Building R Package Tutorial

Amy Yang |Senior Data Scientist |UPTAKE
Twitter @ayanalytics

CSP 2019

Pre-work

- R (version >3.5)
- Rstudio

You need the following packages

devtools

roxygen2

usethis

Optional: Install Git, Create a Github account

https://happygitwithr.com/install-git.html#install-git

Introduction

Packages are the fundamental units of reproducible R code.

They include reusable **R functions**, the **documentation** that describes how to use them, (and sample data).

In this tutorial you’ll learn how to turn your code into packages that others/your future self can easily reuse.

Essentials

- Why write an R package
- The minimal R package structure
- Writing documentation with roxygen2
- Building and installing an R package

Why write an R package

1. **Code Organization**: An R package would help in organizing where your utility functions go. Easier tracking → more likely to reuse them
2. **Consistent documentation**: What does this function do? What's the input and output of the function? An R package provides a great consistent documentation structure and encourages you to document your functions.
3. **Code Distribution**: No more emailing .R scripts! An R package gives an easy way to distribute your code to other users.
4. **It *really* is quite simple to create.** The documentation format is no longer a big pain and barrier with the help of Roxygen2.

The minimum R package structure

Step 0: Packages you will need

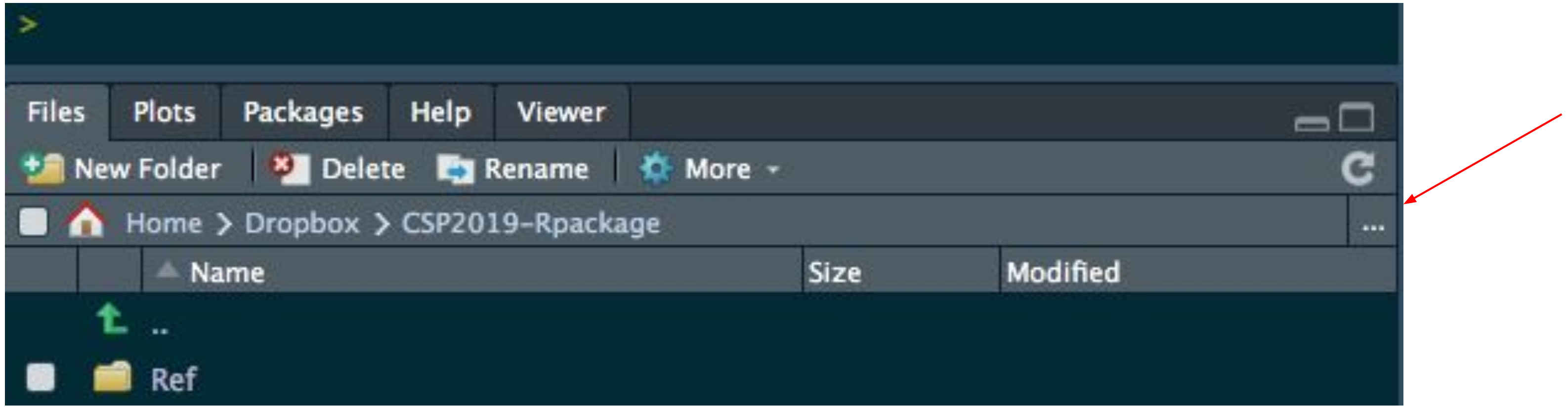
The packages you will need to create a package are **devtools**, **roxygen2** and **usethis**.

```r
install.packages("devtools")
library(devtools)
install.packages("roxygen2")
library(roxygen2)
install.packages("usethis")
library(usethis)
```

Step 1: Create your package directory

You are going to create a directory with the bare minimum folders of R packages. I am going to make a package that calculates zscores for a column and convert the zscores into 5 categories as an illustration.

