Introduction to Building packages in R

Nicolas Attalides

Who am I?

- **Name:** Nicolas Attalides
- **Coding in R since:** 2005 (yes that's before RStudio!)
- **Profession:** Senior Data Scientist and trainer (6+ yrs.)
- **Education:** PhD in Statistical Science from UCL (2015)
- **R Status:** A never-ending evolving R dinosaur
- **Hobbies:** Tennis and coding (not at the same time)

Workshop Setup:

Wi-Fi

- Network Name: N/A
- Password: N/A

Resources

- R (version 3.6.3)
- RStudio (version 1.4.1106)

Packages

- {tidyverse} (version 1.3.0)
- {devtools} (version 2.3.2)
- {usethis} (version 2.0.1)

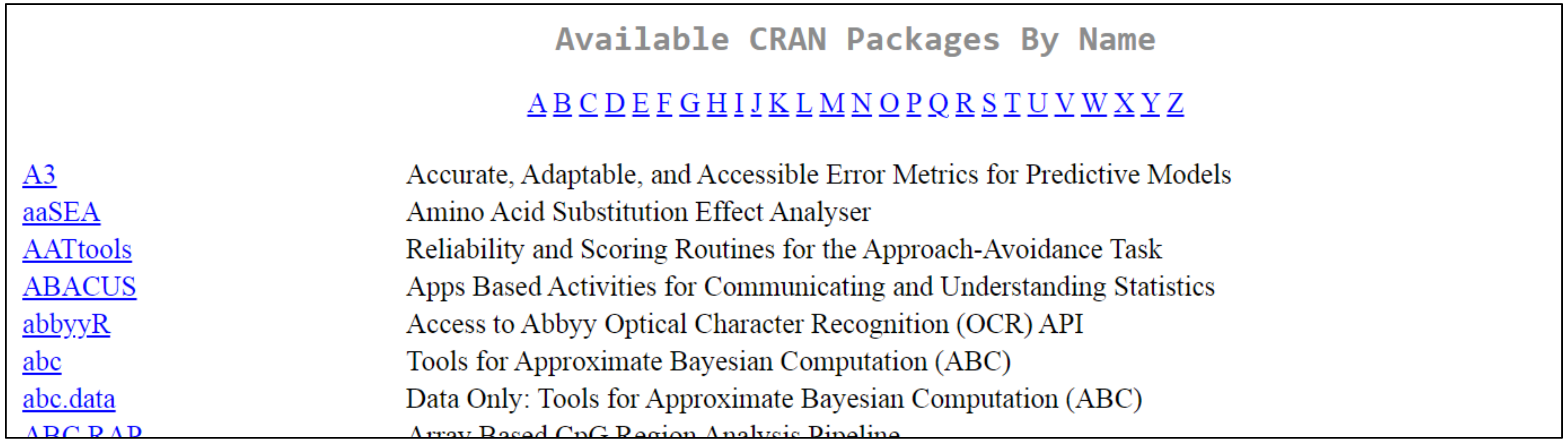
What is an R package?

An R package is like a **collection** of **code**, **data** and **documentation** that follow some standard rules and formats.

This is the best way for an R user to **share** their **work** and enable others to use the functionality that is developed.

Comprehensive R Archive Network (CRAN)

The central repository of R packages is called the Comprehensive R Archive Network (**CRAN**). This contains an archive of R distributions and has more than 17,000 packages ready to be installed and used.



Available CRAN Packages By Name

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A3	Accurate, Adaptable, and Accessible Error Metrics for Predictive Models
aaSEA	Amino Acid Substitution Effect Analyser
AATtools	Reliability and Scoring Routines for the Approach-Avoidance Task
ABACUS	Apps Based Activities for Communicating and Understanding Statistics
abbyyR	Access to Abbyy Optical Character Recognition (OCR) API
abc	Tools for Approximate Bayesian Computation (ABC)
abc.data	Data Only: Tools for Approximate Bayesian Computation (ABC)
ABC.RAP	Array Based CpG Region Analysis Pipeline

Find out more about CRAN here: https://cran.r-project.org/

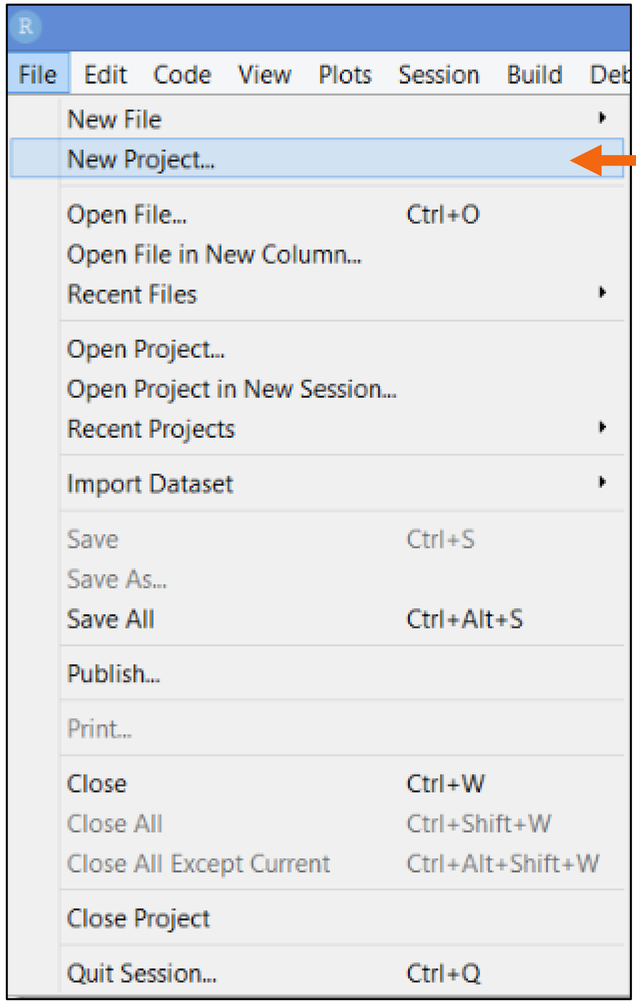
Topics

- Workshop aim:

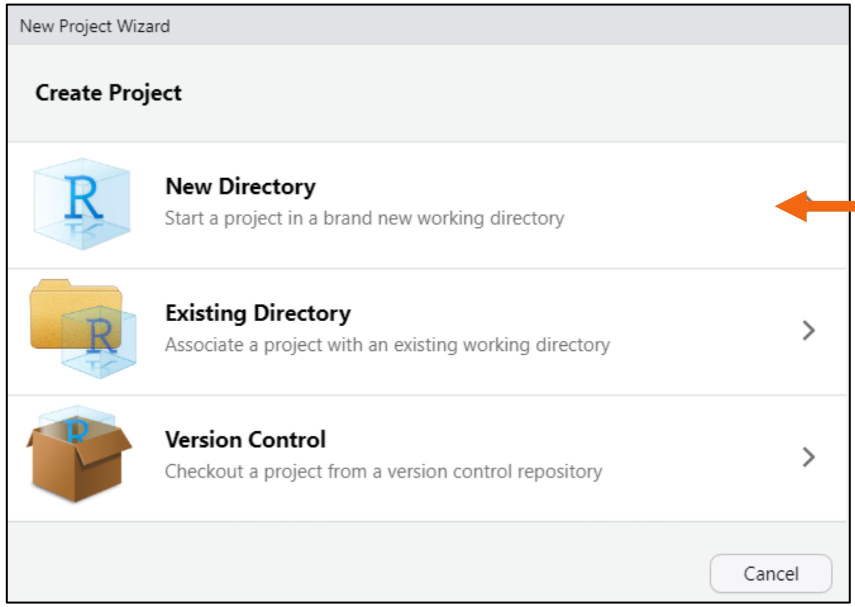
Learn how to build an R package in an easy step-by-step approach.

- Topics:
  - Learn how to create an R package within RStudio
  - Understand the package structure and its various components
  - Learn how to write, document and test functions in R for a package
  - Learn how to check, build and install an R package

Create an R package

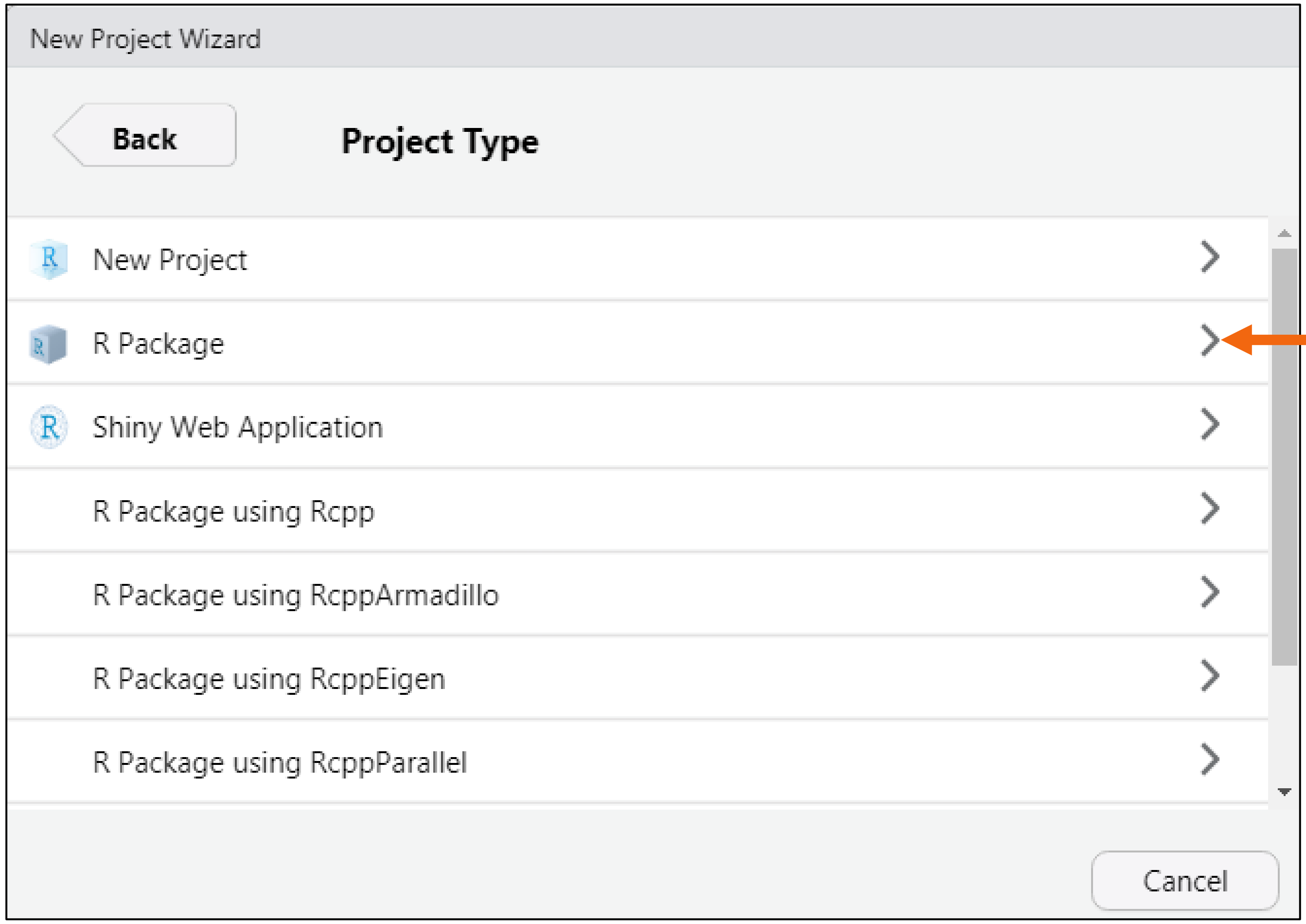


Open the File options and select "New Project..."


