Creating Functions and Packages

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

Saving your code as functions and then distributing these functions 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 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 principles. Functions and packages can 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

Let’s begin by taking an example of a relatively simple function. This function, called show\_in\_excel\_penguins(), opens in Excel the data about penguins we used in Chapter 7.

We first import the data with the read\_csv() function before creating the show\_in\_excel\_penguins() function:

library(tidyverse)

library(fs)

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)

}

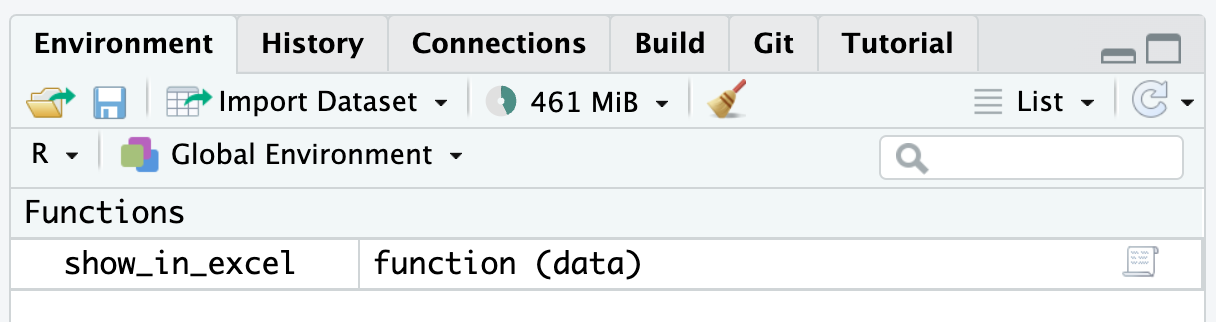
We load the tidyverse and fs packages. We’ll use tidyverse to create a filename for the CSV file and save it. The fs package will enable us to open the CSV file in Excel (or whichever program your computer uses to open CSV files by default).

We give the function a name 1, then use the assignment operator (<-) and function() to specify that show\_in\_excel\_penguins is not 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 our case, the body does three things:

1. 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.
2. Writes penguins to the location set in csv\_file. The x argument in write\_csv() refers to the data frame to be saved. We also make all NA values show up as blanks. By default, they will show up as the text NA.
3. Uses the file\_show() function from the fs package to open the temporarily created CSV file in Excel.

If you wanted to use the show\_in\_excel\_penguins() function, you would run the lines where you define the function by highlighting those lines and hitting command - enter on macOS and ctrl - enter on Windows. You should see the function show up in your global environment, as shown in Figure 12-1.

[F12001.png]



* + - * 1. The function we created 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

Now, you’re probably thinking that the function we wrote 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 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 it to easily share data with his non-R-user colleagues. Whenever he needed data in a CSV file, he could run his function.

To make such a function, we need to add arguments. Below is a slightly simplified version of the actual code that Bruno Rodrigues used. It looks the same as our show\_in\_excel\_penguins() function, with two exceptions:

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

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

write\_csv(x = data,

file = csv\_file,

na = "")

file\_show(path = csv\_file)

}

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

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

show\_in\_excel(data = penguins)

Having created the function with the data argument, we can run it with any data we want to. This code, for example, will import COVID case data we saw in Chapter 10 and open 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 this function 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 re-run 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

Now that you understand how functions work, let’s walk through an example function you could use to simplify some of the activities discussed in previous chapters.

In Chapter 11, when you used the tidycensus package to easily import data from the United States Census Bureau, you learned that it can be hard to remember the names of the census data’s many variables. Say you regularly want to access data about race and ethnicity from the American Community Survey, but never remember which variables enable you to do so. In this section, we’ll create the get\_acs\_race\_ethnicity() function to help with this task, walking through its development step-by-step to show some important concepts about making functions.

A first version of this 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’s body, we call the get\_acs() function from tidycensus to retrieve population data at the state level. But instead of returning the function’s default output, we relabel the hard-to-remember variable names to human-readable names, such as White and Black/African American. Next, we save these names as an object called race\_ethnicity\_data. Lastly, we use this object to return that data when the function is run.

We can run this function by entering 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 Nati… 17417 941

#> 4 01 Alabama Asian 69331 1559

#> 5 01 Alabama Native Hawaiian/Pacific Isl… 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

#> # ℹ 406 more rows

We could improve this function in a few ways. For example, you might want the resulting variables names to follow a consistent syntax. To do this, you could use the clean\_names() function from the janitor package, which formats all variable names using snake case (in which all words are lowercase and separated by underscores). In snake case, the variable name GEOID would become geoid. However, you might also want to leave yourself the option of keeping the original variable names. We can accomplish this by adding an argument to the function:

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) 1

}

race\_ethnicity\_data

}

We add the clean\_variable\_names argument to the function and specify that it should be FALSE by default. Then, in the function body, we set up an if statement. If the argument is TRUE, we run a line that overwrites the variable names with versions formatted in snake case 1. If the argument is FALSE, we leave the variable names unchanged.