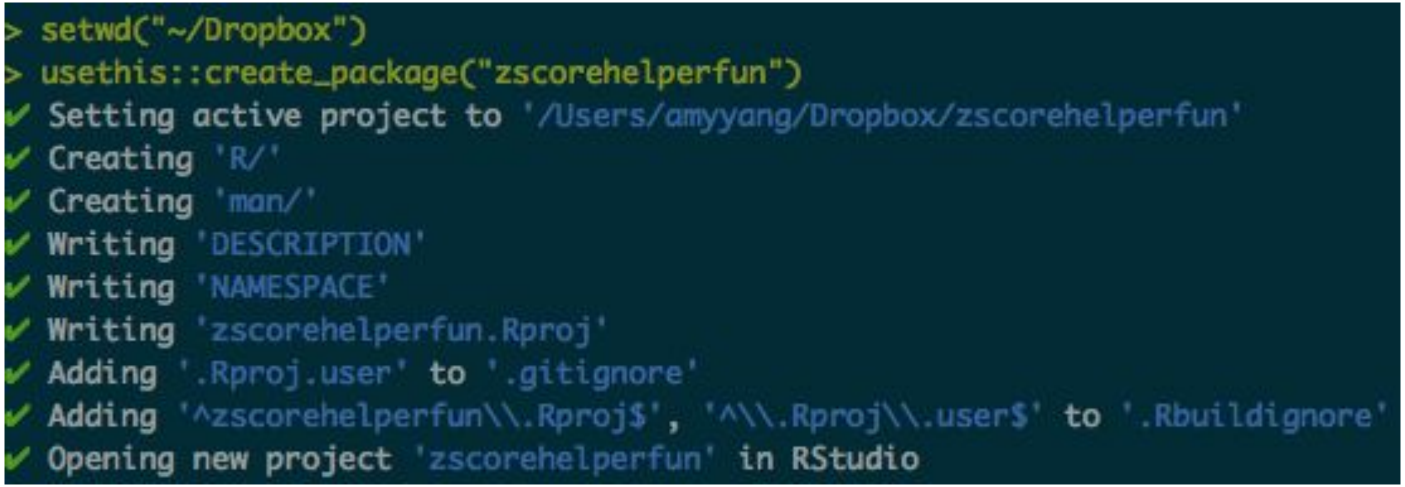
- Navigate to directory where you want to save this project. E.g. ~/Dropbox/CSP2019-Rpackage



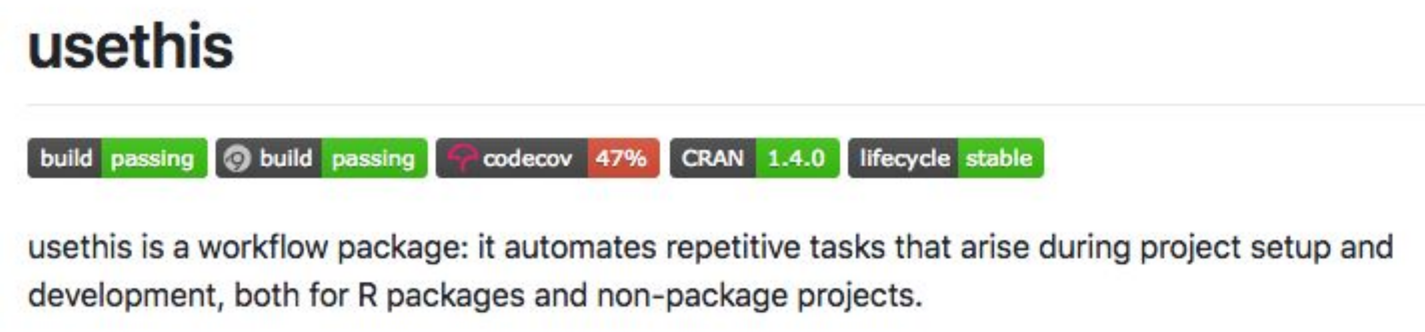
Step 1: Create your package directory

```r
setwd("~/Dropbox/CSP2019-Rpackage")
usethis::create_package("zscorehelperfun")
```

Name of your package



```
> setwd("~/Dropbox")
> usethis::create_package("zscorehelperfun")
✔ Setting active project to '/Users/amyyang/Dropbox/zscorehelperfun'
✔ Creating 'R/'
✔ Creating 'man/'
✔ Writing 'DESCRIPTION'
✔ Writing 'NAMESPACE'
✔ Writing 'zscorehelperfun.Rproj'
✔ Adding '.Rproj.user' to '.gitignore'
✔ Adding '^zscorehelperfun\\.Rproj$', '^\\.Rproj\\.user$' to '.Rbuildignore'
✔ Opening new project 'zscorehelperfun' in RStudio
```



