

Creating a simple R package and unit tests with the `scriptests` package

Tony Plate

February 20, 2011

1 Introduction

`Scriptests` uses text files containing R commands and output, as though copied verbatim from an interactive R session. Here's an example test file from the simple package created in the next section:

```
> plus(3, 4)
[1] 7
>
```

The output in the transcript file must match the actual output from running the command for the test to pass (with some exceptions - see "Control over matching" below). This is the same concept as the standard `.R/.Rout.save` tests that are run by the R `CMD check`, but with some enhancements that are intended to make test development and maintenance faster, more convenient, and easier to automate:

- Only the output file is needed - the inputs are parsed from the output file (i.e., `.Rt` file, which is analogous to an `.Rout.save` file)
- Test output matching is more lenient on white space differences, and more flexible in that some test output can be transformed by regular expressions prior to matching, or ignored entirely
- directives can specify whether a test-case output mismatch should be noted as an informational message, a warning, or an error (one or more errors results in R `CMD check` stopping with an indication of error after running all tests). Unlike the standard tests in R `CMD check`, output mismatch detected by `scriptests` results in R `CMD check` stopping with an error.
- A concise summary of warnings and errors is given at the end
- Testing can continue after errors and can report multiple errors at the end, rather than stopping at the first error.

2 Creating a simple package

In this vignette, we'll create a complete R package named `testpkg` containing 5 files:

- `testpkg/DESCRIPTION`
- `testpkg/R/plus.R`
- `testpkg/tests/runtests.R`
- `testpkg/tests/plus.Rt`
- `testpkg/man/plus.Rd`

Initially, we'll start off with just the 2 files that the `runtests()` function in `scriptests` needs: `DESCRIPTION` and `plus.R`.

```
> # This block of R code creates a simple package containing 2 files
> dir.create("testpkg")
> cat(
+ Package: testpkg
+ Version: 1.0-0
+ License: GPL-3
+ Description: A simple example of using scriptests for unit tests
+ Title: Unit tests with scriptests
+ Author: Joe Blow <joeblo@foobar.org>
+ Maintainer: Joe Blow <joeblo@foobar.org>
+ Suggests: scriptests
+ ')
> dir.create("testpkg/R")
> cat(
+ plus <- function(x, y) x + y
+ ')
file="testpkg/DESCRIPTION", '
file="testpkg/R/plus.R", '
```

2.1 Adding some tests to the package

```
> # This block of R code adds 'runtests.R' and one test file to the package
> dir.create("testpkg/tests")
> cat(
+ library(scriptests)
+ runScripTests()
+ ')
> cat(
+ > plus(3, 4)
+ [1] 7
+ >
+ ')
file="testpkg/tests/runtests.R", '
file="testpkg/tests/plus.Rt", '
```

3 Running the tests interactively

This is often a good way of running tests while developing code. Tests are run in the current R session and can create, modify or delete variables in the R session. This is convenient and fast, partly because it doesn't fully build the package – it just loads the R source files from the package into the R session. (Actually, `source.pkg()` can do a bit more than that, but it doesn't understand namespaces, so if the package being loaded depends on namespaces, it won't work.)

```
> source.pkg(pkg="testpkg")
```

```
Reading 1 .R files into env at pos 2: 'pkgcode:testpkg'  
Sourcing D:/tplate/R/rforge/scriptests/testpkg/R/plus.R
```

```
> # use pattern= to only run test files that match the pattern  
> runtests(pkg="testpkg", pattern="plus")
```

```
* Package 'testpkg' is not loaded as a package; will remove "\btestpkg::~?", "\blibrary\  
* Removing old tests directory testpkg.tests  
* Copying D:\tplate\R\rforge\scriptests\testpkg\tests to testpkg.tests  
* Setting working directory to testpkg.tests  
* Running tests in D:/tplate/R/rforge/scriptests/testpkg/tests/plus.Rt (read 2 chunks)  
..  
plus.Rt: 2 tests with 0 errors, 0 warnings and 0 messages
```

