US Census Bureau is probably the most reliable souce of demographic data on the US population. There are hundreds of use cases of the data from Us Census Bureau. In one of our recent projects we used household income data from Census as a proxy to missing actual household income data in our table.

In this article, I’ll show how we got the income data from ACS5 survey using the ‘tidycensus’ package. In doing so, I’ll show how to write your own function to wrap the out of the box function to serve your customised purpose. As bonus I’ll briefly describe how you may automate this entire process so that you can run this process from a server, save the data in your database and each year automatically update the table with the new Census data.

Brief Background on US Census Surveys

There are different types of surveys conducted by the Census Bureau. Not all of them have the same data or data at the same level based on their nature. Here you can find details about the different survey types and other details: https://www.census.gov/programs-surveys.html. For our purpose we needed a survey that has household income data at the most grannular geographic level. We ended up using the American Community Survey ACS5 survey because this survey contains household income data at the Census Tract level which is the most grannular level for which household income data is available in census.

Preparation

Getting your API Key

For this project we have used an R package called *tidyverse*. Behind the scene *tidycensus* package calls the API provided by the Census Bureau. To call this API you need to have a key, which is basically is unique ID that is automatically generated against each new user by the Census Bureau. You can get a key from here: https://api.census.gov/data/key_signup.html.

Once you have the key save it in a separte text file or an r script. So that later on we can load the file and use that key in our functions.

Setting up the Key

Once you have the API key available you need to set it up to be used by the functions from tidycensus package. You can set it up using following line of code:

```
census_api_key('YOUR_API_KEY', install = FALSE, overwrite = TRUE)
```

Now we have all the prerequisits set to call the Census API.

Calling Census API to Get Tract Household Income

Using the following line of code you can get all the tract level household income data for an entire state in the US. We are using this variable *B19013_001* from the ACS5 survey. You can learn more about all the available variables going to this link: https://api.census.gov/data/2018/acs/acs5/variables.html
So to get tract level median household income in IL for the ACS5 survey of 2018 you can run the following

line of code:

```
get_acs(state = 'IL', year = 2018, geography = 'tract', variables =
'B19013_001',
                                      geometry = FALSE, survey = 'acs5',
show_call = TRUE)
```

Explanation of the code

get_acs() function is used to call the API for the ACS surveys. You should read through the code description for details but here is the brief description of the parameters used in this case:

- state: abbreviated name or names of the state(s),
- year: for which year's ACS survey you are looking for.
- geography: at which geographic level you are looking at. We are looking for *tract* level data. Other options are county, block etc.
- survey: which survey you are looking for. We are looking for **ACS5* survey,
- show_call: setting it up as true prints the message output in the R console while the API is called. Really helpful when you put this function in production to check back reason behind in case the function fails.

Limitations of this Code

Our goal is to fetch that data, store the data in a database and then schedule that script to be re-run automatically every year. Using this out of the box function doesn't serve that entire purpose because of these limitations:

- It has to have a state name as an input. We could make it dynamic so that we could use it to load all states' data or just a selective set of states or just one state,
- We want to schedule this to run automatically at a specific date. But since the ACS5 survey publication date is not exactly same each year, we need to have some flexibility so that in case our desired year's survey isn't populated the function falls back to fetch latest available data.
- We would like to have a column with the date and time recorded when this data is called and store that along with other data in the database table.

In the next sections I'll walk you through writing a customised functions which will address these limitations.

Overcoming the Limitations

Before we start to building the elaborate function, I'll start with a basic wrapper function. And then we'll keep adding additional argumnts to it to overcome the limitations.

Here's how R function skeleton looks like:

```
 functionName = function(input01, input2){
                  Logic
        }
```

You give it a name so that in future you can save the function and reuse it. Inside the function() within parentheses you include the input variable name(s). And you write the logic inside the curly braces.

