Creating Functions and Packages

In this chapter, you’ll learn how to define your own R functions, including the parameters they should accept. Then, you’ll create a package to distribute those functions, add your code and dependencies to it, write its documentation, and choose the license under which to release it.

Saving your code as custom functions and then distributing them in packages can have numerous benefits. First, packages make your code easier for others to use. For example, when researchers at the Moffitt Cancer Center needed to access code from a database, data scientists Travis Gerke and Garrick Aden-Buie used to write R code for each researcher, but they quickly realized they were reusing the same code over and over. Instead, they made a package with functions for accessing databases. Now researchers no longer had to ask for help; they could simply install the package Gerke and Aden-Buie had made and use its functions themselves.

What’s more, developing packages allows you to shape how others work. Say you make a ggplot theme that follows the principles of high-quality data visualization discussed in Chapter 3. If you put this theme in a package, you can give others an easy way to follow these design principles. In short, functions and packages help you work with others using shared code.

Creating Your Own Functions

Hadley Wickham, developer of the tidyverse set of packages, recommends creating a function once you’ve copied some code three times. Functions have three pieces: a name, a body, and arguments.

Writing a Simple Function

You’ll begin by writing an example of a relatively simple function. This function, called show\_in\_excel\_penguins(), opens the penguin data from Chapter 7 in Microsoft Excel:

1 penguins <- read\_csv("https://data.rwithoutstatistics.com/penguins-2007.csv")

2 show\_in\_excel\_penguins <- function() {

csv\_file <- str\_glue("{tempfile()}.csv")

write\_csv(

x = penguins,

file = csv\_file,

na = ""

)

file\_show(path = csv\_file)

}

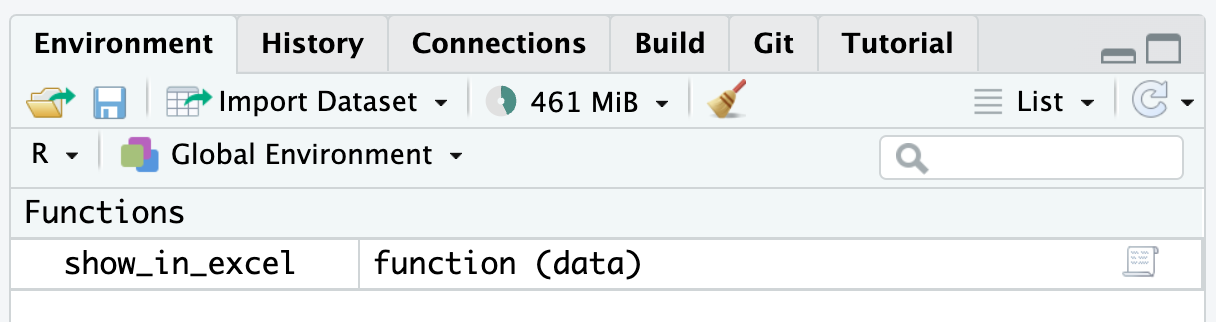
This code first loads the tidyverse and fs packages. You’ll use tidyverse to create a filename for the CSV file and save it, and fs to open the CSV file in Excel (or whichever program your computer uses to open CSV files by default).

Next, the read\_csv() function imports the penguin data and names the data frame penguins 1. Then it creates the new show\_in\_excel\_penguins function, using the assignment operator (<-) and function() to specify that show\_in\_excel\_penguins isn’t a variable name but a function name 2. The open curly bracket ({) at the end of the line indicates the start of the function body, where the “meat” of the function can be found. In this case, the body does three things:

* Creates a location for a CSV file to be saved using the str\_glue() function combined with the tempfile() function. This creates a file at a temporary location with the .csvextension and saves it as csv\_file.
* Writes penguins to the location set in csv\_file. The x argument in write\_csv() refers to the data frame to be saved. It also specifies that all NA values should show up as blanks. (By default, they would display the text NA.)
* Uses the file\_show() function from the fs package to open the temporary CSV file in Excel.

To use the show\_in\_excel\_penguins() function, highlight the lines that define the function and then press command-enter on macOS or ctrl-enter on Windows. You should now see the function in your global environment, as shown in Figure 12-1.