3.1 When tests fail

Let's create a test that says $2+2 = 3$ (the second in the file):

```
> cat(                                                                 file="testpkg/tests/willfail.Rt", '  
+ > plus(1, 1)  
+ [1] 2  
+ > plus(2, 2)  
+ [1] 3  
+ > plus(3, 3)  
+ [1] 6  
+ > '  
> (res <- runtests(pkg="testpkg", pattern="fail"))
```

```
* Package 'testpkg' is not loaded as a package; will remove "\btestpkg::~?", "\blibrary\  
* Removing old tests directory testpkg.tests  
* Copying D:\tplate\R\rforge\scriptests\testpkg\tests to testpkg.tests  
* Setting working directory to testpkg.tests  
* Running tests in D:/tplate/R/rforge/scriptests/testpkg/tests/willfail.Rt (read 4 chunks)  
.  
* Error mismatch on output for test number 2 in D:/tplate/R/rforge/scriptests/testpkg/tes  
> plus(2, 2)
```

```

* Target output:
[1] 3
* Actual output:
[1] 4
..
willfail.Rt: 4 tests with 1 errors, 0 warnings and 0 messages
+++++ Test summary for tests in D:/tplate/R/rforge/scriptests/testpkg/tests/fail.*\Rt$ -
willfail.Rt: 4 tests with 1 errors, 0 warnings and 0 messages
total:          4 tests with 1 errors, 0 warnings and 0 messages

> # Uh-oh, one of the tests failed!
> # Look at the transcript of the tests
> dumprount(res, console=TRUE)

* Transcript of actual output from running commands in 'D:/tplate/R/rforge/scriptests/tes
> plus(1, 1)
[1] 2
> plus(2, 2)
[1] 4
> plus(3, 3)
[1] 6
>

> # To write the transcript to a file, don't supply console=TRUE to dumprount()
> dumprount(res)

* Writing transcript of actual output to willfail.Rout.tmp

See what is in the transcript file:

> cat(paste("....  ", readLines("willfail.Rout.tmp")), sep="\n")

....    * Transcript of actual output from running commands in 'D:/tplate/R/rforge/script
....    > plus(1, 1)
....    [1] 2
....    > plus(2, 2)
....    [1] 4
....    > plus(3, 3)
....    [1] 6
....    >

```

Now the original (failing) tests are in `testpkg/tests/willfail.Rt` and the transcript of the actual output is in `willfail.Rout.tmp`. You can use your favorite editor or diff tool to fix the original tests. In Emacs, the `ediff` function works very well for this purpose. To use `ediff`, visit both `testpkg/tests/willfail.Rt` and `willfail.Rout.tmp` in separate buffers, then do `M-x ediff-buffers` to start it up. Ediff shows a color-coded diff. Use the `'n'` and `'p'` keys to go forward and back in the differences, and the `'a'` and `'b'` keys to accept the current

difference in the A or B buffer and transfer it to the other buffer. Other diff tools have similar functionality, making it quick and easy to update tests if a function changed to produce new output.

In the above chunk of code, `res` was used to save the result of `runtests()` and supply it to `dumprout()`. In ordinary interactive usage, `res` can be left out when `dumprout()` is run immediately after `runtests()`: `dumprout()` uses `.Last.value` by default. However, that couldn't be done here, because `.Last.value` doesn't work in vignettes.

4 Running the tests as part of “R CMD check”

Before running `R CMD check`, let's add an Rd file for `plus` so that `R CMD check` doesn't get upset about missing documentation:

```
> dir.create("testpkg/man")
> cat(                                     file="testpkg/man/plus.Rd", '
+ \\name{plus}
+ \\alias{plus}
+ \\title{Add two numbers together}
+ \\description{Add two numbers together}
+ \\usage{plus(x, y)}
+ \\arguments{
+   \\item{x}{A number}
+   \\item{y}{A number}
+ }
+ \\value{A number}
+ ')
```

Also, let's rename the failing test file so that it doesn't get run

```
> file.rename("testpkg/tests/willfail.Rt", "testpkg/tests/willfail.Rnorun")
```

```
[1] TRUE
```

Normally, you'd type the following at a command line prompt – either in a unix shell or windows command line processor:

```
$ R CMD build testpkg
$ R CMD check testpkg_1.0-0.tar.gz
```

