[Creating Your Own Functions 2](#_Toc133321350)

[Writing a Simple Function 2](#_Toc133321351)

[Adding Arguments 3](#_Toc133321352)

[Creating a ggplot Theme Function 4](#_Toc133321353)

[Saving Functions in Packages 7](#_Toc133321354)

[Creating a Package in RStudio 7](#_Toc133321355)

[Adding Functions with use\_r() 8](#_Toc133321356)

[Checking the Package with devtools 8](#_Toc133321357)

[Adding Dependency Packages with the DESCRIPTION File 9](#_Toc133321358)

[Referring to Functions Correctly 10](#_Toc133321359)

[Adding Documentation with Roxygen 11](#_Toc133321360)

[Editing the DESCRIPTION File 13](#_Toc133321361)

[Installing the Package 14](#_Toc133321362)

[Conclusion 16](#_Toc133321363)



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 2. 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)

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)

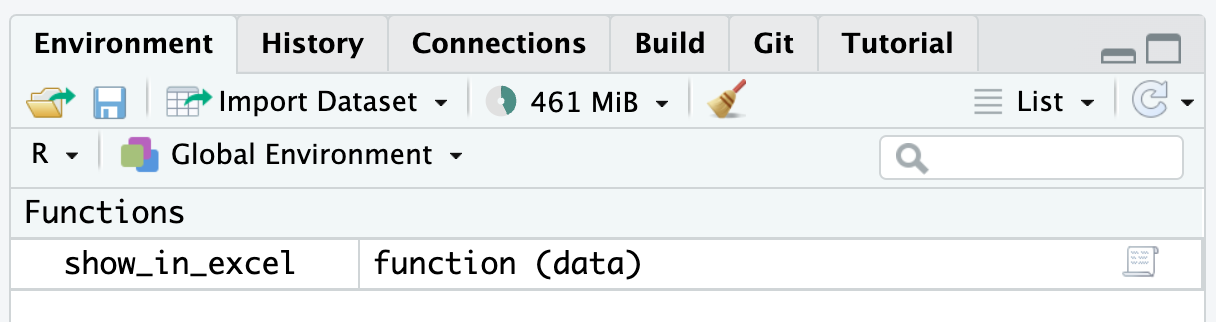
}

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 TODO.



* + - * 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 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",

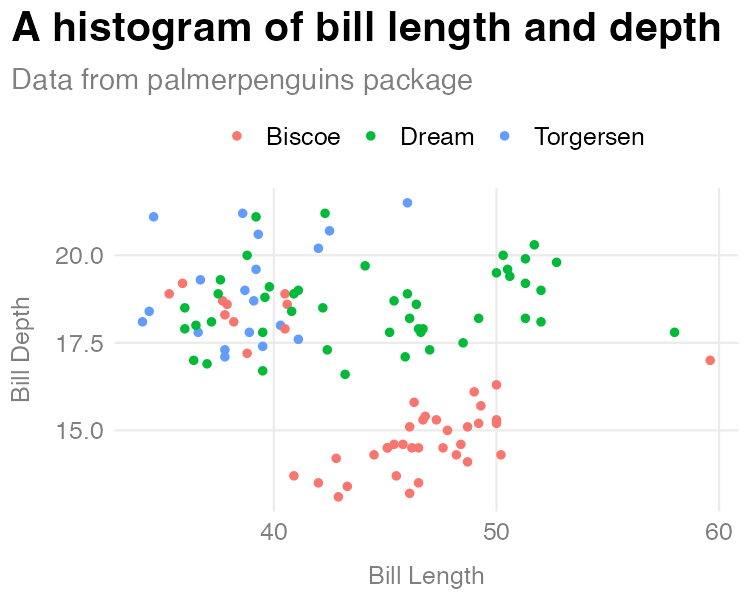
y = "Bill Depth",

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",

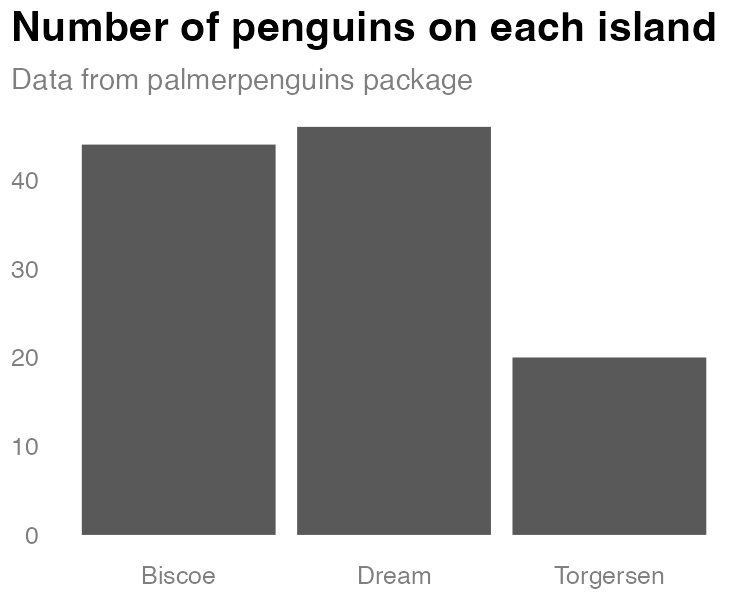
subtitle = "Data from palmerpenguins package") +

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.

