Package Development in R for Predator Free New Zealand 2050

Anthony Davidson

January 2022

Contents

Introduction

I have taken the guide generated at `https://iqss.github.io/dss-rbuild/` for my PhD work and future package development.

Table of Contents

This guide is organized into three parts.

1. First, we'll run through The proper way to structure and test packages.

2. Second, we'll discuss Version Control with Git and GitHub

3. Lastly, we'll briefly look at a couple of IDEs, which are just pieces of software that make it easier to write packages.

Here's a link to a great devtools cheatsheet which puts most of the useful commands in this guide in one place.

NOTE 2022: I hope to do this for PFNZ 2050 too

We also wrote a small development example package (called `devex`) which you can find linked here, on GitHub. If you haven't used GitHub yet, don't worry - we'll go over how to use GitHub later.

Also, this guide is still under development, and we take feedback! If you find anything confusing or think the guide misses important content, please email `help@iq.harvard.edu`

NOTE 2022: I have taken the original guide with the key contributors below and developed some extra code and other resources I have used for my PhD work. Maybe some of it will be helpful in the future? :)
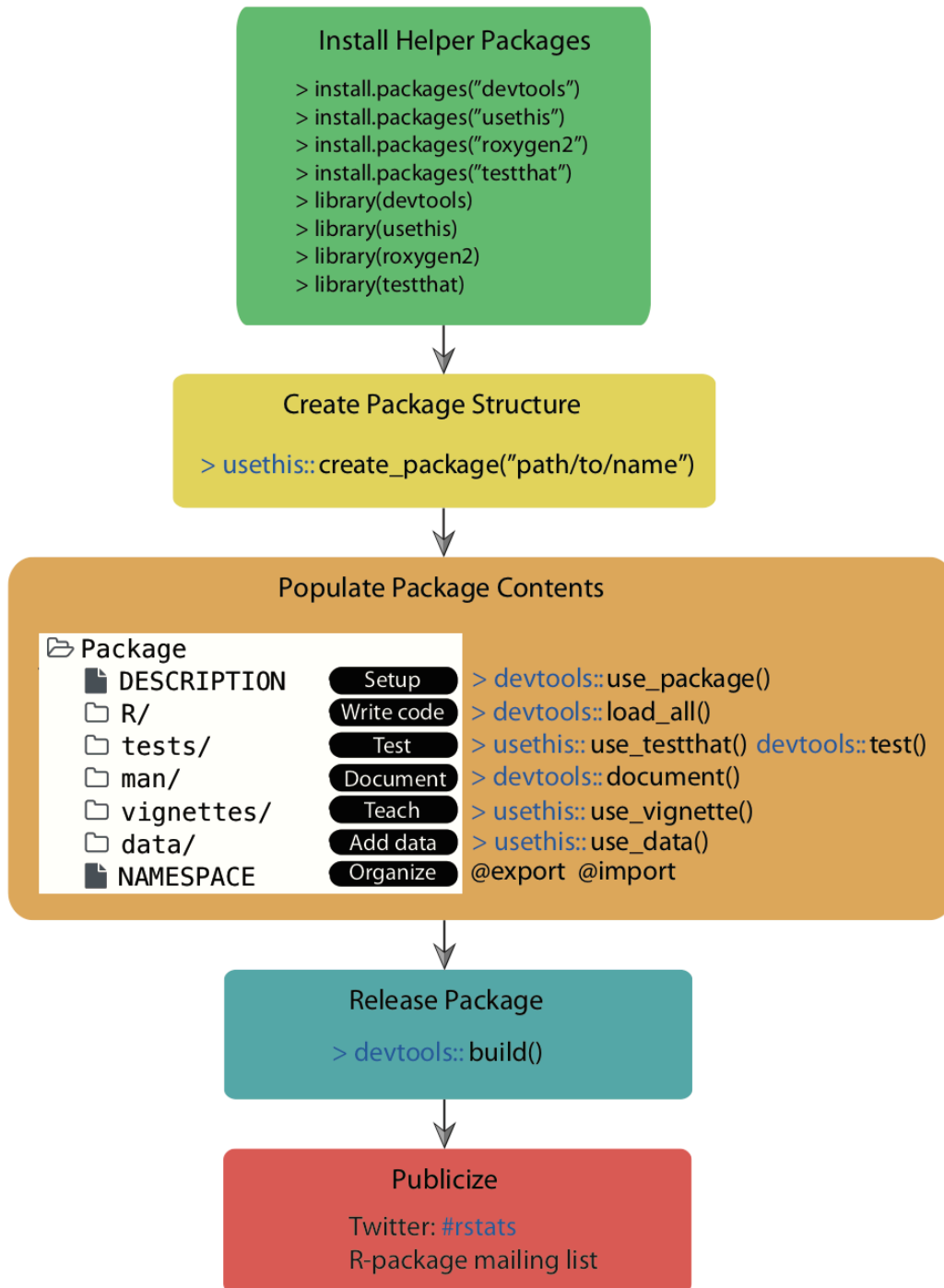
Authors and Sources

It's worth acknowledging a few people who helped make this guide possible. First of all, Simo Goshev and Steve Worthington at Harvard's IQSS helped design the structure of the guide and edited the content. Second, Asher Spector at Harvard College did the hard work of actually writing the tutorial in Rmarkdown and configuring the GitHub repo and website. Third, Jinjie Liu at IQSS helped to polish the content. Fourth, this guide was written for a different audience, but a lot of its structure and content is based on Hadley Wickham's book R Packages. To help write the sections on testing, we also referenced Christopher Gandrud's

'Failing Faster' Presentation, and Christopher Gandrud's Broader Testing Guidelines. For the section on on Version Control, we referenced Karl Broman's Book, a Git-Tower post, and the GitHub documentation here and here to help write this guide.

Chapter 1

Package Development

The following graphic outlines this entire chapter:

Install Helper Packages

> install.packages("devtools")
> install.packages("usethis")
> install.packages("roxygen2")
> install.packages("testthat")
> library(devtools)
> library(usethis)
> library(roxygen2)
> library(testthat)

Create Package Structure

> usethis:: create_package("path/to/name")

Populate Package Contents

📂 Package
📄 DESCRIPTION   [Setup]   > devtools:: use_package()
📁 R/   [Write code]   > devtools:: load_all()
📁 tests/   [Test]   > usethis:: use_testthat() devtools:: test()
📁 man/   [Document]   > devtools:: document()
📁 vignettes/   [Teach]   > usethis:: use_vignette()
📁 data/   [Add data]   > usethis:: use_data()
📄 NAMESPACE   [Organize]   @export @import

Release Package

> devtools:: build()

Publicize

Twitter: #rstats
R-package mailing list

As a quick reminder, you can find the `devex` example package linked here, on GitHub, if you'd like to look through it while reading the guide.

1.1   File Paths

The paths used to locate files differ between UNIX (e.g., MacOS, Linux) and Windows based operating systems. Windows systems used a single backslash \, while UNIX systems use a single forward slash / to delimit files or directories in the path. R follows the UNIX convention of using forward slashes, but forces this on Window's users too. So, file paths should look like this:

1. MacOS and Linux

~/Documents/myproject/myRfile.R

2. Windows

C:/Documents/myproject/myRfile.R

In the examples provided in this guide, we will show MacOS file paths. If you're using Windows, you will need to modify these paths slightly to show the drive letter at the beginning, followed by a colon.

1.2   Downloading Development Tools

Before we get started, you'll want to download four packages that are extremely useful for package development. For Mac/Linux users, head to the R Gui (or your favorite IDE) and run the code below to download the packages. If you are prompted to choose a CRAN mirror for your session, simply pick the mirror closest to your location.

```
install.packages('devtools')
install.packages('usethis')
install.packages('roxygen2')
install.packages('testthat')
```

If you are using Windows, prior to running the above code, you will need to install "RTools" by following these instructions:

1. Go to `https://cran.rstudio.com/` and select "Download R for Windows."

2. Click "RTools" and download the latest version of the tools (or the tools that are compatible with your version of R).

3. Let the installer run itself (the defaults are fine).

1.3   Initializing the Package

Now we can begin to walk through the process of creating an R package. The first thing you'll always want to do is run the following function to initialize the package:

```
usethis::create_package('path/to/desired/location/packagename')
```

Note that the path specified by `usethis::create_package()` must currently be empty, otherwise `usethis` will throw an error. Successfully running this function will create a

couple of important files which constitute a skeletal outline of the package. In particular, it will create:

1. An 'R' subdirectory in the root of your specified directory, which is where all of core the R code of your package will live.
2. A DESCRIPTION file, which will come with a couple of preset fields.
3. A NAMESPACE file.

You can then set the working directory for your project to an "active" status:

```
usethis::proj_set('path/to/desired/location/packagename')
```

Once you've set up the basic structure of your package, you can start modifying files and writing it in earnest!

1.4   The DESCRIPTION

The DESCRIPTION file gives an extremely brief overview to the package. It includes critical information such as the author of the package, the title, a very short summary of its purpose, and the licensing information. Open the "DESCRIPTION" file using your favorite text editor in order to inspect and edit its contents.

DESCRIPTION is a DCF file (Debian control format). This file format may be unfamiliar, but it's quite simple. Each line contains a field name and value, separated by a colon. Sometimes, values are long enough to require multiple lines, in which case they are indented by four spaces. For example, the DESCRIPTION file for a newly created package will look like this:

```
1   Package: my-package-name
2   Title: What the Package Does (One Line, Title Case)
3   Version: 0.0.0.9000
4   Authors@R:
5       person(given = "First",
6           family = "Last",
7           role = c("aut", "cre"),
8           email = "first.last@example.com",
9           comment = c(ORCID = "YOUR-ORCID-ID"))
10  Description: What the package does (one paragraph).
11  License: What license it uses
12  Encoding: UTF-8
13  LazyData: true
```

Let's go through the fields and discuss what they mean. The first seven fields listed are mandatory, meaning that if you do not include them, the development environment will throw an error later on when you're trying to build your package.

1. **Package**: This is the name of the package. It should match the package name you chose earlier, and you should probably just leave this as is.
2. **Title**: A short but more descriptive title of your package than its name.

3. **Version**: The version of your package. Since you're creating this package for the first time, presumably it's version 0.1.0.
4. **Authors**: Here, you should add in your given and family names, role, email, and (optionally) your ORCID.
5. **Description**: This should be a one-paragraph *comprehensive* description of the package. It is necessarily a high level-description, but it should be a complete one.
6. **License**: You should add in a License, which describes how others can legally use the package. Most of the time (especially in the US), you should write 'CC0' in the License field, which implies that the package is open for all use, and you have relinquished all your rights to it. For more information on various licensing options, click this link.
7. **Encoding**: Just leave this as "UTF-8"; discussing what encodings are isn't super important for this guide. If you're dying to learn about encodings, visit this webpage.
8. **LazyData**: Just leave this as 'true', which ensures that if you include any data with your package (which you frequently will), when another user loads your package, they won't automatically load up the data, but will only load it if it becomes necessary during their use. This option reduces the amount of RAM users have to expend when loading packages, especially if you are planning to include a lot of data with your package.

(Note all of the fields from this point on are optional, but encouraged!)

9. **Type**: This describes what type of project you're creating - in this case, because you're creating a package, you should write "Package."
10. **Date**: The date, in YYYY-MM-DD fashion.
11. **RoxygenNote**: Roxygen will automatically fill in the version of `Roxygen2` used to build the package in this field.

(These fields are exceptionally important if you are building a package using tools from other packages)

12. **Imports**: In this field, you should list the packages which your package needs to function. Each package should be indented by two spaces, separated by a comma, and given its own line. For example, a package which requires `ggplot2`, `nlme`, and `rpart` might have an 'imports' field which looks like this:

```
knitr::include_graphics("images/packageSS/imports.PNG")
```

```
14  Imports:
15    ggplot2,
16    rpart,
17    nlme
```

13. **Suggests**: Sometimes, your package will not really *require* the use of other packages, but it might offer a couple of extra wrappers/functions with those other packages. When those extra functions aren't strictly necessary, it's a good idea to have your package *suggest* imports. For example, a package which includes the following function should include `ggplot2` in the 'suggests' part of the description.

```
scalep <- function(d, x=1){
  ...
  # If ggplot2 is available, use its qplot function - else, use the default hist function
  if (requireNamespace("ggplot2", quietly = TRUE)) {
```

```
    ggplot2::qplot(r, geom='histogram')
  } else {
    hist(r)
  }
  ...
}
```

Here, the function `requireNameSpace()` checks if `ggplot2` is available, and if not, the function uses the (slightly less pretty) default histogram function.

Once you know which packages to list in the 'suggests' section, you can list them exactly the same way you'd list functions in the 'imports' section: each package is indented by two spaces, separated by a comma, and gets its own line.

In general, it's best to suggest functions instead of requiring them if you barely use them in your package. This will give users a bit more flexibility, because it won't force them to download packages they will probably never use.

1.5   Writing Code

1.5.1   General Coding Guidelines

All of your code should be in scripts in the 'R' file created in the package development environment, as shown below:

The coding you will do in package development is slightly different than the coding you'll normally do when writing R scripts. This is for a couple of reasons:

1. When you write a script and load that script using `source("script_name")`, the code in the script runs when you load it (specifically, when you run the `source()` command). On the other hand, the code in a package is run when the package is *built* on your computer. As a result, your code should mostly be focused on building functions, as opposed to a series of actions which the computer ought to take.

2. Unlike your personal scripts, other people will be using your package, and if your package is good, they'll be using it in ways you didn't anticipate. This means you ought to really try to make sure your code is as general as possible and can support a variety of approaches and implementations.

3. Also, because other people will be using your package, you should avoid modifying the global environment with your package. This means avoiding using functions like `require()`, `library()`, or `source()`; instead, there are other alternatives which can accomplish the same goal without changing the global environment and potentially giving other users an unwanted surprise. For example, instead of using `library()` and `require()`, you should be listing your necessary imports in the DESCRIPTION file, as outlined above, and then R will make sure anyone who installs and loads your package also has any other packages your package depends on. The one catch is that you'll now have to append `packagename::` in front of the imported functions you want to use, otherwise R won't recognize them. For example, to use the `qplot()` function from the `ggplot2` package, the code should look like:

```
...
scalep <- function(d, x=1){
...
  ggplot2::qplot(r, geom='histogram')
...
}
```

The last thing you should know is that if you want your package to plot things, you will have to surround the `plot()` commands with a `print()` statement, like this:

```
...
scalep <- function(d, x=1){
...
  print(ggplot2::qplot(r, geom='histogram'))
...
}
```

You should also take care to organize your functions properly. It's probably a bad idea to stick them all into one script and title it "functions." Instead, you should organize functions by their purposes - for example, a variety of loss functions might go into a single script. Of course, some very complicated functions might deserve their own script. The file names of the script should be descriptive - for example, a script of loss functions might be named "loss_functions.R".

1.5.2 Code Style

Now let's talk about code style. These recommendations are shortened and adapted from Hadley Wickham's book, which in turn were adapted from Google's style guide.

