R Packages

Lecture 24

Dr. Colin Rundel

What are R packages?

R packages are just a collection of files (R code, compiled code, data, documentation, etc.) that live in your library path.

```
1 .libPaths()
```

```
[1] "/Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/library"
```

```
1 dir(.libPaths())
```

```
  [1] "_cache"            "abind"
  [3] "anytime"           "ape"
  [5] "arrayhelpers"      "arrow"
  [7] "AsioHeaders"       "askpass"
  [9] "assertthat"        "astsa"
 [11] "backports"         "base"
 [13] "base64enc"         "bayesplot"
 [15] "bench"             "BH"
 [17] "bit"               "bit64"
 [19] "blob"              "bonsai"
 [21] "bookdown"          "boot"
 [23] "brew"              "bridgesampling"
 [25] "brio"              "brms"
 [27] "Brobdingnag"       "broom"
 [29] "broom.helpers"     "broom.mixed"
 [31] "bsicons"           "bslib"
 [33] "cachem"            "callr"
```

Search path

When you run `library(pkg)` the functions (and objects) in the package's namespace are attached to the global search path.

```
1 search()
```

```
[1] ".GlobalEnv"        "package:stats"
[3] "package:graphics"  "package:grDevices"
[5] "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"
[9] "package:base"
```

```
1 library(diffmatchpatch)
```

```
1 search()
```

```
 [1] ".GlobalEnv"
 [2] "package:diffmatchpatch"
 [3] "package:stats"
 [4] "package:graphics"
 [5] "package:grDevices"
 [6] "package:utils"
 [7] "package:datasets"
 [8] "package:methods"
 [9] "Autoloads"
[10] "package:base"
```

Loading vs attaching

If you do not want to attach a package you can directly use package functions via `::` or load the package with `requireNamespace()`.

```
1 loadedNamespaces()
```

```
 [1] "compiler"       "fastmap"
 [3] "cli"            "graphics"
 [5] "diffmatchpatch" "tools"
 [7] "htmltools"      "rstudioapi"
 [9] "utils"          "yaml"
[11] "grDevices"      "Rcpp"
[13] "stats"          "datasets"
[15] "rmarkdown"      "knitr"
[17] "methods"        "jsonlite"
[19] "xfun"           "digest"
[21] "rlang"          "base"
[23] "evaluate"
```

```
1 requireNamespace("forcats")
2 loadedNamespaces()
```

```
 [1] "digest"         "methods"
 [3] "diffmatchpatch" "fastmap"
 [5] "xfun"           "magrittr"
 [7] "glue"           "knitr"
 [9] "htmltools"      "rmarkdown"
[11] "lifecycle"      "utils"
[13] "cli"            "graphics"
[15] "grDevices"      "stats"
[17] "compiler"       "forcats"
[19] "base"           "rstudioapi"
[21] "tools"          "evaluate"
[23] "Rcpp"           "yaml"
[25] "rlang"          "jsonlite"
[27] "datasets"
```

```r
search()
```

```
 [1] ".GlobalEnv"
 [2] "package:diffmatchpatch"
 [3] "package:stats"
 [4] "package:graphics"
 [5] "package:grDevices"
 [6] "package:utils"
 [7] "package:datasets"
 [8] "package:methods"
 [9] "Autoloads"
[10] "package:base"
```

Where do R packages come from?

We've already seen the two primary sources of R packages:

CRAN:

```
install.packages("diffmatchpatch")
```

GitHub:

```
remotes::install_github("rundel/diffmatchpatch")
```

there is one other method that comes up (particularly around package development), which is to install a package from local files.

