories with their counts by year, and the years and ballots the question was asked on. That is the step most people skip — and the one that catches the problem when a question's wording changed, or it was only

[3] If you have 16 GB or more of RAM, this is probably less of an issue. If your computer struggles, or as a habit regardless, `select()` the handful of variables you need into a new, smaller object straight away (`gss_small <- gss_all |> select(year, id, wrkstat, industry)`) and then `rm(gss_all)` to free the memory.

asked of some respondents.

Part 2. The Census with `tidycensus`

The Census is a different kind of thing. It is not one file you could put in a package; it is thousands of tables, at every level of geography from the nation down to the city block, for many years and several surveys. The Census Bureau serves all of it through an *API* — a web address that answers requests for data — and `tidycensus` writes the requests for you.

Everything from here on needs your key and a live connection to the Census Bureau, which the browser cannot provide, so open up RStudio or Positron and work alongside this page.

Step 7. Install, and store your API key, once

`tidycensus` is on CRAN. In the console:

```
install.packages("tidycensus")
```

The API wants your key with every request. Give it to `tidycensus` once, also in the console:

```
library(tidycensus)
census_api_key("PASTE YOUR KEY HERE", install = TRUE)
readRenviron("~/.Renviron")
```

`install = TRUE` saves the key in a settings file in your home folder (`.Renviron`), which R reads every time it starts. From now on `tidycensus` finds it on its own, and **your scripts never contain the key.** That matters: a key in a script ends up on GitHub the first time you push, and a published key is one anyone can use under your name.[4] Run this in the console, not your script, for the same reason.

Step 8. A first request: median age by state

```
age20 <- get_decennial(geography = "state",
                       variables = "P13_001N",
                       year = 2020,
                       sumfile = "dhc") # demographic and housing file
head(age20)
```

Each argument is one part of the request:

Argument	Says	Here
`geography`	which units, one row each	the 50 states, DC and Puerto Rico

[4] If you ever do push a key by accident, deleting it in the next commit does not help — it is still in the history. Request a new key and stop using the old one.

Argument	Says	Here
`variables`	which numbers, by the Census Bureau's code	`P13_001N`, median age
`year`	which census	2020
`sumfile`	which collection of tables from that census	the Demographic and Housing Characteristics file

The result is a tibble with four columns: `GEOID` (the state's FIPS code), `NAME`, `variable`, and `value`. Connecticut's median age is 41.1; DC's is 33.9.

Step 9. Graph it

The data comes back clean, so it goes straight into `ggplot()`:

```
age20 |>
  ggplot(aes(x = value, y = reorder(NAME, value))) +
  geom_point()
```

A dot plot, one row per state. `reorder(NAME, value)` sorts the states by their median age instead of alphabetically, which is what turns fifty-odd labels into something readable.

Try it yourself: finish the graph the way Workshop 7 ended — axis labels in words, a title that states the finding, a caption naming the source, a theme.

Step 10. Finding variable codes

Nobody remembers that `P13_001N` is median age. `load_variables()` downloads the full list of variables for a census and file — in effect, the codebook — into a tibble you can search:

```
vars <- load_variables(2020, "dhc", cache = TRUE)
vars
```

`cache = TRUE` saves a copy so the next call is instant. The `name` column is the code; `label` and `concept` describe it. Search the concepts with `str_detect()` from `stringr` (part of the tidyverse); `regex(..., ignore_case = TRUE)` matches regardless of capitals:

```
vars |>
  filter(str_detect(concept, regex("population", ignore_case = TRUE))) |>
  View()
```

`View()` opens the result as a spreadsheet you can scroll and filter. Find `P1_001N`, total population.

Try it yourself: search the concepts for `"median age"`, and confirm `P13_001N` is what Step 8 said it was. Then search for `"race"`.

