

MATH 3080 Lab Lecture 5

Curtis Miller

12/30/2019

Lecture 5

R Package Development

You have been installing and loading packages almost as soon as you learned the basics of R. The strength of R is in its package community, providing packages implementing tools for data cleaning and management, computing algorithms and data structures, and both common and esoteric statistical procedures. Additionally packages are often written by experts in the respective topic (maybe even the procedure’s inventor) and the quality of the code is checked and enforced by CRAN, a repository maintained on a volunteer basis. The R language could remain relevant for years–decades, perhaps–based only on the strength of its package ecosystem (and low price and open source nature).

Today I will introduce you to package development. You may believe that you will never be an R “programmer” and simply will write enough R code to complete the statistical analysis you have been tasked with. While many people reading these lecture notes won’t publish a package on CRAN (though you may be surprised!), several reasons exist for *everyone* to learn package development.

As of this writing I am an academic statistician who studies and develops novel statistical methods. This fact alone would justify me personally writing packages: I can distribute the methods I develop to a wider audience. (As of this writing I have one package on CRAN, **CPAT**: a package for change point hypothesis testing, developed as part of recent research projects.) However, even when my research topics are vague, I start writing code and organizing files in anticipation of a package eventually emerging, and even when I’m still conducting research I use the R package structure as the organizational basis of my code and files. That is, I conduct research *via* package development.

Why conduct research via package development? First, package development encourages good coding and project management habits. Good coding habits include:

- Following the DRY principle: don’t repeat yourself. This means identifying repeated idioms in code and turning those idioms into generalized functions.
- Sufficiently abstracting code with functions. Abstraction brings two benefits. One, it hides implementation details, making those details *easier* to change rather than harder, since they need to be changed only in a handful of places. Second, your code is easier to read and reason about since the code largely consists of well-named functions with well-defined jobs that code readers can investigate only if they feel the need to do so. If this is done well, then there’s no need to extensively comment your code; the code explains itself. Some may even say the code itself *is* the documentation.
- Documenting your code. In package development, this generally means writing documentation for functions that: gives the function a title and description; explains input parameters, what’s assumed about them, and how they’re used; describing function output; any other important information about how the function works (such as what algorithms are used, potential errors, etc.); and some not-too-complicated examples demonstrating how the function should be used. In my own work, rarely will I put comments in code or function bodies; the function documentation is all that I need. (Also, I set up my text editor, Vim, to build templates for function documentation automatically.)

- Keeping code short. Here, “short” can mean no more than one “screen”. That is, an entire body of code can be seen without scrolling. This is not a hard rule; when you feel you must break this rule, do so. But short bodies of code are easier to understand than long bodies of code. Keep code short by hiding execution details in function calls.
- Keeping source files at a manageable size. By source files, I don’t mean `.Rmd` files (which is how you’ve been submitting your code so far) but `.R` files, the original R scripts, which consist *only* of valid R code (though perhaps code for other programs and languages exists in the form of special comments). All functions in a file should be closely related and serve a common purpose. Perhaps one file contains a class definition and related methods, another generic function definitions, another functions implementing a statistical test, another plotting functions, another functions just for data management and cleaning. When splitting code across files, we neither want “tower” files with all the code nor “pancake” files where 1000 files contain only one function each.
- Writing code tests. In programming, errors are great; they tell us immediately something is wrong. Far worse are mistakes that never reveal themselves and thus causing us to sell garbage results. We write code tests to make sure that code functions as we intend it, and so that changes we make don’t have unintended consequences. Additionally, we can give our tests to other users for them to read, run, and verify that our code works as intended.
- Flexible code with replicable results. We can modify our code for new contexts and to meet the new demands of our employers/clients, and others can run the same code and get the same results.

Bad coding habits include:

- Copy/paste programming. Sure, it was quick to copy/paste the first time, but if you need to change that original procedure, you now need to change all 1000 instances of that idiom.
- Long “spaghetti” code that only the original author can read (but probably not a few years later after the original project ended and she forgot everything) with things happening many pages prior to when it became interesting, repeats itself to the point of dulling the senses, and littered with the worst kind of comments: the out-of-date comment that never was changed to reflect important changes in the code and thus *misleads* readers.
- Code without any documentation or explanation, so no one knows what it does without careful reading. That takes time.
- Single source files with all the code of a project that gets executed via `source()` for the analysis and thus doesn’t allow for easy modification. A small change near the end of the file would mean the entire project needs to be re-run, which takes hours. That or someone opens the file and carefully does just the part that needs to change, which invites errors.
- Untested code that gets changed and suddenly produces subtle bugs that takes days to diagnose when a problem should have been flagged much earlier and closer to the original violated assumption.
- Code that’s so rigid and convoluted you don’t dare change it and risk breaking its delicate structure, even when the client just wants one additional statistical test done.

Remember, kids: computer scientists develop the standards they did and the tools they use for good reasons that apply beyond software development. Ignore them at your own peril.

Second, you may discover that code initially written for one specific task is actually generally useful. Both yourself and others can take advantage of all the work you did in the future and save time. Furthermore, this code was put in a package, which means its easily accessed!

Third, package development can be an important part of **reproducible research**. You may have heard that reproducibility is an important part of scientific research. Here, we want to be sure that the original analysis itself can be reproduced by anyone with the code and the same results can be obtained. This can allow others to diagnose any problems in our work (or verify that the work was done correctly) and perhaps modify, say, the input data, to replicate the research project and ensure that the results are robust.

I personally use package development as an important part of my own research pipeline. I combine it with a Linux/Unix development environment (turns out the black screen of death is **extremely useful** when you take the time to learn it, far more useful than GUIs), executable shell scripts (I use R Markdown or LaTeX only for writing documentation or presenting results, *not* for actually obtaining results), GNU **make** for

defining file dependencies and tracking what has changed and what needs to be changed (bringing my project up to date after making changes is as easy as typing **make** at the command line, and anything that needs to be changed will be changed while the rest is untouched), and of course version control. (I won't be teaching version control in this class but you really should learn some tool, such as **git**. Did you make a change that broke everything? Should have used version control. Experiments with your code didn't pan out? Should have used version control. Delete important lines or even an entire file? Should have used version control. Your folder is filled with files with minor name changes just for trying out little changes? Should have used version control.)

You may think you won't ever work on a project that gets big enough to warrant a package. Well, projects have a tendency to get bigger and more involved than initially thought. One day you'll open your laptop and discover that the "quick project" you were doing has several associated files and a single script thousands of lines long. Then you'll wish you had a package to manage all this code. In short: do it right at the beginning. You'll save yourself trouble.

Starting a Package in RStudio

I personally do not use RStudio for writing packages. I write my code (including these lecture notes) in a text editor called Vim. When I want to make a package, I create the necessary files and folders manually, and when I need to build the package I go to the "black screen of death" and type the necessary commands myself. But many people like GUIs so I will show how to use RStudio for starting a package, managing files, and building the package.

You may need to install additional software to get started. Specifically, you may need GNU software development tools (in particular a C/C++ compiler) and a LaTeX installation. Getting these tools depends on your platform (Windows, Mac OS X, Unix/Linux), so prior to proceeding visit this document for further instructions. (You shouldn't need to pay for anything.) Additionally there are two packages you should install that help with the development process: **devtools** and **roxygen2**. The former is a toolbox for development and you may even already have it installed to facilitate installing packages from GitHub. **roxygen2** allows for writing documentation by recognizing certain comments with special syntax as documentation. While you're at it, install **testthat**; it's the package we will use for writing tests, though we won't discuss tests in the current lecture.

Okay, let's start making a package. Recall the code we wrote implementing a class representing a financial account a few lectures ago. Let's turn that code into a package, calling the package **account**. When we start RStudio we may have a screen resembling the following:

Open the **File** menu and click **New Project**.

Projects have a rigid directory structure and care a great deal about what files are in what folders. Since we have a fresh new package we want a fresh new directory. Click **New Directory** in the pop-up window.

When asked what type of project we're starting, click **R Package**.

We should now see in our pop-up window queries about package details. Name the package "account". If you want you can choose which directory the new directory should be placed in.

The window should close and now RStudio should resemble the screen below.

RStudio created a basic package with minimal structure. The most essential files and folders have been made, and we will be modifying them. RStudio even gave us a starting R file with some tips.

Additionally the directory RStudio created looks like this:

The file **hello.R** RStudio created is in the **R/** subdirectory. I will explain package subdirectories more later. Let's first check that our package can be built and installed by pressing (on Windows/Linux) **Ctrl+Shift+B**. When you do you should see output in one of the RStudio panes resembling the following:

Additionally, in the R console, you should see **library(account)** run. If all went smoothly, you should be able to type **hello()** in the console and have R respond with **[1] "Hello, world!"**.

