Data Science for Public Policy

Aaron R. Williams and Alena Stern - Georgetown University

Web Application Programming Interfaces (Web APIs)

Reading

- Read more about custom functions in R4DS
- Read more about `library(purrr)` in R4DS.
- Why we built the Education Data Portal
- Why we built an API for Urban's Education Data Portal

More R Programming

Custom Functions 创建自己的函数

R has a robust system for creating custom functions. To create a custom function, use `function()`:

```r
say_hello <- function() {

  "hello"

}

say_hello()
```

```
## [1] "hello"
```

Oftentimes, we want to pass parameters/arguments to our functions:

```
say_hello <- function(name) {

  paste("hello,", name)

}

say_hello(name = "aaron")
```

```
## [1] "hello, aaron"
```

We can also specify default values for parameters/arguments:

```
say_hello <- function(name = "aaron") {

  paste("hello,", name)
```

这里，将“aaron”设置为default值

```
}

say_hello()
```

```
## [1] "hello, aaron"
```

```
say_hello(name = "alex")
```

```
## [1] "hello, alex"
```

`say_hello()` just prints something to the console. More often, we want to perform a bunch of operations and the then return some object like a vector or a data frame. By default, R will return the last unassigned object in a custom function. It isn't required, but it is good practice to wrap the object to return in `return()`.

It's also good practice to document functions. With your cursor inside of a function, go Insert > Insert Roxygen Skeleton:

```
#' Say hello
#'
#' @param name A character vector with names
#'
#' @return A character vector with greetings to name
#'
```

在这里设置默认值的时候，不用写全function的内容

```
say_hello <- function(name = "aaron") {

  greeting <- paste("hello,", name)

  return(greeting)

}
```

用return的语法，可以指出你希望这个function输出什么。否则它只会输出最后一行code的outcome

```
say_hello()
```

```
## [1] "hello, aaron"
```

As you can see from the Roxygen Skeleton template above, function documentation should contain the following: * A description of what the function does * A description of each function argument, including the class of the argument (e.g. string, integer, dataframe) * A description of what the function returns, including the class of the object

Tips for writing functions:

- Function names should be short but effectively describe what the function does. They generally should be verbs while function arguments should be nouns. See the Tidyverse style guide for more details on function naming and style.
- You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code).
- As a general principle, functions should each do only one task. This makes it much easier to debug your code and reuse functions!

Iteration with `library(purrr)`

Most functions in `R` are vectorized. That means a function operates on every element in a vector by default.

```r
teacher_names <- c("aaron", "alex", "alena")

say_hello(name = teacher_names)
```

```
## [1] "hello, aaron" "hello, alex"  "hello, alena"
```

`say_hello()` is vectorized. It works without iteration, but let's just use it as an example to learn about iteration. Let's start with a simple for loop.

```r
# the vector over which to iterate
teacher_names <- c("aaron", "alex", "alena")

# preallocate the output vector
greetings <- vector(mode = "character", length = length(teacher_names))
for (i in seq_along(teacher_names)) {

  greetings[i] <- say_hello(teacher_names[i])

}

# print the result
greetings
```

```
## [1] "hello, aaron" "hello, alex"  "hello, alena"
```

That's a lot of typing. It is much more common in R to use `library(purrr)`, which replaces apply functions. `map()` iterates over each element of a data structure called `.x`, applies a function called `.f` to each element, and then returns a list.

```
library(purrr)

teacher_names <- c("aaron", "alex", "alena")
map(.x = teacher_names, .f = say_hello)
```

```
## [[1]]
## [1] "hello, aaron"
##
## [[2]]
## [1] "hello, alex"
##
## [[3]]
## [1] "hello, alena"
```

这个输出的结构是data frame

`map()` always returns a list. Sometimes we want to do something called map reduce. Functions like `map_chr()`, `map_dbl()`, and `map_df()` will map reduce. In this case, `map_chr()` returns a character vector instead of a list:

```
map_chr(.x = teacher_names, .f = say_hello)
```

```
## [1] "hello, aaron" "hello, alex"  "hello, alena"
```

We can quickly add anonymous functions in `map()` and its relatives with `~`. A tibble is just a list of columns. This function iterates over each column, counts the number of missing values, and then returns a named vector.

```
map_dbl(.x = msleep, .f = ~sum(is.na(.x)))
```

```
##         name        genus         vore        order conservation  sleep_total
##            0            0            7            0           29            0
##    sleep_rem  sleep_cycle        awake      brainwt       bodywt
##           22           51            0           27            0
```

The purrr library also has functions that can iterate over multiple inputs simultaneously. To figure out which purr function to use ask the following:

- How many inputs do I have? Is it one (`map`), two (`map2`), or 3+ (`pmap`)?
- What do I want back? A numeric vector (`dbl`), character vector (`chr`), dataframe created by column-binding (`dfc`), dataframe created by row-binding (`dfr`), etc?

