

R packages

Rich FitzJohn

Why make a package?

To share code with other people

Why make a package?

To share code with other people

If you're looking for ways to share your code, you've probably found that the best way is to share it with other people. This is the goal of this document. It is a guide for how to share your code with other people in a way that is easy to understand and use.

There are many ways to share code, but the most common is to put it in a package. A package is a collection of code that is organized in a way that makes it easy to find and use.

There are many ways to organize a package, but the most common is to use a flat directory structure. This means that all the code files are in a single directory.

There are many ways to name a package, but the most common is to use a simple, descriptive name.

There are many ways to license a package, but the most common is to use a simple, descriptive license.

There are many ways to distribute a package, but the most common is to use a simple, descriptive distribution method.

There are many ways to maintain a package, but the most common is to use a simple, descriptive maintenance strategy.

There are many ways to promote a package, but the most common is to use a simple, descriptive promotion strategy.

There are many ways to measure the success of a package, but the most common is to use a simple, descriptive measurement strategy.

There are many ways to improve a package, but the most common is to use a simple, descriptive improvement strategy.

There are many ways to document a package, but the most common is to use a simple, descriptive documentation strategy.

There are many ways to test a package, but the most common is to use a simple, descriptive testing strategy.

There are many ways to secure a package, but the most common is to use a simple, descriptive security strategy.

There are many ways to optimize a package, but the most common is to use a simple, descriptive optimization strategy.

There are many ways to integrate a package, but the most common is to use a simple, descriptive integration strategy.

There are many ways to migrate a package, but the most common is to use a simple, descriptive migration strategy.

There are many ways to backup a package, but the most common is to use a simple, descriptive backup strategy.

There are many ways to restore a package, but the most common is to use a simple, descriptive restoration strategy.

There are many ways to upgrade a package, but the most common is to use a simple, descriptive upgrade strategy.

There are many ways to downgrade a package, but the most common is to use a simple, descriptive downgrade strategy.

There are many ways to uninstall a package, but the most common is to use a simple, descriptive uninstall strategy.

There are many ways to install a package, but the most common is to use a simple, descriptive install strategy.

There are many ways to update a package, but the most common is to use a simple, descriptive update strategy.

There are many ways to delete a package, but the most common is to use a simple, descriptive delete strategy.

There are many ways to move a package, but the most common is to use a simple, descriptive move strategy.

There are many ways to rename a package, but the most common is to use a simple, descriptive rename strategy.

There are many ways to create a package, but the most common is to use a simple, descriptive create strategy.

There are many ways to delete a package, but the most common is to use a simple, descriptive delete strategy.

There are many ways to update a package, but the most common is to use a simple, descriptive update strategy.

There are many ways to downgrade a package, but the most common is to use a simple, descriptive downgrade strategy.

There are many ways to uninstall a package, but the most common is to use a simple, descriptive uninstall strategy.

There are many ways to install a package, but the most common is to use a simple, descriptive install strategy.

There are many ways to update a package, but the most common is to use a simple, descriptive update strategy.

There are many ways to downgrade a package, but the most common is to use a simple, descriptive downgrade strategy.

There are many ways to uninstall a package, but the most common is to use a simple, descriptive uninstall strategy.

There are many ways to install a package, but the most common is to use a simple, descriptive install strategy.

There are many ways to update a package, but the most common is to use a simple, descriptive update strategy.

There are many ways to downgrade a package, but the most common is to use a simple, descriptive downgrade strategy.

There are many ways to uninstall a package, but the most common is to use a simple, descriptive uninstall strategy.

There are many ways to install a package, but the most common is to use a simple, descriptive install strategy.

There are many ways to update a package, but the most common is to use a simple, descriptive update strategy.

There are many ways to downgrade a package, but the most common is to use a simple, descriptive downgrade strategy.

There are many ways to uninstall a package, but the most common is to use a simple, descriptive uninstall strategy.

There are many ways to install a package, but the most common is to use a simple, descriptive install strategy.

There are many ways to update a package, but the most common is to use a simple, descriptive update strategy.

There are many ways to downgrade a package, but the most common is to use a simple, descriptive downgrade strategy.

There are many ways to uninstall a package, but the most common is to use a simple, descriptive uninstall strategy.

