Creating Your Own R 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 the data about penguins that we used in Chapter 7 in Excel. We begin by importing the data with the read\_csv() function before creating the show\_in\_excel\_penguins() function:

library(tidyverse)

library(fs)

Functions have three pieces:

1. Name
2. Body
3. Arguments

Before we look at the show\_in\_excel() function, let’s look at a slightly altered version to help us understand how functions work. This function, called show\_in\_excel\_penguins() opens the data on penguins that we saw in 7 in Excel. We begin by importing the data with the read\_csv() function before creating the show\_in\_excel\_penguins() function.

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)

}

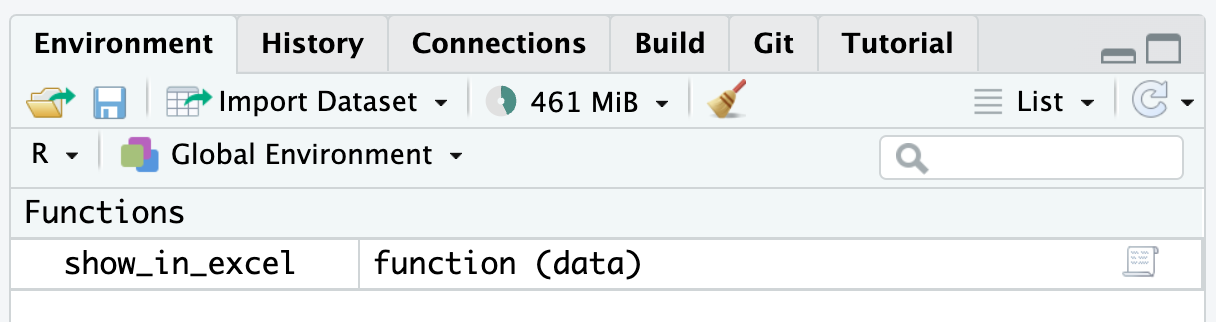
We load the tidyverse and fs packages. We’ll use tidyverse to create a filename for the CSV file and save it. The fs package will enable us to open the CSV file in Excel (or whichever program your computer uses to open CSV files by default).

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 *.csv* extension 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. Highlight the lines where you define the function and hit command/control (Mac/Windows) and Enter at the same time and you will see the function show up in your global environment, as seen in Figure 12-1.

[F12001.png]



* + - * 1. The function we created in the global environment

From there on out, any time you run the code show\_in\_excel\_penguins(), R will open up 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 Automatically Format Race and Ethnicity Data

Let’s look at one more function to make sure you understand how functions work. As we saw in Chapter 12, we can use the tidycensus package to easily pull in data directly from the United States Census Bureau. As we also saw in that same chapter, it can be hard to remember the names of variables. I regularly want to access data on population by race and ethnicity from the American Community Survey, but I never remember which variables I need to access to do so. I’ve created the get\_acs\_race\_ethnicity() function to help me. I’ll walk through its development step-by-step in order to show some important concepts about making functions.

A first version of this function might look like this. My get\_acs\_race\_ethnicity() function calls the get\_acs() function from tidycensus, getting data at the state level. Instead of returning the default output, it relabels the race and ethnicity groups from the hard-to-remember variable names to actual names, such as White and Black/African American. The get\_acs\_race\_ethnicity() function saves the result of the get\_acs() call as an object called race\_ethnicity\_data. And then, at the bottom of the function, the line with race\_ethnicity\_data on its own returns that data when the function is run.

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  
   
}

Once I define this function, I can now run it in the future as follows:

get\_acs\_race\_ethnicity()

Doing so will return the 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

When I use this function, I sometimes like to have my variables names returned in a consistent syntax. To do this, I could use the clean\_names() function from the janitor package, which makes all variable names use snake case (that is all words lower case and underscores between words). In our example above, I’d like to use the variable name geoid rather than GEOID. However, I want to leave myself the option to keep the original variable names. To do this, I need to add an argument to my function. Below, I’ve added the clean\_variable\_names argument to the get\_acs\_race\_ethnicity() function. I’ve also specified that it should be FALSE by default. Then, in the function body, I set up an if statement. If the clean\_variable\_names argument is TRUE, the line race\_ethnicity\_data <- clean\_names(race\_ethnicity\_data) is run, overwriting race\_ethnicity\_data with a version that has snake case names. If the clean\_variable\_names argument is FALSE, race\_ethnicity\_data is unchanged.