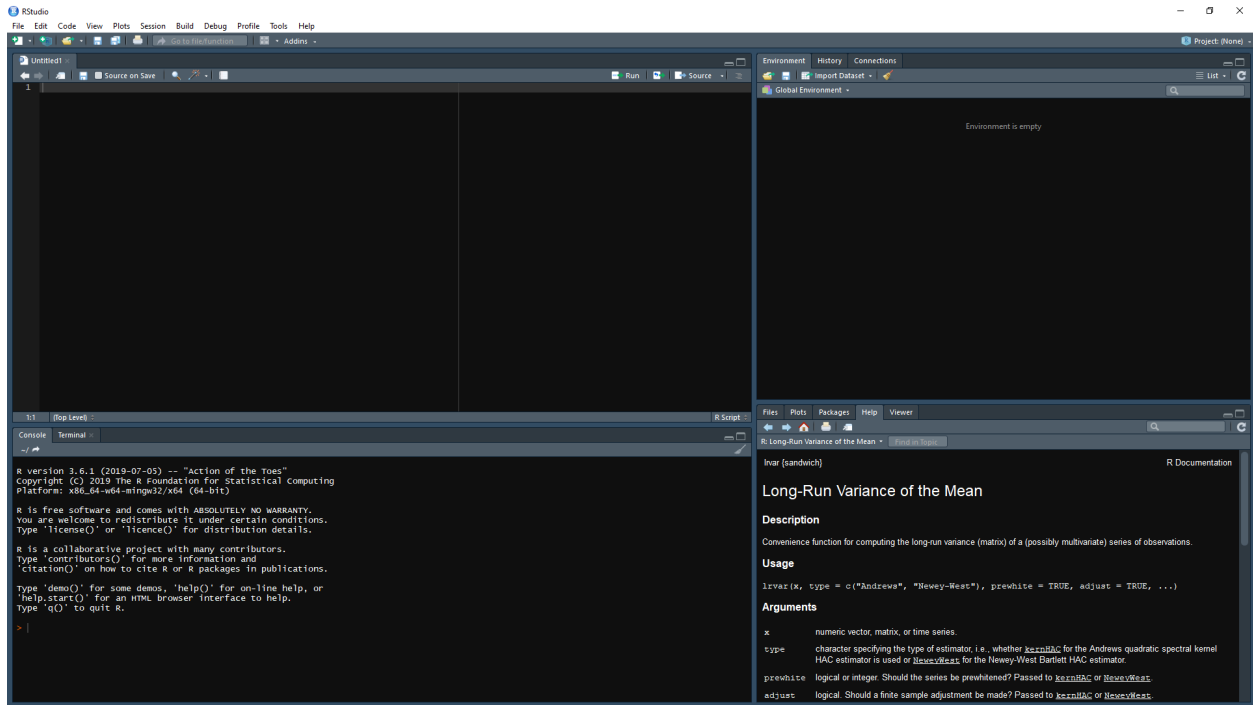


Figure 1: RStudio start

What just happened is RStudio executed programs (specifically `R CMD build` and `R CMD install`) that built and installed the package **account**, then restarted R and loaded in the package. Since the package is currently extremely minimal, there should have been no problem. That doesn't mean there's no problems, though. In fact, in its current form, this package would certainly not be accepted to CRAN. This isn't just because the package doesn't really do anything.

Press `Ctrl+Shift+E` to test the package. RStudio then runs `R CMD check`, which will run various tests on the package to make sure that it adheres to selected standards and appears to be an acceptably functional package. We get output resembling the following:

Near the bottom we should see the following:

In this run, there was a warning and a note. When checking packages, there are three types of flags: errors, warnings, and notes. Errors signify a problem that will cause the package to fail to build or install. Warnings appear for issues that won't necessarily cause the package to fail to build or install but are considered bad practice and need to be addressed. Finally, notes signify issues less serious than warnings; they call attention to those issues. Sometimes notes will be thrown just because a package is new.

If we were thinking of submitting our package to CRAN, there would have to be no errors or warnings at all when the package is checked via `R CMD check --as-cran` (which is a check with more tests to see the package meets CRAN standards). While there can be notes they need to be brought to a minimum, and you may need to defend your package in the presence of the note to the CRAN volunteer checking the submission. In general, though, get in the habit of dealing with errors, warnings, and notes immediately and to the best of your ability. Eliminate all errors and warnings and think about how to handle the notes.

In this case we see that a warning was thrown because `R CMD check` could not understand the package license. (The note is more mysterious; it could have been an issue local to the circumstances of my writing these notes at my grandparents' house in rural Idaho without Internet access. We can probably ignore it.) This brings us to the first important package file: **DESCRIPTION**.

