Create Your Own R Package

In the late 2010s, Travis Gerke and Garrick Aden-Buie were working on the Collaborative Data Services Core team at the Moffitt Cancer Center. Researchers throughout Moffitt needed to access data from several highly secure databases, and Gerke and Aden-Buie were in charge of helping them. Initially, Gerke and Aden-Buie would write R code for each researcher. But they quickly realized that they were reusing the same code over and over. Why not instead share the code with the researchers so they could run it themselves?

Travis Gerke and Garrick Aden-Buie made a package with functions to access databases. In the process they simplified their own work and improved the work of the researchers they supported. No longer did researchers have to ask for help. They could now install the package Gerke and Aden-Buie had made and use its functions themselves.

In this chapter, I discuss how to make your own R package. While privacy concerns preclude me demonstrating the package that Travis Gerke and Garrick Aden-Buie made, they walked me through the development of a simple package that includes a custom ggplot theme. Before getting into package making, though, we’ll look at making functions. We’ve talked about functions throughout this book, but in this chapter I’ll show you how you can make your own functions. Making your own functions can have a huge improvement on your coding efficiency. And bundling your functions into a custom package can help you work with others using shared code.

Creating Your Own Functions

Let’s begin by taking an example of a relatively simple function. It comes from Bruno Rodrigues, head of the Department of Statistics and Data Strategy at the Ministry of Higher Education and Research in Luxembourg. Rodrigues is an avid R user, but the same cannot be said for all of his teammates. So he needs a way to be able to easily share data with his non-R-user colleagues. Being the avid R user that he is, he wrote some R code to help him with this task. Specifically, he wrote a function called show\_in\_excel() that would take a data frame from R, save it as a CSV file, and open the CSV file in Excel. From then on, whenever he needed to have his data in a CSV to share with colleagues, he could run his function and share away.

Let’s look at the show\_in\_excel() function that Bruno Rodrigues created. We begin by loading the tidyverse and fs packages. We’ll use the tidyverse, which you’ve seen throughout this book, to create a file name and save our CSV. 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).

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.

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

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

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

write\_csv(x = penguins,

file = csv\_file,

na = "")

file\_show(path = csv\_file)

}

To understand how our function works, take a look at the first line. In this line, we begin by giving our function a name: show\_in\_excel\_penguins. We next use the assignment operator (<-) and function() to specify that show\_in\_excel\_penguins is not a variable name, but a function name. There’s an open curly bracket ({) at the end of the line, which indicates the start of the function body.

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

The meat of the function can be found in its body. This is all of the code in between the open and closed curly brackets.

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

# Function body goes here

}

In the case of the show\_in\_excel\_penguins() function we’ve created, 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 (the x argument in write\_csv() refers to the data frame to be saved) to the location set in csv\_file. 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.

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

write\_csv(x = penguins,

file = csv\_file,

na = "")

file\_show(path = csv\_file)

If you wanted to use the show\_in\_excel\_penguins() function, you would run the lines where you define the function. From there on out, any time you run the code show\_in\_excel\_penguins(), R will open up the penguins data frame in Excel.

Now, you’re probably thinking: that doesn’t seem like the most useful function. Why would I want to keep opening up the penguins data frame? You wouldn’t, of course. To make your function more useful, we need to add arguments. Below is a slightly simplified version of the actual code that Bruno Rodrigues used. Everything looks the same as our show\_in\_excel\_penguins() function with one exception: 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 one other change. Within write\_csv(), instead of x = penguins, we now have x = data. This allows us to use the function with any data, not just penguins.

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

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

write\_csv(x = data,

file = csv\_file,

na = "")

file\_show(path = csv\_file)

}

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

show\_in\_excel(data = penguins)

But, 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 only include 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 creates the code once and can run it as many times as necessary.

Creating a ggplot Theme Function

Let’s look at one more function to make sure you understand how functions work. In the show\_in\_excel() function, nothing is changed within R; it simply allows us to view data in Excel. A more typical use case is to create a function that you use while working in R to do something to your data or, in the case below, plots. A common example is creating your own ggplot theme.