Now let's write a basic function to wrap the two pieces of codes we have written earlier to get ACS data:

```
 getAcsIncome = function(names, year, KEY = 'YOUR_KEY'){
        ## setting up API call key
        census_api_key(apiKey, install = FALSE, overwrite = TRUE)

        ## calling get_acs()
        get_acs(state = names, year = year, geography = 'tract', variables =
'B19013_001',
                                      geometry = FALSE, survey = 'acs5',
show_call = TRUE)
        }
```

I have saved my API KEY in a separate script. So I have loaded the script and using the KEY from the script to get track level data for IL from 2018 ACS5 survey.

```
# Loading libraries and key
library(tidycensus)
source('KEY.R')

# Wrapper function
getAcsIncome = function(names, year, KEY){
        ## setting up API call key
        census_api_key(key = API_KEY, install = FALSE, overwrite = TRUE)

        ## calling get_acs()
        get_acs(state = names, year = year, geography = 'tract', variables =
'B19013_001',
                                         geometry = FALSE, survey = 'acs5',
show_call = TRUE)
}

# Calling the function and display glimpse of result
IL_HH_Income = getAcsIncome(names = 'IL', year = 2018, KEY = API_KEY)
head(IL_HH_Income)
```

```
## # A tibble: 6 x 5
##   GEOID       NAME                                        variable  estimate
moe
##
## 1 17001000100 Census Tract 1, Adams County, Illinois    B19013_0~    44613
6384
## 2 17001000201 Census Tract 2.01, Adams County, Illinois B19013_0~    44878
4356
## 3 17001000202 Census Tract 2.02, Adams County, Illinois B19013_0~    46964
10202
## 4 17001000400 Census Tract 4, Adams County, Illinois    B19013_0~    33750
7386
## 5 17001000500 Census Tract 5, Adams County, Illinois    B19013_0~    38526
4846
## 6 17001000600 Census Tract 6, Adams County, Illinois    B19013_0~    51491
10117
```

Making the state name input flexible

Now we have a operating function, we'll move to the next steps where we'll add first set of arguments to it to make the state name input flexible.

We'll use a built-in constant in R namely: state.abb. It includes the 50 state name abbreviations. In our customised wrapper function we'll add changes to address these following use cases:

- download all states data when input is 'all'/'ALL'
- download selected state(s) data when input is one/multiple state names in abbreviations
- provide an error message if provided input doesn't match any of the above two input types

```
# Wrapper function
getAcsIncome = function(names, year, KEY){
        ## setting up API call key
        census_api_key(key = API_KEY, install = FALSE, overwrite = TRUE)
        ## setting up blank array to store state names
        stateNames = NULL

        # when all states are required
        if(names %in% c('all', 'ALL')){
          stateNames = state.abb
        }

        # when specific state or states are mentioned in names
        else if(names %in% c(state.abb)){
          stateNames = names
        }

        # in any other cases
        else{
          print("Provide a value in stateNames variable. Available options:
all/ALL/any of the 50 states (abb.)")
        }

        ## calling get_acs()
        get_acs(state = stateNames, year = year, geography = 'tract', variables
= 'B19013_001', geometry = FALSE, survey = 'acs5', show_call = TRUE)
}

head(getAcsIncome(names = 'all', year = 2018, KEY = API_KEY))
```

```
## # A tibble: 6 x 5
##   GEOID       NAME                                      variable  estimate
moe
##
## 1 01001020100 Census Tract 201, Autauga County, Alabama B19013_0~    58625
14777
## 2 01001020200 Census Tract 202, Autauga County, Alabama B19013_0~    43531
6053
## 3 01001020300 Census Tract 203, Autauga County, Alabama B19013_0~    51875
8744
## 4 01001020400 Census Tract 204, Autauga County, Alabama B19013_0~    54050
5166
## 5 01001020500 Census Tract 205, Autauga County, Alabama B19013_0~    72417
14919
## 6 01001020600 Census Tract 206, Autauga County, Alabama B19013_0~    46688
13043
```

Adding fall back capability in the year input

To add that capability we'll use a package called *tryCatchLog*. The basic sceleton of tryCatch() function that we'll use is like following:

```
result = tryCatch({
    expr
}, warning = function(w) {
    warning-handler-code
}, error = function(e) {
    error-handler-code
}, finally = {
    cleanup-code
}
```

Here inside the curly braces you add the code to evaluate and inside second function, following warning/error, provide the logic to execute if the first code block fails. The above skeleton was copied from this article. That article has a more detailed discussion on how to apply try catch function.