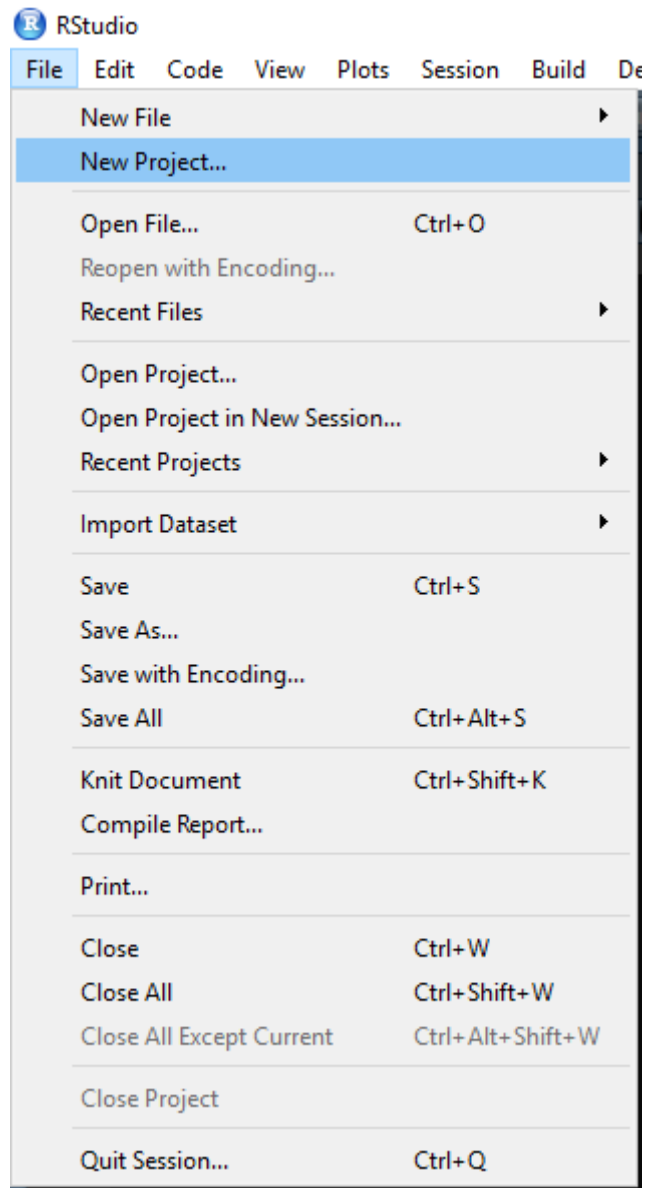


Figure 2: Click File->New Project

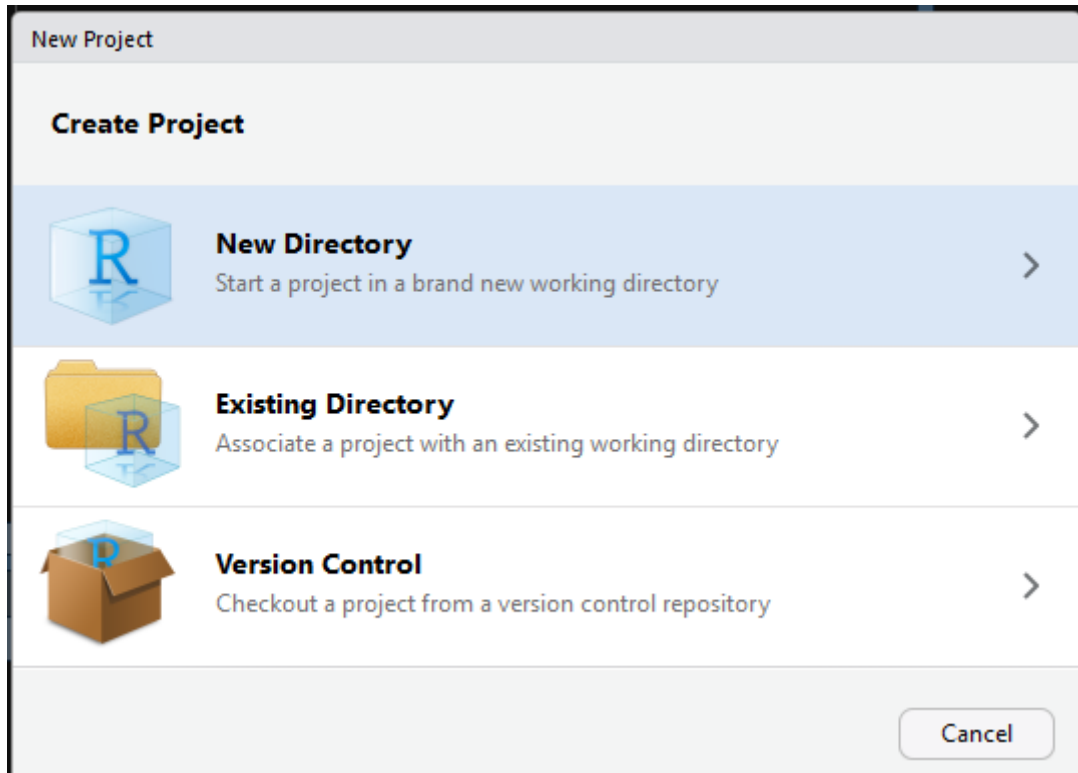


Figure 3: Click New Directory

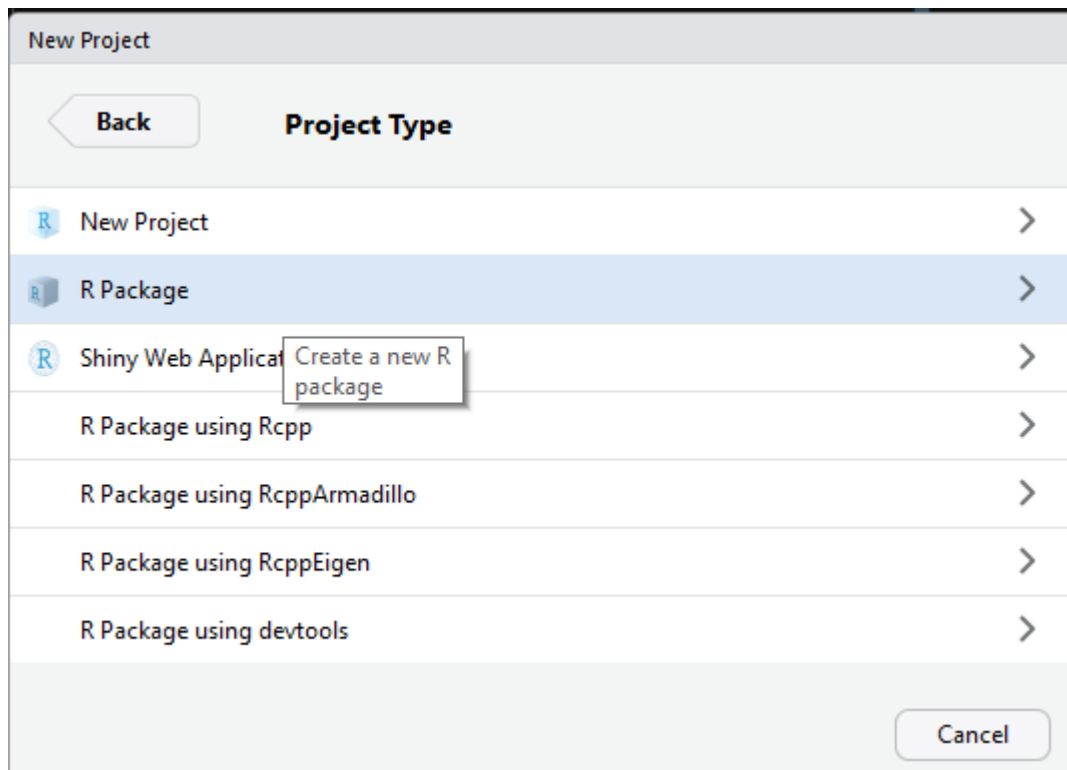


Figure 4: Click R Package

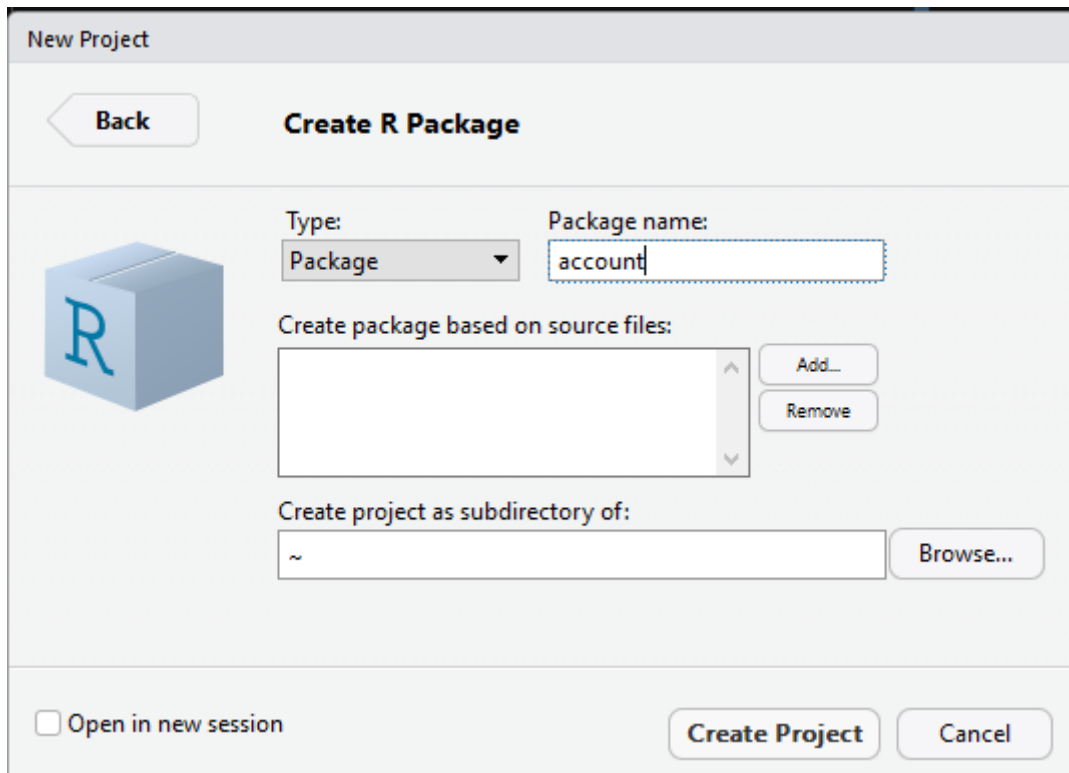


Figure 5: Name the package

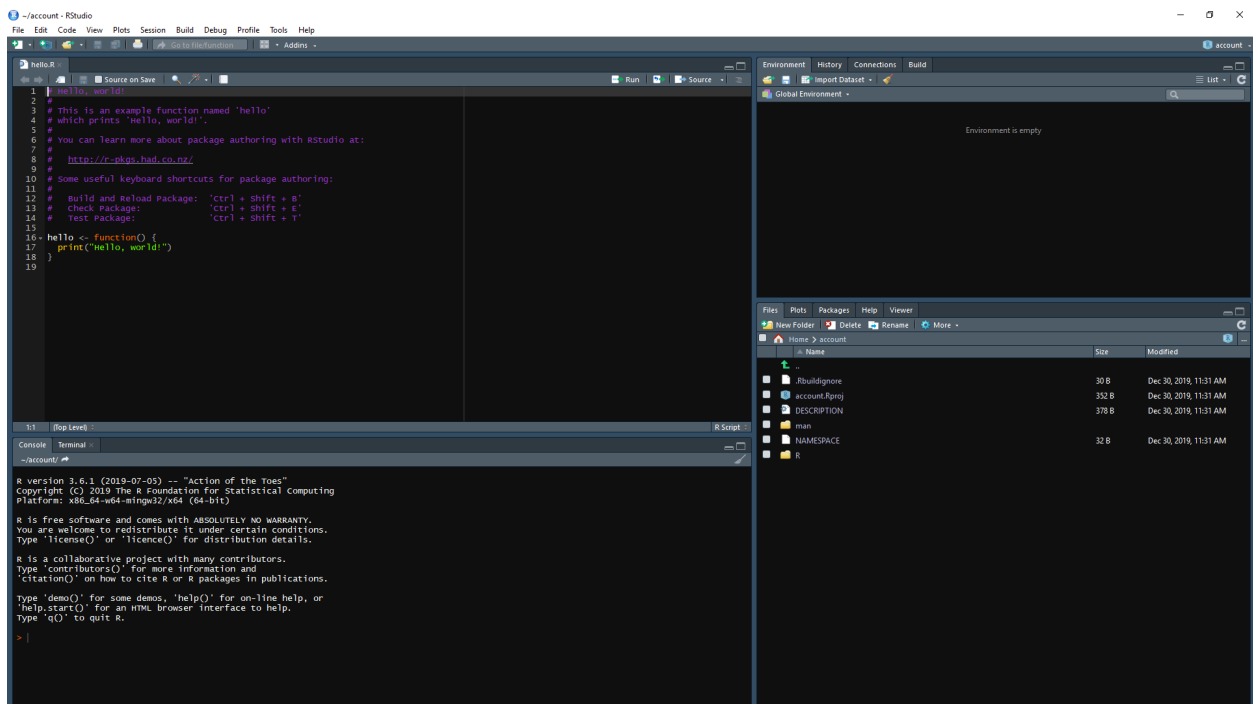
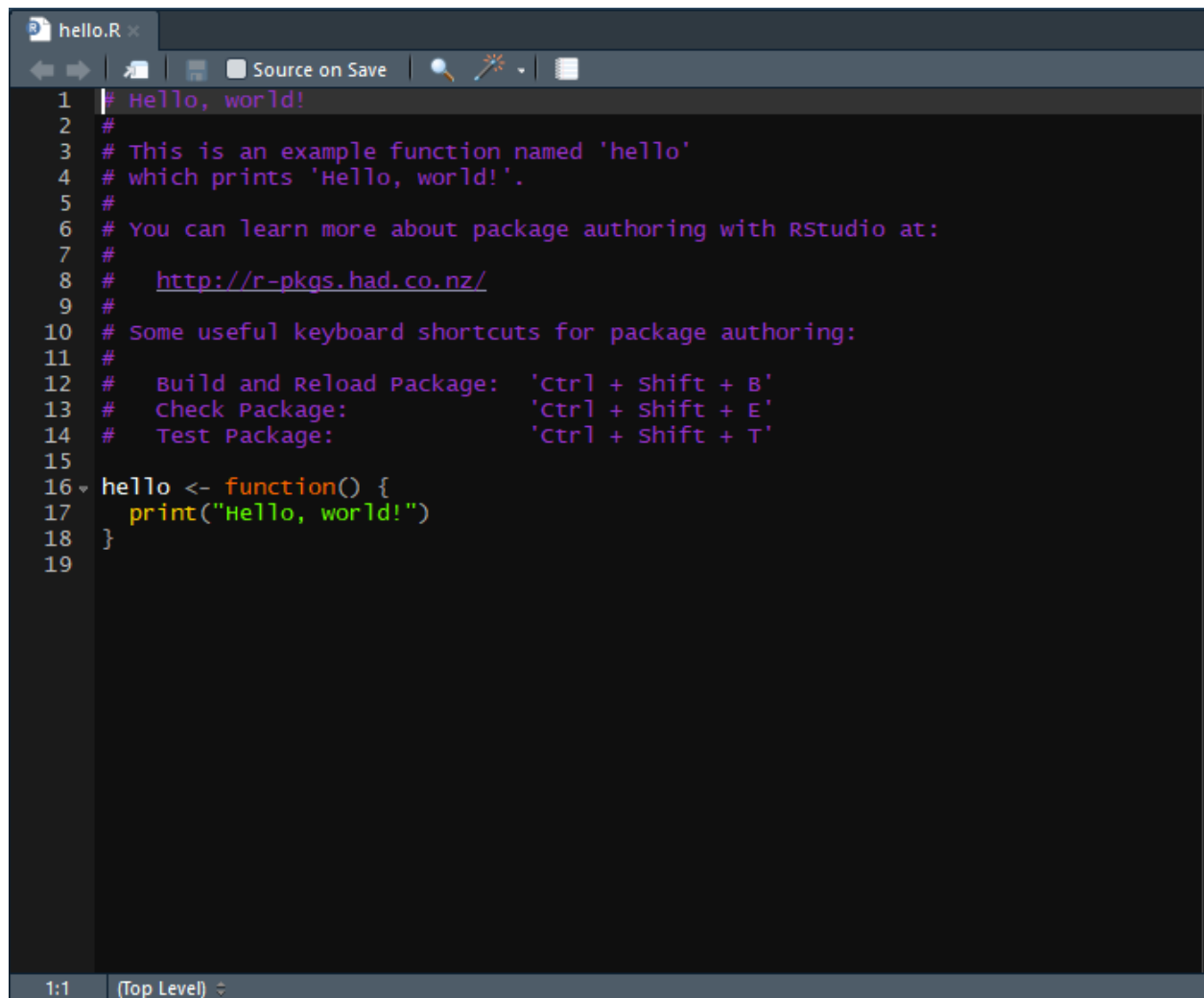


