Q-STEP R 'HOW TO' GUIDES:

BUILDING PACKAGES IN R

Creator: Dr Patrick English

Building packages in R might seem like a very high-end programming level skill, but in fact if you have cracked writing functions then you are just a couple of steps away from developing your own R package.

This guide will walk you through existing packages and their components, give a brief recap on function writing and will then take you through the nuts and bolts of constructing R packages. It is split into the following sections:

Those familiar with R packages and comfortable writing functions may wish to proceed straight to section three, but should consult section 2 of the guide for the codes necessary to create the functions used in the example.

This guide assumes a fairly competent level of R knowledge from the start – users should already have good knowledge of what a function is and how to use one before proceeding with this guide. This Quick-R guide provides a good resource for users needing to quickly familiarise themselves with functions and their applications. I also work with R Studio throughout this piece.

Q-Step
A step-change in quantitative social science skills
Funded by the Nuffield Foundation, ESRC and HEFCE
Nuffield Foundation
Sheffield Methods Institute.
The University Of Sheffield.

1. What is an R package and why might we want to develop one?

Users familiar and experienced in R probably already know what a package is and how to use them, but the following section provides some base level knowledge which might be worth reading to help you get thinking about how packages actually work and how writing one might be useful to you.

Quite simply, packages are the organs of R and functions are their arteries and veins. Almost every time we create, manipulate, examine, and even display objects, we are using functions and commands which are nested in together in R packages.

Some of the most common packages that users will work with include the graphing package ggplot (now on its second full version), stats (the basic R stats package which contains functions for running ANOVAs, linear regressions, and correlation analyses), and plyr (and dplyr, the packages most frequently used for transforming, combining, and describing data).

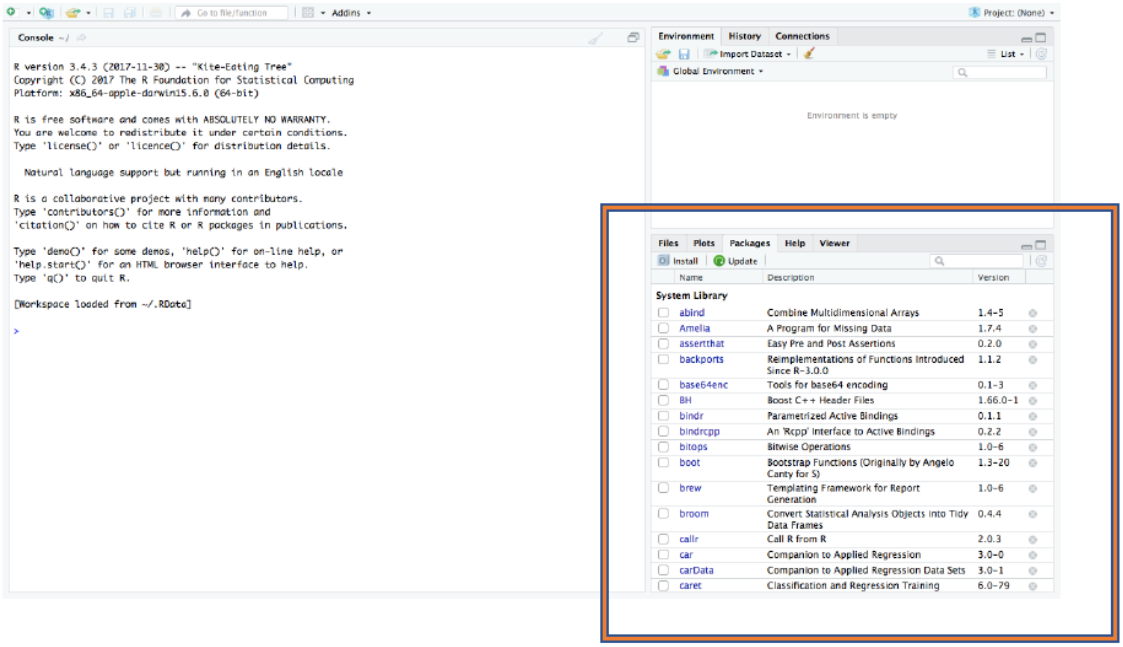
Quick-R describes packages as "collections of R functions, data, and compiled code in a well-defined format". Essentially, a package serves as a place to collate and store R scripts and code which we can load up and call in a flash. When we install and load up packages to use in our R workings, we are essentially simultaneously 'calling up' a whole bunch of interrelated and interconnected functions and code into our workspace, ready for us to use at a moment's notice.

Which leads to the first reason why developing our own R package might be useful to us as we go about our working in R: just like writing and using functions, constructing R packages and loading them up when we open up R or clear the global environment can save us a huge amount of time and coding (and head!) space.

The second reason lies in the ability for other users to examine, help us, and improve on our code.