[F12001.png]



* + - * 1. The new function in the global environment

From now on, any time you run the code show\_in\_excel\_penguins(), R will open the penguins data frame in Excel.

Adding Arguments

You’re probably thinking that this function doesn’t seem very useful. All it does it open the penguins data frame. Why would you want to keep doing that? A more practical function would let you open any data in Excel so you can use it in a variety of contexts.

The show\_in\_excel() function does just that: it takes any data frame from R, saves it as a CSV file, and opens the CSV file in Excel. Bruno Rodrigues, head of the Department of Statistics and Data Strategy at the Ministry of Higher Education and Research in Luxembourg, wrote show\_in\_excel() to easily share data with his non-R-user colleagues. Whenever he needed data in a CSV file, he could run this function.

Replace your show\_in\_excel\_penguins() function definition with this slightly simplified version of the code that Rodrigues used:

show\_in\_excel <- function(data) {

csv\_file <- str\_glue("{tempfile()}.csv")

write\_csv(

x = data,

file = csv\_file,

na = ""

)

file\_show(path = csv\_file)

}

This code. Notice that the first line now says function(data). Items listed within the parentheses of the function definition are arguments. If you look further down, you’ll see the second change. Within write\_csv(), instead of x = penguins, it now says x = data. This allows you to use the function with any data, not just penguins.

To use this function, you simply tell show\_in\_excel() what data to use, and the function opens the data in Excel. For example, tell it to open the penguins data frame as follows:

show\_in\_excel(data = penguins)

Having created the function with the data argument, now you can run it with any data you want to. This code, for example, imports the COVID case data from Chapter 10 and opens it in Excel:

covid\_data <- read\_csv("https://data.rwithoutstatistics.com/

us-states-covid-rolling-average.csv")

show\_in\_excel(data = covid\_data)

You can also use show\_in\_excel() at the end of a pipeline. This code filters the covid\_data data frame to include only data from California before opening it in Excel:

covid\_data %>%

filter(state == "California") %>%

show\_in\_excel()

Bruno Rodrigues could have copied the code within the show\_in\_excel() function and rerun it every time he wanted to view his data in Excel. But, by creating a function, he was able to write the code just once and then run it as many times as necessary.

Creating a Function to Format Race and Ethnicity Data

Hopefully now you better understand how functions work, so let’s walk through an example function you could use to simplify some of the activities from previous chapters.

In Chapter 11, when you used the tidycensus package to automatically import data from the United States Census Bureau, you learned that the census data has many variables with non-intuitive names. Say you regularly want to access data about race and ethnicity from the American Community Survey, but you can never remember which variables enable you to do so. To make your task more efficient, you’ll create a get\_acs\_race\_ethnicity() function step-by-step in this section, learning some important concepts about custom functions along the way.

A first version of the get\_acs\_race\_ethnicity() function might look like this:

library(tidycensus)

get\_acs\_race\_ethnicity <- function() {

race\_ethnicity\_data <-

get\_acs(

geography = "state",

variables = c(

"White" = "B03002\_003",

"Black/African American" = "B03002\_004",

"American Indian/Alaska Native" = "B03002\_005",

"Asian" = "B03002\_006",

"Native Hawaiian/Pacific Islander" = "B03002\_007",

"Other race" = "B03002\_008",

"Multi-Race" = "B03002\_009",

"Hispanic/Latino" = "B03002\_012"

)

)

race\_ethnicity\_data

}

Within the function body, this code calls the get\_acs() function from tidycensus to retrieve population data at the state level. But instead of returning the function’s default output, it updates the hard-to-remember variable names to human-readable names, such as White and Black/African American, and saves them as an object called race\_ethnicity\_data. The code then uses the race\_ethnicity\_data object to return that data when the get\_acs\_race\_ethnicity() function is run.