Figure 6: New RStudio screen



```
1 # Hello, world!
2 #
3 # This is an example function named 'hello'
4 # which prints 'Hello, world!'.
5 #
6 # You can learn more about package authoring with RStudio at:
7 #
8 # http://r-pkgs.had.co.nz/
9 #
10 # Some useful keyboard shortcuts for package authoring:
11 #
12 #   Build and Reload Package: 'Ctrl + Shift + B'
13 #   Check Package:           'Ctrl + Shift + E'
14 #   Test Package:            'Ctrl + Shift + T'
15 #
16 hello <- function() {
17   print("Hello, world!")
18 }
19
```

Figure 7: New R file

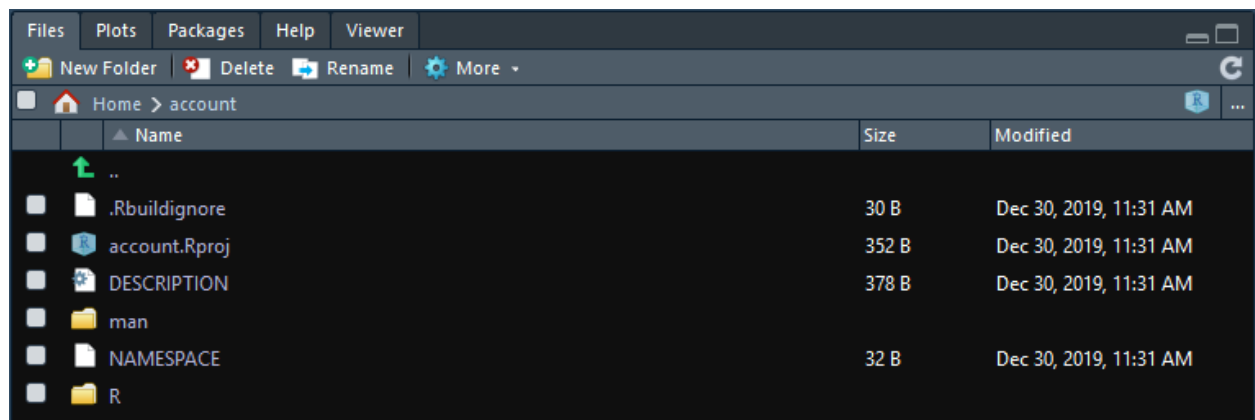
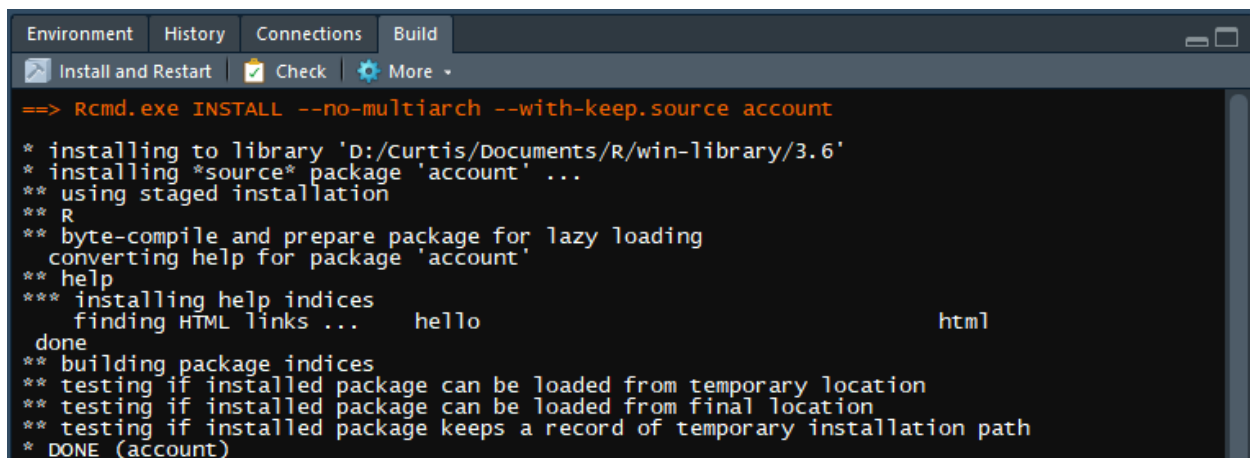
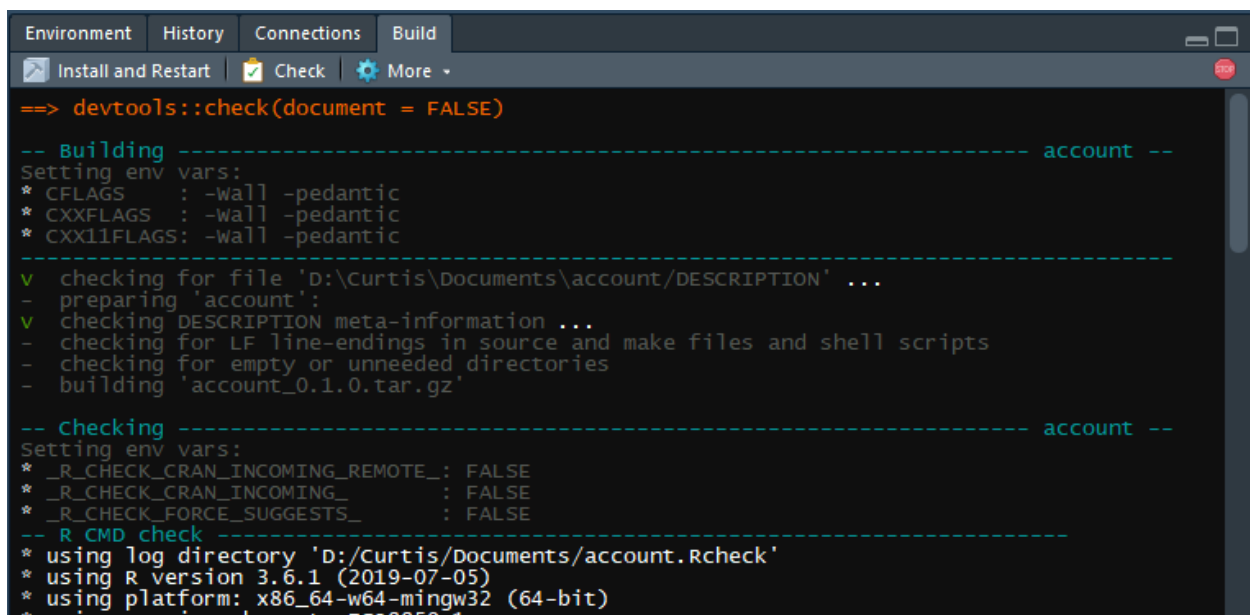


Figure 8: Package directory



```
Environment History Connections Build
Install and Restart Check More
==> Rcmd.exe INSTALL --no-multiarch --with-keep.source account
* installing to library 'D:/Curtis/Documents/R/win-library/3.6'
* installing *source* package 'account' ...
** using staged installation
** R
** byte-compile and prepare package for lazy loading
** converting help for package 'account'
*** help
*** installing help indices
    finding HTML links ...      hello      html
done
** building package indices
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (account)
```

Figure 9: Package build



```
Environment History Connections Build
Install and Restart Check More
==> devtools::check(document = FALSE)

-- Building ----- account --
Setting env vars:
* CFLAGS      : -Wall -pedantic
* CXXFLAGS    : -Wall -pedantic
* CXX11FLAGS  : -Wall -pedantic
-----
v checking for file 'D:/Curtis/Documents/account/DESCRIPTION' ...
- preparing 'account':
v checking DESCRIPTION meta-information ...
- checking for LF line-endings in source and make files and shell scripts
- checking for empty or unneeded directories
- building 'account_0.1.0.tar.gz'

-- Checking ----- account --
Setting env vars:
* _R_CHECK_CRAN_INCOMING_REMOTE_: FALSE
* _R_CHECK_CRAN_INCOMING_: FALSE
* _R_CHECK_FORCE_SUGGESTS_: FALSE
-----
-- R CMD check -----
* using log directory 'D:/Curtis/Documents/account.Rcheck'
* using R version 3.6.1 (2019-07-05)
* using platform: x86_64-w64-mingw32 (64-bit)
* using session charset: UTF-8
```

