R Packages

* R packages bundle together code, data, documentation, and tests, and are designed to be easy to share with others.
* They are the main way R users distribute and reuse functionality.

There are already well over 20,000 packages available on CRAN, covering everything from visualization to machine learning, bioinformatics, and more. As an R user, you will quickly become familiar with installing and using packages as part of your workflow.

## Why build R packages?

* To share code with others (or even just with yourself).
* To leverage existing tooling for consistency & efficiency.
* To gain deeper understanding of how R works, benefiting both you and the community.

### Sharing code

* Bundling code into a package makes it easy for others to use.
* Most R users are already familiar with installing and loading packages, so sharing functionality this way reduces friction.
* Even if you only want to reuse code across multiple personal projects, packaging is often the best solution.

### Leveraging tooling

R packages follow standardized conventions. This unlocks a rich ecosystem of tools:

* Automated documentation generation (roxygen2)
* Unit testing (testthat)
* Package building (devtools, usethis)

### Package learnings benefit everyone

Learning how packages work will make you a better R programmer, since you’ll better understand how existing packages are structured. It also allows you to contribute to open-source R packages by fixing bugs, writing documentation, or adding features.

## What do you need?

* The [RStudio IDE](https://posit.co/products/open-source/rstudio/#Desktop) is highly recommended because of its integration with devtools,` but it’s not strictly required.
* You’ll need the following R packages: [devtools](https://github.com/hadley/devtools), usethis, [roxygen2](https://github.com/klutometis/roxygen), [testthat](https://github.com/hadley/testthat), [knitr](https://github.com/yihui/knitr)
* For most pure R packages you don’t need a separate development environment, but compiled code may require one. Often devtools handles setup automatically, and you can confirm readiness using has\_devel().

Check if your system is ready:

library(devtools)  
has\_devel() # Your system is ready to build packages!

# Basic Setup Process

## 1. Create a skeleton project (package directory)

Using usethis, create a new package directory called datasci:

usethis::create\_package("datasci")

This will create a new directory datasci/ with important files:

* DESCRIPTION — metadata about your package
* NAMESPACE — functions to export/import
* R/ — where your R code goes
* .Rbuildignore — files to exclude from build

You can also do this in RStudio: > File -> New Project -> New Directory -> R package

The function usethis::create\_package(“datasci”) creates a new folder called datasci/ in your current working directory.

If you want the package to be created elsewhere - either change working directory with setwd() or provide a relative/full path directly, e.g:

usethis::create\_package("~/Development/datasci")

Once the project is open in RStudio, you can build and reload the package.

The simplest way is through the Rstudio menu: Build -> Clean & Install (Ctrl + Shift + B)

This builds and installs the package, restarts your R session, and reloads it. At this stage, the package is still empty, so this step mainly ensures everything is set up correctly.

An alternative is: Build -> Load All (Ctrl + Shift + L)

or programmatically:

devtools::load\_all()

Key difference:

* Clean & Install fully builds, installs, and reloads your package — thorough but slower.
* Load All simulates installation and loads your package into memory — faster for iterative development.

Both approaches are useful, and you’ll likely use each at different points in the development workflow.

## 2. Add function(s)

All package functions live in the R/ folder. Create a file add.R:

add <- function(x, y) {  
 x + y  
}

Reload the package & test function:

devtools::load\_all()  
add(2, 3) # now able to call the function

So now we’ve got a basic package which contains a function for addition. But it’s missing documentation:

?add

## 3. Documentation

Packages need help files so others (and your future self) can understand how functions work.

When you build a package, documentation is stored in the man/ directory as .Rd files (R documentation files). These files look similar to LaTeX and are what ?function\_name displays when you request help in R.

For example, the desc.Rd file in the dplyr package looks like this:

% Generated by roxygen2: do not edit by hand  
% Please edit documentation in R/desc.R  
\name{desc}  
\alias{desc}  
\title{Descending order}  
\usage{  
desc(x)  
}  
\arguments{  
\item{x}{vector to transform}  
}  
\description{  
Transform a vector into a format that will be sorted in descending order.  
This is useful within \code{\link[=arrange]{arrange()}}.  
}  
\examples{  
desc(1:10)  
desc(factor(letters))  
  
first\_day <- seq(as.Date("1910/1/1"), as.Date("1920/1/1"), "years")  
desc(first\_day)  
  
starwars \%>\% arrange(desc(mass))  
}

While .Rd files are readable, maintaining them separately from the code is cumbersome and error-prone. For example, if you change a function’s arguments but forget to update its .Rd file, the help page becomes misleading. This extra overhead is why most developers rely on roxygen2 to generate .Rd files automatically.

roxygen2 lets you attach documentation directly above the function using special comments (#'). These comments are then used to generate .Rd files automatically during the package build process.