We’ve seen ggplot themes in Chapters 3 and 5, but these uses were one-offs. Chapter 4 showed how the BBC made a custom theme to be used across projects. This multi-project use case shows the value of creating a function. Rather than copying your ggplot theme code from one project to the next, you can create it once and use it across all projects.

Below, we can see a custom theme that I’ve created for myself. Called theme\_dk(), this function sets the defaults that I like to use in my plots, including removing axis ticks, adjusting the font size of axis titles as well as the plot title and subtitle, and putting the legend above the plot and increasing its font size.

theme\_dk <- function(show\_grid\_lines = TRUE,

show\_axis\_titles = TRUE) {

custom\_theme <- theme\_minimal() +

theme(panel.grid.minor = element\_blank(),

axis.ticks = element\_blank(),

axis.title = element\_text(size = 12,

color = "grey50"),

axis.title.x = element\_text(margin = margin(t = 10)),

axis.title.y = element\_text(margin = margin(r = 10)),

axis.text = element\_text(size = 12,

color = "grey50"),

plot.title.position = "plot",

plot.title = element\_text(size = 20,

face = "bold",

margin = margin(b = 8)),

plot.subtitle = element\_text(size = 14,

color = "grey50"),

legend.text = element\_text(size = 12),

legend.position = "top")

if (show\_grid\_lines == FALSE) {

custom\_theme <- custom\_theme +

theme(panel.grid.major = element\_blank())

}

if (show\_axis\_titles == FALSE) {

custom\_theme <- custom\_theme +

theme(axis.title = element\_blank(),

axis.title.x = element\_blank(),

axis.title.y = element\_blank())

}

custom\_theme

}

I can then apply my theme as I would any other ggplot theme.

penguins %>%

ggplot(aes(x = bill\_length\_mm,

y = bill\_depth\_mm,

color = island)) +

geom\_point() +

labs(title = "A histogram of bill length and bill depth",

subtitle = "Data from palmerpenguins package",

x = "Bill Length",

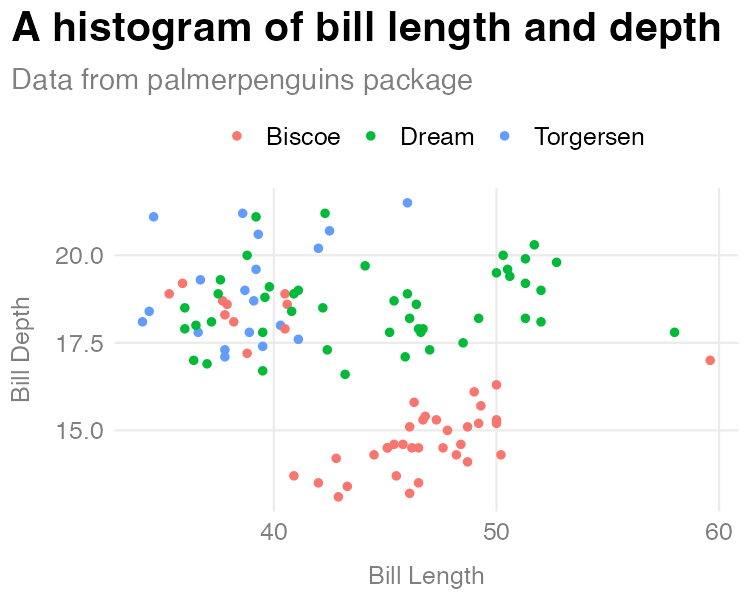
y = "Bill Depth",

color = NULL) +

theme\_dk()

In Figure 12-1 below, we can see what a histogram made with theme\_dk() looks like using the same penguins data.