To run this function, enter the following:

get\_acs\_race\_ethnicity()

Doing so should return data with easy-to-read race and ethnicity group names:

#> # A tibble: 416 × 5

#> GEOID NAME variable estimate moe

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

#> 1 01 Alabama White 3241003 2076

#> 2 01 Alabama Black/African American 1316314 3018

#> 3 01 Alabama American Indian/Alaska Na... 17417 941

#> 4 01 Alabama Asian 69331 1559

#> 5 01 Alabama Native Hawaiian/Pacific I... 1594 376

#> 6 01 Alabama Other race 12504 1867

#> 7 01 Alabama Multi-Race 114853 3835

#> 8 01 Alabama Hispanic/Latino 224659 413

#> 9 02 Alaska White 434515 1067

#> 10 02 Alaska Black/African American 22787 769

#> # 406 more rows

You could improve this function in a few ways. You might want the resulting variable names to follow a consistent syntax, for example, so you could use the clean\_names() function from the janitor package to format them in snake case (in which all words are lowercase and separated by underscores). However, you might also want to have the option of keeping the original variable names. To accomplish this, add the clean\_variable\_names argument to the function definition as follows:

get\_acs\_race\_ethnicity <- function(clean\_variable\_names = FALSE) {

race\_ethnicity\_data <-

get\_acs(

geography = "state",

variables = c(

"White" = "B03002\_003",

"Black/African American" = "B03002\_004",

"American Indian/Alaska Native" = "B03002\_005",

"Asian" = "B03002\_006",

"Native Hawaiian/Pacific Islander" = "B03002\_007",

"Other race" = "B03002\_008",

"Multi-Race" = "B03002\_009",

"Hispanic/Latino" = "B03002\_012"

)

)

if (clean\_variable\_names == TRUE) {

race\_ethnicity\_data <- clean\_names(race\_ethnicity\_data)

}

race\_ethnicity\_data

}

This code adds the clean\_variable\_names argument to get\_acs\_race\_ethnicity() and specifies that its value should be FALSE by default. Then, in the function body, an if statement says that if the argument is TRUE, the variable names should be overwritten by versions formatted in snake case 1. If the argument is FALSE, the variable names remain unchanged.

If you run the function now, nothing should change, because the new argument is set to FALSE by default. Try setting clean\_variable\_names to TRUE as follows:

get\_acs\_race\_ethnicity(clean\_variable\_names = TRUE)

This function call should return data with consistent variable names:

#> # A tibble: 416 × 5

#> geoid name variable estimate moe

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

#> 1 01 Alabama White 3241003 2076

#> 2 01 Alabama Black/African American 1316314 3018

#> 3 01 Alabama American Indian/Alaska Na... 17417 941

#> 4 01 Alabama Asian 69331 1559

#> 5 01 Alabama Native Hawaiian/Pacific I... 1594 376

#> 6 01 Alabama Other race 12504 1867

#> 7 01 Alabama Multi-Race 114853 3835

#> 8 01 Alabama Hispanic/Latino 224659 413

#> 9 02 Alaska White 434515 1067

#> 10 02 Alaska Black/African American 22787 769

#> # 406 more rows

Notice that GEOID and NAME now appear as geoid and name.

Now that you’ve seen how to add arguments to two separate functions, you’ll learn how to pass arguments from one function to another.

Using . . . to Pass Arguments to Another Function

The get\_acs\_race\_ethnicity() function you’ve created retrieves population data at the state level by passing the geography = "state" argument to the get\_acs() function. But what if you wanted to obtain county-level or census tract data? You could do so using get\_acs(), but get\_acs\_race\_ethnicity() isn’t currently written in a way that would allow this. How could you modify the function to make it more flexible?

Your first idea might be to add a new argument for the level of data to retrieve. You could edit the first two lines of the function as follows to add a my\_geography argument and then use it in the get\_acs() function like so:

get\_acs\_race\_ethnicity <- function(clean\_variable\_names = FALSE, my\_geography) {

race\_ethnicity\_data <- get\_acs(geography = my\_geography,

--snip--

But what if you also want to select the year for which to retrieve data? Well, you could add an argument for that as well. However, as you saw in Chapter 11, the get\_acs() function has many arguments, and repeating them all in your code would quickly become cumbersome.

The ... syntax gives you a more efficient option. Placing ... in the get\_acs\_race\_ethnicity() function allows you to automatically pass any of its arguments to get\_acs() by including ... in that function as well:

get\_acs\_race\_ethnicity <- function(

clean\_variable\_names = FALSE,

...