For example, the desc() function in dplyr is documented using roxygen comments:

#' Descending order  
#'  
#' Transform a vector into a format that will be sorted in descending order.  
#' This is useful within [arrange()].  
#'  
#' @param x vector to transform  
#' @export  
#' @examples  
#' desc(1:10)  
#' desc(factor(letters))  
#'  
#' first\_day <- seq(as.Date("1910/1/1"), as.Date("1920/1/1"), "years")  
#' desc(first\_day)  
#'  
#' starwars %>% arrange(desc(mass))  
  
desc <- function(x) {  
 obj\_check\_vector(x)  
 -xtfrm(x)  
}

* Roxygen comments start with #', so regular comments can coexist.
* Special tags like @param, @return, @examples, and @export define specific parts of the documentation.
* The first line is used as the title, the following paragraph as the description, and @details is optional.

You can enable Markdown formatting in roxygen2 by adding the line below to your DESCRIPTION file: Roxygen: list(markdown = TRUE)

### Initializing roxygen2

For the datasci package, generate documentation and update the namespace:

roxygen2::roxygenise()

This will create or update:

* .Rd files in the man/ directory
* The NAMESPACE file (managing exported functions)
* The RoxygenNote field (specifies which version of Roxygen the package is using) in DESCRIPTION

If you inspect the NAMESPACE file you’ll see that the first line is:

# Generated by roxygen2: do not edit by hand

Roxygen handles two main tasks based on two sets of tags:

1. Generate .Rd files from tags like @param and @examples.
2. Manage NAMESPACE exports using tags like @export.

It’s recommended to use roxygen2 for both purposes, rather than editing NAMESPACE manually. For more on namespaces, read the [section on namespaces](https://r-pkgs.org/dependencies-mindset-background.html#sec-dependencies-namespace) in Hadley’s book.

### Adding roxygen2 to your own function

Next let’s add Roxygen comments to add().

#### Open the function file

* In your datasci package project, go to the R/ folder.
* Open the file add.R in RStudio (or create it if it doesn’t exist).
* Make sure your cursor is inside the body of the function or somewhere above it.

#### Insert a Roxygen skeleton

RStudio can generate a template for Roxygen comments, so you don’t have to type all the tags manually:

Code -> Insert Roxygen Skeleton (Ctrl + Alt + Shift + R)

This will automatically insert a skeleton comment like this above your function:

#' Title  
#'  
#' Description  
#'  
#' @param x   
#' @param y   
#' @return   
#' @export  
#' @examples  
#'

1. Update the skeleton with meaningful content

Replace the placeholders with actual information for your function. For add(), it could look like this:

#' Add numbers together  
#'  
#' This is a description of the function. It works like [base::sum()].  
#' @param x First number  
#' @param y Second number  
#'  
#' @return Sum of the first and second number  
#' @export  
#'  
#' @examples  
#' add(1, 1)  
#' add(20, 22)  
add <- function(x, y) {  
 x + y  
}

#### 4. Generate the documentation

Once the comments are in place, run:

devtools::document()

* This generates an .Rd file in man/ for the add() function.
* It also updates NAMESPACE to export the function.

Alternatively you can also use Ctrl + Shift + D.

You can now view the help for the add() function with:

?add

Tip: You can always regenerate documentation after edits by running devtools::document() again. For quick iteration, Ctrl/Cmd + Shift + D is a shortcut in RStudio that calls roxygen2::roxygenise() under the hood.

Obviously this is just a very simple example of function documentation to illustrate the process. I recommend you read the other [vignettes for roxygen2](https://cran.r-project.org/web/packages/roxygen2/vignettes/) in order to learn more.

Note:

* Functions without @export remain internal and are not directly accessible when the package is loaded (function will remain unexported).
* It often makes sense for certain utility functions within packages to remain unexported since though they may be used/shared by various functions within a package, these functions themselves often aren’t directly useable (or useful) outside of the package environment.

Internal functions can still be documented and accessed with:

pkgname:::unexported\_function()

All exported objects, including datasets, should also be documented, and the package itself should have overall documentation. Hadley’s book has a [section on object documentation](http://r-pkgs.had.co.nz/man.html) which is a good reference for this.

