R package workshop

Contents

Preface

About this workshop

This workshop was created by COMBINE, an association for Australian students in bioinformatics, computational biology and related fields. You can find out more about COMBINE at http://combine.org.au.

The goal of this workshop is to explain the basics of R package development. By the end of the workshop you should have your own minimal R package that you can use to store your personal functions.

The materials were written using the **bookdown** package (https://bookdown.org/home/), which is built on top of R Markdown and **knitr**.

Requirements

The workshop assumes that you are familar with basic R and the RStudio IDE. This includes topics such as installing packages, assigning variables and writing functions. If you are not comfortable with these you may need to complete an introductory R workshop first.

R and RStudio

You will need a recent version of R and RStudio. These materials were written using R version 3.6.0 (2019-04-26) and RStudio version 1.2.1335. You can download R from https://cloud.r-project.org/ and RStudio from https://www.rstudio.com/products/rstudio/download/.

Packages

The main packages used in the workshop are below with the versions used in these materials:

- **devtools** (v2.0.2)
- **usethis** (v1.5.1)
- **roxygen2** (v6.1.1)
- **testthat** (v2.1.1)
- **knitr** (v1.23)
- **ggplot2** (v3.2.0)
- **rlang** (v0.4.0)

Please make sure these packages are installed before starting the workshop. You can install them by running the following code.

```
pkgs <- c("devtools", "usethis", "roxygen2", "testhat", "knitr", "ggplot2",
          "rlang")
install.packages(pkgs)
```

GitHub

Version control using git is very useful and should be part of your package development process but it is outside the scope of this workshop. However, uploading your package to code sharing websites such as GitHub is the easiest way to distribute it. Towards the end of the workshop is a section showing you to upload your package to GitHub using R commands (no knowledge of git necessary). If you would like to try this and don't already have a GitHub account please create one at https://github.com/join.

License

These materials are covered by the Creative Commons Attribution 4.0 International (CC BY 4.0) license (https://creativecommons.org/licenses/by/4.0/).

Chapter 1

Introduction

1.1 What is a package?

An R package is a collection of functions that are bundled together in a way that lets them be easily shared. Usually these functions are designed to work together to complete a specific task such as analysing a particular kind of data. You are probably familiar with many packages already, for example **ggplot2** or **data.table**.

Packages can take various forms during their life cycle. For example the structure you use when writing package code is not exactly the same as what will be installed by somebody else. While you don't need to know about these forms in detail to create a package it is useful to be aware of them. For more details have a look at the "What is a package?" section of Hadley Wickham's "R packages" book (http://r-pkgs.had.co.nz/package.html#package).

1.2 Why write a package?

Packages are the best way to distribute code and documentation, and as we are about to find out they are very simple to make. Even if you never intend to share your package it is useful to have a place to store your commonly used functions. You may have heard the advice that if you find yourself reusing code then you should turn it into a function so that you don't have to keep rewriting it (along with other benefits). The same applies to functions. If you have some functions you reuse in different projects then it probably makes sense to put those in a package. It's a bit more effort now but it will save you a lot of time in the long run.

Of course often you will want to share your package, either to let other people use your functions or just so people can see what you have done (for example when you have code and data for a publication). If you are thinking about making a software package for public use there are a few things you should consider first:

- Is your idea new or is there already a package out there that does something similar?
- If there is does your package improve on it in some way? For example is it easier to use or does it have better performance?
- If a similar package exists could you help improve it rather than making a new one? Most package developers are open to collaboration and you may be able to achieve more by working together.

1.2.1 Packages for writing packages

This workshop teaches a modern package development workflow that makes use of packages designed to help with writing packages. The two main packages are **devtools** and **usethis**. As you might gather from the name **devtools** contains functions that will help with development tasks such as checking, building and installing packages. The **usethis** package contains a range of templates and handy functions for making life easier, many of which were originally in **devtools**[1]. All of the core parts of package development can be performed in other ways such as typing commands on the command line or clicking buttons in RStudio but we choose to use these packages because they provide a consistent workflow with sensible defaults. Other packages we will use that will be introduced in the appropriate sections are:

- **roxygen2** for function documentation
- **testthat** for writing unit tests
- **knitr** for building vignettes

[1]This is important to remember when looking at older tutorials or answers to questions on the internet. If `devtools::func()` doesn't seem to exist any more try `usethis::func()` instead

Chapter 2

Setting up

2.1 Open RStudio

The first thing we need to do is open RStudio. Do this now. If you currently have a project open close it by clicking *File > Close project*.

2.2 Naming your package

Before we create our package we need to give it a name. Package names can only consist of letters, numbers and dots (.) and must start with a letter. While all of these are allowed it is generally best to stick to just lowercase letters. Having a mix of lower and upper case letters can be hard for users to remember (is it **RColorBrewer** or **Rcolorbrewer** or **rcolorbrewer**?). Believe it or not choosing a name can be one of the hardest parts of making a package! There is a balance between choosing a name that is unique enough that it is easy to find (and doesn't already exist) and choosing something that makes it obvious what the package does. Acronyms or abbreviations are one option that often works well. It can be tricky to change the name of a package later so it is worth spending some time thinking about it before you start.

Checking availability

If there is even a small chance that your package might be used by other people it is worth checking that a package with your name doesn't already exist. A handy tool for doing this is the **available** package. This package will check common package repositories for your name as well as things like Urban Dictionary to make sure your name doesn't have some meanings you weren't aware of!

At the end of this workshop we want you to have a personal package that you can continue to add to and use so we suggest choosing a name that is specific to you. Something like your initials, a nickname or a username would be good options. For the example code we are going to use `mypkg` and you could use that for the workshop if you want to.

2.3 Creating your package

To create a template for our package we will use the `usethis::create_package()` function. All it needs is a path to the directory where we want to create the package. For the example we put it on the desktop but you should put it somewhere more sensible.

```r
usethis::create_package("~/Desktop/mypkg")
```

You will see some information printed to the console, something like (where USER is your username):

```
  Creating 'C:/Users/USER/Desktop/mypkg/'
  Setting active project to 'C:/Users/USER/Desktop/mypkg'
  Creating 'R/'
  Writing 'DESCRIPTION'
Package: mypkg
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (<https://orcid.org/YOUR-ORCID-ID
Description: What the package does (one paragraph).
License: What license it uses
Encoding: UTF-8
LazyData: true
  Writing 'NAMESPACE'
  Writing 'mypkg.Rproj'
  Adding '.Rproj.user' to '.gitignore'
  Adding '^mypkg\\.Rproj$', '^\\.Rproj\\.user$' to '.Rbuildignore'
  Opening 'C:/Users/USER/Desktop/mypkg/' in new RStudio session
  Setting active project to '<no active project>'
```

You will see something similar whenever we run a **usethis** command. Green ticks indicate that a step has been completed correctly. If you ever see a red dot that means that there is something **usethis** can't do for you and you will need to follow some instructions to do it manually. At the end a new RStudio window with your package should open. In this window you should see the following files:

- `DESCRIPTION` - The metadata file for your package. We will fill this in next and it will be updated as we develop our package.
- `NAMESPACE` - This file describes the functions in our package. Traditionally this has been a tricky file to get right but the modern development tools mean that we shouldn't need to edit it manually. If you open it you will see a message telling you not to.
- `R/` - This is the directory that will hold all our R code.

These files are the minimal amount that is required for a package but we will create other files as we go along. Some other useful files have also been created by **usethis**.

- `.gitignore` - This is useful if you use git for version control.
- `.Rbuildignore` - This file is used to mark files that are in the directory but aren't really part of the package and shouldn't be included when we build it. Most of the time you won't need to worry about this as **usethis** will edit it for you.
- `mypkg.Rproj` - The RStudio project file. Again you don't need to worry about this.