) {

race\_ethnicity\_data <-

get\_acs(

...,

variables = c(

"White" = "B03002\_003",

"Black/African American" = "B03002\_004",

"American Indian/Alaska Native" = "B03002\_005",

"Asian" = "B03002\_006",

"Native Hawaiian/Pacific Islander" = "B03002\_007",

"Other race" = "B03002\_008",

"Multi-Race" = "B03002\_009",

"Hispanic/Latino" = "B03002\_012"

)

)

if (clean\_variable\_names == TRUE) {

race\_ethnicity\_data <- clean\_names(race\_ethnicity\_data)

}

race\_ethnicity\_data

}

Try running your function by passing it the geography argument set to "state":

get\_acs\_race\_ethnicity(geography = "state")

This should return the following:

#> # A tibble: 416 × 5

#> GEOID NAME variable estimate moe

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

#> 1 01 Alabama White 3241003 2076

#> 2 01 Alabama Black/African American 1316314 3018

#> 3 01 Alabama American Indian/Alaska Na... 17417 941

#> 4 01 Alabama Asian 69331 1559

#> 5 01 Alabama Native Hawaiian/Pacific I... 1594 376

#> 6 01 Alabama Other race 12504 1867

#> 7 01 Alabama Multi-Race 114853 3835

#> 8 01 Alabama Hispanic/Latino 224659 413

#> 9 02 Alaska White 434515 1067

#> 10 02 Alaska Black/African American 22787 769

#> # 406 more rows

You’ll see that the GEOID and NAME variables are uppercase because the clean\_variable\_names argument is set to FALSE by default, and we didn’t change it when using the get\_acs\_race\_ethnicity() function.

Alternatively, you could change the value of the argument to get data by county:

get\_acs\_race\_ethnicity(geography = "county")

You could also run the function with the geometry = TRUE argument to return geospatial data alongside demographic data:

get\_acs\_race\_ethnicity(geography = "county", geometry = TRUE)

The function should return data like the following:

#> Simple feature collection with 416 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

#> 1 56 Wyoming White 478508

#> 2 56 Wyoming Black/African American 4811

#> 3 56 Wyoming American Indian/Alaska Na... 11330

#> 4 56 Wyoming Asian 4907

#> 5 56 Wyoming Native Hawaiian/Pacific I... 397

#> 6 56 Wyoming Other race 1582

#> 7 56 Wyoming Multi-Race 15921

#> 8 56 Wyoming Hispanic/Latino 59185

#> 9 02 Alaska White 434515

#> 10 02 Alaska Black/African American 22787

#> moe geometry

#> 1 959 MULTIPOLYGON (((-111.0546 4...

#> 2 544 MULTIPOLYGON (((-111.0546 4...

#> 3 458 MULTIPOLYGON (((-111.0546 4...

#> 4 409 MULTIPOLYGON (((-111.0546 4...

#> 5 158 MULTIPOLYGON (((-111.0546 4...

#> 6 545 MULTIPOLYGON (((-111.0546 4...

#> 7 1098 MULTIPOLYGON (((-111.0546 4...

#> 8 167 MULTIPOLYGON (((-111.0546 4...

#> 9 1067 MULTIPOLYGON (((179.4825 51...

#> 10 769 MULTIPOLYGON (((179.4825 51...

The ... syntax allows you to create your own function and pass arguments from it to another function without repeating all of that function’s arguments in your own code. This approach gives you flexibility while keeping your code concise.

Now let’s look at how to put your custom functions into a package.