Alternative documentation workflow:

* devtools::load\_all() is fast but shows limited preview for help pages and links.
* Install and Restart (Ctrl/Cmd + Shift + B) is slower but thorough: it installs the package, regenerates all documentation, and restarts R to reflect changes fully.

To ensure documentation is always rebuilt correctly, configure Roxygen options within the Build Tools section of RStudio Project Options:

## Testing

A good R package includes tests to ensure that its code behaves as expected. The testthat package provides a simple, structured framework for writing and running tests.

### Setting up tests

You can initialize testthat infrastructure and create a test file for a function with:

usethis::use\_testthat()  
usethis::use\_test("add")

usethis::use\_testthat() creates the necessary testing infrastructure:

* tests/testthat.R
* tests/testthat/ folder
* Adds testthat to the suggested packages

usethis::use\_test("add") creates tests/testthat/test-add.R containing an example test:

test\_that("multiplication works", {  
 expect\_equal(2 \* 2, 4)  
})

### Structure of tests

Testing in testthat follows a clear, organized hierarchy:

* Test files: Each file should begin with test- and typically groups tests for a specific function or feature (e.g. test-add.R contains tests for the add() function). This helps keep your tests organized and easy to maintain.
* Test blocks: Within each test file, tests are written in blocks using the test\_that() function. Each block represents a single logical test scenario, describing what is being tested in plain language (the first argument of test\_that() is a descriptive string).
* Expectations: Inside a test\_that() block, you write expectation functions such as expect\_equal(), expect\_error(), or expect\_true(). These functions check whether your code behaves as expected.
* Outcome: If an expectation passes, the test silently succeeds. If it fails, testthat throws an error and reports which expectation failed, making it easy to identify problems in your code.

### Example tests for add()

Replace the content of test-add.R with tests for the add() function:

# check numeric addition is correct  
  
test\_that("1 + 1 equals 2", {  
 expect\_equal(add(1, 1), 2)  
})  
  
test\_that("1 + -1 equals 0", {  
 expect\_equal(add(1, -1), 0)  
})  
  
# check non-numeric input results in an error")  
  
test\_that("string input results in an error", {  
 expect\_error(add("1", 1))  
})

Run all tests in your package with:

devtools::test()

* Output will appear in the RStudio Build Pane, showing which tests passed or failed.
* Regular testing helps catch errors early and ensures your package functions remain correct as you add features or refactor code.

Tip: You can also run individual test files with test\_file(“tests/testthat/test-add.R”) if you want to focus on a specific function.

## Sharing & Collaboration

Always use Git (or another version control system), both for personal workflow management and for eventual collaboration.

When you are ready to share your package, consider the following key steps:

### 1. Update package information in the DESCRIPTION

The DESCRIPTION file contains essential metadata/info about your package:

* Title: a concise, descriptive name.
* Description: explains what the package does and why it’s useful.
* Authors and Maintainers: list contributors with their roles and contact information.

Keeping metadata accurate improves usability and helps others understand and trust your package.

### 2. Add a software license

Licenses specify how others can use, modify, and distribute your package. Common choices for R packages:

* MIT License: permissive, allows reuse with minimal restrictions.
* GPL3 License: ensures that derivative works remain open source.

usethis provides helpers to add a license:

usethis::use\_mit\_license("Your Name")  
usethis::use\_gpl3\_license("Your Name")

This will create a LICENSE file in your package.

[tldrlegal.com](https://tldrlegal.com/) and [choosealicense.com/licenses/](https://choosealicense.com/licenses/) are two good places to get more information on different software licenses, and the R project website also has [a page](https://www.r-project.org/Licenses/) which lists the licenses in use for R packages.

### 3. Add a package README

A README is the first point of contact for anyone using your package, especially on GitHub. It should cover:

* Why the package exists
* How to install it (e.g. remotes::install\_github('username/packagename'))
* Basic usage examples

For easier formatting, write the README in R Markdown and knit to Markdown:

usethis::use\_readme\_rmd()

his creates a template README.Rmd that you can edit and knit to README.md.

### 4. Add a package vignette

Vignettes are long-form, example-driven guides that show how to use your package in practice. They are excellent for teaching users advanced workflows or illustrating best practices.

Create a template vignette with usethis:

usethis::use\_vignette("my-vignette")

Include explanations, code examples, and visualizations to make your package approachable and user-friendly.