Local install:

From the terminal,

```
R CMD install diffmatchpatch_0.1.0.tar.gz
```

or from R,

```
devtools::install("diffmatchpatch_0.1.0.tar.gz")
```

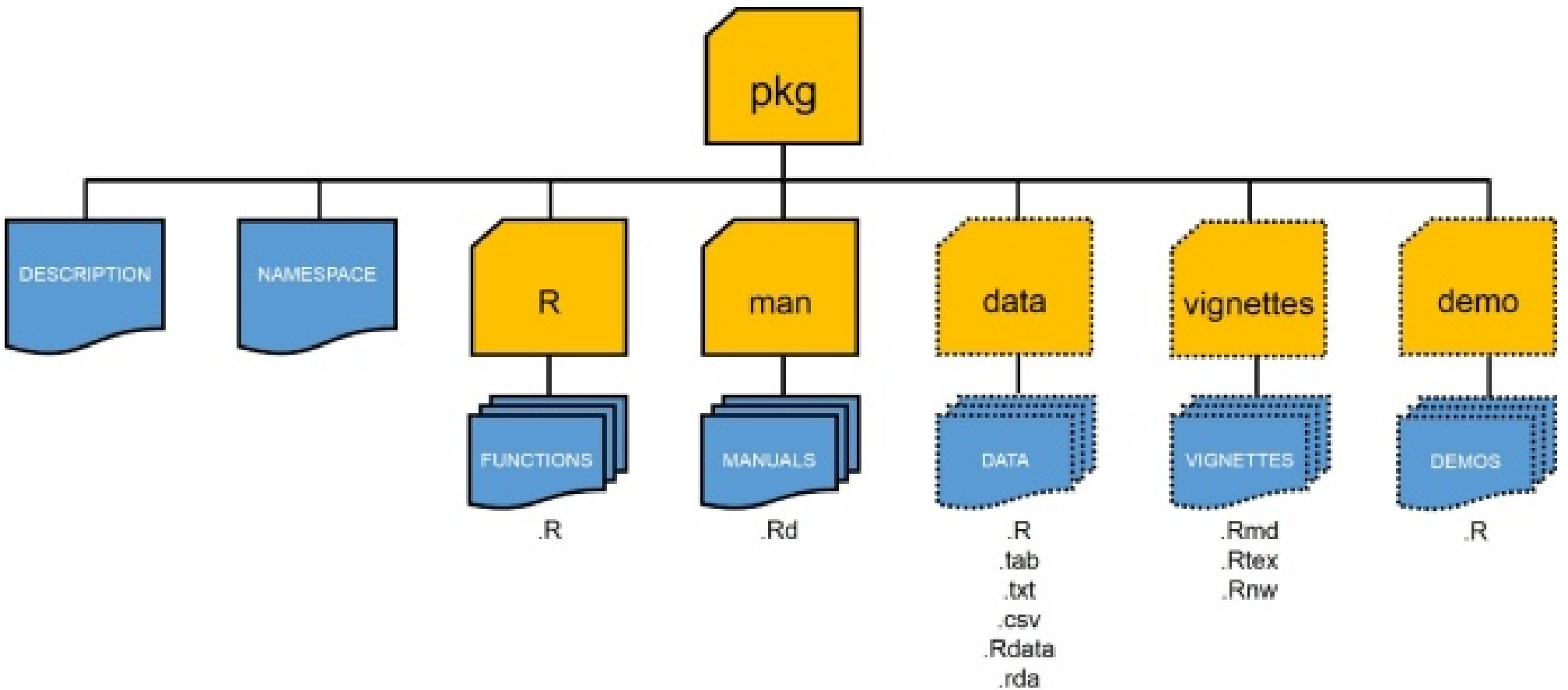
What is CRAN

The Comprehensive R Archive Network which is the central repository of R packages.

- Maintained by the R Foundation and run by a team of volunteers, ~20k packages
- Retains all current versions of released packages as well as archives of previous versions
- Similar in spirit to Perl's CPAN, TeX's CTAN, and Python's PyPI
- Some important features:
  - All submissions are reviewed by humans + automated checks
  - Strictly enforced submission policies and package requirements
  - All packages must be actively maintained and support upstream and downstream changes

See Writing R Extensions

Structure of an R Package



From A Quickstart Guide for Building Your First R Package

Core components

- `DESCRIPTION` - file containing package metadata (e.g. package name, description, version, license, and author details). Also specifies package dependencies,
- `NAMESPACE` - details which functions and objects are exported by your package
- `R/` - folder containing R script files (`.R`)
- `man/` - folder containing R documentation files (`.Rd`)

Optional components

The following components are optional, but quite common:

- `tests/` - folder contain unit tests
- `src/` - folder containing code to be compiled (usually C / C++)
- `data/` - folder containing example data sets
- `inst/` - files that will be copied to the package's top-level directory when it is installed (e.g. C/C++ headers, examples or data files that don't belong in `data/`)
- `vignettes/` - long form documentation, can be static (`.pdf` or `.html`) or literate documents (e.g. `.qmd`, `.Rmd` or `.Rnw`)

Package contents

Source Package

```
1 fs::dir_tree("~/Desktop/Projects/diffmatchpatch/
```

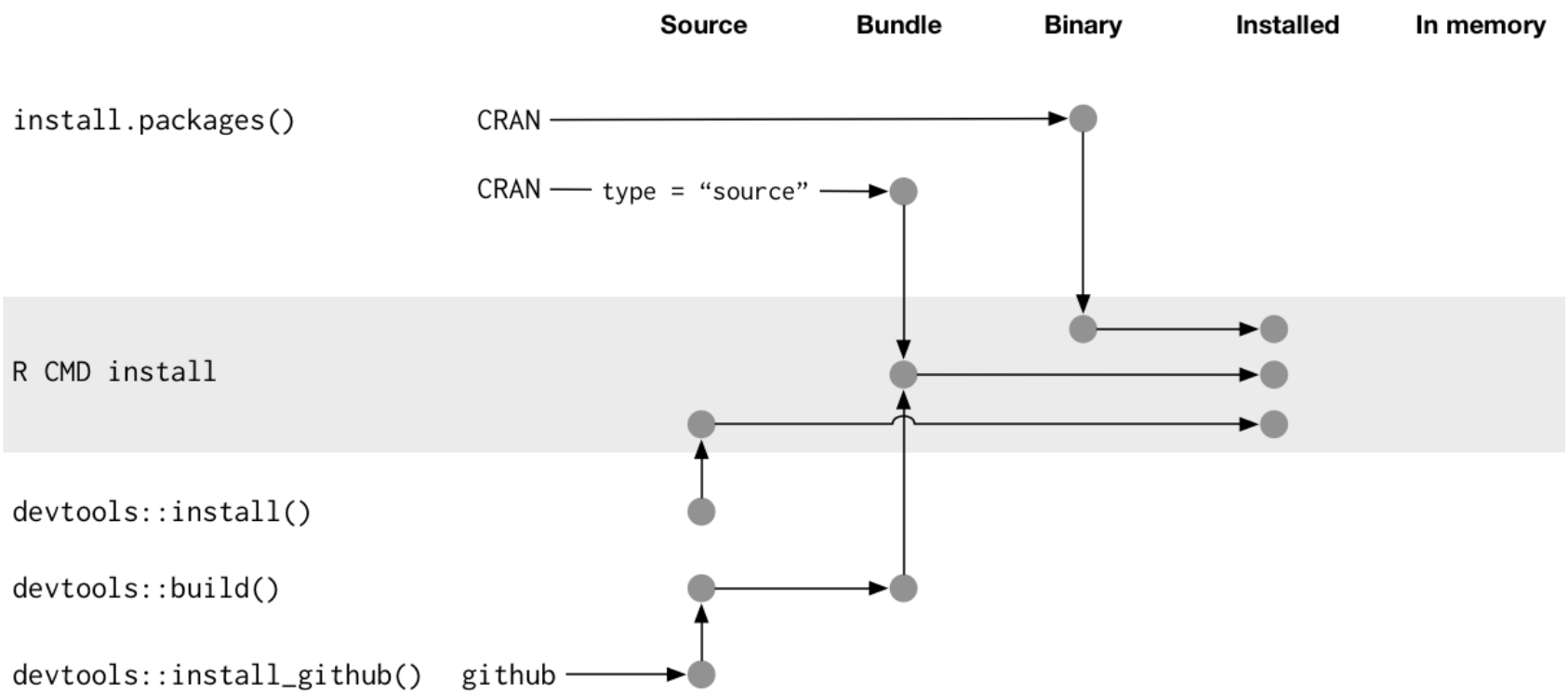
```
~/Desktop/Projects/diffmatchpatch/
├── DESCRIPTION
├── LICENSE.md
├── NAMESPACE
├── NEWS.md
├── R
│   ├── RcppExports.R
│   ├── diff.R
│   ├── diffmatchpatch-package.R
│   ├── match.R
│   ├── options.R
│   ├── patch.R
│   └── print.R
├── README.Rmd
├── README.md
├── cran-comments.md
├── diffmatchpatch.Rproj
```