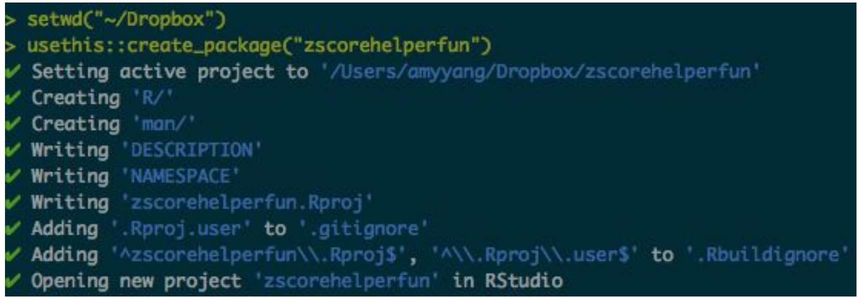
usethis

build passing | build passing | codecov 47% | CRAN 1.4.0 | lifecycle stable

usethis is a workflow package: it automates repetitive tasks that arise during project setup and development, both for R packages and non-package projects.

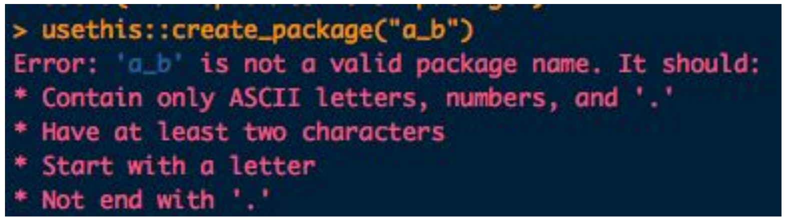
`usethis` is quite chatty, explaining what it's doing and assigning you tasks.

✔ indicates something `usethis` has done for you.



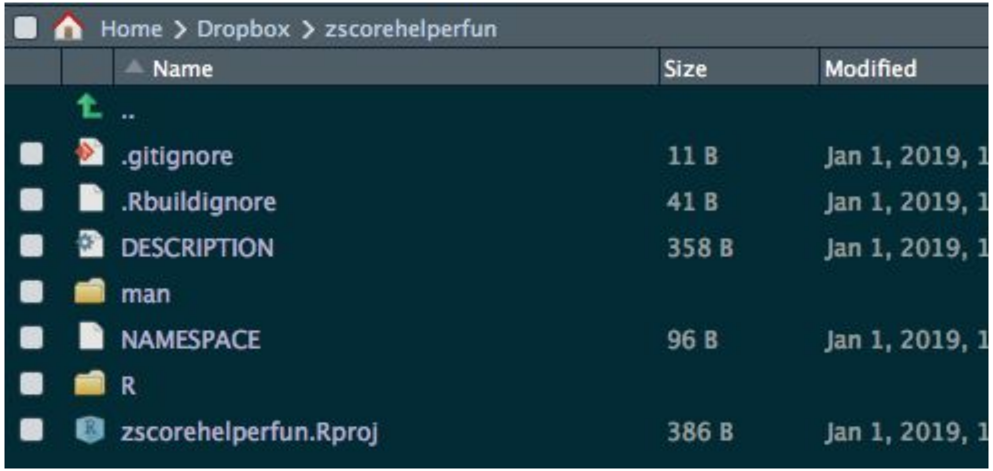
```
> setwd("~/Dropbox")
> usethis::create_package("zscorehelperfun")
✔ Setting active project to '/Users/amyyang/Dropbox/zscorehelperfun'
✔ Creating 'R/'
✔ Creating 'man/'
✔ Writing 'DESCRIPTION'
✔ Writing 'NAMESPACE'
✔ Writing 'zscorehelperfun.Rproj'
✔ Adding '.Rproj.user' to '.gitignore'
✔ Adding '^zscorehelperfun\\.Rproj$', '^\\.Rproj\\.user$' to '.Rbuildignore'
✔ Opening new project 'zscorehelperfun' in RStudio
```

* indicates that you'll need to do some work yourself.



```
> usethis::create_package("a_b")
Error: 'a_b' is not a valid package name. It should:
* Contain only ASCII letters, numbers, and '.'
* Have at least two characters
* Start with a letter
* Not end with '.'
```

This ends up creating a folder with the following files inside the folder:



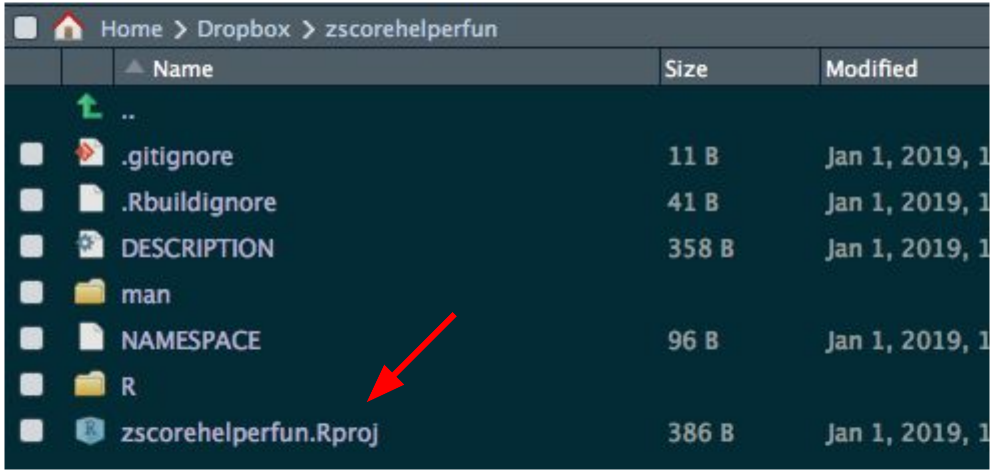
Home > Dropbox > zscorehelperfun

Name	Size	Modified
..		
.gitignore	11 B	Jan 1, 2019, 1
.Rbuildignore	41 B	Jan 1, 2019, 1
DESCRIPTION	358 B	Jan 1, 2019, 1
man		
NAMESPACE	96 B	Jan 1, 2019, 1
R		
zscorehelperfun.Rproj	386 B	Jan 1, 2019, 1

- DESCRIPTION: This is where all the metadata about your package goes.
- zscorehelperfun.Rproj: Indicating this is a Rproject.
- NAMESPACE: This file indicates what needs to be exposed to users for your R package. Automatically generated.
- R/: This is where all your R code goes for your package.
- man/: Manual for documentation files (.Rd format)

You now have the bare bones of your first R package.

What does it mean to be an RStudio Project?

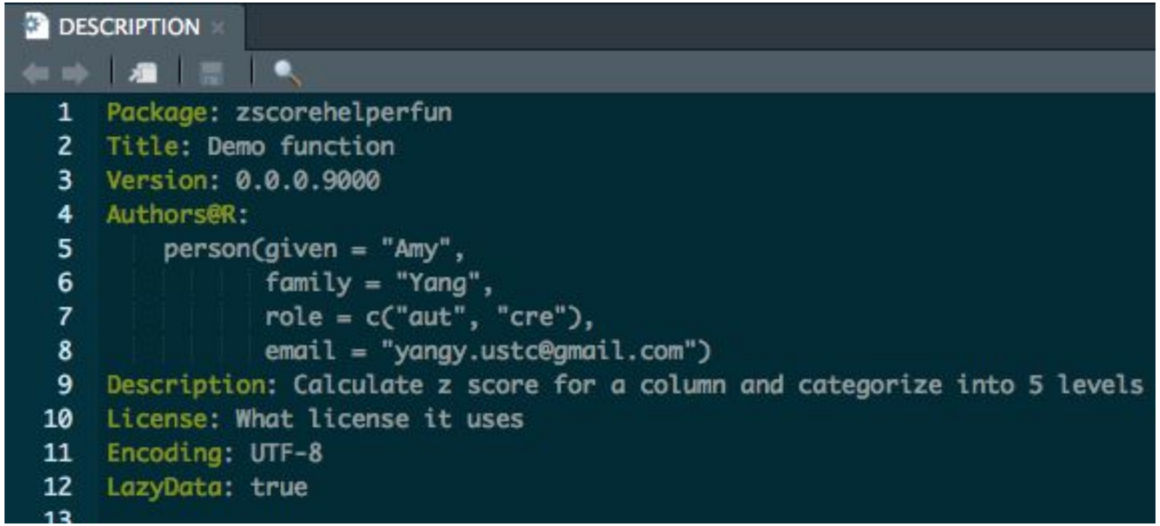


Home > Dropbox > zscorehelperfun

Name	Size	Modified
..		
.gitignore	11 B	Jan 1, 2019, 1
.Rbuildignore	41 B	Jan 1, 2019, 1
DESCRIPTION	358 B	Jan 1, 2019, 1
man		
NAMESPACE	96 B	Jan 1, 2019, 1
R		
zscorehelperfun.Rproj	386 B	Jan 1, 2019, 1

- RStudio leaves notes to itself in foo.Rproj
- Open Project = dedicated instance of RStudio
- File browser pointed at Project directory
- Working directory set to Project directory