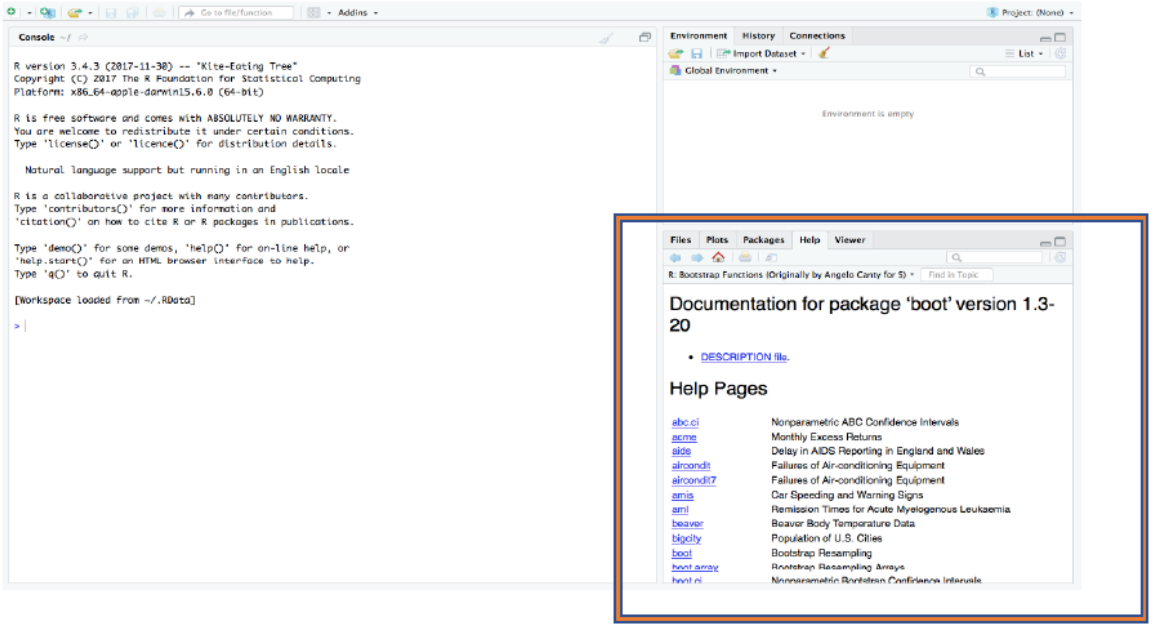
The guide will later take readers through how to publish R packages to GitHub as open repositories. This allows our friends, colleagues and peers to look over our coding and formatting and help us out with areas we might be stuck on or which might need improving or adapting for generalisability/further use. Sharing, learning and improving knowledge is vital to the success of every professional (in any career).

Packages are listed as a tab on the bottom-right pane in our R-studio workspace (using default view), as below:



This is our package library, where all the default packages which come with the R-studio installation are listed. Whenever we add packages through typing directly into the console or using the CRAN/GitHub repositories, they will appear in this list. We access downloaded packages by typing library("PACKAGE NAME") into the R console.

If we click on a package – for example 'boot' – we are presented with a bunch of links to various help and description files. The first 'DESCRIPTION' file contains information on the package itself. Following this are help pages for all of the package's various components – its functions (such as boot, boot.ci) and example data frames (such as beaver, bigcity):



When we load a package by calling it using the library() command, we are bringing each of these elements into our workspace and global environment (though to keep things clean, many of them are 'masked' from actually appearing in the global environment panel itself). Clicking through on any of the links gives us descriptive information on the object. For example, click the 'boot' link to be presented with the following screen:

Here we can see a description of the object – in this case it is a function to be applied to data – its usage (the commands which are called, and arguments required upon applying the function), and a list of the 'arguments' taken by the function (what user inputs are required). Further down are further details and examples of how to use function.

This little exploration gives us a great visualisation as to what a package actually is and sets up nicely for developing our own.

2. Writing functions – a quick recap

Common and extensively developed R packages such as ggplot2, boot, and plyr come with an extensive amount of documentation, data, and how-to guides which enable users to familiarise themselves with and master using the package.

For our purposes, we are not so interested (yet!) in developing packages for world-wide usage and we are going to focus our package on building a solid collection of the most important components of all R packages – functions.

This section is a quick reminder of what functions are and how to specify them in R scripts. Note that it is important at this stage to stress that, rather than placing commands directly into the R console and working from there, when developing functions and packages it is best practice to use R scripts. Open a new script by following (from the top menu in R Studio) 'File > New File > R Script'.

Functions are essentially shortcuts for some more complex code which can apply any number of functions and loops to objects (or other functions) following user-defined arguments.

For example, we might specify a function called 'heating_advice' which takes the arguments 'feeling' – a string variable whether or not the user feels hot or cold - and 'thermostat' – a numeric variable reporting the current thermostat level. We will use a couple of if statements to inform the function to display advice on what the user should do to the thermostat depending on whether or not they report feeling hot or cold:

```
heating_advice <- function(feeling, thermostat)    {
    if(feeling == "Hot") {
        new_thermostat <- as.numeric(thermostat - 3) print("You are feeling too
        hot, follow this advice:")
        cat(paste('Turn the thermostat down to', new_thermostat, '\n'))
    }

    if(feeling == "Cold") {
        new_thermostat <- as.numeric(thermostat + 3) print("You are feeling too
        cold, follow this advice:")
        cat(paste('Turn the thermostat up to', new_thermostat, '\n'))
        }
}
```