New Project Wizard
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

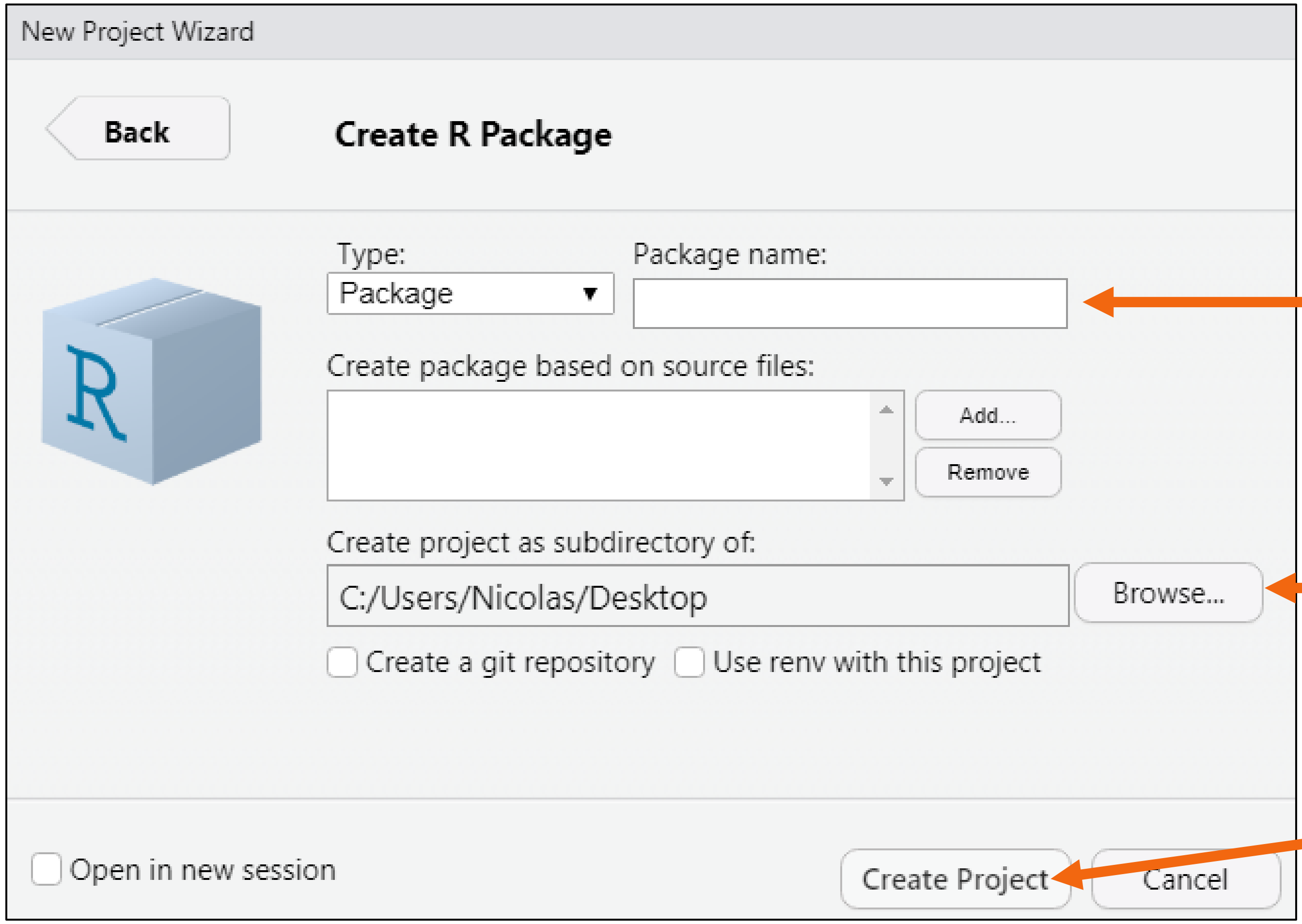
Then select "New Directory

Create an R package



Select “R Package”

Create an R package



New Project Wizard

Back

Create R Package

Type: Package

Package name:

Create package based on source files:

Add...

Remove

Create project as subdirectory of:

C:/Users/Nicolas/Desktop

Browse...

Create a git repository

Use renv with this project

Open in new session

Create Project

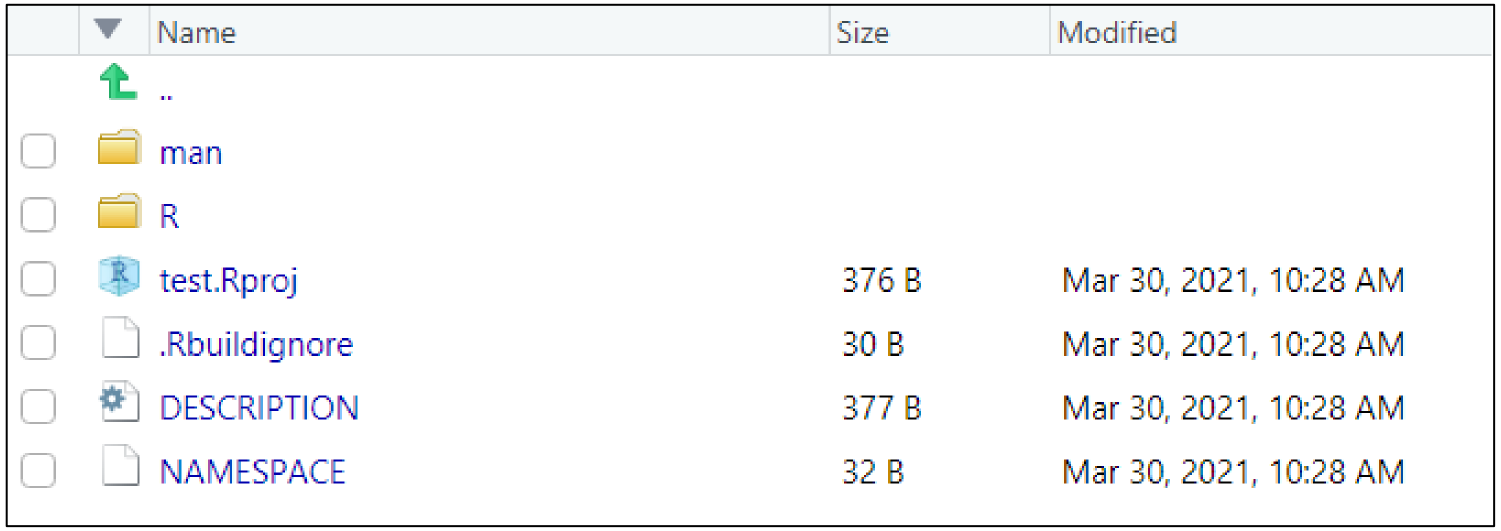
Cancel

Type in package name

Select the location of the package

Select “Create Project”

Congratulations! You created an R package*!



	Name	Size	Modified
	..		
	man		
	R		
	test.Rproj	376 B	Mar 30, 2021, 10:28 AM
	.Rbuildignore	30 B	Mar 30, 2021, 10:28 AM
	DESCRIPTION	377 B	Mar 30, 2021, 10:28 AM
	NAMESPACE	32 B	Mar 30, 2021, 10:28 AM

*An empty package with the basic structure…

Create an R package

An alternative way to create an R package with the same result is to use the **`create_package()`** function from the **{usethis}** package.

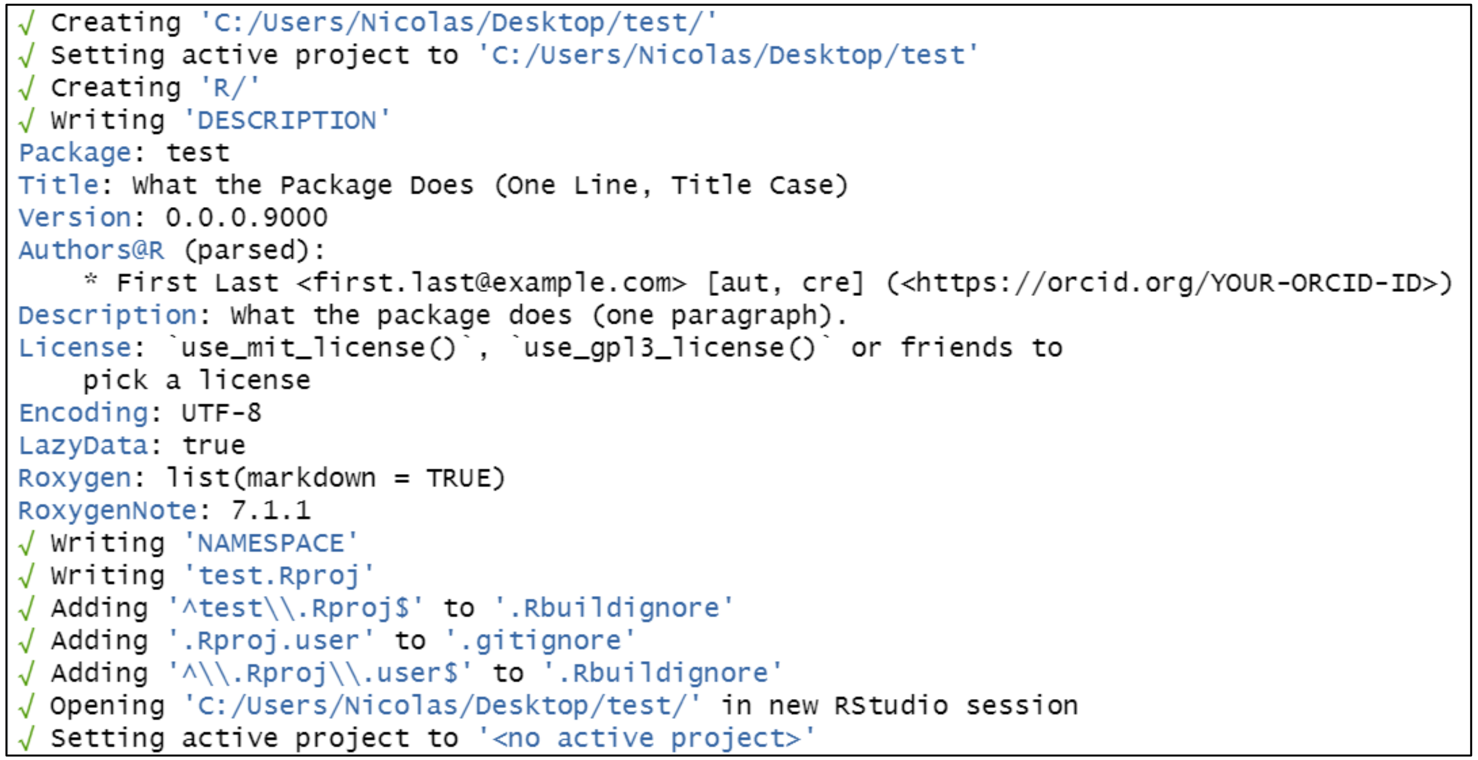
```r
library(devtools)

create_package("C:/Users/Nicolas/Desktop/myRpackage")
```

Package path

Package name

This opens a new RStudio session with the new project loaded

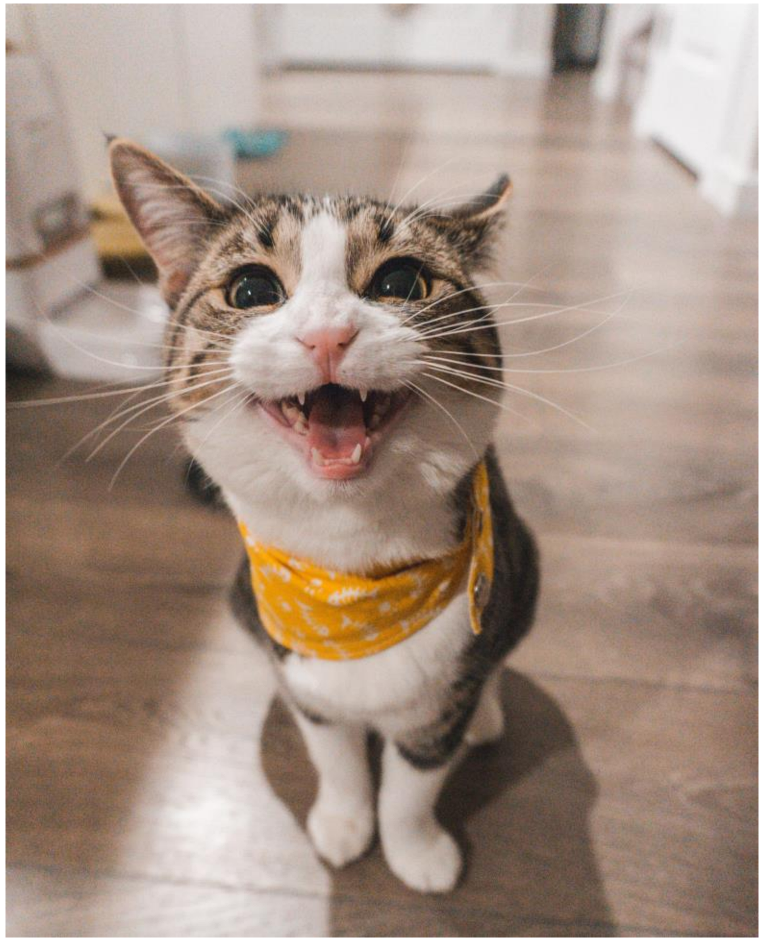
Congratulations! You created an R package*!



```
√ Creating 'C:/Users/Nicolas/Desktop/test/'
√ Setting active project to 'C:/Users/Nicolas/Desktop/test'
√ Creating 'R/'
√ Writing 'DESCRIPTION'
Package: test
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (<https://orcid.org/YOUR-ORCID-ID>)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
√ Writing 'NAMESPACE'
√ Writing 'test.Rproj'
√ Adding '^test\\.Rproj$' to '.Rbuildignore'
√ Adding '.Rproj.user' to '.gitignore'
√ Adding '^\\.Rproj\\.user$' to '.Rbuildignore'
√ Opening 'C:/Users/Nicolas/Desktop/test/' in new RStudio session
√ Setting active project to '<no active project>'
```

*An empty package with the basic structure...