There are many ways to install a package, but the most common is to use a simple, descriptive install strategy.

There are many ways to update a package, but the most common is to use a simple, descriptive update strategy.

There are many ways to downgrade a package, but the most common is to use a simple, descriptive downgrade strategy.

There are many ways to uninstall a package, but the most common is to use a simple, descriptive uninstall strategy.

There are many ways to install a package, but the most common is to use a simple, descriptive install strategy.

There are many ways to update a package, but the most common is to use a simple, descriptive update strategy.

```
#!/usr/bin/perl

use strict;
use warnings;

my $script = $0;
my $dir = dirname($script);

my @files = glob("$dir/*.*");

for my $file (@files) {
    my $name = basename($file);
    my $type = "text";

    if ($name =~ /\.pl$/) {
        $type = "perl";
    }

    my $content = read_file($file);

    if ($content =~ /^#!/) {
        $content =~ s/^#!/usr/bin/perl\n/;
    }

    write_file($file, $content);
}

sub read_file {
    my $file = shift;

    open(my $fh, "<$file");
    my $content = do {
        local $* = $fh;
        <*>
    };
    close($fh);

    return $content;
}

sub write_file {
    my $file = shift;
    my $content = shift;

    open(my $fh, ">$file");
    print $fh $content;
    close($fh);
}

__END__

if ($?) {
    print "Done!\n";
} else {
    print "Failed!\n";
}

exit $?;
```

```
#!/usr/bin/perl

use strict;
use warnings;

my $script = $0;
my $dir = dirname($script);

my @files = glob("$dir/*.*");

for my $file (@files) {
    my $name = basename($file);
    my $type = "text";

    if ($name =~ /\.pl$/) {
        $type = "perl";
    }

    my $content = read_file($file);

    if ($content =~ /^#!/) {
        $content =~ s/^#!/usr/bin/perl\n/;
    }

    write_file($file, $content);
}

sub read_file {
    my $file = shift;

    open(my $fh, "<$file");
    my $content = do {
        local $* = $fh;
        <*>
    };
    close($fh);

    return $content;
}

sub write_file {
    my $file = shift;
    my $content = shift;

    open(my $fh, ">$file");
    print $fh $content;
    close($fh);
}

__END__

if ($?) {
    print "Done!\n";
} else {
    print "Failed!\n";
}

exit $?;
```

```
#!/usr/bin/perl

use strict;
use warnings;

my $script = $0;
my $dir = dirname($script);

my @files = glob("$dir/*.*");

for my $file (@files) {
    my $name = basename($file);
    my $type = "text";

    if ($name =~ /\.pl$/) {
        $type = "perl";
    }

    my $content = read_file($file);

    if ($content =~ /^#!/) {
        $content =~ s/^#!/usr/bin/perl\n/;
    }

    write_file($file, $content);
}

sub read_file {
    my $file = shift;

    open(my $fh, "<$file");
    my $content = do {
        local $* = $fh;
        <*>
    };
    close($fh);

    return $content;
}

sub write_file {
    my $file = shift;
    my $content = shift;

    open(my $fh, ">$file");
    print $fh $content;
    close($fh);
}

__END__

if ($?) {
    print "Done!\n";
} else {
    print "Failed!\n";
}

exit $?;
```

```
#!/usr/bin/perl

use strict;
use warnings;

my $script = $0;
my $dir = dirname($script);

my @files = glob("$dir/*.*");

for my $file (@files) {
    my $name = basename($file);
    my $type = "text";

    if ($name =~ /\.pl$/) {
        $type = "perl";
    }

    my $content = read_file($file);

    if ($content =~ /^#!/) {
        $content =~ s/^#!/usr/bin/perl\n/;
    }

    write_file($file, $content);
}

sub read_file {
    my $file = shift;

    open(my $fh, "<$file");
    my $content = do {
        local $* = $fh;
        <*>
    };
    close($fh);

    return $content;
}

sub write_file {
    my $file = shift;
    my $content = shift;

    open(my $fh, ">$file");
    print $fh $content;
    close($fh);
}

__END__

if ($?) {
    print "Done!\n";
} else {
    print "Failed!\n";
}

exit $?;
```


