Packages II

Lecture 21

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)

Department of Statistics, University of Illinois

Last Lecture

- Discussion of *why* one would want a package
- Basic package structure, key files and directories
- Base R commands to build, install, and check packages
- Writing documentation for package functions
    - either directly by editing `.Rd` files
    - or via `roxygen2` which converts from R file annotations

Plan for this Lecture 21: Packages II

This Lecture

- Testing:
    - RUnit, testthat and tinytest
- Git(Hub) Intregration
- Continuous Integration
    - Travis CI
- Repositories

Testing

Basic Approach via Base R

- We use `R CMD check` as a way to verify a package
- This already runs a lot of (implicit) checks for quality
- The one way *supported by base R* is scripts in `tests/`
    - Any file ending in `.R` will be run
    - Any such file creates an output `.Rout` (same basename)
    - if saved with the same basename ending in `.Rout.save`
    - then the current run’s output is checked against it
- This approach may look somewhat basic, but is comprehensive
- You can try with any package and any function below `tests/`

Package Testing

Unit Testing

- *Testing* has become a fairly central tenet of writing software
- The term *unit tests* is very frequenly used
- The basic idea is to test isolated components: units
- A *unit* most often correspond to a function
- The motivation for writing tests is to improve
  - code quality
  - code structure / organization
- This goes as far as ‘test-driven development’

Tests Frameworks in R

- In R, three sets of packages provide support:
- `RUnit` is oldest but still used by some
  - installs its tests files with the package
  - often in `inst/tests` or `inst/unitTests`, named `runit.*`
- `testthat` has become very popular and widely used
  - files generally in `tests/testthat`
  - tests only in source package, cannot test installed package
- `tinytest` is a fresh new approach I now prefer
  - allows tests per file, or directory, or package, ...
  - zero dependency, simple, straightforward
  - can test source and installed package easily

Quick Illustration and Comparison

- Using the sample package in this demo repo
- Ilustrating use of all three frameworks
- For each of the three, it starts with a function below `tests/`
- We will illustrate the respective `tests/*` functions first
- They all load the package being tested, and the test framework
- We then illustrate the per-framework functions

Package Testing

RUnit: `tests/doRUnit.R`

```r
library(sampleTestPackage)  ## GitHub: eddelbuettel/samplettestpackage
library(RUnit)

## define test suite
ts <- defineTestSuite(”c2f”,
                      dirs = system.file(”runittests”, package=”sampleTestPackage”),
                      testFileRegexp = ”^runit.+\\.r”,   # default
                      testFuncRegexp = ”^test.+”)        # also default

## run test suite:
res <- runTestSuite(ts)

## print text protocol to console:
printTextProtocol(res)

if (getErrors(res)$nFail > 0) stop(”TEST FAILED!”)
if (getErrors(res)$nErr > 0) stop(”TEST HAD ERRORS!”)
if (getErrors(res)$nTestFunc < 1) stop(”NO TEST FUNCTIONS RUN!”)
```

Package Testing

RUnit: `inst/tests/runit.c2f.r`

```
# simple RUnit examples for c2f()
test.c2f <- function() {
  checkEquals(c2f(0), 32)
  checkEquals(c2f(10), 50)
  checkException(c2f(”xx”))
}
```

Several `check*` redicates grouped in wrapper function.

More `check*` helpers exist: `checkEqualsNumeric()`, `checkTrue()`, `checkIdentical()`.

Often multiple wrapper functions per file.

Package Testing

testthat

Launch script `tests/testthat.R` very simple:

```r
library(testthat)
library(simpleTestPackage)

test_check(”simpleTestPackage”)
```

Package Testing

Test script: `tests/testthat/test-c2f.R`

```
context(”Check c2f functionality”)
library(simpleTestPackage)

test_that(”c2f functionality”, {
  expect_equal(c2f(0), 32)
  expect_equal(c2f(10), 50)
  expect_error(c2f(”xx”))
})
```

One defines a ‘context’ in which tests run.

