

Create Your Own Package

Presented by:

Partow Moradi

Rebecca Abt

Raphael Semiz



Why should you program your own R package?

- ... you want to easily reuse your code
- ... you collaborate with others on an ongoing project
- ... you want to document your functions clearly
- ... you want to standardize your R code (consistent layout for diagrams)

Best Reason:

-> *they save you time*

What steps do you need to take?

- Step 1: Set Up Your Environment
- Step 2: Create Package Structure
- Step 3: Add Functions
- Step 4: Document Your Functions
- Step 5: Add Metadata
- Step 6: Build and Install Your Package

Set Up Your Environment

- **Install Necessary Packages:** Make sure you have `devtools` and `roxygen2` installed. You can install them using (command line):

```
install.packages("devtools")  
install.packages("roxygen2")
```

- **Load the Packages:**

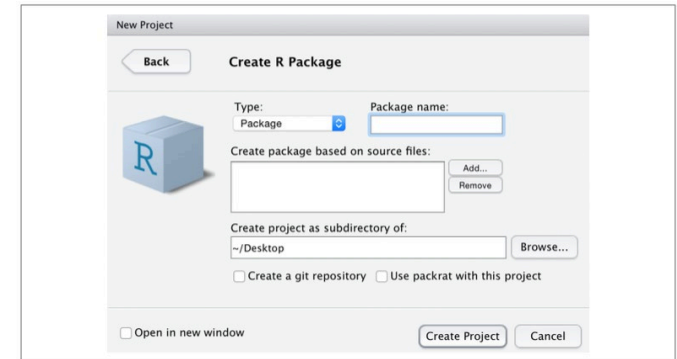
```
library(devtools)  
library(roxygen2)
```

Naming your package

"There are only two hard things in computer science: cache invalidation and naming things"
- Phil Karlton

Tips for the perfect name:

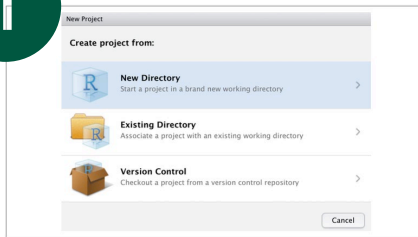
- Choose a name that can be easily Googled
- Avoid using upper- and lowercase letters:
-> *Rgtk2*, *RGTK2*, *RGtk2*
- Find a word that evokes the problem:
-> *plyr* evokes plier
-> *knitr* (knit+r) is "neater" than *sweave* (s+weave)
- Use abbreviations:
-> *Rcpp* = R+C++ (plus plus)
-> *lvplot* = letter value plots
- Add an extra R:
-> *stringr*, *tourr*



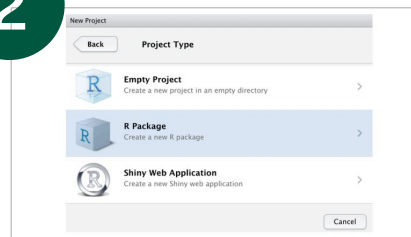
You can also check if a name is already used on CRAN by loading
[http://cran.r-project.org/web/packages/\[PACKAGE_NAME\]](http://cran.r-project.org/web/packages/[PACKAGE_NAME])

Creating a package

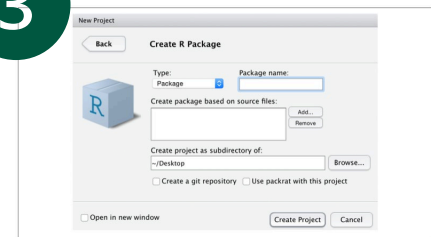
1



2



3



Alternatively, you can create a new package from within R by running the following:

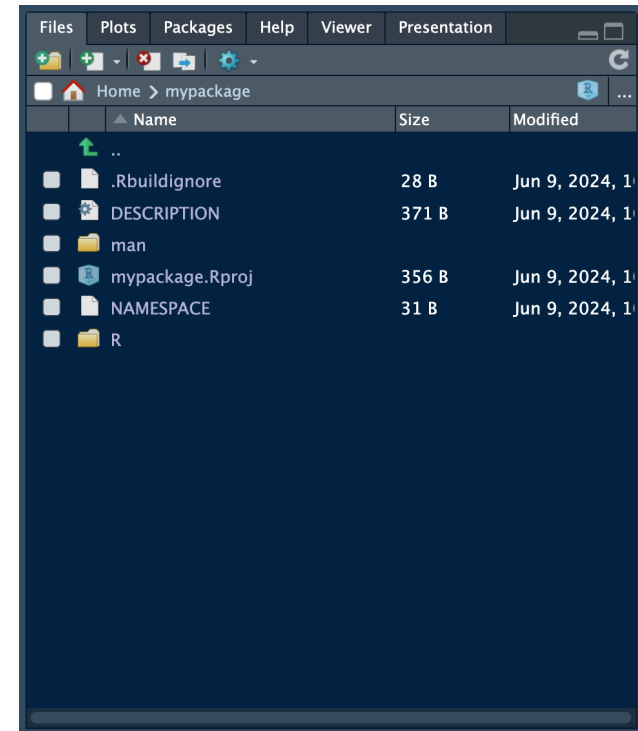
```
devtools::create("path/to/package/pkgname")
```

Five Stages of a package

- *Source package*: development version
- *Bundled package*: compressed into single file
- *Binary package*: platform specific
- *Installed package*: decompressed into package library
- *In-Memory-package*: `library()`

Components of a package

- Directory *R/*
- File *DESCRIPTION*
- Directory *man/*



Directory R/

- This is where your code is located
- Good coding style
- Object names
 - > variables and functions should be lowercase
 - > variables = nouns; functions = verbs
 - > avoid using names of existing names

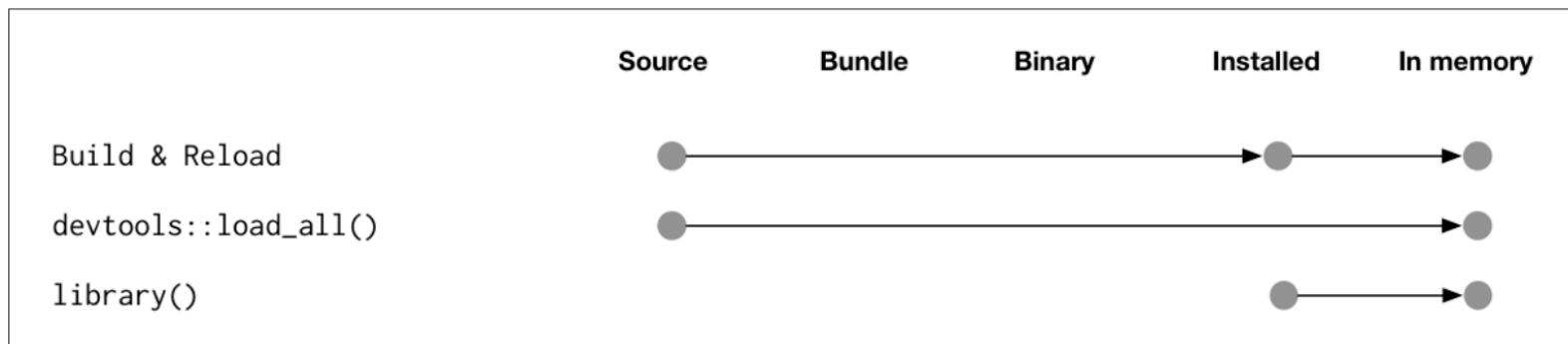
```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

R Code Worklow

1. Edit an R file
2. Press Ctrl/Cmd-Shift L (*devtools::load_all*)
3. Explore the code in the console
4. Rinse an repeat

devtools::load_all()



DESCRIPTION File

- Adding important METADATA about your package
- `Devtools::create()` automatically adds a description file

```
Package: mypackage
Title: What The Package Does (one line, title case required)
Version: 0.1
Authors@R: person("First", "Last", email = "first.last@example.com",
                  role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: What license is it under?
LazyData: true
```

DESCRIPTION: Dependencies

- List the packages necessary for your package to work
- `use_package("packagename")` for selecting the packages you need

```
Imports:  
  dplyr,  
  ggvis
```

- > imports must be present for your package to work
- > when your package is installed, those packages will be installed

DESCRIPTION: Dependencies

- Your package can take advantage of other packages
- `use_package("packagename", "Suggests")`

```
Suggests:  
  dplyr,  
  ggvis,
```

-> your package can use these packages, but doesn't require them

DESCRIPTION: Dependencies

- If you need a particular version of a package, specify it after the package name:

```
Imports:  
  ggvis (>= 0.2),  
  dplyr (>= 0.3.0.1)  
Suggests:  
  MASS (>= 7.3.0)
```

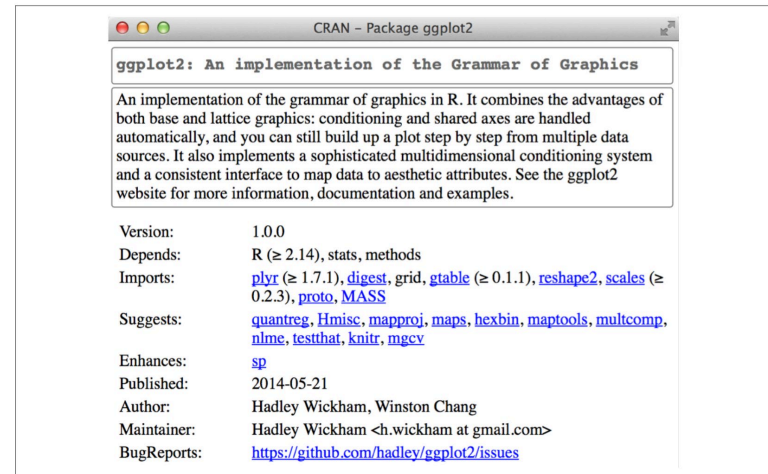
DESCRIPTION: Title and Description

- Important, especially if you plan to release your package on CRAN

The Title and Description for ggplot2 are as follows:

Title: An implementation of the Grammar of Graphics

Description: An implementation of the grammar of graphics in R. It combines the advantages of both base and lattice graphics: conditioning and shared axes are handled automatically, and you can still build up a plot step by step from multiple data sources. It also implements a sophisticated multidimensional conditioning system and a consistent interface to map data to aesthetic attributes. See the ggplot2 website for more information, documentation and examples.



CRAN - Package ggplot2

ggplot2: An implementation of the Grammar of Graphics

An implementation of the grammar of graphics in R. It combines the advantages of both base and lattice graphics: conditioning and shared axes are handled automatically, and you can still build up a plot step by step from multiple data sources. It also implements a sophisticated multidimensional conditioning system and a consistent interface to map data to aesthetic attributes. See the ggplot2 website for more information, documentation and examples.

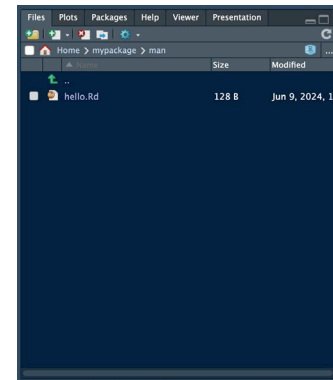
Version: 1.0.0
Depends: R (≥ 2.14), stats, methods
Imports: [plyr](#) (≥ 1.7.1), [digest](#), grid, [table](#) (≥ 0.1.1), [reshape2](#), [scales](#) (≥ 0.2.3), [proto](#), [MASS](#)
Suggests: [quantreg](#), [Hmisc](#), [mapproj](#), [maps](#), [hexbin](#), [maptools](#), [multcomp](#), [nlme](#), [testthat](#), [knitr](#), [mgcv](#)
Enhances: [sp](#)
Published: 2014-05-21
Author: Hadley Wickham, Winston Chang
Maintainer: Hadley Wickham <h.wickham at gmail.com>
BugReports: <https://github.com/hadley/ggplot2/issues>

DESCRIPTION: other components

- Author
- License
- Version
- Collate
- LazyData

Directory man/

- Important for your **documentation**
- Works like a dictionary -> *help()*
- **roxygen2** is used for writing



```
hello.R x hello.Rd x
Save current document (§S)
review on Save ABC Preview
1 \name{hello}
2 \alias{hello}
3 \title{Hello, World!}
4 \usage{
5 hello()
6 }
7 \description{
8 Prints 'Hello, world!'.
9 }
10 \examples{
11 hello()
12 }
13
```

Documentation Workflow

1. Add roxygen comments to your .R files
(add template: "Code" -> "Insert Roxygen Skeleton")
2. Run `devtools::document()` to convert comments to .Rd files (Ctrl+Shift+D)
3. Preview documentation with `?myFunction`
4. Rinse and repeat until the documentation looks the way you want

Roxygen comments

- They start with `#'`
- First paragraph = title
- Second paragraph = description

1

Roxygen comments in script

```
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#' @return The sum of x and y.
#' @examples
#' add(1, 1)
#' add(10, 1)
add <- function(x, y) {
  x + y
}
```

2

Generated .Rd file in man/