### 5. Follow best practices

The goodpractice package provides automated checks and suggestions for improving your package:

goodpractice::gp("my-package")

It will analyze code structure, documentation, testing coverage, and adherence to conventions, helping you maintain a professional, high-quality package.

## Other Topics (not covered)

* Namespaces, scoping etc - NAMESPACE
* Compiled code (e.g. Rcpp) - src/
* Installed files - inst/
* Data - data/
* Versioning, dependencies etc - DESCRIPTION
* CRAN - preparing your package for submission (e.g. NEWS.md)

All of the above topics are covered very nicely in [R packages](https://r-pkgs.org/).

## Review

### R code workflow

1. Edit your .R files in the R/ directory.
2. Load your package for testing using: devtools::load\_all() (or Ctrl + Shift + L in RStudio).
3. Explore your functions interactively in the R console.
4. Repeat as needed while refining your code.

### Documentation workflow

1. Add roxygen comments to your .R files.
2. Convert comments to .Rd files and update the NAMESPACE: devtools::document() (or Ctrl + Shift + D in RStudio)
3. Preview documentation with: ?function\_name
4. Repeat until the documentation is complete and clear.

### Testing workflow

1. Modify your code in R/ or tests in tests/testthat/.
2. Run tests with: devtools::test() (or Ctrl + Shift + T in RStudio)
3. Repeat until all tests pass.

## Learn More

The resources below are ordered (roughly) from shortest/simplest to longest/most detailed.

* [Writing an R package from Scratch (Hilary Parker)](https://hilaryparker.com/2014/04/29/writing-an-r-package-from-scratch/)
* [R package primer - a minimal tutorial (Karl Broman)](https://kbroman.org/pkg_primer/)
* [Write your own R package (Stat 545)](https://stat545.com/packages00_index.html)
* [R Packages (Hadley Wickham)](https://r-pkgs.org/)
* [Writing R extensions (CRAN)](https://cran.r-project.org/doc/manuals/r-release/R-exts.html)

Another excellent way to learn is to explore existing packages on GitHub. Mature, actively maintained packages such as dplyr, tidyr, and stringr are great examples of well-structured code, documentation, testing, and workflows.

### About devtools and usethis

While devtools remains widely used, its functionality is being modularized. A key part of this is usethis, which automates many common package and project setup tasks, improving workflow efficiency. Some previously available devtools helpers have been deprecated and removed as of version 2.1.

Most usethis functions require minimal setup, but some allow further configuration. Refer to the usethis [setup article](https://usethis.r-lib.org/articles/usethis-setup.html) for more info.

# Building an R Package Tutorial

## Goal

* Create, document, test, build and release an R package
* Use {usethis}, {devtools}, {roxygen2}, {testthat}
* Keep slides minimal: commands first, commentary verbal

## Prerequisites

* Install core tooling
* Reload or attach packages
* Verify setup

library("usethis")  
library("devtools")  
library("roxygen2")  
library("testthat")  
library("pkgdown")  
library("lintr")  
library("goodpractice")

## Create Package Skeleton

usethis::create\_package("~/Development/datasci")  
# Inside the new project:  
usethis::use\_git()  
# Or create within current project:  
# usethis::create\_package(".")

* Creates standard package structure (R/, man/, DESCRIPTION, NAMESPACE)
* Initializes version control (Git)
* Opens the new project in RStudio

## Fill DESCRIPTION

The DESCRIPTION file is your package’s metadata—it tells users and R what your package does and who maintains it.

usethis::use\_mit\_license("Ian Durbach")  
usethis::use\_description(fields = list(  
 Title = "Data Science Essentials",  
 Description = "What the package does (one paragraph).",  
 `Authors@R` = 'person("Ian","Durbach", role = c("aut","cre"), email = "ian.durbach@uct.ac.za")',  
 Depends = "R (>= 4.1.0)"))

Key fields: Title (short), Description (detailed), Authors, License, and version requirements.

## Add a Function

* Create an R script in R/
* Write a small function
* Keep minimal args and examples

usethis::use\_r("add")  
# In R/add.R  
add <- function(x, y) {  
 x + y  
}

## Document with roxygen2

Documentation is critical for usability. Use roxygen2 comments to generate help files automatically.

1. Run roxygen2::roxygenise() to initialize
2. Insert skeleton: **Code Insert Roxygen Skeleton** (Ctrl/Cmd + Alt + Shift + R)
3. Edit roxygen comments
4. Run devtools::document() to generate NAMESPACE and man/add.Rd

#' Add numbers together  
#'  
#' This is a description of the function. It works like [base::sum()].  
#' @param x First number  
#' @param y Second number  
#'  
#' @return Sum of the first and second number  
#' @export  
#'  
#' @examples  
#' add(1, 1)  
#' add(20, 22)  
add <- function(x, y) {  
 x + y  
}

## Manage Dependencies

Properly declaring dependencies ensures your package works across different environments.