1. **Comments**: Comments are the best way to make your code readable. In general, you should err on the side of commenting too much rather than too little, and your comments should explain the *motivation* of your code as opposed to what your code actually does (although admittedly the line between those two things is a bit blurry). Moreover, you can use lines of '# ——————————' or '# ==================' to separate sections of your code. Here are some

examples:

```r
# Returns the squared element-wise difference between two vectors
loss <- function(x,y) {
  error <- (x-y)**2
  return(error)
}
#-----------------------------------------------------------------------------

# Takes the square root of any real number, returning a complex number
general_sqrt <- function (x) {
  # Return the normal square root if x > 0
  if (x > 0 || x == 0) {
    return(complex(real=sqrt(x), imaginary=0))
  }
  # Else return the complex square root
  else {
    return(complex(real = 0, imaginary = sqrt(-x)))
  }
}
```

2. **Names**: Variable and function names should be descriptive but concise, and variable names should generally be nouns whereas function names tend to be verbs. Most R developers keep their function/variable names all lowercase and separate multiple words with underscores. There are no strict rules on this, but it's nice to be consistent with *some* rules because it makes your code readable.

```r
# Bad example - function name
f <- function(x) {
  return(sqrt(x))
}

# Good example - function name
take_sqrt <- function(x){
  return(sqrt(x))
}

# Bad example - variable name
s <- read.table(path)

# Good example - variable name
car_data <- read.table(path)
```

3. **Curly Braces**: You should start a new line after you write an opening curly brace, and ending curly braces should get their own lines, unless you have an else clause or the line is exceptionally simple.

```r
# Bad examples

if (condition) {
```

```r
  complicated_function(x)} # Ending curly brace should get a new line

# Good examples
if (condition) {do(x)} else {do(y)}

if (condition) {
  complicated_function_call(arg1, arg2, arg3)
} else {
  other_complex_function_call(arg7, arg3, arg5)
}
```

As always, you can break the rules if you have a good reason to.

Different organizations and programmers may have different styles, but in general, you should remember:

1. Your goal should always be to make your code readable!
2. Whatever style guide you follow, follow it *consistently*.
3. When in doubt, follow the conventions of the organization you're working for.

1.5.3 Warnings and Simplicity

Consider the case of the `scalep()` function, which currently takes a 2-column dataframe as an input, divides the second column by the first, and returns/graphs some scaled proportion of the quotient vector. One version of this function, which follows almost all of the guidelines above, might look like this:

```r
scalep <- function(d, x=1){

  # Intialize resulting vector
  result <- c()

  # Iterate through and divide column 2 of d by column 1 of d
  i <- 0
  while(i < length(d[ ,1]) + 1){
    row <- d[i,]
    result <- append(result, row[[2]]/row[[1]])
    i <- i + 1
  }

  # Print the graph using either ggplot2 or the hist function
  if (requireNamespace("ggplot2", quietly = TRUE)) {
    print(ggplot2::qplot(result, geom='histogram'))
  } else {
    print(hist(result))
  }

  # Return the result, multiplying by the optional scalar
  result <- x*result
```

```r
  return(result)
}
```

However, this function still has a couple of problems. It's not super easy to use because (a) it's understandably hard for other programmers to remember which column is divided by which and (b) there's a simpler way to accomplish the code above which will make it more readable. Specifically, it might be easier to just have arguments called 'factors' and 'divisors' and then divide them, like this:

```r
scalep <- function(factors, divisors, constant = 1) {

  # Divide and multiply by optional scalar
  proportions <- constant*factors/divisors

  # Print the graph using either ggplot2 or the hist function
  if (requireNamespace("ggplot2", quietly = TRUE)) {
    print(ggplot2::qplot(proportions, geom='histogram'))
  } else {
    print(hist(proportions))
  }

  # Return the result
  return(proportions)
}
```

The new argument structure will make it a bit easier to use. Similarly, the new structure simplifies the code, making it a bit more readable. However, it does pose one problem: whereas the previous structure mandated that the two vectors be the same length (because they were part of a dataframe), in this function, the two vectors might not be the same length and the function would not always throw an error (specifically, if 'factors' has a length which is is an integer multiple of the length of 'divisors', R will not warn the user at all). This problem is a type of **silent error** (silent errors are bugs which do not issue warnings or errors). Silent errors are terrible because they make bug-hunting extremely difficult: in large repositories of code, it becomes nearly impossible to find which specific line is causing problems without some kind of warning. Thus, it's also worth adding in a couple of lines to warn the user if the factors and divisors are of different lengths, as is outlined below:

```r
  # Check divisors and factors are the same length
  if (length(divisors) != length(factors)) {
    warning('Length of divisors argument is not equal to length of factors argument')
  }
```

Lastly, it's just worth adding an extra optional argument to let your users turn off the graphing feature of `scalep()`, just to make the function more useable, as follows:

```r
scalep <- function(factors, divisors, constant = 1, graph = FALSE) {

  ...
```

```r
# If graph = True, print the graph using either ggplot2 or the hist function
if (requireNamespace("ggplot2", quietly = TRUE) & graph) {
  print(ggplot2::qplot(proportions, geom='histogram'))
} else if (graph) {
  print(hist(proportions))
}

...
```

To summarize, this subsection thus contained three core ideas: (1) make your code simple, (2) make it easy to use by labeling arguments, and (3) always avoid silent errors.

1.5.4 Loading Your Code

If you've finished writing your code and want to play with it a little bit, you can use the following function:

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

which (according to its documentation) "roughly simulates what happens when a package is installed and loaded with library." As we'll see in the build, testing, and RStudio sections, there are better ways to simulate the user experience and test your code, but `load_all()` is often a useful intermediate step.

1.6 Testing

1.6.1 Why should you test?

Suppose an imaginary programmer named Grace has created a package and has been using it for a while, but she decides she'd like to modify one function to improve it. She modifies her function, tests it a bit, and then publishes a new version of the package. Yet two weeks later, another imaginary programmer named Carlos discovers that the changes she made created a bug in *another* function in the package! This situation is very annoying, especially if Carlos has no idea what has caused the bug or how to fix it. Unfortunately, it's also an extremely common problem.

The solution to problems like this is to test your package *systematically* and *automatically*. If Grace had rigorously tested the entire package before publishing it, Carlos would never have had to deal with the new bug- Grace would have found out immediately. In other words, a good principle in package development is to make sure your code *fails as fast as possible,* so you can find out and fix it. Of course, all programmers test their code, but not everyone tests systematically and automatically.

1.6.2 What are unit tests?

Tests compare the *expected* output of a block of code to its *actual* output. For example, the following test tests whether the "generalized square root" function actually returns 2 as the square root of 4.

```
expect_equal(general_sqrt(4), complex(real = 2, imaginary = 0))
```

Unit tests usually run on the computer of the developer who is modifying a package and also should run automatically upon building a package.

We'll talk a little more about how exactly to create tests below, but hopefully this makes the general concept clear (you've also probably been using the general concept as you program).

1.6.3   Setting up the testing environment

Creating unit tests is actually quite easy, thanks to a package called `testthat` which works in combination with `usethis`. To begin, you should run the following command in your favorite IDE, or even in the R Gui:

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

This will do a couple of things. First, it will add `testthat` to the Suggests part of the DESCRIPTION, which will help other collaborators know to use `testthat` when modifying/working on the package. It will also create a 'tests/testthat' directory in your project, as well a file called 'test/testthat.R', as shown below.
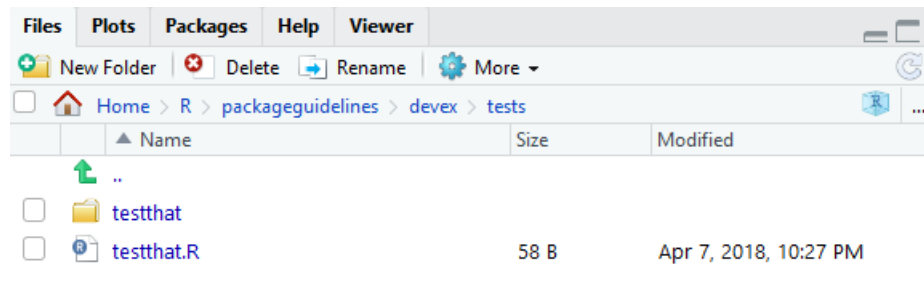
1.6.4   Expectations

Before discussing how to write unit tests, we need to properly describe an expectation. An expectation tests whether the actual output of a single function call is what the developer expected. The `testthat` package has a number of functions which compare outputs to expected values. When calling one of these functions, one of two things can happen:

1. If the actual output matches the expectation, nothing will happen!
2. If the actual output does not match the expectation, it will throw an error.

For example, `expect_equal()` uses the base R function `all.equal()` to check whether an output is (approximately) equal to an expectation. In the following code, the first function call will do nothing - the second function call will throw an error, displayed below.

```
library(testthat)
testthat::expect_equal(2, 2)
testthat::expect_equal(2, 4)
```

```
Error: 2 not equal to 4. 1/1 mismatches [1] 2 - 4 == -2      ↥ Show Traceback
```

Here's an (abbreviated) list of the expectation functions:

1. **expect_equal**, as aforementioned, checks equality using the "all.equal()" base function.
2. **expect_identical** checks equality using the `identical()` base function. Generally, it's better to use `expect_equal()` because lots of R functions use numerical approximations which will cause expect_identical to fail when you don't want it to.
3. **expect_match**, **expect_output**, **expect_message**, **expect_warning**, and **expect_error** all respectively test whether a string, output, warning, or error match a regular expression. For example, the following two expectations functions will not throw errors:

```r
testthat::expect_match('hello1234', 'hello')
testthat::expect_warning(sqrt(-2), 'NaNs produced')
```

The tests do not fail because (i) 'hello1234' contains 'hello' and (ii) the error message produced by `sqrt(-2)` contains the phrase 'NaNs produced'.

4. **expect_is** tests whether an object inherits from a class, specified in quotes. For example, the following test passes:

```r
testthat::expect_is(sqrt(2), 'numeric')
```

5. **expect_true** and **expect_false** respectively expect a statement to evaluate to TRUE or FALSE.

1.6.5 Structure and Location of Unit Tests

Each *unit test* (which is written in an R script) should use a couple of expectations to test a single core function. It should use the function **test_that()** (from the `testthat` package). `test_that()` takes two parameters: a string, which describes the test, and a couple of expectations, surrounded by curly braces. For example, the following code will test whether the `general_sqrt()` function from the `devex` package returns a complex number.

```r
test_that("Returns complex number", {
  expect_is(general_sqrt(-2), 'complex')
  expect_is(general_sqrt(2), 'complex')
  expect_is(general_sqrt(0), 'complex')
})
```

Multiple tests with similar functions should be put in the same file, and those test files must be put in the tests/testthat/ directory. Moreover, their name must start with the word 'test' - this will help R automatically run your tests for you. For example, in the `devex` package, there are two very simple helper functions (`general_sqrt()` and `loss()`) and one moderately complex function (`scalep()`). As a result, the `devex` package has exactly two testing files: one called 'testhelpers', which tests the helper functions, and another called 'testscalep', which tests the `scalep()` function. The 'testhelpers' file looks like this:

```r
library(devex)
context("generalized sqrt and loss")

# Generalized sqrt --------------------------------------
```

```r
test_that("Returns complex number", {
  expect_is(general_sqrt(-2), 'complex')
  expect_is(general_sqrt(2), 'complex')
  expect_is(general_sqrt(0), 'complex')
})

test_that("Returns correct sqrt", {
  expect_equal(general_sqrt(-1.53), complex(real = 0, imaginary = sqrt(1.53)))
  expect_equal(general_sqrt(-2), complex(real = 0, imaginary = sqrt(2)))
})

test_that("Warnings for vectors of length > 1", {
  expect_warning(general_sqrt(c(2, 0)))
  expect_warning(general_sqrt(c(-2, 0, 2)), 'NaNs produced')
})

# Loss -------------------------------------------------

test_that("Returns correct loss", {
  expect_equal(loss(0, 3), 9)
  expect_equal(loss(c(1, 1, 1), c(1, 2, 3)), c(0, 1, 4))
  expect_equal(loss(c(-1, -5, -2), c(0, 0, 0)), c(1, 25, 4))
})
```

Each test file, as demonstrated above, needs to load the package of interest (using `library()` is fine) and also should supply a string which succinctly describes the general purpose of all of the tests in the test file to the `context()` function.

You can run all of the tests in the test/testthat directory by running the following `devtools` function:

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

If any test throws an error, R will report two things. First, it will report the string given in the test which was given to the `test_that()` function call. Second, it will report the filename of the test file as well as the line of code that threw an error. For example, running the above tests yields the following result:

```
  testthat results ================================================
==
  OK: 9 SKIPPED: 0 FAILED: 1
  1. Failure: Warnings for vectors of length > 1 (@testhelpers.R#18)

  Error: testthat unit tests failed
  Execution halted
```

This indicates that line 18 of testhelpers.R failed in the test "Warnings for vectors of length > 1." Looking at the test code reveals that the `general_sqrt()` function does not return a warning for positive vectors of length greater than one.

```
expect_warning(general_sqrt(c(2, 0)))
```

To fix this, it might be worth adding in an extra line or two which ensures that the input to `general_sqrt()` is as it should be (to prevent users from getting unexpected results).

1.6.6 Writing Good Tests

Good tests have a couple of characteristics.

First, good tests have high *coverage,* meaning that they test a large percentage of the lines of code of the package. For example, the code for the `general_sqrt()` function is as follows:

```
# This function takes the complex square root of real numbers

general_sqrt <- function (x){

  # Issue warning for longer vectors
  if (length(x) > 1) {
    warning('Argument of general_sqrt has length greater than 1')
  }

  # Return the normal square root if x > 0
  if (x > 0 || x == 0){
    return(complex(real = sqrt(x), imaginary = 0))
  }

  # Else return the complex square root

  else {
    return(complex(real = 0, imaginary = sqrt(-x)))
  }

}
```

The following test has low coverage for the `general_sqrt()` function:

```
test_that("Returns correct sqrt", {
  expect_equal(general_sqrt(-1.53), complex(real = 0, imaginary = sqrt(1.53)))
  expect_equal(general_sqrt(-2), complex(real = 0, imaginary = sqrt(2)))
})
```

because it only tests whether `general_sqrt()` returns the correct square root for negative numbers. This test thus only covers half of the code in `general_sqrt()`, because the mechanism for dealing with nonnegative numbers is entirely separate. The following test is a better example, because it tests both positive and negative numbers.

```
test_that("Returns correct sqrt", {
  expect_equal(general_sqrt(1.53), complex(real = sqrt(1.53), imaginary = 0))
  expect_equal(general_sqrt(-2), complex(real = 0, imaginary = sqrt(2)))
})
```

Second, it's important to remember that coverage is only important because tests with high coverage tend to test all the different functionalities of a package. It's possible to have tests which have very high coverage but aren't great tests. Consider the following example.

```
print_it <- function(text){
  print('hello')
}


testthat::expect_warning(print_it('hi'), NA)
```

```
## [1] "hello"
```

This expectation has 100% coverage because it will run every line of code (the expectation will also pass because no warning will be thrown). However, it's not sufficient alone because it doesn't actually test whether print_it returns the desired output: in this case, print_it will always print 'hello'. In other words, the expectation does not test all of the functionality of the function.