Saving Functions in Packages

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.

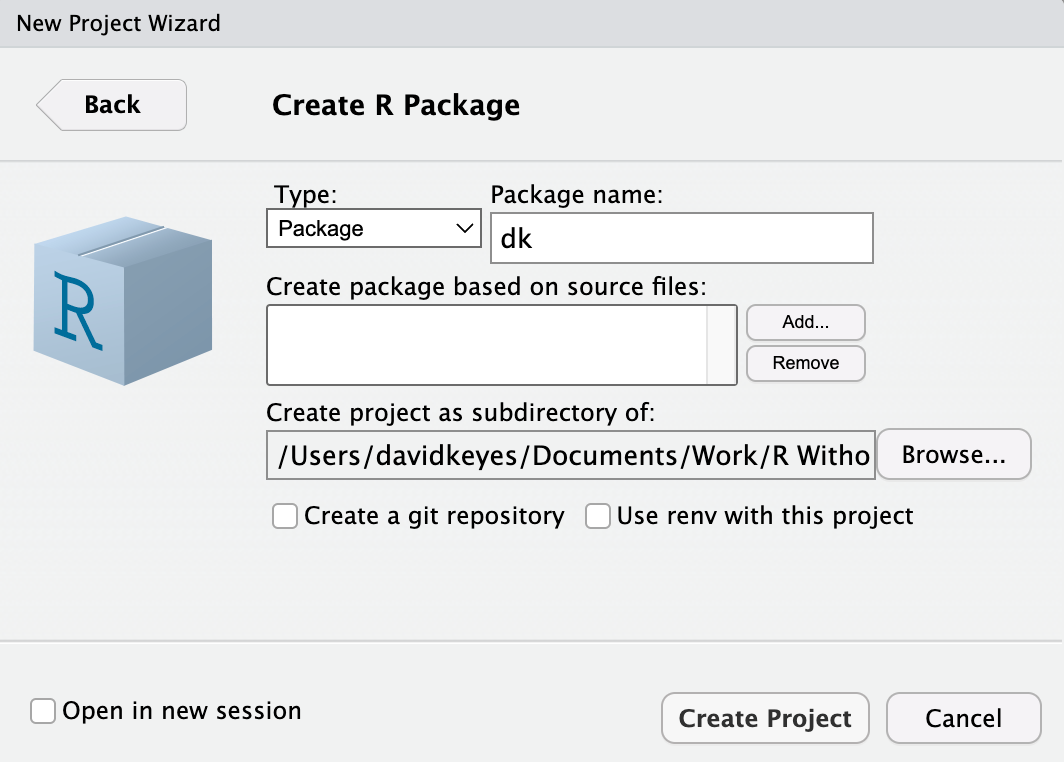
While you can run the functions from a functions.R file in your own environment, this code in may not work on someone else’s computer. Other users may not have the necessary packages installed, or they may be confused about how your functions’ arguments work and not know where to go for help. If you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

Creating a Package in RStudio

To create a package in RStudio, go to the File menu, then select New Project4New 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 11-3, I’ve called mine dk. Also 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 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:

XXXX

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

usethis::use\_r("theme")

The package::function() syntax makes it possible to use the function without loading the package. This function should create a file called theme.R in the R directory with the name you give it as an argument. Now you can open the file and add code to it. Copy the theme\_dk() function to the package.

Checking the Package with devtools

We need to change the theme\_dk() 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: 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.

1 ❯ 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, 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 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.

Adding Dependency Packages with the DESCRIPTION File

To ensure the package’s code will work, we need to install ggplot2 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 = "ggplot2")

You should 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 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:

ggplot2

Look for the imports way down at the bottom of the file. 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

The R CMD check output also included this line: “Refer to functions with ggplot2::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:

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 with Roxygen

Take a look at the “checking for missing documentation entries” warning. This warning tells us that we need to add documentation of the 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 enter ?theme\_mimimal() and see documentation about that function, we want them to be able to enter ?theme\_dk() to learn how it works.

To create documentation for theme\_dk(), 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 theme\_dk() function that looks like this:

#' Title

#'

#' @param show\_grid\_lines

#' @param show\_axis\_titles

#'

#' @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:

#' David Keyes's custom ggplot theme

Next are lines beginning with the text @param. Roxygen automatically locates the arguments in the function and creates a line for each one. Describe what each argument does as follows:

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

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

The @return line should tell the user what the dk\_theme() function returns. In our case, it returns 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 we’ll skip it here. 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, run devtools::document(). This will create a theme\_dk.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(theme\_dk)

The theme\_dk() 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 XXXXX. 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:

ggplot2

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, make sure you have the remotes package installed, then run the code remotes::install\_github("dgkeyes/dk"). XXXX

Once you or others install the dk package, anyone can add the line library(dk) in any R code then use the theme\_dk() function. For example, 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()

The resulting 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) to gain access to the theme\_dk() function. Another advantage is that if I were to make changes to theme\_dk(), users could simply reinstall the package to get access to the most up-to-date version of the function.

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.