

Advanced R Programming - Lecture 3

Krzysztof Bartoszek, Shashi Nagarajan
includes Agustín Valencia's and Marcos Mourao's slides
(slides based on Leif Jonsson's and Måns Magnusson's)

Linköping University

krzysztof.bartoszek@liu.se

8 IX 2025 (U2)

Today

Scientific Computing: Best Practices

R packages

Git and GitHub

Creating R packages

Documentation with ROxygen

Unit testing with testthat

R-Studio debugger

Questions since last time?

Scientific Computing: Best Practices

Based on the article referred to on course page...

REMEMBER:

1. Nothing replaces thinking and predicting.
2. Crystal balls do not exist.
3. Best practices do not work if used without thinking.

Write code for people

1. Write code for people

- 1.1 A program should not require its readers to hold more than a handful of facts in memory at once
- 1.2 Make names consistent, distinctive, and meaningful (Hungarian notation)
- 1.3 Make code style and formatting consistent
 - ▶ camelCase
 - ▶ PascalCase
 - ▶ snake_case (Python: lower_case_with_underscores)
 - ▶ kebab-case
 - ▶ *avoid* dot.case

Let the computer do the work

2. Let the computer do the work
 - 2.1 Make the computer repeat tasks
 - 2.2 Save recent commands in a file for re-use (use `.Rhistory` file)
 - 2.3 Use a build tool to automate workflows

Make incremental changes

3. Make incremental changes

- 3.1 Work in small steps with frequent feedback and course correction
- 3.2 Use a version control system
- 3.3 Put everything that has been created manually in version control

Do not repeat yourself (or others)

4. Do not repeat yourself (or others)
 - 4.1 Every piece of data must have a single authoritative representation in the system
 - 4.2 Modularize code rather than copying and pasting
 - 4.3 Re-use code instead of rewriting it

Plan for mistakes

5. Plan for mistakes

5.1 Add assertions to programs to check their operation

5.2 Use an off-the-shelf unit testing library

5.3 Turn bugs into test cases

5.4 Use a symbolic debugger

5.5 **ALWAYS CHECK CORRECTNESS OF INPUT!**

(also on exam ...)

Optimize software only after it works correctly

6. Optimize software only after it works correctly
 - 6.1 Use a profiler to identify bottlenecks
 - 6.2 Write code in the highest-level language possible

But prepare code for optimal algorithm ...

Document design and purpose, not mechanics

- 7. Document design and purpose, not mechanics
 - 7.1 Document interfaces and reasons, not implementations
(but make sure that you are able to understand implementation)
 - 7.2 Refactor code in preference to explaining how it works
 - 7.3 Embed the documentation for a piece of software in that software

Collaborate

8. Collaborate

- 8.1 Use pre-merge code reviews
- 8.2 Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems
- 8.3 Use an issue tracking tool

R packages

An environment with functions and/or data

The way to share code and data

4 000 developers (date?)

≈ 11100 packages (as of 19 July 2017)

nearly 12500 packages (as of 4 May 2018)

18428 packages (as of 15 August 2022) according to CRAN

```
available.packages(available_packages_filters = c("CRAN"))
```

Package basics

Usage

```
library()  
::  
:::
```

Installation

```
install.packages()  
devtools::install_github()  
devtools::install_local()
```

Package namespace

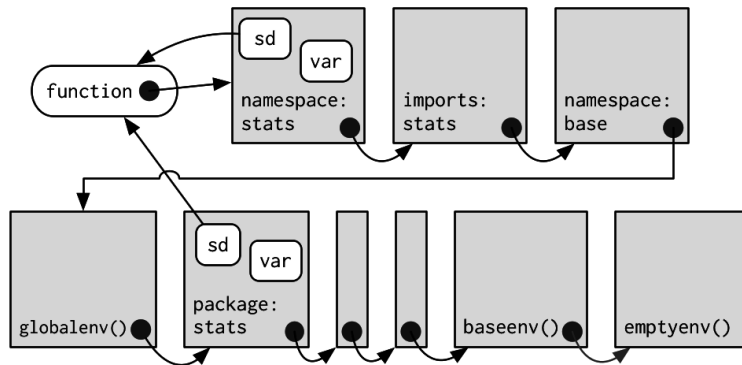


Figure: Package namespace (H. Wickham p.136)

Package namespace

`library()`: **ATTACHES** a package, its functions are available in the search path

`requireNamespace()`: **LOADS** package's code, data, methods, etc., runs `onLoad()`. Package is available in memory but not in search path, package **not** attached, access its components by `::`

NEVER use `library()`, `require()` inside your package, CRAN forbids it

H. Wickham, R packages (p. 82–84)

Which are good packages

Examine the package

1. Who?
2. When updated?
3. In development?

Why Version Control?