Figure 10: Package check

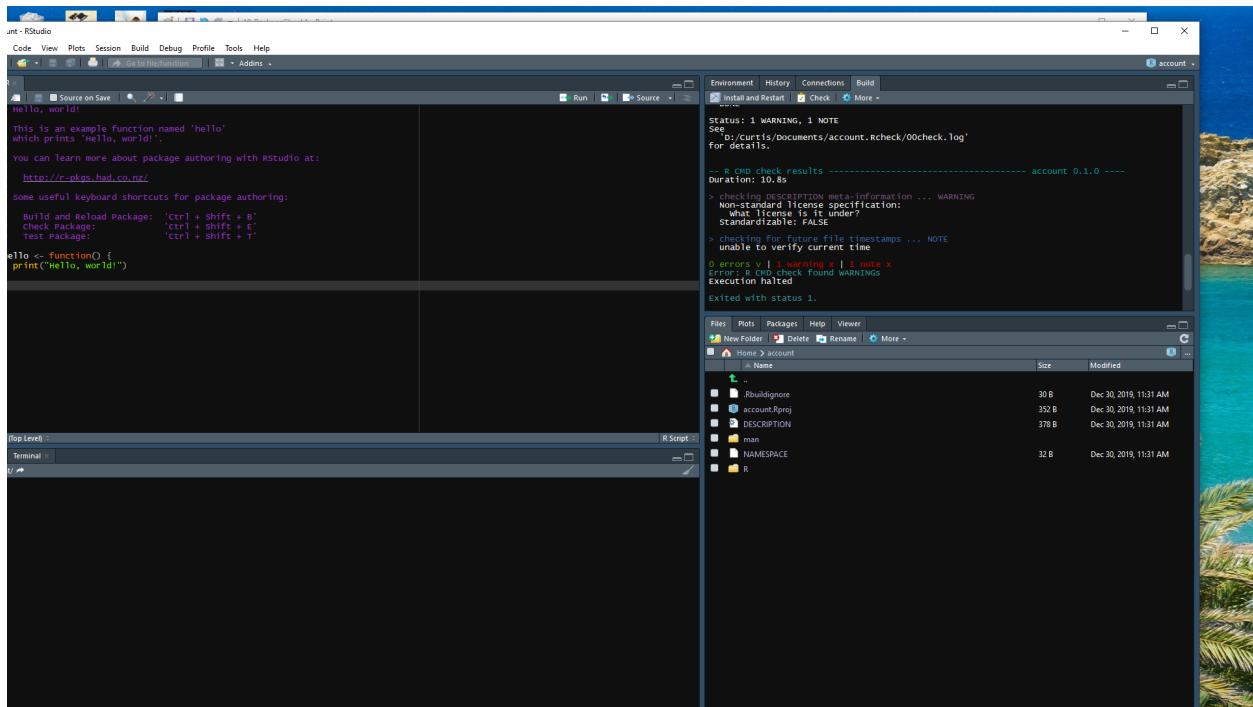


Figure 11: Package problems

DESCRIPTION

DESCRIPTION is simply a structured plain text file with the package’s metadata. This include the package’s name, description, version, author, maintainer, and dependencies. Below is the **DESCRIPTION** file generated by RStudio:

```
Package: account
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Who wrote it
Maintainer: The package maintainer <yourself@somewhere.net>
Description: More about what it does (maybe more than one line)
    Use four spaces when indenting paragraphs within the Description.
License: What license is it under?
Encoding: UTF-8
LazyData: true
```

Many of these fields are self-explanatory and we can easily fill them out. Some fields we may never need to edit (such as **Encoding** and **LazyData**). That said, let’s explain some of these fields:

- **Package:** The name of the package. Make it short and sensible. Bonus points if there’s some logic to the naming scheme you use for packages (such as all lower case, all upper case, lowerCamelCase, UpperCamelCase, etc.; programmers fight holy wars over naming conventions).
- **Type:** This is an R package.
- **Title:** The title of the package (a short description, not even a sentence).
- **Version:** The package version number. Version numbering matters and signals a lot about the state and stability of the package, so here are some conventions. When the package is initially born, the version number should be 0.0.0.9000. The first number in the version number indicates “major version” level, the second “minor version” level, and third “patch version” level. In this case we threw on a

fourth number, 9000, as a big number to get readers' attention. This number alerts any reader that this version of the package is currently under development and could change on even a daily basis; it's as unstable as a package can get. When we are no longer developing this version, we will drop the 9000 and simply use version 0.0.1 or 0.1.0, depending on our opinion on how developed the package is. The former number would signal that the package is in **alpha** development stage; important features in the package are missing and most of its features should not be considered stable. The latter indicates a **beta** development stage; the package can be safely used and the features there are considered somewhat stable, but the package is missing important features that should eventually be put in. A 1.0.0 release would be a **stable** release; all planned features are present and the interface is stable. As we update our package we would increment each of these three numbers depending on how significant the changes made were. We would change the last number for minor patches and bug fixes. Changing the middle number signifies noticeable changes in the package's functionality, but nothing drastic. Changing the first number indicates major changes in the package's functionality, perhaps even breaking backwards compatibility.

- **Author:** Give yourself credit! Also, provide an e-mail in angle brackets, like Joe Diamond <joe@arkhampi.com>.
- **Maintainer:** The initial author of the package may not be the current maintainer of the package (the person fixing bugs and releasing updates), but often the author and maintainer lines have a lot of overlap.
- **Description:** What does the package provide? What does it do? Why is it here? Put that information here.
- **License:** Under what legal license is the package released under? Under what legal conditions may others use, copy, or distribute this package? Is this package proprietary or free and open source? If this is a personal package that you don't plan on releasing to the world, you're welcome to ignore this field. But you probably shouldn't, though; if you don't provide a license then it's harder for you yourself to safely share your package with others. R recognizes a number of values for this field, including common open source licenses (GPL-2, GPL-3, MIT, CC0) and also the value `file LICENSE` (or in the case of the MIT license, `MIT + file LICENSE`, since MIT is merely a license template). If `file LICENSE` is used, R will expect a plain text file in the base directory of the package called `LICENSE`, containing the licensing information. If your package is proprietary, you would use this option. Note that CRAN needs some open source license in order to distribute your package.
- **Encoding:** The file encoding. You'll edit this if you know what you're doing.
- **LazyData:** Should data be lazy loaded? See above.

Some additional fields not shown above but often appearing in packages:

- **Imports:** The other packages this package *must* have in order to function. We will be leaving this field blank for now, but below is an example of how it's used:

Imports:

```
dplyr,
ggplot,
data.table
```

- **Depends:** Dependencies of the package. We should put at minimum R (`>= current.version.number`), but it's possible that packages will be listed here in a manner similar to how they were listed in **Imports**. The difference between listing a package under **Depends** and listing one under **Imports** is that a package listed under **Depends** will also be loaded to the namespace via a command like `library()` in addition to our package. **Imports**, in comparison, does not attach the other package to the namespace but loads without attaching. (The package is available but users can't access its content without `::`.)
- **Suggests:** Similar to **Imports**, but the packages listed are not *essential*; the package may work better if these are present, though.
- **VignetteBuilder:** Packages that build vignettes (which we will discuss later).
- **RdMacros:** Packages with macros for `.Rd` files. See **Encoding** above. (I only mention it because the packages I used for citation management require editing this field.)

Many other fields for the DESCRIPTION file exist, though they don't need to be present in order for the file to be valid. Look them up if you need them.

We already have enough information to update our DESCRIPTION file, though. Here's the new DESCRIPTION; copy and paste the text below to replace the old file.

```
Package: account
Type: Package
Title: Data Structures for Managing Banking Accounts
Version: 0.1.0
Author: Curtis Miller <cmiller@math.utah.edu>
Maintainer: Curtis Miller <cmiller@math.utah.edu>
Description: This package implements an S3 class for representing and managing
             bank accounts, with accompanying helper functions and methods.
Depends: R (>= 3.6.0)
License: CCO
Encoding: UTF-8
LazyData: true
```

Other Files in the Base Directory

I next describe other important files in the package's base directory. All of these are plain text files. **NAMESPACE** and **.Rbuildignore** are essential, while others don't *need* to be present, though including them is considered good practice.

NAMESPACE

The NAMESPACE file is a plain text file with content resembling R code. This file controls what objects or packages from other packages are imported and what the package should make available to users loading the package via `library()`. The file defines what objects are **public** (available on package attachment or via `::`) and what objects are **private** (used only internally in the package, not intended to be used by package users, and can be used only via `:::`).

If you are using **roxygen2** you may not need to edit this file yourself. In this lecture we won't use **roxygen2**. That said, we will edit this file later.

.Rbuildignore

This file lists regular expression (regex) patterns identifying files that R should ignore when building the package. R will assume that every file in the package directory matters to the package. If it doesn't recognize a file, an error will be thrown and the package won't build. The answer though isn't that we shouldn't put irrelevant files in the package directory, but instead that we should inform R that these files need to be ignored during package building. So we list those files here.