Live Coding Example 1

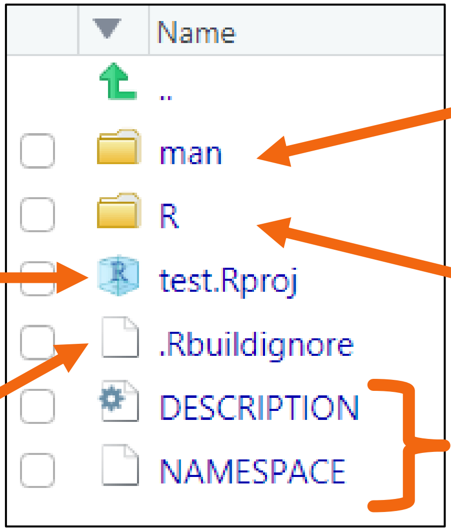


1. Load the **{devtools}** package
2. Create an empty R package called "myRpackage"

<u>Note:</u>

- The functionality of this package will be kept simple
- The main goal is to help the learning process and practice the package development cycle!

File and Folder structure of an R package



Project file

Specifies which files and folders not to include in the package when installed

The **man** folder contains the documentation for the functions

The **R** folder contains the code for the functions

We cover these files later

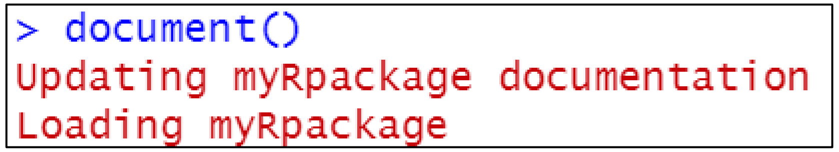
The **man** folder will be missing when you use the `create_package()` function but it will be automatically created with the first documentation step

Document an R package

The function **`document()`** from **{devtools}** is used to build all the documentation for a package.

```
document() # CTRL + SHIFT + D
```

For an empty package this function will simply create the “man” folder if it does not exist.

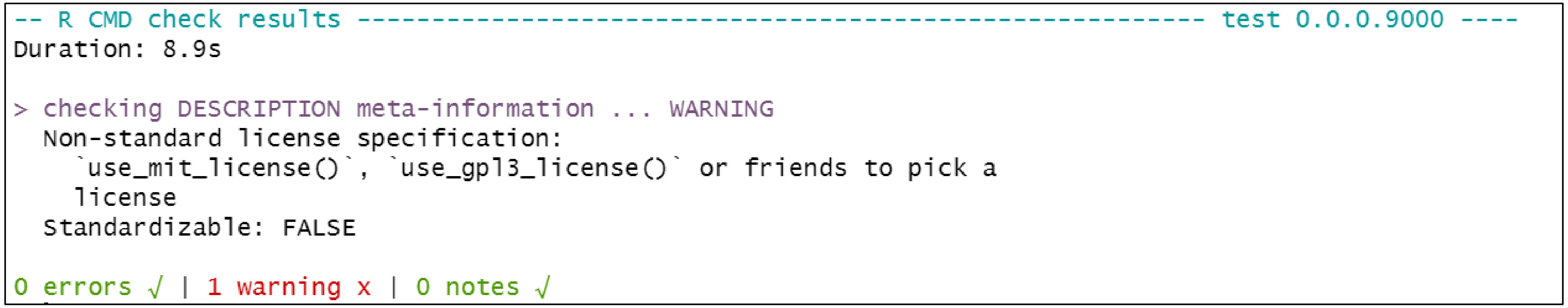


```
> document()
Updating myRpackage documentation
Loading myRpackage
```

Check an R package

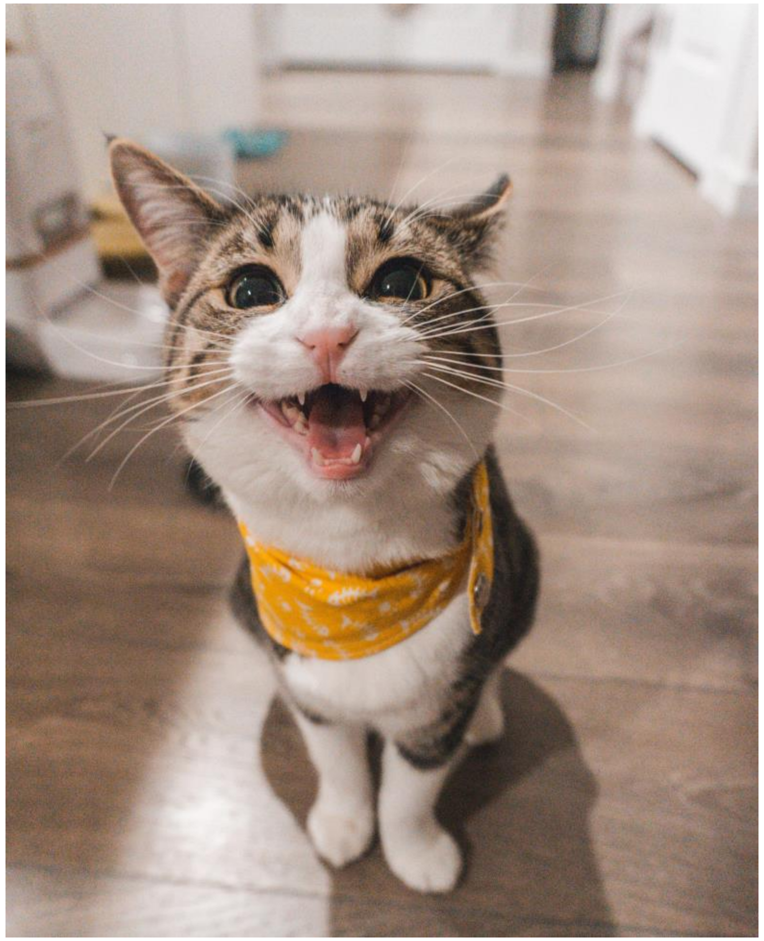
The function **check()** from **{devtools}** automatically builds and checks a package. It runs through a number of checks and will return a summary of the check results.

```
check() # CTRL + SHIFT + E
```



```
-- R CMD check results ---------------------------------------------------------- test 0.0.0.9000 ----
Duration: 8.9s

> checking DESCRIPTION meta-information ... WARNING
  Non-standard license specification:
    `use_mit_license()`, `use_gpl3_license()` or friends to pick a
    license
  Standardizable: FALSE

0 errors √ | 1 warning x | 0 notes √
```

Live Coding Example 2

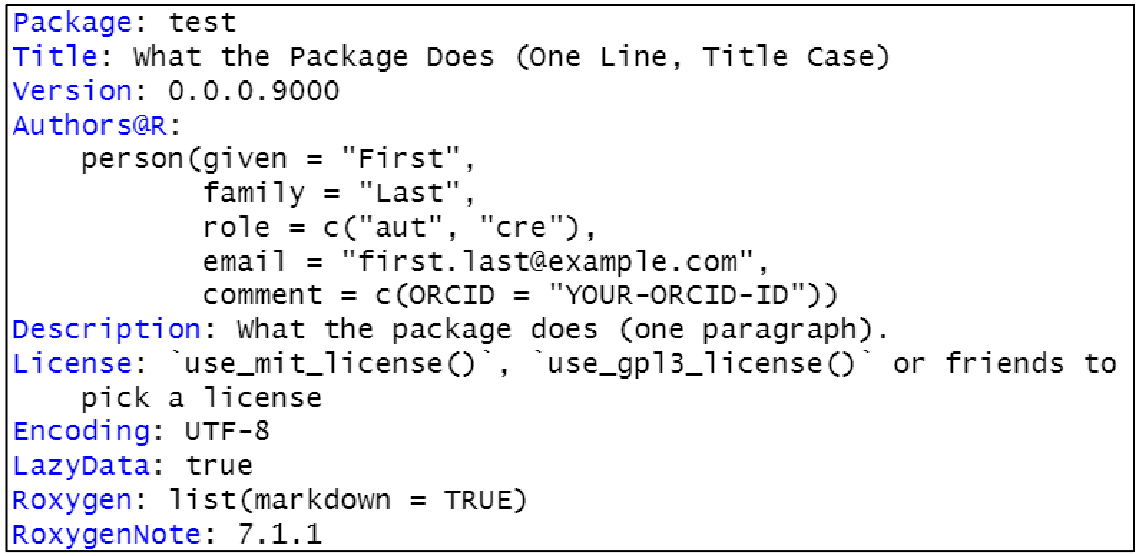


1. Load the **{devtools}** package
2. Document the package **{myRpackage}**
3. Check the package **{myRpackage}**

DESCRIPTION file

The **DESCRIPTION** file is used to store important **metadata** about the package. For example:

- What is the package title
- What is the package version
- Who to contact
- Who can use it (the license)
- What other packages are needed for it to work



```
Package: test
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
    person(given = "First",
           family = "Last",
           role = c("aut", "cre"),
           email = "first.last@example.com",
           comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
```

DESCRIPTION fields

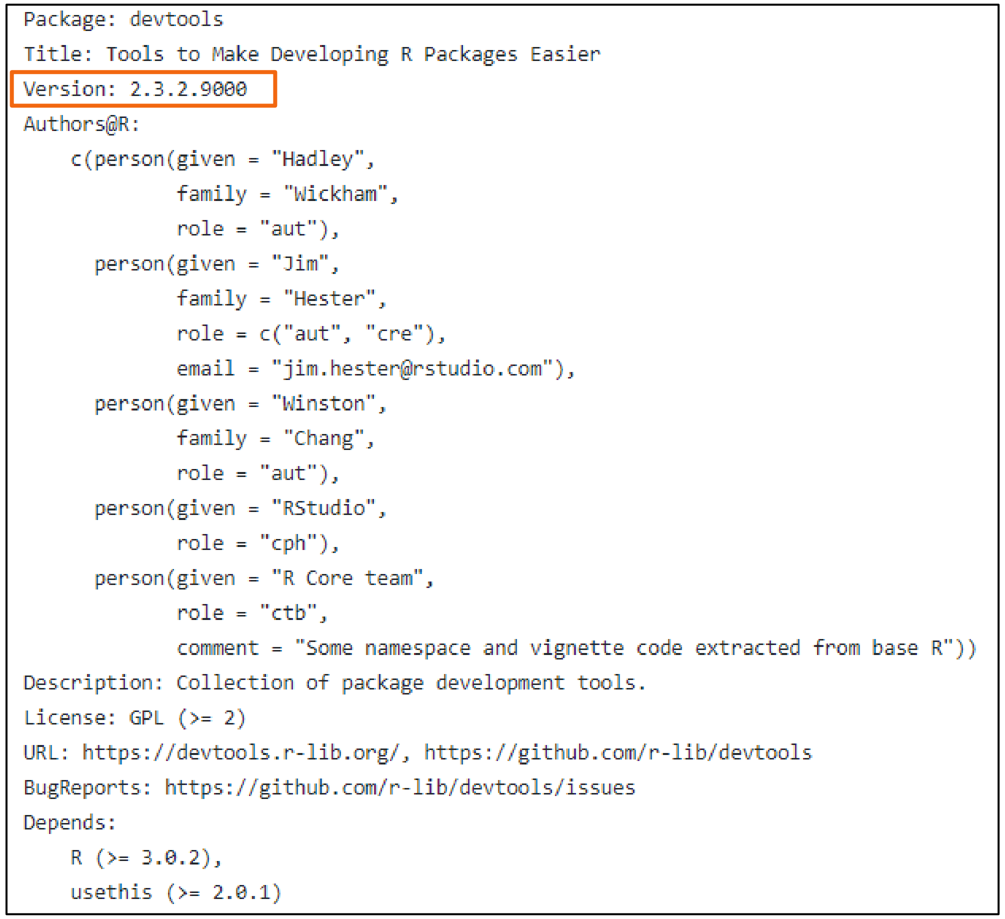
Below is some important guidelines to follow for the DESCRIPTION file fields.

Field	Description
Title	This is typically a one line description of the package. It should be **plain text**, not more than **65 characters** long, **capitalised like a title**, and **NOT end in a period**!
Description	This is a more detailed text about your package. It can be multiple sentences but it is **limited to one paragraph**, each line must be no more than 80 characters wide and new lines must be indented with 4 spaces!
Imports	The packages that are listed in this field **must** be installed on your computer for your package to work because they are being used by your package. If any package is missing, it **will** be **automatically installed** when your package is installed.
Suggests	The packages that are listed in this field can be used by your package but they are **not required**. If any package is missing, it **will not** be **automatically installed** when your package is installed.

Package version

The **Version** field of the DESCRIPTION file shows the package's version number.