```
% Generated by roxygen2 (4.0.0): do not edit by hand
\name{add}
\alias{add}
\title{Add together two numbers}
\usage{
  add(x, y)
}
\arguments{
  \item{x}{A number}

  \item{y}{A number}
}
\value{
  The sum of x and y
}
\description{
  Add together two numbers
}
\examples{
  add(1, 1)
  add(10, 1)
}
```

3

help()

```
add (rvest) R Documentation

Add together two numbers

Description
  Add together two numbers

Usage
  add(x, y)

Arguments
  x A number
  y A number

Value
  The sum of x and y

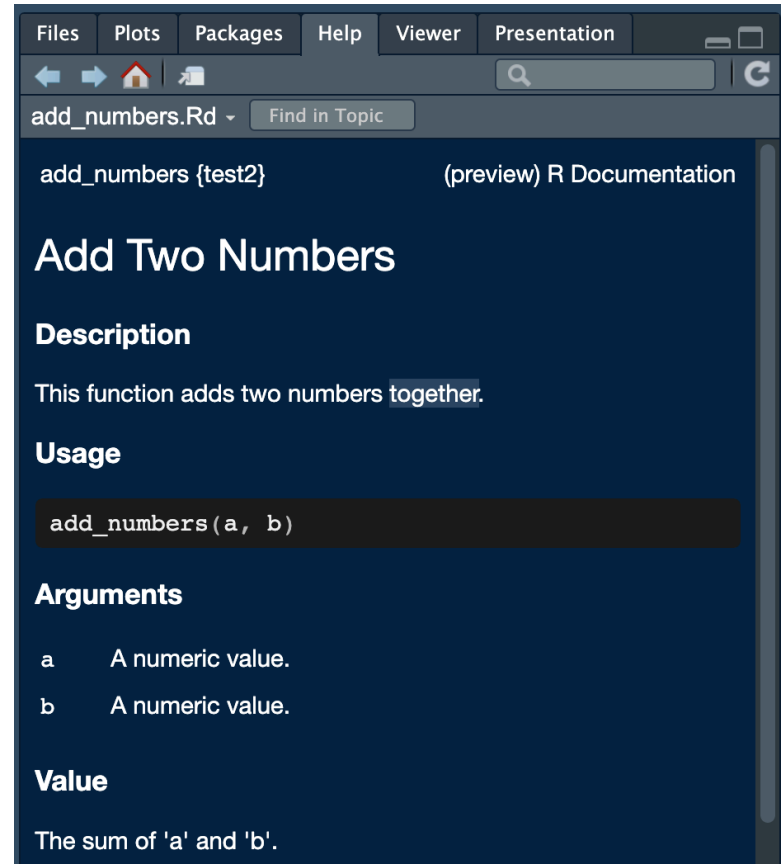
Examples
  add(1, 1)
  add(10, 1)
```

Documenting functions

- **@param:**
describes the function's inputs or parameters
-> start with a capital letter end with full stop
-> separate names with commas
-> *@param x,y Numeric vectors*
- **@return:**
describes the output from the function
- **@examples:**
shows how to use the function in practice
- **@imports:**
imports all functions from external package (use carefully)
- **@importFrom:**
imports function from package
- **@export:**
specifies, if function should be accessible for package user
- Type ``@`` and there will be many more options!

Exercise

- Create your own package
- Add function that adds two numbers
- Create Helpfile (see picture)



Automated Testing

Workflow until now:

- 1) Write a function