2.4 Filling in the DESCRIPTION

The `DESCRIPTION` file is one of the most important parts of a package. It contains all the metadata about the package, things like what the package is called, what version it is, a description, who the authors are, what other packages it depends on etc. Open the `DESCRIPTION` file and you should see something like this (with your package name).

```
Package: mypkg
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
    person(given = "First",
           family = "Last",
           role = c("aut", "cre"),
           email = "first.last@example.com",
           comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph).
License: What license it uses
Encoding: UTF-8
LazyData: true
```

2.4.1 Title and description

The package name is already set correctly but most of the other fields need to be updated. First let's update the title and description. The title should be a single line in Title Case that explains what your package is. The description is a paragraph which goes into a bit more detail. For example you could write something like this:

```
Package: mypkg
Title: My Personal Package
Version: 0.0.0.9000
Authors@R:
    person(given = "First",
           family = "Last",
           role = c("aut", "cre"),
           email = "first.last@example.com",
           comment = c(ORCID = "YOUR-ORCID-ID"))
Description: This is my personal package. It contains some handy functions that
    I find useful for my projects.
License: What license it uses
Encoding: UTF-8
LazyData: true
```

2.4.2 Authors

The next thing we will update is the Authors@R field. There are a couple of ways to define the author for a package but Authors@R is the most flexible. The example shows us how to define an author. You can see that the example person has been assigned the author ("aut") and creator ("cre") roles. There must be at least one author and one creator for every package (they can be the same person) and the creator must have an email address. There are many possible roles (including woodcutter ("wdc") and lyricist ("lyr")) but the most important ones are:

- cre: the creator or maintainer of the package, the person who should be contacted with there are problems
- aut: authors, people who have made significant contributions to the package
- ctb: contributors, people who have made smaller contributions
- cph: copyright holder, useful if this is someone other than the creator (such as their employer)

Adding an ORCID

> If you have an ORCID you can add it as a comment as shown in the example. Although not an official field this is recognised in various places (including CRAN) and is recommended if you want to get academic credit for your package (or have a common name that could be confused with other package authors).

Update the author information with your details. If you need to add another author simply concatenate them using `c()` like you would with a normal vector.

```
Package: mypkg
Title: My Personal Package
Version: 0.0.0.9000
Authors@R: c(
    person(given = "Package",
           family = "Creator",
           role = c("aut", "cre"),
           email = "package.creator@mypkg.com"),
    person(given = "Package",
           family = "Contributor",
           role = c("ctb"),
           email = "package.contributor@mypkg.com")
    )
Description: This is my personal package. It contains some handy functions that
    I find useful for my projects.
License: What license it uses
Encoding: UTF-8
LazyData: true
```

2.4.3 License

The last thing we will update now is the software license. The describes how our code can be used and without one people must assume that it can't be used at all! It is good to be as open and free as you can with your license to make sure your code is as useful to the community as possible. For this example we will use the MIT license which basically says the code can be used for any purpose and doesn't come with any warranties. There are templates for some of the most common licenses included in **usethis**.

```r
usethis::use_mit_license("Your Name")
```

This will update the license field.

```
Package: mypkg
Title: My Personal Package
```

```
Version: 0.0.0.9000
Authors@R: c(
    person(given = "Package",
           family = "Creator",
           role = c("aut", "cre"),
           email = "package.creator@mypkg.com"),
    person(given = "Package",
           family = "Contributor",
           role = c("ctb"),
           email = "package.contributor@mypkg.com")
    )
Description: This is my personal package. It contains some handy functions that
    I find useful for my projects.
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true
```

It will also also create two new files, `LICENSE.md` which contains the text of the MIT license (it's very short if you want to give it a read) and `LICENSE` which simply contains:

```
YEAR: 2019
COPYRIGHT HOLDER: Your Name
```

There are various other licenses you can use but make sure you choose one designed for software not other kinds of content. For example the Creative Commons licenses are great for writing or images but aren't designed for code. For more information about different licenses and what they cover have a look at http://choosealicense.com/ or https://tldrlegal.com/. For a good discussion about why it is important to declare a license read this blog post by Jeff Attwood http://blog.codinghorror.com/pick-a-license-any-license/.

Chapter 3

Functions

3.1 Adding a function

Now that our package is all set up it's time to add our first function! We can use the `usethis::use_r()` function to set up the file. Our function is going to be about colours so we will use that as the name of the R file.

```
usethis::use_r("colours")
```

Organising your code

There are no rules about how to organise your functions into different files but you want generally want to group similar functions into a file with a a clear name. Having all of your functions in a single file isn't great, but neither is having a separate file for each function. A good rule of thumb is that if you are finding it hard to locate a function you might need to move it to a new file. There are two shortcuts for finding functions in RStudio, selecting a function name and pressing **F2** or pressing **Ctrl + .** and searching for the function.

As an example we are going to write a function that takes the red, green and blue values for a colour and returns a given number of shades. Copy the following code into your R file and save it (you can ignore the comments if you want to, they are just there to explain how the function works).

```
make_shades <- function(red, green, blue, n, lighter = TRUE) {
    # Convert the colour to RGB
    colour_rgb <- grDevices::col2rgb(colour)[, 1]
    # Decide if we are heading towards white or black
    if (lighter) {
        end <- 255
    } else {
        end <- 0
    }

    # Calculate the red, green and blue for the shades
    # we calculate one extra point to avoid pure white/black
    red <- seq(colour_rgb[1], end, length.out = n + 1)[1:n]
    green <- seq(colour_rgb[2], end, length.out = n + 1)[1:n]
    blue <- seq(colour_rgb[3], end, length.out = n + 1)[1:n]

    # Convert the RGB values to hex codes
    shades <- grDevices::rgb(red, green, blue, maxColorValue = 255)

    return(shades)
}
```

3.2 Using the function

Now that we have a function we want to see if it works. Usually when we write a new function we load it by copying the code to the console or sourcing the R file. When we are developing a package we want to try and keep our environment empty so that we can be sure we are only working with objects inside the package. Instead we can load functions using `devtools::load_all()`.

```
devtools::load_all()
```

The function doesn't appear in the environment, just like all the functions in a package don't appear in the environment when we load it using `library()`. But if we try to use it the function should work.

```
make_shades("goldenrod", 5)
```

Congratulations, you now have a functional package! In the next section we will perform some checks to see if we have forgotten anything.

Chapter 4

Checking you package

Although what is absolutely required for a package is fairly minimal there are a range of things that are needed for a package to be considerd "correct". Keeping track of all of these can be difficult but luckily the `devtools::check()` function is here to help! This function runs a series of checks developed by some very smart people over a long period of time that are designed to make sure your package is working correctly. It is highly recommended that you run `devtools::check()` often and follow it's advice to fix any problems. It's much easier to fix one or two problems when they first come up than to try many at once after you have moved on to other things. Let's run the checks on our package and see what we get.

```r
devtools::check()
```

```
-- Building ----------------------------------------------------------- mypkg --
Setting env vars:
* CFLAGS    : -Wall -pedantic
* CXXFLAGS  : -Wall -pedantic
* CXX11FLAGS: -Wall -pedantic
--------------------------------------------------------------------------------
√  checking for file 'C:\Users\USER\Desktop\mypkg/DESCRIPTION' (3.1s)
-  preparing 'mypkg':
√  checking DESCRIPTION meta-information ...
-  checking for LF line-endings in source and make files and shell scripts
-  checking for empty or unneeded directories
-  building 'mypkg_0.0.0.9000.tar.gz'

-- Checking ----------------------------------------------------------- mypkg --
Setting env vars:
* _R_CHECK_CRAN_INCOMING_REMOTE_: FALSE
* _R_CHECK_CRAN_INCOMING_       : FALSE
* _R_CHECK_FORCE_SUGGESTS_      : FALSE
-- R CMD check -----------------------------------------------------------------------
-  using log directory 'C:/Users/USER/AppData/Local/Temp/Rtmp8eH30T/mypkg.Rcheck' (2.3s
-  using R version 3.6.0 (2019-04-26)
-  using platform: x86_64-w64-mingw32 (64-bit)
-  using session charset: ISO8859-1
-  using options '--no-manual --as-cran'
√  checking for file 'mypkg/DESCRIPTION' ...
-  this is package 'mypkg' version '0.0.0.9000'
-  package encoding: UTF-8
√  checking package namespace information ...
√  checking package dependencies (1s)
√  checking if this is a source package ...
√  checking if there is a namespace
√  checking for .dll and .exe files
√  checking for hidden files and directories ...
√  checking for portable file names ...
√  checking serialization versions ...
√  checking whether package 'mypkg' can be installed (1.4s)
√  checking package directory
√  checking for future file timestamps (815ms)
√  checking DESCRIPTION meta-information (353ms)
√  checking top-level files ...
√  checking for left-over files
√  checking index information
√  checking package subdirectories ...
√  checking R files for non-ASCII characters ...
√  checking R files for syntax errors ...
√  checking whether the package can be loaded ...
√  checking whether the package can be loaded with stated dependencies ...
√  checking whether the package can be unloaded cleanly ...
√  checking whether the namespace can be loaded with stated dependencies ...
√  checking whether the namespace can be unloaded cleanly ...
√  checking loading without being on the library search path ...
√  checking dependencies in R code ...
√  checking S3 generic/method consistency (410ms)
√  checking replacement functions ...
√  checking foreign function calls ...
√  checking R code for possible problems (2.2s)
W  checking for missing documentation entries ...
   Undocumented code objects:
     'make_shades'
   All user-level objects in a package should have documentation entries.
   See chapter 'Writing R documentation files' in the 'Writing R
   Extensions' manual.
```

```
-  checking examples ... NONE (956ms)

   See
     'C:/Users/USER/AppData/Local/Temp/Rtmp8eH30T/mypkg.Rcheck/00check.log'
   for details.


-- R CMD check results ------------------------------------------ mypkg 0.0.0.9000 ----
Duration: 12.3s

> checking for missing documentation entries ... WARNING
  Undocumented code objects:
    'make_shades'
  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 √ | 1 warning x | 0 notes √
```

You can see all the different types of checks that **devtools** has run but they most important section is at the end where it tells you how many errors, warnings and notes there are. Errors happen when you code has broken and failed one of the checks. If errors are not fixed your package will not work correctly. Warnings are slightly less serious but should also be addressed. Your package will probably work without fixing thise but it is highly advised that you do. Notes are advice rather than problems. It can be up to you whether or not to address them but there is usally a good reason to. Often the failed checks come with hints about how to fix them but sometimes they can be hard to understand. If you are not sure what they mean try doing an internet search and it is likely that somebody else has come across the same problem. Our package has received one warning telling us that we are missing some documentation.