Creating a Package

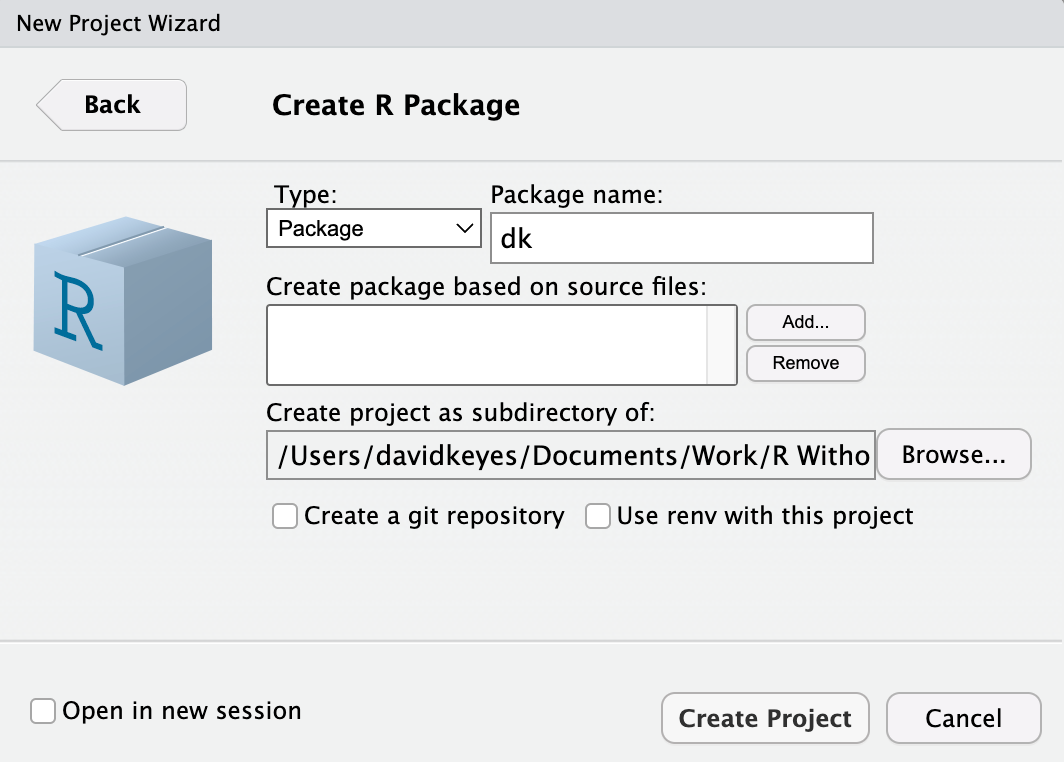
Packages bundle your functions so you can use them in multiple projects. If you find yourself copying functions from one project to another, or from a functions.R file into each new project, that’s a good indication that you should make a package.

While you can run the functions from a functions.R file in your own environment, this code might not work on someone else’s computer. Other users may not have the necessary packages installed, or they may be confused about how your functions’ arguments work and not know where to go for help. Putting your functions in a package makes them more likely to work for everyone, as they include the necessary dependencies as well as built-in documentation to help others use the functions on their own.

Starting the Package

To create a package in RStudio, go to File4New Project4New Directory. Select R Package from the list of options and give your package a name. In Figure 12-2, I’ve called mine dk. Also decide where you want your package to live on your computer. You can leave everything else as is.

[F12002.png]



* + - * 1. The RStudio menu for creating your own package

RStudio will now create and open the package. It should already contain a few files, including hello.R, which has a prebuilt function called hello() that, when run, prints the text Hello, world! in the console. You’ll get rid of this and a few other default files so you can start with a clean slate. Delete hello.R, NAMESPACE, and hello.Rd in the man directory.

Adding Functions with use\_r()

All of the functions in a package should go in separate files in the R folder. To add these files to the package automatically and test that they work correctly, you’ll use the usethis and devtools packages. Install them using install.packages() like so:

install.packages("usethis")

install.packages("devtools")

To add a function to the package, run the use\_r() function from the usethis package in the console:

usethis::use\_r("acs")

The package::function() syntax allows you to use a function without loading the associated package. The use\_r() function should create a file in the R directory with the argument name you provide—in this case, the file is called acs.R. The name itself doesn’t really matter, but it’s a good practice to choose something that gives an indication of the functions the file contains. Now you can open the file and add code to it. Copy the get\_acs\_race\_ethnicity() function to the package.

Checking the Package with devtools

You need to change the get\_acs\_race\_ethnicity() function in a few ways to make it work in a package. The easiest way to figure out what changes you need to make is to use built-in tools to check that your package is built correctly. Run the function devtools::check() in the console to perform what is known as an R CMD check, which makes sure that others can install your package on their system. Running R CMD check on the dkpackage outputs this long message:

── R CMD check results ─────────────── dk ────

Duration: 4s

❯ checking DESCRIPTION meta-information ... WARNING

Non-standard license specification:

What license is it under?

Standardizable: FALSE

❯ checking for missing documentation entries ... WARNING

Undocumented code objects:

'get\_acs\_race\_ethnicity'

All user-level objects in a package should have documentation entries.

See chapter 'Writing R documentation files' in the 'Writing R

Extensions' manual.