Remember to run the R script once it is written out in order to load the function into the global environment. We can call the function and then use “Hot” and 21 as examples written into the console to check that the function is working:

```
heating_advice("Hot", 21)

[1] "You are feeling too hot, follow this advice:" Turn the thermostat down to 18
```

Let’s imagine that we want to do something similar but this time using windows instead of thermostats. The ‘window_advice’ function again takes the argument ‘feeling’ (self-reported hot or coldness) but this time takes ‘windows’ (how many windows are open) instead of the thermostat temperature. Write and run the following in a new R script window.

```
window_advice <- function(feeling, windows) {
    if(feeling == "Hot") {
        new_windows <- as.numeric(windows + 1)
        print("You are feeling too hot, follow this advice:") cat(paste('Try opening',
        new_windows, 'windows \n'))
    }

    if(feeling == "Cold") {
        new_windows <- as.numeric(windows - 1)
        print("You are feeling too hot, follow this advice:") cat(paste('Try having',
        new_windows, 'window(s) open \n'))
        }
    }
```

Again, we can test the function works properly with an imaginary situation of feeling cold with 3 windows open written into the console:

```
window_advice("Cold", 3)

[1] "You are feeling too cold, follow this advice:"
Try having 2 window(s) open
```

These are the two functions from which we will build our R package.

3. Constructing an R package

Once we have a set of functions, developing an R package is really quite easy thanks to the 'devtools' and 'roxygen' packages. The former helps us construct the basics of R packages (all the necessary components and functions) and build in other packages into our script, while the latter helps us write up the documentation. This helpful explainer on devtools is definitely worth referring to and revising from as we call commands and functions in throughout this example.

Note: at this stage, it will be necessary to 'active' development tools on your system. This is a fairly easy task and there are really simple walkthrough guides available for Mac users here and Windows users here.

Once we have activated development tools, we can install the devtools directly from the CRAN repository using the following command: install.packages("devtools"). Once downloaded, load the package and then use the following command to install roxygen from the GitHub repository: devtools::
install_github("klutometis/roxygen"). Altogether, that code should look like this:

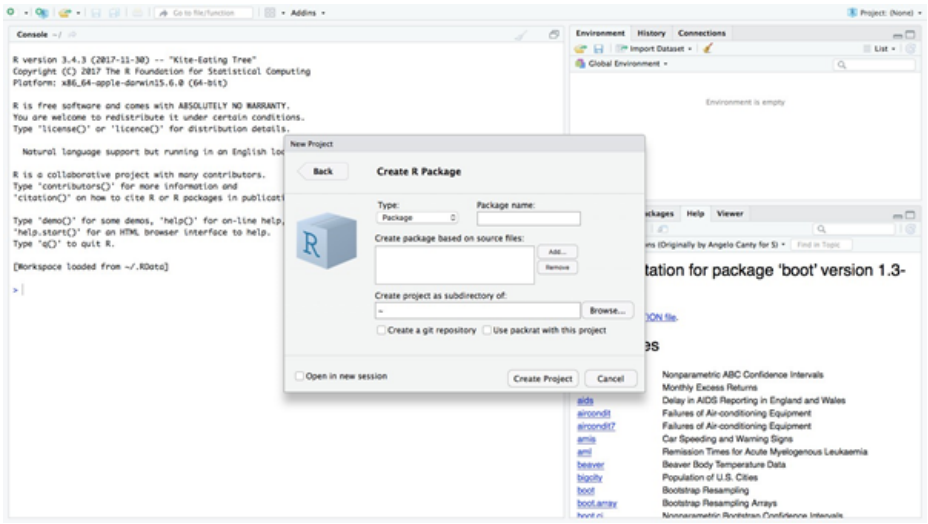
```
install.packages("devtools")
library("devtools")
devtools::install_github("klutometis/roxygen")
library(roxygen2)
```

You will of course notice that the command to install roxygen is quite different to that which we normally use to access packages (such as devtools). This is because, like many thousands of packages, roxygen is listed on the GitHub repository rather than CRAN. As such, we have to specify to R to search for and download roxygen from GitHub using devtools' install_github command.

While GitHub is an entirely open and accessible platform on which we can store and disseminate R packages (among a large number of other programming resources), listing on the CRAN repositories requires the passing of a number of strict tests and analysis on packages. This helps to ensure high standards of stability and usability of packages.