If you run the function, nothing should change, because our 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 Nati… 17417 941

#> 4 01 Alabama Asian 69331 1559

#> 5 01 Alabama Native Hawaiian/Pacific Isl… 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

#> # ℹ 406 more rows

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

Using ... to Pass Arguments to Another Function

The function we created retrieves population data at the state level by passing the geography = "state" argument to the get\_acs() function. But what if we wanted to obtain county-level data, or else data about census tracts? We could do so using get\_acs(), but we’ve written the get\_acs\_race\_ethnicity() function in a way that isn’t flexible enough to allow this.

Your first thought about how to modify our function might involve adding an additional argument for the level of data to retrieve. We 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:

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

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

--snip--

But what if I also want to select the year for which to retrieve data? Well, we 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 our code would become cumbersome.

The ... syntax allows us to avoid doing so. By placing ... in the get\_acs\_race\_ethnicity() function, we can automatically pass any arguments listed there to the get\_acs() function by including ... there 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 this 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 Nati… 17417 941

#> 4 01 Alabama Asian 69331 1559

#> 5 01 Alabama Native Hawaiian/Pacific Isl… 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

#> # ℹ 406 more rows

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

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

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 Native 11330

#> 4 56 Wyoming Asian 4907

#> 5 56 Wyoming Native Hawaiian/Pacific Islander 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 it the arguments of 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 that we’ve discussed how to make functions, you’ll learn how to put them into a package.

Creating a Package

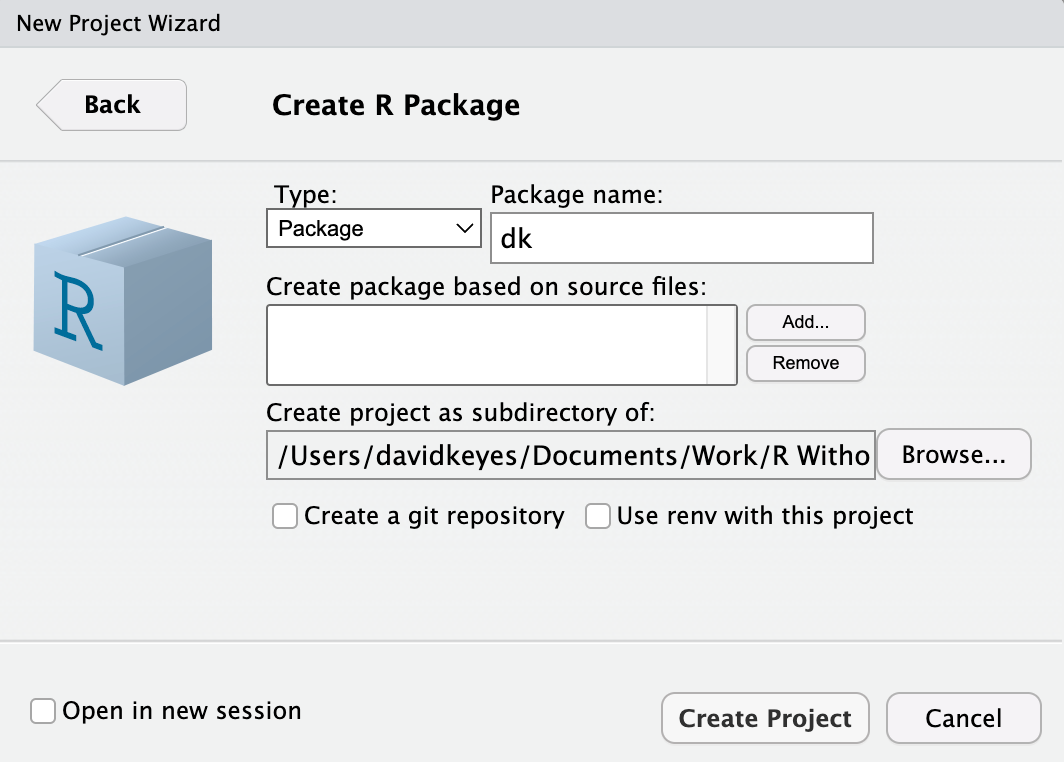
Packages bundle your functions so you can use them in multiple projects. If you find yourself copying functions from one project to another, or perhaps have a set of functions you’ve saved in a functions.R file that you copy into each new project, these are good indications that you should make a package.

While you can run the functions from a functions.R file in your own environment, this code in may 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. If you put your functions in a package, they are more likely to work, as they include necessary dependencies. R packages also contain built-in documentation to help others use the functions on their own.

Starting the Package

To create a package in RStudio, go to the File menu, then select New Project. From there, select New Directory. You’ll be given a list of options, one of which is R Package. Select it, 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 pre-built function called hello() that, when run, prints the text “Hello, world!” in the console. Let’s get rid of it and a few other default files so we can start with a clean slate. Delete hello.R, NAMESPACE, and hello.Rd in the man directory.

