Advanced R Programming - Lecture 3

Krzysztof Bartoszek
(slides based on Leif Jonsson's and Måns Magnusson's)

Linköping University

krzysztof.bartoszek@liu.se

10 September 2018 (U1)

Today

- Best practices for scientific computing
- R packages
- Git and GitHub
- Creating R packages
- Documentation with ROxygen
- Unit testing with testthat
- R-Studio debugger

Questions since last time?

Best practices for scientific computing

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

1. Write code for people

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

Dont repeat yourself (or others)

4. Dont 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

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

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?)
$\approx$ 11100 packages (as of 19 July 2017)

Package basics

Usage

`library()`

`::`

`:::`

Installation

`install.packages()`

`devtools::install_github()`

`devtools::install_local()`

Package namespace

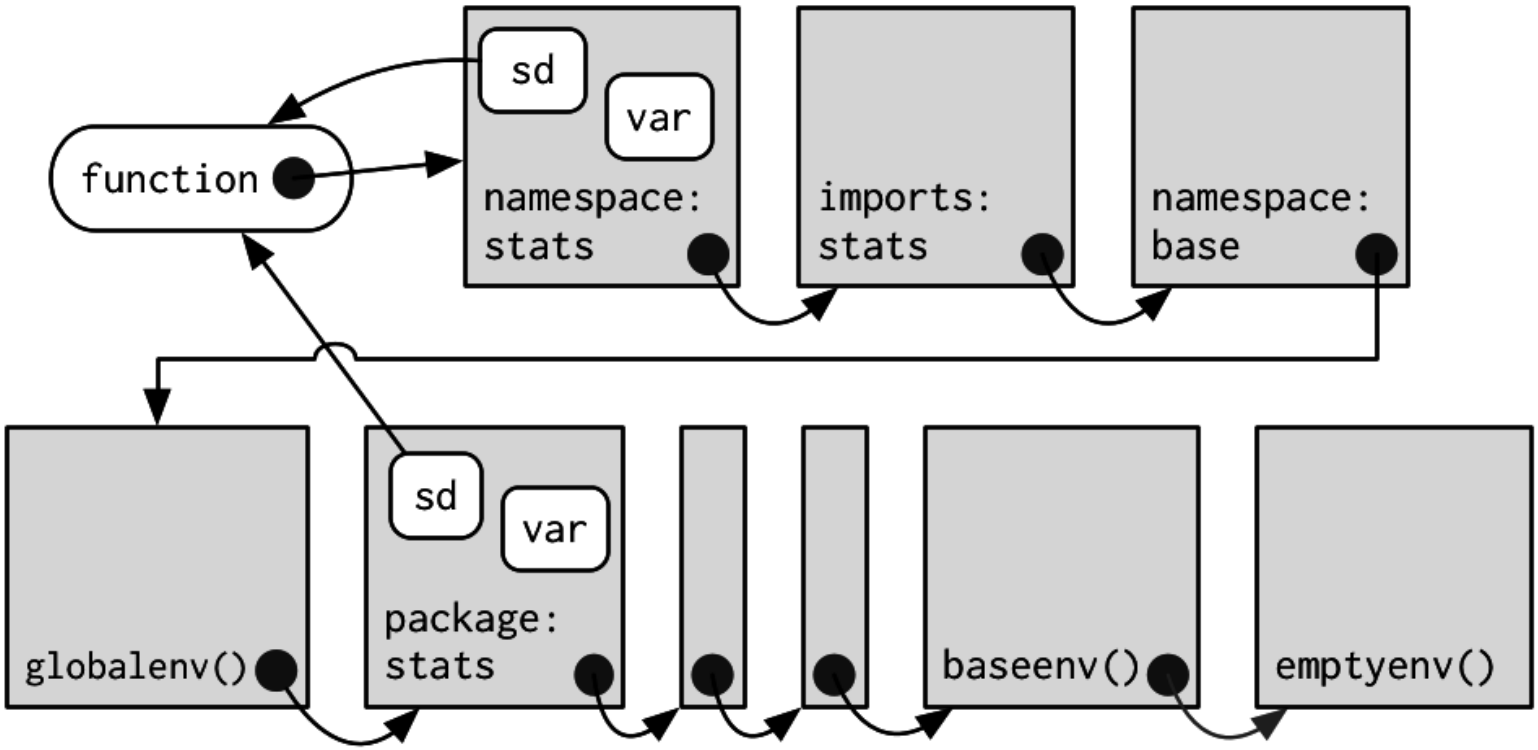


Figure: Package namespace (H. Wickham p.136)

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, acccess 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?

What is Version control?

Video!!

`https://vimeo.com/41027679`

Why version control?