[F12001.pdf]



* + - * 1. A histogram made with my custom theme

Let’s now discuss how the custom ggplot theme function works. The first three lines define our function and provide two arguments: show\_grid\_lines and show\_axis\_titles. Whereas the data argument in show\_in\_excel() required us to give the function data, the arguments in theme\_dk() have defaults built in. The lines show\_grid\_lines = TRUE and show\_axis\_lines = TRUE mean that grid lines and axis titles will be visible on our plot by default.

The first piece of the code in theme\_dk() starts with theme\_minimal() (for more on the specifics of making a custom theme refer to how the bbc\_style() function was created in Chapter 4). The changes that are then made with the theme() function are saved as an object called custom\_theme.

custom\_theme <- theme\_minimal() +

theme(panel.grid.minor = element\_blank(),

axis.ticks = element\_blank(),

axis.title = element\_text(size = 12,

color = "grey50"),

axis.title.x = element\_text(margin = margin(t = 10)),

axis.title.y = element\_text(margin = margin(r = 10)),

axis.text = element\_text(size = 12,

color = "grey50"),

plot.title.position = "plot",

plot.title = element\_text(size = 20,

face = "bold",

margin = margin(b = 8)),

plot.subtitle = element\_text(size = 14,

color = "grey50"),

legend.text = element\_text(size = 12),

legend.position = "top")

One difference between theme\_dk() and bbc\_style() is that theme\_dk() is a bit more flexibile. I do this using two if statements. While I defined the default plot to show grid lines and axis titles, if the user sets the show\_grid\_lines and show\_axis\_titles arguments to FALSE, those elements will be removed. The if statements test whether show\_grid\_lines and show\_axis\_titles are set to TRUE. If so, it changes the custom\_theme object. At the end of the function, we do what’s called returning an object. In this case, we return the custom\_theme object, meaning that custom\_theme, with or without our changes to grid lines and axis titles is returned, making it available to use in our plot.

To show an example of when we might want to remove grid lines and axis titles, I’ll create a simple bar chart of the number of penguins on each island.

penguins %>%

count(island) %>%

ggplot(aes(x = island,

y = n)) +

geom\_col() +

labs(title = "Number of penguins on each island",

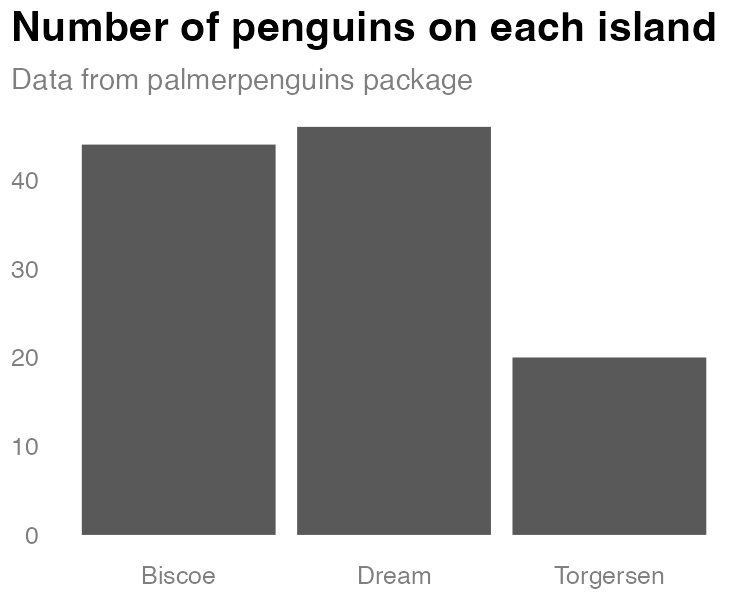
subtitle = "Data from palmerpenguins package") +

theme\_dk(show\_grid\_lines = FALSE,

show\_axis\_titles = FALSE)

When I use theme\_dk() here, I set show\_grid\_lines and show\_axis\_titles to FALSE, removing those elements from our chart. We can see the result in Figure 12-2.

[F12002.pdf]



* + - * 1. A bar chart using my custom theme

Creating a function with default arguments allows us to set the options that we are most likely to want, while also giving us flexibility to change the arguments each time we use the function.