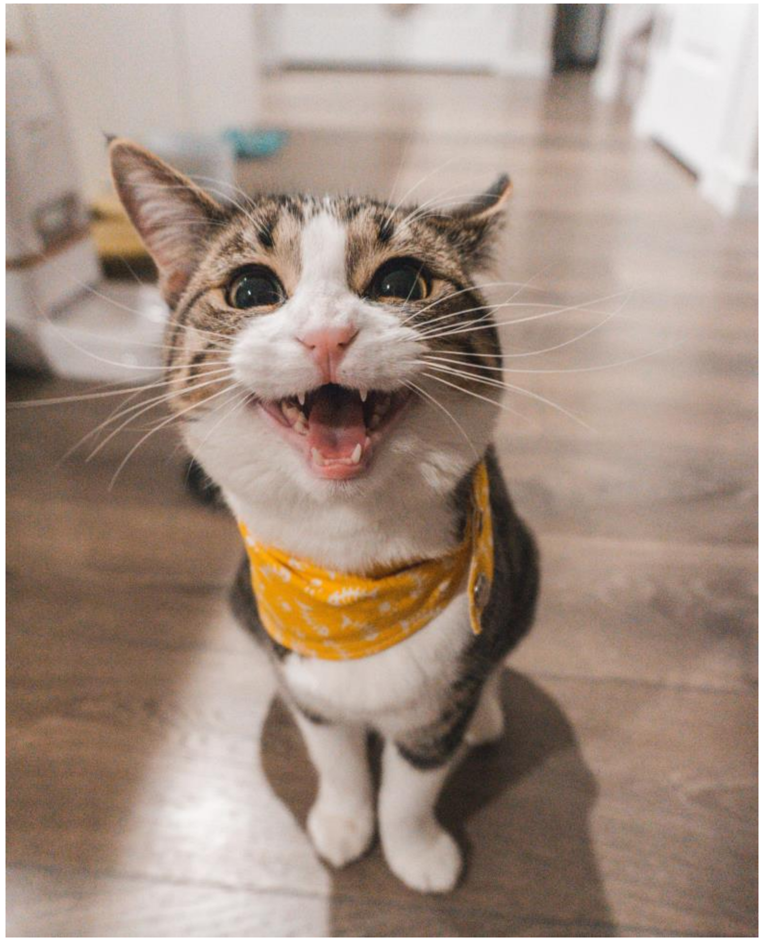
- Typically a version number is made up of three numbers: **<major>.<minor>.<patch>** For example: 1.3.0
- A package that is **in-development** usually has ends with "9000" For example: 1.3.0.9000



```
Package: devtools
Title: Tools to Make Developing R Packages Easier
Version: 2.3.2.9000
Authors@R:
    c(person(given = "Hadley",
             family = "Wickham",
             role = "aut"),
      person(given = "Jim",
             family = "Hester",
             role = c("aut", "cre"),
             email = "jim.hester@rstudio.com"),
      person(given = "Winston",
             family = "Chang",
             role = "aut"),
      person(given = "RStudio",
             role = "cph"),
      person(given = "R Core team",
             role = "ctb",
             comment = "Some namespace and vignette code extracted from base R"))
Description: Collection of package development tools.
License: GPL (>= 2)
URL: https://devtools.r-lib.org/, https://github.com/r-lib/devtools
BugReports: https://github.com/r-lib/devtools/issues
Depends:
    R (>= 3.0.2),
    usethis (>= 2.0.1)
```

Example of DESCRIPTION file of the **{devtools}** package

Live Coding Example 3

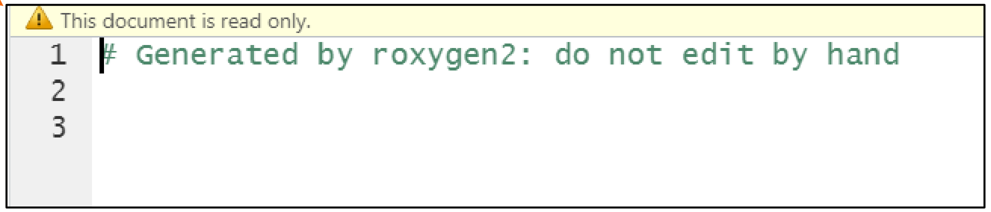


1. Edit the DESCRIPTION file with the metadata about the package (*Title* and *Description*) and your details (*Author*)
2. Save the changes
3. Document the package **{myRpackage}**
4. Check the package **{myRpackage}**

NAMESPACE file

The **NAMESPACE** file can be quite confusing and is considered an advanced topic (hence the warning!).

In a simplified way, it is the file that controls the **communication between packages** and their functions. For example, it manages the **functions** to **export** (from your package) and functions to **import** (from other packages).


This document is read only.

```
# Generated by roxygen2: do not edit by hand


```

What is an R function

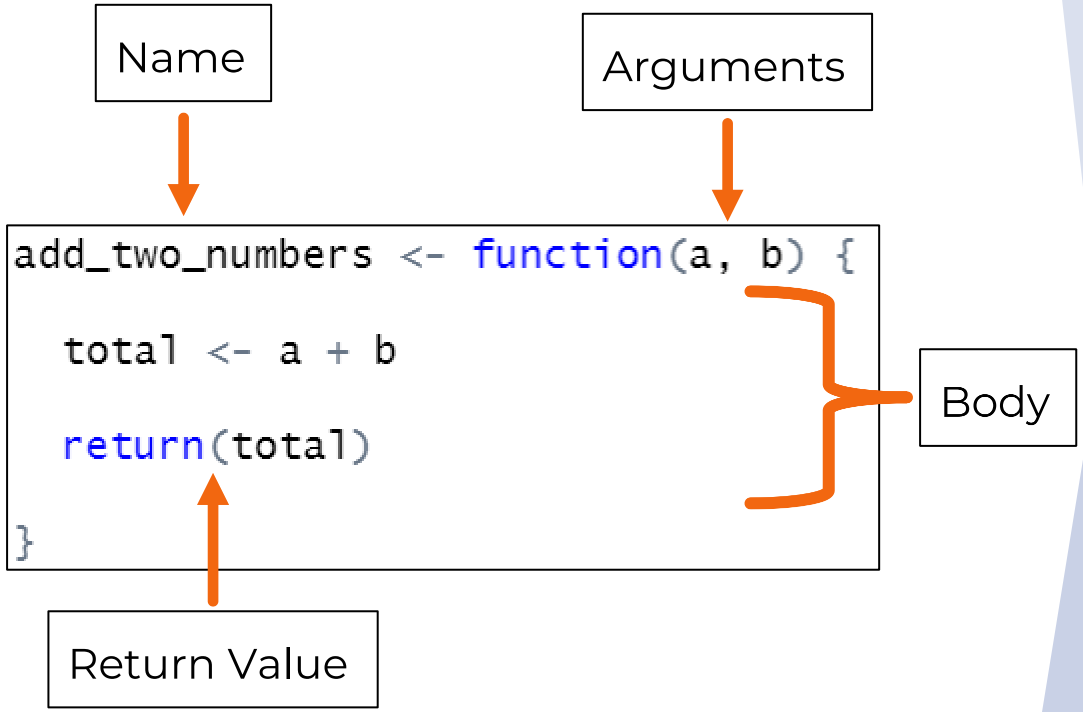
An R function is an **R object** that contains code to be executed.

- In a simplified way, an R function takes **inputs** and generates **outputs**
- R functions are useful when we have code that is **repeated** in a script and therefore help to avoid “copy-pasting” code
- An R function ideally **self-contains** a complex piece of code and is dedicated to solving one task

Components of an R function

An R function is created just like any other object in R and follows a specific structure.

Component	Description
Name	The name of the function
Arguments	The values passed to the function (inputs)
Body	R code that the function executes
Return Value	The value/s the function is required to return (outputs)



```r
add_two_numbers <- function(a, b) {

  total <- a + b

  return(total)

}
```

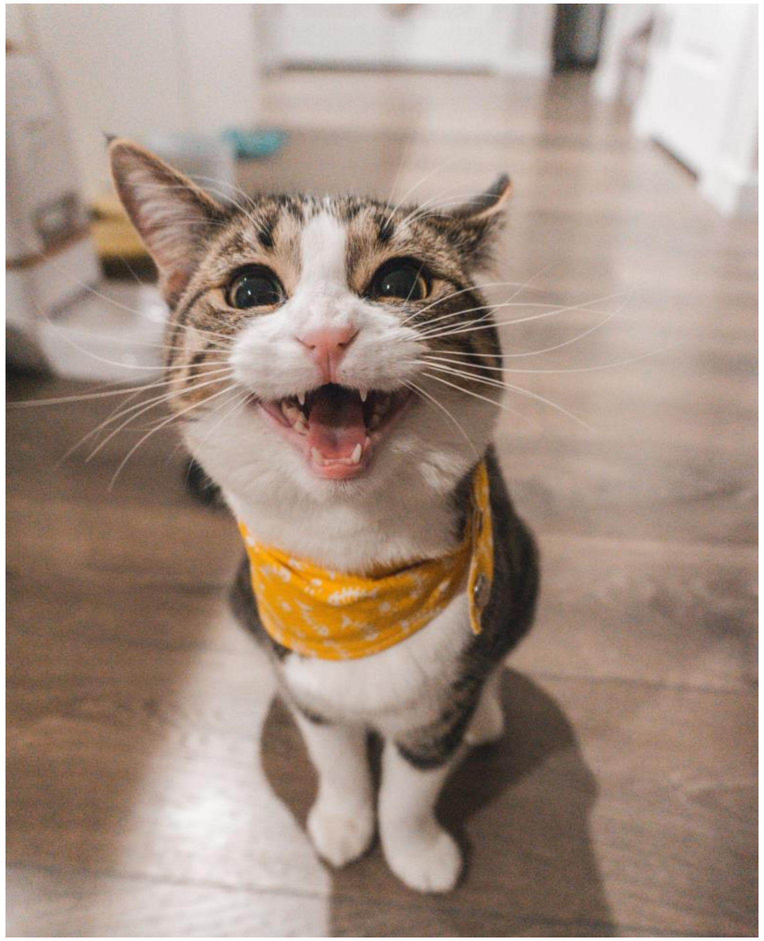
The return value is the last expression evaluated by the function. If the `return()` expression is used then the function will output the contents of `return()`

Create a function

The package **{usethis}** makes it easy for us to **add** R functions to a package. The function **use_r()** takes as input the **function name** and creates the ".R" file for that function inside the "R" folder.

```r
# Create a function file in the R folder
use_r("add_two_numbers")

# Place the function code inside add_two_numbers.R
add_two_numbers <- function(a, b) {

  total <- a + b

  return(total)

}
```

Live Coding Example 4



1. Create the following functions for the package **{myRpackage}**

- `add_two_numbers()`
- `subtract_two_numbers()`

Live Coding Example 4

```r
# Create a function file in the R folder
use_r("subtract_two_numbers")

# Place the function code inside add_two_numbers.R
subtract_two_numbers <- function(a, b) {

  total <- a - b

  return(total)

}
```

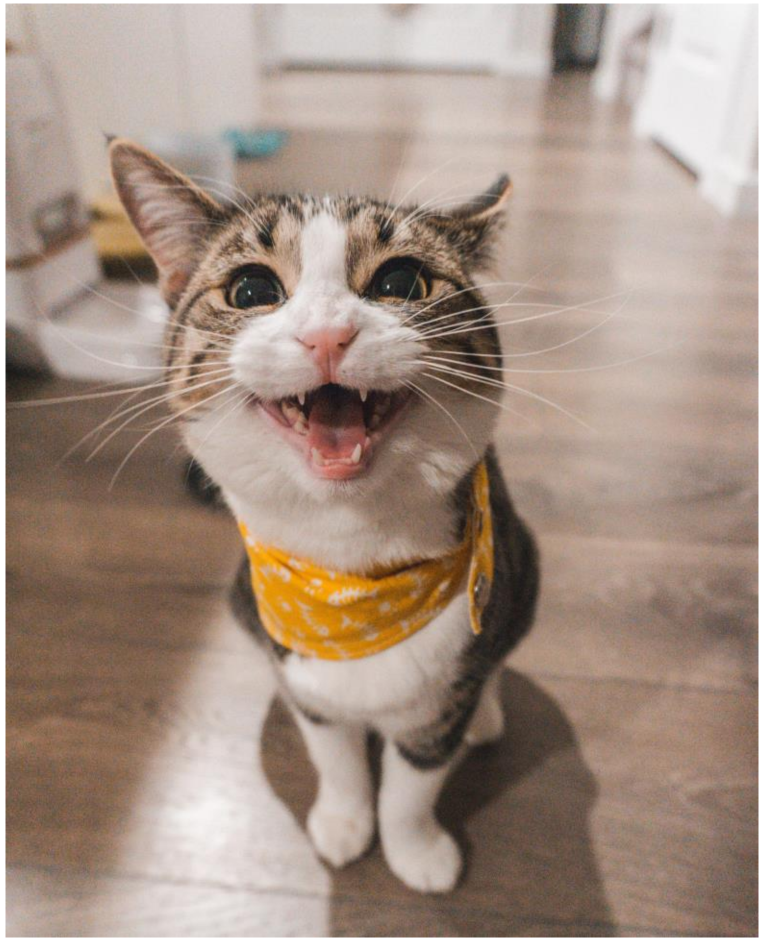
Try out the function

Before we invest more time to properly document and test the function, it is a good idea to first **check if the function works**!