But in this vignette we'll run those commands from R:

```
> mysystem <- function(cmd) cat(system(cmd, intern=TRUE), sep="\n")
> mysystem("R CMD build testpkg")

* checking for file 'testpkg/DESCRIPTION' ... OK
* preparing 'testpkg':
* checking DESCRIPTION meta-information ... OK
```

```

* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* building 'testpkg_1.0-0.tar.gz'

> mysystem("R CMD check testpkg_1.0-0.tar.gz")

* using log directory 'D:/tplate/R/rforge/scriptests/testpkg.Rcheck'
* using R version 2.12.1 (2010-12-16)
* using platform: i386-pc-mingw32 (32-bit)
* using session charset: ISO8859-1
* checking for file 'testpkg/DESCRIPTION' ... OK
* this is package 'testpkg' version '1.0-0'
* checking package dependencies ... OK
* checking if this is a source package ... OK
* checking for executable files ... OK
* checking whether package 'testpkg' can be installed ... OK
* checking package directory ... OK
* checking for portable file names ... OK
* checking DESCRIPTION meta-information ... OK
* checking top-level files ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking R files for non-ASCII characters ... OK
* checking R files for syntax errors ... OK
* checking whether the package can be loaded ... OK
* checking whether the package can be loaded with stated dependencies ... OK
* checking whether the package can be unloaded cleanly ... OK
* checking for unstated dependencies in R code ... OK
* checking S3 generic/method consistency ... OK
* checking replacement functions ... OK
* checking foreign function calls ... OK
* checking R code for possible problems ... OK
* checking Rd files ... OK
* checking Rd metadata ... OK
* checking Rd cross-references ... OK
* checking for missing documentation entries ... OK
* checking for code/documentation mismatches ... OK
* checking Rd \usage sections ... OK
* checking Rd contents ... OK
* checking for unstated dependencies in examples ... OK
* checking examples ... NONE
* checking for unstated dependencies in tests ... OK
* checking tests ...
  Running 'runtests.R'
OK
* checking PDF version of manual ... OK

```

When tests run without any errors, no output regarding tests appears other than these three lines (which should be in the output above):

```
* checking tests ...
  Running 'runtests.R'
OK
```

We can look at the test output left in the `testpkg.Rcheck/tests` directory:

```
> cat(readLines("testpkg.Rcheck/tests/test-summary.txt"), sep="\n")
```

```
plus.Rt: 2 tests with 0 errors, 0 warnings and 0 messages
total:   2 tests with 0 errors, 0 warnings and 0 messages
```

```
> cat(readLines("testpkg.Rcheck/tests/plus.Rt.log"), sep="\n")
```

```
..
plus.Rt: 2 tests with 0 errors, 0 warnings and 0 messages
```

Note that R CMD check is applied here to the built package (i.e., to `testpkg_1.0-0.tar.gz`). So if any tests or code are updated in the package, be sure to rerun R CMD build before rerunning R CMD check.

4.1 Running the tests as part of “R CMD check” – when tests fail

Rename the failing test file back so that it does get run

```
> file.rename("testpkg/tests/willfail.Rnorun", "testpkg/tests/willfail.Rt")
```

```
[1] TRUE
```

```
> mysystem <- function(cmd) cat(system(cmd, intern=TRUE), sep="\n")
> mysystem("R CMD build testpkg")
```

```
* checking for file 'testpkg/DESCRIPTION' ... OK
* preparing 'testpkg':
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* building 'testpkg_1.0-0.tar.gz'
```

```
> mysystem("R CMD check testpkg_1.0-0.tar.gz")
```

```
* using log directory 'D:/tplate/R/rforge/scriptests/testpkg.Rcheck'
* using R version 2.12.1 (2010-12-16)
* using platform: i386-pc-mingw32 (32-bit)
* using session charset: ISO8859-1
* checking for file 'testpkg/DESCRIPTION' ... OK
```

```
* this is package 'testpkg' version '1.0-0'
* checking package dependencies ... OK
* checking if this is a source package ... OK
* checking for executable files ... OK
* checking whether package 'testpkg' can be installed ... OK
* checking package directory ... OK
* checking for portable file names ... OK
* checking DESCRIPTION meta-information ... OK
* checking top-level files ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking R files for non-ASCII characters ... OK
* checking R files for syntax errors ... OK
* checking whether the package can be loaded ... OK
* checking whether the package can be loaded with stated dependencies ... OK
* checking whether the package can be unloaded cleanly ... OK
* checking for unstated dependencies in R code ... OK
* checking S3 generic/method consistency ... OK
* checking replacement functions ... OK
* checking foreign function calls ... OK
* checking R code for possible problems ... OK
* checking Rd files ... OK
* checking Rd metadata ... OK
* checking Rd cross-references ... OK
* checking for missing documentation entries ... OK
* checking for code/documentation mismatches ... OK
* checking Rd \usage sections ... OK
* checking Rd contents ... OK
* checking for unstated dependencies in examples ... OK
* checking examples ... NONE
* checking for unstated dependencies in tests ... OK
* checking tests ...
```

```
Running 'runtests.R'
```

```
ERROR
```

```
Running the tests in 'tests/runtests.R' failed.
```

```
Last 13 lines of output:
```

```
See testpkg.Rcheck/tests/runtests.Rout.fail for a transcript of test comparisons
***** 13 lines of low-level output here b/c R CMD check wants to display just 13 lines
```

```
.
```

```
* Error mismatch on output for test number 2 in willfail.Rt:
```

```
> plus(2, 2)
```

```
* Target output:
```

```
[1] 3
```

```
* Actual output:
```

```
[1] 4
```



```
..
willfail.Rt: 4 tests with 1 errors, 0 warnings and 0 messages
See testpkg.Rcheck/tests/runtests.Rout.fail for more info
```