* Declare Imports/Suggests
  + Imports: Required for your package to function
  + Suggests: Optional packages (for vignettes, tests, or enhanced features)
* Use helper functions (e.g. pipe)
* Best practice: Use explicit :: notation (e.g. dplyr::mutate()) to avoid namespace conflicts.

usethis::use\_package("dplyr", type = "Imports")  
usethis::use\_pipe() # adds magrittr pipe  
# In code, prefer explicit calls:  
# dplyr::mutate(...)

## Testing with testthat

Automated testing catches bugs early and ensures code reliability.

* Set up testing infrastructure
* Create test files
  + Test expected behavior (happy path)
  + Test edge cases and error conditions
* Run tests locally with devtools::test()

usethis::use\_testthat()  
usethis::use\_test("add")

# In tests/testthat/test-add.R  
test\_that("add works", {  
 expect\_equal(add(1, 1), 2)  
 expect\_equal(add(20, 22), 42)  
 expect\_error(add("1", 1))  
})

## README & Vignette

Make your package accessible to users with clear documentation.

* Create README.Rmd (badges optional) - your package’s front page - include installation instructions, basic usage, and examples.
* Add a vignette for longer examples - long-form documentation for complex workflows or tutorials.
* Knit/build documents

usethis::use\_readme\_rmd()  
usethis::use\_lifecycle\_badge("stable")  
usethis::use\_vignette("getting-started")  
devtools::build\_vignettes()

## Code Quality Checks (Optional but Recommended)

Maintain high code quality with automated linting and best practice checks.

lintr::lint\_package()  
goodpractice::gp("datasci")

What they check:

* lintr: Code style, syntax issues, potential bugs
* goodpractice: Package structure, documentation completeness, testing coverage, DESCRIPTION file quality

Address warnings and suggestions to improve package quality.

## Build & Install

Test your package locally before sharing it with others.

devtools::build()  
devtools::install()  
library(datasci)  
add(20, 22)

What happens:

* build() creates a source tarball (.tar.gz)
* install() installs the package locally
* Load and test your functions interactively

## R CMD check

The gold standard for package validation—this checks everything.

devtools::check()  
# For CRAN-like checks:  
devtools::check(environment\_variables = c("\_R\_CHECK\_CRAN\_INCOMING\_"="TRUE"))

* Run full package checks
* Fix NOTEs/WARNINGs/ERRORs
* Repeat until all checks pass cleanly.
* Goal: Zero ERRORs, zero WARNINGs, zero NOTEs.
* Common issues: Missing documentation, namespace problems, example errors, or encoding issues.

## Versioning & NEWS

Track your package’s evolution with semantic versioning and a NEWS file.

usethis::use\_news\_md()  
usethis::use\_version("minor") # or "patch"/"major"

Semantic versioning (x.y.z):

* Major (x): Breaking changes
* Minor (y): New features, backward-compatible
* Patch (z): Bug fixes

Document all changes in NEWS.md for transparency.

## GitHub & Continuous Integration (CI)

Automate testing and deployment with GitHub Actions.

* Create GitHub repo and push
* Enable standard R-CMD-check workflow
* Optionally add pkgdown deployment

usethis::use\_github() # requires gh PAT  
usethis::use\_github\_action\_check\_standard()  
usethis::use\_pkgdown()  
pkgdown::build\_site()

Benefits:

* Automatic R CMD check on multiple platforms
* Professional package website with pkgdown
* Collaborative development with version control
* Visible build status badges

## Release Options

Share your package with the R community.

Distribution channels:

* GitHub: Easiest, install with remotes::install\_github()
* CRAN: Widest reach, but requires passing strict policies - via devtools

usethis::use\_github\_release()  
  
# CRAN (read policies first!):  
devtools::submit\_cran()

## Summary

Core workflow:

1. **Edit** code and documentation
2. **Document** with devtools::document()
3. **Test** with devtools::test()
4. **Check** with devtools::check()
5. **Build** and install locally

**Repeat this cycle** frequently to catch issues early and maintain code quality.

Use devtools::load\_all() (Ctrl/Cmd + Shift + L) during development to quickly test changes without reinstalling.