- 2) Load the function in your package: `devtools::load_all()` or
- 3) experiment with function in console

Ctrl + Shift + L

→ works fine for very simple packages with very few functions

But often, you have many functions, dependencies on helper functions etc.



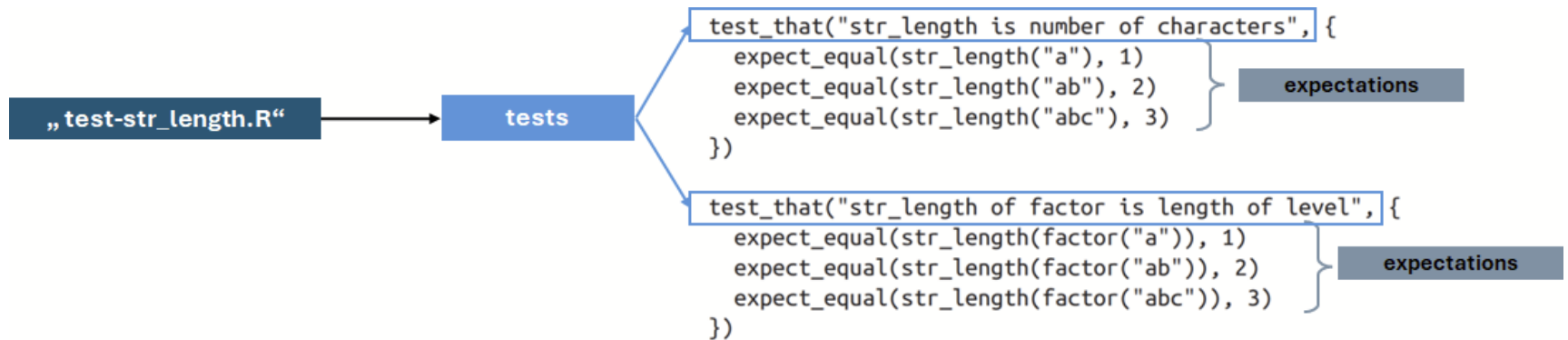
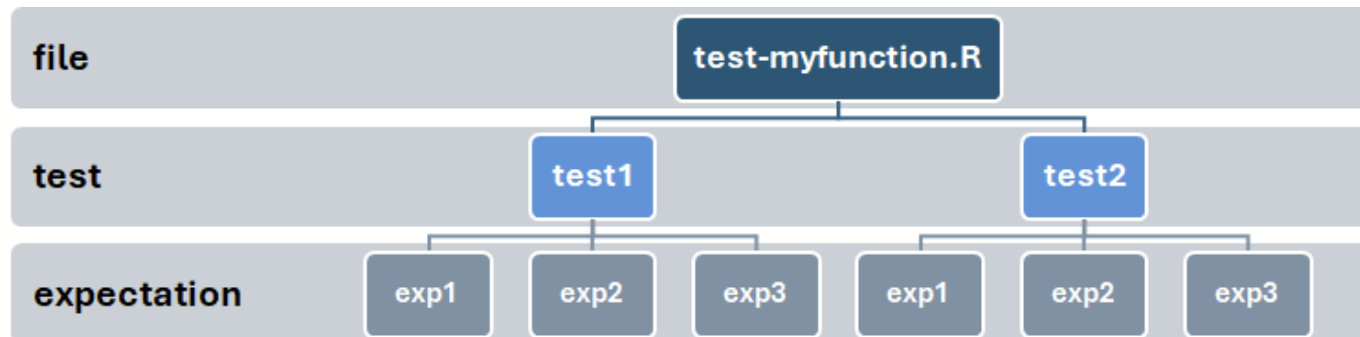
Better: Automated testing (Unit testing)

Advantages:

- Fewer bugs
- Better code structure
- Easier restarts (easier top ck up where you left)
- Robust code

saves time & trouble

Automatic Testing: Hierarchy of Organisation



Tests – How to (I)

Step 1: First-time setup: `devtools::use_testthat()`

- automatically adds `tests/testthat` directory → This is where your test-files will be
- automatically adds `testthat` to the **Suggests** field in the **DESCRIPTION**.

Step 2: Create a test file for one of your scripts:

- Type `usethis::use_test("functionName")` in the command line
 - Creates **“test-functionName.R”**
- or type `usethis::use_test()` to create a test file for the currently used tab
 - Creates **“test-ScriptName.R”**

Try it yourself!

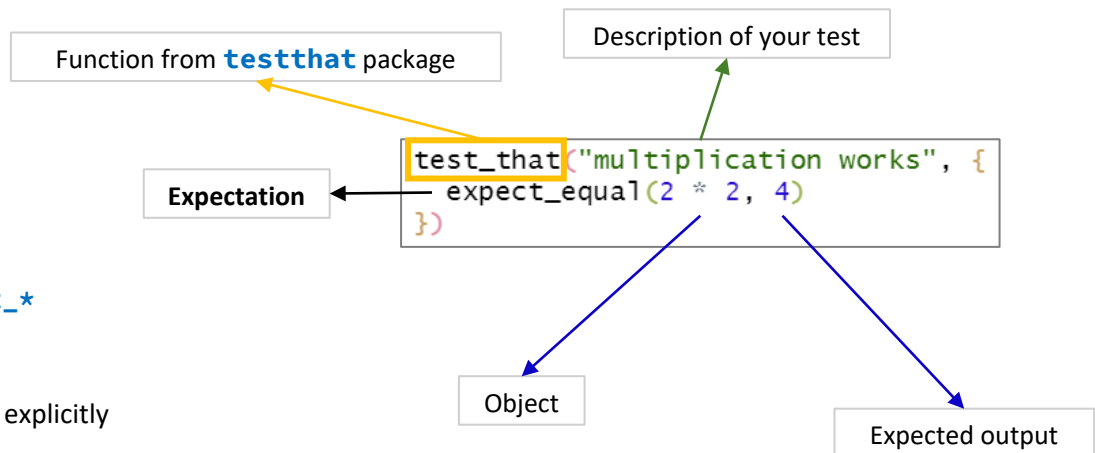
- 1) Create new script called “minmax” in your package.
- 2) Insert this function, which
 - computes the min and max of an input value or input vector.
 - Gives the output should be a named vector, which contains `c(Min, Max)`