Functions and workflow

1. Long script file with a series of commands. Lots of cut and paste.

```
----- myproject.R -----  
data <- read.table(file.choose(), header=T, sep=",")  
data[[1]] <- as.character(data[[1]])  
  
...  
  
plot(data[[1]])  
plot(data[[2]])  
  
...  
  
fit1 <- lm(data[[1]] ~ data[[2]] + data[[3]])  
theresult <- fitted(fit1)  
fit2 <- lm(data[[1]] ~ data[[2]])  
theresult <- fitted(fit2) - theresult
```

Functions and workflow

2. Abstract into functions, put in its own file

```
— project-fun.R —
load.data <- function(filename) {
  dat <- read.csv(filename)
  ## Error checking
  ...
  ## Fix up data types
  ...
  ## Make transformed data
  ...
  dat
}

fit.models <- function(data.set) {
  ## Fit each model
  ...
  list(model1, model2, model3)
}
```

Functions and workflow

2. Abstract into functions, put in its own file

```
— project-fun.R —  
load.data <- function(filename) {  
  dat <- read.csv(filename)  
  ## Error checking  
  ...  
  ## Fix up data types  
  ...  
  ## Make transformed data  
  ...  
  dat  
}  
  
fit.models <- function(data.set) {  
  ## Fit each model  
  ...  
  list(model1, model2, model3)  
}
```

```
— project.R —  
source("project-fun.R")  
dat <- load.data("data/mydataset.csv")  
obj <- fit.models(dat)  
write.paper(obj) # still buggy.
```

Functions and workflow

3. Collect functions into package and use that

R package contents

- ▶ Some information for R
- ▶ R files containing functions
- ▶ A help file for each function

Optionally,

- ▶ Data sets
- ▶ PDF "vignette" documents
- ▶ Compiled code (C, C++, Fortran)

The "DESCRIPTION" file

DESCRIPTION

```
Package: hammer
Title: A Hammer
Description: An example package
Version: 1.0
Author: Rich FitzJohn
Maintainer: Rich FitzJohn <fitzjohn@zoology.ubc.ca>
License: Unlimited
```

- ▶ Tells R a few things about the package
- ▶ Used to construct the lists on CRAN
- ▶ Can be shown by `library(help=hammer)`

R files containing functions

```
R/functions.R  
squash <- function(x)  
  x * runif(length(x))
```

- ▶ Any number of R files that define functions
- ▶ Concatenated and `source()`d when the package is loaded

Help files

man/squash.Rd

```
\name{squash}
\alias{squash}
\title{Squash Some Numbers}
\usage{
  squash(x)
}
\arguments{
  \item{x}{A numeric vector of numbers to squash.}
}
\description{
  ...a longer description would go here...
}
\examples{
squash(1:10)
}
\author{Rich FitzJohn}
\keyword{model}
```

- ▶ Every function must be documented.

Loading packages

What happens when a package is loaded?

```
> library(hammer)
```

Loading packages

What happens when a package is loaded?

```
> library(hammer)
```

...the `squash` function is now available

Loading packages

What happens when a package is loaded?

```
> library(hammer)
```

...the `squash` function is now available

```
> squash(1:5)
```

```
[1] 0.07671335 0.89909571 2.28221560
```

```
[4] 1.59110592 4.91386456
```

Loading packages

Before the package is loaded:

```
> search()
```

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


Loading packages

Before the package is loaded:

```
> search()

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

After loading...

```
> library(hammer)
> search()

[1] ".GlobalEnv"          "package:hammer"    "package:stats"
[4] "package:graphics"    "package:grDevices" "package:utils"
[7] "package:datasets"    "package:methods"    "Autoloads"
[10] "package:base"
```

Loading packages

Before the package is loaded:

```
> search()

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

After loading...

```
> library(hammer)
> search()

[1] ".GlobalEnv"          "package:hammer"    "package:stats"
[4] "package:graphics"    "package:grDevices" "package:utils"
[7] "package:datasets"    "package:methods"    "Autoloads"
[10] "package:base"
```

R will use functions found in the first position in this list.

Loading packages

Which is the source of this warning:

```
> squash <- function(x) 1  
> library(hammer)
```

```
Attaching package: 'hammer'
```

```
The following object(s) are masked _by_ '.GlobalEnv':
```

```
  squash
```

Loading packages

Which is the source of this warning:

```
> squash <- function(x) 1  
> library(hammer)
```

```
Attaching package: 'hammer'
```

```
The following object(s) are masked _by_ '.GlobalEnv':
```

```
  squash
```

(But this is OK, as the global object is not a function...)