Installed Package

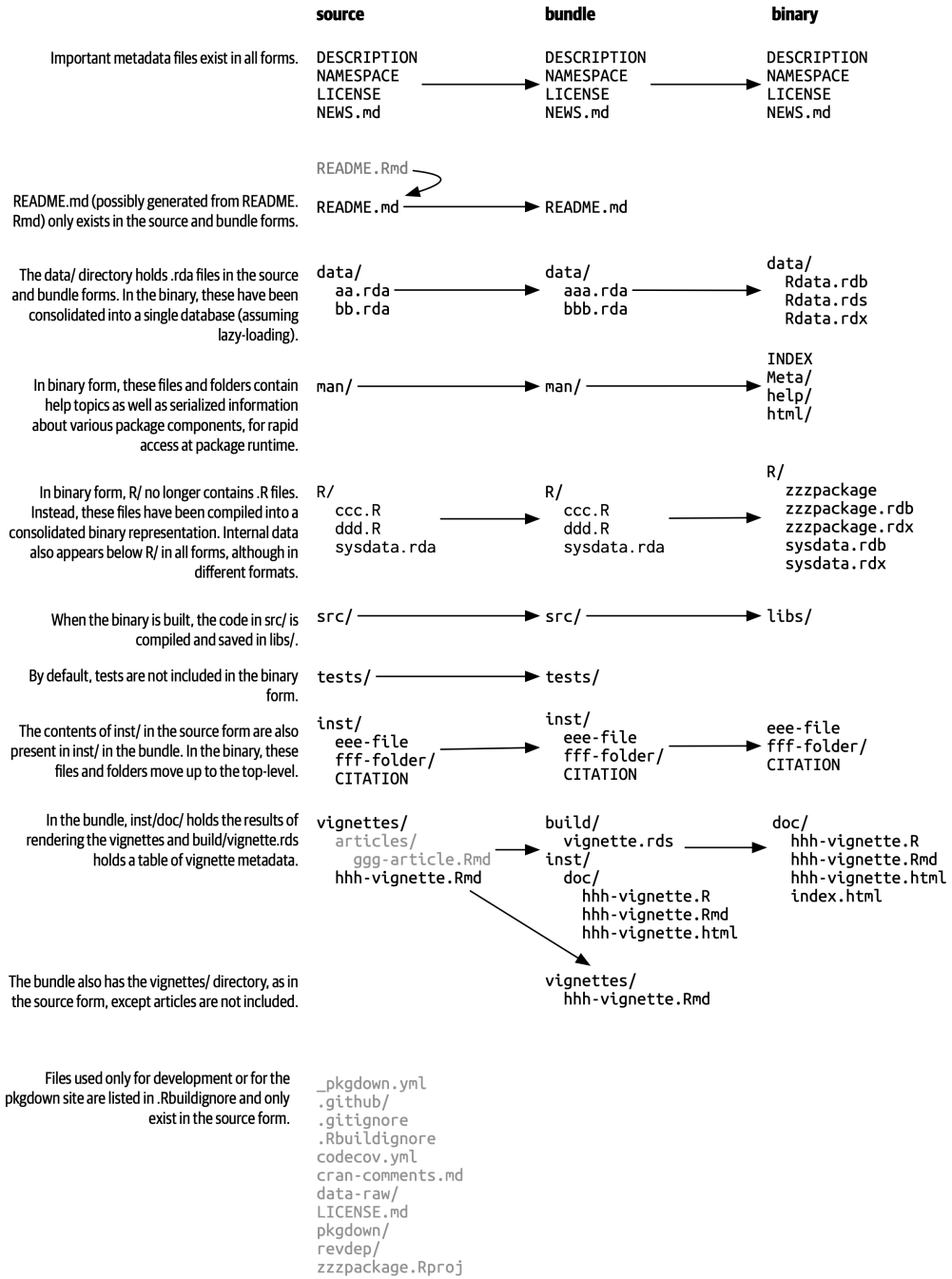
```
1 fs::dir_tree(system.file(package="diffmatchpatch
```

```
/Library/Frameworks/R.framework/Versions/4.3-
arm64/Resources/library/diffmatchpatch
├── DESCRIPTION
├── INDEX
├── Meta
│   ├── Rd.rds
│   ├── features.rds
│   ├── hsearch.rds
│   ├── links.rds
│   ├── nsInfo.rds
│   └── package.rds
├── NAMESPACE
├── NEWS.md
├── R
│   ├── diffmatchpatch
│   ├── diffmatchpatch.rdb
│   └── diffmatchpatch.rdx
```