1 ❯ checking R code for possible problems ... NOTE

get\_acs\_race\_ethnicity: no visible global function definition for

'get\_acs'

get\_acs\_race\_ethnicity: no visible global function definition for

'clean\_names'

Undefined global functions or variables:

clean\_names get\_acs

0 errors ✔ | 2 warnings ✖ | 1 note ✖

, so let’s review the output from bottom to top. The line 0 errors ✔ | 2 warnings ✖ | 1 note ✖ highlights three levels of issues identified in the package. Errors are the most severe, as they mean others won’t be able to install your package, while warnings and notes may cause problems for others. It’s best practice to eliminate all errors, warnings, and notes.

We’ll start by addressing the note at 1. To help you understand what R CMD check is saying here, I need to explain a bit about how packages work. When you install a package using the install.packages() function, it often takes a while. That’s because the package you’re telling R to install likely uses functions from other packages. To access these functions, R must install these packages (known as dependencies) for you; after all, it would be a pain if you had to manually install a whole set of dependencies every time you installed a new package. But to make sure that the appropriate packages are installed for any user of the dk package, you still have to make a few changes.

R CMD check is saying this package includes several “undefined global functions or variables” and “no visible global function definition” for various functions. This is because you’re trying to use functions from the tidycensus and janitor packages, but you haven’t specified where these functions come from. I can run this code in my environment because I have tidycensus and janitor installed, but you can’t assume the same of everyone.

Adding Dependency Packages

To ensure the package’s code will work, you need to install tidycensus and janitor for users when they install the dk package. To do this, run the use\_package() function from the usethis package in the console, first specifying "tidycensus" for the package argument:

usethis::use\_package(package = "tidycensus")

You should get the following message:

✔ Setting active project to '/Users/davidkeyes/Documents/Work/R Without Statistics/dk'

✔ Adding 'tidycensus' to Imports field in DESCRIPTION

• Refer to functions with `tidycensus::fun()`

The Setting active project... line indicates that you’re working in the dk project. The second line indicates that the DESCRIPTION file has been edited. This file provides metadata about the package you’re developing.

Next, add the janitor package the same way you added tidyverse:

usethis::use\_package(package = "janitor")

which should give you the following output:

✔ Adding 'janitor' to Imports field in DESCRIPTION

• Refer to functions with `janitor::fun()`

If you open the DESCRIPTION file in the root directory of your project, you should see the following:

Package: dk

Type: Package

Title: What the Package Does (Title Case)

Version: 0.1.0

Author: Who wrote it

Maintainer: The package maintainer <yourself@somewhere.net>

Description: More about what it does (maybe more than one line)

Use four spaces when indenting paragraphs within the Description.

License: What license is it under?

Encoding: UTF-8

LazyData: true

Imports:

janitor,

tidycensus

The Imports section at the bottom of the file indicates that when a user installs the dk package, the tidycensus and janitor packages will also be imported.

Referring to Functions Correctly

The output from running usethis::use\_package(package = "janitor") also included the line Refer to functions with tidycensus::fun() (where fun() stands for function name). This tells you that in order to use functions from other packages in the dk package, you need to specify both the package name and the function name to ensure that the correct function is used at all times. On rare occasions, you’ll find functions with identical names used across multiple packages, and this syntax avoids ambiguity. This line from the R CMD check

Undefined global functions or variables:

clean\_names get\_acs

appeared because you were using functions without saying what package they came from. The clean\_names() function comes from the janitor package, and get\_acs() comes from tidycensus, so you need to add these package names before each function:

get\_acs\_race\_ethnicity <- function(

clean\_variable\_names = FALSE,

...

) {

race\_ethnicity\_data <- tidycensus::get\_acs(

...,

variables = c(

"White" = "B03002\_003",

"Black/African American" = "B03002\_004",

"American Indian/Alaska Native" = "B03002\_005",

"Asian" = "B03002\_006",

"Native Hawaiian/Pacific Islander" = "B03002\_007",

"Other race" = "B03002\_008",

"Multi-Race" = "B03002\_009",

"Hispanic/Latino" = "B03002\_012"

)

)

if (clean\_variable\_names == TRUE) {

race\_ethnicity\_data <- janitor::clean\_names(race\_ethnicity\_data)

}

race\_ethnicity\_data

}

Now you can run devtools::check() again, and you should see that the notes have gone away:

❯ checking DESCRIPTION meta-information ... WARNING

Non-standard license specification:

What license is it under?

