

# CSHS Workshop: R for hydrologists

## Functions, projects and packages

CWRA 2022

Kevin Shook

Canadian Society for Hydrological Sciences (CSHS)

June 5, 2022



# Efficiency, safety and reproducibility

- ▶ Objectives of this presentation and exercises
- ▶ Want to make you a better **R** user
- ▶ Will show you tools that
  - ▶ will make you more efficient
  - ▶ make your work safer
  - ▶ make your code more reproducible

# Functions

- ▶ Very important for efficiency, safety and reproducibility
- ▶ Once a function has been debugged and tested, you can use it with some confidence
- ▶ **R** has a built-in debugger which only works with functions
- ▶ Always a very good idea to have someone else check your code
  - ▶ functions make this much easier
- ▶ Functions work very well with Notebooks
- ▶ Functions are only way of putting code into packages

# Debugger

The screenshot shows the RStudio interface with the following components:

- Source Editor:** Displays the function `f2c` in `f2c.R`. The code is:

```
1 f2c <- function(temp_f) {  
2   temp_c <- (temp_f - 32) / 1.8  
3   return(temp_c)  
4 }
```

Line 3 is highlighted, indicating a breakpoint.
- Environment:** Shows the current environment with variables `temp_c` and `temp_f`, both with values of `-40`.
- Console:** Shows the execution of `f2c(-40)` and the resulting output. The console also displays the workspace loaded from `~/projects/R_training/CWRA_2022_R_workshop/.RData`.
- Debugger:** The `Next` button is highlighted, indicating the next step in the debugging process.

The console output shows the following sequence of events:

```
> f2c(-40)  
Called from: eval(expr, p)  
Browse[1]> n  
debug at ~/projects/R_training/CWRA_2022_R_workshop/tutorials/f2c.R#3: return(temp_c)  
Browse[2]>
```

# Function documentation

- ▶ It's critical that you document what your function does
- ▶ You won't be able to remember in the future
- ▶ No-one likes writing documentation, but it needs to be done
- ▶ If you have **devtools** installed then you can use `roxygen` to insert a skeleton for the documentation
  - ▶ once skeleton is created, you can edit the tags
  - ▶ **roxygen** tags are used by **R** to create package documentation
- ▶ Still need to add comments to describe what your code is doing

# Exercise

- ▶ Load the file “f2c.R” into your workspace
- ▶ Place your cursor anywhere inside the function
- ▶ Click on **Code | Insert Roxygen Skeleton**

```
## Title  
##  
## @param temp_f  
##  
## @return  
## @export  
##  
## @examples  
f2c <- function(temp_f) {  
  temp_c <- (temp_f - 32) / 1.8  
  return(temp_c)  
}
```

# Projects

- ▶ A project is a collection of **R** files
- ▶ Has its own directory
- ▶ Increases efficiency, safety and reproducibility
- ▶ Can have its own set of options

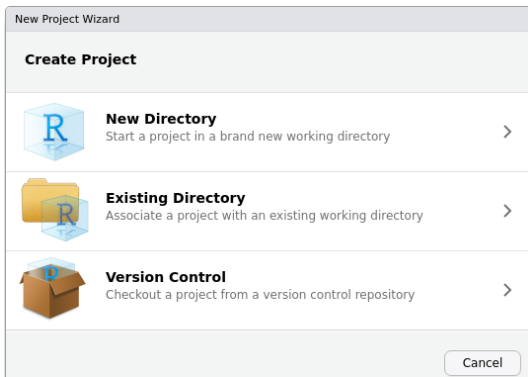
# Why create a project?

- ▶ Makes your code more reproducible
- ▶ Keeps code separate from other projects
- ▶ Lets your code work with `git` and GitHub or GitLab
  - ▶ a very good idea for code safety and reliability
- ▶ Basis for creating packages



# How to create a project

- ▶ Command is **File | New Project**
- ▶ Several alternatives appear



# Decisions, decisions. . . .

- ▶ New Directory
  - ▶ allows you to create any type of project, including packages
  - ▶ can use **git** (always a good idea), *but*
  - ▶ *won't* work with **GitHub**
- ▶ Existing Directory
  - ▶ only creates a simple project
  - ▶ doesn't set up **git**, but you can add it later
  - ▶ *won't* work with **GitHub**
- ▶ Version Control
  - ▶ clones a project from a repository like **GitHub** or **GitLab**
  - ▶ project has to be set up on the repository *first*