There are three ways to do this:

- (Messy way): Create the function arguments as objects and then run the code inside the function line by line
- (Script way): Source the ".R" file containing the function and call the function
- (Dev way): Use `load_all()` and call the function – more on this later

Live Coding Example 5



Try out the function **`add_two_numbers()`** using the "Messy" and "Script" ways.

Document a function

Function documentation can be somewhat fiddly to work with…

Typically each “.R” file in the “R” folder (containing a function) has its corresponding “.Rd” file in the “man” folder with the same name (containing the documentation).

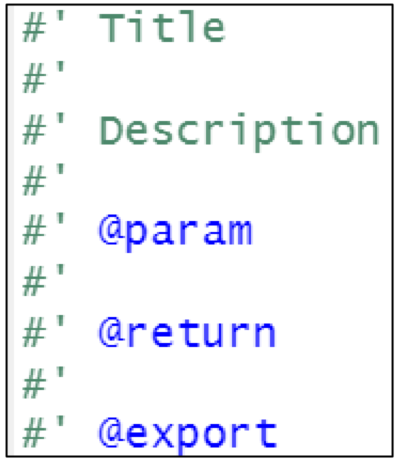
The “.Rd” file that contains the **documentation** in an R-specific markup language … luckily we do not have to worry about that!

Document a function

What we need to do in order to document a function is write specially formatted comments above the function. These are called **roxygen** comments and the package **{roxygen2}** creates and edits the ".Rd" files for us!

Roxygen comments

A **roxygen block** of comments above a function will determine the help provided to the user about the function.



```
#' Title
#'
#' Description
#'
#' @param
#'
#' @return
#'
#' @export
```

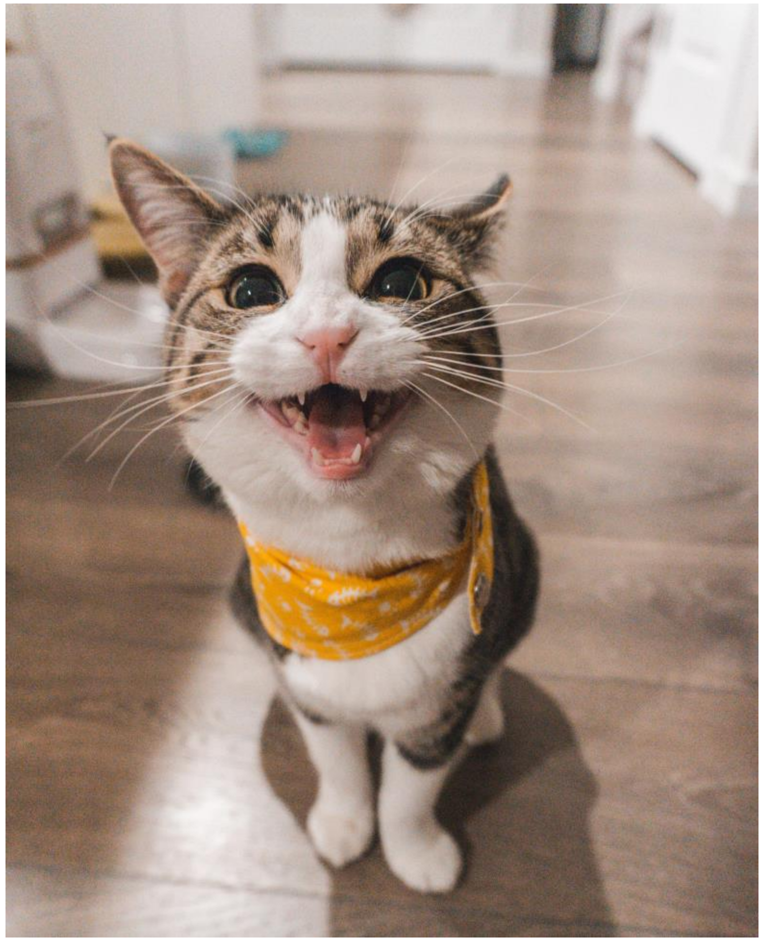
Comment	Description
Title	The first sentence and represents the title of the documentation
Description	The second paragraph and describes what the package does
@param	Argument name followed by a description and what it does
@return	Describes the output of the function
@export	Specifies that the function is usable outside of package

Step 1: Open the ".R" file of the function

Step 2: Place cursor somewhere in the function code

Step 3: Go to → Code → Insert Roxygen Skeleton or CTRL + ALT + SHIFT + R

Live Coding Example 6

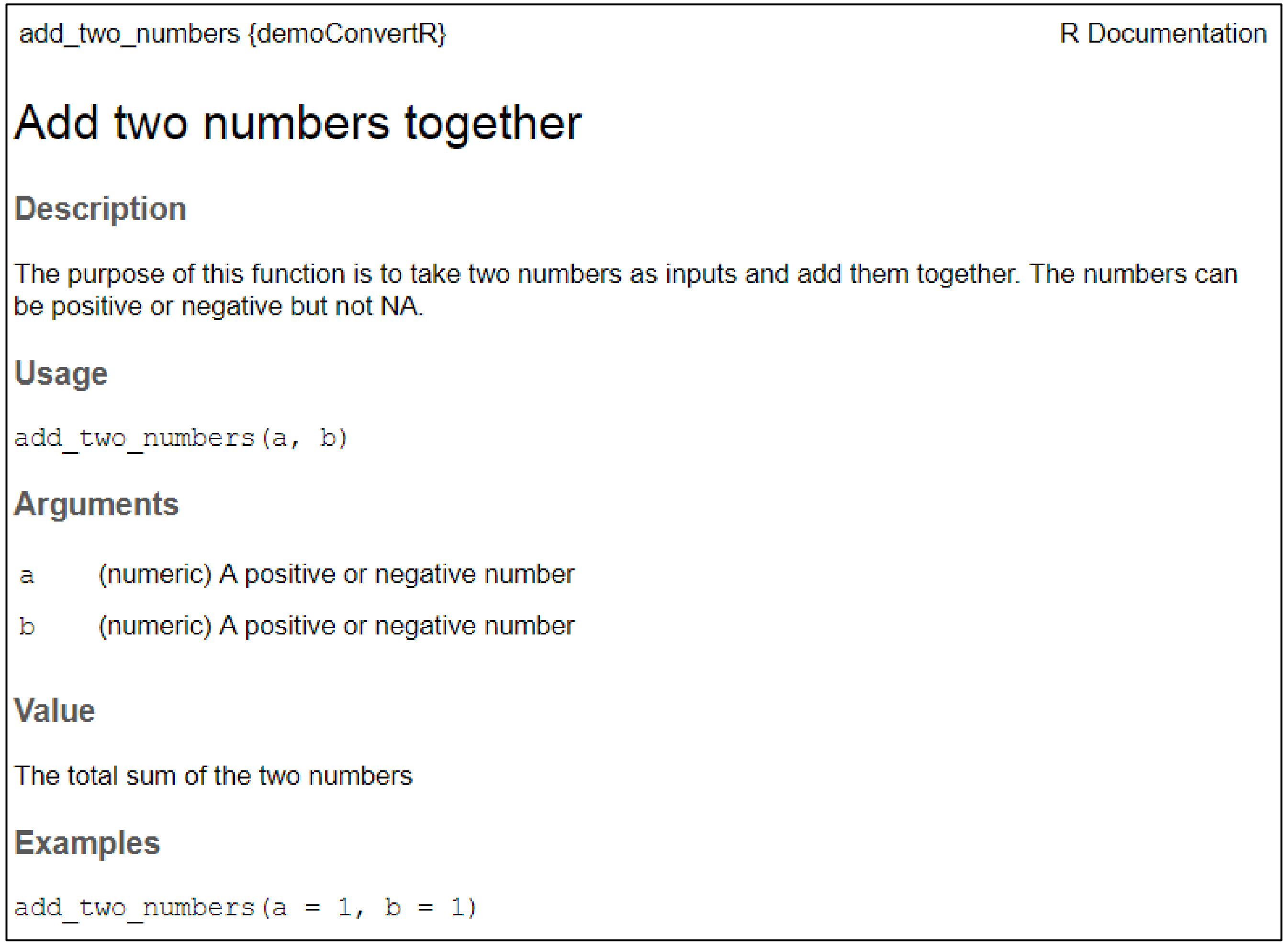


1. Documenting the function **`add_two_numbers()`**
2. Document the package **{myRpackage}**
3. Inspect help information of the function **`add_two_numbers()`**
4. Check the package **{myRpackage}**

Live Coding Example 6

```r
#' Add two numbers together
#'
#' The purpose of this function is to take two numbers as inputs and add them
#' together. The numbers can be positive or negative but not NA.
#'
#' @param a (numeric) A positive or negative number
#' @param b (numeric) A positive or negative number
#'
#' @return The total sum of the two numbers
#'
#' @export
#' @examples
#' add_two_numbers(a = 1, b = 1)
add_two_numbers <- function(a, b) {

  total <- a + b

  return(total)

}
```

Live Coding Example 6



add_two_numbers {demoConvertR}   R Documentation

Add two numbers together

Description

The purpose of this function is to take two numbers as inputs and add them together. The numbers can be positive or negative but not NA.

Usage

```
add_two_numbers(a, b)
```

Arguments

`a`	(numeric) A positive or negative number
`b`	(numeric) A positive or negative number

Value

The total sum of the two numbers

Examples

```
add_two_numbers(a = 1, b = 1)
```

Add tests to a package

The next step is to add **tests**. This is a **formal** way to test the functionality of your package and that your functions work as you expect them to! First we need to initialise testing for the package.

```
use_testthat()
```

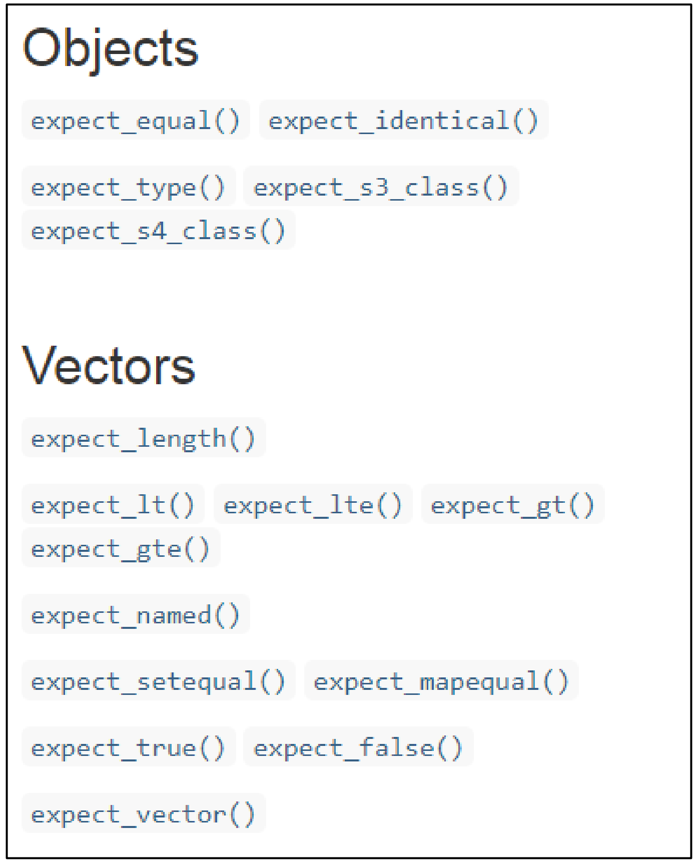
The function **`use_testthat()`** from the **{usethis}** package adds "*Suggests: testthat*" to the DESCRIPTION file. It also creates the folders "tests/testthat/" and adds a generic script "testthat.R" in the "tests" folder.

Spend some time to write unit tests for your functions!

Test a function

The **use_test()** function from **{usethis}** takes as input the function name and creates the "test-[name].R" file for that function inside the "testthat" folder with a generic test ready to edit.

NOTE: It is the developer's responsibility to write the unit tests.