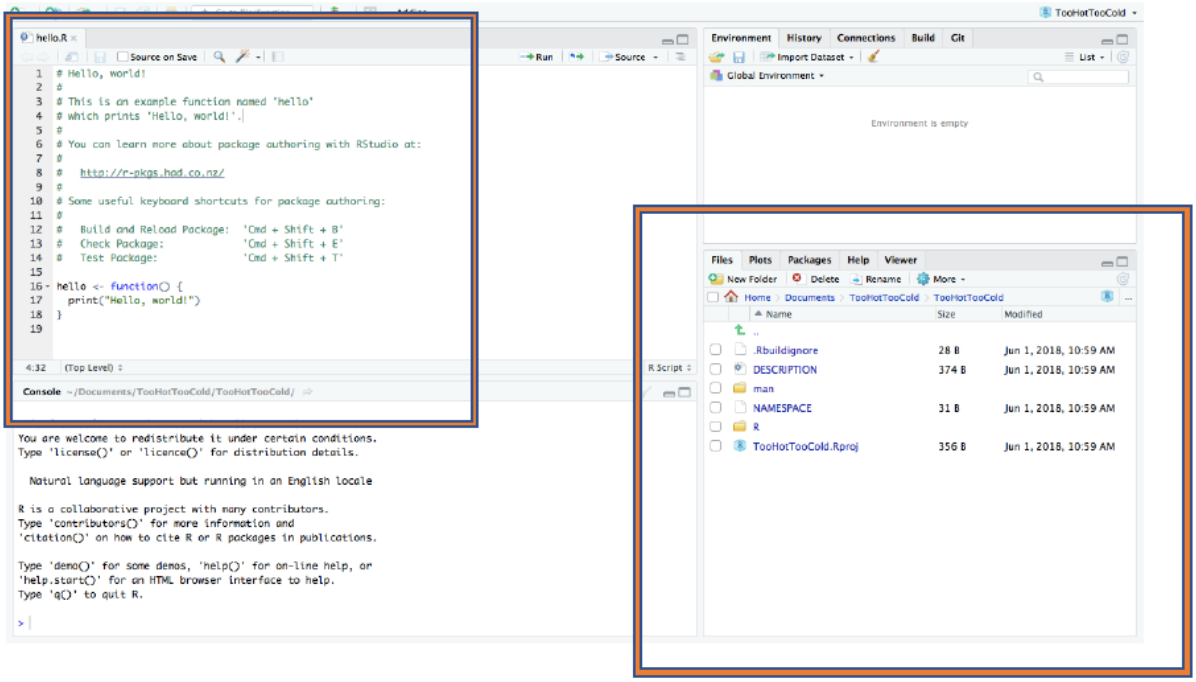
With devtools installed, we can move to developing and publishing our R package. Let's imagine that we want to be able to quickly and easily load up and share our heating and window advice functions to the rest of the world by making a package and sharing it online.

The first step is to 'begin' a package building project in R Studio. Navigate through 'File > New Project' and select 'New Directory > R Package' to proceed to the following screen:

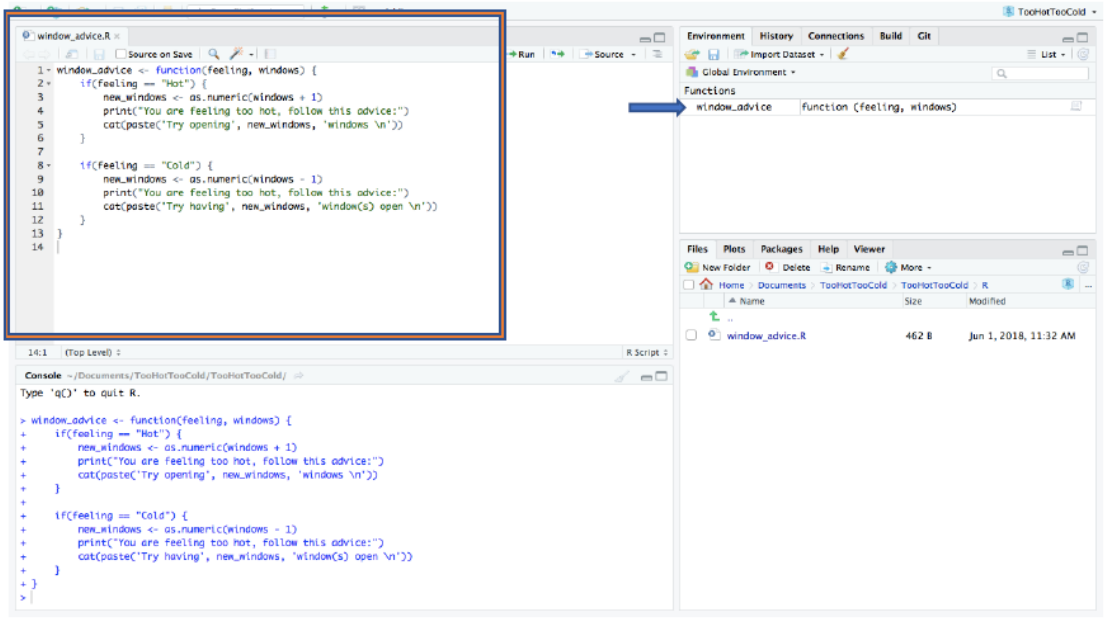


Insert 'TooHotTooCold' as the package name, specify a working directory (I will be using '~Documents/TooHotTooCold') and leave all other options blank1. Click 'Create project' to establish the project and load up the build tools in R Studio. You will notice that there are options to 'Create a git repository' and 'use packrat with this project'. While these options can be very useful, at this stage we will keep things nice and simple and work directly inside R Studio before moving 'up and out' to publishing via GitHub.