Package Installation



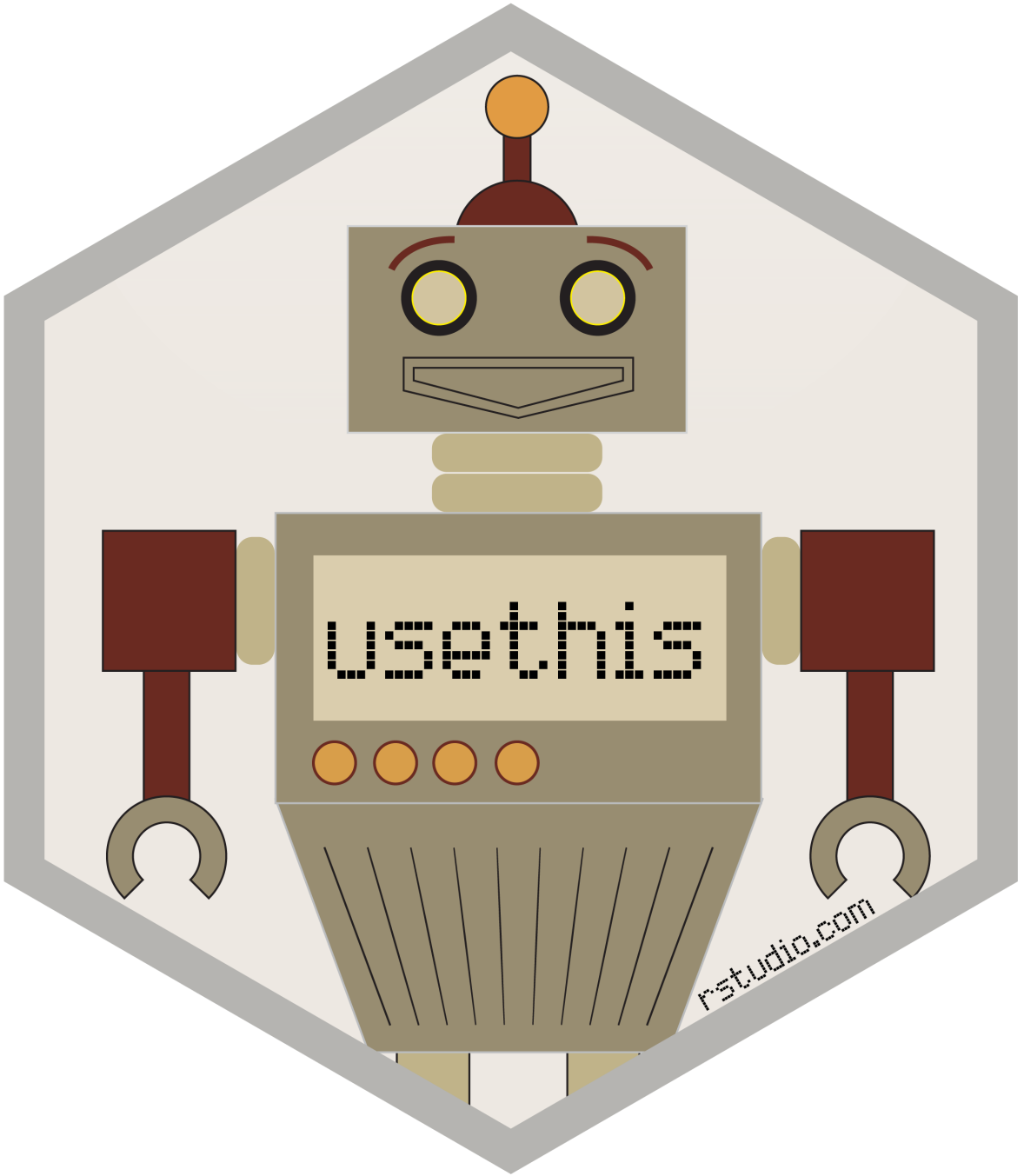
Package Installion - Files


source
bundle
binary
Important metadata files exist in all forms.
DESCRIPTION
NAMESPACE
LICENSE
NEWS.md
README.Rmd
README.md
README.md
The data/ directory holds .rda files in the source and bundle forms. In the binary, these have been consolidated into a single database (assuming lazy-loading).
data/
aa.rda
bb.rda
aaa.rda
bbb.rda
Rdata.rdb
Rdata.rds
Rdata.rdx
In binary form, these files and folders contain help topics as well as serialized information about various package components, for rapid access at package runtime.
man/
INDEX
Meta/
help/
html/
In binary form, R/ no longer contains .R files. Instead, these files have been compiled into a consolidated binary representation. Internal data also appears below R/ in all forms, although in different formats.
R/
ccc.R
ddd.R
sysdata.rda
zzzpackage
zzzpackage.rdb
zzzpackage.rdx
sysdata.rdb
sysdata.rdx
When the binary is built, the code in src/ is compiled and saved in libs/.
src/
libs/
By default, tests are not included in the binary form.
tests/
The contents of inst/ in the source form are also present in inst/ in the bundle. In the binary, these files and folders move up to the top-level.
inst/
eee-file
fff-folder/
CITATION
In the bundle, inst/doc/ holds the results of rendering the vignettes and build/vignette.rds holds a table of vignette metadata.
vignettes/
articles/
ggg-article.Rmd
hhh-vignette.Rmd
build/
vignette.rds
doc/
hhh-vignette.R
hhh-vignette.Rmd
hhh-vignette.html
index.html
The bundle also has the vignettes/ directory, as in the source form, except articles are not included.
Files used only for development or for the pkgdown site are listed in .Rbuildignore and only exist in the source form.
_pkgdown.yml
.github/
.gitignore
.Rbuildignore
codecov.yml
cran-comments.md
data-raw/
LICENSE.md
pkgdown/
revdep/
zzzpackage.Rproj

Package development

What follows is an *opinionated* introduction to package development,