Third, tests should run relatively quickly, if possible. Sometimes, it's okay to maximize coverage even if you don't test every single functionality to save time, because *usually* high coverage ensures you test most of the functionality of the package. This is particularly true because lots of integrated testing software (which we'll discuss in integrated tests) will not be able to easily run tests which take too long. More on that later.

Fourth, tests should be clear to the reader, because sometimes there are bugs in tests too. If others eventually help develop or maintain your packages, they'll want to know what it means when a test fails. Moreover, for large packages, you yourself may have trouble remembering the exact details of every test you've written. Thus, your tests should return clear error messages and be readable. For example, the following test is a bad example, for two reasons:

```
sigmoid <- function(x, a, b){
  return(exp(a*x)/(exp(a*x) + b))
}


test_that('sigmoid output', {
  expect_equal(sigmoid(0.3068528, 1, 1.3591409), 0.5, 10^-7)
})
```

The string 'sigmoid output' does not describe the purpose of the test, which is to test the precision of the sigmoid output. This means that if the test fails, it will be hard to tell what's wrong. Additionally, the purpose of the test is not clear to begin with - what do the seemingly random decimals mean? At the very least, it's probably worth putting comments in explaining the point of the test, as shown below.

```
sigmoid <- function(x, a, b){
  return(exp(a*x)/(exp(a*x) + b))
}


test_that('test sigmoid precision', {
```

```
  # Check sigmoid(ln(e/2), ln(e/2), e/2) is very close to 1/2.

  expect_equal(sigmoid(0.3068528, 1, 1.3591409), 0.5, 10^-7)
})
```

This test is a bit more interpretable/readable and delivers a better error message.

1.6.7 Automated Checking

The `usethis::test()` function is pretty nice, but all it does is run your unit tests - it doesn't check everything else in your package. Thankfully, the `devtools::check()` function fills this gap.

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

Running the check function will ensure your documentation is up to date, automatically run all of your unit tests, and even check your code for common problems. It will also create a new directory called 'Man' within your package folder, which will later be populated with the help files for your functions. Note that even if your package passes all of its tests, you might still see additional warnings for other reasons, as exemplified below:

```
checking data for ASCII and uncompressed saves ... WARNING
  Warning: package needs dependence on R (>= 2.10)

checking R code for possible problems ... NOTE
scalep: no visible global function definition for 'hist'
Undefined global functions or variables:
  hist
Consider adding
  importFrom("graphics", "hist")
to your NAMESPACE file.
R CMD check results
0 errors | 2 warnings | 1 note

R CMD check succeeded
```

Although it's not necessary to understand every check that the check function runs, it's worth noting that every check it performs is relatively important, and if it signals any warnings or errors, it's definitely worth fixing them. It's also probably worth fixing any "notes" it issues. If you're curious, you can read more about what each type of check in the automated check does here.

1.6.8 Bonus: The goodpractice Package

The aforementioned automated checking system for R is pretty good, but there is a slightly more comprehensive version: the goodpractice package. The `goodpractice` package has an informative name - the entire purpose of the package is to check whether your package follows proper package development conventions and procedures (i.e. whether your package follows good practices). For example, running the `goodpractice` package on the `devex` package yielded the following helpful results:

```
It is good practice to

  ✖ omit "Date" in DESCRIPTION. It is not required and it
    gets invalid quite often. A build date will be added to the package
    when you perform `R CMD build` on it.
  ✖ add a "URL" field to DESCRIPTION. It helps users find
    information about your package online. If your package does not
    have a homepage, add an URL to GitHub, or the CRAN package package
    page.
  ✖ add a "BugReports" field to DESCRIPTION, and point it to
    a bug tracker. Many online code hosting services provide bug
    trackers for free, https://github.com, https://gitlab.com, etc.
```

Let's run through the key points of using `goodpractice` below.

To install the goodpractice package, just run the following command in the R console:

```
install.packages('goodpractice')
```

Once you've installed the `goodpractice` package, you basically only need to use a single function from it: the `gp()` or `goodpractice()` function (which call the same code). This function will run comprehensive automated checks on your package, and it takes exactly one input, the path of your built package. However, it's important to note that the path of your *built* package is not identical to the path of the repo containing it. For example, I work on the `devex` package in "/Users/name/Documents/R/packageguidelines/devex", but the built package lives at another location. For `goodpractice()` to work, you need to use this latter path. If you don't know what the path is, you can use the "system.file" function to retrieve it for you, as demonstrated below.

```
# Retrieve path
package_path <- system.file(package = 'devex')

# Check package
library(goodpractice)
gp(package_path)
```

And that's it! The `goodpractice` package is a bit picky, so it's okay to leave a few concerns unresolved, but in general it gives good advice. If you're diligent, you might eventually see a result like this:

```
♥ Mhm! Top-notch package! Keep up the fantastic work!
```

which signals that your package conforms to all good practices. Lastly, although it's beyond the scope of this guide, you ought to know that you can create custom checks using the `goodpractice` package, as documented here.

[up to here 2022]

1.6.9 Tips and Tricks

When testing, there are a couple of key principles to keep in mind:

1. You want to expose bugs as quickly as possible so they don't create even larger headaches down the road! As Christopher Gandrud puts it, testing is all about 'failing faster.' To this end, you should continuously test your packages.
2. Make sure your tests *cover* the package code and also test all of the key functionality of the package. In an ideal world, a package should pass all of its tests only if all of its core functionality is bug-free.
3. Test names, organization, and error messages must be descriptive and easy to understand. One of the main purposes of tests is to inform you *where* your code is failing, and to understand that, you need informative error messages. Otherwise, you will find yourself spending hours traversing your code to find bugs.

The `devtools` cheatsheet, linked here, references a lot of the key components of the `testthat` package.

1.7 Documentation

Documentation is an absolutely essential part of any package - most people won't be willing to read your source code to figure out how your functions work. Thankfully, creating documentation for your package is incredibly easy with `Roxygen2`.

1.7.1 Documenting Functions

(Note: here, we describe how to document functions. For a more detailed description of how to document S3, S4, and reference classes, check out this page.)

To understand how to use `Roxygen2`, it's best to start with an example. Consider the following code, which generates the documentation for the `general_sqrt()` function above.

```
#' Generalized Square Roots
#'
#' \code{general_sqrt} returns the square root of any real number
#'
#' @param x A real number - integers or doubles are both acceptable.
#' @return A complex value. If x is positive, the imaginary component is equal to 0;
#' if x is negative, the real component is equal to 0.
#'
#' @examples
#' general_sqrt(10)
#' general_sqrt(-10)
#' general_sqrt(-1)
#'
#' @export
# This function takes the complex square root of real numbers

general_sqrt <- function (x) {

  # Return the normal square root if x > 0
  if (x > 0 || x == 0) {
    return(complex(real=sqrt(x), imaginary=0))
```

```
  }

  # Else return the complex square root

  else {
    return(complex(real = 0, imaginary = sqrt(-x)))
  }

}
```


general_sqrt {devex}   R Documentation

Generalized Square Roots

Description

general_sqrt returns the square root of any real number

Usage

general_sqrt(x)

Arguments

x   A real number - integers or doubles are both acceptable.

Value

A complex value. If x is positive, the imaginary component is equal to 0; if x is negative, the real component is equal to 0.

Examples

```
general_sqrt(10)
general_sqrt(-10)
general_sqrt(-1)
```
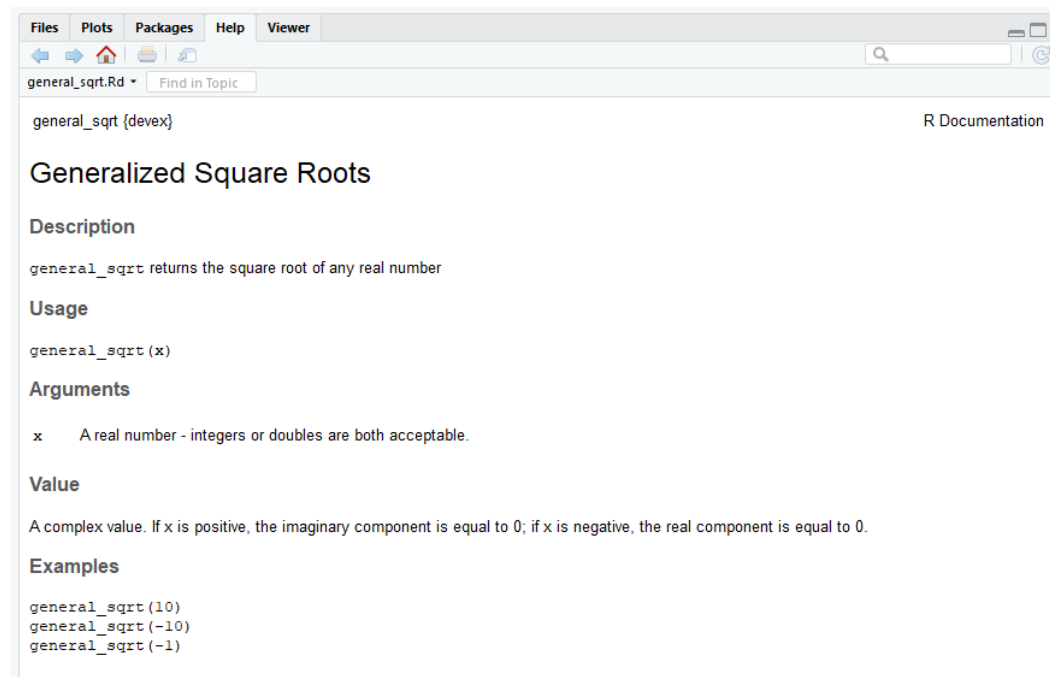
Inside the R script, the documentation appears right above the definition of the function - this is helpful because it will help you remember to keep your documentations/functions up to date with each other. Let's work through how this documentation was generated.

1. Note that everything you write in `Roxygen2` should be preceded by a #' character combo. This signals to the R environment that you're writing documentation, not code.

2. You should start your documentation with a very short (2-4 word) title of the function. In the example given above, the `general_sqrt()` function is titled "Generalized Square Roots."

3. You should describe your functions' parameters using the '**?**' signifier. This should succinctly describe the type (i.e. double, integer, character) of the parameter, as well as its function, as well as any potential default value.

4. You should document the type of output your function returns. Is it an integer, a dataframe, a matrix? Does it depend on the input? Your documentation should

answer these questions!

5. You *must* provide examples of your function's use. These are pretty critical, because a lot of programmers will just skip straight to the examples (and only look at the rest of the documentation if the examples are unclear).

6. You may choose to write '**?**' at the end of your documentation block. You should only do this if you want other people to use the function that you're exporting, because exporting it will make sure it shows up in the **namespace**, a document that makes sure your package works in combination with other packages. For example, if your package includes data labelled 'lm', the namespace will prevent errors when using that data in combination with the R stats package (which has a function called 'lm') and will *not* throw an error. The exact mechanics of the namespace are slightly beyond the scope of this guide, but thankfully `Roxygen2` will automatically generate a namespace for you when you create documentation with "**?**" tags.

Note that most functions you write won't be exported - for example, if you write a helper function like `loss()` which is only used in service of a larger function, it shouldn't be exported (exporting too many functions 'clutters' the namespace).

Once you've written all your documentation, it's fairly simple to check what it looks like. Simply run the following function:

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

which will automatically generate your documentation. Then, if you've documented a function, you can type `?function-name` into the console, and the documentation should automatically pop up!

After documenting your package, you can also click the '/man' folder to inspect the documentation html files `Roxygen2` generates, but it probably won't be more informative than simply typing `?function-name` into the console.

1.7.2 Adding a README

The README file is a bit different than the others, because your package will actually work fine even if you don't have one. However, if you want other people to use your package, it's best to have a README. The purpose of a README is basically to bridge the gap between the DESCRIPTION and the actual documentation in your package. In other words, someone using your package might know what it does in a general sense from your DESCRIPTION, but they won't necessarily know exactly how to set up the package or how to use specific functions. The README takes care of that. In general, READMEs should do at least two things, with a couple of optional ones:

(Important):

1. Offer a longer (one to three paragraph) description of the package, including core functions and bits and pieces of syntax
2. Help users install and set up the package

(Optional from here on):

3. Tell developers what to do if they want to contribute to your package

4. Help contributors figure out how to run the packages' unit tests (we'll talk more about unit tests in the next section)

5. Offer some acknowledgements

This template README is a good starting point. You could copy this file into your package directory and modify it to reflect the content of your project. Again, because your package will technically function without your README, the actual structure and content of a README can be flexible. However, just remember that if you don't have a README which outlines why and how to use your package, other developers are unlikely to want to use it.

1.8 Vignettes

Documentation is useful, but not necessarily a comprehensive guide to your package. You may want to include details about your implementation, extra examples, and more organization than your documentation provides, which is exactly what vignettes are for.

Vignettes are basically articles which motivate and describe your packages. They are generally written in RMarkdown, which allows you to mix code, mathematical equations, and formatted text with ease. If you are using RStudio, then writing vignettes will be very easy, because RMarkdown works automatically with RStudio. On the other hand, if you don't have RStudio, you will need to (i) run the 'install.packages("rmarkdown")' command, and (ii) install pandoc.

To write a vignette, start by running the `use_vignette()` function from the `usethis` package:

```
usethis::use_vignette('vignette-name')
```

This function will create a 'vignettes' subdirectory and populate it with a file based on the name you specified. The file should look something like this:

example-vignette.Rmd ×

Knit ▾ ⚙ ▾ Insert ▾ ⬆ ⬇ Run ▾

```
 1  ---
 2  title: "Vignette Title"
 3  author: "Vignette Author"
 4  date: "`r Sys.Date()`"
 5  output: rmarkdown::html_vignette
 6  vignette: >
 7    %\VignetteIndexEntry{Vignette Title}
 8    %\VignetteEngine{knitr::rmarkdown}
 9    %\VignetteEncoding{UTF-8}
10  ---
11
12  ```{r setup, include = FALSE}
13  knitr::opts_chunk$set(
14    collapse = TRUE,
15    comment = "#>"
16  )
17  ```
18
19  Vignettes are long form documentation commonly
    included in packages. Because they are part of the
    distribution of the package, they need to be as
    compact as possible. The `html_vignette` output type
    provides a custom style sheet (and tweaks some
    options) to ensure that the resulting html is as
    small as possible. The `html_vignette` format:
20
21  - Never uses retina figures
22  - Has a smaller default figure size
23  - Uses a custom CSS stylesheet instead of the
    default Twitter Bootstrap style
```

12:30   C Chunk 1: setup ⇕   R Markdown

The top of the file, between the two lines of dashes, is written in the YAML language. It's simply a convenient way to specify metadata about a vignette, and you should fill in the title and author fields.