## .Rproj file

- ▶ Every project contains a project file (`project_name.Rproj`)
  - ▶ a text file which contains the project settings
- ▶ Double-clicking on the file in your file manager will load **RStudio** with the project
  - ▶ default directory will be set to the project directory
- ▶ Can also load a project manually in **RStudio** using  
**File | Open Project** or  
**File | Recent Projects**
- ▶ You can only have one project open at a time
  - ▶ opening a project will close your current project

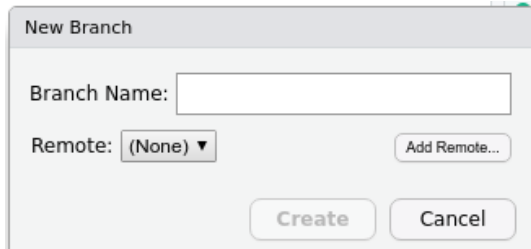
- ▶ **git** is a program for version control
- ▶ Created by Linus Torvalds (creator of Linux)
- ▶ Allows you to manage versions of your documents
- ▶ **RStudio** allows you to do most operations without typing commands
  - ▶ if you screw up, you *will* have to type **git** commands
- ▶ Can sync with **GitHub**

# Working with git

- ▶ **git** is based on *branches*
  - ▶ each branch is a separate set of files
- ▶ There is always a **main** (or **master**) branch
  - ▶ best version of the files
- ▶ When a branch is ready, it can be merged into the **main** branch
- ▶ You can switch between branches at any time

# git branches

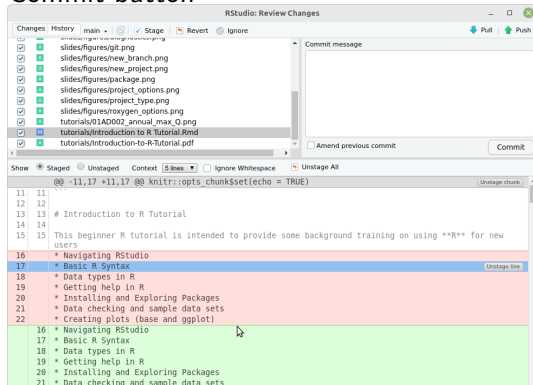
- ▶ *ALWAYS* create a new branch before working on a project
  - ▶ if you don't it will be a huge PITA
- ▶ Click on **New Branch** button in the **Git** tab



The image shows a 'New Branch' dialog box with a light gray background. At the top, the title 'New Branch' is displayed in a small, dark font. Below the title, there is a text input field labeled 'Branch Name:'. Underneath this, there is a 'Remote:' label followed by a dropdown menu currently showing '(None)' with a small downward arrow. To the right of the dropdown is a button labeled 'Add Remote...'. At the bottom of the dialog, there are two buttons: 'Create' and 'Cancel', both with rounded corners and a light gray background.

# Committing

- ▶ When you have finished some work, you can commit your changes by
- ▶ selecting the files to commit and
- ▶ clicking on **Commit** in the **Git** tab
- ▶ You will then see a window which lets you review your changes
- ▶ You **must** type a Commit message describing your changes before clicking on the Commit button

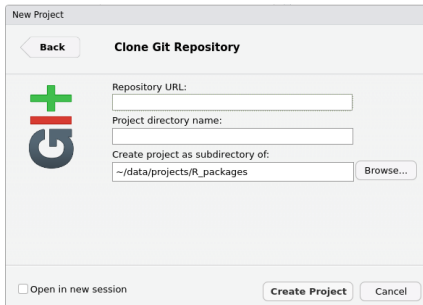
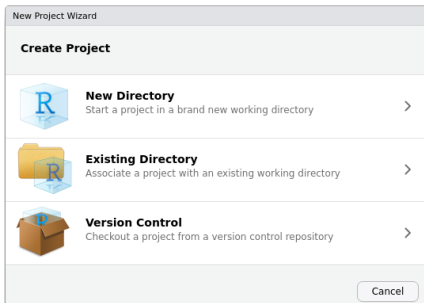