Chapter 5

Documenting functions

The output of our check tells us that we are missing documentation for the `make_shades` function. Writing this kind of documentation is another part of package development that has been made much easier by modern packages, in this case one called **roxygen2**. R help files use a complicated syntax similar to LaTeX that can be easy to mess up. Instead of writing this all ourselves using Roxygen lets us just write some special comments at the start of each function. This has the extra advantage of keeping the documentation with the code which make it easier to keep it up to date.

5.1 Adding documentation

To insert a documentation skeleton in RStudio click inside the `make_shades` function then open the *Code* menu and select *Insert Roxygen skeleton* or use **Ctrl** + **Alt** + **Shift** + **R**. The inserted code looks like this:

```
#' Title
#'
#' @param colour
#' @param n
#' @param lighter
#'
#' @return
#' @export
#'
#' @examples
```

Roxygen comments all start with `#'`. The first line is the title of the function then there is a blank line. Following that there can be a paragraph giving a more detailed description of the function. Let's fill those in to start with.

```
#' Make shades
#'
#' Given a colour make n lighter or darker shades
#'
#' @param colour
#' @param n
#' @param lighter
#'
#' @return
#' @export
#'
#' @examples
```

The next section describes the parameters (or arguments) for the function marked by the `@param` field. RStudio has helpfully filled in names of these for us but we need to provide a description.

```
#' Make shades
#'
#' Given a colour make n lighter or darker shades
#'
#' @param colour The colour to make shades of
#' @param n The number of shades to make
#' @param lighter Whether to make lighter (TRUE) or darker (FALSE) shades
#'
#' @return
#' @export
#'
#' @examples
```

The next field is `@return`. This is where we describe what the function returns. This is usually fairly short but you should provide enough detail to make sure that the user knows what they are getting back.

```
#' Make shades
#'
#' Given a colour make n lighter or darker shades
#'
#' @param colour The colour to make shades of
#' @param n The number of shades to make
#' @param lighter Whether to make lighter (TRUE) or darker (FALSE) shades
#'
```

```
#' @return A vector of n colour hex codes
#' @export
#'
#' @examples
```

After `@return` we have `@export`. This field is a bit different because it doesn't add documentation to the help file, instead it modifies the `NAMESPACE` file. Adding `@export` tells Roxygen that this is a function that we want to be available to the user. When we build the documentation Roxygen will then add the correct information to the `NAMESPACE` file. If we had an internal function that wasn't meant to be used by the user we would leave out `@export`.

The last field in the skeleton is `@examples`. This is where we put some short examples showing how the function can be used. These will be placed in the help file and can be run using `example("function")`. Let's add a couple of examples. If you want to add a comment to an example you need to add another `#`.

```
#' Make shades
#'
#' Given a colour make n lighter or darker shades
#'
#' @param colour The colour to make shades of
#' @param n The number of shades to make
#' @param lighter Whether to make lighter (TRUE) or darker (FALSE) shades
#'
#' @return A vector of n colour hex codes
#' @export
#'
#' @examples
#' # Five lighter shades
#' make_shades("goldenrod", 5)
#' # Five darker shades
#' make_shades("goldenrod", 5, lighter = FALSE)
```

Other fields

In this example we only fill in the fields in the skeleton but there are many other useful fields. For example `@author` (specify the function author), `@references` (any associated references) and `@seealso` (links to related functions).

5.2 Building documentation

Now we can build our documentation using **devtools**.

```
devtools::document()
```

```
Updating mypkg documentation
Updating roxygen version in C:\Users\USER\Desktop\mypkg/DESCRIPTION
Writing NAMESPACE
Loading mypkg
Writing NAMESPACE
Writing make_shades.Rd
```

The output shows us that **devtools** has done a few things. Firstly it has set the version of **roxygen2** we are using in the `DESCRIPTION` file by adding this line:

```
RoxygenNote: 6.1.1
```

Next it has updated the `NAMESPACE` file. If you open it you will see:

```
# Generated by roxygen2: do not edit by hand

export(make_shades)
```

Which tells us that the `make_shades` function is exported.

The last thing it has done is create a new file called `make_shades.Rd` in the `man/` directory (which will be created if it doesn't exist). The `.Rd` extension stands for "R documentation" and this is what is turned into a help file when the package is installed. Open the file and see what it looks like.

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/colours.R
\name{make_shades}
\alias{make_shades}
\title{Make shades}
\usage{
make_shades(colour, n, lighter = TRUE)
}
\arguments{
\item{colour}{The colour to make shades of}

\item{n}{The number of shades to make}

\item{lighter}{Whether to make lighter (TRUE) or darker (FALSE) shades}
}
\value{
```

```
A vector of n colour hex codes
}
\description{
Given a colour make n lighter or darker shades
}
\examples{
# Five lighter shades
make_shades("goldenrod", 5)
# Five darker shades
make_shades("goldenrod", 5, lighter = FALSE)
}
```

Hopefully you can see why we want to avoid writing this manually! This is only a simple function but already the help file is quite complicated with lots of braces. To see what the rendered documentation looks like just run `?make_shades`.

5.3 Formatting documentation

The rendered output already looks pretty good but we might want to add some extra formatting to it to make it a bit clearer. As we have seen above there is a special syntax for different kinds of formatting. For example we can mark code in the documentation using `\code{}`.

```
#' Make shades
#'
#' Given a colour make \code{n} lighter or darker shades
#'
#' @param colour The colour to make shades of
#' @param n The number of shades to make
#' @param lighter Whether to make lighter (\code{TRUE}) or darker (\code{FALSE})
#' shades
#'
#' @return A vector of \code{n} colour hex codes
#' @export
#'
#' @examples
#' # Five lighter shades
#' make_shades("goldenrod", 5)
#' # Five darker shades
#' make_shades("goldenrod", 5, lighter = FALSE)
```

Run `devtools::document()` again and see what has changed in the rendered file. There are many other kinds of formatting we could use, for example:

`\code{}`, `\eqn{}`, `\emph{}`, `\strong{}`, `\itemize{}`, `\enumerate{}`, `\link{}`, `\link[]{}`, `\url{}`, `\href{}{}`, `\email{}`.

Using Markdown

If you are familiar with Markdown you may prefer to use it for writing documentation. Luckily Roxygen has a Markdown mode that can be activated using `usethis::use_roxygen_md()`. See the Roxygen Markdown vignette for more details https://cran.r-project.org/web/packages/roxygen2/vignettes/markdown.html.

Chapter 6

Testing

Now that we have some documentation `devtools::check()` should run without any problems.

```
devtools::check()
```

```
-- R CMD check results ------------------------------------------ mypkg 0.0.0.9000 ----
Duration: 15.2s

0 errors ✓ | 0 warnings ✓ | 0 notes ✓
```

(This is just the bottom part of the output to save space)

While we pass all the standard package checks there is one kind of check that we don't have yet. Unit tests are checks to make sure that a function works in the way that we expect. The examples we wrote earlier are kind of like informal unit tests because they are run as part of the checking process but it is better to have something more rigorous. One approach to writing unit tests is what is known as "test driven development". The idea here is to write the tests before you write a function. This way you know exactly what a function is supposed to do and what problems there might be. While this is a good principal it can take a lot of advance planning. A more common approach could be called "bug-driven testing". For this approach whenever we come across a bug we write a test for it before we fix it, that way the same bug should never happen a again. When combined with some tests for obvious problems this is a good compromise better testing for every possible outcome and not testing at all. For example let's see what happens when we ask `make_shades()` for a negative number of shades.