The rest of the vignette should be written in RMarkdown, which is basically a mix of Markdown, Latex, or code. The vignette template generated by `usethis` *is itself a guide to using RMarkdown*, so we won't dive too deep into using RMarkdown. However, there are a couple of core things you should know:

1. By default, text in RMarkdown files is assumed to be written in pandoc's flavor of Markdown.

2. If you would like to include inline equations, you can do so by surrounding math symbols with a single dollar sign on each end. If you'd like to give an equation its own line, you can use two dollar signs on each end of the equations.

```
$$ \int_k e^{- \lambda k^2} \times \begin{bmatrix}
x_1 & 0 \\ 0 & x_4 \end{bmatrix} $$
```

$$ \int_k e^{-\lambda k^2} \times \begin{bmatrix} x_1 & 0 \\ 0 & x_4 \end{bmatrix} $$

Figure 1.1: *Including math is easy in RMarkdown*

3. You can add chunks of R code (and even other languages!) to your Vignette file by wrapping R code in "```" symbols, as demonstrated below:

```{r}
# Writing code inside RMarkdown is easy!
library(ggplot2)
mtcars <- mtcars
g <- ggplot(data = mtcars, aes(x = mpg))
print(g + geom_histogram(binwidth = 3, fill = 'cornflowerblue'))
```
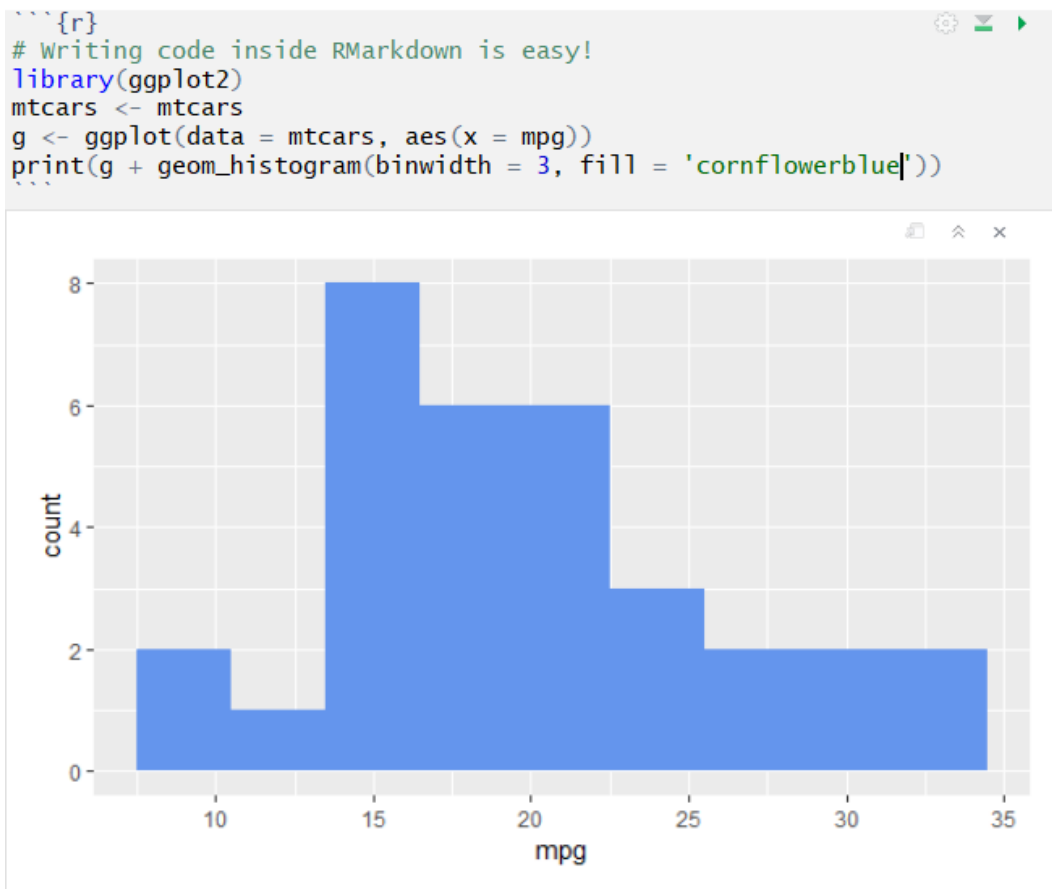

Once you've modified your vignette, you can *knit* it into a beautiful HTML document by

running the following function in the console:

```
rmarkdown::render('path/to/rmarkdown/file.Rmd')
```

If you do choose to write vignettes, remember that it's critical to motivate *why* you wrote your package in the first place. Additionally, you should structure the vignette so that it gives users an impression of the overarching structure of the package itself (i.e. group and organize your functions!).

1.9 Adding and Documenting Data

Sometimes, you'll want to include data as part of your package, either to serve as an example for users or because your functions need it to work. This is totally optional - not all packages need to include data - but can be useful, so let's walk through how to include (and document) data in your package. Note that there are at least two kinds of data you should think about including, but for both kinds of data, you should generally save them as '.rdata' or '.rda' files (which are the same thing).

1.9.1 Including data which should be available to users

All of the data you want to be available to users should be saved in a folder called 'data' inside your project. The way to do this is to write and run a script which loads your data into R and then uses the `usethis` function `use_data()` to save it to a path inside 'data'. Even if you do not already have a folder called 'data' in your package, the `use_data()` function will make it for you. Note that for this to work, your working directory must be set to the package you're writing, otherwise `usethis` won't know where to put your data.

For example, when documenting the `scalep()` function, one might want to include data as an example of a way to use `scalep()`. To this end, we can use the built-in R dataset `mtcars` and include it in the package.

```
usethis::use_data(mtcars)
```

In this way, you can include in your package any data you want, providing it has been stored in an R object. Data in your 'data' folder will effectively always be exported, so you always must document it. To document data, you should create an R Script in your 'R' directory called ['data.R'] and use `Roxygen2` to document the data similarly to the way you'd document a function. For example, one might document the aforementioned data in the following way:

```
#' Motor trend car road tests dataset
#'
#' This dataset lists the properties of 32 motorcars
#'
#' @format A dataframe with 32 rows and 11 columns.
#' \describe{
#'   \item{mpg}{Mile per gallon}
#'   \item{cyl}{Number of cylinders}
#'   \item{disp}{Displacement (cu.in)}
#'   \item{hp}{Gross horsepower}
```

```
#'   \item{drat}{Rear axle ratio}
#'   \item{wt}{Weight (1000 lbs)}
#'   \item{qsec}{1/4 mile time}
#'   \item{vs}{Engine: (0 = V-shaped, 1 = straight)}
#'   \item{am}{Transmission: (0 = automatic, 1 = manual)}
#'   \item{gear}{Number of forward gears}
#'   \item{carb}{Number of carburetors}
#'}
#' @source This dataset is a built-in R dataset and is
#' intended only to be used as an example for package development.
"mtcars"
```

This should all look pretty similar to documenting functions. Note that '**?**' is a tag which will allow you to describe the structure of a dataset, and it's good practice to list what each column measures in this section. The '**?**' section describes where the data came from. Never write '**?**' in this section, as data here is already automatically exported.

1.9.2   Including data for your functions

Some functions may rely on a large, predefined set of coefficients or other inputs which need to be included in the package. However, users shouldn't generally have access to such data because otherwise they might accidentally radically change the way your function works. It's best to put such data in 'R/sysdata.rda', because then users won't easily be able to access and accidentally modify it. As before, the way to include data in this way is to write a script which loads the data into R and then use the `use_data()` function to save it, but you should also include a parameter `internal = TRUE` in the function call to let R know that this is interior, not exterior, data. For example, if a function depends on a matrix called "coefficients", one might run the following code:

```
coefficients <- read.csv('Users/name/Documents/R/coefs.csv')
usethis::use_data(coefficients, internal = TRUE)
```

Data in 'R/sysdata.rda' is never exported, so there's no need to document it.

1.10   Releasing Your Package

You're almost done at this point! You've written your functions, modified the description, documented your functions, presumably exported some of them, and hopefully tested all of them; you're now ready to *release* and *publicize* your package. There are basically two main ways to do this.

1.10.1   Pushing to GitHub

The easiest way to publish your package is to simply push it to GitHub, (we'll discuss how to do this later). The pros of this approach are that it makes it very easy for users to download your package - they can literally do it in one line. Additionally, Github offers free services to host a website for your package and its documentation, and most importantly,

it's very easy for **you** to publish your package this way. Thus, pushing to Github is sort of the "default" way to publish a smaller package.

1.10.2   CRAN

On the other hand, if you've written a larger package which you would like to distribute to the entire R community, you might consider submitting it to CRAN, the Comprehensive R Archive Network. CRAN is basically the official package authority designated by the R community, and successfully adding your package to CRAN will make it more legitimate as well as easier for R users to find and install.

Logistically, submitting your package to CRAN is pretty simple. The first step is to **build** your package, which means bundling it into a format that is easy to distribute and easy for users to install.

The best way to build your package is to zip it as a .tar.gz file. `devtools` will do this for you if you run the following command:

```
devtools::build(binary = FALSE)
```

and then you should see a tar.gz file pop up just outside your working directory.

The next step is to submit the bundled file to CRAN at the link here, along with a couple of comments. Although this seems pretty simple, in actuality, CRAN has very high standards for packages, so it can be rather tricky to get a package accepted. CRAN's specific standards are beyond the scope of the current iteration of this guide, but if you decide you want to publish a package on CRAN, you should read Hadley Wickham's advice on the subject **very carefully.**

1.10.2.1   Optional: Building precompiled binaries

tar.gz files are useful because anyone who has a working R development environment can install and unzip your package, regardless of their operating system. However, you do need a development environment to install packages built as tar.gz files, and some users (in particular on Windows) may not have development environments set up yet. To address this potential issue, another way to build your package is as a **precompiled binary file**. Precompiled binaries are useful because unlike tar.gz files, they do not require a development environment to install. However, they are platform specific: a precompiled binary built by a Windows machine can't be installed on Mac machine. Although tar.gz files are much more common, if you do want to build a binary, you can just change an argument of `devtools::build()`:

```
devtools::build(binary = TRUE)
```

and your precompiled binary will be built.

1.10.3   Publicizing

Once you've released your package either on Github or perhaps on CRAN, you should publicize it! You can of course publicize it any way you choose, but there are at least two things you should consider doing.

1. Tweet about your package using the #rstats hashtag, which reaches a substantial portion of the R community.
2. You may also want to send an email out to the R-Packages email list.

And that's it!

1.11   Tips and Tricks

First, there's a wonderful cheat sheet for package development linked here.

Second, if you're having trouble, you can always just reference stackoverflow.

Chapter 2

Version Control

Version Control is the process of organizing and sharing different versions of your code, and it's surprisingly important. This chapter is a beginner-friendly introduction to Version Control with Git and GitHub. In it, we'll motivate Git/GitHub, and then discuss what precisely Git/GitHub do and how to use them.

There are other tools for Version Control out there, but Git/GitHub are by far the most common, so we've chosen to focus on them. As you might expect, much (but not all) of the logic behind GitHub applies to other version control systems too.

2.1   Why use Git/GitHub?

Git and GitHub are actually two different things, and deserve their own sections. Before diving into specifics, however, we should talk about why Git/GitHub exist in the first place. Broadly, Git and GitHub are just tools that developers use to organize their code, which specifically solve three main organizational problems.

First, imagine that two imaginary programmers named Carlos and Grace are working together to build a website that automatically displays polling results from US elections, and suppose Grace modifies a script to change the method for aggregating polling data. It would be tedious for Carlos and Grace to email the script back and forth each time they updated it. Instead, they need a *repository* where they can update and sync their code to make sure they're on the same page. GitHub allows programmers to create such repositories.

Second, imagine that farther along in the project, Grace decides that she needs to totally rewrite the code which stores and organizes polling data. Unfortunately, while Grace modifies the system which stores the data, it will be offline, which is unfortunate because Carlos needs to use the data for his work on the project. To avoid this dilemma, Git and GitHub allow programmers to create different versions, called *branches*, of their code. By branching, Grace can modify the storage system, and in the meantime, Carlos can continue to use the old storage system for his work.

Third, suppose that once Grace and Carlos have finished their website, they want to publish all the statistical functions and tools they built so that other researchers can use those tools

to reproduce Grace and Carlos's research. Putting all of their work on GitHub is probably the quickest and easiest way to make it accessible to other researchers.

2.1.1 What is Git?
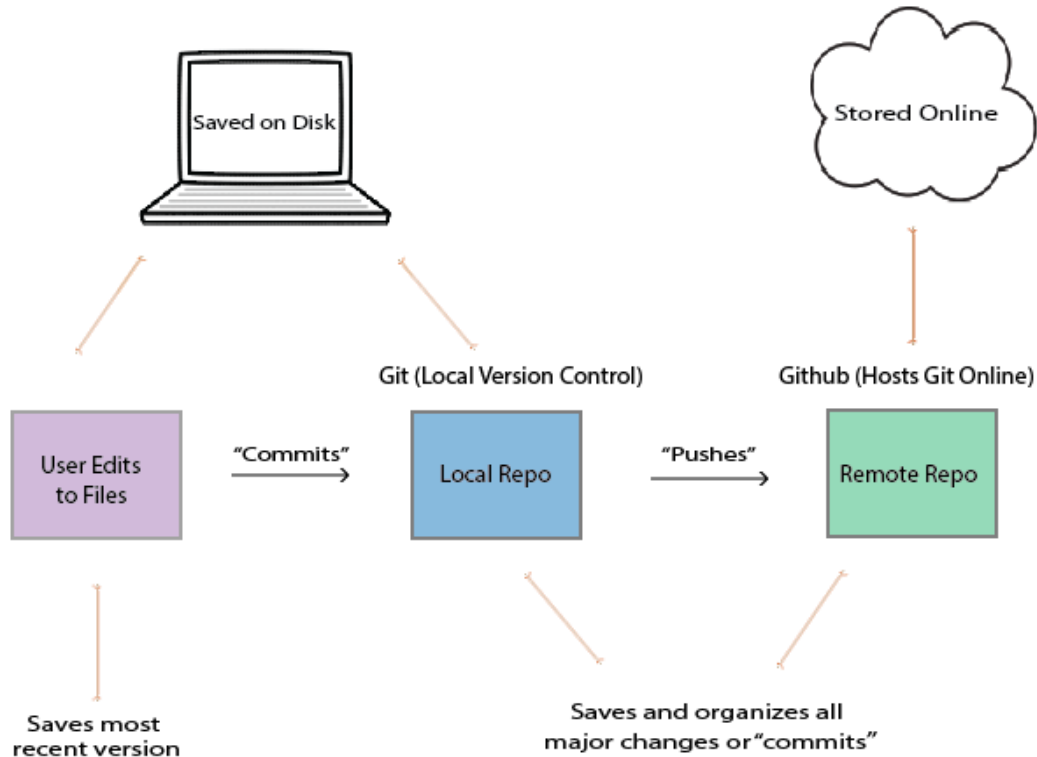
Git is a local version control system. In other words, Git automatically organizes and saves versions of code on your computer, but does not connect to the internet: it's purely accessible through the command prompt on a computer (there are also other ways to use Git, but it's generally best to use the command prompt). Git organizes code in two ways.

First, Git sends discrete versions of code to a local database or repository on the computer on which it is installed. To understand this, imagine that Carlos is writing an R script which determines the proper weights for aggregating polling data, and at some point he finishes the first version of the script. At this point, Carlos can send his code to the local repository, in an action called *commiting* his script. We'll discuss how to do this later, but note that scripts are not automatically committed to the repository. If Carlos deleted all the code in the script and saved it, he could still access the script by going into the repository. Moreover, the repository also stores all of the previous versions of code, and can backtrack through the various commits, so if Carlos decides the version of the script he built last year worked better, he can use Git to access it.