Objects

`expect_equal()` `expect_identical()`

`expect_type()` `expect_s3_class()` `expect_s4_class()`

Vectors

`expect_length()`

`expect_lt()` `expect_lte()` `expect_gt()` `expect_gte()`

`expect_named()`

`expect_setequal()` `expect_mapequal()`

`expect_true()` `expect_false()`

`expect_vector()`

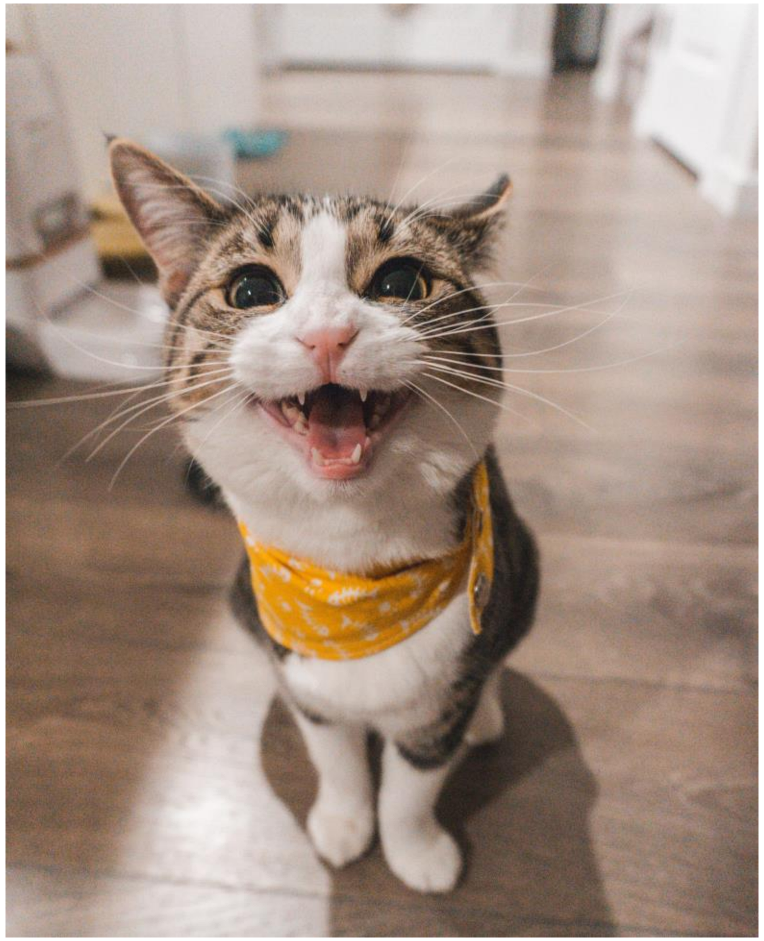
Check out more information here: https://testthat.r-lib.org/reference/index.html

Test an R package

The function **`test()`** from the package **{devtools}** is used to run all of the tests of a package. It also prints out a test report about test **failures**, **warnings**, skipped tests and of course **passes**!

```
test() # CTRL + SHIFT + T
```

Live Coding Example 7



1. Add testing to the package **{myRpackage}**
2. Write some unit tests for the function **`add_two_numbers()`**
3. Test the package **{myRpackage}**
4. Document the package **{myRpackage}**
5. Check the package **{myRpackage}**

Live Coding Example 7

```r
test_that("add_two_numbers returns the correct value and type", {
  expect_identical(object = add_two_numbers(a = 1, b = 2), expected = 3)

  expect_identical(object = add_two_numbers(a = 1, b = -1), expected = 0)

  expect_type(object = add_two_numbers(a = 1, b = 2), type = "double")
})

test_that("add_two_numbers returns NA if one of the arguments is NA", {
  expect_identical(object = add_two_numbers(a = NA, b = 2), expected = as.numeric(NA))
})

test_that("add_two_numbers returns Inf (+/-) if one of the arguments is infinite", {
  expect_identical(object = add_two_numbers(a = Inf, b = 2), expected = Inf)

  expect_identical(object = add_two_numbers(a = -Inf, b = 2), expected = -Inf)
})
```

Live Coding Example 7

❖ Add testing components

```
√ Adding 'testthat' to Suggests field in DESCRIPTION
√ Setting Config/testthat/edition field in DESCRIPTION to '3'
√ Creating 'tests/testthat/'
√ Writing 'tests/testthat.R'
```

❖ Add test for function

```
√ Writing 'tests/testthat/test-add_two_numbers.R'
* Modify 'tests/testthat/test-add_two_numbers.R'
```

❖ Check DESCRIPTION file

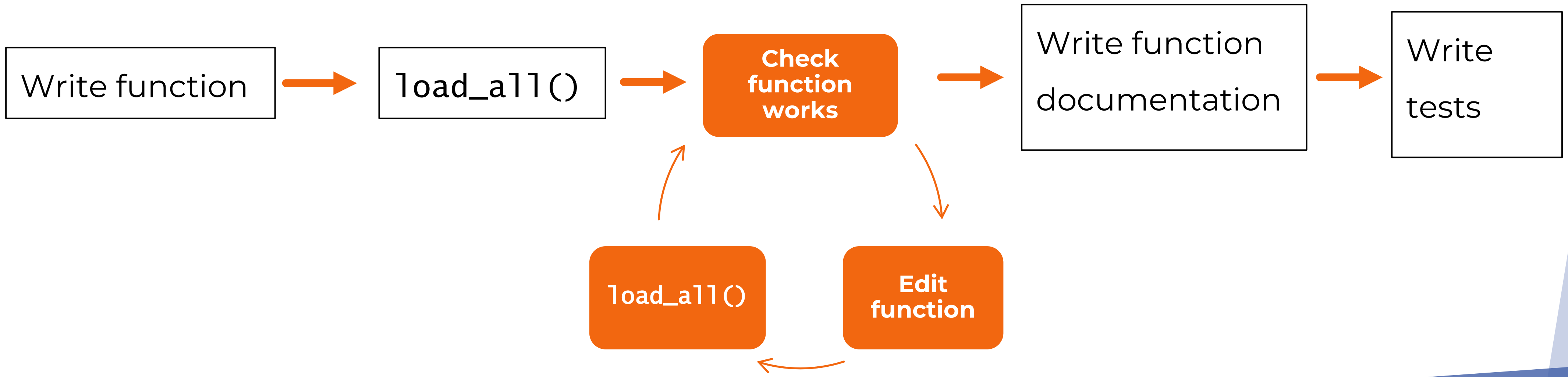
```
Suggests:
    testthat (>= 3.0.0)
Config/testthat/edition: 3
```

❖ Run tests

```
Loading demoConvertR
Testing demoConvertR
√ |  OK F W S | Context
√ |   6       | add_two_numbers [0.1 s]

== Results ==============================
Duration: 0.2 s

[ FAIL 0 | WARN 0 | SKIP 0 | PASS 6 ]
```

Test drive a package

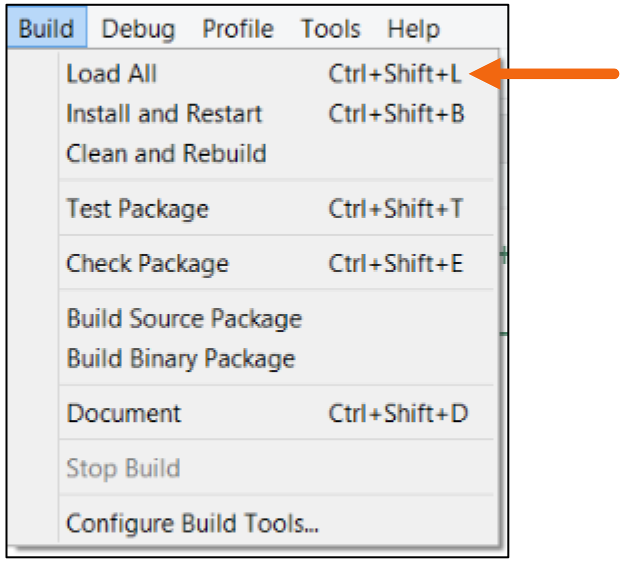
It is a good idea to **regularly** test drive the functionality that we develop in our package. This means **fewer bugs** to worry about! An important function to use during the development cycle is the **`load_all()`** of the **{devtools}** package.



Test drive a package

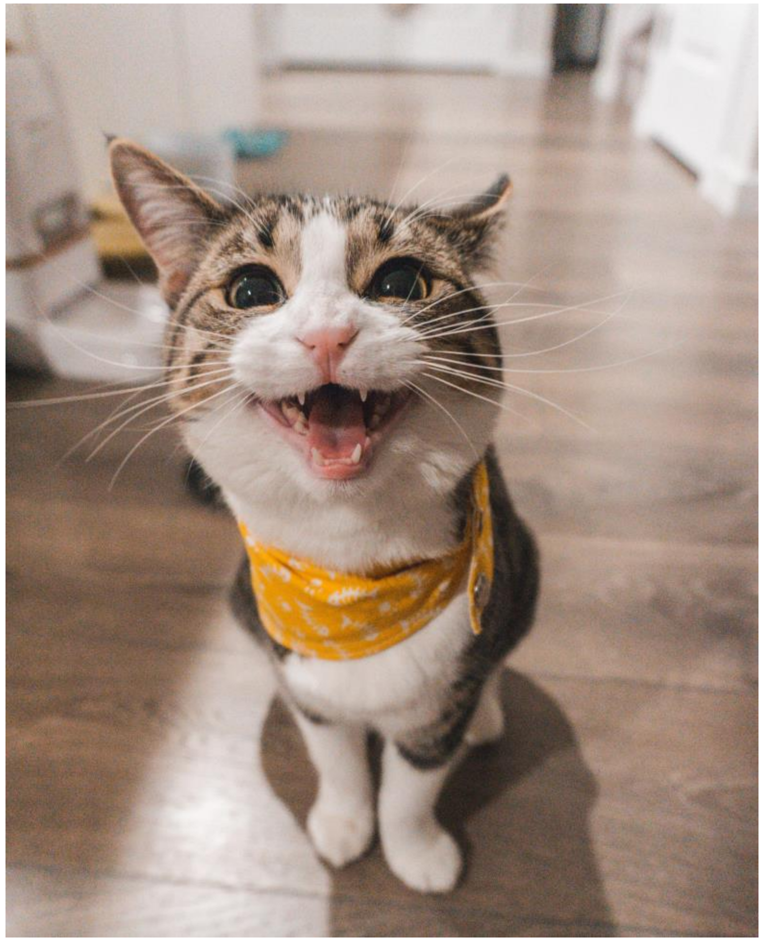
The function **`load_all()`** is very useful because it allows you to **interact** with your package and its functions.

You can think of it as a way to **simulate** what happens when a package is installed and loaded with **`library()`**.



```r
library(devtools)

# Working directory is set at the top level of package

load_all() # CTRL + SHIFT + L
```

Live Coding Example 8



1. Restart the R session

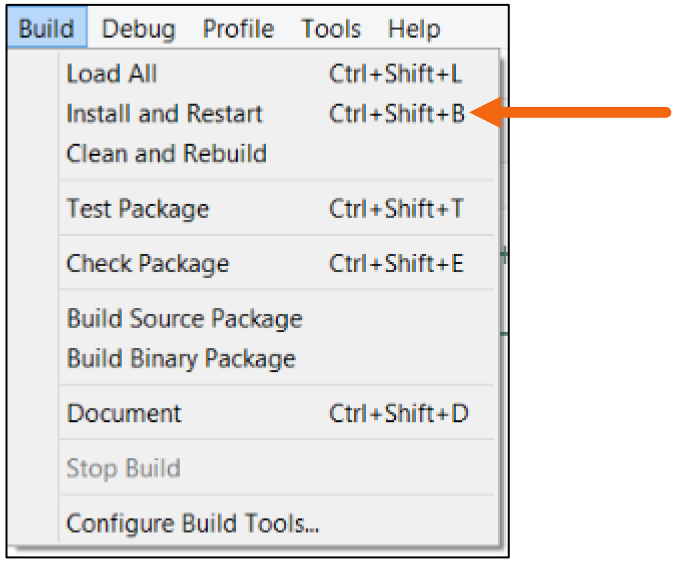
Go to → Session → Restart R

(or CTRL + SHIFT + F10)

2. Test drive the **{myRpackage}** package using the `load_all()` function

Install a package

The function **`install()`** from the **{devtools}** package installs the package from the source state. More specifically the “R CMD INSTALL” command is executed behind the scenes.



```
install()
```

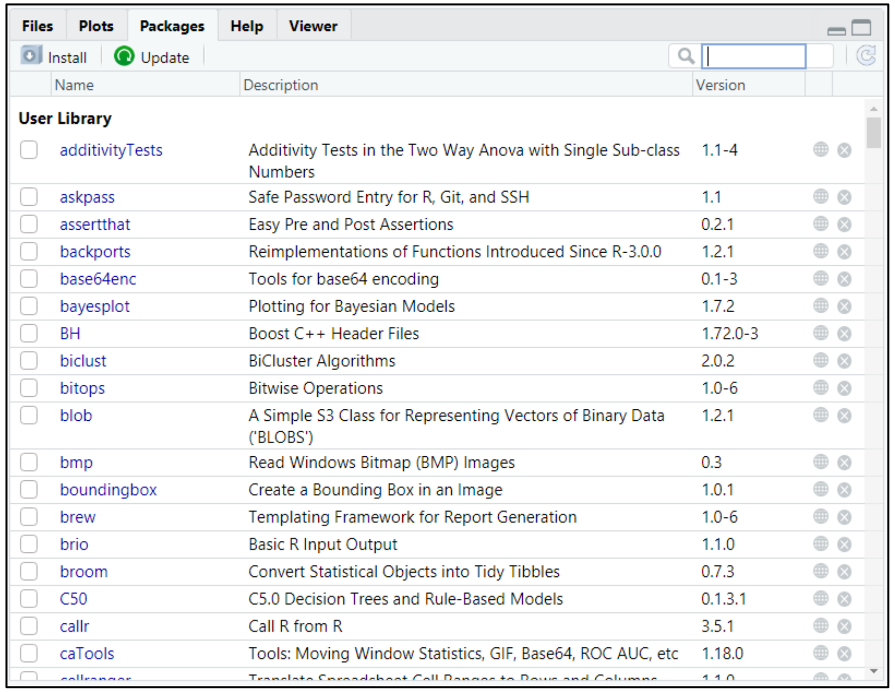
Alternatively the option “Install and Restart” (CTRL + SHIFT + B) installs the package, restarts R and loads it.