The purrr cheatsheet is a great place to go for help using purrr and outlines the different function options. You can also see the purrr documentation for more details.

API Background

An *Application Programming Interface (API)* is a set of functions for programmatically accessing and/or processing data. Packages like `library(dplyr)` and `library(ggplot2)` have APIs that were carefully designed to be approachable. The dplyr website says, "dplyr is a part of the tidyverse, an ecosystem of packages designed with common APIs and a shared philosophy."

A *web API* is a way to interact with web-based software through code - usually to retrieve data or use a tool. We will only focus on web APIs in this tutorial. To use web APIs, actions between computers and web applications are typically communicated through URLs that are passed between the computers. Manipulating URLs is key to using web APIs.

Popular Web APIs

- Google Maps APIs
- Twitter API
- GitHub API
- World Bank
- Census API
- OpenFEC
- Urban Institute Education Data Portal
- Stanford CoreNLP API
- OpenTripPlanner API

Why APIs?

Why use APIs?

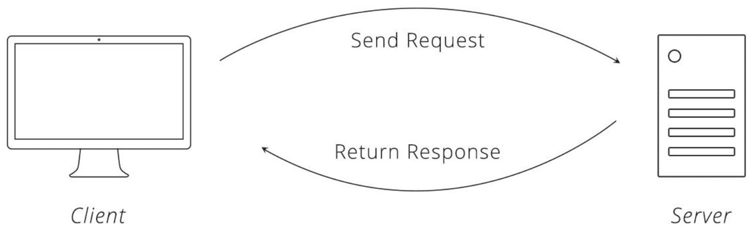
- **The data or method is only available through an API:** Census data and IPEDS are available through point-and-click interfaces and APIs, but many data and methods are only available through APIs. For example, data from the GoogleMaps API and methods from the Stanford CoreNLP software are only available in R through APIs.
- **APIs promote reproducibility:** Even with good documentation, pointing-and-clicking is tough to reproduce. APIs are useful for writing code to access data in an analysis before processing, modeling, and/or visualizing.
- **APIs enable scalability:** It's simple to point-and-click data for one state or one county. It's much tougher to point-and-click state or county level data if the data are stored in separate files. APIs can be accessed programtically, which means anything can be iterated to scale.

Why build APIs?

- **APIs enable scalability:** APIs enable scalability for data or methods creators. For example, the Urban Institute builds data visualization applications like Explore Your School's Changing Demographics on top of APIS.
- **APIs create standardization:** APIs standardize the communication between applications. By creating a protocol for a store of data or a model, it makes it simpler for downstream applications to access the data or methods of upstream applications.
- **Democratize access to data or methods:** Centralized storage and processing on the cloud are cheaper than personal computing on desktops and laptops. APIs allow users to access subsets of data or summary statistics quicker and cheaper than accessing all data and/or processing that data locally. For example, the new Urban Institute prototype summary API endpoints allow users to create summary statistics in API calls, which saves time and local storage.

Technical Background Part I

- **Server:** Remote computer that stores and processes information for an API. Note: This does not need to be a literal server. Many APIs are serverless and hosted in the cloud.
- **Client:** Computer requesting information from the API. In our cases, this is your computer.
- **Hyper-Text Transfer Protocol (HTTP):** Set of rules clients use to communicate with servers via the web (etiquette).
- **Request:** Contacting a web API with specifics about the information you wish to have returned. In general, this is built up as a URL.
- **Response:** A formatted set of information servers return to clients after requests.



Source: Zapier

- **Uniform Resource Locators (URL):** Text string that specifies a web location, a method for retrieving information from that web location, and additional parameters. We use these every day!

Process (without R code)

1. Client builds up a URL and sends a request to a server
2. Client receives a response from the server and checks for an error message
3. Client parses the response into a useful object: data frame

1. REQUEST (build up a URL)

Let's walk through an example from the Census Bureau.