```r
make_shades("goldenrod", -1)
```

```
Error in seq(colour_rgb[1], end, length.out = n + 1)[1:n] :
  only 0's may be mixed with negative subscripts
```

This doesn't make sense so we expect to get an error but it would be useful if the error message was more informative. What if we ask for zero shades?

```r
make_shades("goldenrod", 0)
```

```
[1] "#DAA520"
```

That does work, but it probably shouldn't. Before we make any changes to the function let's design some tests to make sure we get what we expect. There are a few ways to write unit tests for R packages but we are going to use the **testthat** package. We can set everything up with **usethis**.

```r
usethis::use_testthat()
```

```
  Adding 'testthat' to Suggests field in DESCRIPTION
  Creating 'tests/testthat/'
  Writing 'tests/testthat.R'
  Call `use_test()` to initialize a basic test file and open it for editing.
```

Now we have a `tests/` directory to hold all our tests. There is also a `tests/testthat.R` file which looks like this:

```r
library(testthat)
library(mypkg)

test_check("mypkg")
```

All this does is make sure that our tests are run when we do `devtools::check()`. To open a new test file we can use `usethis::use_test()`.

```r
usethis::use_test("colours")
```

```
  Increasing 'testthat' version to '>= 2.1.0' in DESCRIPTION
  Writing 'tests/testthat/test-colours.R'
  Modify 'tests/testthat/test-colours.R'
```

Just like R files our test file needs a name. Tests can be split up however you like but it often makes sense to have them match up with the R files so things are easy to find. Our test file comes with a small example that shows how to use **testthat**.

```r
test_that("multiplication works", {
  expect_equal(2 * 2, 4)
})
```

Each set of tests starts with the `test_that()` function. This function has two arguments, a description and the code with the tests that we want to run. It looks a bit strange to start with but it makes sense if you read it as a sentence, "Test that multiplication work". That makes it clear what the test is for. Inside the code section we see an `expect` function. This function also has two parts, the thing we want to test and what we expect it to be. There are different functions for different types of expectations. Reading this part as a sentence says something like "Expect that 2 * 2 is equal to 4". For our test we want to use the `expect_error()` function, because that is what we expect.

```r
test_that("n is at least 1", {
    expect_error(make_shades("goldenrod", -1),
                 "n must be at least 1")
    expect_error(make_shades("goldenrod", 0),
                 "n must be at least 1")
})
```

To run our tests we use `devtools::test()`.

```r
devtools::test()
```

```
Loading mypkg
Testing mypkg
✓ |  OK F W S | Context
x |   0 2     | colours
────────────────────────────────────────────────────────────────────────────────
test-colours.R:2: failure: n is at least 1
`make_shades("goldenrod", -1)` threw an error with unexpected message.
Expected match: "n must be at least 1"
Actual message: "only 0's may be mixed with negative subscripts"

test-colours.R:4: failure: n is at least 1
`make_shades("goldenrod", 0)` did not throw an error.
────────────────────────────────────────────────────────────────────────────────
```

```
== Results =====================================================================
OK:       0
Failed:   2
Warnings: 0
Skipped:  0

No one is perfect!
```

We can see that both of our tests failed. That is ok because we haven't fixed the function yet. The first test fails because the error message is wrong and the second one because there is no error. Now that we have some tests and we know they check the right things we can modify our function to check the value of `n` and give the correct error.

Let's add some code to check the value of `n`. We will update the documentation as well so the user knows what values can be used.

```r
#' Make shades
#'
#' Given a colour make \code{n} lighter or darker shades
#'
#' @param colour The colour to make shades of
#' @param n The number of shades to make, at least 1
#' @param lighter Whether to make lighter (\code{TRUE}) or darker (\code{FALSE})
#' shades
#'
#' @return A vector of \code{n} colour hex codes
#' @export
#'
#' @examples
#' # Five lighter shades
#' make_shades("goldenrod", 5)
#' # Five darker shades
#' make_shades("goldenrod", 5, lighter = FALSE)
make_shades <- function(colour, n, lighter = TRUE) {

    # Check the value of n
    if (n < 1) {
        stop("n must be at least 1")
    }

    # Convert the colour to RGB
    colour_rgb <- grDevices::col2rgb(colour)[, 1]

    # Decide if we are heading towards white or black
    if (lighter) {
        end <- 255
    } else {
        end <- 0
    }

    # Calculate the red, green and blue for the shades
    # we calculate one extra point to avoid pure white/black
    red <- seq(colour_rgb[1], end, length.out = n + 1)[1:n]
    green <- seq(colour_rgb[2], end, length.out = n + 1)[1:n]
    blue <- seq(colour_rgb[3], end, length.out = n + 1)[1:n]

    # Convert the RGB values to hex codes
    shades <- grDevices::rgb(red, green, blue, maxColorValue = 255)

    return(shades)
}
```

Writing parameter checks

These kinds of checks for parameter inputs are an important part of a function that is going to be used by other people (or future you). They make sure that all the input is correct before the function tries to do anything and avoids confusing error messages. However they can be fiddly and repetitive to write. If you find yourself writing lots of these checks two packages that can make life easier by providing functions to do it for you are **checkmate** and **assertthat**.

Here we have used the `stop()` function to raise an error. If we wanted to give a warning we would use `warning()` and if just wanted to give some information to the user we would use `message()`. Using `message()` instead of `print()` or `cat()` is important because it means the user can hide the messages using `suppressMessages()` (or `suppressWarnings()` for warnings). Now we can try our tests again and they should pass.

```
devtools::test()
```

```
Loading mypkg
Testing mypkg
√ |  OK F W S | Context
√ |   2       | colours

== Results =====================================================================
OK:       2
Failed:   0
Warnings: 0
Skipped:  0
```

There are more tests we could write for this function but we will leave that as an exercise for you. If you want to see what parts of your code need testing you can run the `devtools::test_coverage()` function (you might need to install the **DT** package first). This function uses the **covr** package to make a report showing which lines of your code are covered by tests.

Chapter 7

Dependencies

Our `make_shades()` function produces shades of a colour but it would be good to see what those look like. Below is a new function called `plot_colours()` that can visualise them for us using **ggplot2** (if you don't have **ggplot2** installed do that now). Add this function to `colours.R`.

```r
#' Plot colours
#'
#' Plot a vector of colours to see what they look like
#'
#' @param colours Vector of colour to plot
#'
#' @return A ggplot2 object
#' @export
#'
#' @examples
#' shades <- make_shades("goldenrod", 5)
#' plot_colours(shades)
plot_colours <- function(colours) {
    plot_data <- data.frame(Colour = colours)

    ggplot(plot_data,
           aes(x = .data$Colour, y = 1, fill = .data$Colour,
               label = .data$Colour)) +
        geom_tile() +
        geom_text(angle = "90") +
        scale_fill_identity() +
        theme_void()
}
```

Now that we have added something new we should run our checks again (`devtools::document()` is automatically run as part of `devtools::check()` so we can skip that step).

```
devtools::check()
```

```
-- R CMD check results ------------------------------------------ mypkg 0.0.0.9000 --
Duration: 15.4s

> checking examples ... ERROR
  Running examples in 'mypkg-Ex.R' failed
  The error most likely occurred in:

  > base::assign(".ptime", proc.time(), pos = "CheckExEnv")
  > ### Name: plot_colours
  > ### Title: Plot colours
  > ### Aliases: plot_colours
  >
  > ### ** Examples
  >
  > shades <- make_shades("goldenrod", 5)
  > plot_colours(shades)
  Error in ggplot(plot_data, aes(x = .data$Colour, y = 1, fill = .data$Colour,  :
    could not find function "ggplot"
  Calls: plot_colours
  Execution halted

> checking R code for possible problems ... NOTE
  plot_colours: no visible global function definition for 'ggplot'
  plot_colours: no visible global function definition for 'aes'
  plot_colours: no visible binding for global variable '.data'
  plot_colours: no visible global function definition for 'geom_tile'
  plot_colours: no visible global function definition for 'geom_text'
  plot_colours: no visible global function definition for
    'scale_fill_identity'
  plot_colours: no visible global function definition for 'theme_void'
  Undefined global functions or variables:
    .data aes geom_text geom_tile ggplot scale_fill_identity theme_void

1 error x | 0 warnings ✓ | 1 note x
```

The checks have returned one error and one note. The error is more serious so let's have a look at that first. It says `could not find function "ggplot"`. Hmmmm…the `ggplot()` function is in the **ggplot2** package. When we used `col2rgb()` in the `make_shades()` function we had to prefix it with `grDevices::`, maybe we should do the same here.