When working alone, we just might manage our work the way we each prefer. But things can get hairy when we must collaborate:

1. Imagine you and a colleague writing different modules of a software. You changed something this morning that your colleague was expecting to remain unchanged. 🍷
2. An important stakeholder doesn't link a recent change and wants your team to rollback. Have you stored previous versions? Can you retrieve them easily?
3. Your main server just crashed. Do you have backup?
4. Version Control is ubiquitous. IOW, your **recruiter** probably wants you to have Version Control skills.

Version Control: What is it exactly?

Hear from a professional!

<https://vimeo.com/41027679>

Why Git?

1. Simple to use
2. Distributed
 - ▶ No single copy of the work product (it's distributed!)
 - ▶ Each collaborator has all the files and change history
3. Fast
4. Common in practice
5. R packages uses GitHub
6. Integrated with R-Studio

Why Git?

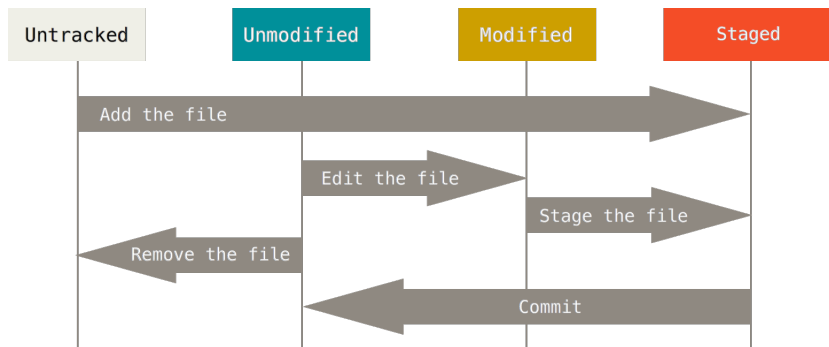
1. Simple to use
2. Distributed
 - ▶ No single copy of the work product (it's distributed!)
 - ▶ Each collaborator has all the files and change history
3. Fast
4. Common in practice
5. R packages uses GitHub
6. Integrated with R-Studio

Created by Linus Torvalds! ;)

Git: Getting Started & Basics

The Lifecycle of Files on Git

Source: Pro Git, a book by Chacon and Straub



Chapters 1 & 2 of the book are highly recommended if you are new to Git!

Git: Getting Started & Basics

1. A repository (repo) is basically where the files of your project will reside
2. Any file that have only just been created will, by default, be 'untracked'
3. To track a file (which you must do if you want to share it, for instance, with your collaborators), 'add' it to the repo
4. 'Commit' an added file to save it in the repo; when a file is committed, its state will change to 'unmodified'
5. To edit an 'unmodified' file, you (collaborators) can access a copy of the file and perform edits; this will change the file's state to 'modified'
6. To save your edits, stage and commit the file again

GitHub

1. Local (commit)
2. Remote (push/pull)
3. Barebone homepage (using md)
4. Collaborations
5. Issue tracker / Wiki / discussions

Public and private repos

2FA: time-based one-time password,
open algorithm: value based on initial key and current time
R package **otp**

Merge conflicts

1. Competing commits from different contributors.
2. Git cannot resolve this by itself
3. Manually resolve.
4. Supporting merge tools.
 - 4.1 p4merge
presentation by Agustín Valencia and Marcos Mourao
 - 4.2 sublime merge
<https://www.sublimemerge.com/>

Setting p4merge as diff/merge tool for git

Agustín Valencia

Marcos Mourao

Use this presentation under GPLv3 License

The merge issue

About merge conflicts

Merge conflicts happen when you merge branches that have competing commits, and Git needs your help to decide which changes to incorporate in the final merge.

Source : <https://help.github.com/en/articles/about-merge-conflicts>

Git way to solve merge conflicts

- Open the conflicting file in a text editor.
- Conflict markers:

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed.
<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=====
Git makes conflict resolution easy.
>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

- Manually delete the conflict markers and choose which version (or both) to maintain in your file.