When the project space loads, it will look a little something like the below. In the top left we will see an 'example' script with a function named 'hello world' and some helpful keyboard shortcuts. In the bottom right is the working directory for TooHotTooCold. In there we can see that R Studio has already created a number of files and folders in which we will deposit (in the 'R' folder) and generate (in the 'man' folder) scripts to build the package:



Close the 'hello.R' script and delete it from the R folder. Instead, replace it with a new script titled 'window_advice.R' with just the same code as before. Save this file into the R folder and run it to load the function into the global environment (this means we can instantly call and check it at any time):



As mentioned above, the roxygen package allows us to quickly and easily write up documentation for our R package. Documentation is very important as it helps users to understand what our functions do, how to use them, and what options and arguments the functions take which might be customisable or interchangeable.

Using roxygen to write documentation is very easy. First, re-load the devtools and roxygen packages. Then place the following lines into the top of the window_advice script and re-save it:

```
#' @title Window_advice: should you open or close windows?
#' @description This function will tell you how to adjust your temperature depending on how many windows you have open, and if you are feeling hot or cold
#' @param feeling specify whether you are feeling hot or cold, string inputs "Hot" or "Cold"
#' @param windows specify how many windows you currently have open, numeric inputs
#' @examples window_advice(feeling = "hot", windows = 1)
#'
```

#' specifies that there are roxygen sections of the script which we want to turn into documentation. The code following @ statements tells roxygen how to order and arrange the documentation. The roxygen package provides a number of different @ paramaters to help us buid our documentation which can all be seen by calling the ryoxgen help screen. Included above are the most important – the title and description give a brief but comprehensive overview as to what the function does, the param code will tell users what arguments the function takes, while the examples code gives users and example of how to call and specify our function.

Repeat exactly the same process on the second script 'heating_advice' using the following code and saving the R script in the same location.

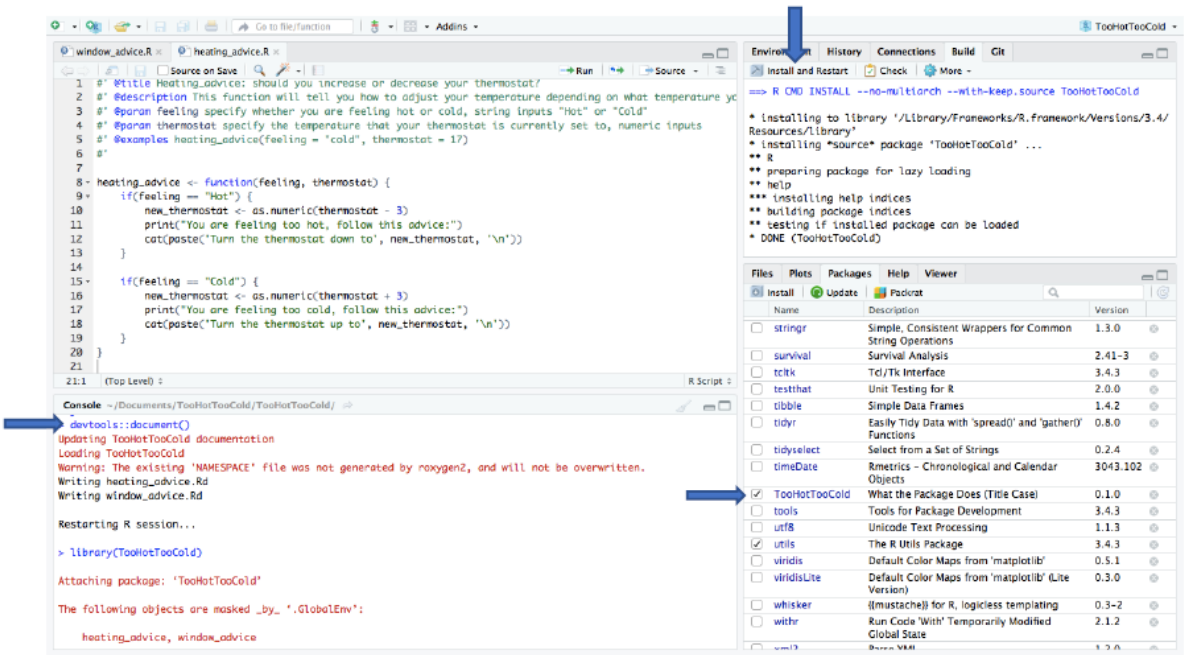
```
#' @title Heating_advice: should you increase or decrease your thermostat?
#' @description This function will tell you how to adjust your temperature depending on what temperature your thermostat is set to, and if you are feeling hot or cold
#' @param feeling specify whether you are feeling hot or cold, string inputs "Hot" or "Cold"
#' @param thermostat specify the temperature that your thermostat is currently set to, numeric inputs
#' @examples heating_advice(feeling = "cold", thermostat = 17) #'

heating_advice <- function(feeling, thermostat) {
    if(feeling == "Hot") {
        new_thermostat <- as.numeric(thermostat - 3)
        print("You are feeling too hot, follow this advice:")
        cat(paste('Turn the thermostat down to', new_thermostat, '\n'))
}
```

```
    if(feeling == "Cold") {
        new_thermostat <- as.numeric(thermostat + 3) print("You are feeling too
        cold, followthis advice:")
        cat(paste('Turn the thermostat up to', new_thermostat, '\n'))
    }
}
```