1. The Census Bureau API contains hundreds of distinct data sets. Copy-and-paste https://api.census.gov/data.html into your web browser to see the list of datasets and documentation. Each one of these data sets is an API resource.
2. All Census Bureau API datasets are available from the same host name. Copy-and-paste https://api.census.gov/data/ into your browser (the previous link without .html). This contains information about all of the datasets in the previous link, only now the information is stored as JavaScript Object Notation (JSON) instead of HTML.
3. Each dataset is accessed through a *base URL* which consists of the host name, the year, and the "dataset name". Copy-and-paste https://api.census.gov/data/2014/pep/natstprc into your web browser. This returns Vintage 2014 Population Estimates: US, State, and PR Total Population and Components of Change. Each base URL corresponds to an API endpoint.
4. At this point, we're still only seeing metadata. We need to add a query string with a method and parameters. Copy-and-paste https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:* into your browser. Note the query string begins with `?`, includes the get method, and includes three parameters.

2. Check for a server error in the response

1. All of the URLs in part 1 were correctly specified and all of them returned results. What happens when incorrect URLs are passed to an API? Open a new browser window and copy-and-paste https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:57. Note: this call requests information for FIPs 57, which does not exist.

3. Parse the response

APIs need to return complicated hierarchical data as text. To do this, most APIs use **JavaScript Object Notation (JSON)**. JSON is a plain text hierarchical data structure. JSON is not JavaScript code. Instead, it's a non-rectangular method for storing data that can be accessed by most web applications and programming languages. Lists are made with `[]`. Objects are made with `{}` and contain key-value pairs. JSON is good at representing non-rectangular data and is standard on the web.

web API组织数据的几种形式：JSON，XML，HTML，不过JSON是最受欢迎的
我们可以使用R来把JSON转换成为合适的data frame。

```
{
  "Class name": "Intro to Data Science",
  "Class ID": "PPOL 670",
  "Instructors": ["Aaron R. Williams", "Alex C. Engler"],
  "Location": {
    "Building": "Healy Hall"
    "Room": "105"
  }
}
```

Web APIs can also return Extensible Markup Language (XML) and HyperText Markup Language (HTML), but JSON is definitely most popular. We'll use `library(jsonlite)` to parse hierarchical data and turn it into tidy data.

Technical Background Part II

- **API Resource** An object in an API. In our case, a resource will almost always be a specific data set.
- **API endpoint** The point where a client communicates with a web API.
- **Method/verb:** A verb that specifies the action the client wants to perform on the resource through an API endpoint: 语法：在地址栏（URL）输入什么
  - GET - Ask an API to send something to you
  - POST - send something to an API
  - HEAD, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH
- **Query string** Part of a URL that assigns values to parameters. Query strings begin with `?`. The general form is **key=value**. STATE=01
- **Parameters:** Arguments that are passed to APIs. Parameters are separated by `&`.
- **Headers** Meta-information about GET requests. We will only use the User-Agent HTTP Header.
- **Body** Additional information sent to the server. This isn't important for GET requests.
- **Server response code** (HTTP status code)
  - 100s: information responses
  - 200s: success
  - 300s: redirection
  - 400s: client error 404:网页不存在
  - 500s: server error

R example 1

Let's walk through the example above using R. First install `library(httr)` and `library(jsonlite)` with `install.packages(c("httr", "jsonlite"))`.

httr contains tools for working with HTTP and URLs. It contains functions for all HTTP methods including `GET()` and `POST()`.

1. REQUEST (build up a URL)

Using the final URL from the above example, lets query state names and state population in July 2014 for all states, Washington, D.C., and Puerto Rico. Note, it is good practice to create the link outside of `GET()` because we may need to manipulate the URL string in later examples. Note that `*` is a wildcard character which requests data for all possible values of the parameter.

```r
library(tidyverse)
library(httr)
library(jsonlite)

# make the URL
url <-
  "https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:*"

# use the URL to make a request from the API
pop_json <- GET(url = url)
```

2. Check for a server error in the response

```r
http_status(pop_json)
```

```
## $category
## [1] "Success"
##
## $reason
## [1] "OK"
##
## $message
## [1] "Success: (200) OK"
```

3. Parse the response

```r
# get the contents of the response as a text string
pop_json <- content(pop_json, as = "text")

# create a character matrix from the JSON
```