- this is not the only way to do thing (none of the following are required)
- I would strongly recommend using:
  - RStudio
  - RStudio projects
  - GitHub
  - usethis
  - roxygen2
- Read and follow along with R Packages (2e) - Chap. 1 - "The Whole Game"


usethis
rstudio.com

`usethis`

This is an immensely useful package for automating all kinds of routine (and tedious) tasks within R

- Tools for managing git and GitHub configuration
- Tools for managing collaboration on GitHub via pull requests (see `pr_*()`)
- Tools for creating and configuring packages
- Tools for configuring your R environment (e.g. `.Rprofile` and `.Renviron`)
- and much much more

Live demo
Building a Package

Choosing a license

An important early step in developing a package is choosing a license - this is not trivial but is important to do early on, particularly if collaborating with others.

There are many resources available to help you choose a license, including:

https://choosealicense.com/

Package data

Exported data

Many packages contain sample data (e.g. `nycflights13`, `babynames`, etc.)

Generally these files are made available by saving a single data object as an `.Rdata` file (using `save()`) into the `data/` directory of your package.

- An easy option is to use `usethis::use_data(obj)` to create the necessary file(s)
- Data is usually compressed, for large data sets it may be worth trying different options (there is a 5 Mb package size limit on CRAN)
- Exported data must be documented (possible via roxygen)

Lazy data

By default when attaching a package all of that packages data is loaded - however if `LazyData: true` is set in the packages' `DESCRIPTION` then data is only loaded when used.

```
1 pryr::mem_used()
```

```
47.9 MB
```

```
1 library(nycflights13)
2 pryr::mem_used()
```

```
51.7 MB
```

```
1 invisible(flights)
2 pryr::mem_used()
```

```
92.4 MB
```

If you use `usethis::use_data()` this option will be set in `DESCRIPTION` automatically.