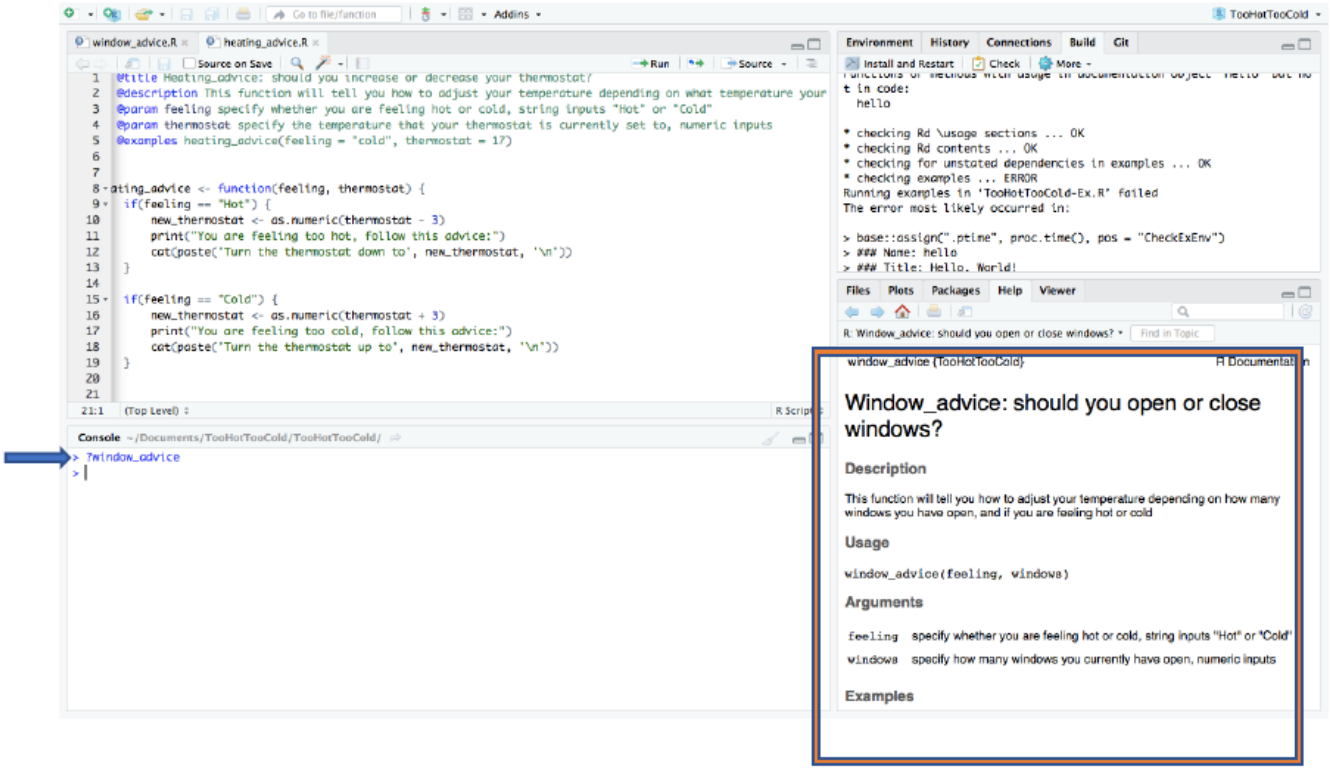
With our functions defined and documentation script added, we are now ready to build and process the package. The two most important tools to do this are the devtools::document() function and the 'Install and Restart' option under the 'Build' tab in the top-right pane of your R Studio window.

The devtools::document() function will automatically write up the R scripts into .Rd files and will generate any missing documentation files not already generated. Run this function directly into the console. Once this process has been completed, click the 'Install and Restart' tool in the 'Build' tab to tie together and load up our R package:



We can see that after the document() function wrote up our two function scripts into .Rd files, the 'Install and Restart' option restarted our R session and loaded the package up into the global environment. Finally, we can now see that the newly constructed TooHotTooCold package is listed and loaded in our package library. It really is that simple!

We can check that the functions are working correctly by manually calling the functions and running our examples, and calling the help files by running "?heating_advice" and "?window_advice" in the console. You can see that all of our ryoxgen commands in the function scripts have created a nicely laid out, clean help file:

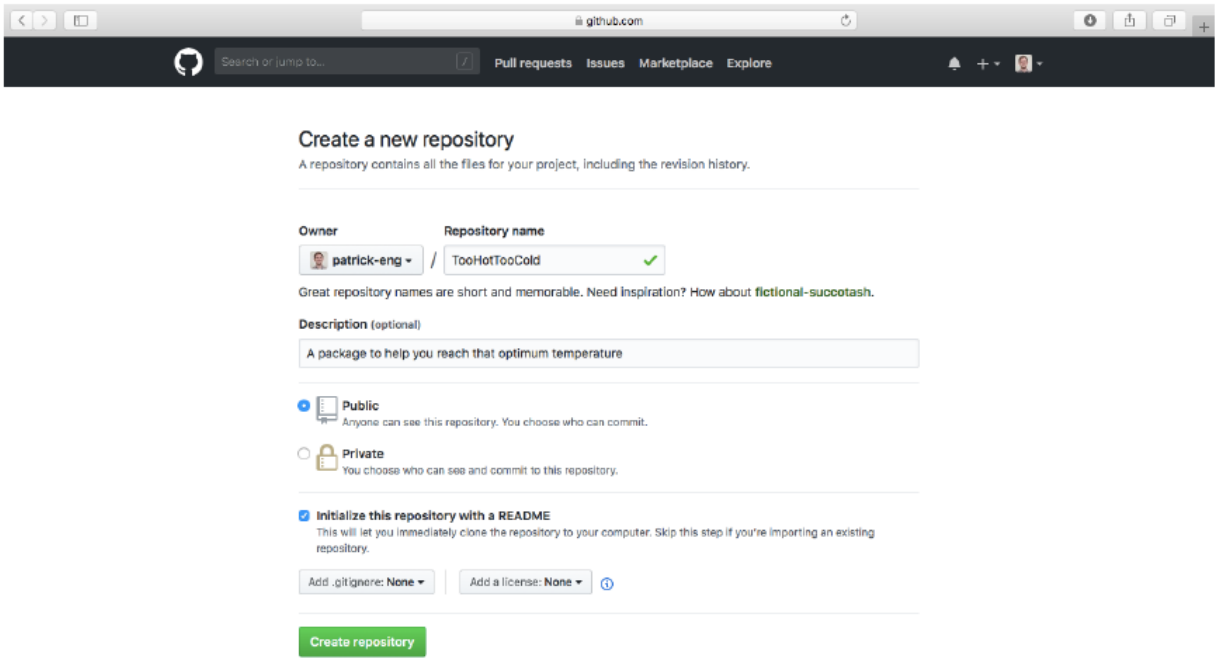


Re-run the same examples as we used to test the functions originally, and if everything checks out then we can move to publishing. Remember, now that the package has been built and installed, we don't need to re-run or re-load our functions – just load up the package and everything will arrive seamlessly into the work space ready to be called. Lastly, we can update and add functions to our package at any time by altering/generating new R scripts to deposit into the R folder within the package directory (remember then to update the documents and re-install the package!)

4. Depositing R packages on GitHub

This final section outlines how to publish completed packages on the GitHub repository. If you don't already have a GitHub account, it is very easy to set up and establish a repository by following this guide.

Establish a new repository on your GitHub with the same name as the package – TooHotTooCold. Add a short description, select 'Initialize this repository with a README', and leave all else as it is:



To prepare our package for publication we must add a few extra details. We need to make sure we outline the package information in the DESCRIPTION text file provided by the package build.

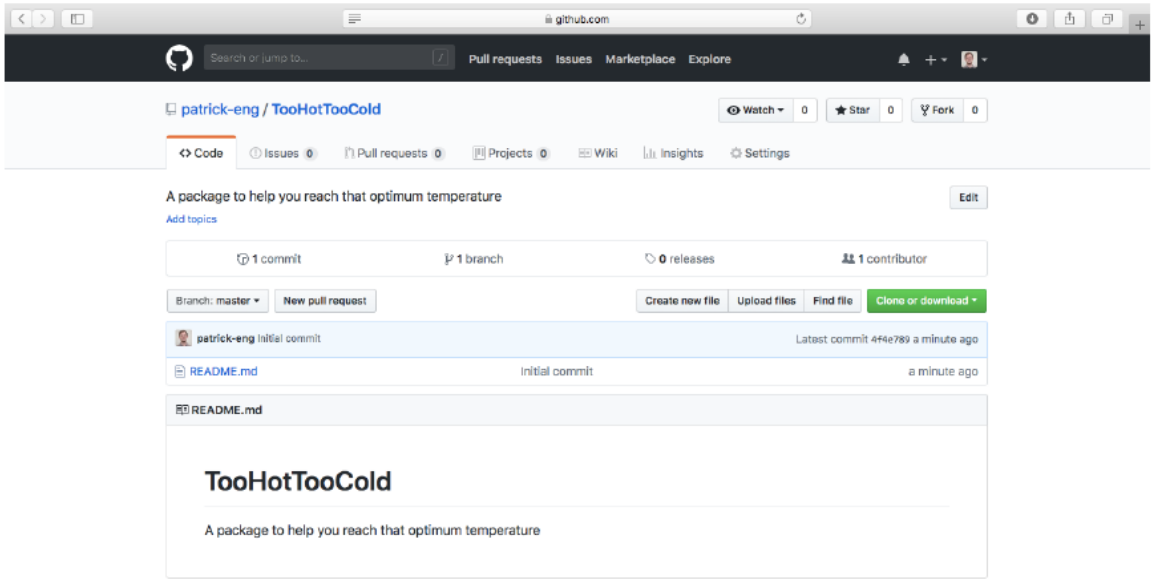
This is a simple manual process of opening the file up in your designated text editing service and filing in titles, short descriptions, author names and contact information, and so on and so forth. This is important information for people to learn a little more about your package and how to use it. Be sure to use the pre-built template and try something like this:

```
Package: TooHotTooCold Type: Package
Title: TooHotTooCold: A package to get your temperature just right
Version: 0.1.0
Author: Patrick English
Maintainer: Patrick English <p.english@sheffield.ac.uk>
Description: This package will help you achieve the optimum temperature through
opening windows and changing your thermostat. Simply input how you are
feelings and your thermostat and window information, and the package will come
up with some helpful suggestions to get you feeling just right!
License: GNU General Public Licence v3.0
Encoding: UTF-8
LazyData: true RoxygenNote: 6.0.1.9000
```