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  
   
}

If I run get\_acs\_race\_ethnicity(), nothing will change because clean\_variable\_names is FALSE by default. However, if I want to set clean\_variable\_names to TRUE, I can do it as follows:

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

And this will return my data with clean 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 in both the show\_in\_excel() and get\_acs\_race\_ethnicity() functions, let’s talk about how you can pass arguments from the function you make to another function.

Using ... to Pass Arguments to Another Function

In my get\_acs\_race\_ethnicity() function above, I made a choice to get the data at the state level using geography = "state" within the get\_acs() function. But what if I want to get the data at the county level? Or the census tract level? These are all possible with get\_acs(), but I’ve written my get\_acs\_race\_ethnicity() function in a way that is not flexible enough to allow this.

Your first thought on how to do this might involve adding an additional argument for the geography. Something like the code below, which adds a my\_geography argument and then uses it within the get\_acs() function.

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

But what if I want to also select the year? Well, I could add an argument for that as well. As we saw in Chapter 12, the get\_acs() function has many arguments. I don’t want to repeat them all as arguments in my get\_acs\_race\_ethnicity() function, and the ... allows us to avoid doing so. By putting ... in the get\_acs\_race\_ethnicity() function, any arguments listed there are then passed to the get\_acs() function (note the ... 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

}

I can then run this code to set the geography to state.

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

Which would 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

Or I could switch this to get data by county.

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

Which would return the following:

#> # A tibble: 25,768 × 5

#> GEOID NAME variable estimate moe

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

#> 1 01001 Autauga County, Alabama White 42571 215

#> 2 01001 Autauga County, Alabama Black/Afric… 11464 309

#> 3 01001 Autauga County, Alabama American In… 98 85

#> 4 01001 Autauga County, Alabama Asian 647 206

#> 5 01001 Autauga County, Alabama Native Hawa… 0 30

#> 6 01001 Autauga County, Alabama Other race 109 140

#> 7 01001 Autauga County, Alabama Multi-Race 1575 397

#> 8 01001 Autauga County, Alabama Hispanic/La… 1775 NA

#> 9 01003 Baldwin County, Alabama White 187548 959

#> 10 01003 Baldwin County, Alabama Black/Afric… 19749 712

#> # ℹ 25,758 more rows

I could also run my get\_acs\_race\_ethnicity() function again and add geometry = TRUE as follows.

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

geometry = TRUE)

My get\_acs\_race\_ethnicity() function will now return geospatial data alongside my demographic data.

#> 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 ... allows me to create my own function and pass any arguments from it to another function without repeating all of the child function’s arguments. This gives me flexibility while maintaining brevity.

Now that we’ve discussed how to make functions, let’s talk about how to put them into a package.

How to Create 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.