Note that “R CMD check” outputs only the last 13 lines when a test fails. `scriptests` is programmed to put 12 lines of low-level errors and some pointers to files so that output is still somewhat useful when automated tests are run on a platform which makes nothing other than the output of “R CMD check” available. Under these conditions, tests are expected to all pass, and test summaries are less informative than the detailed description of why some tests failed.

A transcript of the failed test is in the file `testpkg.Rcheck/tests/willfail.Rout` – this file can be compared against the original test file `testpkg/tests/willfail.Rt` to fix the code or the test output.

```
> cat(readLines("testpkg.Rcheck/tests/test-summary.txt"), sep="\n")
```

```
plus.Rt:      2 tests with 0 errors, 0 warnings and 0 messages
willfail.Rt:  4 tests with 1 errors, 0 warnings and 0 messages
total:        6 tests with 1 errors, 0 warnings and 0 messages
```

```
> cat(readLines("testpkg.Rcheck/tests/plus.Rt.log"), sep="\n")
```

```
..
plus.Rt: 2 tests with 0 errors, 0 warnings and 0 messages
```

```
> cat(readLines("testpkg.Rcheck/tests/willfail.Rt.log"), sep="\n")
```

```
.
* Error mismatch on output for test number 2 in willfail.Rt:
> plus(2, 2)
* Target output:
[1] 3
* Actual output:
[1] 4
..
willfail.Rt: 4 tests with 1 errors, 0 warnings and 0 messages
```

5 Interactive tests, using an installed package

We’ve looked at how to run tests in the current R session, and how to run them using `R CMD CHECK`. The advantage of running in the current R session are that it is quick to update code and tests, and it’s easy to run particular tests. The disadvantage is that the R code is not loaded as a proper R package, so anything that depends on namespaces or the package loading machinery won’t work properly. Using `R CMD check` is the most solid way to run tests, but it’s slower and all tests are run.

The middle way is to run tests from an interactive R session the same way that `R CMD check` does: by firing off a separate R session for each test file. This can be done by invoking `runtests()` with the `full=TRUE` argument. Doing this gets tests and the installed package from two different places:

- the installed package comes from `testpkg.Rcheck/testpkg` (left behind by the most recent invocation of `R CMD check`)
- the tests come from `testpkg/tests`

Using `runtests()` this way makes it quick to update test files: any change to the source code of tests is immediately picked up by the next invocation of `runtests()`. However, if R code or some other aspect of the package is changed, the package must be reinstalled by invoking `R CMD build` and `R CMD check` again.

```
> runtests(full=TRUE)

* Using package in 'testpkg.Rcheck/testpkg' for running tests
* Removing old tests directory testpkg.Rcheck/tests
* Copying D:\tplate\R\rforge\scriptests\testpkg\tests to testpkg.Rcheck/tests
* Setting working directory to testpkg.Rcheck/tests

### Show output from here ###
** Running 'plus.R' in D:/tplate/R/rforge/scriptests/testpkg.Rcheck/tests
..
plus.Rt: 2 tests with 0 errors, 0 warnings and 0 messages
** Running 'willfail.R' in D:/tplate/R/rforge/scriptests/testpkg.Rcheck/tests
.
* Error mismatch on output for test number 2 in willfail.Rt:
> plus(2, 2)
* Target output:
[1] 3
* Actual output:
[1] 4
..
willfail.Rt: 4 tests with 1 errors, 0 warnings and 0 messages
### Test Summary: 1 file without errors
plus.Rt:      2 tests with 0 errors, 0 warnings and 0 messages
### 1 file with 1 errors
willfail.Rt: 4 tests with 1 errors, 0 warnings and 0 messages
### Overall
total:      6 tests with 1 errors, 0 warnings and 0 messages
***** 13 lines of low-level output here b/c R CMD check wants to display just 13 lines, t
.
* Error mismatch on output for test number 2 in willfail.Rt:
> plus(2, 2)
```

```
* Target output:
[1] 3
* Actual output:
[1] 4
..
willfail.Rt: 4 tests with 1 errors, 0 warnings and 0 messages
NULL
```

6 Programming scripts to check whether tests passed or failed

At the end of running `R CMD check`, the file `test-summary.txt` will be left in the `<package>.Rcheck/tests` directory. To be entirely sure that the tests were run, a script should check for the existence of `test-summary.txt`.