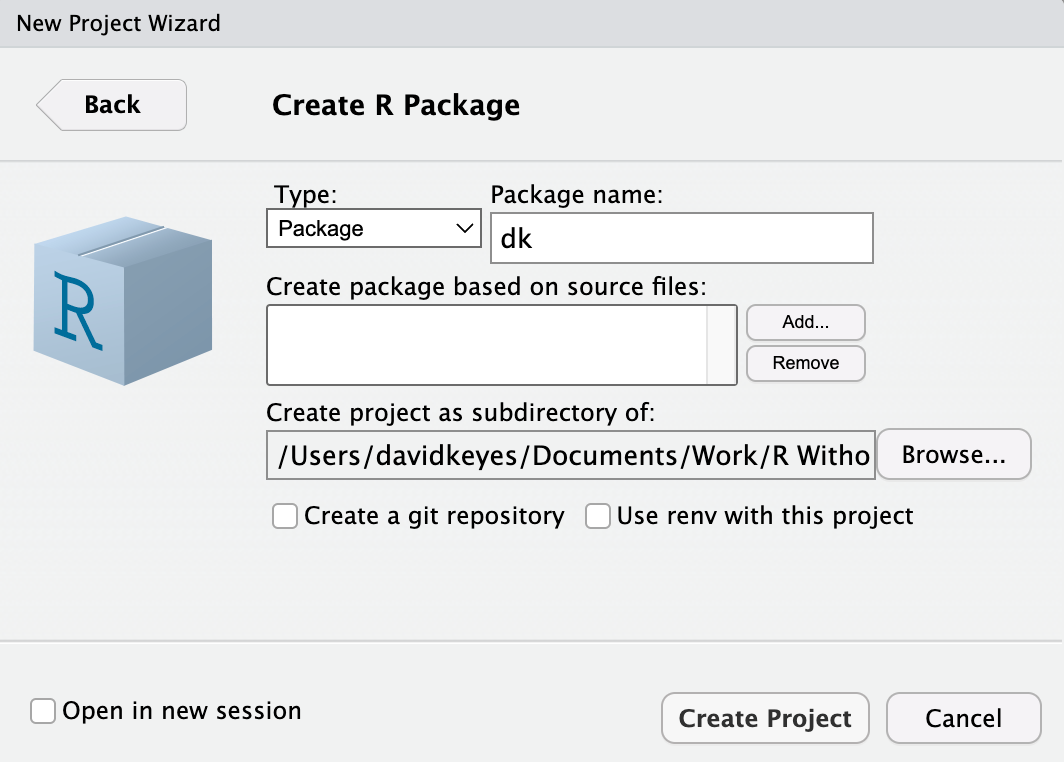
How to Create a Package

Hadley Wickham, developer of the tidyverse set of packages, recommends creating a function once you’ve copied code three times. There’s another step on the road to coding efficiency: bundling your functions into a package. How do you know whether functions you write are good candidates for a package? The answer is simple: if you plan to use the function in multiple projects, put it into a package. You may find yourself copying functions from one project to another. Or perhaps you have a set of functions you’ve saved in a functions.R file that you copy into each new project you work. Both are good indications that you’re ready for a project (Travis Gerke of the Moffitt Cancer Center said that a functions.R file “screams ‘make a package’”). The theme\_dk() function I made is an example of this. I can use it in any project where I make data visualization so let’s put it in a package.

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 that and you’ll then need to give your package a name (I’m going to call mine dk) and decide where you want it to live on your computer. You can leave everything else as is.

[F12003.png]



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

RStudio will now create and open my package. There are a few files in the package already, including hello.R, which has a prebuilt 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’re starting with a clean slate. I’ll delete hello.R, NAMESPACE, and the hello.Rd file in the man directory.