## Exercise

- ▶ Note - this can't be done on Rstudio Cloud
- ▶ Create a new project in a new directory
- ▶ Check "Create a git repository"
- ▶ Don't check "Use renv with this project"
  - ▶ **renv** is a package which keeps copies of all of the packages that you use with the project
- ▶ Quit **RStudio**
- ▶ Copy the file "f2c.R" to the project directory
- ▶ Copy the file "Introduction\_to\_R\_Tutorial.Rmd" to the project directory
- ▶ Go to your file manager and double-click on the ".Rproj" file in the new directory
  - ▶ you should now see "f2c.R" in the Files tab
- ▶ Create a new branch in the Git tab
  - ▶ load "f2c.R"
  - ▶ make an edit to the file "f2c.R"
  - ▶ commit the change
- ▶ In **RStudio** click on **File | Recent Projects** to re-load *this* project



- ▶ You can sync your project with an online repository at GitHub or GitLab
- ▶ Have to set up the online repo *first*
  - ▶ need an account (which you should have)
  - ▶ have to have **ssh** set up on your computer, and to tell GitHub your **ssh** key
- ▶ When you create a project, you select Version Control and then indicate the source to clone from



## Example exercise

- ▶ This can't be done on RStudio Cloud
- ▶ Create a local copy of the workshop project  
`git@github.com:CSHS-CWRA/CWRA_2022_R_workshop.git`
- ▶ If you are using the RStudio Cloud, you are already using the project

# R Packages

- ▶ **R** packages are a special type of project
- ▶ Only hold functions - do *not* use them for Notebooks
- ▶ Great for distributing your work to others
- ▶ Also useful for making your own work more reproducible
- ▶ *Must* contain documentation for all functions
- ▶ Can also contain test data sets

# Why create a package?

- ▶ The best way to distribute **R** code
- ▶ Makes your code reproducible
  - ▶ makes code reusable
- ▶ Improves code quality
- ▶ Takes care of dependencies
- ▶ Self-documenting
- ▶ Should work for anyone, on any computer

# Building a package

- ▶ All components are text files
  - ▶ You could build them manually
- ▶ **DON'T!**
- ▶ Use the package **devtools**
  - ▶ makes it *much* easier
- ▶ Need packages **roxygen2**, and **rmarkdown**
- ▶ Need LaTeX installed to create manuals
- ▶ Also, make sure to have **git** installed on your system

# Mandatory package components

- ▶ 2 Files
  - ▶ DESCRIPTION
  - ▶ NAMESPACE
- ▶ 2 directories are mandatory
  - ▶ /R – contains code .R files
  - ▶ /man – contains documentation .Rd files
- ▶ may have other directories

# DESCRIPTION

- ▶ Contains package description
- ▶ Has to have a specific format
- ▶ Has to indicate the packages required by your package
- ▶ You can see the DESCRIPTION file for any package on your system

# Exercise

- ▶ In the Packages tab, click on **CSHShydRology**
- ▶ Then select the **DESCRIPTION** file



**Canadian Hydrological Analyses** 



Documentation for package 'CSHShydRology'  
version 1.2.1

- [DESCRIPTION file](#).



# NAMESPACE

- ▶ Contains detailed information about imports and exports of each function
- ▶ Do NOT create or edit this file
  - ▶ **roxygen2** will automatically create and maintain it

# R directory

- ▶ Contains the **R** code
- ▶ Code must be written as functions
- ▶ Each function must be in a separate file
  - ▶ file name is same as function name
  - ▶ file extension must be **.R**

## man directory

- ▶ Contains the documentation files
- ▶ Creates the help system for the package
- ▶ Also creates the manual
- ▶ Each .R file has a .Rd file in **man**
  - ▶ Don't create these files manually

- ▶ Used by **devtools**, installed by it
- ▶ Automatically creates the .Rd files
  - ▶ uses comments at the beginning of each **.R** file

# Example

- ▶ All lines begin with #
- ▶ First line contains a 1 line description of the file
- ▶ Should not end with a period!

```
#' @description - optional multi-line description  
#' @param - required for each parameter  
#' @return - required, specifies return value of function  
#' @author - optional, lists author  
#' @export - required if function is accessible outside your package  
#' @examples - required, a working example
```