Second, Git can create branches, or separate versions of code, on the computer it's installed to. Remember how Grace wanted to revamp the storage system for polling data without preventing Carlos from accessing the old storage system? Using Git, Grace should create a

new *branch* in the repository. For most practical purposes, each branch exists as a separate version of the entire polling project in the Git repository, allowing Grace to modify and experiment without compromising the functionality of the *master* or original branch.

While she's working, Grace can use Git to move back and forth between the branches. Then, when she finishes designing and testing the new storage system, she can *merge* the two branches together, and Carlos's scripts will automatically start using the new storage system.

2.1.2 What is GitHub?

Where Git creates *local* repositories on an individual computer, GitHub allows programmers to create *remote* repositories, stored online, for their code. This is a pretty simple concept, but it's extroardinarily useful. It means that when Grace finishes updating a series of scripts, she can simply *push* (push means send) them to GitHub, and then Carlos can in one command *pull* (download) the modified versions onto his computer to work with them. GitHub provides other important tools, but we will not discuss those until later.

2.2 Setting up Git/GitHub

2.2.1 Making a GitHub account

Signing up for a GitHub is fairly simple - just go to GitHub.com, click "sign up" in the upper right hand corner, and follow the instructions. We recommend, at least to begin with, getting a free version of GitHub.

2.2.2 How to Install and Set Up Git

Because Git is a form of *local* source control, it does require installation. To install Git onto your computer, go to one of the following sites:

- For Linux: Head to `https://git-scm.com/download/linux` and follow the instructions.

- For Windows: Go to `https://git-scm.com/download/win` and the download should automatically start.

- For Mac: Click on `https://git-scm.com/download/mac` and the download should automatically start.

If you are running Windows/Mac and the websites above download an installer, simply follow the instructions from the installer - it's fine to just use default settings.

Before you can start using Git, you do have to do a one time set up. Git is built to track and organize old versions of code, so it will need you to set your username and email before it will let you start modifying programs. To set your username and email after installing, open the command line on your computer and type the following commands into it:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "youremail@email.com"
```

You might not get explicit confirmation from the command line - for example, when I set the username/email, I see the image below - but don't worry, Git has still received the message.

Now you're ready to start using Git/GitHub!

2.3   Using Git/GitHub

2.3.1   Initializing Projects and Repositories

The first step to using Git is creating a project, or repository (repo for short). To do so, log into your GitHub account. Look near the top-right for your profile icon, click it, and click "Your Profile." Then you should see "Repositories" in the top tab, and click on "Repositories". Then click the green "New" button on the right.
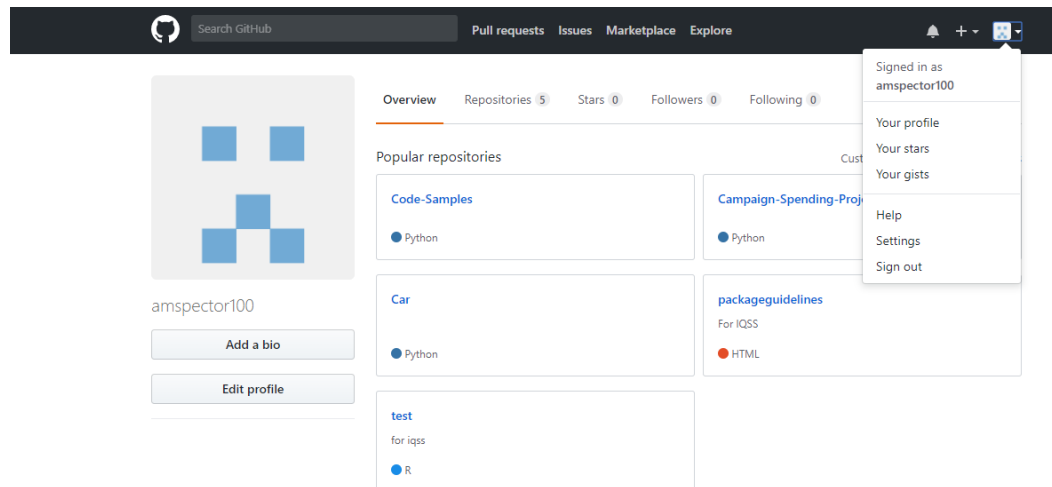
GitHub will prompt you to choose a repository name and description of the repo, which you should enter, and will also ask you whether you'd like to initialize the repository with a README, a .gitignore file, and a license. For now, just initialize the repo with the README file.

Once you've created the repository on GitHub, the next step is to link it with a local Git repository. To do this, open the *command prompt* on your computer. You'll need to enter a couple of commands to get the repo set up on your computer.

First, you'll need to set the *working directory* in the command prompt. The working directory is the location where Git and command prompt will execute all of your commands -

ie. if you tell Git to create a new file, the new file will be located your working directory. Thus, before you download your repository, you'll want to navigate to the folder in which you want the repository to be located. For example, Carlos might want to put a repo in his "R" folder inside the "Documents" folder on his computer. To do this, he'd use the `cd` command, which stands for "change directory," and then copy and paste the path of the folder he'd like to place the repository in. An example is given below:

```
$ cd ~/Documents/R
```

Next, you will need to tell Git to download, or *clone*, the repo. You can do this using the `git clone` command, which allows you to download and sync a repository *for the first time*. This will download the repo, with all its contents, onto your computer and automatically set up a local version of the repo. See an example of the command below:

```
$ git clone https://GitHub.com/Carlos_Account/repository_name/
```

2.3.2 Modifying Projects

It's best practice to create new R code files in your local project repository. When you save your code files, initially git will not be aware of them - that is, they will not be *tracked*, but we can change this by *adding* them to the local repository using the command `git add .`. Here, the period following `git add` tells Git that we want to start tracking all the files in the directory.

```
$ cd ~/Documents/R/repository_name
$ git add .
```

Once new files are being tracked, we can put them under version control by *commiting* them, using the command `git commit`. You can add a message to indicate the changes you've made by typing `git commit -m "<message-here>"` followed by your message in quotes.

```
$ cd ~/Documents/R/repository_name
$ git commit -m "Fixed small bug"
```

Lastly, you need to *push* the files, which means sending them to GitHub online. You can do this by writing `git push origin master`, which will send the file to the remote GitHub repo.

```
$ git push origin master
```

If you move new files into your local repository, they will need to be tracked using `git add .` and then committed using `git commit -m "<message-here>"` before they are under version control.

2.3.3 Accessing older versions of code

You should never be scared of Git/GitHub, because they only add information - they almost never delete it. In practice, this means that if you commit horrible changes to your code, you can easily revert to a previous version. For example, let's say Grace wrote some new functions for a script, deleted them and committed the deletes later on, but now wants the functions back. She can easily access the older script using Git. First, she should use the `git log` command to see what commits have been made recently.

```
$ git log
```

This command might give an output like this:

```
commit 06c347sa (HEAD -> master, origin/master, origin/HEAD)
Author: Grace Smith <gracesmith@gmail.com>
Date:   Sun Feb 18 23:13:32 2018 -0500

    Remove storage functions

commit b69a14b
Author: Grace Smith <gracesmith@gmail.com>
Date:   Sat Feb 17 23:11:28 2018 -0500

    Add storage functions

commit 8a237d1
Author: Grace Smith <gracesmith@gmail.com>
Date:   Fri Feb 16 21:28:46 2018 -0500

    Initial commit
```

Clearly, Grace should restore the second commit listed, named 'b69a14b,' in order to access the functions she deleted. There are generally two situations:

1. If the changes have not been pushed onto the remote repository, to access the deleted functions, Grace can use the `git reset` command with the `--hard` option and the commit number: `git reset --hard b69a14b`. This will update the current version of the script to the `b69a14b` version, which Grace can then inspect and test until she's ready to commit it to Git/GitHub. Note that this is a pretty serious action to take because it will effectively delete all the commits after the version being restored.

2. If the changes have been pushed onto the remote repository, Grace can access the deleted functions using the following commands:

```
$ git revert --no-commit head
$ git commit -m "<message-here>"
```

This will update the current version of the script to the `b69a14b` version in Grace's local repository. Then Grace can push it onto the remote repository using `git push origin master` command.

2.3.4   The .gitignore file

There's one more thing you should know about adding/modifying files that will make your life a lot easier. Sometimes, when modifying and commiting projects, you won't want to type out every individual file you've modified: you'll instead want to use the `git add .` command which will simply queue every modified file to be committed.

```
$ git add .
$ git commit -m "<message-here>"
```
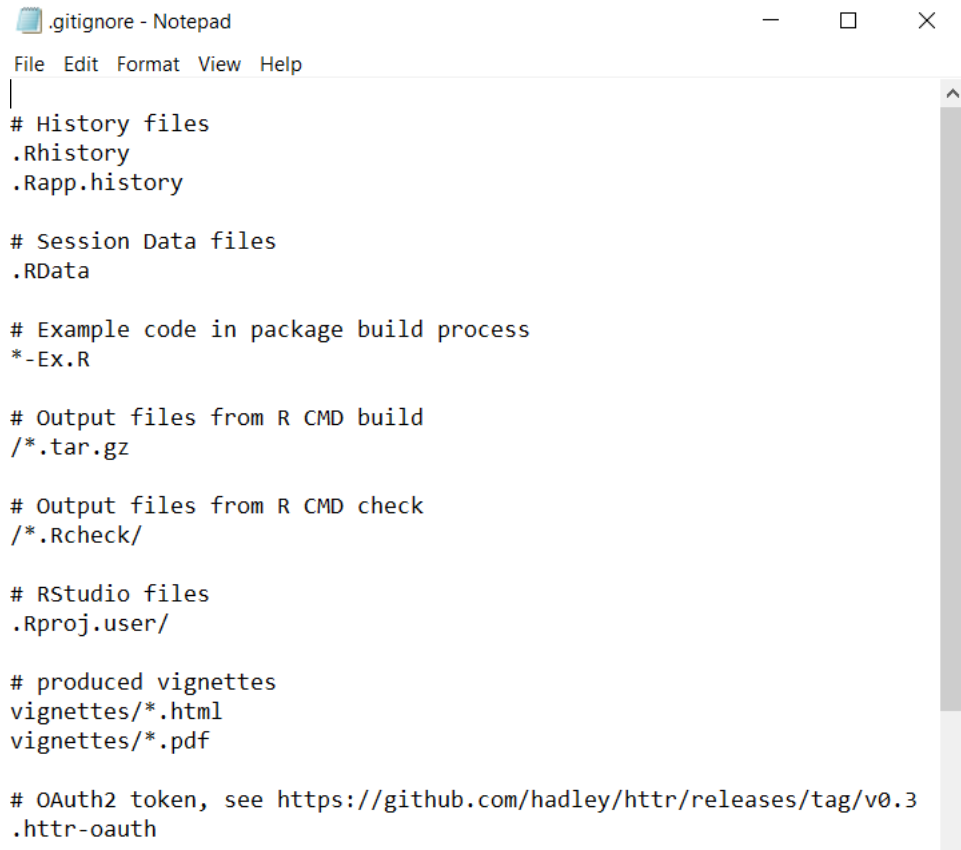
```
$ git push
```

However, often, you'll have files in your directory which you do not want to commit or push to GitHub because they'll clutter up the directory. For example, R directories create ".RHistory" and ".RData" files which will save your history and working environment. These are useful perhaps when writing code, but you don't want them taking up space in your final packaged product.

To solve this problem, you can use a *.gitignore file.* A .gitignore file tells Git to ignore certain kinds of files when you write `git add ..` For example, gitignore files are frequently configured to tell git to ignore csv files, .RData files, installer logs, and more.

Let's discuss how *.gitignore* files actually work. A .gitignore file is really just a glorified text file. However, its filename should be just '.gitignore', because that's how Git recognizes it - then, if you put a file titled '.gitignore' in a directory you're working on, Git will use the .gitignore file to decide what to ignore in that specific directory. For example, if Grace is working on a project in '~/Documents/R/repo_name', she should create a '.gitignore' file inside that folder.

This still leaves an important question - how does Git use the .gitignore file to determine which files to ignore? Essentially, .gitignore files are just a bunch of independent lines of text, and each line tells git another specific pattern of filepath to ignore. For example, consider the following example of a .gitignore file:

```
.gitignore - Notepad                                    —    □    ×

File  Edit  Format  View  Help

# History files
.Rhistory
.Rapp.history

# Session Data files
.RData

# Example code in package build process
*-Ex.R

# Output files from R CMD build
/*.tar.gz

# Output files from R CMD check
/*.Rcheck/

# RStudio files
.Rproj.user/

# produced vignettes
vignettes/*.html
vignettes/*.pdf

# OAuth2 token, see https://github.com/hadley/httr/releases/tag/v0.3
.httr-oauth
```
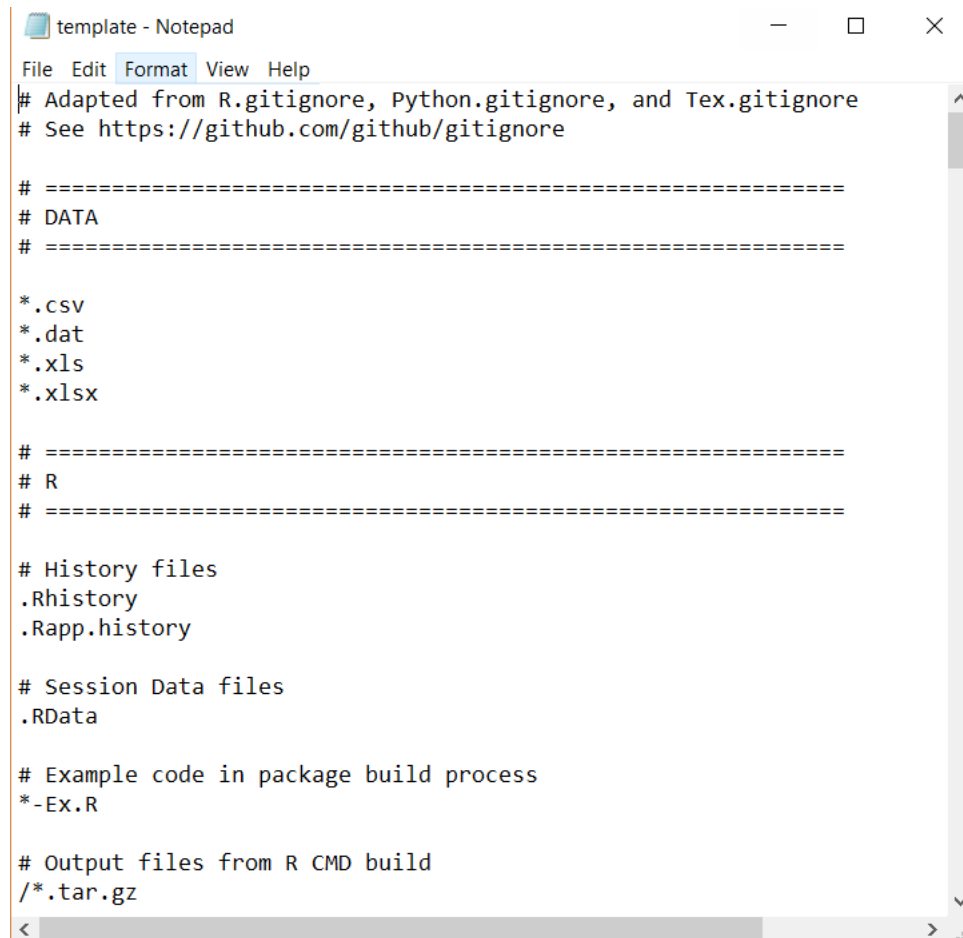
Each line beginning with a # is just a comment explaining the function of that part of the .gitignore file. However, there are a couple basic syntactical rules you need to understand.

1. If you write '.filetype', then all files which end with '.filetype' will be ignored by git. For example, the third line in the above example reads '.RHistory', so all files with a '.RHistory' in the file name (even if the files are in subdirectories) will be ignored.

2. If you write 'string', then everything in the directory which contains that 'string' in it will be ignored. For example, if Grace wrote 'dog' in her .gitignore, then git would ignore both a file called 'dog.csv' and every file in a subdirectory called 'dog'.

3. If you prepend a pattern with an asterisk *, then the asterisk * serves as a wildcard which can match 0 or more characters. For example, if I write "*car", then git will ignore the file 'mycar.R'.

4. If you prepend a pattern with a forward slash '/', then git will only ignore files which match that pattern in the parent (root) directory. For example, I might write '/.csv' in my gitignore file. This would cause git to ignore a file with the path 'dog.csv', but it would not ignore a file called 'data/dog.csv'.

5. If you prepend a pattern with an exclamation mark, git will *not* ignore that pattern. For example, if Grace really wanted to make sure her pictures of her dog were pushed to Git, she might write '!dog.PNG' to ensure git did not ignore that picture.

The rules can seem confusing at times, but it's hard to go wrong - usually, there are at least 10 ways to ignore any given file you want to ignore. Moreover, if you're still confused, you might visit Atlassian, which has a wonderful table of all of the various syntactical tricks you can use in your own gitignores, linked here.

Lastly (and most importantly), to save you time, the wonderful NaLette Brodnax has created a template .gitignore file which you can use to declutter your GitHub repos. Using it requires three steps:

Step 1: Download the file 'template.gitignore' from here. It should look something like this when you open it:

Step 2: Manually move the template file from your downloads folder to the directory of interest. For example, if Grace is working on a project in '~/Documents/R/repo_name', she should copy the template into that folder.

Step 3: Open the 'template.gitignore' file in the folder and save it just as '.gitignore'. This will help git recognize that your .gitignore file is in fact a .gitignore file. Your operating system might protest at this - Windows in particular freaks out at '.gitignore' files, which is why we initially named the template 'templates.gitignore' instead of just '.gitignore'. However, once

the .gitignore file is in the correct directory, it's safe to change the path. Your end result should look something like this:

| □📄 | 3/9/2018 7:50 PM | Text Document | 1 KB |

Figure 2.1: A .gitignore file

It has no name, but it will serve its intended purpose.

If you decide to create your own .gitignore file, you can follow very similar steps.

1. Go into the directory you want and create a blank text file called 'template.gitignore'. It's necessary at first to keep the 'template' part because otherwise some operating systems will not let you create the file.

2. Start writing your gitignore file.

3. Save the file as '.gitignore.'

.gitignore files aren't glamorous, but they really are important - nobody wants to work with cluttered repositories.

2.3.5   Branching
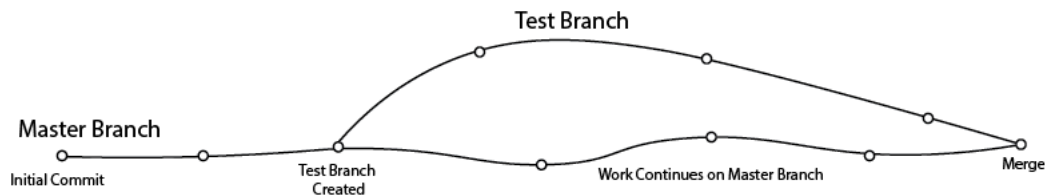
As we discussed earlier, branches allow you to modify scripts while simultaneous keeping the old versions easily accessible. In practice, the way this works is that the branch you select in git changes the appearance of the directory structure and files therein, when viewed in explorer/finder. For example, if Carlos has the following script as the master (default) branch in Git, the script might look something like this when he opens it on his computer:
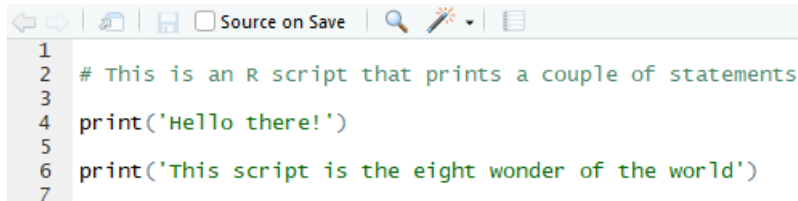

```r
# This is an R script that prints a couple of statements

print('Hello world!')

print('This script is pretty boring')
```

Figure 2.2: *Master Branch of the Hello World Script*

However, if he uses Git to switch to a test branch, he might open the exact same file using his file explorer and see the following:

Figure 2.3: *Test Branch of the Hello World Script*

Let's discuss how to actually use Git to branch.

2.3.5.0.1 Creating and Switching Branches To create a branch, use the `git checkout` command followed with a `-b` and the name of the branch:

```
$ git checkout -b branch_name
```

This will create a new branch which is identical to the initial (master) branch. You can modify it as you like, and you will still be able to easily access the initial (master) branch. To do this, simply use the `git checkout` command without the `-b` and type the name of the branch you want to switch to:

```
$ git checkout switch_to_this_branch
```

Remember, switching to a branch will change the way the file shows up once you open it from your computer.

2.3.5.0.2 Merging Branches Suppose you have created a test branch, tested it to make sure it works, and now want to combine it with the original master branch. To do this, use the `git checkout` command in combination with the `git merge` command as follows:
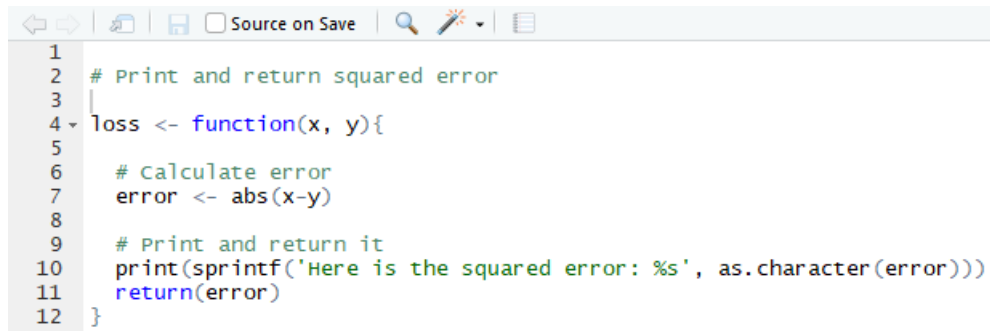
```
$ git checkout master
$ git merge test_branch_name
```

These two commands will merge the test branch into the master branch, meaning that the master branch at the end of the merging will look like the test branch.

2.3.6 Conflicts

Sometimes Git will be unable to push or pull branches because it is getting conflicting information from two users. For example, suppose Grace has written the following function, which calculates the squared error between two values:

This function clearly has a bug in it, because it claims to return the squared error, but instead, it returns the absolute value of the error. Carlos notices this, and modifies the function so that it accurately reports that it calculates the absolute value, and pushes his modified version to GitHub:
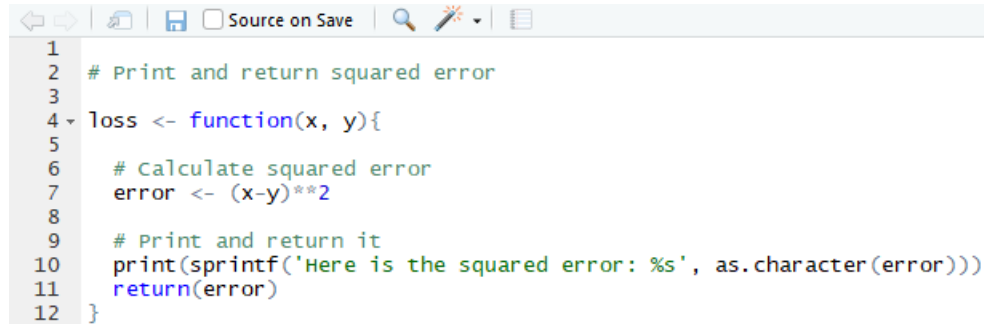
Figure 2.4: *Original Version of Loss Script*

Figure 2.5: *Carlos's Pushed Version of Loss Script*

But before Grace realizes Carlos has modified the script, she also fixes the bug, but instead by changing the absolute value to a squaring function:

Figure 2.6: *Grace's Modified Version of Loss Script*

Git is smart, so when Grace tries to push her version to GitHub, Git will recognize that the two versions of the script conflict. As a result, it will throw the following error:

```
C:\Users\gracesmith\Documents\R\repo_name>git push
To https://GitHub.com/gracesmith/repo_name
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://GitHub.com/gracesmith/repo_name'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Effectively, Git recognizes that Carlos has made modifications to the script which were pushed to GitHub that Grace doesn't have. This usually happens *when two programmers modify exactly the same line in a single script*, such as line 6 in the above examples. To solve this problem, Grace should follow Git's advice, and try the `git pull` command. This will lead to the following message from git:

```
C:\Users\gracesmith\Documents\R\repo_name>git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://GitHub.com/gracesmith/repo_name
   e719863..59fc0ba  master       -> origin/master
Auto-merging scriptname.R
CONFLICT (content): Merge conflict in scriptname.R
Automatic merge failed; fix conflicts and then commit the result.
```

Again, Git is letting Grace know that it sees there's a conflict and that Git needs human help to fix it. Once Grace has pulled Carlos's changes, she can go open the script in her computer and should see something like the following:

Figure 2.7: *Conflicted Version of Loss Script*

The `<<<<` and `>>>>` lines are in the script to signal the beginning and end of a merge conflict, whereas the `===` line separates the two different versions. On the top is Grace's version and on the bottom is Carlos's. To resolve the merge conflict, Grace should manually select the two lines she prefers and then delete all of the `<, =, >` symbols to let Git know that the conflict has been resolved. Then, she can commit and push the file to Git, and she'll be good to go!

Overall: merge conflicts are rather annoying and can be confusing, so the best way to avoid them is to clearly delineate which programmers will be working on which sections of which scripts. This will prevent Git (and you) from getting confused.

2.3.6.0.1 Suggested Workflow There are many ways to use Git, but this article by Vincent Driessen describes a pretty intuitive and productive model for Git branching. It's worth reading or skimming the article (it includes some great graphics), but it basically involves a couple of core kinds of branches:

- The **master** branch is only used for new releases of a package/software.
- There are **hotfixes** branches which are used to fix relatively important bugs on releases.
- The **development** branch is the central branch on which most of the development happens; i.e. fixing minor bugs and adding new features between releases.
- There are **feature branches** which are used to develop new features between releases, which are periodically updated with the development branch.
- Finally, there are **release branches** which are used exclusively to fix bugs pre-release.

Here's an illustration of the core model which Driessen made:
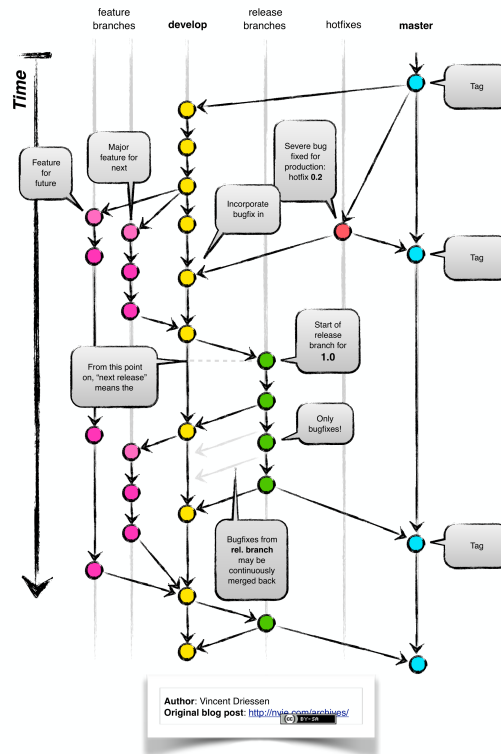
2.3.7 Tips and Tricks

Git can be confusing, but there are three general rules of thumb you can follow to figure out any bugs that come up.

1. Git is super smart. If there's a bug and Git is recommending a particular way of fixing it, there's a very high chance that Git is right.

2. Use the `git status` command. After entering `git status`, Git will give you a run-down of the status of the entire repository.

3. If that doesn't work (or you don't understand what is going on), you can always copy and paste the error from the command line into Google and reference stackoverflow. There is a wealth of online help!

Remember that Git almost exclusively *adds information*, meaning that you can always just go back and restore a previous version - you'll never lose your work.

Lastly, there are a couple of cheat sheets that you can use to make your life easier.

- Stackoverflow provides online help for programming in R

- Karl Broman's tutorial runs you through initializing a repository

- This official GitHub cheat sheet lists all of the general commands you'll need

2.4   Optional: Integrated Testing

2.4.1   Why use Integrated Tests?

When working on packages alone, you may not push to GitHub too often. However, in large projects/packages where many programmers are working on the same package at once, it's important to ensure each programmer continuously commits changes to GitHub to make sure all the changes are compatible with each other. This process of rapid updating of packages is referred to as *continuous integration*, and it can be very difficult to manage properly (it is sometimes referred to as *integration hell*, specifically because developers often waste lots of hours trying to make code integrate properly). When you commit changes continuously, it does not guarantee that all the changes will be compatible. However, if there are compatibility problems in updates, it does ensure that you'll be able to find those problems more easily, because each individual change/commit is smaller. For reasons we'll discuss in a second, continuous integration is a bit easier said than done. Thankfully, there are two wonderful free continuous integration services that will make your life a lot easier. If you're using windows, you'll want to use the service called *Appveyor* - if you're using Mac or Linux, you'll want to use the service called *Travis CI*. These services will make continuous integration easy, specifically because both Appveyor and Travis CI link to GitHub and will run your build and tests for you on what are called virtual machines in the cloud.

Maybe the principle of continuous integration makes sense, but why are continuous integration services useful? Why can't each developer build packages, run the tests locally, and then only push changes to GitHub if the tests are successful? There are three answers to this question.

1. For large projects, it often takes a long time to build them. Being able to build them on the cloud (using Appveyor/Travis) saves an enormous amount of time. (This is especially true for programming languages besides R, but it's true for R too!)
2. Sometimes there may be global environmental settings on your particular computer which change the way the package works and the way tests run. Building and testing the package in a "clean" environment online makes the testing more robust.
3. Lastly, CI services offer settings to automatically 'deploy' or publish successful builds, for example by publishing code for a website. These are a bit beyond the scope of this guide, but they are documented here (Appveyor) and here (Travis).