Much of the work we’ll do working with our package relies on the usethis and devtools packages. Install those using install.packages() if you don’t already have them installed. Once you’ve done that, you’re ready to add a function to the package. To do so, run the use\_r() function from the usethis package. I typically run functions from usethis and devtools in the console because I only need to run them once. I also tend to run them with the syntax package::function(), which makes it possible to use a function without loading the package (it’s also how we have to refer to functions in our package, as we’ll see below). Here’s how I would run the use\_r() function:

usethis::use\_r("theme")

This function will create a file called theme.R in the R directory with the name you give it as an argument (all functions in a package go in files in the R folder). I can open up the file and add code to it. I’ll begin by copying the theme\_dk() function I created. We know theme\_dk() works, but we need to change it 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.

Checking our Package

To test that our package works, run the function devtools::check() (typically done in the console). This runs 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: 9.5s

❯ 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:

‘theme\_dk’

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

See chapter ‘Writing R documentation files’ in the ‘Writing R

Extensions’ manual.

❯ checking R code for possible problems ... NOTE

theme\_dk: no visible global function definition for ‘theme\_minimal’

theme\_dk: no visible global function definition for ‘theme’

theme\_dk: no visible global function definition for ‘element\_blank’

theme\_dk: no visible global function definition for ‘element\_text’

theme\_dk: no visible global function definition for ‘margin’

Undefined global functions or variables:

element\_blank element\_text margin theme theme\_minimal

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 (meaning others won’t be able to install your package), followed by warnings and notes (both of which may cause problems for others). It’s best practice to eliminate all errors, warnings, and notes. Let’s start with the note, which shows up in this section:

❯ checking R code for possible problems ... NOTE

theme\_dk: no visible global function definition for ‘theme\_minimal’

theme\_dk: no visible global function definition for ‘theme’

theme\_dk: no visible global function definition for ‘element\_blank’

theme\_dk: no visible global function definition for ‘element\_text’

theme\_dk: no visible global function definition for ‘margin’

Undefined global functions or variables:

element\_blank element\_text margin theme theme\_minimal

To understand what R CMD check is saying, we need to explain a bit about how packages work.

Adding Dependency Packages

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 ggplot2 package, but we haven’t specified that these functions are from ggplot2. I can use this code because I have ggplot2 installed, but we can’t assume that others will have ggplot2 installed. In order to ensure the code will work, we need to install ggplot2 for users when they install the dk package. To do this, we run the use\_package() function from the usethis package as follows:

usethis::use\_package(package = "ggplot2")

After I run the use\_package() code in the console, I get the following message:

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

✔ Adding 'ggplot2' to Imports field in DESCRIPTION

• Refer to functions with `ggplot2::fun()`

The first line tells me that it is working in the dk project. The second line tells me that the DESCRIPTION file has been edited. This file provides meta information about the package we’re developing. If we open up the DESCRIPTION file (it’s in the root directory of our project), we 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:

ggplot2

Way down at the bottom, look for the line that says “Imports:” followed by “ggplot2” on the next line. This indicates that, when a user installs the dk package, the ggplot2 package will also be installed for them. It’s important to note that, while I use the tidyverse package (this meta package contains multiple packages, of which ggplot2 is just one) when working on my own computer, when making your own package you want to have it install the individual packages needed to make it run.

Referring to Functions Correctly

We’ll return to the DESCRIPTION file in a bit, but for now, let’s take a look at this line: “Refer to functions with ggplot2::fun()”. It is telling us that, in order to use functions from other packages in the dk package, we need to refer to them in a unique way. By specifying both the package name and the function name, you ensure that the correct function is used at all times. It is rare, but there are functions with identical names across multiple packages. Using this syntax avoids any ambiguity. Remember when we ran R CMD check and got this?