Source: <https://git-scm.com/docs/git-merge>

```
let's create a conflict
Hey this is Agustin's second line
Agustin : Edit this line please <---
Thank you
```

```
let's create a conflict
Hey this is Agustin's second line
Agustin : Better no, I changed my mind
Thank you
```

Agustín: commit

```
let's create a conflict
Hey this is Agustin's second line
Marcos: I deleted your line and replaced it with a better one ;) <---
Thank you
```


Marcos: commit & push

Try to push **CONFLICT !**

Trying to push an outdated file

```
[158] agustinvalencia : ~/Documents/732A94 - Advanced R Programming/Repo
(09:34) -> git push
To https://github.com/agustinvalencia/732A94_Group_6_Work.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/agustinvalencia/7
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pus
hint: to the same ref. You may want to first integrate the remote change
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for detail

[159] agustinvalencia : ~/Documents/732A94 - Advanced R Programming/Repo
(09:35) -> git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/agustinvalencia/732A94_Group_6_Work
   8847c54..15216e5  master    -> origin/master
Auto-merging conflict_test
CONFLICT (content): Merge conflict in conflict_test
Automatic merge failed; fix conflicts and then commit the result.
```



Git diff

```
[(09:35) -> git diff
diff --cc conflict_test
index 394c7d3,01e91e3..0000000
--- a/conflict_test
+++ b/conflict_test
@@@ -1,7 -1,7 +1,11 @@@
    let's create a conflict
    Hey this is Agustin's second line

++<<<<<< HEAD
+Agustin : Better no, I changed my mind
+=====
+ Marcos: I deleted your line and replaced it with a better one ;) <-----
++>>>>>> 15216e5aa1bdc6a699f9400315f200f7e212574e

Thank you
```

How the file looks now

```
let's create a conflict
Hey this is Agustin's second line

<<<<<< HEAD
Agustin : Better no, I changed my mind
=====
Marcos: I deleted your line and replaced it with a better one ;) <-----
>>>>>> 15216e5aa1bdc6a699f9400315f200f7e212574e

Thank you
█
```


How we did it

```
[162] agustinvalencia : ~/
(09:42) -> git mergetool
```

conflict_test

1 diffs (ignore line ending differences) | Tab spacing: 4 | File format (Encoding: UTF-8 Line endings: Mac OS X)

Base: conflict_test_BASE_77670

Left: conflict_test_REMOTE_77670

Right: conflict_test_LOCAL_77670

Merge: conflict_test

Differences from base: 0

Differences from base: 0

Conflicts: 1

Repo_Collab/732A94_Group_6_Work/./conflict_test_REMOTE_77670

1 let's create a conflict

2 Hey this is Agustin's second line

3

4 Marcos: I deleted your line and replaced it wit

5

6 Thank you

7

8

1g/Repo_Collab/732A94_Group_6_Work/./conflict_test_BASE_77670

1 let's create a conflict

2 Hey this is Agustin's second line

3

4 Agustin : Edit this line please <---

5

6 Thank you

7

8

/Repo_Collab/732A94_Group_6_Work/./conflict_test_LOCAL_77670

1 let's create a conflict

2 Hey this is Agustin's second line

3

4 Agustin : Better no, I changed my mind

5

6 Thank you

7

8

conflict_test

let's create a conflict

Hey this is Agustin's second line

Agustin : Edit this line please <---

Marcos: I deleted your line and replaced it with a better one ;) <-----

Agustin : Better no, I changed my mind

Thank you

How to setup git to use p4merge

```
This is Git's per-user configuration file.
[user]
    name = agustinvalencia
# Please adapt and uncomment the following lines:
#     name = Agustin Valencia
#     email = valencia.ag@gmail.com

[merge]
    keepBackup = false
    tool = p4merge
[mergetool "p4merge"]
    cmd = /Applications/p4merge.app/Contents/Resources/launchp4merge
"\$PWD/\$BASE\" " \"\$PWD/\$REMOTE\" " \"\$PWD/\$LOCAL\" " \"\$PWD/\$MERGED\"
    keepTemporaries = false
    trustExitCode = false
    keepBackup = false
[diff]
    tool = p4merge
[difftool "p4merge"]
    cmd = /Applications/p4merge.app/Contents/Resources/launchp4merge
"\$REMOTE\" " \"\$LOCAL\" "
```

.gitconfig file

Source : <https://gist.github.com/SeanSSDing/d56d8b8aa4c79a05939b9ad3a6a63c8f>

Why part of the course?