```r
#' Plot colours
#'
#' Plot a vector of colours to see what they look like
#'
#' @param colours Vector of colour to plot
#'
#' @return A ggplot2 object
#' @export
#'
#' @examples
#' shades <- make_shades("goldenrod", 5)
#' plot_colours(shades)
plot_colours <- function(colours) {
    plot_data <- data.frame(Colour = colours)

    ggplot2::ggplot(plot_data,
                    ggplot2::aes(x = .data$Colour, y = 1, fill = .data$Colour,
                                 label = .data$Colour)) +
        ggplot2::geom_tile() +
        ggplot2::geom_text(angle = "90") +
        ggplot2::scale_fill_identity() +
        ggplot2::theme_void()
}
```

Now what do our checks say?

```r
devtools::check()
```

```
-- R CMD check results ------------------------------------------ mypkg 0.0.0.9000 ----
Duration: 15s

> checking examples ... ERROR
  Running examples in 'mypkg-Ex.R' failed
  The error most likely occurred in:

  > base::assign(".ptime", proc.time(), pos = "CheckExEnv")
  > ### Name: plot_colours
  > ### Title: Plot colours
  > ### Aliases: plot_colours
  >
  > ### ** Examples
  >
  > shades <- make_shades("goldenrod", 5)
  > plot_colours(shades)
  Error in loadNamespace(name) : there is no package called 'ggplot2'
  Calls: plot_colours ... loadNamespace -> withRestarts -> withOneRestart -> doWithOneR
  Execution halted

> checking dependencies in R code ... WARNING
  '::' or ':::' import not declared from: 'ggplot2'

> checking R code for possible problems ... NOTE
  plot_colours: no visible binding for global variable '.data'
  Undefined global functions or variables:
    .data

1 error x | 1 warning x | 1 note x
```

There is now one error, one warning and one note. That seems like we are going in the wrong direction but the error is from running the example and the warning gives us a clue to what the problem is. It says "'::' or ':::' import not declared from: 'ggplot2'". The important word here is "import". Just like when we export a function in our package we need to make it clear when we are using functions in another package. To do this we can use `usethis::use_package()`.

```r
usethis::use_package("ggplot2")
```

```
 Setting active project to 'C:/Users/Luke/Desktop/mypkg'
 Adding 'ggplot2' to Imports field in DESCRIPTION
 Refer to functions with `ggplot2::fun()`
```

The output tells us to refer to functions using "::" like we did above so we were on the right track. It also mentions that it has modified the `DESCRIPTION` file. Let's have a look at it now.

```
Package: mypkg
Title: My Personal Package
Version: 0.0.0.9000
Authors@R: c(
    person(given = "Package",
           family = "Creator",
           role = c("aut", "cre"),
           email = "package.creator@mypkg.com"),
    person(given = "Package",
           family = "Contributor",
           role = c("ctb"),
           email = "package.contributor@mypkg.com")
    )
Description: This is my personal package. It contains some handy functions that
```

```
    I find useful for my projects.
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true
RoxygenNote: 6.1.1
Suggests:
    testthat (>= 2.1.0)
Imports:
    ggplot2
```

The two lines at the bottom tell us that our package uses functions in **ggplot2**. There are three main types of dependencies[1]. Imports is the most common. This means that we use functions from these packages and they must be installed when our package is installed. The next most common is Suggests. These are packages that we use in developing our package (such as **testthat** which is already listed here) or packages that provide some additional, optional functionality. Suggested packages aren't usually installed so we need to do a check before we use them. The output of `usethis::use_package()` will give you an example if you add a suggested package. The third type of dependency is Depends. If you depend on a package it will be loaded whenever your package is loaded. There are some cases where you might need to do this but you should avoid Depends unless it is absolutely necessary.

Should you use a dependency?

Deciding which packages (and how many) to depend on is a difficult and philosophical choice. Using functions from other packages can save you time and effort in development but it might make it more difficult to maintain your package. Some things you might want to consider before depending on a package are:

- How much of the functionality of the package do you want to use?
- Could you easily reproduce that functionality?
- How well maintained is the package?
- How often is it updated? Packages that change a lot are more likely to break your code.
- How many dependencies of it's own does that package have?
- Are you users likely to have the package installed already?

Packages like **ggplot2** are good choices for dependencies because they are well maintained, don't change too often, are commonly used and perform a single task so you are likely to use many of the functions.

[1]There is a fourth kind (Enhances) but that is almost never used.

Hopefully now that we have imported **ggplot2** we should pass the checks.

```
devtools::check()
```

```
-- R CMD check results ---------------------------------------- mypkg 0.0.0.9000 ---
Duration: 16.4s

> checking R code for possible problems ... NOTE
  plot_colours: no visible binding for global variable '.data'
  Undefined global functions or variables:
    .data

0 errors √ | 0 warnings √ | 1 note x
```

Success! Now all that's left is that pesky note. Visualisation functions are probably some of the most common functions in packages but there are some tricks to programming with **ggplot2**. The details are outside the scope of this workshop but if you are interested see the "Using ggplot2 in packages" vignette https://ggplot2.tidyverse.org/dev/articles/ggplot2-in-packages.html.

To solve our problem we need to import the **rlang** package.

```
usethis::use_package("rlang")
```

```
 Adding 'rlang' to Imports field in DESCRIPTION
 Refer to functions with `rlang::fun()`
```

Writing `rlang::.data` wouldn't be very attractive or readable[2]. When we want to use a function in another package with `::` we need to exlicitly import it. Just like when we exported our functions we do this using a Roxygen comment.

```
#' Plot colours
#'
#' Plot a vector of colours to see what they look like
#'
#' @param colours Vector of colour to plot
#'
#' @return A ggplot2 object
#' @export
#'
#' @importFrom rlang .data
#'
#' @examples
```

[2]Also for technical reasons it won't work in this case.

```r
#' shades <- make_shades("goldenrod", 5)
#' plot_colours(shades)
plot_colours <- function(colours) {
    plot_data <- data.frame(Colour = colours)

    ggplot2::ggplot(plot_data,
                    ggplot2::aes(x = .data$Colour, y = 1, fill = .data$Colour,
                                 label = .data$Colour)) +
        ggplot2::geom_tile() +
        ggplot2::geom_text(angle = "90") +
        ggplot2::scale_fill_identity() +
        ggplot2::theme_void()
}
```

When we use `devtools::document()` this comment will be read and a note placed in the `NAMESPACE` file, just like for `@export`.

```
# Generated by roxygen2: do not edit by hand

export(make_shades)
export(plot_colours)
importFrom(rlang,.data)
```

Those two steps should fix our note.

```r
devtools::check()
```

```
-- R CMD check results ------------------------------------------ mypkg 0.0.0.9000 ----
Duration: 16.8s

0 errors √ | 0 warnings √ | 0 notes √
```

If we used `rlang::.data` in multiple functions in our pacakge it might make sense to only import it once. It doesn't matter where we put the `@importFrom` line (or how many times) it will still be added to `NAMESPACE`. This means we can put all import in a central location. The advantage of this is that they only appear once and are all in one place but it makes it harder to know which of our functions have which imports and remove them if they are no longer needed. Which approach you take is up to you.

We should write some tests for this function as well but we will leave that as an exercise for you to try later.

Chapter 8

Other documentation

In a previous section we documented our functions using Roxygen comments but there are a few other kinds of documentation we should have.

8.1 Package help file

Users can find out about our functions using `?function-name` but what if they want to find out about the package itself? There is some information in the `DESCRIPTION` but that can be hard to access. Let's add a help file for the pacakge.

```
usethis::use_package_doc()
```

```
  Writing 'R/mypkg-package.R'
```

This creates a special R file for us called `mypkg-package.R`. The contents of this file doesn't look like much it is understood by **devtools** and **roxygen2**.

```
#' @keywords internal
"_PACKAGE"

# The following block is used by usethis to automatically manage
# roxygen namespace tags. Modify with care!
## usethis namespace: start
## usethis namespace: end
NULL
```

Run `devtools::document()`.

```r
devtools::document()
```

```
Updating mypkg documentation
Writing NAMESPACE
Loading mypkg
Writing NAMESPACE
Writing mypkg-package.Rd
```

We can see that a new `.Rd` file has been created and we can view the contents using `?mypkg`. The information here has been automatically pulled from the `DESCRIPTION` file so we only need to update it in one place.

8.2 Vignettes

The documentation we have written so far explains how individual functions work in detail but it doesn't show what the package does as a whole. Vignettes are short tutorials that explain what the package is designed for and how different functions can be used together. There are different ways to write vignettes but usually they are R Markdown files. We can create a vignette with `usethis::use_vignette()`. There can be multiple vignettes but it is common practice to start with one that introduces the whole package.

What is R Markdown?

Markdown is a simple markup language that makes it possible to write documents with minimal formatting. See *Help > Markdown Quick Reference* in RStudio for a quick guide to how this formatting works. R Markdown adds chunks of R code that are run and the output included in the final document.