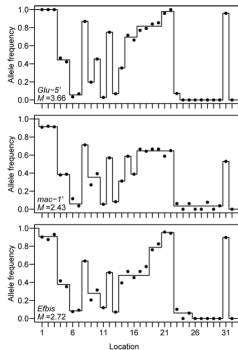
```
> squash <- 1  
> library(hammer)
```

My packages

- ▶ TRAMPR: TRFLP Analysis and Matching Package for R

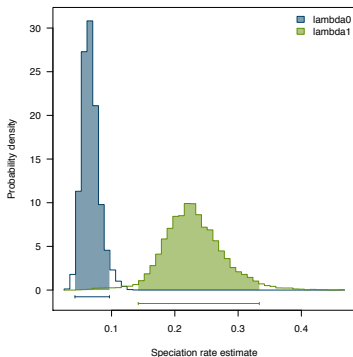
My packages

- ▶ TRAMPR: TRFLP Analysis and Matching Package for R
- ▶ "Mosaic" (with Leithen M'Gonigle)



My packages

- ▶ TRAMPR: TRFLP Analysis and Matching Package for R
- ▶ "Mosaic" (with Leithen M'Gonigle)
- ▶ diversitree



Version Control

Version control

If you're going to spend time writing code, you don't want to lose any.

Version control

If you're going to spend time writing code, you don't want to lose any. Typical version control by most biologists looks like this:

Version control

If you're going to spend time writing code, you don't want to lose any. Typical version control by most biologists looks like this:

1. Write `my_functions.R`

Version control

If you're going to spend time writing code, you don't want to lose any. Typical version control by most biologists looks like this:

1. Write `my_functions.R`
2. Make some changes, save as `my_functions2.R`

Version control

If you're going to spend time writing code, you don't want to lose any. Typical version control by most biologists looks like this:

1. Write `my_functions.R`
2. Make some changes, save as `my_functions2.R`
3. Work with Bob, save as `my_functions2_after_bob.R`

Version control

If you're going to spend time writing code, you don't want to lose any. Typical version control by most biologists looks like this:

1. Write `my_functions.R`
2. Make some changes, save as `my_functions2.R`
3. Work with Bob, save as `my_functions2_after_bob.R`
4. Come back to the project after a year, save as `my_functions_after_field_season.R`

Version control

If you're going to spend time writing code, you don't want to lose any. Typical version control by most biologists looks like this:

1. Write `my_functions.R`
2. Make some changes, save as `my_functions2.R`
3. Work with Bob, save as `my_functions2_after_bob.R`
4. Come back to the project after a year, save as `my_functions_after_field_season.R`
5. ...

In the end, there is a huge mess of files, and no record of *what* changed between versions.

Worse: different versions of functions depend on other versions of files/data.

Version control: solutions

Version Control with



Version control: revisit old versions

```
rich@kea <git log --graph --decorate --pretty=oneline --abbrev-commit> ~/Documents/Projects/di...
* 3185d16 (HEAD, origin/master, github/master, master) bib/studies.bib: Typo in Boucher-2012 entry diversitree/man/mcmc.Rd: Cl
* cd1f28c * R/model-mkn-multitrait.R, R/model-musse-multitrait.R: restore 'pars.only' argument to multitrait fun
* c830849 Archive of the 0.9-2 release tarball
* a94efb7 (tag: v0.9-2, origin/release, github/release, release) Spelling, keyword change only.
* 16e9715 Website changes
* 8aa1a26 Improvements to documents, formatting, figures.
* 57b84a8 * R/check.R: Negative branch length check was being skipped.
* f84bdce Merge branch 'master' of /Users/rich/Library/git/diversitree
| \
| * 1127f66 Formatting changes to tutorials.
| * e0b90c9 Fixed some display issues.
| * 634a255 Markdown fixes
| * 6b002ff Markdown fixes
| * 92a49f2 Basic readme file
| \
* 397da35 (ext) Reorder asr tests.
* 25e5079 First version of the extending diversitree manual done
* a680ad3 * R/check.R: typo in check.scalar(), more informative error in check.loaded.symbol().
* 6e7491d Ignore temporary files
* e1ae37c Simplify push-ww
* 6550cb2 Ignore file created. Plus global ww push.
* 885b376 Updated two scoping problems, updated website information.
* b3f82a3 Initial reworking of "extending diversitree" manual. Mk2 reimplementaton done.
* 12090c5 Starting back on "extending diversitree"
* c5836e5 Website updates (old source archives, faq page missing from import)
* 331d6e3 Small changes to the tutorial so that it compiles.
:
_
```