把matrix变成tibble格式的，这样就能把变量变成不同格式的，比如字符/数值

```
pop_matrix <- fromJSON(pop_json)

# turn the body of the character matrix into a tibble
pop_data <- as_tibble(pop_matrix[2:nrow(pop_matrix), ],
                      .name_repair = "minimal")

# add variable names to the tibble
names(pop_data) <- pop_matrix[1, ]

pop_data
```

```
## # A tibble: 52 x 4
##    STNAME               POP      DATE_ state
##    <chr>                <chr>    <chr> <chr>
##  1 Alabama              4849377  7     01
##  2 Alaska               736732   7     02
##  3 Arizona              6731484  7     04
##  4 Arkansas             2966369  7     05
##  5 California           38802500 7     06
##  6 Colorado             5355866  7     08
##  7 Connecticut          3596677  7     09
##  8 Delaware             935614   7     10
##  9 District of Columbia 658893   7     11
## 10 Florida              19893297 7     12
## # ... with 42 more rows
```

Parsing the response can be trickiest step. Here, the data of interest are rectangular and the JSON object is simple to parse. Sometimes, the returned object will be a complicated hierarchical structure, which will demand writing more R code.

Terms of Service of User Agents

Always read an API's terms of service to ensure that use of the API conforms to the API's rules.

Furthermore, it is a good idea to only run one API request at a time and to identify yourself as a user-agent in the header of the HTTP request. This is simple with `user_agent()` from `library(httr)`:

```
GET(url_link,
    user_agent("Georgetown Univ. Student Data Collector (student@georgetown.edu)."))
```

Exercise 1

1. Visit the Urban Institute Education Data Portal API.
2. Read the Terms of Use.
3. Let's pull data from IPEDS about full-time equivalent enrollment in graduate schools in 2016 in Washington, DC (FIP = 11). Create a new .R script called get-ed-data.R.

4. Read the instructions for direct API access. Copy-and-paste the generic form of the API call into you .R script.
5. Navigate to the documentation for Enrollment - Full time equivalent. Note that you may have to click "Full-time equivalent" under "Enrollment" in the menu on the left-hand side of the page to get to the relevant section. The example URL is "/api/v1/college-university/ipeds/enrollment-full-time-equivalent/{year}/{level_of_study}/" **Note:** The Urban Institute created an R package that simplifies this entire process. Let's build our request from scratch just to prove that we can.
6. Combine the endpoint URL with the example URL from earlier. This means combining the base URL with the URL and parameters for the endpoint.
7. Fill in the necessary parameters and filters.
8. Make a GET request with `GET()`. Be sure to include the User-Agent.
9. Check the HTTP status.
10. Parse the response. li

还没来得及做，回去check一下是怎么回事（之前需要去掉一个东西，这里的code好像不太一样）

. . .

Authentication

Many APIs require authentication. For example, the Census API requires a user-specific API key for all API calls. (sign up now) This API key is simply passed as part of the path in the HTTP request. "https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:*&key=your key here" There are other authentication methods, but this is most common.

It is a bad idea to share API credentials. NEVER post a credential on GitHub. A convenient solution is to use a credentials file with the `library(dotenv)` package as follows:

First, install the package using `install.packages("dotenv")` in the console and create a file called `.env` in the directory where your Rproject is located. You may get a message that files starting with a "." are reserved for the system, you should hit "ok" to proceed. You can store as many credentials as you want in this file, with each key-value pair on a new line. Note you need to hit enter after the last key-value pair so the file ends with a blank new line.

`census_api_key=<key value>`

Then, you can access and use credentials as follows:

```r
library(dotenv)
credential <- Sys.getenv("census_api_key")

url <- str_glue(
  "https://api.census.gov/data/2014/pep/natstprc?get=STNAME,POP&DATE_=7&for=state:*&key={credential}"
)
```

Note that you may have to restart your R session to load the file. Be sure to add this `.env` credentials file to your `.gitignore`!

Pagination 分页

A single API call could potentially return an unwieldy amount of information. This would be bad for the server, because the organization would need to pay for lots of computing power. This would also be bad for the client because the client could quickly become overwhelmed by data. To solve this issue, many APIs are paginated. Pagination is simply breaking API responses into subsets.

For example, the original example returned information for all states in the United States. When information is requested at the Census tract level, instead of returning information for the entire United States, information can only be requested one state at a time. Getting information for the entire United States will require iterating through each state.

Rate Limiting

Rate limiting is capping the number of requests by a client to an API in a given period of time. This is most relevant when results are paginated and iterating requests is necessary. It is also relevant when developing code to query an API–because a developer can burden the API with ultimately useless requests.