Tests then executed from within an expression.

Many additional predicates available.

Package Testing

ILLINOIS

tinytest

Launch script `tests/tinytest.R` very simple:

```r
if (requireNamespace(”tinytest”, quietly=TRUE)) {
  set.seed(42)              # Set seed to make test deterministic
  Sys.setenv(”R_TESTS”=””)  # R makes us to this
  ## there are more granular ways to test files in a tinytest directory,
  ## see its package vignette; tests can also run once the package is installed
  ## using the same command `test_package(pkgName)`, or by director or file
  tinytest::test_package(”sampleTestPackage”), ncpu=getOption(”Ncpus”, 1))
}
```

Our use of a conditional is optional, but good format. The `R_TESTS` variable setting is an R issue required for all three frameworks with more complex packages.

Package Testing

Test script: `tests/tinytest/test-c2f.R`

```r
# simple tinytest examples for c2f()
library(sampleTestPackage)

expect_equal(c2f(0), 32)
expect_equal(c2f(10), 50)
expect_error(c2f(”xx”))
```

Simple: test files are just script files.

Each test is a direct function call.

Extensible, see for example package ttdo which we use in PrairieLearn.

Package Testing

Common `testthat` and `RUnit` features

- “setup” function initializes (*e.g.* setup database connection)
- “teardown” function to cleanup / restore / return resource

Common `tinytest`, `RUnit` and `testthat` features

- extensive documentation, many examples
- good exercise:
    - on a working package / test setup
    - invalidate an answer, see how framework reports

Git(hub)

Getting a package onto GitHub

- In your local package, do `git init`
- If you are setup correctly that is all we need
  - May need `git config --global ...` to add user & email
  - If you know about `ssh`, putting public key at GitHub very useful
  - Else look into connection caching *e.g.* here at 'Happy git with R'
- We can do this in RStudio Cloud in the 'Terminal'
- When we reload project a 'Git' tab in top right appears
- Select all files and click 'Commit'
- Now we also have a (local !!) `git` history

Getting a package onto GitHub

- Log into GitHub, select ‘+’ then ‘New repository’
- Give it a name and description
- You can leave README, gitignore, license empty (or fill it …)
- Hit ‘Create repository’
- The next screen contains *important* next steps:
    - to either create a new repo (but we have one)
    - or to push from existing repo (our case)

Getting a package onto GitHub

- So we do those steps (in the RStudio Cloud terminal window)
  - `git remote add origin https://github.com/U/R` (where U and R are your username and repo name)
  - `git push -u origin master`
- That is all! You can now see your repo, changes, log in a browser from anywhere
- (This assumes the repo is public – our project team repos are private so you only see them when logged into GitHub as 'you')
- (There can be complications if you use two-factor auth etc)
- (We assume that you can authenticate over https, see *e.g.* this)
- Once pushed, refresh at GitHub and see your file(s)

Continuous Integration

Continuous Integration: Travis

Set up Travis

- One of the benefits of GitHub is integration with other services
- Travis CI is such a service: log into `https://travis-ci.org`
- On initial login
  - align with GitHub account and
  - let Travis CI read your GitHub accound and repos
- Under account or profile find ‘Sync Account’
  - this updates what repos Travis knowns about
- You should find the new package (here: `sampleTestPackage`)
- Move the slider so that it turns ‘green’ ie “enabled”
- Default settings should be fine

Continuous Integration: Travis

Set up Travis as user `stat-430dspm`

1. We fork the repo eddelbuettel/sampletestpackage
2. We sign into Travis CI using our GitHub credentials (click ‘Sign in with GitHub’, then on next page ‘Authorize’ with defaults))
3. ‘First Sync’ happens automatically, it sees our two repositories (you can always request a sync from the UI via ‘Sync account’)
4. We enable ‘sampletestpackage’ with default settings
5. We can now see the Travis page at `https://travis-ci.org/stat-430dspm/sampletestpackage`