If any tests fail, the file `test-summary.fail` (a copy of `test-summary.txt`) will also be left in the `tests` directory – the existence of this file can be used in a programmatic check for whether all tests passed.

7 Package dependencies

It's generally not a good idea to list `scriptests` in the `Depends:` field of a package `DESCRIPTION` file, because that would cause `scriptests` to be loaded whenever the package is loaded. Instead, add the line `Suggests: scriptests` to `DESCRIPTION` file. If there is an existing `Suggests:` line, just add `scriptests` to it. If the `scriptests` package is not available when `R CMD check` is run on the package, the tests will fail (because it won't find `runScriptTests()`).

8 Rt format

All commands in the transcript file must be prefixed with command or continuation prompts, exactly as they appear in a transcript. Any uninterpretable lines will be ignored with warnings.

`scriptests` uses simple heuristics to identify commands, comments and output. If the transcript cannot be separated into comments, commands and output by these heuristics (e.g., if a command prints out a line starting with the command prompt `"> "`), things will not work properly.

9 Controlling testing and test-output matching

9.1 Continuing tests after an error

To have tests continue to run after encountering an error, put the command `options(error=function() NULL)` at the beginning of the transcript file. This

will cause the non-interactive R session that runs the commands in the scripts to continue after an error, instead of stopping, which is the default behavior for non-interactive R.

9.2 Control over matching

Actual output is matched to desired output extracted from the transcript file in a line-by-line fashion. If text is wrapped differently over multiple lines, the tests will fail (unless `ignore-linebreaks` is used). Different output width can easily happen if `options("width")` was different in the session that generated the desired output. Before trying to match, `scriptests` converts all white-space to single white-space, unless a control line specifies otherwise.

The following control lines can be present in the transcript after a command and before its output:

```
#@ignore-output: Ignore the output of this particular command – a test
                  with this control line will always pass (unless it causes an R error, and
                  options(error=function() NULL) was not set earlier in the file.)
```

```
#@gsub(pattern, replacement, WHAT): where WHAT is target, actual or both
                                       (without quotes). Make a global substitution of replacement
                                       text for pattern text (a regular expression) in the desired
                                       (target) output or the actual output. E.g.,
```

```
> cat("The date is <", date(), ">\n", sep="")
#@gsub("<[^>]*>", "<a date>", both)
The date is <Sat Jul 10 16:20:01 2010>
>
```

```
#@warn-only: OPTIONAL-TEXT: A mismatch is treated as an "warning",
              not an error
```

```
#@info-only: OPTIONAL-TEXT: A mismatch is treated as an "info" event,
              not an error
```

```
#@diff-msg:  OPTIONAL-TEXT: Output OPTIONAL-TEXT if the desired
              and actual output do not match
```

```
#@keep-whitespace: Leave the whitespace as-is in the desired and actual out-
                   put
```

```
#@ignore-linebreaks: Target and actual will match even if wrapped differently
                    over multiple lines
```

9.3 CONFIG file

The `tests` directory can also contain an optional CONFIG file, which can specify the functions to call for testing. The defaults are equivalent to the following lines in the CONFIG file:

```

Depends: scriptests
Debug: FALSE
Initialize: scriptests::initializeTests()
Diff: scriptests::ScripDiff()
Finalize: scriptests::summarizeTests() }

```

10 Scriptests and Emacs and ESS

The standard Emacs ESS functions for writing out ".Rt" files will strip trailing white space, which can result in many unimportant mismatches when using `ediff` to compare ".Rt" and ".Rout" files (e.g., because an R transcript will have "> " for empty command lines). Also, ".Rt" files are read-only by default, and the return key is bound to a command to send the current line to an R interpreter. It is more convenient if all these special behaviors are turned off. Put the following in your `.emacs` file to tell ESS not mess with ".Rt" files prior to saving them:

```

(add-hook 'ess-transcript-mode-hook
  ;; According to the ess docs, ess-nuke-trailing-whitespace-p
  ;; is supposed to be nil by default (see the defvar in ess-utils.el).
  ;; But it gets set to t somewhere else, so disable it here for
  ;; .Rt files, and also make RET behave the regular way.
  (lambda ()
    (if (string-match "[Rr]t$" (buffer-name))
      (progn
        (define-key ess-transcript-mode-map (kbd "RET") 'newline)
        (toggle-read-only 0)
        (make-variable-buffer-local 'ess-nuke-trailing-whitespace-p)
        (setq ess-nuke-trailing-whitespace-p nil)))) t)

```