Uninstall a package

An installed package is stored on your computer in a **library directory**.

Typically we update, however it is a good idea to know how to uninstall them.

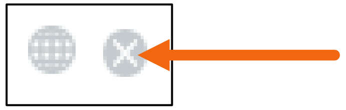
The function **`remove.packages()`** from the **{utils}** package removes a package from the library directory.



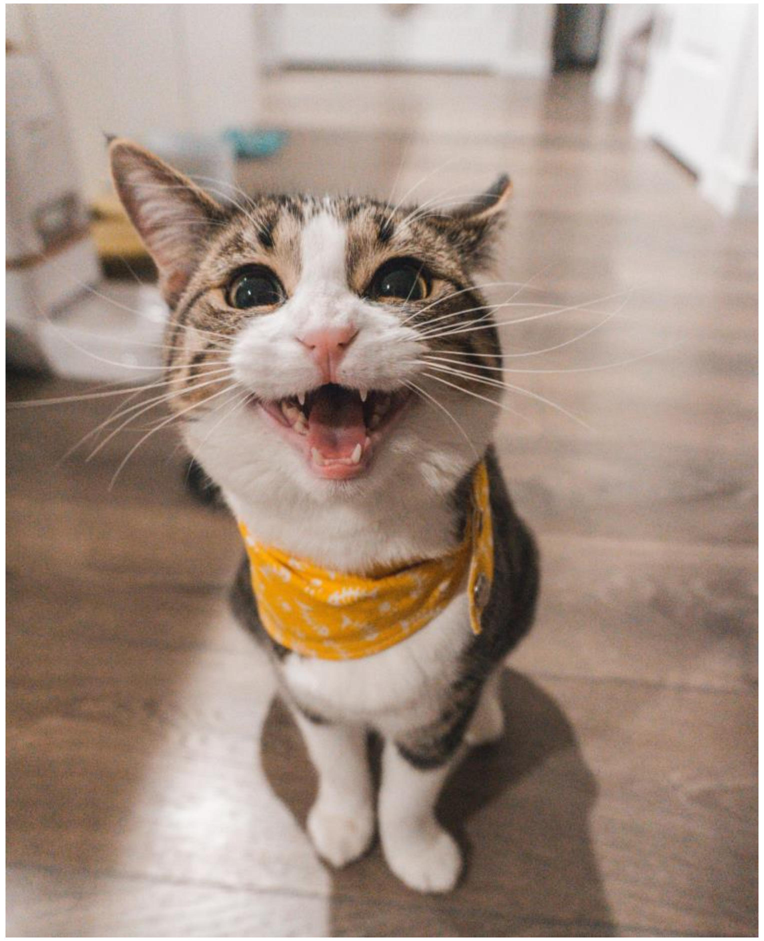
	Name	Description	Version
User Library			
	additivityTests	Additivity Tests in the Two Way Anova with Single Sub-class Numbers	1.1-4
	askpass	Safe Password Entry for R, Git, and SSH	1.1
	assertthat	Easy Pre and Post Assertions	0.2.1
	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.2.1
	base64enc	Tools for base64 encoding	0.1-3
	bayesplot	Plotting for Bayesian Models	1.7.2
	BH	Boost C++ Header Files	1.72.0-3
	biclust	BiCluster Algorithms	2.0.2
	bitops	Bitwise Operations	1.0-6
	blob	A Simple S3 Class for Representing Vectors of Binary Data ('BLOBS')	1.2.1
	bmp	Read Windows Bitmap (BMP) Images	0.3
	boundingbox	Create a Bounding Box in an Image	1.0.1
	brew	Templating Framework for Report Generation	1.0-6
	brio	Basic R Input Output	1.1.0
	broom	Convert Statistical Objects into Tidy Tibbles	0.7.3
	C50	C5.0 Decision Trees and Rule-Based Models	0.1.3.1
	callr	Call R from R	3.5.1
	caTools	Tools: Moving Window Statistics, GIF, Base64, ROC AUC, etc	1.18.0

```r
remove.packages()
```

You can also navigate to the “Packages” tab, search for the package and click on the cross to uninstall the package



Live Coding Example 9



1. Install the **{myRpackage}** package
2. Try out the package
3. Uninstall the **{myRpackage}** package

Install a package from GitHub

We can also install an R package that is available on a **public GitHub repo**. The **{devtools}** package offers the function **`install_github()`** which installs a package directly from GitHub (GitHub **username** and repository **name** are needed for this function).

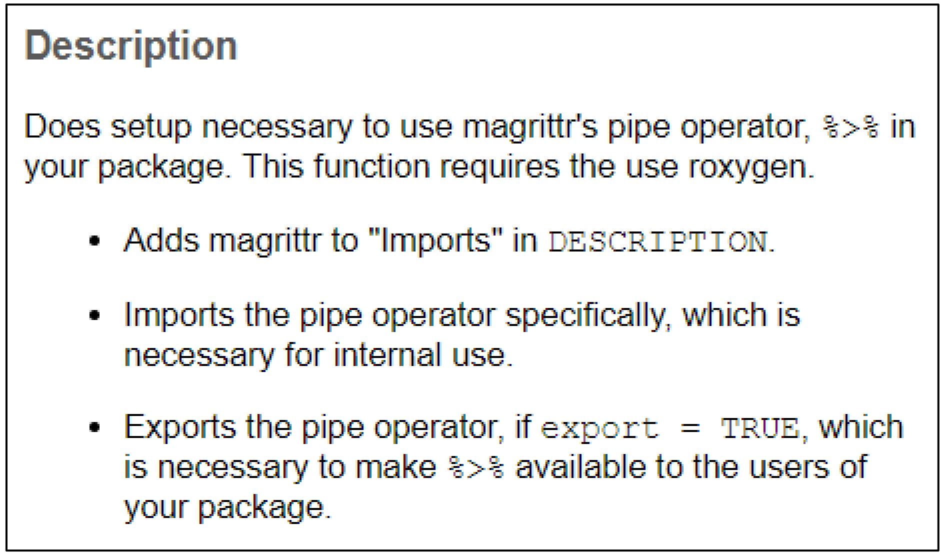
For example:

```
install_github("tidyverse/dplyr")
```

This is useful when a package is not available to install from CRAN or you want to install the latest in-development version of a package.

Add the pipe operator to your package

The pipe operator (**%>%**) from the **{magrittr}** package is extremely useful when writing code and especially for data transformations. To use the pipe operator within a package that you are developing call the function **`use_pipe()`** from **{usethis}** which carries out the necessary setup.



Description

Does setup necessary to use magrittr's pipe operator, `%>%` in your package. This function requires the use roxygen.

- Adds magrittr to "Imports" in `DESCRIPTION`.
- Imports the pipe operator specifically, which is necessary for internal use.
- Exports the pipe operator, if `export = TRUE`, which is necessary to make `%>%` available to the users of your package.

Use other packages in your package

It is very likely that you will want to use functionality from another package within your package. To do this you need to **add** the "external" package to the "**Imports**" field of the **DESCRIPTION** file.

An easy way to do this is by using the function **`use_package()`** from **{usethis}**. For example:

```
use_package("dplyr")
```

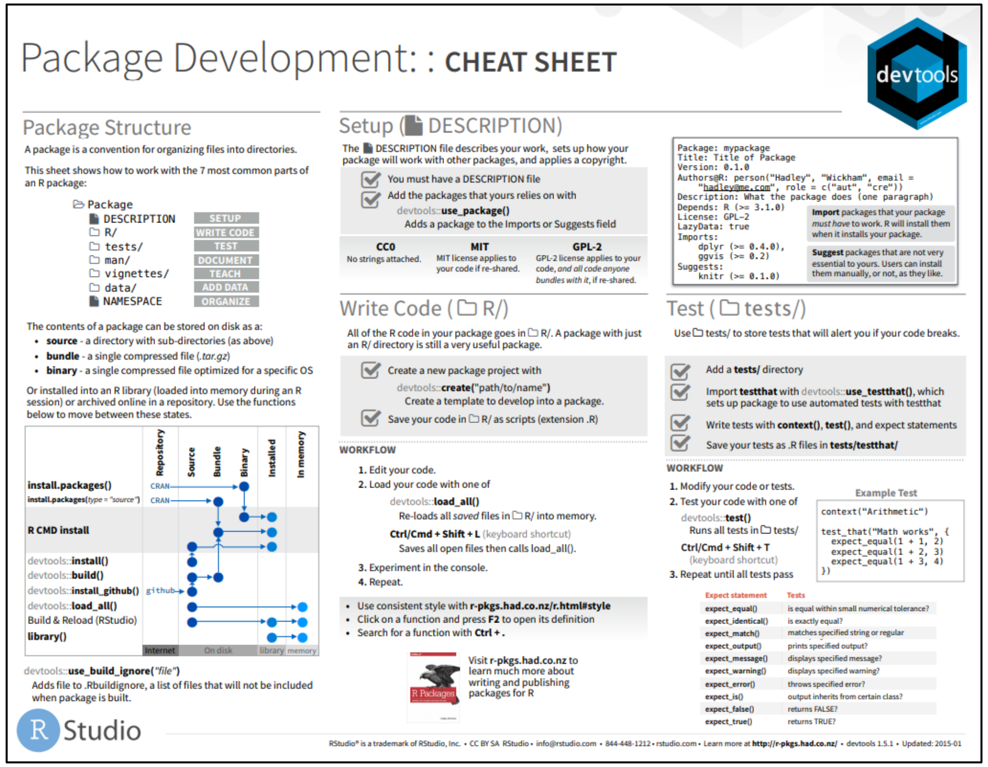
This adds the **{dplyr}** package to the "**Imports**" field of the **DESCRIPTION** file. It is recommended to use the `::` operator when using functions from other packages. For example: `dplyr::filter()`

Keyboard Shortcuts

Below is the collection of keyboard shortcuts seen in this course that helps speed up the development cycle of an R package.

Shortcut	Description
CTRL + SHIFT + D	Runs `document()` – build all the documentation for a package
CTRL + SHIFT + E	Runs `check()` – builds and checks a package
CTRL + ALT + SHIFT + R	Go to “Code” then “Insert Roxygen Skeleton”
CTRL + SHIFT + T	Runs `test()` – run all the tests of a package
CTRL + SHIFT + L	Runs `load_all()` – enables test driving a package
CTRL + SHIFT + F10	Go to “Session” then “Restart R” to Restart R session
CTRL + SHIFT + B	Installs the package, restarts R and loads it

Package Development cheat sheet



Package Development: : **CHEAT SHEET**

Package Structure

A package is a convention for organizing files into directories.

This sheet shows how to work with the 7 most common parts of an R package:

```
Package
  DESCRIPTION    SETUP
  R/             WRITE CODE
  tests/         TEST
  man/           DOCUMENT
  vignettes/     TEACH
  data/          ADD DATA
  NAMESPACE      ORGANIZE
```

The contents of a package can be stored on disk as a:

- **source** - a directory with sub-directories (as above)
- **bundle** - a single compressed file (*.tar.gz*)
- **binary** - a single compressed file optimized for a specific OS

Or installed into an R library (loaded into memory during an R session) or archived online in a repository. Use the functions below to move between these states.

devtools::**use_build_ignore**("*file*")
Adds file to .Rbuildignore, a list of files that will not be included when package is built.

Setup (DESCRIPTION)

The DESCRIPTION file describes your work, sets up how your package will work with other packages, and applies a copyright.