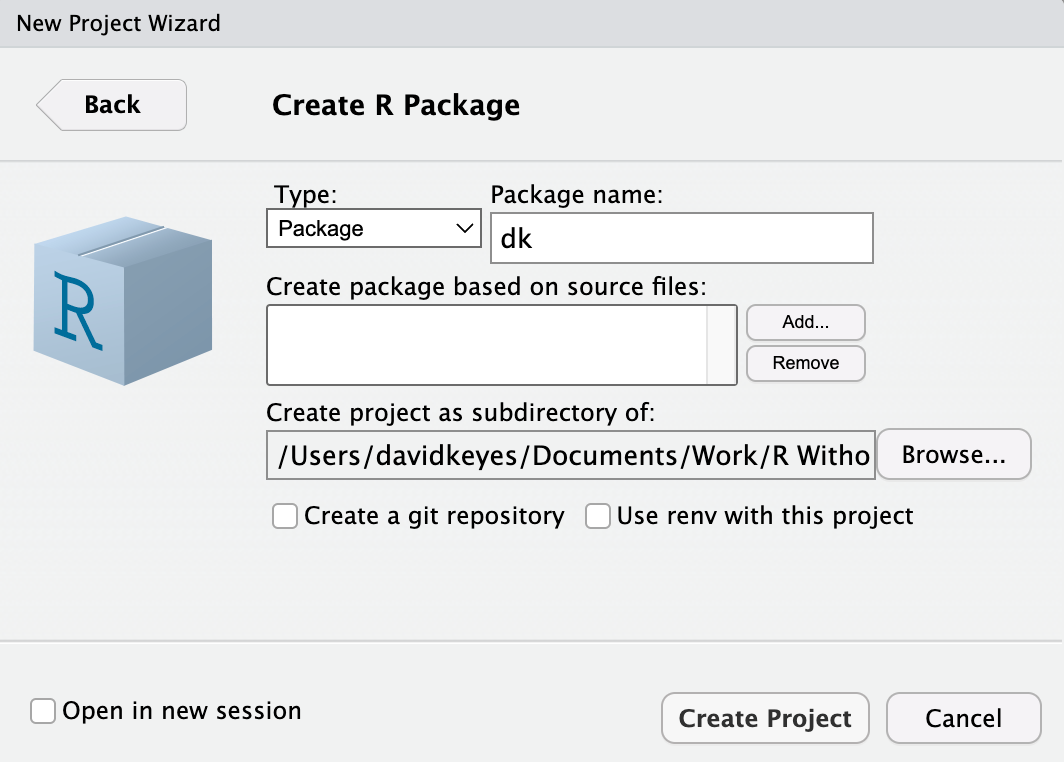
While you can run the functions from a functions.R file in your own environment, this code in may not work on someone else’s computer. Other users may not have the necessary packages installed, or they may be confused about how your functions’ arguments work and not know where to go for help. If you 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 them use the functions on their own.

The get\_acs\_race\_ethnicity() function I made is an example of this. I can use it in any project where I need race and ethnicity data.

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 it to live on your computer. You can leave everything else as is.

[F12002.png]



* + - * 1. The RStudio menu for creating your 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 our 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 R CMD check, which makes sure that others can install your package on their system. Running R CMD check on the dk package gives us a long message. The last part is the most important:

── R CMD check results ─────────────── dk 0.1.0 ────

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 ✖” gives us the three levels of issues with 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.

Let’s start with the note at 1. To understand what R CMD check is saying, 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, while you are telling R to install one package, that single package likely uses functions from other packages. In order to have access to these functions, R will install these packages (known as dependencies) for you. It would be a pain if, every time you installed package, you had to manually install a whole set of dependencies. But in order to make sure that the appropriate packages are installed for any users of the dk package, we have to make a few changes.

When we run R CMD check, we are told that 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 that these functions are from tidycensus and janitor I can use this code because I have tidycensus and janitor installed, but we can’t assume that others will have them installed.

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()`

You’ll then need to run usethis::use\_package(package = "tidycensus"), which should give you the following output:

✔ Adding 'janitor' to Imports field in DESCRIPTION

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

The “Setting active project …” line tells me that it is working in the dk project. The second line in both outputs tells me that the DESCRIPTION file has been edited. This file provides meta information about the package we’re developing. 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 way down at the bottom of the file. This indicates 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()”. This tells us that, in order to use functions from other packages in the dk package, we need to specifying both the package name and the function name to ensure that the correct function is used at all times. In 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 come from. The clean\_names() function comes from the janitor package and get\_acs() comes from tidycensus so I need to add the package names before each in my 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 I’ve specified the packages the functions come from, I can run devtools::check() again. When I do so, I now 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, I can see that there are still two warnings that I need to deal with. Let’s do that next.

Adding Documentation

Take a look at the “checking for missing documentation entries” warning. This warning tells us that we need to add documentation of the 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 get\_acs\_race\_ethnicity(), we’ll use Roxygen, a way to document functions in R packages using 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:

Doing this will add some 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, you can see lines with the text @param. Roxygen automatically looks at the arguments in our function and creates a line for each one.

Next are lines beginning with the text @param. Roxygen automatically locates the arguments in the function and creates a line for each one. On the first line, we describe what the clean\_variable\_names argument does. On the second, we 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 only used 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. If you do want to learn more, the second edition of Hadley Wickham and Jenny Bryan’s book R Packages is a great place to learn about adding examples.

Now that we’ve added the documentation with Roxygen, run devtools::document() in the console. This will 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 change the 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.

Adding 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. This package, which can be installed by running install.packages("remotes"), allows you to install packages from a wide range of locations, including GitHub. Once you’ve installed the remotes package, 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.