```r
usethis::use_vignette("mypkg")
```

```
 Adding 'knitr' to Suggests field in DESCRIPTION
 Setting VignetteBuilder field in DESCRIPTION to 'knitr'
 Adding 'inst/doc' to '.gitignore'
 Creating 'vignettes/'
 Adding '*.html', '*.R' to 'vignettes/.gitignore'
 Adding 'rmarkdown' to Suggests field in DESCRIPTION
 Writing 'vignettes/mypkg.Rmd'
 Modify 'vignettes/mypkg.Rmd'
```

Because this is our first vignette **usethis** has added some information to the `DESCRIPTION` file including adding the **knitr** package as a suggested dependency. It also creates a `vignettes/` directory and opens our new `mypkg.Rmd` file.

~~~
---
title: "mypkg"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{mypkg}
  %\VignetteEngine{knitr::rmarkdown}
  %\VignetteEncoding{UTF-8}
---

```{r, include = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>"
)
```

```{r setup}
library(mypkg)
```
~~~

If you are familiar with R Markdown you might note some unusual content in the header. This is important for the vignette to build properly. There are also some **knitr** options set which are the convention for vignettes.

Let's add a short example of how to use our package.

~~~
---
title: "mypkg"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{mypkg}
  %\VignetteEngine{knitr::rmarkdown}
  %\VignetteEncoding{UTF-8}
---

```{r, include = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>"
)
```

```{r setup}
library(mypkg)
```
~~~

~~~
# Introduction

This is my personal package. It contains some handy functions that I find useful
for my projects.

# Colours

Sometimes you want to generate shades of a colour. The `make_shades()` function
makes this easy!

```{r}
shades <- make_shades("goldenrod", 5)
```

If you want to see what the shades look like you can plot them using
`plot_colours()`.

```{r}
plot_colours(shades)
```

This function is also useful for viewing any other palettes.

```{r}
plot_colours(rainbow(5))
```
~~~

To see what the vignette looks like run `devtools::build_vignettes()`. Asking **devtools** to build the vignette rather than rendering it in another way (such as the *Knit* button in RStudio) makes sure that we are using the development version of the package rather than any version that is installed.

```r
devtools::build_vignettes()
```

```
Building mypkg vignettes
--- re-building 'mypkg.Rmd' using rmarkdown
--- finished re-building 'mypkg.Rmd'

Moving mypkg.html, mypkg.R to doc/
Copying mypkg.Rmd to doc/
Building vignette index
```

This creates a new directory called `doc/` that contains the rendered vignette. Click on the `mypkg.html` file and open it in your browser.

If you want to use any other packages in your vignette that the package doesn't already depend on you need to add them as a suggested dependency.

8.3 README

If you plan on sharing the source code rather than the built package it is useful to have a README file to explain what the package is, how to install and use it, how to contribute etc. We can create a template with `usethis::use_readme_md()` (if we wanted to and R Markdown file with R code and output we might use `usethis::use_readme_md()` instead).

```r
usethis::use_readme_md()
```

```
 Writing 'README.md'
 Modify 'README.md'
```

~~~
# mypkg

<!-- badges: start -->
<!-- badges: end -->

The goal of mypkg is to ...

## Installation

You can install the released version of mypkg from [CRAN](https://CRAN.R-project.org) with:

``` r
install.packages("mypkg")
~~~

## Example

This is a basic example which shows you how to solve a common problem:

``` r
library(mypkg)
## basic example code
```
```

There are the comments near the top that mention badges and you might have seen badges (or shields) on README files in code repositories before. There are several **usethis** functions for adding badges. For example we can mark this package as been at the experimental stage using `usethis::use_lifecycle_badge()`.
```

```
usethis::use_lifecycle_badge("experimental")
```

```
# mypkg

<!-- badges: start -->
[![Lifecycle: experimental](https://img.shields.io/badge/lifecycle-experimental-orange
<!-- badges: end -->

The goal of mypkg is to ...
```

The rest of the template isn't very useful so replace it with something better.

8.4 Package website

If you have a publicly available package it can be useful to have a website displaying the package documentation. It gives your users somewhere to go and helps your package appear in search results. Luckily this is easily achieved using the **pkgdown** package. If you have it installed you can set it up with **usethis**.

```
usethis::use_pkgdown()
```

Chapter 9

Versioning

We now have at least something for all of the major parts of our package. Whenever you reach a milestone like this it is good to update the package version. Having a good versioning system is important when it comes to things like solving user issues. Version information is recorded in the `DESCRIPTION` file. This is what we have at the moment.

```
Version: 0.0.0.9000
```

This version number follows the format `major.minor.patch.dev`. The different parts of the version represent different things:

- `major` - A significant change to the package that would be expected to break users code. This is updated very rarely when the package has been redesigned in some way.
- `minor` - A minor version update means that new functionality has been added to the package. It might be new functions to improvements to existing functions that are compatible with most existing code.
- `patch` - Patch updates are bug fixes. They solve existing issues but don't do anything new.
- `dev` - Dev versions are used during development and this part is missing from release versions. For example you might use a dev version when you give someone a beta version to test. A package with a dev version can be expected to change rapidly or have undiscovered issues.

Now that we know how this system works let's increase our package version.

```
usethis::use_version()
```

```
Current version is 0.0.0.9000.
Which part to increment? (0 to exit)

1: major --> 1.0.0
2: minor --> 0.1.0
3: patch --> 0.0.1
4:   dev --> 0.0.0.9001

Selection:
```

The prompt asks us which part of the version we want to increment. We have added some new functions so let's make a new minor version.

```
Selection: 2
 Setting Version field in DESCRIPTION to '0.1.0'
```

Whenever we update the package version we should record what changes have been made. We do this is a `NEWS.md` file.

```r
usethis::use_news_md()
```

```
 Writing 'NEWS.md'
 Modify 'NEWS.md'
```

Modify the file to record what we have done during the workshop.

```
# mypkg 0.1.0

* Created the package
* Added the `make_shades()` function
* Added the `plot_colours()` function
* Added a vignette
```

Chapter 10

Building, installing and releasing

If you want to start using your package in other projects the simplest thing to do is run `devtools::install()`. This will install your package in the same way as any other package so that it can be loaded with `library()`. However this will only work on the computer you are developing the package on. If you want to share the package with other people (or other computers you work on) there are a few different options.

10.1 Building

One way to share your package is to manually transfer it to somewhere else. But rather then copying the development directory what you should share is a prebuilt package archive. Running `devtools::build()` will bundle up your package into a `.tar.gz` file without any of the extra bits required during development. This archive can then be transferred to wherever you need it and installed using `install.packages("mypkg.tar.gz", repos = NULL)` or `R CMD INSTALL mypkg.tar.gz` on the command line. While this is fine if you just want to share the package with yourself or a few people you know, it doesn't work if you want it to be available to the general community.

10.2 Official repositories

10.2.1 CRAN

The most common repository for public R packages is the Comprehensive R Archive Network (CRAN). This is where packages are usually downloaded from when you use `install.packages()`. Compared to similar repositories for other programming languages getting your package accepted to CRAN means meeting a series of requirements. While this makes the submission process more difficult it gives users confidence that your package is reliable and will work on multiple platforms. It also makes your package much easier to install for most users and makes it more discoverable. The details of the CRAN submission process are beyond the scope of this workshop but it is very well covered in the "Release" section of Hadley Wickham's "R packages" book (http://r-pkgs.had.co.nz/release.html) and the CRAN section of Karl Broman's "R package primer" (https://kbroman.org/pkg_primer/pages/cran.html). You should also read the offical CRAN submission checklist https://cran.r-project.org/web/packages/submission_checklist.html. The CRAN submission process has a reputation for being prickly and frustrating to go through but it is important to remember that the maintainers are volunteering their time to do this for thousands of packages. Because of their hard work CRAN is a large part of why R is so successful.

10.2.2 Bioconductor

If your package is designed for analysing biological data you might want to submit it to Bioconductor rather than CRAN. While Bioconductor has a smaller audience it is more specialised and is often the first place researchers in the life sciences look. Building a Bioconductor package also means that you can take advantage of the extensive ecosystem of existing objects and packages for handling biological data types. While there are lots of advantages to having your package on Bioconductor the coding style is slightly different to what is often used for CRAN packages. If you think you might want to submit your package to Bioconductor in the future have a look at the Bioconductor package guideline (https://www.bioconductor.org/developers/package-guidelines/) and the how to guide to building a Bioconductor package (https://www.bioconductor.org/developers/how-to/buildingPackagesForBioc/). The Bioconductor submission process is conducted through GitHub (https://bioconductor.org/developers/package-submission/). The Bioconductor maintainers will guide you through the process and make suggestions about how to improve your package and integrate it with other Bioconductor packages. Unlike CRAN which uploads packages all year round Bioconductor has two annual releases, usually in April and October. This means that all the packages in a release are guaranteed to be compatible with each other but make sure you submit in time or you will have to wait another six months for your package to be available to most users.