In our case we'll use trycatch function to update a variable. Then we'll add additional code block that will run based on the value of that variable. Also if the first code block fails, we'll print out a message where the error message will be printed starting with the date showing which year it tried.

The tryCatch block of our code inside the function will look like following:

```
  # starting with variable: an.error.occured with value of FALSE
  an.error.occured <- FALSE
  tryCatch({

    # trying for current year - 2
    year = as.numeric(substr(Sys.Date(), start = 1, stop = 4)) - 2

    # calling api to get data
    data = tidycensus::get_acs(state = name, year = year, geography = 'tract',
variables = 'B19013_001', geometry = FALSE, survey = 'acs5', show_call = TRUE)
    }, error = function(e) {

    # updating the variable
    an.error.occured <<- TRUE
    # printing out error message to be stored in log with the
    message(paste0("Year tried: ", year, "/n", e))})
```

In the above block we are capturing if our first try of the code block fails. If it fails we are updating *an.error.occured* variable to TRUE. Which will trigger the next block where we'll use one year older year value.

Eventually the final function with the added full trycatch functionality will look like this:

```
getAcsIncome = function(names, year, KEY){

   ## setting up API call key
        census_api_key(key = API_KEY, install = FALSE, overwrite = TRUE)

        ## setting up blank array to store state names
        stateNames = NULL

        # when all states are required
        if(names %in% c('all', 'ALL')){
          stateNames = state.abb
        }

        # when specific state or states are mentioned in names
        else if(names %in% c(state.abb)){
          stateNames = names
        }

        # in any other cases
        else{
          print("Provide a value in stateNames variable. Available options:
all/ALL/any of the 50 states (abb.)")
        }
```

```
  # starting with variable: an.error.occured with value of FALSE
  an.error.occured <- FALSE
  tryCatch({

    # calling api to get data
    data = tidycensus::get_acs(state = stateNames, year = year, geography =
'tract', variables = 'B19013_001', geometry = FALSE, survey = 'acs5', show_call
= TRUE)
    }, error = function(e) {

    # updating the variable
    an.error.occured <<- TRUE
    # printing out error message to be stored in log
    message(paste0("Year tried: ", year, "\n", e))})


  #  try for 2 year older data
  if(an.error.occured == TRUE){
    year = year - 2

    # calling api to get data
    data = tidycensus::get_acs(state = stateNames, year = year, geography =
'tract', variables = 'B19013_001', geometry = FALSE, survey = 'acs5', show_call
= TRUE)
  }

  ## returning resulting data
  return(data)
}

head(getAcsIncome(names = 'IL', year = 2020, KEY = API_KEY))
```

```
## To install your API key for use in future sessions, run this function with
`install = TRUE`.

## Getting data from the 2016-2020 5-year ACS

## Census API call: https://api.census.gov/data/2020/acs/acs5?get=B19013_001E%
2CB19013_001M%2CNAME&for=tract%3A%2A&in=state%3A17

## Year tried: 2020
## Error: Your API call has errors.  The API message returned is
```

HTTP Status 404 - /data/2020/acs/acs5

.

```
## Getting data from the 2014-2018 5-year ACS

## Census API call: https://api.census.gov/data/2018/acs/acs5?get=B19013_001E%
2CB19013_001M%2CNAME&for=tract%3A%2A&in=state%3A17

## # A tibble: 6 x 5
##   GEOID       NAME                                      variable   estimate
moe
##
## 1 17001000100 Census Tract 1, Adams County, Illinois    B19013_0~    44613
6384
```

```
## 2 17001000201 Census Tract 2.01, Adams County, Illinois B19013_0~    44878
4356
## 3 17001000202 Census Tract 2.02, Adams County, Illinois B19013_0~    46964
10202
## 4 17001000400 Census Tract 4, Adams County, Illinois    B19013_0~    33750
7386
## 5 17001000500 Census Tract 5, Adams County, Illinois    B19013_0~    38526
4846
## 6 17001000600 Census Tract 6, Adams County, Illinois    B19013_0~    51491
10117
```

Among the messages printed, this following message block shows that our code block inside the trycatch function failed. Then it fell back to 2 year's older data. The reason is the latest survey data available in ACS5 is for 2018.

```
## Year tried: 2020
## Error: Your API call has errors.  The API message returned is
```

HTTP Status 404 - /data/2020/acs/acs5

.