Undefined global functions or variables:

element\_blank element\_text margin theme theme\_minimal

This is because we were using functions without saying what package they come from. All of the functions listed here come from the ggplot2 package so I can add ggplot2:: before them as follows:

theme\_dk <- function(show\_grid\_lines = TRUE,

show\_axis\_titles = TRUE) {

custom\_theme <- ggplot2::theme\_minimal() +

ggplot2::theme(panel.grid.minor = ggplot2::element\_blank(),

axis.ticks = ggplot2::element\_blank(),

axis.title = ggplot2::element\_text(size = 12,

color = "grey50"),

axis.title.x = ggplot2::element\_text(

margin = ggplot2::margin(t = 10)

),

axis.title.y = ggplot2::element\_text(

margin = ggplot2::margin(r = 10)

),

axis.text = ggplot2::element\_text(size = 12,

color = "grey50"),

plot.title.position = "plot",

plot.title =

ggplot2::element\_text(size = 20,

face = "bold",

margin = ggplot2::margin(b = 8)

),

plot.subtitle = ggplot2::element\_text(size = 14,

color = "grey50"),

legend.text = ggplot2::element\_text(size = 12),

legend.position = "top")

if (show\_grid\_lines == FALSE) {

custom\_theme <- custom\_theme +

ggplot2::theme(panel.grid.major = ggplot2::element\_blank())

}

if (show\_axis\_titles == FALSE) {

custom\_theme <- custom\_theme +

ggplot2::theme(axis.title = ggplot2::element\_blank(),

axis.title.x = ggplot2::element\_blank(),

axis.title.y = ggplot2::element\_blank())

}

custom\_theme

}

Now that I’ve specified that the various functions I’m using come from ggplot2, I can run the devtools::check() function 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:

‘theme\_dk’

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 ✔

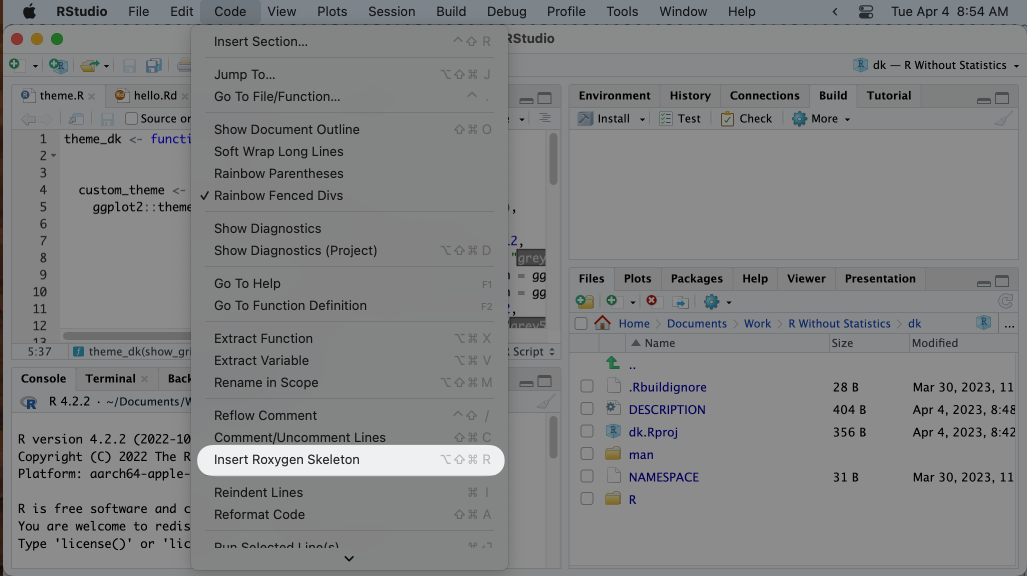
However, I can see that there are still two warnings that I need to deal with. Let’s do that next.

Adding Documentation

Let’s look now at the “checking for missing documentation entries” warning. This warning is telling us that we need to add documentation of our theme\_dk() 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 type ?theme\_mimimal() and see documentation about that function, we want them to also be able to type ?theme\_dk() and learn how it works.

To create documentation for theme\_dk(), we’ll use what’s known as Roxygen, which is a way to create documentation of 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 and select Insert Roxygen Skeleton, as seen in Figure 12-4 below.