10.2.3 rOpenSci

rOpenSci is not a package repository as such but an organisation that reviews and improves R packages. Packages that have been accepted by rOpenSci should meet a certain set of standards. By submitting your package to rOpenSci you will get it reviewed by experienced programmers who can offer suggestions on how to improve it. If you are accepted you will receive assistance with maintaining your package and it will be promoted by the organisation. Have a look at their submission page for more details https://github.com/ropensci/software-review.

10.3 Code sharing websites

Uploading your package to a code sharing website such as GitHub, Bitbucket or GitLab offers a good compromise between making your package available and going through an official submission process. This is a particularly good option for packages that are still in early development and are not ready to be submitted to one of the major repositories. Making your package available on one of these sites also gives it a central location for people to ask questions and submit issues. Code sharing websites are usually accessed through the git version control system. If you are unfamiliar with using git on the command line there are functions in **usethis** that can run the commands for you from R. The following steps will take you through uploading a package to GitHub but they are similar for other websites. If you don't already have a GitHub account create one here https://github.com/join.

10.3.1 Set up git

First we need to configure our git details.

```r
usethis::use_git_config(user.name = "Your Name", user.email = "your@email.com")
```

The email address should be the same one you used to sign up to GitHub. Now we can set up git in our package repository.

```r
usethis::use_git()
```

```
  Initialising Git repo
There are 13 uncommitted files:
```

```
* '.gitignore'
* '.Rbuildignore'
* 'DESCRIPTION'
* 'LICENSE'
* 'LICENSE.md'
* 'man/'
* 'mypkg.Rproj'
* 'NAMESPACE'
* 'NEWS.md'
* 'R/'
* 'README.md'
* 'tests/'
* 'vignettes/'
Is it ok to commit them?

1: Not now
2: Yup
3: Negative

Selection: 2
  Adding files
  Commit with message 'Initial commit'
  A restart of RStudio is required to activate the Git pane
Restart now?

1: No
2: No way
3: Yes

Selection: 1
```

If you are already familiar with git this should make sense to you. If not, what this step does (in summary) is set up git and save the current state of the package. If you chose to restart RStudio you will see a new git pane that can be used to complete most of the following steps by pointing and clicking.

10.3.2 Connect to GitHub

The next step is to link the directory on your computer with a repository on GitHub. First we need to create a special access token. The following command will open a GitHub website.

```r
usethis::use_github()
```

```
  Opening URL 'https://github.com/settings/tokens/new?scopes=repo,gist&description=R:GI
  Call `usethis::edit_r_environ()` to open '.Renviron'.
  Store your PAT with a line like:
   GITHUB_PAT=xxxyyyzzz
   [Copied to clipboard]
  Make sure '.Renviron' ends with a newline!
```

Click the "Generate token" button on the webpage and then copy the code on the next page. As it says you can only view this once so be careful to copy it now and don't close the page until you are finished. When you have the code follow the rest of the instructions from **usethis**.

```r
usethis::edit_r_environ()
```

```
  Modify 'C:/Users/Luke/Documents/.Renviron'
  Restart R for changes to take effect
```

Edit the file to look something like this (with your code).

```
GITHUB_PAT=YOUR_CODE_GOES_HERE
```

Save it then restart R by clicking the *Session* menu and selecting *Restart R* (or using **Ctrl+Shift+F10**).

```
Restarting R session...
```

Copying that code and adding it to your `.Renviron` gives R on the computer you are using access to your GitHub repositories. If you move to a new computer you will need to do this again. Now that we have access to GitHub we can create a repository for our packages.

```r
usethis::use_github()
```

```
  Setting active project to 'C:/Users/Luke/Desktop/mypkg'
  Checking that current branch is 'master'
Which git protocol to use? (enter 0 to exit)

1: ssh   <-- presumes that you have set up ssh keys
2: https <-- choose this if you don't have ssh keys (or don't know if you do)

Selection: 2
  Tip: To suppress this menu in future, put
   `options(usethis.protocol = "https")`
   in your script or in a user- or project-level startup file, '.Rprofile'.
  Call `usethis::edit_r_profile()` to open it for editing.
 Check title and description
  Name:        mypkg
  Description: My Personal Package
Are title and description ok?

1: For sure
2: No way
3: Negative

Selection: 1
 Creating GitHub repository
 Setting remote 'origin' to 'https://github.com/lazappi/mypkg.git'
 Adding GitHub links to DESCRIPTION
 Setting URL field in DESCRIPTION to 'https://github.com/lazappi/mypkg'
 Setting BugReports field in DESCRIPTION to 'https://github.com/lazappi/mypkg/issues'
 Pushing 'master' branch to GitHub and setting remote tracking branch
 Opening URL 'https://github.com/lazappi/mypkg'
```

Respond to the prompts from **usethis** about the method for connecting to GitHub and the title and description for the repository. When everthing is done a website should open with your new package repository. Another thing this function does is add some extra information to the description that let's people know where to find your new website.

```
URL: https://github.com/user/mypkg
BugReports: https://github.com/user/mypkg/issues
```

10.3.3 Installing from GitHub

Now that your package is on the internet anyone can install it using the `install_github()` function in the **remotes** package (which you should already have installed as a dependency of **devtools**). All you need to give it is the name of the user and repository.

```r
remotes::install_github("user/mypkg")
```

If you are familiar with git you can install from a particular branch, tag or commit by adding that after `@`.

```r
remotes::install_github("user/mypkg@branch_name")
remotes::install_github("user/mypkg@tag_id")
remotes::install_github("user/mypkg@commit_sha")
```

10.3.4 Updating GitHub

After you make improvements to your package you will probably want to update the version that is online. To do this you need to learn a bit more about git. Jenny Bryan's "Happy Git with R" tutorial (https://happygitwithr.com) is a great place to get started but the (very) quick steps in RStudio are:

1. Open the *Git* pane (it will be with *Environment*, *History* etc.)
2. Click the check box next to each of the listed files
3. Click the *Commit* button
4. Enter a message in the window that opens and click the *Commit* button
5. Click the *Push* button with the green up arrow

Refresh the GitHub repository website and you should see the changes you have made.

Chapter 11

Advanced topics

In this workshop we have worked though the basic steps required to create an R package. In this section we introduce some of the advanced topics that may be useful for you as you develop more complex packages. These are included here to give you an idea of what is possible for you to consider when planning a package. Most of these topics are covered in Hadley Wickhams "Advanced R" book (https://adv-r.hadley.nz) but there are many other guides and tutorials available.

11.1 Including datasets

It can be useful to include small datasets in your R package which can be used for testing and examples in your vignettes. You may also have reference data that is required for your package to function. If you already have the data as an object in R it is easy to add it to your package with `usethis::use_data()`. The `usethis::use_data_raw()` function can be used to write a script that reads a raw data file, manipulates it in some way and adds it to the package with `usethis::use_data()`. This is useful for keeping a record of what you have done to the data and updating the processing or dataset if necessary. See the "Data" section of "R packages" (http://r-pkgs.had.co.nz/data.html) for more details about including data in your package.

11.2 Designing objects

If you work with data types that don't easily fit into a table or matrix you may find it convenient to design specific objects to hold them. Objects can also be useful for holding the output of functions such as those that fit models or perform tests. R has several different object systems. The S3 system is the simplest and probably the most commonly used. Packages in the Bioconductor ecosystem make use of the more formal S4 system. If you want to learn more about desiging R objects a good place to get started is the "Object-oriented programming" chapter of Hadley Wickham's "Advanced R" book (https://adv-r.hadley.nz/oo.html). Other useful guides include Nicholas Tierney's "A Simple Guide to S3 Methods" (https://arxiv.org/abs/1608.07161) and Stuart Lee's "S4: a short guide for the perplexed" (https://stuartlee.org/post/content/post/2019-07-09-s4-a-short-guide-for-perplexed/).