Standardizable: FALSE

❯ checking for missing documentation entries ... WARNING

Undocumented code objects:

'get\_acs\_race\_ethnicity'

All user-level objects in a package should have documentation entries.

See chapter 'Writing R documentation files' in the 'Writing R

Extensions' manual.

0 errors ✔ | 2 warnings ✖ | 0 notes ✔

However, there are still two warnings to deal with. You’ll do that next.

Creating Documentation with Roxygen

The checking for missing documentation entries warning indicates that you need to document your get\_acs\_race\_ethnicity() function. One of the benefits of creating a package is that you can add documentation to help others use your code. In the same way that users can enter ?get\_acs() and see documentation about that function, you want them to be able to enter ?get\_acs\_race\_ethnicity() to learn how your function works.

To create documentation for get\_acs\_race\_ethnicity(), you’ll use Roxygen, a documentation tool that uses a package called roxygen2. To get started, place your cursor anywhere in your function. Then, in RStudio go to Code4Insert Roxygen Skeleton. This should add the following text above the get\_acs\_race\_ethnicity() function:

#' Title

#'

#' @param clean\_variable\_names

#' @param ...

#'

#' @return

#' @export

#'

#' @examples

This text is the documentation’s skeleton. Each line starts with the special characters #', which indicate that you’re working with Roxygen. Now you can edit the text to create your documentation. Begin by replacing Title with a sentence that describes the function:

#' Access race and ethnicity data from the American Community Survey

Next, turn your attention to the lines beginning with @param. Roxygen automatically creates one of these lines for each function argument, but it’s up to you to fill them in with a description. Begin by describing what the clean\_variable\_names argument does. Next, specify that the ... will pass additional arguments to the tidycensus::get\_acs() function:

#' @param clean\_variable\_names Should variable names be cleaned (i.e. snake case)

#' @param ... Other arguments passed to tidycensus::get\_acs()

The @return line should tell the user what the get\_acs\_race\_ethnicity() function returns. In this case, it returns data, which you document as follows:

#' @return A tibble with five variables: GEOID, NAME, variable, estimate, and moe

Below @return is @export. You don’t need to change anything here. Most functions in a package are known as exported functions, meaning they’re available to users of the package. In contrast, internal functions, which are used only by the package developers, don’t have @export in the Roxygen skeleton.

The last section is @examples. This is where you can give examples of code that users can run to learn how the function works. Doing this introduces some complexity and isn’t required, so you can skip it here and delete the line with @examples on it.

Note If you want to learn more about adding examples to your documentation, the second edition of Hadley Wickham and Jenny Bryan’s book R Packages (O’Reilly, 2023) is a great resource.

Now that you’ve added documentation with Roxygen, run devtools::document() in the console. This should create a get\_acs\_race\_ethnicity.Rd documentation file in the man directory using the very specific format that R packages require. You’re welcome to look at it, but you can’t change it; it’s read-only.

Running the function should also create a NAMESPACE file, which lists the functions that your package makes available to users. It should look like this:

# Generated by roxygen2: do not edit by hand

export(get\_acs\_race\_ethnicity)

Your get\_acs\_race\_ethnicity() function is now almost ready for users.

Adding a License and Metadata

Run devtools::check() again to see if you’ve fixed the issues that led to the warnings. The warning about missing documentation should no longer be there. However, you do still get one warning:

❯ checking DESCRIPTION meta-information ... WARNING

Non-standard license specification:

What license is it under?

Standardizable: FALSE

0 errors ✔ | 1 warning ✖ | 0 notes ✔

This warning reminds you that you haven’t given your package a license. If you plan to make your package publicly available, choosing a license is important because it tells other people what they can and cannot do with your code. For information about how to choose the right license for your package, see https://choosealicense.com.