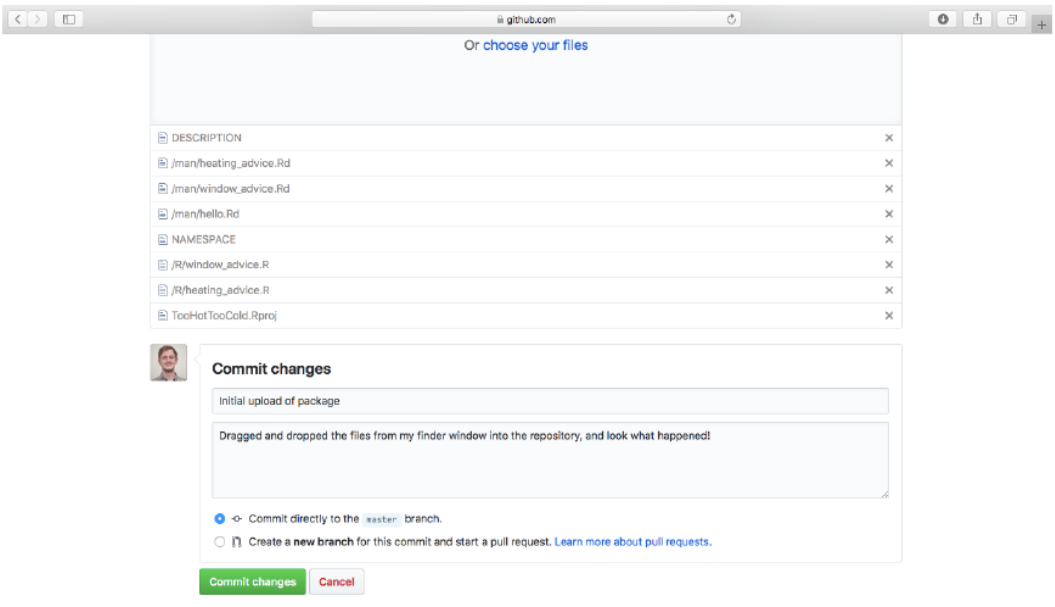
You can find out more about Licenses from this guide.

With details entered, close the package build in R Studio by following 'File > Close Project'.

Then, all is left to do is to load up our newly created package files to the GitHub repository. This is a very simple 'drag and drop' process. First, open up the TooHotTooCold repository home page, which should look like this:



Then, 'drag' your **entire package folder** from your explorer/finder window into the repository. Once the upload has been completed, we are asked to 'commit' our changes with the option to provide a brief overview of what we've done. It is really useful to get into the habit of labelling all of your changes on GitHub so that if something goes wrong or you have another reason to need to go back to a previous version, you can track your work and changes really quickly and easily:



Once we click 'Commit changes', the files will be processed into the repository and then appear in the '<> Code' tab. And that's that! The package is uploaded and ready to be accessed and used by any researcher across the globe.

You can test the build by running the install_GitHub command through the devtools package: devtools::install_GitHub("YOUR-GITHUB- USERNAME/TooHotTooCold").

As you develop more complex packages, it is also a good idea to test this on a separate machine which was not involved in the build, just to check that any and all of the dependencies and external functions have been correctly specified and included in the package documentation.

5. A quick note on running external package functions

Although it is not covered in this example, often we might find ourselves building functions which call other functions from different packages into the script. Without proper directing, our package documentation will not automatically pick this up and so our package functions will not run properly once processed and disseminated.

Luckily, devtools provides another really easy way for us to include functions from other packages into our scripts. For example, let's say we want to make use of some functions from 'dplyr' perhaps to summarise or manipulate our data.

Firstly, we can tell the package to 'require' (in other words download and attach) the dplyr package when installing our package by using the devtools::use_package() function. Not only will this function write the necessary code into our documentation file to ensure that dplyr is downloaded (if needed) and attached upon installing and attaching the TooHotTooCold package, but it also tells us the code we need to use in our own package scripts to call and apply functions from dplyr. Running "devtools::use_package("dplyr")" into the console produces an output looking like this:

```
devtools::use_package("dplyr")
* Adding dplyr to Imports
Next:
Refer to functions with dplyr::fun()
```

Devtools is telling us that in order to use any dplyr functions in our own package, we would have to place "dplyr::" in front of them. For example, a function to summarise a data frame must be called like this: "dplyr::summarise(data, mean=mean(data$varname) ...)".