RStudio populated this file with the following:

```
^.*\.Rproj$
^\.Rproj\.user$
```

The first line says ignore any file that ends with **.Rproj** (an RStudio project file) and the second to ignore the file **.Rproj.user** in the base directory. These are regular expressions. Many resources exist for learning regular expressions but for now I will give you two basic ways to add files:

- If you want to list a specific file to be ignored, include the line `^filename$`. You *must* include `^` at the beginning and `$` at the end. `^` in regex means "beginning of line" and `$` means "end of line". Failing to include these could lead to any file with the string **filename** in its path being matched and thus excluded from the build. Above **filename** is assumed to be in the base directory of the package; if you want to list a file in a subdirectory, use `^path/to/filename$` instead (always use `/` for file paths, even

on Windows systems, where `\\` is customary). If your file name include a `.` (even if it's to separate an extension from the main file name), escape the `.` with a backslash, like so: `^filename\\.ext$` to match file `filename.ext`. (`.` has a special meaning in regex, meaning “zero or more instances of”.)

- If you want to exclude an entire directory, use `^directory/` or `^path/to/directory/`.
- If you want to exclude all files with a certain extension, use `^.*\\.ext$`, where `ext` is the extension you want to exclude (such as `txt` or `csv`). In regex, `.*` means “match zero or more things”, and `\\.` means a literal period. This line will match files not just in the base directory but any subdirectory in the package.

For more nuanced patterns, study regex, and realize that the string being matched is the entire file path beyond the package directory.

We will be adding more files to the package directory we want ignored, so we will add the following lines to `.Rbuildignore`:

```
^README\\.md$
^NEWS\\.md$
```

README.md

This file is a Markdown file describing the package, going into more depth than what was written in `DESCRIPTION`. This file is meant to be read by humans, and it's formatted using Markdown syntax. (R Markdown is a special flavor of Markdown; if you can write documents in R Markdown, you can write Markdown files.) This file is meant for new users. In `README.md` I recommend:

- Describing the package, what it's for, and why people should use it;
- How people can get and install the package (and any prerequisites); and
- How people should use the package, including perhaps example code.

`README.md` should not be too long and any examples included in it as simple as possible. This will be the `README.md` file of our package:

```
# account
*Version 0.1.0*
```

```
**account** is a package providing functions and class definitions for handling
accounts resembling traditional banking accounts. If installing from the base
directory of the package and the **devtools** package is installed, **account**
can be installed using `devtools::install()`.
```

Below are examples of using ****account****.

```
```r
library(account)

Creating an account
my_account <- account(title = "My Personal Account",
 owner = "Joe Diamond",
 start = "1925-01-01",
 init = 1000)

Adding a transaction to the account
my_account <- my_account + account_transaction("1925-01-02",
 -10,
 "Ammo for guns")

print(my_account)

Example account object
```

```

accdemo
summary(accdemo)
plot(accdemo)
```

```

NEWS.md

This file is a Markdown document intended for existing users of the package. This file tracks changes to the package made with each version of the package, such as any functions that were changed or implementation changes made between versions, along with any bug fixes.

Our package is brand new so there's not much news. We will get NEWS.md started with the following:

```
# account News
```

```
## Version 0.1.0
```

```
*2019-12-31*
```

```
---
```

- Package **account** created
- S3 classes ``account``, ``transaction``, and ``summary.account`` with associated constructor functions defined
- ``print()``, ``summary()``, ``sort()``, ``plot()``, and ``+`` methods for ``account`` objects created
- Generic function ``as.account()`` created
- ``print()`` and ``as.account()`` methods for ``transaction`` objects created
- ``print()`` method for ``summary.account`` objects created
- ``account`` helper functions ``account_title()``, ``account_owner()``, ``account_transactions()``, with associated assignment versions (such as ``account_title<-(())``) created
- ``account`` helper functions ``bad_Initial()``, ``all_transactions()``, ``account_dates()``, ``account_trans_amounts()``, ``account_memos()``, ``transaction_count()``, ``account_new_transaction()``, ``account_delete_transaction()``, and ``is.account()`` created
- ``transaction`` helper functions ``transaction_date()``, ``amount()``, ``memo()``, with associated assignment versions (such as ``amount<-(())``) created
- ``transaction`` helper function ``is.transaction()`` created
- Function ``read_csv_account()`` created
- Example ``account`` object ``accdemo`` created

Package Directory Structure

R packages use a well-defined directory structure, with files in certain files being used in specific ways. Below are descriptions of the directories and what goes in them.

R/

The R/ directory is the most important directory of the package. It contains the code for all the R objects you're providing the users of your package. The files in the directory should all be .R files (that is, R source files, or "scripts", even though these scripts should do nothing other than create objects and maybe document them).

If we wanted we could put all our package code in a single .R file, we could. In fact, if all our package does is provide a couple functions, this may be a perfectly reasonable organization. But if your package is more involved, you should spread your code out over multiple files, where the code in a file serves a common purpose. The order in which these files are loaded should not matter. If it does, something is terribly wrong.

RStudio go you started with the file `hello.R` in the `R/` directory, listed below:

```
# Hello, world!
#
# This is an example function named 'hello'
# which prints 'Hello, world!'.
#
# You can learn more about package authoring with RStudio at:
#
#   http://r-pkgs.had.co.nz/
#
# Some useful keyboard shortcuts for package authoring:
#
#   Build and Reload Package:  'Ctrl + Shift + B'
#   Check Package:            'Ctrl + Shift + E'
#   Test Package:              'Ctrl + Shift + T'

hello <- function() {
  print("Hello, world!")
}
```

You're expected to delete this file and put something useful in the `R/` directory. In our case, we will take the functions we wrote for modeling banking accounts from a previous lecture (plus a couple others to round the package out). The files and their code are listed below. I've chosen to organize the files around the classes and function augmenting them. Copy them verbatim.

transaction.R This file includes `transaction()`, the function for making `transaction`-class objects, along with associated methods.

```
transaction <- function(date, amount, memo = "") {
  obj <- list(date = as.Date(date),
             amount = as.numeric(amount),
             memo = as.character(memo)
  )
  class(obj) <- "transaction"
  obj
}

is.transaction <- function(x) {class(x) == "transaction"}

print.transaction <- function(x, space = 10) {
  tdate <- as.character(x$date)
  datestring <- paste0("    ", tdate, ":")
  formatstring <- paste0("%+", space[[1]], ".2f") # See sprintf() to explain
  amountstring <- sprintf(formatstring, x$amount)
  if (x$memo == "") {
    memostring <- ""
  } else {
    memostring <- paste0("(", x$memo, ")")
  }
  cat(datestring, amountstring, memostring, "\n")
}

as.account.transaction <- function(x, title = "Transaction", owner = "noone") {
  res <- account(start = transaction_date(x), init = amount(x), owner = owner,
```

```

        title = title)
memo(account_transactions(res)[[1]]) <- memo(x)
res
}

```

transactionHelpers.R This file includes functions that are meant to work with **transaction**-class objects.

```

transaction_date <- function(trns) {
  trns$date
}

`transaction_date<-` <- function(trns, value) {
  trns$date <- as.Date(value)
  trns
}

amount <- function(trns) {
  trns$amount
}

`amount<-` <- function(trns, value) {
  trns$amount <- as.numeric(value)
  trns
}

memo <- function(trns) {
  trns$memo
}

`memo<-` <- function(trns, value) {
  trns$memo <- as.character(value)
  trns
}

```

account.R This file includes **account()**, the function for making **account**-class objects, along with associated methods.

```

account <- function(start, owner, init = 0, title = "Account") {
  obj <- list(
    title = as.character(title),
    owner = as.character(owner),
    transactions = list(transaction(start, init, "Initial"))
  )
  class(obj) <- "account"
  obj
}

is.account <- function(x) {class(x) == "account"}

sort.account <- function(x, decreasing = FALSE, ...) {
  # There might be multiple entries with memo Initial; design code for that
  memo_Initial <- which(account_memos(x) == "Initial")
  if (length(memo_Initial) == 0) {
    date_Initial <- min(account_dates(x))
  }
}

```



```

    true_initial <- which.min(account_dates(x))
  } else {
    date_initial <- account_dates(x)[memo_initial]
    true_initial <- which((account_dates(x) == min(date_initial)) &
                          (account_memos(x) == "Initial"))
  }
  tcount <- transaction_count(x)
  nix <- (1:tcount)[-true_initial]
  ordered_nix <- nix[order(account_dates(x)[nix], decreasing = decreasing, ...)]
  if (decreasing) {
    final_order <- c(ordered_nix, true_initial)
  } else {
    final_order <- c(true_initial, ordered_nix)
  }
  account_transactions(x) <- account_transactions(x)[final_order]
  x
}

print.account <- function(x, presort = FALSE) {
  if (presort) {
    x <- sort(x)
  }
  cat("Title:", account_title(x))
  cat("\nOwner:", account_owner(x))
  cat("\nTransactions:\n-----\n")
  for (t in account_transactions(x)) {
    print(t)
  }
}

`+.account` <- function(x, y) {
  account_transactions(x) <- c(account_transactions(x), account_transactions(y))
  sort(x)
}

plot.account <- function(x, y, ...) {
  if (bad_initial(x)) stop("Malformed account object")
  x <- sort(x)
  unique_dates <- unique(account_dates(x))
  date_trans_sum <- sapply(unique_dates, function(d) {
    idx <- which(account_dates(x) == d)
    sum(account_trans_amounts(x)[idx])
  })
  plot(unique_dates, cumsum(date_trans_sum), type = "l",
       main = account_title(x), xlab = "Date", ylab = "Balance", ...)
}

plot(acc)

```

accountHelpers.R This file includes functions that are meant to work with `account`-class objects.

```

account_title <- function(account) {
  account$title