Adding Functions with use\_r()

All 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, we’ll use the usethis and devtools packages. Install them using install.packages() if you don’t already have them installed:

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 makes it possible to use the function without loading the package. This function should create a file called acs.R in the R directory with the name you give it as an argument. The name of the file itself doesn’t really matter, but choosing something that gives an indication of the functions within it is good practice. 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

We need to change the get\_acs\_race\_ethnicity() in a few ways to make it work in a package. The easiest way to figure out what changes we need to make is to use built-in tools to check that our 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 an R CMD check on the dkpackage gives us a long message. The last part is the most important:

── 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 ✖

Let’s review the output from bottom to top. The line 0 errors ✔ | 2 warnings ✖ | 1 note ✖ highlights three levels of issues identified in our 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 understand what R CMD check is saying here, we 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 are 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 package. But to make sure that the appropriate packages are installed for any user of the dk package, we have to make a few changes.

R CMD check tells us we have several “undefined global functions or variables” and “no visible global function definition” for various functions. This is because we are attempting to use functions from the tidycensus and janitor packages, but we haven’t specified where these functions come from. I can run this code in my environment because I have tidycensus and janitor installed, but we can’t assume the same of others.

Adding Dependency Packages

To ensure the package’s code will work, we 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:

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 we’re working in the dk project. The second line indicates that the DESCRIPTION file has been edited. This file provides meta information about the package we’re developing.

Next, in order to use the clean\_names() function, you’ll need to add the janitor package using the code 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

Look for the Imports section way down at the bottom of the file. Its contents indicate that when a user installs the dk package, the tidycensus and janitor packages will also be installed for them.

Referring to Functions Correctly

The R CMD check output also included this line: “Refer to functions with tidycensus::fun()” (where fun() stands for function name) This tells us that, in order to use functions from other packages in the dk package, we 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. Remember when we ran R CMD check and got this?

Undefined global functions or variables:

clean\_names get\_acs

This is because we 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 we 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 that we’ve specified which packages the functions come from, we can run devtools::check() again. When you do so, 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 we need to deal with. Let’s do that next.

Creating Documentation with Roxygen

Take a look at the “checking for missing documentation entries” warning. This warning tells us that we need to document our 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, we want them to be able to enter ?get\_acs\_race\_ethnicity() to learn how it works.

To create documentation for the function, we’ll use Roxygen, a documentation tool that uses a package called roxygen2. To get started, place your cursor anywhere in the function. Then, go to the Code menu in RStudio and select Insert Roxygen Skeleton. Doing this should add text above the get\_acs\_race\_ethnicity() function that looks like this:

#' 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 we’re working with Roxygen. We can now edit the text to create our 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 of our function’s arguments, but it’s up to us 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 our case, it returns data, which I document as follows:

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

Below @return is @export. We don’t need to change anything here. Most functions in a package are known as exported functions, meaning they are available to users of the package. In contrast, internal functions, which are used only by the package developers, do not 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 we’ll skip it here and delete the line with @examples on it. 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 is a great resource.

Now that we’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, since it is 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)

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

Adding a License and Metadata

Try running devtools::check() again to see if we’ve fixed the issues that led to the warnings. The warning about missing documentation should no longer be there. However, we do still have 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 tells us that we haven’t given our 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, we use the MIT license. which allows people 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. Doing so returns the following:

✔ 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 our package. Most importantly for us, it specifies the license in the DESCRIPTION file. If you open it, you should see a 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. We can make a few changes to it 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, let’s run devtools::check() one more time to make sure everything is in order:

0 errors ✔ | 0 warnings ✔ | 0 notes ✔

We get exactly what we hoped to see.

Writing Additional Functions

You’ve now got a package with one working function in it. If you wanted to add additional 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 of 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

Having developed the package, we’re now ready to install and use it. 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 we can cover here, but the book Happy Git and GitHub for the useR by Jenny Bryan is a great place to get started.

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

Conclusion

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 they can allow advanced R users to help those with less experience. When Gerke and Aden-Buie provided researchers at the Moffitt Cancer Center with a package that contained functions for easily accessing their databases, the researchers began 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.

Learn More

Consult the following resources to learn how to make your own packages in R:

R Packages, Second Edition by Hadley Wickham and Jennifer Bryan (O'Reilly Media, Forthcoming), <https://r-pkgs.org/>

Package Development with R, course by Malcolm Barrett, <https://rfortherestofus.com/courses/package-development-course/>

Wrapping Up

R was invented in 1993 as a tool for statistics, and in the years since then, 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 to automate how you access data, as well as assist you in collaborating with others through functions and packages that you make.

If R was new to you when you started this book, you should now feel inspired to use it. If you are an experienced R user, this book should have shown you ways to use R that you hadn’t previously considered. No matter your background, you should understand how to use R like a pro, and without any statistics.