It is sometimes useful to add `Sys.sleep()` to R code, to pause the R code to give the API a break from requests between each request. Even 0.5 seconds can be the difference.

R example 2 (Advanced)

This example pulls information at the Census tract level. Because of pagination, the example requires a custom function and iterates that function using `map_df()` from `library(purrr)`. It includes `Sys.sleep()` to pause the requests between each query.

The example pulls the estimated number of males (B01001_002E) and females (B01001_026E) in the 2018 5-year ACS for each Census tract in Alabama and Alaska.

Here are a few select columns for the 2018 5-year ACS from the Census API documentation page:

Vintage	Dataset Name	Dataset Type	Geography List	Variable List	Group List	Examples
2018	acs>acs5>profile	Aggregate	geographies	variables	groups	examples

1. Write a custom function

This function 1. builds a URL and requests from the API, 2. checks for a server error, and 3. parses the response

```r
get_acs <- function(fips, credential) {

  # build a URL
  # paste0() is only used because the URL was too wide for the PDF
  url <- str_glue(
    paste0(
      "https://api.census.gov/data/2018/acs/acs5",
      "?get=B01001_002E,B01001_026E&for=tract:*&in=state:{fips}&key={credential}"
    )
  )

  # use the URL to make a request from the API
  acs_json <- GET(url = url)

  # get the contents of the response as a text string
  acs_json <- content(acs_json, as = "text")

  # create a character matrix from the JSON
  acs_matrix <- fromJSON(acs_json)

  # turn the body of the character matrix into a tibble
  acs_data <- as_tibble(acs_matrix[2:nrow(acs_matrix), ],
                        .name_repair = "minimal")

  # add variable names to the tibble
  names(acs_data) <- acs_matrix[1, ]

  # pause to be polite
  Sys.sleep(0.5)

  return(acs_data)

}
```

2. Create a vector of FIPs

This could be all states, districts, and territories. It's only Alabama and Alaska for brevity.

```r
fips <- c("01", "02")
```

3. Iterate the functions

`map_df()` iterates `get_acs()` along the vector of state FIPs and returns a `tibble`.

```
# load credentials
credential <- Sys.getenv("census_api_key")

# iterate the function over Alabama and Alaska
map_df(fips, .f = get_acs, credential = credential)
```

```
## # A tibble: 1,348 x 5
##    B01001_002E B01001_026E state county tract
##    <chr>       <chr>       <chr> <chr>  <chr>
##  1 2409        2800        01    097    006501
##  2 5961        6372        01    097    006502
##  3 3098        2888        01    097    006701
##  4 1837        1993        01    097    006702
##  5 1410        1503        01    097    007201
##  6 1538        1136        01    097    007202
##  7 1586        1311        01    097    007400
##  8 680         705         01    097    007500
##  9 981         997         01    097    007600
## 10 537         691         01    097    007700
## # ... with 1,338 more rows
```

R Packages

There are R packages that simplify interacting with many popular APIs. `library(tidycensus)` (tutorial here) and `library(censusapi)` (tutorial here) simplifies navigating Census documentation, checking the status code, building URLs for the Census API, and parsing JSON responses. This can save a lot of time and effort! The following code is one iteration of the advanced example from above!

```
library(censusapi)

getCensus(name = "acs/acs5",
          vars = c("B01001_002E", "B01001_026E"),
          region = "tract:*",
          regionin = "state:01",
          vintage = 2018,
          key = credential) %>%
  as_tibble()
```

```
## # A tibble: 1,181 x 5
##    state county tract  B01001_002E B01001_026E
##    <chr> <chr>  <chr>        <dbl>       <dbl>
##  1 01    097    006501        2409        2800
##  2 01    097    006502        5961        6372
##  3 01    097    006701         3098        2888
##  4 01    097    006702         1837        1993
##  5 01    097    007201         1410        1503
##  6 01    097    007202         1538        1136
##  7 01    097    007400         1586        1311
##  8 01    097    007500          680         705
##  9 01    097    007600          981         997
## 10 01    097    007700          537         691
## # ... with 1,171 more rows
```

Popular R API packages

- censusapi
- tidycensus
- Urban Institute Education Data Portal

Resources

- Zapier Intro to APIs
- Best practices for API packages
- How we built the API for the Education Data Portal
- Democratizing Big Data Processing for the Education Data Portal
- Building R and Stata packages for the Education Data Portal