```
minmax <- function(x) {  
  c(Min = min(x), Max = max(x))  
}
```

- 3) Initiate unit testing and create a test file for your “minmax” file.
 - Check, if you find the new test directory
 - Check if **testthat** is now listed under **Suggest** in your *DESCRIPTION*

Tests – How to (II)

What you see now: An automatically created a default `testthat` function in `test/testthat`



- `testthat` functions usually start with `expect_*`
- Many functions available!
- For a better overview: load `testthat` package explicitly

Step 3: `library(testthat)`

- Then type `?expect` in the console and have a look at the many options for expectations there are!

Expectations

Objects

<code>expect_equal()</code> <code>expect_identical()</code>	Does code return the expected value?
<code>expect_type()</code> <code>expect_s3_class()</code> <code>expect_s4_class()</code>	Does code return an object inheriting from the expected base type, S3 class, or S4 class?

Vectors

<code>expect_length()</code>	Does code return a vector with the specified length?
<code>expect_lt()</code> <code>expect_lte()</code> <code>expect_gt()</code> <code>expect_gte()</code>	Does code return a number greater/less than the expected value?
<code>expect_named()</code>	Does code return a vector with (given) names?
<code>expect_setequal()</code> <code>expect_mapequal()</code> <code>expect_contains()</code> <code>expect_in()</code>	Does code return a vector containing the expected values?
<code>expect_true()</code> <code>expect_false()</code>	Does code return TRUE or FALSE?
<code>expect_vector()</code>	Does code return a vector with the expected size and/or prototype?

Sideeffects

<code>expect_error()</code> <code>expect_warning()</code> <code>expect_message()</code> <code>expect_condition()</code>	Does code throw an error, warning, message, or other condition?
<code>expect_no_error()</code> <code>expect_no_warning()</code> <code>expect_no_message()</code> <code>expect_no_condition()</code>	Does code run without error, warning, message, or other condition?
<code>expect_invisible()</code> <code>expect_visible()</code>	Does code return a visible or invisible object?
<code>expect_output()</code>	Does code print output to the console?
<code>expect_silent()</code>	Does code execute silently?
<code>local_test_context()</code> <code>local_reproducible_output()</code>	Locally set options for maximal test reproducibility

Exercise 2

1. Change the `test_that` function of the **"test-minmax.R"** file:
 - Add a description
 - Change the *expectation* to test, if the output of "minmax" is a vector of length 2
 - Run the test
2. Create a test-file for the **add_numbers** function
 - Make a test, which expects that the output is "NA" if, one of the arguments passed to the function is NA
 - Run the test
(and don't panic if the test fails 😊 We will discuss this afterwards.)

Sidenote: $NA \neq NA$?

The test you wrote in Exercise 2 probably came back failed:

```
✓ | F W S OK | Context
✗ | 1      0 | add_nums

Failure (test-add_nums.R:8:5): Output NA if Input NA
add_numbers(6, NA) (`actual`) not equal to NA (`expected`).