Writing performant code (best practice)

The way to collaborate (R ecosystem)

Combine code, data and analysis

Easy to distribute and reuse (public api)

Learn how to reuse code from other packages

Package structure

Package structure

DESCRIPTION

DESCRIPTION: Imports, Suggests, Depends (LABS!!)

Your package will nearly always use functions from other packages!

Imports: These packages have to be present (or installed) when your package is installed. However, attaching your package `library(your package)` will **load** them (not attach). To use functions from them it is recommended to call them in your package as `packagename::function()`.

Suggests: Your package can use these functions, but does not require them, e.g. datasets, for tests, vignettes. Before using functions from them you need to check if they are available, `requireNamespace()`

Depends: Packages here will be **attached**. **NOT** recommended, heavy on the environment, CRAN has a limit on the number of packages in Depends. Can be used to specify version of R.

H. Wickham, R packages (p. 34–36, 84)

Package structure

DESCRIPTION

NAMESPACE

NAMESPACE (LABS!!)

What is used from other packages and provided to others!

`importFrom(package, function)`: package has to be listed in DESCRIPTION, do not `import(package)` (i.e. all) but only what you need. No need to call function as `package::function` now but **RECOMMENDED**

`export(function)`: what you make available for your users

`exportPattern(regular expression)`: make available functions with name matching a pattern, e.g. does not start with .

`S3method(method, class)`: export S3 methods

S4 classes, methods import and export **see book**

`useDynLib`: Import a function from C

H. Wickham, R packages (p. 84–90)

Imported functions (tips)

Do not call imported functions directly but through your own wrapper function.

1. Better control over what you pass: wrapper's interface.
2. Foreign package or function gets deprecated: change only in one place.

Use syntax `package::function()` for imported functions, as it makes code more clear and easier to update.

Package structure

DESCRIPTION

NAMESPACE

R/

Package structure

DESCRIPTION

NAMESPACE

R/

man/

Package structure

DESCRIPTION

NAMESPACE

R/

man/

vignette/

Package structure

DESCRIPTION

NAMESPACE

R/

man/

vignette/

tests/

Package structure

DESCRIPTION

NAMESPACE

R/

man/

vignette/

tests/

data/

Package structure

DESCRIPTION

NAMESPACE

R/

man/

vignette/

tests/

data/

scr/

Package structure

DESCRIPTION

NAMESPACE

R/

man/

vignette/

tests/

data/

scr/

inst/

Why roxygen2?

R requires that package documentation be stored in **.Rd** files in the `man/` folder. These files must conform to a number of specs, which may be tedious to accomplish manually.

Enter **roxygen2**: package elements can be documented in **.R** files (in the `R/` folder), and roxygen2 will generate the `man/` files.

Remember:

1. **roxygen2** is no free-lunch, but it does simplify documentation; importantly, it help manage the package NAMESPACE
2. Having docs near code can help with writing performant code
3. **roxygen2** is similar to JavaDoc & DOxygen, which are widely adopted standards for documenting Java & C++ packages
4. **roxygen2** needs Encoding: UTF-8 in DESCRIPTION

roxygen2: Workflow and Syntax

Standard Workflow:

1. Add comments to .R files (see Syntax below)
2. Run the command `devtools::document()`
3. Preview documents and repeat above steps if/as needed

Generic Syntax:

1. All roxygen2 comments must begin with the token `#!`
2. These must be placed **before** the function code
3. Most (if not all) such comments must be written out as tags; each tag must be in the `@tagnames details` format.
4. roxygen2 requires that specific elements of a function/dataset/class be documented by means of specific tags; see link below for details.

Source: <https://r-pkgs.org/man.html>

roxygen2: Example 1

<code>#' Function to make a Scatterplot using a couple of features of the \code{iris} dataset</code>	Title [Tag not necessary]
<code>#' Given a dataset like \code{iris}, the function \code{iris_plot} makes a scatterplot of #'\code{Sepal.Length} vs. \code{Petal.Width} by \code{Species}</code>	Function Description (Tag not necessary)
<code>#' @param data A dataset like the \code{iris} dataset. Must be a \code{data.frame} object #'\code{with continuous variables \code{Sepal.Length} and. \code{Petal.Width}, and a categorical #'\code{variable, \code{Species}}</code>	Function Parameter Description (Format @param <param name> <Description>)
<code>#' @return Scatterplot of \code{Sepal.Length} vs. \code{Petal.Width} by \code{Species}</code>	Function Output Description (Format @return <Description>)
<code>#' @export</code>	@export signals to roxygen2 that the function being documented must be exported to the package NAMESPACE
<code>#' @examples</code> <code>#' data(iris)</code> <code>#' iris_plot(iris)</code>	Examples of how to use the function (Format @examples <Code, possibly over multiple lines>)
<code>#' @import ggplot2</code>	@import <Package Name> signals to roxygen2 that a certain package must be imported; roxygen2 adds the package to NAMESPACE
<code>iris_plot <- function(data) { # get rid of 'no visible binding for global variable' note Sepal.Length <- Petal.Width <- Species <- NULL # code for plotting ggplot2::ggplot(data) + ggplot2::aes(x = Sepal.Length, y = Petal.Width, col = Species) + ggplot2::geom_point(alpha = 0.5) }</code>	