```

```

}

`account_title<-` <- function(account, value) {
  account$title <- as.character(value)
  account
}

account_owner <- function(account) {
  account$owner
}

`account_owner<-` <- function(account, value) {
  account$owner <- as.character(value)
  account
}

account_transactions <- function(account) {
  account$transactions
}

`account_transactions<-` <- function(account, value) {
  account$transactions <- value
  account
}

all_transactions <- function(account) {
  all(sapply(account_transactions(account), is.transaction))
}

account_dates <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  Reduce(c, lapply(account_transactions(account), transaction_date))
}

account_trans_amounts <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  sapply(account_transactions(account), amount)
}

account_memos <- function(account) {
  if (!all_transactions(account)) stop("Malformed account object")
  sapply(account_transactions(account), memo)
}

transaction_count <- function(account) {
  length(account_transactions(account))
}

bad_initial <- function(account) {
  if (!all_transactions(account)) stop("Not all transactions of right class")
  if (!("Initial" %in% account_memos(account))) stop("No Initial transaction")

  memo_initial <- which(account_memos(account) == "Initial")

```

```

date_Initial <- min(account_dates(account)[memo_Initial])
sort(account_dates(account))[1] < date_Initial
}

account_new_transaction <- function(...) {
  as.account(transaction(...))
}

account_delete_transaction <- function(account, date = NULL, memo = NULL) {
  if (is.null(date) && is.null(memo)) {
    stop("Must specify at least one of date or memo")
  }

  if (!is.null(date)) {
    filter_dates <- which(account_dates(account) == as.Date(date))
  } else {
    filter_dates <- 1:transaction_count(account)
  }

  if (!is.null(memo)) {
    filter_memos <- which(account_memos(account) == memo)
  } else {
    filter_memos <- 1:transaction_count(account)
  }

  final_filter <- intersect(filter_dates, filter_memos)
  if (length(final_filter) == 0) {
    warning("No transactions with date/memo combination")
    return(account)
  } else {
    account_transactions(account)[final_filter] <- NULL
    return(account)
  }
}

```

summary.account.R This file includes `summary.account()`, the function for making `summary.account`-class objects, along with associated methods. (I treat `summary.account()` as both a method of `summary()` and as a constructor of an object.)

```

summary.account <- function(object, recent = 5) {
  if (bad_Initial(object)) warning("account object malformed!")
  res <- list()
  res$title <- account_title(object)
  res$owner <- account_owner(object)
  res$balance <- sum(account_trans_amounts(object))
  res$tcount <- transaction_count(object)
  res$rtrans <- account_transactions(sort(object,
                                          decreasing = TRUE)
                                   )[1:min(recent, res$tcount)]

  class(res) <- "summary.account"
  res
}

```

```

print.summary.account <- function(x, prefix = "\t") {
  cat("\n")
  cat(strwrap(x$title, prefix = prefix), sep = "\n")
  cat("\n")
  cat("Owner:  ", sprintf("%20s", x$owner), "\n", sep = "")
  cat("Transactions:  ", sprintf("%13s", x$tcount), "\n", sep = "")
  cat("Balance:  ", sprintf("%18.2f", x$balance), "\n", sep = "")
  cat("\nRecent Transactions:\n-----\n")
  for (t in x$rtrans) {
    print(t)
  }
}

```

Generics.R This file defines generic functions, along with any methods for external classes.

```

as.account <- function(x, ...) UseMethod("as.account")

as.account.data.frame <- function(x, title = "Account", owner = "noone",
                                   datecol = 1, amountcol = 2, memocol = 3) {
  if (length(x) < 3) stop("Too few columns to contain valid transactions")
  dat <- data.frame("date" = as.character(x[[datecol]]),
                    "amount" = as.numeric(x[[amountcol]]),
                    "memo" = as.character(x[[memocol]]),
                    stringsAsFactors = FALSE)
  acc <- Reduce(`+.account`, lapply(1:nrow(dat), function(i) {
    r <- dat[i, ]
    account_new_transaction(date = r$date, amount = r$amount,
                           memo = r$memo)
  }))
  account_owner(acc) <- owner
  account_title(acc) <- title
  acc
}

as.account.matrix <- function(x, ...) {
  as.account(as.data.frame(x, stringsAsFactors = FALSE), ...)
}

```

read_csv_account.R File containing the function `read_csv_account()`.

```

read_csv_account <- function(file, title = "Account", owner = "noone",
                              datecol = 1, amountcol = 2, memocol = 3, ...) {
  dat <- read.csv(file = file, ...)
  as.account(dat, title = title, owner = owner, datecol = datecol,
             amountcol = amountcol, memocol = memocol)
}

```

When we create these files our R/ directory in RStudio should look like so:

Now if we build the package and load it all these files are available to us.

Remember: these files should only create R objects (usually functions). These are not executable scripts that themselves run functions. That said, with **roxygen2**, you will likely be writing the documentation in these files too.

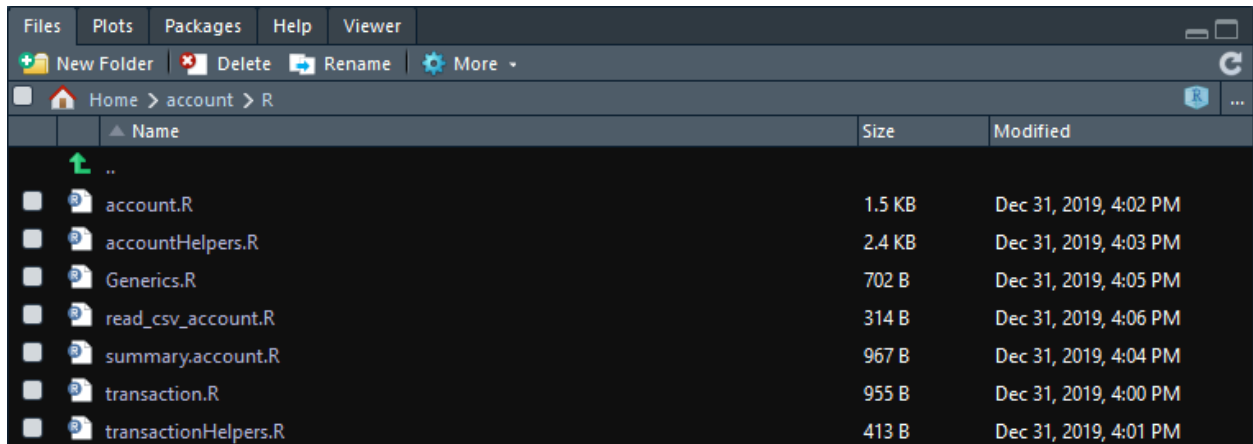


Figure 12: R directory

data/

The `data/` directory contains data sets used in the package. These could be data sets used in examples, they could be important for how certain functions work, or they could even be the reason for the existence of the package. Data sets should be stored in R data files with the `.RData` extension. The objects stored in these files can be just about anything.

Our package does have an example object, `accddata`. We will create this object ourselves. Below I've created a CSV file containing our example account.

```
date,amount,memo
1925-10-08,0.55,Initial
1925-10-14,694.81,Deposit
1925-10-14,-200.00,Withdrawal
1925-10-16,-21.25,Fee (service)
1925-10-16,-1.50,Check to General Store
1925-10-20,-2.99,Electric Bill
1925-10-21,-300.00,Withdrawal
1925-10-22,-100.00,Withdrawal
1925-10-23,-29.08,Check to General Store
1925-10-24,2.99,Deposit
1925-10-27,-6.77,Check to American Telephone & Telegraph
1925-10-28,694.81,Deposit
1925-10-30,50.00,Transfer from savings
1925-11-03,-33.55,Check to Arkham Insurance
1925-11-03,-100.00,Check to Joey Vigil
1925-11-06,-710.49,Mortgage payment
1925-11-07,-5.00,Fee (overdraft)
1925-11-08,-5.00,Fee (monthly)
1925-11-10,150.00,Transfer from savings
1925-11-14,694.81,Deposit
1925-11-14,-200.00,Withdrawal
1925-11-16,-21.25,Fee (service)
1925-11-17,-2.36,Check to General Store
1925-11-17,-40.00,Check to Ye Olde Magick Shoppe
1925-11-18,-1.22,Check to Velma's Diner
1925-11-20,-2.99,Electric Bill
1925-11-22,-200.00,Withdrawal
```

```

1925-11-23,-30.44,Check to General Store
1925-11-27,-6.81,Check to American Telephone & Telegraph
1925-11-28,694.81,Deposit
1925-11-30,-19.65,Check to General Store
1925-12-01,-150.00,Withdrawal
1925-12-03,-33.55,Check to Arkham Insurance
1925-12-06,-710.49,Mortgage payment
1925-12-08,-5.00,Fee (monthly)

```

Let's create a directory, `data-raw`, in the main directory of the package, then save this data in the file `accdata.csv`. Also, add the following line to `.Rbuildignore`:

```
^data-raw/
```

This directory simply contains raw data. We don't absolutely need it, but it's nice to have these files around in case we need them. Create directory `data/` in the main package directory. Then try executing the following (be sure the file path makes sense relative to your working directory):

```

accdata <- read_csv_account("data-raw/accdata.csv",
                             title = "Personal Checking Accounting",
                             owner = "Joe Diamond")
save(accdata, file = "data/accdata.RData")

```

Now that the file `accdata.Rda` has our object `accdata` and has it saved in the `data/` directory, when we load a fresh R session, we should be able to use `accdata` when we load the **account** package. You can confirm this by issuing the command `rm(list = ls())` (to clear everything in the global namespace) and then in RStudio typing `Ctrl+Shift+F10` (on Windows/Linux), then rebuilding and loading the **account** package.

man/

The `man/` directory contains documentation for R objects. These are generally `.Rd` files, which are plain text files filled with syntax that strongly resembles LaTeX. RStudio populated this directory with a file already, `hello.Rd`, listed below:

```

\name{hello}
\alias{hello}
\title{Hello, World!}
\usage{
hello()
}
\description{
Prints 'Hello, world!'.
}
\examples{
hello()
}

```

This is the documentation for the function `hello()` that we deleted. If you were to type `?hello` in the console, you would see a rendered version of this file.