`actual` is a double vector (NA)
`expected` is a logical vector (NA)
```

Wait...what?

Solution:

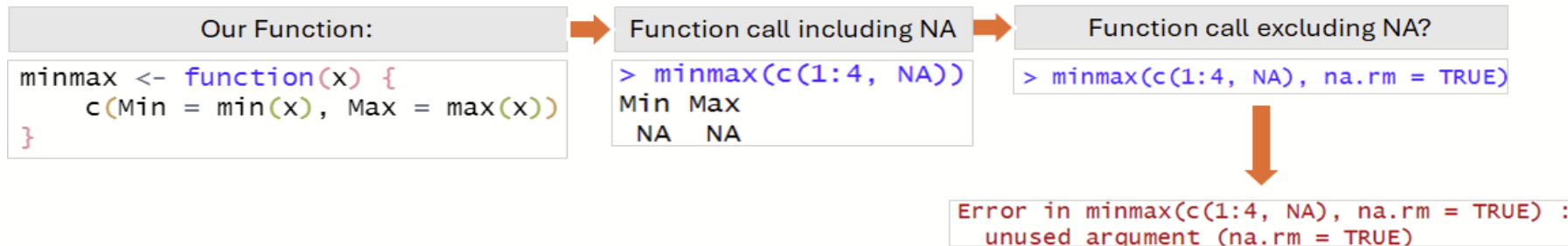
```
test_that("Output NA if Input NA", {
  expect_equal(add_numbers(6, NA), NA_integer_)
})
```

If not specified otherwise, NA is handled as logical vector. To match the output of `add_numbers`, it must be converted into a numeric vector

→ Use `NA_*_` to transform `NA` to the correct vector type

Excursus: Pass additional arguments to functions

NAs are annoying...let's just remove them when we call the function!



→ Enable to use of **additional arguments** using **...**



Skip Tests

Useful e.g. ...

- when an important file is missing
- when some parts of the functionality dependent on operating system (and therefore Tests are not necessary)
- for long-running Tests
- for known breakages
- experimental Features

```
skip(message = "Skipping")

skip_if_not(condition, message = NULL)

skip_if(condition, message = NULL)

skip_if_not_installed(pkg, minimum_version = NULL)

skip_if_offline(host = "captive.apple.com")

skip_on_cran()

skip_on_os(os, arch = NULL)

skip_on_ci()

skip_on_covr()

skip_on_bioc()

skip_if_translated(msgid = "'%s' not found")
```

Namespace

- „context for looking up the value of an object associated with a name“
- `search()` → List of all packages that are **attached**

```
> search()
[1] ".GlobalEnv"          "package:flopo"      "devtools_shims"
[4] "package:showtext"   "package:showtextdb" "package:sysfonts"
[7] "package:testthat"   "package:ggplot2"    "package:usethis"
[10] "tools:rstudio"      "package:stats"      "package:graphics"
[13] "package:grDevices"  "package:utils"      "package:datasets"
[16] "package:methods"    "AutoLoads"          "package:base"
```

- **Attaching package:**
 - In data analysis scripts: `library()` or `require()`
 - Inside a package; better option: `requireNamespace(x, quietly = TRUE)`
- **Loading package:**
 - Package is available, but not on `searchPath`
 - `::` needed to access functions from package
 - Other options to explicitly load package (rarely used):
 - `loadNamespace()`

Namespace (II)

Packages listed in description file

- **Imports / Depends:** makes sure packages is installed
- **Depends:** loads package
- **Imports:** attaches package (--> recommended)

```
Imports:
  dplyr,
  ggplot2,
  ggstatsplot,
  showtext
Suggests:
  extrafont,
  ggimage,
  grid,
  png,
  testthat (>= 3.0.0)
```

10 namespace directives

import		export	
<code>import()</code>	Import all functions from a package	<code>export()</code>	Export functions (including S3 and S4 generics)
<code>importFrom()</code>	Import selected functions (including S4 generics)	<code>exportPattern()</code>	Export all functions that match a pattern
<code>importClassesFrom()</code> and <code>importMethodsFrom()</code>	Import S4 classes and methods	<code>exportClasses()</code> and <code>exportMethods()</code>	Export S4 classes and methods
<code>useDynLib()</code>	Import a function from C	<code>S3method()</code>	Export S3 methods

In NAMESPACE, „Each directive describes an R object, and says whether it’s exported from this package to be used by others, or if it’s imported from another package to be used locally

Best way to generate NAMESPACE file: roxygen2
(before each function)

```
#' @importFrom ggstatsplot ggbetweenstats
#' @export
```

Namespace (III)

If your NAMESPACE shows:

```
exportPattern("^[[[:alpha:]]+")
```

...and you want to use roxygen comments to define your *NAMESPACE*

1) delete the NAMESPACE file manually.

2) use `devtools::document` or **Ctrl + Shift + L**
to update the Roxygen comments → this will create a new
NAMESPACE file based on your roxygen comments

Demonstration

My R package

- Functions
- Custom Theme
- Color Palette



Reference & Sources

- Wickham, H. (2015). R packages. O'Reilly Media, Inc.
 - (! Some functions described are now in other packages
→ `usethis::use_testthat()` instead of `devtools::use_testthat()`
newer Version:
- <https://r-pkgs.org/>
- <https://cran.r-project.org/web/packages/roxygen2/>
- <https://docs.posit.co/ide/user/ide/guide/pkg-devel/writing-packages.html>

Videos

- https://www.youtube.com/watch?v=KbwYdRbmgBY&list=PL4ZUIAlk7Qic9a6aBIMcRs7_CLblzCaIW&index=7 (Automated testing & examples)

ToDos (@me)

- Include Slide with references/sources (Book, videos etc at the end)
- Write solutions to exercises (remember Which Shortcuts = which actual function! (Write list)
- One more exercise? --> how much time?
- Quickly Present my package --> make notes, make last check
- Notes for presentation

How to push in your github?

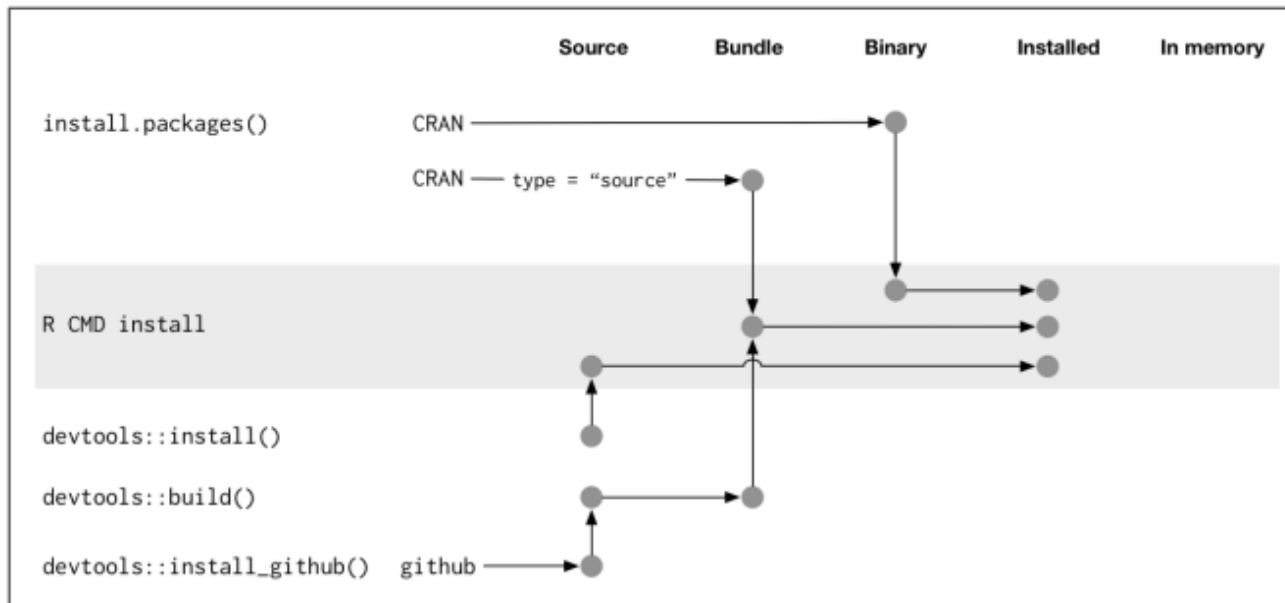


Figure 2-8. Five ways to install a package

- To push your R package to GitHub and manage it effectively, you can use the **devtools** and **usethis** packages. Here's a step-by-step guide on how to set up, develop, and push your R package to GitHub:
- **Install devtools and usethis:**
- `install.packages(c("devtools", "usethis"))`
- **Create a New Package:** If you haven't already created your package structure, you can create it using **usethis**:
- `library(usethis)`
- `create_package("path/to/your/package")`

- Step 2: Initialize Git and Create GitHub Repository
- **Navigate to Your Package Directory:**
`setwd("path/to/your/package")`
- **Initialize Git Repository:**
`usethis::use_git()`
- **Create GitHub Repository:**
`usethis::use_github()`

- **Add Your Function**
- `devtools::document()`
- `devtools::build()`
- `devtools::check()`
- `git add .`
- `git commit -m "Initial commit with package structure and unicorn function"`