1. Collaboration
2. Storing versions (properly)
3. Restoring versions
4. Understanding what happens
5. Backup

Why git?

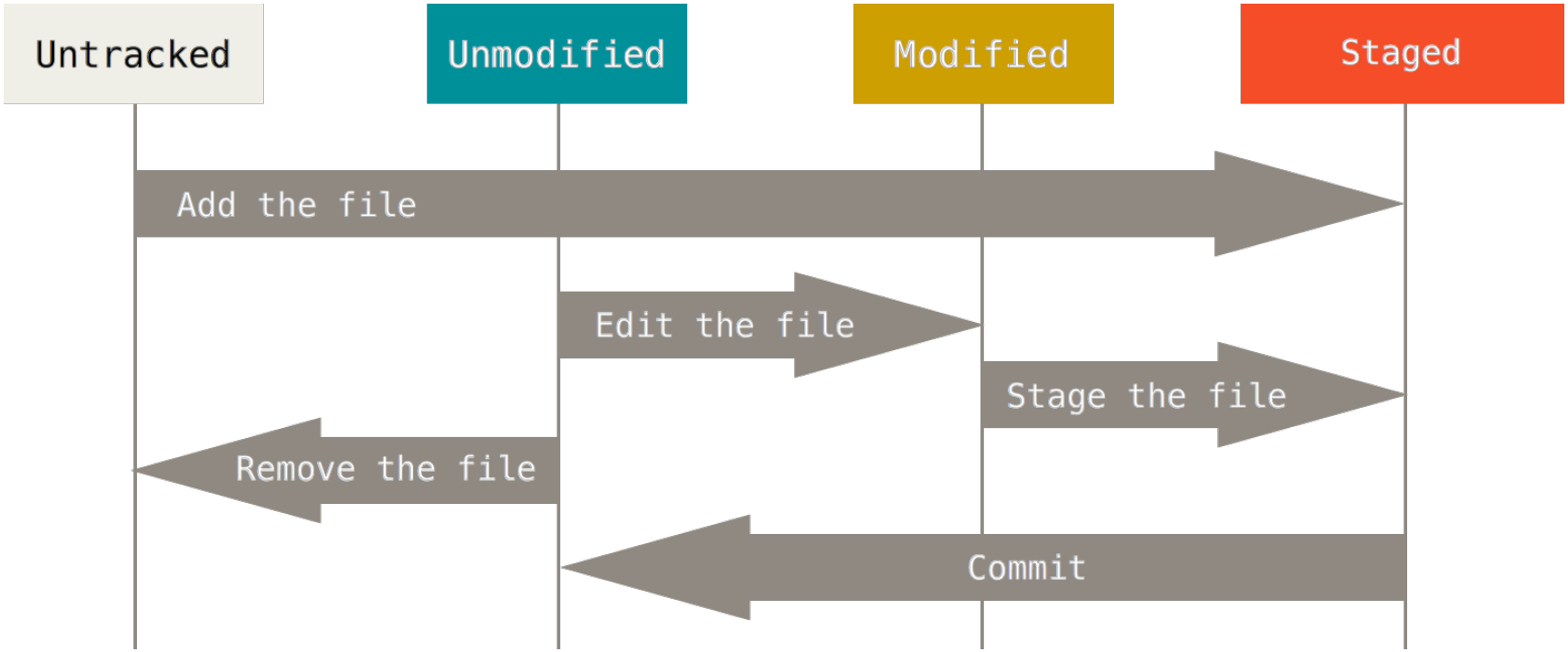
1. Simple to use
2. Distributed
3. Fast
4. Common in practice
5. R packages uses github
6. Integrated with R-Studio

Why git?

1. Simple to use
2. Distributed
3. Fast
4. Common in practice
5. R packages uses github
6. Integrated with R-Studio

Created by Linus Torvalds! ;)

Basic git



https://git-scm.com/book/id/v2/Git-Basics-Recording-Changes-to-the-Repository

GitHub

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

Free for public repos
Private repos cost
Student accounts

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)

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?

1. Performant code (docs close to code)
2. Automatically generates all man files
3. Simple to use
4. Handles NAMESPACE
5. Similar to JavaDoc and DOxygen

roxygen2 syntax

[example]

`https://github.com/rOpenGov/sweidnumbr`
sweidnumbr
Full support in R-Studio

Why unit testing?

Fewer bugs
Better code structure
Faster restarts
Robust code - correct a bug only once

A must in complicated projects!

Types of testing

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

testthat

Unit testing framework for R
Integrated with R-Studio

[example]

`https://github.com/rOpenGov/sweidnumbr/tree/master/tests`
sweidnumbr testsuite

Introduction to Debugging in R

Another Video!!
Debugging in R
`https://vimeo.com/99375765`

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