Raw data

When published a package should generally only contain the final data set, but it is important that the process to generate the data is documented as well as any necessary preliminary data.

- These can live any where but the general suggestion is to create a `data-raw/` directory which is included in `.Rbuildignore`
- `data-raw/` then contain scripts, data files, and anything else needed to generate the final object
- See examples babynames or nycflights
- Use `usethis::use_data_raw()` to create and ignore the `data-raw/` directory.

Internal data

If you have data that you want to have access to from within the package but not exported then it needs to live in a special Rdata object located at `R/sysdata.rda`.

- Can be created using `usethis::use_data(obj1, obj2, internal = TRUE)`
- Each call to the above will overwrite, so needs to include all objects
- Not necessary for small data frames and similar objects - just create in a script. Use when you want the object to be compressed.
- Example nflplotR which contains team logos and colors for NFL teams.

Raw data files

If you want to include raw data files (e.g `.csv`, shapefiles, etc.) there are generally placed in `inst/` (or a nested folder) so that they are installed with the package.

- Accessed using `system.file("dir", package = "package")` after install
- Use folders to keep things organized, Hadley recommends and uses `inst/extdata/`
- Example sf

Package vigenette

Vignette

Long form documentation for your package that live in `vignette/`, use `browseVignette(pkg)` to see a package's vignettes.

- Not required, but adds a lot of value to a package
- Generally these are literate documents (`.Rmd`, `.Rnw`) that are compiled to `.html` or `.pdf` when the package is built.
- Built packages retain the rendered document, the source document, and all source code
  - `vignette("colwise", package = "dplyr")` opens rendered version
  - `edit(vignette("colwise", package = "dplyr"))` opens code chunks
- Use `usethis::use_vignette()` to create a RMarkdown vignette template

Articles

These are an un-official extension to vignettes where package authors wish to include additional long form documentation that is included in their `pkgdown` site but not in the package (usually for space reasons).

- Use `usethis::use_article()` to create
- Files are added to `vignette/articles/` which is added to `.Rbuildignore`

Package checking

`R CMD check`

Last time we saw the usage of `R CMD check`, or rather `Build > Check Package` from within RStudio.

This is a good idea to run regularly to make sure nothing is broken and you are meeting the important package quality standards, but this only in the context of your machine, your version of R, your OS, and so on.

If using GitHub it is highly recommended that you run `usethis::use_github_action_check_standard()` to enable GitHub actions checks of the package each time it is pushed.

On each push this runs R CMD check on: * Latest R on MacOS, Windows, Linux (Ubuntu) * Previous and devel version of R on Linux (Ubuntu)

Package testing

Basic test structure

Package tests live in `tests/`,

- Any R scripts found in the folder will be run when Checking the package (not Building)
- Generally tests fail on errors, but warnings are also tracked
- Testing is possible via base R, including comparison of output vs. a file but it is not recommended (See Writing R Extensions)
- Note that R CMD check also runs all documentation examples (unless explicitly tagged dont run) - which can be used for basic testing


test
that
www.rstudio.com

testthat basics

Not the only option but probably the most widely used and with the best integration into RStudio.

Can be initialized in your project via `usethis::use_testthat()` which creates `tests/testthat/` and some basic scaffolding.

- `test/testthat.R` is what is run by R CMD Check and runs your other tests - handles some basic config like loading package(s)
- Test scripts go in `tests/testthat/` and should start with `test_`, suffix is usually the file in `R/` that is being tested.

`usethis::use_testthat()` has an edition argument, this is a way of maintaining backwards compatibility, generally

testthat script structure

From the bottom up,

- a single test is written as an expectation (e.g. `expect_equal()`, `expect_error()`, etc.)
- multiple related expectations are combined into a test group (`test_that()`), which provides
  - a human readable name and
  - local scope to contain the expectations and any temporary objects
- multiple test groups are combined into a file