[F12004.png]



* + - * 1. The menu item to insert Roxygen skeleton

Doing this will add some text above the theme\_dk() function that looks like this:

#' Title

#'

#' @param show\_grid\_lines

#' @param show\_axis\_titles

#'

#' @return

#' @export

#'

#' @examples

This text is the skeleton of what is needed to create documentation of the theme\_dk() function. Each line starts with the special characters #', which indicate we’re working with Roxygen. We can now go in and edit the Roxygen text to create our documentation. Starting at the top, I’ll replace Title with a sentence that describes my function.

#' David Keyes's custom ggplot theme

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. On each line, we want to describe what the argument does. I can do this as follows.

#' @param show\_grid\_lines Should grid lines be shown (TRUE/FALSE)

#' @param show\_axis\_titles Should axis titles be shown (TRUE/FALSE)

Going down the Roxygen skeleton, we next see @return. This is where we tell the user what the dk\_theme() function returns. In our case, it is a complete ggplot2 theme, which I document as follows:

#' @return A complete ggplot2 theme

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 I’m going to skip it. If you do want to learn more, Chapter 17 of 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, there’s one more step: running devtools::document(). This will do two things:

1. **Create a** theme\_dk.Rd **file in the** man **directory**. This file is the documentation of our theme\_dk() function in 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.
2. **Create a** NAMESPACE **file**. This file lists the functions that your package makes available to users. My NAMESPACE file now looks like this:

# Generated by roxygen2: do not edit by hand

export(theme\_dk)

The theme\_dk() function is now almost ready for users.

Editing the DESCRIPTION File

We can run devtools::check() again to see if we have fixed the issues that led to the warnings. Doing so shows that the warning about missing documentation is no longer 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 is telling us that we haven’t given our package a license. For packages made publicly available, choosing a license is important (for information on how to choose the right one for your package, see <https://choosealicense.com/>). It’s less important since we’re developing a package for our personal use, but we can still select one. I’ll use the MIT license (it allows people to do essentially whatever they want with the 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 I open up that file, I can see confirmation that I’ve added the MIT license.

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: MIT + file LICENSE

Encoding: UTF-8

LazyData: true

Imports:

ggplot2

In addition to the license, the DESCRIPTION file contains meta data about the package. We can make a few changes to change the Title, add an Author and Maintainer, and add a Description. My 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:

ggplot2

Having made these changes, let’s run devtools::check() one more time to make sure everything is in order. When I do so, I get exactly what I hope to see:

0 errors ✔ | 0 warnings ✔ | 0 notes ✔

We’ve not got a package with one working function in it. If you wanted to add additional functions, you would follow the same exact procedure:

1. Create a new .R file with usethis::use\_r() (you can also combine multiple functions in a single .R file).
2. Developer 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 our 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, simply run the code remotes::install\_github("dgkeyes/dk") (make sure you have the remotes package installed first). Once you or others install the dk package, anyone can just add the line library(dk) in any R code then use the theme\_dk() function. We can recreate the penguin bill length and depth histogram from Chapter ?? with the following code.

library(dk)

library(tidyverse)

library(palmerpenguins)