11.3 Integrating other languages

If software for completing a task already exists but is in another language it might make sense to write an R package that provides an interface to the existing implementation rather than replementing it from scratch. Here are some of the R packages that help you integrate code from other languages:

- **Rcpp** (C++) http://www.rcpp.org/
- **reticulate** (Python) https://rstudio.github.io/reticulate/
- **RStan** (Stan) https://mc-stan.org/users/interfaces/rstan
- **rJava** (Java) http://www.rforge.net/rJava/

Another common reason to include code from another language is to improve performance. While it is often possible to make code faster by reconsidering how things are done within R sometimes there is no alternative. The **Rcpp** package makes it very easy to write snippets of C++ code that is called from R. Depending on what you are doing moving even very small bits of code to C++ can have big impacts on performance. Using **Rcpp** can also provide access to existing C libraries for specialised tasks. The "Rewriting R code in C++" section of "Advanced R" (https://adv-r.hadley.nz/rcpp.html) explains when and how to use **Rcpp**. You can find other resources including a gallery of examples on the official **Rcpp** website (http://www.rcpp.org/).

11.4 Metaprogramming

Metaprogramming refers to code that reads and modifies other code. This may seem like an obscure topic but it is important in R because of it's relationship to non-standard evaluation (another fairly obscure topic). You may not have heard of non-standard evaluation before but it is likely you have used it. This is what happens whenever you provide a function with a bare name instead of a string or a variable. Metaprogramming particularly becomes relevant to package development if you want to have functions that make use of packages in the Tidyverse such as **dplyr**, **tidy** and **purrr**. The "Metaprogramming" chapter of “Advanced R” (https://adv-r.hadley.nz/metaprogramming.html) covers the topic in more detail and the “Tidy evaluation” book (https://tidyeval.tidyverse.org/) may be useful for learning how to write functions that use Tidyverse packages.

Chapter 12

Good practices and advice

This section contains some general advice about package development. It may be opinionated in places so decide which things work for you.

12.1 Design advice

- **Compatibility** - Make your package compatible with how your users already work. If there are data structure that are commonly used write your functions to work with those rather than having to convert between formats.
- **Ambition** - It's easy to get carried away with trying to make a package that does everything but try to start with whatever is most important/novel. This will give you a useful package as quickly and easily as possible and make it easier to maintain in the long run. You can always add more functionality later if you need to.
- **Messages** - Try to make your errors and messages as clear as possible and other advice about how to fix them. This can often mean writing a check yourself rather than relying on a default message from somewhere else.
- **Check input** - If there are restrictions on the values parameters can take check them at the beginning of your functions. This prevents problems as quickly as possible and means you can assume values are correct in the rest of the function.
- **Useability** - Spend time to make your package as easy to use as possible. Users won't know that your code is faster or produces better results if they can't understand how to use your functions. This includes good documentation but also things like having good default values for parameters.
- **Naming** - Be obvious and consistent in how you name functions and parameters. This makes it easier for users to guess what they are without

looking at the documentation. One option is to have a consistent prefix to function names (like **usethis** does) which makes it obvious which package they come from and avoids clashes with names in other packages.

12.2 Coding style

Unlike some other languages R is very flexible in how your code can be formatted. Whatever coding style you prefer it is important to be consistent. This makes your code easier to read and makes it easier for other people to contribute to it. It is useful to document what coding style you are using. The easiest way to do this is to adopt a existing style guide such as those created for the Tidyverse (https://style.tidyverse.org/) or Google (https://google.github.io/styleguide/Rguide.html) or this one by Jean Fan (https://jef.works/R-style-guide/). If you are interested in which styles people actually use check out this analysis presented at useR! 2019 https://github.com/chainsawriot/rstyle. When contibuting to other people's projects it is important (and polite) to conform to their coding style rather than trying to impose your own.

If you want to make sure the style of your package is consistent there are some packages that can help you do that. The **lintr** package will flag any style issues (and a range of other programming issues) while the **styler** package can be used to reformat code files. The **goodpractice** package can also be used to analyse your package and offer advice. If you are more worried about problems with the text parts of your package (documentation and vignettes) then you can activate spell checking with `usethis::use_spell_check()`.

12.3 Version control

There are three main ways to keep tracks of changes to your package:

1. Don't keep track
2. Save files with different versions
3. Use a version control system (VCS)

While it can be challenging at first to get your head around git (or another VCS) it is highly recommended and worth the effort, both for packages and your other programming projects. Here are something of the big benefits of having your package in git:

- You have a complete record of every change that has been made to the package
- It is easy to go back if anything breaks or you need an old version for something

- Because of this you don't have to worry about breaking things and it is easier to experiment
- Much easier to merge changes from collaborators who might want to contribute to your package
- Access to a variety of platforms and services built around this technology, for example installing your package, hosting a package website and continuous integration (see below)

As mentioned earlier a great way to get started with git for R projects is Jenny Bryan's "Happy Git with R" (https://happygitwithr.com) but there are many more tutorials and workshops available.

12.4 Continuous integration

During the workshop we showed you how to run checks and tests on your package but this will only tell you if they pass on your particular computer and platform. Continuous integration services can be used to automatically check your package on multiple platforms whenever you make a significant change to your package. They can be linked to your repository on code sharing websites like GitHub and whenever you push a new version they will run the checks for you. This is similar to what CRAN and Bioconductor do for their packages but we doing it yourself you can be more confident that you won't run into issues when you submit your package to them. If your package isn't on one of the major repositories it helps give your users confidence that it will be reliable. Some continuous integration services are:

- Travis CI (https://travis-ci.com/)
- AppVeyor (https://www.appveyor.com/)
- CircleCI (https://circleci.com/)

Each of these has a free service at it is easy to set them up for your package using the appropriate `usethis::use_CI_SERVICE()` function.

Resources

This section has links to additional resources on package development, many of which were used in developing these materials.

Official guides

CRAN

- **Writing R extensions** https://cran.r-project.org/doc/manuals/R-exts.html

Bioconductor

- **Bioconductor Package Guidelines** https://www.bioconductor.org/developers/package-guidelines/
- **Building Packages for Bioconductor** https://www.bioconductor.org/developers/how-to/buildingPackagesForBioc/
- **Bioconductor Package Submission** https://bioconductor.org/developers/package-submission/

rOpenSci

- **rOpenSci Packages: Development, Maintenance, and Peer Review** https://devguide.ropensci.org/

RStudio

- **Developing Packages with RStudio** https://support.rstudio.com/hc/en-us/articles/200486488-Developing-Packages-with-RStudio
- **Building, Testing and Distributing Packages** https://support.rstudio.com/hc/en-us/articles/200486508-Building-Testing-and-Distributing-Packages

- **Writing Package Documentation** https://support.rstudio.com/hc/en-us/articles/200532317-Writing-Package-Documentation

Books

- **R Packages** (Hadley Wickham) http://r-pkgs.had.co.nz/
- **Advanced R** (Hadley Wickam) https://adv-r.hadley.nz/
- **Advanced R Course** (Florian Privé) https://privefl.github.io/advr38book/

Tutorials

- **Writing an R package from scratch** (Hilary Parker) https://hilaryparker.com/2014/04/29/writing-an-r-package-from-scratch/
- **Writing an R package from scratch (Updated)** (Thomas Westlake) https://r-mageddon.netlify.com/post/writing-an-r-package-from-scratch/
- **usethis workflow for package development** (Emil Hvitfeldt) https://www.hvitfeldt.me/blog/usethis-workflow-for-package-development/
- **R package primer** (Karl Broman) https://kbroman.org/pkg_primer/
- **R Package Development Pictorial** (Matthew J Denny) http://www.mjdenny.com/R_Package_Pictorial.html
- **Building R Packages with Devtools** (Jiddu Alexander) http://www.jiddualexander.com/blog/r-package-building/
- **Developing R packages** (Jeff Leek) https://github.com/jtleek/rpackages
- **R Package Tutorial** (Colautti Lab) https://colauttilab.github.io/RCrashCourse/Package_tutorial.html
- **Instructions for creating your own R package** (MIT) http://web.mit.edu/insong/www/pdf/rpackage_instructions.pdf

Workshops and courses

- **R Forwards Package Workshop** (Chicago, February 23, 2019) https://github.com/forwards/workshops/tree/master/Chicago2019
- **Write your own R package** (UBC STAT 545) http://stat545.com/packages00_index.html

Blogs

- **How to develop good R packages (for open science)** (Malle Salmon) https://masalmon.eu/2017/12/11/goodrpackages/

Style

- **Tidyverse** https://style.tidyverse.org/
- **Google** https://google.github.io/styleguide/Rguide.html
- **Jean Fan** https://jef.works/R-style-guide/
- **A Computational Analysis of the Dynamics of R Style Based on 94 Million Lines of Code from All CRAN Packages in the Past 20 Years.** (Yen, C.Y., Chang, M.H.W., Chan, C.H.) https://github.com/chainsawriot/rstyle