Version control: see what you changed

Mar 23, 2012



* R/check.R: typo in check.scalar(), more informative error in

richfitz authored 6 days ago

a600ad30e5

Browse code

Mar 22, 2012



Updated two scoping problems, updated website information.

richfitz authored 7 days ago

885b37696d

Browse code



Initial reworking of "extending diversitree" manual. Mk2 reimplementa...

richfitz authored 7 days ago

b3f02a3880

Browse code



* Too many files to list: huge rewrite of most behind the scenes

richfitz authored 7 days ago

b326bdb7cd

Browse code

Mar 13, 2012



* DESCRIPTION: Bump to 0.9-1, add mention of Classe and BISSE-ness

richfitz authored 16 days ago

3650ad4217

Browse code

Feb 22, 2012



* R/model-bisseness.R: Added the "BISSE-ness" model from Karen and S...

richfitz authored a month ago

99adc29dc5

Browse code

Feb 13, 2012



* DESCRIPTION: Bump R dependency up, and add support for current

richfitz authored a month ago

6f83cb141d

Browse code

Version control: see when you wrote that mistake

7fd4d683 * richfitz 2010-06-14 * R/constrain.R: Better constraining of cons...	62	mcmc.loop <- function() {
	63	for (i in seq_len(nsteps)) {
c6ff3c25 * richfitz 2010-09-22 * DESCRIPTION: Bumped version number ...	64	hist[[i]] <<- tmp <- sampler(posterior, x.init, y.init, w,
	65	lower, upper, control)
	66	x.init <- tmp[[1]]
	67	y.init <- tmp[[2]]
7fd4d683 * richfitz 2010-06-14 * R/constrain.R: Better constraining of cons...	68	if (print.every > 0 && i %% print.every == 0)
	69	cat(sprintf("%d: (%s) -> %2.5f\n", i,
	70	paste(sprintf("%2.4f", tmp[[1]]), collapse=","),
	71	tmp[[2]])
6f83cb14 * richfitz 2012-02-13 * DESCRIPTION: Bump R dependency up...	72	if (save.every > 0 && i %% save.every == 0) {
	73	ok <- try(write.csv(clean.hist(hist[seq_len(i)]),
	74	save.file, row.names=FALSE))
	75	if (inherits(ok, "try-error"))
	76	warning("Error while writing progress file (continuing)",
	77	immediate.=TRUE)
	78	}
7fd4d683 * richfitz 2010-06-14 * R/constrain.R: Better constraining of cons...	79	}
35ed842c * richfitz 2010-06-23 * R/diverstree-branches.R: Many changes ...	80	hist
7fd4d683 * richfitz 2010-06-14 * R/constrain.R: Better constraining of cons...	81	}
	82	

Version control: see when you wrote that mistake

```
51 59   mcmc.loop <- function() {
52 60     for ( i in seq_len(nsteps) ) {
53 61       hist[[i]] <- tmp <- sampler(posterior, x.init, y.init, w,
... .. @@ -58,6 +66,13 @@ mcmc.default <- function(lik, x.init, nsteps, w, prior=NULL,
58 66         cat(sprintf("%d: {%s} -> %2.5f\n", i,
59 67           paste(sprintf("%2.4f", tmp[[1]]), collapse=" "),
60 68           tmp[[2]]))
69 +       if ( save.every > 0 && i %% save.every == 0 ) {
70 +         ok <- try(write.csv(clean.hist(hist[seq_len(i)]),
71 +           save.file, row.names=FALSE))
72 +         if ( inherits(ok, "try-error") )
73 +           warning("Error while writing progress file (continuing)",
74 +             immediate.=TRUE)
75 +       }
61 76     }
62 77     hist
63 78   }
```

Questions?