2.4.2   The integrated testing workflow

It's important to note that Travis/Appveyor do not prevent you from pushing bad code to a repo - if you break your package and all your tests fail on Travis/Appveyor, you can still push the broken version to the master branch on GitHub. As a result, most developer teams use the following workflow to keep the master branch working at all times.

1. Someone sets up a "master repo" with a master branch on GitHub.
2. Each developer on the team creates a copy (also called a *fork*) of the master repo. Thus, each developer has their own repository.
3. Each developer makes changes to the project and then pushes those changes to their personal fork. After each push, Appveyor or Travis builds the package and test it.
4. If the Appveyor/Travis builds are successful, the developer will likely send a pull request to the master branch of the master repo, which will probably be accepted.

This workflow allows each developer to make changes in their own repo and continuously build them using Travis/Appveyor without potentially breaking the master repo's copy.

2.4.3 Setting up Integrated Tests

Step 1: To get started using continuous integration services, head to the Travis CI (Linux/Mac OS) or AppVeyor (Windows) website. Once you are on AppVeyor website, click 'SIGNUP FOR HOSTED SERVICE'. We recommend that you create a new AppVeyor account to sign up by signing into GitHub. At some point, Travis/AppVeyor will ask you to authorize its connection to GitHub, which you should authorize.

Step 2: Once you authorize the connection, you need to specifically select which repos you'd like to use Travis/AppVeyor with. For example (in AppVeyor), upon signing up, you'll see something like this screen:
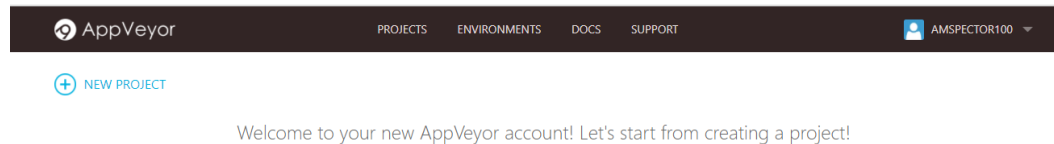
You should click "new project" and then select the specific repo you'd like to use in combination with Travis/Appveyor by clicking on "ADD". This means that you should at least initialize your project and push it to GitHub before you start using Travis/Appveyor.

Step 3: Depending on whether you're using Appveyor or Travis, you should run `usethis::use_appveyor()` or `usethis::use_travis()` in the console with your local repository open. This will create a file named 'appveyor.yml' or 'travis.yml' in the root directory of your local repository. These '.yml' files will tell AppVeyor/Travis what to do. It's actually not too important to understand everything they do, just because the devtools template generally works pretty well. However, you occasionally will have to modify it to suit specific needs. AppVeyor and Travis CI both have great documentation, linked here for Appveyor and here for Travis.

(Note - `usethis` also adds 'appveyor.yml' and 'travis.yml' to the .Rbuildignore to reduce clutter in your package.)

Step 4: Dependency Issues: If you're using Travis, it's important to talk quickly about dependencies in packages. Travis was not originally designed for R projects, so occasionally it's a bit shaky, specifically with regard to dependencies in packages (i.e. Travis will build your package online, but if your package depends on `ggplot2`, then Travis will have to install `ggplot2` before building your package). By default, Travis/Appveyor automatically install anything listed in the dependencies listed in the DESCRIPTION file for your project. However, occasionally something goes wrong. If you get error messages that look like Travis did not install all of the dependencies, it's worth specifying them in the .yml file. You can by adding a line like the following:

```
build_script:
  - r_packages: package_name
```

under *build_script* section in the .yml file (this will install the package the same way that R installs it when you run `install.packages()`). If you'd like to install a package using

a different method (i.e. from a GitHub version of the package), there are lots of options listed in the R-specific Travis documentation. However, it's important to remember that this is generally NOT necessary - Travis/Appveyor do a pretty good job of automatically detecting/installing dependencies as long as your DESCRIPTION file specifies them.

Step 5: If you are working with many other developers on a single package, you should also add a '.gitattributes' file into the root of your directory. This is because different operating systems handle different characters, for example line terminators, slightly differently. Modern Mac and Unix/Linux systems use `\n` to terminate the end of each line, in what is called the 'line feed' or 'LF' system, whereas Windows uses `\r\n` in the 'Carriage Return Line Feed' or 'CRLF' system. The point is that the '.gitattributes' file will ensure that each developer uses exactly the same line endings, because you can set the 'text' setting which controls the method of writing line terminators.

The way it works is that the '.gitattribute' file should be a line of path-patterns followed by specific attributes. These path-patterns (for a refresher, see the Atlanian guide here) are exactly the same path-patterns used in the .gitignore file, just for a different purpose. For example, one line might be

```
* text=auto
```

where the "*" refers to all files in the root directory, and the 'text=auto' sets the 'text' attribute to auto. Alternatively, you could write

```
R/* text=lf
```

which forces all the scripts directly in the R subdirectory to use the 'lf' line terminator. You can personalize the .gitattributes file if you like, but you can also just use the sample .gitattributes linked here. You can download this file and move it to the root directory of your package (it must be in the root directory).

The .gitattributes file actually is useful for setting a variety of other attributes besides line endings, although some of them are a bit technical. If you're curious to find out more about what the .gitattributes file can do, the full documentation is linked here.

Once you do all of this, you're set to go! When you push changes to GitHub, all the new files (.yml, .gitignore, .gitattributes) you created will be pushed to the remote. By clicking "NEW BUILD", Travis/AppVeyor will automatically build and test the package for you. For example on Appveyor, if you sign in, you should see something like this:

packageguidelines

LATEST BUILD   HISTORY   DEPLOYMENTS   SETTINGS   ▶ NEW BUILD   ▶ RE-BUILD COMMIT   ⬆ DEPLOY   ⬇ LOG

testing appveyors   1.0.6
3 hours ago by Asher Spector   ⅄ master   ◇ 89c347e2   3 hours ago in 1 min 17 sec

CONSOLE   MESSAGES   TESTS   ARTIFACTS

```
 1 Build started
 2 $ErrorActionPreference = "Stop"
 3 Invoke-WebRequest http://raw.github.com/krlmlr/r-appveyor/master/scripts/appveyor-tool.ps1 -OutFile "..\appveyor-tool.ps1"
 4 Import-Module '..\appveyor-tool.ps1'
 5
 6 git clone -q --branch=master https://github.com/amspector100/packageguidelines.git C:\projects\packageguidelines
 7 git checkout -qf 89c347e248afcea8c645cddd3aedd35f52b3716c
 8 Running Install scripts
 9 Bootstrap
10 == 04/25/2018 02:39:35: Bootstrap: Start
11 == 04/25/2018 02:39:35: Adding GnuWin32 tools to PATH
12 == 04/25/2018 02:39:35: Setting time zone
13 UTC
14 GMT Standard Time
15 == 04/25/2018 02:39:35: Version: patched
16 == 04/25/2018 02:39:36: URL path:
17 == 04/25/2018 02:39:36: Downloading R from: https://cloud.r-project.org/bin/windows/base/R-3.4.4patched-win.exe
18 == 04/25/2018 02:39:42: Running R Installer
19 R is now available on drive C:
20 == 04/25/2018 02:40:08: Setting PATH
21 == 04/25/2018 02:40:08: Testing R installation
22 R version 3.5.0 (2018-04-23)
23 Platform: i386-w64-mingw32/i386 (32-bit)
24 Running under: Windows Server 2012 R2 x64 (build 9600)
25
26 Matrix products: default
27
28 locale:
29 [1] LC_COLLATE=English_United States.1252
```

and if your build succeeded, you'll see a green line at the bottom like this:

```
335 + [[ -n   ]]
336 Collecting artifacts...
337 No artifacts found matching '*.Rcheck\**\*.log' path
338 No artifacts found matching '*.Rcheck\**\*.out' path
339 No artifacts found matching '*.Rcheck\**\*.fail' path
340 No artifacts found matching '*.Rcheck\**\*.Rout' path
341 No artifacts found matching '*_*.tar.gz' path
342 No artifacts found matching '*_*.zip' path
343 Updating build cache...
344 Cache 'C:\RLibrary' - Updated
345 Build success
```

otherwise you'll see an error. If your build takes a while, that's okay - you don't need to wait for it to build, because Travis/Appveyor will email you automatically if the test fails. And that's all there is to it!

2.5 Further reading

If you're interested in learning more about Git and GitHub, you might want to take a look at the following resources:

- The Software Carpentry Foundation has a great mid level Git tutorial here
- Atlassian has some wonderful advanced Git tutorials here

Chapter 3

Integrated Development Environments

3.1 General Suggestions

An Integrated Development Environment, or IDE, is simply a piece of software which aims to make it easier to write and work with packages. If you're reading this guide, you'll probably spend a decent amount of time writing packages, so we thought it'd be helpful to review a couple of different IDEs.

Broadly, there are two kinds of IDEs.

1. Some IDEs are **language agnostic**, meaning that they support any programming language you care to program in. Language-agnostic IDEs are nice because (a) if you use one, you won't have to switch IDEs every time you use a different programming language, and (b) because you can write packages in multiple languages in them.
2. On the other hand, some IDEs are **language specific** in that they are built for certain languages. Language specific IDEs can be useful because they have a host of speciailized features for their specific language.

There are lots of different IDEs available for R users. Common language agnostic choices include Atom, Visual Studio Code, JupyterLab, emacs and aquamacs, and even Sublime Text 3. However, RStudio is probably the most common and beginner-friendly IDE for R Users, so we'll focus on RStudio.

3.2 RStudio

RStudio is a language-specific IDE built for R, although it does support plenty of other languages.

3.2.1   Why use RStudio?

RStudio integrates documentation, makes graphics more accessible, and also combines beautifully with `devtools` and `usethis` to make it easy to write packages. We'll walk through how to set up RStudio, and then look at two concrete examples: initializing and building R packages.

3.2.1.1   Setting up R

Presumably you've already been working with R, but you should make sure you have a recent enough version of R intalled. To do so, type `R.Version()` into the editor you've been working with previously. It should return an output titled `version.string`, for example:

```
R.Version()
```

```
## $platform
## [1] "x86_64-w64-mingw32"
##
## $arch
## [1] "x86_64"
##
## $os
## [1] "mingw32"
##
## $system
## [1] "x86_64, mingw32"
##
## $status
## [1] ""
##
## $major
## [1] "4"
##
## $minor
## [1] "1.2"
##
## $year
## [1] "2021"
##
## $month
## [1] "11"
##
## $day
## [1] "01"
##
## $`svn rev`
## [1] "81115"
##
## $language
## [1] "R"
```

```
##
## $version.string
## [1] "R version 4.1.2 (2021-11-01)"
##
## $nickname
## [1] "Bird Hippie"
```

If the number after the R version string is anything lower than 3.0.1, you'll have to install a new version of R in order to use R Studio. To do so, follow the steps below:

1. Go to `https://cran.rstudio.com/` and select "Download R for [Your Operating System]."

2. Click "base" and click "Download R [latest version] for [Your Operating System]" at the top of the page.

3. You can then let the installer run itself - the default settings should be fine for our purposes.

Lastly, updating R on Windows can be a bit tricky. If you use Windows, you can also use the `installr` package to quickly update R, as documented here. However, if you are having trouble installing the `installr` package, you can always just follow the instructions above.

3.2.1.2 Setting up RStudio

To download and set up RStudio, follow the instructions below:

1. Head to `https://www.rstudio.com/products/rstudio/download/` and click "download" for the free RStudio Desktop version

2. Download the *installer* for your operating system, not the *zip/tarball.* The picture below illustrates what the webpage should look like.

3. Let the installer run - it's fine again to just use the default options.

and now you're ready!

3.2.2 Getting Started

3.2.2.1 Screen Breakdown

RStudio will split your screen into four parts, as shown below. We'll refer to them as the *script*, the *environment*, the *shell* or *console*, and the *Viewer.*

The *script* is in the upper left hand corner of the screen, and it's where you will do the bulk of your programming. To write an R script, click File>New File>R Script in the top left hand corner. You can execute multiple lines of a script (or the entire thing) at a time by selecting the lines you want to run and clicking the green "Run" button on the right hand side of the script box or by holding down the command key (MacOS) or control key (Windows) and pressing return/enter.

The *console* or *shell* is in the lower left hand corner of the screen. Unlike scripts, which presumably you've worked with before, shells only run one line of code at a time. However, shells also remember what you did before. For example, if you download some data into

RStudio Desktop 1.1.423 — Release Notes

RStudio requires R 3.0.1+. If you don't already have R, download it here.

Installers for Supported Platforms

Installers	Size	Date	MD5
RStudio 1.1.423 - Windows Vista/7/8/10	85.8 MB	2018-02-07	a2411be84794b61fd8e79e70e7c0f0b0
RStudio 1.1.423 - Mac OS X 10.6+ (64-bit)	74.5 MB	2018-02-07	3e3e3db076b44f3c5276eb008614b4cf
RStudio 1.1.423 - Ubuntu 12.04-15.10/Debian 8 (32-bit)	89.3 MB	2018-02-07	8515d8f5c78ac15b331bd9be0c1ea412
RStudio 1.1.423 - Ubuntu 12.04-15.10/Debian 8 (64-bit)	97.4 MB	2018-02-07	f6e385c13ff7a1218891937f016e9383
RStudio 1.1.423 - Ubuntu 16.04+/Debian 9+ (64-bit)	65 MB	2018-02-07	1b5599d9f19c0971e87a5bcbf77aa8bc
RStudio 1.1.423 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)	88.1 MB	2018-02-07	27664d49e08deee206879d259fd10512
RStudio 1.1.423 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)	90.6 MB	2018-02-07	8d3d8c49260539a590d8eeea555eab08

Zip/Tarballs

Zip/tar archives	Size	Date	MD5
RStudio 1.1.423 - Windows Vista/7/8/10	122.9 MB	2018-02-07	13f278a1fc35ee3cefda788792a5617b
RStudio 1.1.423 - Ubuntu 12.04-15.10/Debian 8 (32-bit)	90 MB	2018-02-07	a1e64ddc9f6ceab89b61a48a3254a0a8
RStudio 1.1.423 - Ubuntu 12.04-15.10/Debian 8 (64-bit)	98.3 MB	2018-02-07	bd91123b7b3b9d41d0659f14159d4d02
RStudio 1.1.423 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)	88.8 MB	2018-02-07	ffde3594a52cf4aadfeb715f1e31b1f1
RStudio 1.1.423 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)	91.4 MB	2018-02-07	d12d69d2926486db1e3b57194ae10a01

Source Code

A tarball containing source code for RStudio v1.1.423 can be downloaded from here

Figure 3.1: *RStudio Webpage*

the shell, you can manipulate it repeatedly without having to download it again before each manipulation - it will stay in the shell's memory until you manually remove it.

All scripts that you run will be run through the shell. This means that you can run a script and manipulate the results directly from the bottom of the shell, without having to rerun the script each time you want to manipulate the results. You can also directly access the command prompt for your computer through the shell by clicking the "terminal" option.

Note that any errors thrown by your code will show up in the shell. To clear the console, hit Ctrl+L.

On the upper right hand corner of the screen, you can see the *environment* you're working in, which lists all the objects in the shell's "memory" or *namespace*. For example, if you download some data in a script and then run the script, the data will show up in the right hand corner of the screen.