Figure: Including roxygen2 compatible documentation

Source Code: `./iris_course/R/iris_plot.R`

roxygen2: Example II

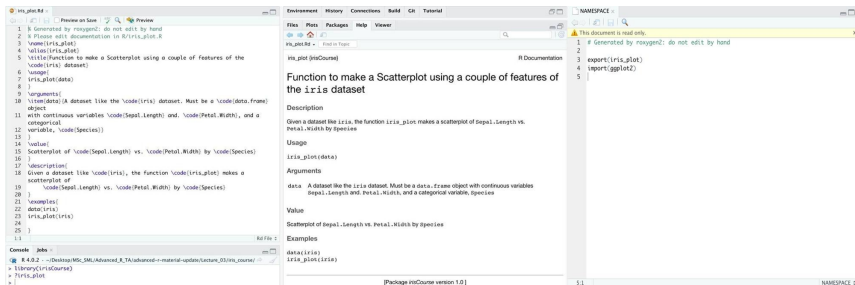


Figure: man and NAMESPACE files automatically updated by roxygen2!

Source Code: `./iris_course/R/iris_plot.R`

Why unit testing?

Fewer bugs

Better code structure

Faster restarts

Robust code - correct a bug only once

A must in complicated projects!

One unit testing framework for R integrated with R-Studio is
`testthat`

Types of testing

1. White box testing
2. Black box testing
3. Probabilistic testing

testthat: Workflow and Test File Structure

Workflow

1. Setup the test framework with `usethis::use_testthat()`
2. Write out tests in .R files (viz. test files) and save them in the `tests/testthat/` folder (created through step 1 above) with filename starting with 'test' (for e.g. `test_myfun.R`)
3. Run tests with `Ctrl/Cmd + Shift + T` or `devtools::test()`

Test File Structure

1. As with usual .R scripts, test files must have `library`, `data` or similar statements if needed for the tests
2. Test files typically have 'test groups', which in turn have 'expectations', the atomic units of the `testthat` framework
3. For those familiar, note that the `context` statement is now deprecated

Source: <https://r-pkgs.org/tests-basics.html>; review this link for syntax details

Test File Example

The screenshot displays the RStudio interface with a test file example. The editor shows the following R code:

```
1 data(iris)
2
3 test_that("Input must be a 'data.frame'", {
4   iris_matrix <- data.matrix(iris)
5   expect_error(iris_plot(iris_matrix))
6 })
```

Annotations in the image explain the code:

- File name starts with 'test'**: Points to the filename `test_iris_plot.R`.
- Include library and data statements as may be needed**: Points to `data(iris)`.
- Wrap tests around the 'test_that' construct**: Points to the `test_that` function.
- A description for your test**: Points to the string `"Input must be a 'data.frame'"`.
- Test result depends on the 'expect' function**: Points to `expect_error`.
- In this example, the call 'iris_plot(iris_matrix)' will return an error, because iris_matrix is not a 'data.frame', as expected, thus passing the test**: Points to the `iris_matrix` variable.

The console output shows the test results:

```
Environment History Connections Build Git Tutorial
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 1 ] Done!
Test complete
```

The Files pane shows the directory structure:

```
Files Plots Packages Help Viewer
ktop MSc_SML Advanced_R_TA advanced-r-material-update Lecture_03 iris_course tests
testthat
testthat.R
```

A note indicates: "Place test files (in this case 'test_iris_plot.R') inside the 'testthat' folder".

Source Code: `./iris_course/tests/test_that/test_iris_plot.R`

Introduction to Debugging in R

Another Video!!

Debugging in R

<https://vimeo.com/99375765>

REMEMBER
ALWAYS
CHECK INPUT!

The End... for today.
Questions?
See you next time!