penguins %>%

ggplot(aes(x = bill\_length\_mm,

y = bill\_depth\_mm,

color = island)) +

geom\_point() +

labs(title = "A histogram of bill length and depth",

subtitle = "Data from palmerpenguins package",

x = "Bill Length",

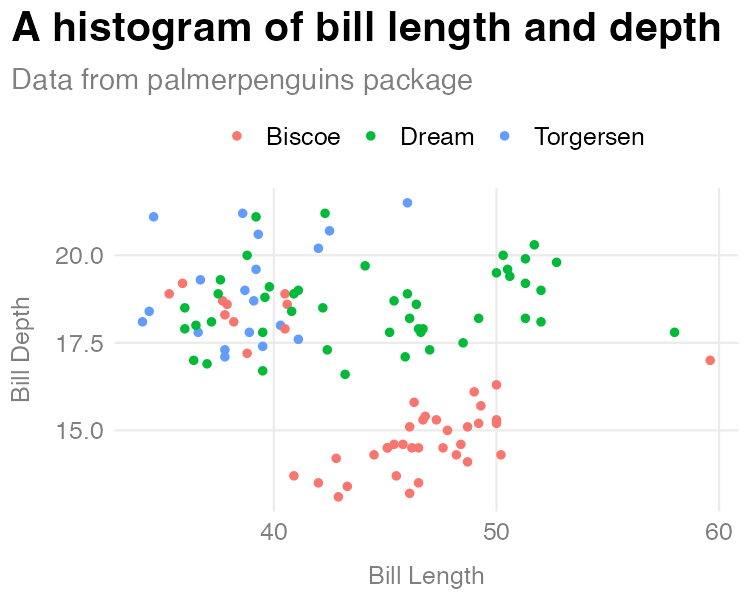
y = "Bill Depth",

color = NULL) +

theme\_dk()

Figure 12-5 shows the visualization (which is identical to that seen in Figure 12-1) that this code produces.

[F12005.pdf]



* + - * 1. A histogram made with the theme\_dk() function from the dk package

The histogram may be identical, but the process is not. While I had to manually define and run the code to create theme\_dk() in Chapter ??, here I only have to run library(dk) and I have access to the theme\_dk() function. And the great thing is that, if I were to make changes to theme\_dk(), users can simply reinstall the dk package to get access to the most up-to-date functions.

In Conclusion: You’re Already Ready to Make Your Own R Package

In my conversation with Travis Gerke and Garrick Aden-Buie of the Moffitt Cancer Center I had a realization: the name package refers to packaging up of several things that allow you and others to reliably run your code. A package is a set of functions, instructions to automatically install of dependency packages, and code documentation. If you write code that you share with others through a functions.R file, it may or may not work on their computer. They may not have the necessary packages installed. They may be confused about how arguments work and not know where to go. If you put your functions in a package, they are close to guaranteed to work. And users will have built-in documentation to help them use the functions on their own.

Creating your own R package may seem scary, but it’s not as complicated as you might think. In 2021, experienced package developer and educator Malcolm Barrett gave a talk titled “You’re Already Ready: Zen and the Art of R Package Development.” The abstract for the talk resonated deeply with me:

Many believe package development is too advanced for them or that they have nothing to offer. A fundamental belief in Zen is that you are already complete, that you already have everything you need. I’ll talk about why your project is already an R package, why you’re already an R package developer, and why you already have the skills to walk the path of development.

I hope, after reading this chapter, this message resonates with you as well.

Making packages can help you as an individual. Creating a ggplot theme like I did in this chapter, for example, can allow you to easily use it on any data visualization you make. Packages can be especially beneficial for organizations. This is exactly what Travis Gerke and Garrick Aden-Buie found during their time at the Moffitt Cancer Center. Accessing databases was so complicated that many people got stuck at that point. But when Gerke and Aden-Buie provided them with a package that contained functions to easily access the databases, Moffitt researchers began to do more with R. Gerke described the transformation: “People got to actually write code and do the data queries instead of handling administrative silliness.” Developing packages can allow more advanced users to help those with less experience using R. Aden-Buie told me that packages create “a number of gradual on-ramps for people who are getting more skills in coding.” Because more people are able to use these on-ramps, they come to see the value of using code.

What’s more, developing packages allows you as the package developer to shape how others work. Say you make a ggplot theme that follows the principles of high-quality data visualization (see Chapter 3). If you put this theme in a package, you can give others an easy way to follow these principles. Garrick Aden-Buie calls this a “happy path.” If you create a package, you can guide people on the happy path that you think is best. Packages are a way to ensure that others use best practices without them even being aware they are doing so.

Making your own R package is not as hard as you might think and comes with many benefits. Packages make it easy to reuse functions across projects, allow you to help others, and encourage best practices. If you haven’t ever made your own R package, now’s the time to give it a try.