In this example, you’ll use the MIT license, which allows users to do essentially whatever they want with your code, by running usethis::use\_mit\_license(). The usethis package has similar functions for other common licenses. You should get the following output:

✔ Setting active project to '/Users/davidkeyes/Documents/Work/R Without Statistics/dk'

✔ Setting License field in DESCRIPTION to 'MIT + file LICENSE'

✔ Writing 'LICENSE'

✔ Writing 'LICENSE.md'

✔ Adding '^LICENSE\\.md$' to '.Rbuildignore'

The use\_mit\_license() function handles a lot of the tedious parts of adding a license to your package. Most importantly for our purposes, it specifies the license in the DESCRIPTION file. If you open it, you should see this confirmation that you’ve added the MIT license:

License: MIT + file LICENSE

In addition to the license, the DESCRIPTION file contains metadata about the package. You can make a few changes to identify its title and add an author, a maintainer, and a description. The final DESCRIPTION file might look something like this:

Package: dk

Type: Package

Title: David Keyes's Personal Package

Version: 0.1.0

Author: David Keyes

Maintainer: David Keyes <david@rfortherestofus.com>

Description: A package with functions that David Keyes may find

useful.

License: MIT + file LICENSE

Encoding: UTF-8

LazyData: true

Imports:

janitor,

tidycensus

Having made these changes, run devtools::check() one more time to make sure everything is in order:

0 errors ✔ | 0 warnings ✔ | 0 notes ✔

This is exactly what you want to see!

Writing Additional Functions

You’ve now got a package with one working function in it. If you wanted to add more functions, you would follow the same procedure:

1. Create a new .R file with usethis::use\_r() or copy another function to the existing .R file.
2. Develop your function using the package::function() syntax to refer to functions from other packages.
3. Add any dependency packages with use\_package().
4. Add documentation for your function.
5. Run devtools::check() to make sure you did everything correctly.

Your package can contain a single function, like dk, or as many functions as you want.

Installing the Package

Now you’re ready to install and use the new package. When you’re developing your own package, installing it for your own use is relatively straightforward. Simply run devtools::install(), and the package will be ready for you to use in any project.

Of course, if you’re developing a package, you’re likely doing it not just for yourself, but for others as well. The most common way to make your package available to others is with the code-sharing website GitHub. The details of how to put your code on GitHub are beyond what I can cover here, but the book Happy Git and GitHub for the useR by Jenny Bryan (self-published at https://happygitwithr.com) is a great place to start.

I’ve pushed the dk package to GitHub, and you can find it at https://github.com/dgkeyes/dk. If you’d like to install it, first make sure you have the remotes package installed, then run the code remotes::install\_github("dgkeyes/dk") in the console.

Summary

In this chapter, you’ve seen that packages are useful because they let you bundle several elements needed to reliably run your code: a set of functions, instructions to automatically install dependency packages, and code documentation.

Creating your own R package is especially beneficial when you’re working for an organization, as packages can allow advanced R users to help colleagues with less experience. When Travis Gerke and Garrick Aden-Buie provided researchers at the Moffitt Cancer Center with a package that contained functions for easily accessing their databases, the researchers began to use R more creatively.

If you create a package, you can also guide people to use R in the way you think is best. Packages are a way to ensure that others follow best practices (without even being aware they are doing so). They make it easy to reuse functions across projects, help others, and adhere to a consistent style.

Further Resources

Malcolm Barrett, “Package Development with R,” online course, accessed December 2, 2023, <https://rfortherestofus.com/courses/package-development/>.

Hadley Wickham and Jennifer Bryan, R Packages, 2nd ed. (Sebastopol, CA: O’Reilly Media, 2023), https://r-pkgs.org.

Wrapping Up

R was invented in 1993 as a tool for statistics, and in the years since, it has been used for plenty of statistical analysis. But over the last three decades, R has also become a tool that can do much more than statistics.

As you’ve seen in this book, R is great for making visualizations. You can use it to create high-quality graphics and maps, make your own theme to keep your visuals consistent and on-brand, and generate tables that look good and communicate well. Using R Markdown or Quarto, you can create reports, presentations, and websites. And best of all, these documents are all reproducible, meaning that updating them is as easy as rerunning your code. Finally, you’ve seen that R can help you automate how you access data, as well as assist you in collaborating with others through the functions and packages you create.

If R was new to you when you started this book, I hope you now feel inspired to use it. If you’re an experienced R user, I hope this book has shown you some ways to use R that you hadn’t previously considered. No matter your background, my hope is that you now understand how to use R like a pro—and without any statistics.