Before we move on to adding our next argument block to overcome the final limitation, we need to make one more change. Since our eventual goal is to run this function from a server, let's make the year input embedded inside the function.

We'll introduce a variable named *year* inside the function with a default value of (current year – 2) value and then in the fall back we'll update that variable to (current year – 3). Which will make sure that whenver we run the code, it'll ask for the 2 year older data and even if that 2 year data is not available it'll call for 3 year older data.

Here's the two lines of codes that will be added:

```
# creating year variable with default value
    year = as.numeric(substr(Sys.Date(), start = 1, stop = 4)) - 2

    #updating year variable
    year = as.numeric(substr(Sys.Date(), start = 1, stop = 4)) - 3
```

You can see the final code chunk with that year functionality added.

Adding a column for data and time

This is the simplest part of this tutorial. Basically we'll add Sys.time() as an additional column to the already fetched data.

Here's the final code chunk:

```
getAcsIncome = function(names, KEY){

   ## setting up API call key
        census_api_key(key = API_KEY, install = FALSE, overwrite = TRUE)

        ## setting up blank array to store state names
        stateNames = NULL

        # when all states are required
        if(names %in% c('all', 'ALL')){
          stateNames = state.abb
        }
        # when specific state or states are mentioned in names
        else if(names %in% c(state.abb)){
          stateNames = names
        }

        # in any other cases
        else{
          print("Provide a value in stateNames variable. Available options:
all/ALL/any of the 50 states (abb.)")
        }

  # starting with variable: an.error.occured with value of FALSE
  an.error.occured <- FALSE
  tryCatch({

    # creating year variable with default value
    year = as.numeric(substr(Sys.Date(), start = 1, stop = 4)) - 2

    # calling api to get data
    data = tidycensus::get_acs(state = stateNames, year = year, geography =
'tract', variables = 'B19013_001', geometry = FALSE, survey = 'acs5', show_call
= TRUE)
    }, error = function(e) {

    # updating the variable
    an.error.occured <<- TRUE
    # printing out error message to be stored in log
    message(paste0("Year tried: ", year, "\n", e))})


  #  try for 2 year older data
  if(an.error.occured == TRUE){

    #updating year variable
    year = as.numeric(substr(Sys.Date(), start = 1, stop = 4)) - 3

    # calling api to get data
    data = tidycensus::get_acs(state = stateNames, year = year, geography =
'tract', variables = 'B19013_001', geometry = FALSE, survey = 'acs5', show_call
= TRUE)
  }

  # adding update data to a column
  data$UPDATE_DATE = Sys.time()

  ## returning resulting data
  return(data)
}

summary(getAcsIncome(names = 'all', KEY = API_KEY))
```

```
##     GEOID               NAME              variable            estimate
##  Length:72877       Length:72877       Length:72877       Min.   :  2499
##  Class :character   Class :character   Class :character   1st Qu.: 42353
##  Mode  :character   Mode  :character   Mode  :character   Median : 57099
##                                                           Mean   : 64289
##                                                           3rd Qu.: 78323
##                                                             Max.   :250001
##                                                             NA's   :1013
##       moe          UPDATE_DATE
##  Min.   :   550   Min.   :2020-07-10 09:03:57
##  1st Qu.:  6051   1st Qu.:2020-07-10 09:03:57
##  Median :  8711   Median :2020-07-10 09:03:57
##  Mean   : 10212   Mean   :2020-07-10 09:03:57
##  3rd Qu.: 12521   3rd Qu.:2020-07-10 09:03:57
##  Max.   :126054   Max.   :2020-07-10 09:03:57
##  NA's   :1092
```

What's next?

There are two things left now to set this script in a server to be run automatically:

- Adding log file. Anytime you want to keep a script running from a server, you should consider adding logging capability to it. It'll come real handy to debug in case the script fails.
- Automating this script. One easy way in Windows is to use windows' task scheduler. You can take a look at my other tutorial[Automate Your Repetitive Reports!]](https://curious-joe.net/post/automate-your-repetitive-reports/) to know detail about how to automate a script using windows task scheduler.

US Census Bureau is a great source of data on the US population. There are all sorts of interesting data available such as unemployment data, race related data, education related data and so on. …