On the bottom right hand side is the *Viewer*. Any plots you generate in R will automatically show up there. Additionally, as we'll discuss later, documentation of functions and packages will appear in the right hand corner. You can also use it to see the file structure of your working directory and the packages you're using by clicking "files" or "packages."

3.2.3 Ex: Initializing Packages

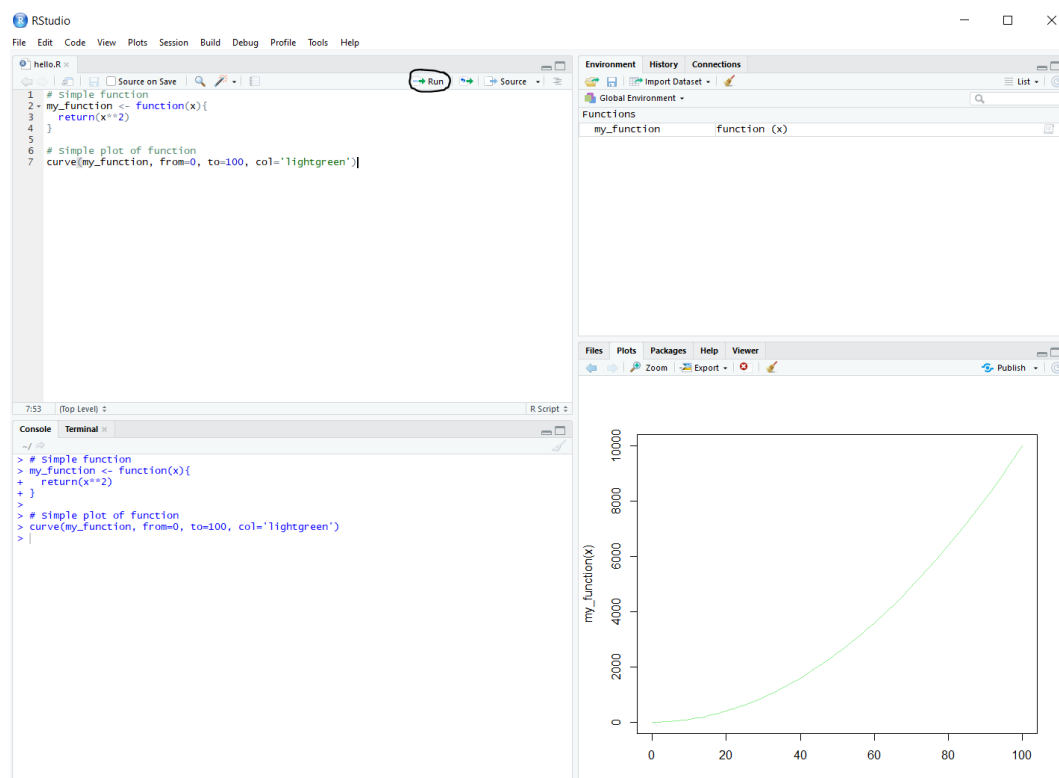RStudio makes it easy to create packages. To start, click File>New Project. Then, click "New Directory" when you see the screen below:
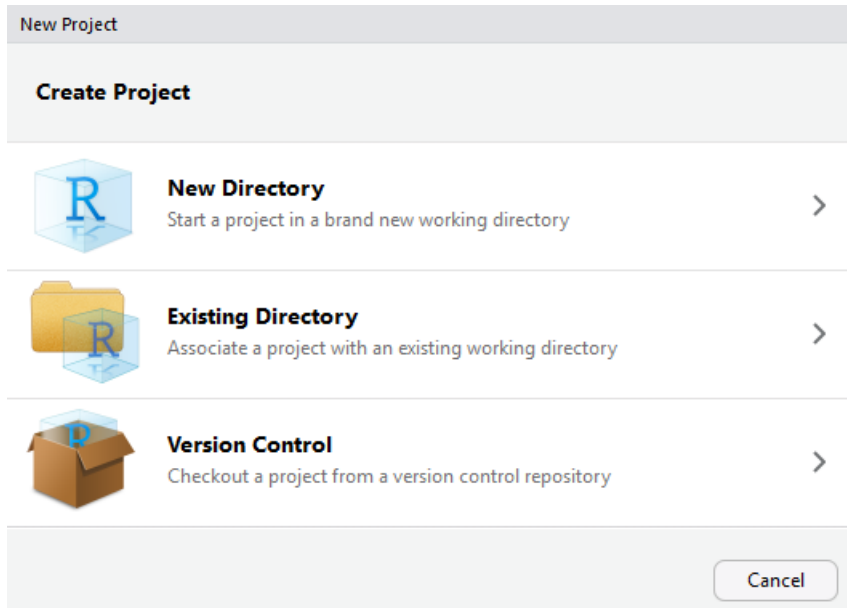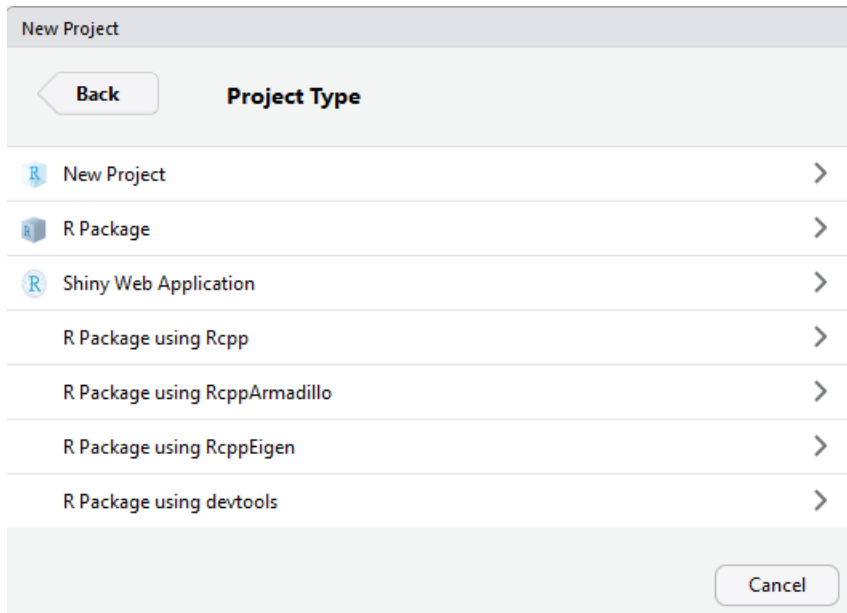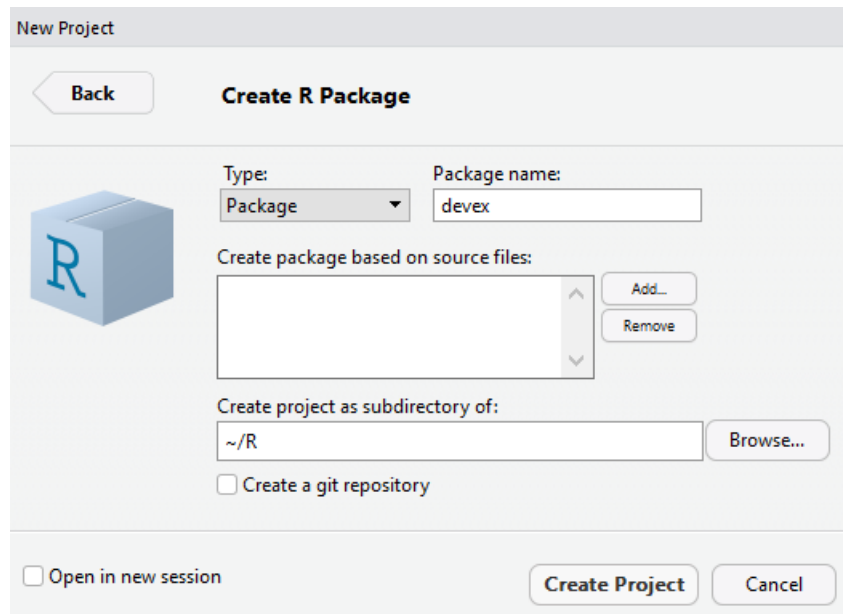
Figure 3.2: The RStudio Interface

Then click "R Package" when you see the screen below:

Lastly, name your package. Your name should be something short and descriptive. Since we'll be walking through a development example, we'll title our new package `devex`.

Once you've created your package, your RStudio should look something like this. It should come preloaded with a "Hello World" script which includes a handy function that prints "Hello world."
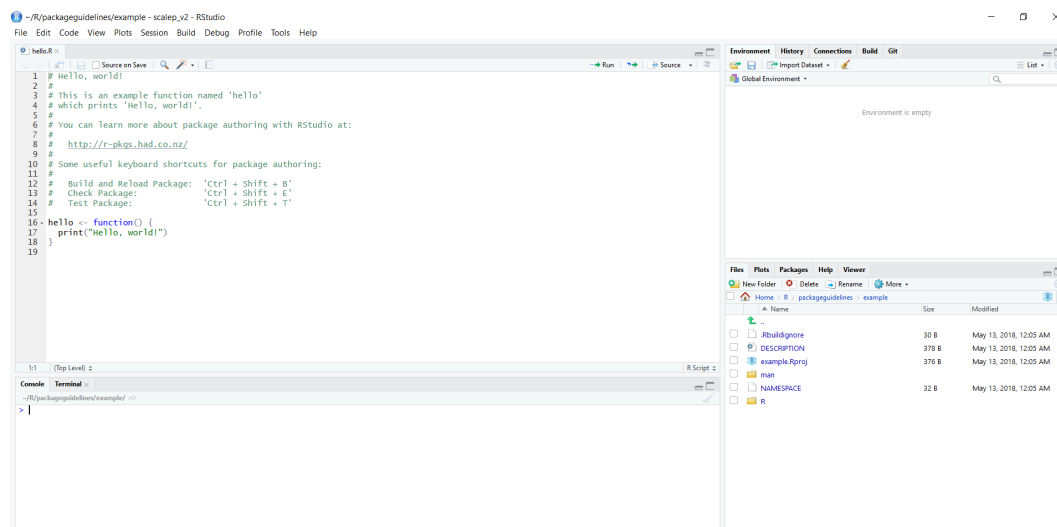
Everything should look mostly the same as normal except for the "Files" tab on the bottom right, which should include an ".Rproj" file, a "DESCRIPTION" file, a "NAMESPACE" file, and a folder titled "R." You should start by checking the little boxes next to the "NAMES-PACE" file and the "Hello.R" R script inside the R file and clicking "delete" above them, just because we will have Roxygen2 automatically generate the namespace, and because presumably your package doesn't need a "Hello World" function.

You should also go to "Tools > Project Options" and select the following options, which

will help you generate documentation using roxygen2. This requires "roxygen2" package to be loaded into the search path first to have the "Generate documentation with Roxygen" option displayed.
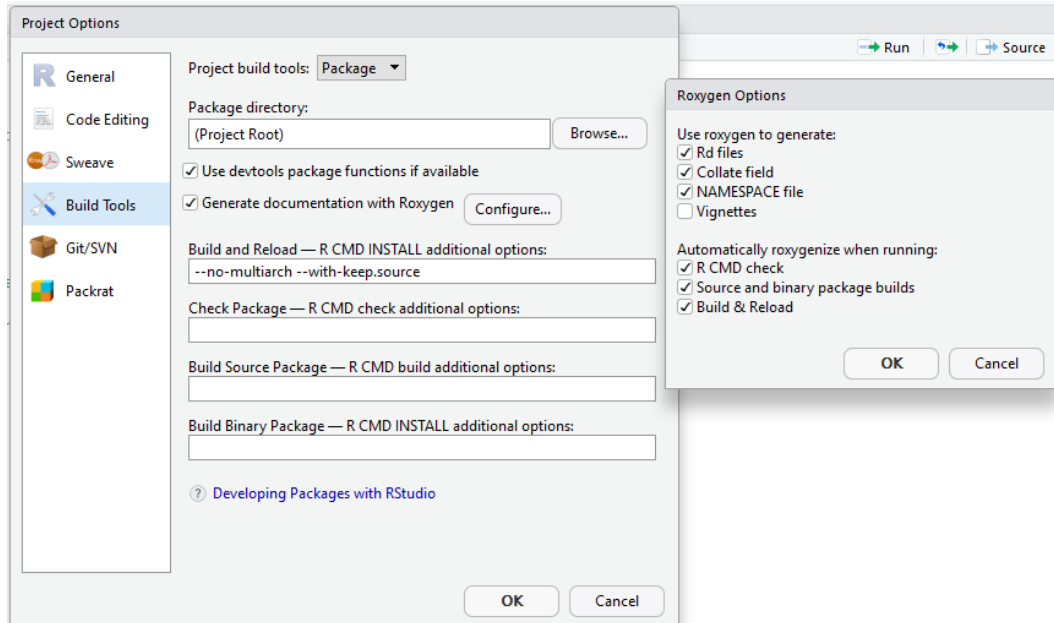
3.2.4   Ex: Building Packages

RStudio will build your package a little bit more conveniently than the `devtools::build()` function will. Whereas the `devtools::build()` function, without extra arguments, will simply bundle your package into one file, building your package from RStudio will actually install it into your R library, allowing you to easily use the finished version.

The way to do this is to go to Build > Install and Restart, and then RStudio will build your package, restart, and load your package into R (using the `library()` command). Then, you can use and/or play with your package!

(If you don't see any options under Build, you probably need to open your .Rproj file in the "file" section of RStudio. Upon doing this, RStudio should restart briefly and then you will see more Build options. Note, the .Rproj file may not exist if you created the package with `usethis` outside of RStudio).

Chapter 4

Resources

4.1 General

- This link contains a number of truly fantastic cheat sheets, documenting everything from RStudio itself to data visualization and machine learning.
- To read more about using R, take a look at the following website, built by Chapman and Hall
- As always, it's worth referencing stackoverflow if you're ever confused.

4.2 Package Development

- There's a wonderful cheat sheet for package development linked here. This also describes a lot of key components of the testthat package.
- If you are looking for a very simple example of a package, the devex package can be found here. If you're having trouble understanding the workflow for package development, it's worth looking through the `devex` package and making sure you understand all its components. Better yet, you can practice using `roxygen2` and `devtools` by creating a very small/useless package (2-3 simple functions)
- It can sometimes be instructive to look through the source code and documentation for `devtools`.

Additionally, as you build larger and more complex packages, you might need a deeper understanding of package structure. For a slightly more in-depth explanation of package development, you'll want to reference Hadley Wickham's R Packages. For a serious dive into package mechanics, you should consult the official R Extensions Manual, which is published by CRAN. However, at least for mid-sized packages, this guide probably has given you most of what you need to know.

4.3 Version Control

- The official GitHub cheat sheet lists all of the general commands you'll need
- Karl Broman's tutorial runs you through initializing a repository

- Atlassian's table of the .gitignore syntactical rules is listed here.
- Here are the links to the build configuration docs for Appveyor and Travis.

If you're interested in reading more:

- The Software Carpentry Foundation has a great mid level Git tutorial here
- Atlassian has some wonderful advanced Git tutorials here

4.4   IDEs

- This page lists some more of RStudio's advantages.