- ▶ Documentation can include formatting codes

```
\pkg{} - package  
\option{} - option  
\emph{} - emphasis  
\code{} - code
```

# Example

```
#' Calculates quantile values for a Quantile-Quantile plot  
#' @description The built-in \code{qqplot} function does not work with \pk  
#' @param x Required. A numeric vector.  
#' @param y Required. A numeric vector.  
#' @return Returns a dataframe with the quantiles of x and y.  
#' @author Kevin Shook  
#' @export  
#' @examples  
#' quantiles <- qqplotValues(runif(20), runif(50))
```



# Package function

- ▶ You should create a function that has the name of your package
- ▶ Example: CSHShydRology-package.R
- ▶ Gives overview of the package and what it's for
- ▶ Contains information to create NAMESPACE

# Other folders

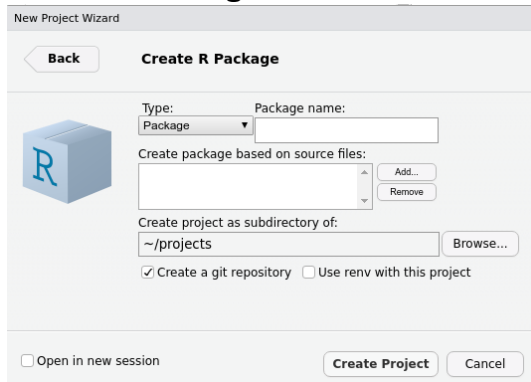
- ▶ You may see these folders in packages:
  - ▶ /data, data files used by the package
  - ▶ /vignettes, documentation written in Markdown
  - ▶ /inst, contains the file CITATION showing how to cite the package
  - ▶ /src, source code written in C, C++ or Fortran

# Workflow

1. Create the package
2. Add code
3. Build package
4. Check package
5. Create package file

# 1. Creating the package

- ▶ Create a new project in a new directory
  - ▶ **File|New Project**
  - ▶ select **R Package**
  - ▶ then give your package a name and a location
- ▶ Make sure to use **git**!



The screenshot shows the 'New Project Wizard' dialog box in RStudio. The title bar reads 'New Project Wizard'. Inside the dialog, there is a 'Back' button on the left and the title 'Create R Package' in the center. On the left side, there is a blue 3D cube icon with a white 'R' on it. The main area contains the following fields and controls:

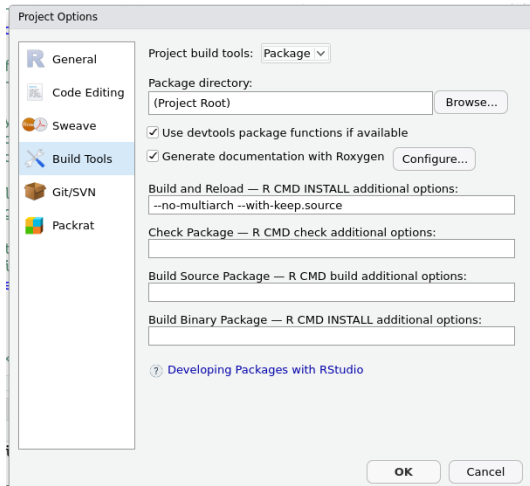
- Type:** A dropdown menu with 'Package' selected.
- Package name:** An empty text input field.
- Create package based on source files:** A list box with an upward arrow on the left and two buttons, 'Add...' and 'Remove', on the right.
- Create project as subdirectory of:** A text input field containing '~/projects' and a 'Browse...' button to its right.
- Options:** Two checkboxes at the bottom: ☒ 'Create a git repository' and ☐ 'Use renv with this project'.
- Footer:** At the bottom left, there is a checkbox ☐ 'Open in new session'. At the bottom right, there are two buttons: 'Create Project' and 'Cancel'.