We will be using **roxygen2** for managing documentation, in which case we can mostly ignore the contents of this directory. The packages we will be using will automatically populate this directory. That said, we should delete `hello.Rd`, since we don't want documentation for a function that doesn't exist.

vignettes/

The `vignettes/` directory contains files for generating **vignettes**, which is long-form documentation of a package. In short, a vignette is like a tutorial, paper, or book chapter. This is where extensive tutorials would

be placed, or explanations for statistical methods or algorithms used in the package. Many of the packages you work with have vignettes. Try typing `browseVignettes("roxygen2")`, `browseVignettes("devtools")`, or `browseVignettes("dplyr")` (assuming you have these packages installed). If you want to learn more about how to use a package, vignettes are a good place to start.

The documentation that we will provide with functions serves as user manuals that should explain everything a user needs to be able to use the function, explaining what each parameter does and the results. Vignettes, in comparison, contain big picture documentation. The examples provided with functions are often terse, enough to see how the function should be used. Vignettes may demonstrate package use with complete analyses.

We can choose a vignette engine that controls how vignettes are rendered. Many package authors are academics already familiar with LaTeX and so prefer writing vignettes in LaTeX. However, if you're not interested in learning LaTeX (you should, though; it's really useful), you can tell R that you want **knitr** to be the vignette engine and write your documentation in R Markdown.

Let's create a vignette introducing our package. We will make the following changes:

1. Create in the package's base directory a `vignettes/` directory.
2. In the `DESCRIPTION` file, add the following lines:

```
Suggests: knitr
```

```
VignetteBuilder: knitr
```

3. In the new `vignettes/` directory, create the file `IntroToaccount.Rmd`.

Copy and paste the following into the file `IntroToaccount.Rmd`. This will be our basic vignette.

```
---
title: "Intro to account"
author: "Curtis Miller"
date: "2020-02-13"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Intro to account}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

```
# Introduction
```

The **account** package was developed to introduce R S3 programming. It implements tools useful for modelling banking accounts.

Try typing the following after installing the package:

```
```r
library(account)
acc <- account(start = "2020-01-01", title = "My Account", init = 1000,
 owner = "John Smith")
print(acc)
```
```

```
*To be continued...*
```

When we knit this file, the vignette should build when the package builds and we should see it when we type `browseVignettes("account")`. However I have personally had issues with vignettes apparently not being added when I build and install a package on Windows systems using **Ctrl+Shift+B**. You may need to build

the package source and manually install it yourself to see that vignettes are in fact installed correctly and for `browseVignettes()` to work. See the section “Distributing Packages” section for more details.

tests/

The **tests/** directory contains tests that will be run when the package is checked. We will discuss test writing in a later lecture, when we discuss the **testthat** package.

Other Directories

Other important directories that may appear in packages:

- **inst/** contains files that should be copied into the top-level package directory when the package is installed. I myself have used this directory to put files containing references for citations in function documentation. The **CITATION** file, containing information on how to cite the package, is put here as well.
- **src/** contains the source code for compiled code from languages such as C/C++. This is beyond the scope of this course. (I have written C++ code for my packages and I’m far from alone; this directory is definitely used.)
- **exec/** contains executable scripts. These might be needed for the package itself to function, but I have used this mostly as a place to put executable R scripts used in research projects.
- **po/** contains translations for messages. If you need your package to also be friendly to Chinese users, you may be interested in working with this directory. But it’s beyond the scope of this course.
- **tools/** contains auxiliary files needed for configuration or generating scripts.
- **demo/** contains package demos, which are **.R** files that *are* executable (unlike those in **R/**). Demos are basically long examples. Vignettes and examples are better.

Things to Never Do in Packages

Writing code for packages is different from writing non-package code. If you’re not going to distribute your package, you could perhaps ignore these rules, but breaking these rules would prevent the package from being accepted to CRAN, as breaking these rules is considered anti-social.

- Never load another package using `library()` or `require()` in your package. Dependencies on other packages needs to be handled via **Depends** and **Imports** in the **DESCRIPTION** file.
- Never call `q()` or `quit()` in a package. This will quit the user’s session.
- Don’t start external software in examples or tests unless the software is explicitly closed afterwards.
- Don’t write to the user’s home filesystem or anywhere else in the filesystem save for the temporary directory R creates for itself when running. Don’t install into the system’s R installation either.
- Do not modify the global environment.

Read the CRAN repository policies to see what other behaviors might be consider “anti-social” and should not be done in packages.

Distributing Packages

When we tell RStudio to build and install a package via hotkeys, RStudio is taking the following steps:

1. It builds a package binary, the source file, a **.tar.gz** file, containing the package.
2. It installs the package from this file.

R packages are distributed via **.tar.gz** files. Users of Linux should be familiar with these files, known as **tarballs**; they’re simply compressed files containing other files, similar to **.zip** files that probably most of you have used. In fact, when packages are installed via `install.packages()`, what actually happens is R downloads the corresponding tarball from CRAN and installs it, running the system commands (specifically, R CMD `install`) necessary to install the package.

Packages exist for sharing and distributing code, so we need to learn how to produce the installation tarball. In RStudio click on the **Build** menu, then click **Build Source Package**.

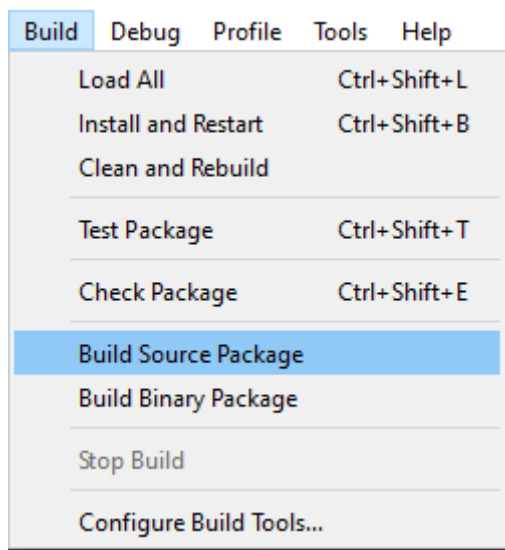


Figure 13: Build source

R will then build a tarball called `account_0.1.0.tar.gz` in the directory hosting the directory we've been using to build the package. We can in fact install the package in this file directly by clicking **Tools** in RStudio then clicking **Install Packages**.

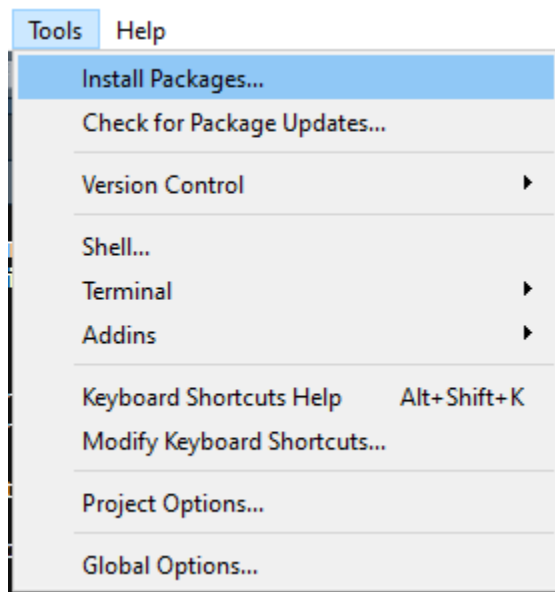


Figure 14: Install packages

When we do a popup window will appear. Tell R to install the package from a package archive file. This may prompt another popup window to appear; if not, click **Browse**. Select the file in the resulting file browser, or you can type the file path directly if a file browser was not used.

(This workflow installs any R package in the form of a tarball, not just ones we build.) After installation the package should be available for use.

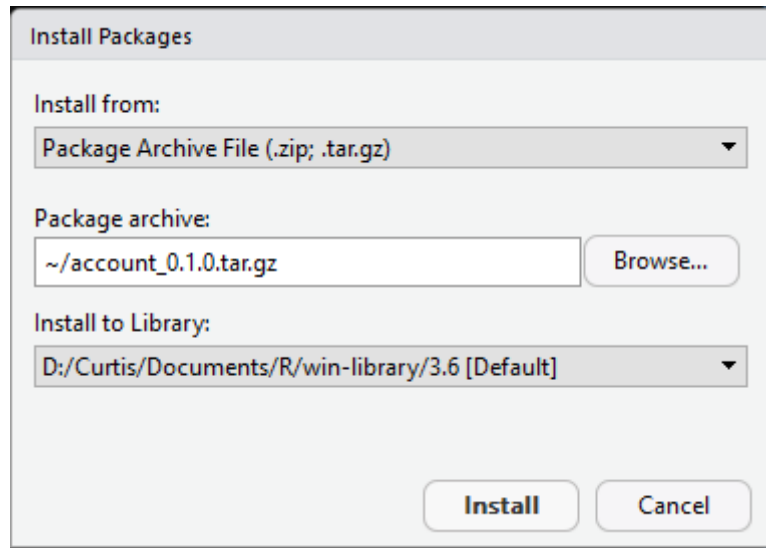


Figure 15: Install packages popup

If you're interested in submitting your package to CRAN, you will need to submit the tarball to them. But before you submit a package to CRAN, you should test the package with the CRAN checks and make sure that there were no errors or warnings in the check, and be prepared to explain any notes that are thrown as well (if they cannot be eliminated). We will not discuss submitting packages to CRAN in depth in this course.