Step 11. Total population, with readable axis labels

```
totalpop <- get_decennial(geography = "state",
                          variables = "P1_001N",
                          year = 2020,
                          sumfile = "dhc")

totalpop |>
  ggplot(aes(x = value, y = reorder(NAME, value))) +
  geom_point() +
  scale_x_continuous(
    breaks = c(10000000, 20000000, 30000000, 40000000),
    labels = c("10m", "20m", "30m", "40m"))
```

Without the `scale_x_continuous()` line the axis reads `1e+07`, `2e+07`... in scientific notation. Writing the breaks and labels by hand works; the `scales` package can also do it for you, which keeps working if the data changes. Swap this in for the `scale_x_continuous()` line above:

```
  scale_x_continuous(labels = scales::label_number(scale = 1e-6, suffix = "m"))
```

Step 12. A ratio: two variables, wide

Counts are rarely the point. Usually you want a share, which means two variables — a part and a total — in one request. Name them as you ask, and ask for the table `wide`:

```
pop <- get_decennial(geography = "state",
                     variables = c(total = "P1_001N",
                                   asian = "P3_005N"),
                     year = 2020,
                     sumfile = "dhc",
                     output = "wide") # without this, it comes back long

pop <- pop |>
  mutate(asian_share = asian / total)

pop |>
  ggplot(aes(x = asian_share, y = reorder(NAME, asian_share))) +
  geom_point()
```

Two things happened in the request. The names in `c(total = ..., asian = ...)` became the column names, instead of the codes. And `output = "wide"` put the two variables side by side, one row per state, so `mutate()` can divide one by the other. Without it you get the long shape from Step 8 — two rows per state, one per variable — which is tidy, but needs a `pivot_wider()` before you can do arithmetic across variables.

`P3_005N` is "Asian alone" from table P3, *Race*, which counts everyone.[5] Checking a code against its label before you trust it is the single most useful habit with Census data; the codes are close together and one digit changes the meaning entirely.

Try it yourself: look up `P3_005N` and `P10_007N` in `vars` to see the difference. Then change the request to the share Black or African American alone (also in table P3) and redraw the graph.

Step 13. Smaller geography

The same call works at every level of geography; smaller units usually need a `state` (and sometimes a `county`) to say which ones. Median age for every county in New York:

```
ny_age <- get_decennial(geography = "county",
                        state = "NY",
                        variables = "P13_001N",
                        year = 2020,
                        sumfile = "dhc")

ny_age |>
  ggplot(aes(x = value, y = reorder(NAME, value))) +
  geom_point()
```

Sixty-two counties, from the five boroughs of New York City to Hamilton County in the Adirondacks. Find Onondaga. The labels all end in ", New York"; `str_remove(NAME, " County, New York")` inside a `mutate()` cleans them up.

Try it yourself: change `geography` to `"tract"` and add `county = "Onondaga"` for every census tract in the county — well over a hundred rows, from the same one call.

A Practical Word

You could get every number on this page without code. The Census Bureau's data.census.gov will build and download the same tables, and the GSS is a file you already know how to open. So why bother with all this?

Two reasons:

1) **Simplicity**: This is actually *easier* once you get used to it! No filesystem, no file downloads, less local management. Easier is better!

2) **Reproducibility**: The script is now a more complete record. The goal of all analysis is to move from original source files to output in a fully documented way, and a manual file download broke - or at least complicated - that chain. A file downloaded from a

[5] The lecture slide used `P10_007N`. Look it up in `vars`: table P10 is *Race for the population 18 years and over*, and its row 7 is Native Hawaiian and Other Pacific Islander alone. Divided by total population, it is neither the right group nor the right denominator, and the graph would have looked perfectly reasonable anyway.

website carries no memory of which table, which year, which geography, or which file you clicked; a `get_decennial()` call carries all four, and re-running it next year — or handing it to someone else — reproduces the data exactly. For the GSS the gain is mostly convenience. For the Census, which is too large and too finely divided to keep a copy of, a scripted request is essentially the only sensible way to work.

Work on Your Own

Two packages, one idea: ask the source for exactly the rows and columns you need, and let the script document the request. For more, the gssr site has a longer tour of the GSS data, and Kyle Walker's free book *Analyzing US Census Data* covers `tidycensus` in depth — in particular, look at chapter 2, which covers both the decennial census you used here and `get_acs()` for the American Community Survey, which is where most current Census data lives.