- ✓ You must have a DESCRIPTION file
- ✓ Add the packages that yours relies on with
  devtools::**use_package()**
  Adds a package to the Imports or Suggests field

CC0	MIT	GPL-2
No strings attached.	MIT license applies to your code if re-shared.	GPL-2 license applies to your code, *and all code anyone bundles with it*, if re-shared.

```
Package: mypackage
Title: Title of Package
Version: 0.1.0
Authors@R: person("Hadley", "Wickham", email =
    "hadley@me.com", role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: GPL-2
LazyData: true
Imports:
    dplyr (>= 0.4.0),
    ggvis (>= 0.2)
Suggests:
    knitr (>= 0.1.0)
```

Import packages that your package *must have* to work. R will install them when it installs your package.

Suggest packages that are not very essential to yours. Users can install them manually, or not, as they like.

Write Code (R/)

All of the R code in your package goes in R/. A package with just an R/ directory is still a very useful package.

- ✓ Create a new package project with
  devtools::**create**("path/to/name")
  Create a template to develop into a package.
- ✓ Save your code in R/ as scripts (extension .R)

WORKFLOW

1. Edit your code.
2. Load your code with one of
   devtools::**load_all()**
   Re-loads all *saved* files in R/ into memory.
   Ctrl/Cmd + Shift + L (keyboard shortcut)
   Saves all open files then calls load_all().
3. Experiment in the console.
4. Repeat.

- Use consistent style with **r-pkgs.had.co.nz/r.html#style**
- Click on a function and press **F2** to open its definition
- Search for a function with **Ctrl + .**

Visit **r-pkgs.had.co.nz** to learn much more about writing and publishing packages for R

Test (tests/)

Use tests/ to store tests that will alert you if your code breaks.

- ✓ Add a **tests/** directory
- ✓ Import **testthat** with devtools::**use_testthat()**, which sets up package to use automated tests with testthat
- ✓ Write tests with **context()**, **test()**, and expect statements
- ✓ Save your tests as .R files in **tests/testthat/**

WORKFLOW

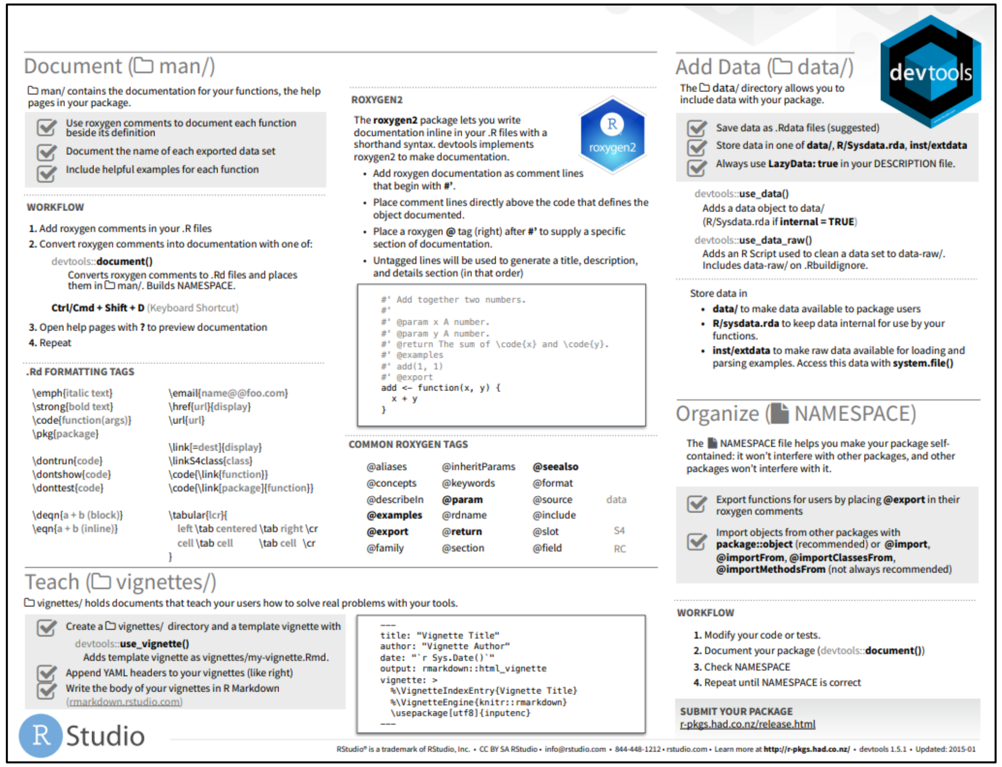
1. Modify your code or tests.
2. Test your code with one of
   devtools::**test()**
   Runs all tests in tests/
   Ctrl/Cmd + Shift + T
   (keyboard shortcut)
3. Repeat until all tests pass

Example Test

```
context("Arithmetic")

test_that("Math works", {
  expect_equal(1 + 1, 2)
  expect_equal(1 + 2, 3)
  expect_equal(1 + 3, 4)
})
```

Expect statement	Tests
expect_equal()	is equal within small numerical tolerance?
expect_identical()	is exactly equal?
expect_match()	matches specified string or regular
expect_output()	prints specified output?
expect_message()	displays specified message?
expect_warning()	displays specified warning?
expect_error()	throws specified error?
expect_is()	output inherits from certain class?
expect_false()	returns FALSE?
expect_true()	returns TRUE?

RStudio



Document (man/)

man/ contains the documentation for your functions, the help pages in your package.

- ✓ Use roxygen comments to document each function beside its definition
- ✓ Document the name of each exported data set
- ✓ Include helpful examples for each function

WORKFLOW

1. Add roxygen comments in your .R files
2. Convert roxygen comments into documentation with one of:
   devtools::**document()**
   Converts roxygen comments to .Rd files and places them in man/. Builds NAMESPACE.
   Ctrl/Cmd + Shift + D (Keyboard Shortcut)
3. Open help pages with **?** to preview documentation
4. Repeat

.Rd FORMATTING TAGS

```
\emph{italic text}          \email{name@@foo.com}
\strong{bold text}          \href{url}{display}
\code{function(args)}       \url{url}
\pkg{package}
                            \link[=dest]{display}
\dontrun{code}              \linkS4class{class}
\dontshow{code}             \code{\link{function}}
\donttest{code}             \code{\link[package]{function}}

\deqn{a + b (block)}        \tabular{lcr}{
\eqn{a + b (inline)}          left \tab centered \tab right \cr
                              cell \tab cell     \tab cell  \cr
                            }
```

ROXYGEN2

The **roxygen2** package lets you write documentation inline in your .R files with a shorthand syntax. devtools implements roxygen2 to make documentation.

- Add roxygen documentation as comment lines that begin with **#'**.
- Place comment lines directly above the code that defines the object documented.
- Place a roxygen **@** tag (right) after **#'** to supply a specific section of documentation.
- Untagged lines will be used to generate a title, description, and details section (in that order)

```
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#' @return The sum of \code{x} and \code{y}.
#' @examples
#' add(1, 1)
#' @export
add <- function(x, y) {
  x + y
}
```

COMMON ROXYGEN TAGS

@aliases	@inheritParams	**@seealso**	
@concepts	@keywords	@format	
@describeIn	**@param**	@source	data
@examples	@rdname	@include	
@export	**@return**	@slot	S4
@family	@section	@field	RC

Teach (vignettes/)

vignettes/ holds documents that teach your users how to solve real problems with your tools.

- ✓ Create a vignettes/ directory and a template vignette with
  devtools::**use_vignette()**
  Adds template vignette as vignettes/my-vignette.Rmd.
- ✓ Append YAML headers to your vignettes (like right)
- ✓ Write the body of your vignettes in R Markdown (rmarkdown.rstudio.com)

```
---
title: "Vignette Title"
author: "Vignette Author"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

Add Data (data/)

The data/ directory allows you to include data with your package.

- ✓ Save data as .Rdata files (suggested)
- ✓ Store data in one of **data/**, **R/Sysdata.rda**, **inst/extdata**
- ✓ Always use **LazyData: true** in your DESCRIPTION file.

devtools::**use_data()**
Adds a data object to data/
(R/Sysdata.rda if **internal = TRUE**)

devtools::**use_data_raw()**
Adds an R Script used to clean a data set to data-raw/. Includes data-raw/ on .Rbuildignore.

Store data in

- **data/** to make data available to package users
- **R/sysdata.rda** to keep data internal for use by your functions.
- **inst/extdata** to make raw data available for loading and parsing examples. Access this data with **system.file()**

Organize (NAMESPACE)

The NAMESPACE file helps you make your package self-contained: it won't interfere with other packages, and other packages won't interfere with it.

- ✓ Export functions for users by placing **@export** in their roxygen comments
- ✓ Import objects from other packages with **package::object** (recommended) or **@import**, **@importFrom**, **@importClassesFrom**, **@importMethodsFrom** (not always recommended)

WORKFLOW

1. Modify your code or tests.
2. Document your package (devtools::**document()**)
3. Check NAMESPACE
4. Repeat until NAMESPACE is correct

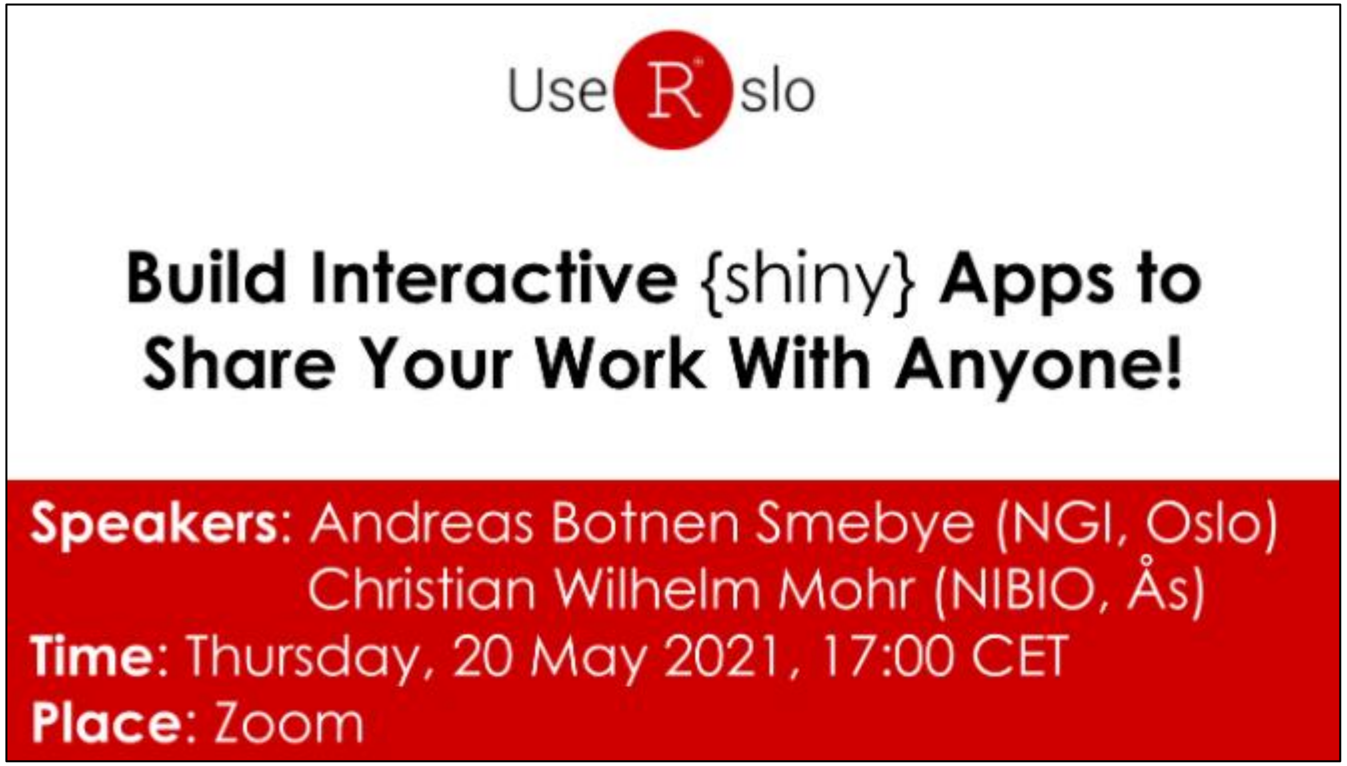
SUBMIT YOUR PACKAGE
r-pkgs.had.co.nz/release.html

RStudio

https://github.com/rstudio/cheatsheets/raw/master/package-development.pdf

Next online R event!

Build Interactive {shiny} Apps to Share Your Work With Anyone!



UseR Oslo

Build Interactive {shiny} **Apps to Share Your Work With Anyone!**

Speakers: Andreas Botnen Smebye (NGI, Oslo)
Christian Wilhelm Mohr (NIBIO, Ås)
Time: Thursday, 20 May 2021, 17:00 CET
Place: Zoom

https://www.meetup.com/Oslo-useR-Group/events/277702734/

Thank you to our sponsors and partners!

MANGO SOLUTIONS

RStudio®