Continuous Integration: Travis

Adding a `.travis.yml`

The following simple default file should do as a `.travis.tml`:

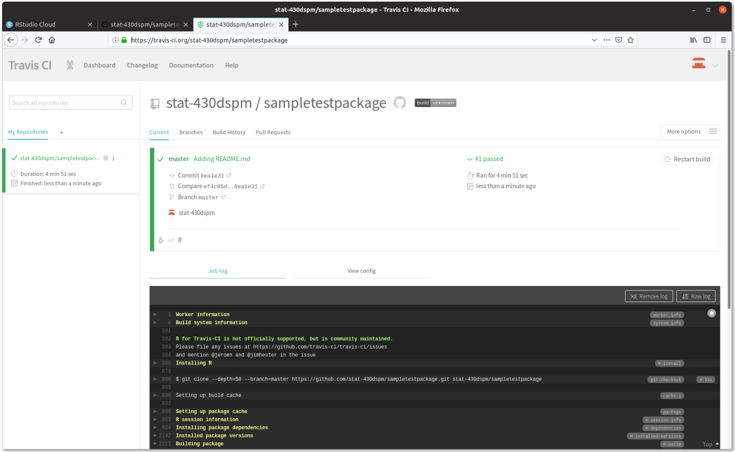
```
language: R
sudo: false
cache: packages
```

It says we use R, do not need `sudo` and will cache dependencies at Travis. We can also add a line with `.travis.yml` to `.Rbuildignore`

Once committed and pushed to GitHub and new (first !!) automated test should run – on this and every (!!) subsequent commit and push.

Continuous Integration: Travis

ILLINOIS



stat-430dspm / sampletestpackage

✓ stat-430dspm/sampletestpacka # 1
Duration: 4 min 51 sec
Finished: less than a minute ago

✓ master Adding README.md
Commit 8ea1e31
Compare ef4c05d..8ea1e31
Branch master
stat-430dspm
R

#1 passed
Ran for 4 min 51 sec
less than a minute ago

```
▶ Worker information                                                                          worker_info
▶ Build system information                                                                    system_info

R for Travis-CI is not officially supported, but is community maintained.
Please file any issues at https://github.com/travis-ci/travis-ci/issues
and mention @jeroen and @jimhester in the issue
▶ Installing R                                                                                R-install

▶ $ git clone --depth=50 --branch=master https://github.com/stat-430dspm/sampletestpackage.git stat-430dspm/sampletestpackage   git.checkout   0.53s

▶ Setting up build cache                                                                      cache.1

▶ Setting up package cache                                                                    package
▶ R session information                                                                       R-session-info
▶ Installing package dependencies                                                             R-dependencies
▶ Installed package versions                                                                  R-installed-versions
▶ Building package                                                                            R-build
```

Build #1 was successful!

(Because the repo was forked months ago, the newer commits at eddelbuettel/sampletestpackage are not reflected here. That is normal for a fork.)

Coverage

Coverage Tests

`covr`

- Unit tests are very useful and have become a standard
- But we may not know the proportion of code that is tested
- Enter “coverage analysis” which aims to quantify this
- The `covr` package is very useful, and well integrated with Travis
- We will not cover this here—plenty of on-line resources

Repositories

Package Repositories

Where to put packages

- Now that you learned how to create package …

  … finding where to provide them is a next question
- The `drat` package can help for simple collections of packages
  - either locally for your workgroup or lab
  - or simply via GitHub
- See the package documentation and vignettes

Summary of Lecture 21

- We discussed unit testing and use of `RUnit` and `testthat`
- We saw how to transfer a local package to GitHub
- We learned how to enable Travis CI automated testing
- We mentioned
    - testing coverage reports, *e.g.* via `covr`
    - local repositories via `drat`