# New package

- ▶ **Rstudio** will create all of the files and directories
  - ▶ /R
  - ▶ /man
  - ▶ DESCRIPTION
  - ▶ NAMESPACE
- ▶ Also creates a sample file **hello.R** in /R
- ▶ Adds folders and files for **git**

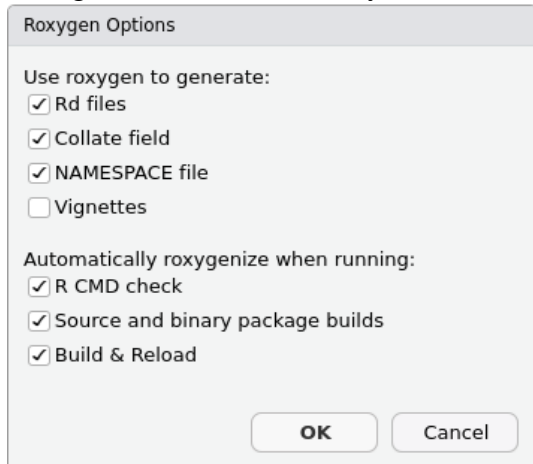
# Setting up roxygen

- ▶ Not enabled by default
- ▶ Set it up using **Tools|Project Options**



# Set roxygen options

## ► Configure to run automatically



The image shows a 'Roxygen Options' dialog box with a title bar. It contains two sections of options. The first section, 'Use roxygen to generate:', has four checkboxes: 'Rd files' (checked), 'Collate field' (checked), 'NAMESPACE file' (checked), and 'Vignettes' (unchecked). The second section, 'Automatically roxygenize when running:', has three checkboxes: 'R CMD check' (checked), 'Source and binary package builds' (checked), and 'Build & Reload' (checked). At the bottom right are 'OK' and 'Cancel' buttons.

Roxygen Options

Use roxygen to generate:

- ☒ Rd files
- ☒ Collate field
- ☒ NAMESPACE file
- ☐ Vignettes

Automatically roxygenize when running:

- ☒ R CMD check
- ☒ Source and binary package builds
- ☒ Build & Reload

OK Cancel

## 2. Adding R code

- ▶ Put your R code files in /R
- ▶ Must be functions
  - ▶ one function per file
- ▶ Add the roxygen skeleton to your code for each file
  - ▶ **Code|Insert Roxygen Skeleton**
- ▶ Fill in skeleton



# Converting your R code

- ▶ You will need to make some changes to your code
- ▶ Don't use the **library()** function to load packages
  - ▶ package importation handled by `NAMESPACE`
- ▶ Specify the name of the package in every function (outside of your package and Base R) call
  - ▶ syntax is **package::function**

### 3. Building package

- ▶ Use command **Build | Clean and Rebuild**
- ▶ Expect to get error messages!
- ▶ Fix until package builds
- ▶ If the package builds, it will be added to your list of packages

## 4. Checking the package

- ▶ Just because a package can build, doesn't mean that it is good!
- ▶ Use command **Build|Check**
  - ▶ does a detailed check of entire package
  - ▶ can be slow for large packages
  - ▶ tries to run your examples
  - ▶ *very* picky
  - ▶ you will probably get many, many errors, warnings and notes at first
  - ▶ eliminating all warnings and notes really improves your code

## 5. Creating the package file

- ▶ 2 options:
  - ▶ **Build | Build Source Package** - contains source code (all languages)
  - ▶ **Build | Build Binary Package** - contains compiled Fortran, C, C++ code
- ▶ Reason is that Windows computers usually don't have compilers
- ▶ If just using **R** code, make it a source package
- ▶ If you are using Fortran, C, C++, create both types

# Building the manual .pdf

- ▶ When the package is built, should also create the .pdf
- ▶ Must have LaTeX installed
- ▶ For some reason, this doesn't work for me
  - ▶ have to do it manually
  - ▶ type in this command in the **R** console:

```
system("R CMD Rd2pdf mypackage")
```

# Unit tests

- ▶ New feature, part of **devtools**
  - ▶ tests the results of functions
  - ▶ compares function outputs to known values
  - ▶ allows automated testing of functions

# Exercise

- ▶ Create a package from scratch
- ▶ Build the package
- ▶ Copy some functions into the /R directory
- ▶ Add the **roxygen** skeleton to the functions
- ▶ See if you can get the package to build properly

# Summary

- ▶ Learning to code in functions will take some time but is worth the investment
  - ▶ improves your code quality
  - ▶ makes code more reliable
  - ▶ makes *you* more efficient
- ▶ Creating an **R** project should be your first step when starting a new task
- ▶ Creating your own **R** packages, when using **git**, is the ultimate way to ensure efficiency, safety and reproducibility