Step 2. Edit the DESCRIPTION file



```
Package: zscorehelperfun
Title: Demo function
Version: 0.0.0.9000
Authors@R:
    person(given = "Amy",
           family = "Yang",
           role = c("aut", "cre"),
           email = "yangy.ustc@gmail.com")
Description: Calculate z score for a column and categorize into 5 levels
License: What license it uses
Encoding: UTF-8
LazyData: true
```

Title

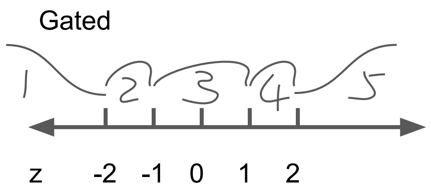
Author

Description

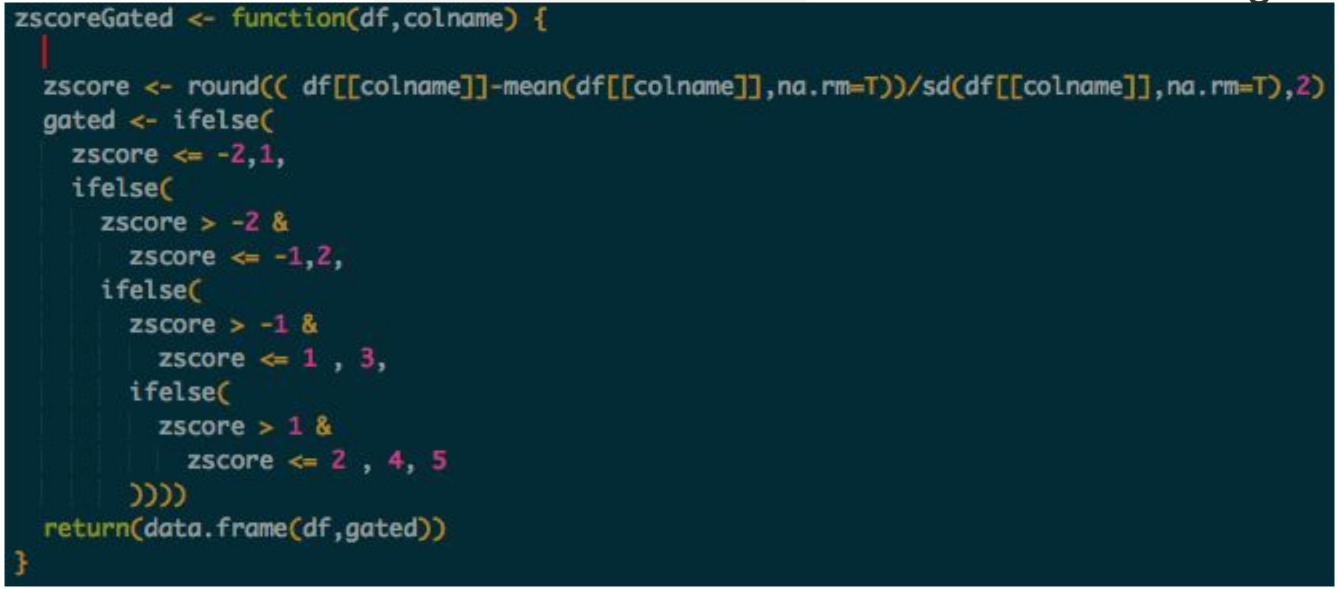
Step 3. Add in your functions

All your R functions that you want in your R package belong in the R directory. You can create an .R file that has the same name as the function you want in it.

For instance, let's create a file called `R/zscoreGated.R` and add the following contents to the file:



zscore=(raw-mean)/SD



```r
zscoreGated <- function(df,colname) {

  zscore <- round(( df[[colname]]-mean(df[[colname]],na.rm=T))/sd(df[[colname]],na.rm=T),2)
  gated <- ifelse(
    zscore <= -2,1,
    ifelse(
      zscore > -2 &
        zscore <= -1,2,
      ifelse(
        zscore > -1 &
          zscore <= 1 , 3,
        ifelse(
          zscore > 1 &
            zscore <= 2 , 4, 5
        ))))
  return(data.frame(df,gated))
}
```

Link to this code: https://github.com/amysheep/CSP2019-Rpackage/blob/master/zscoreGated.R

This is a simple function that takes in a data frame and a column name and calculate zscores for that column and categorize the zscores into 5 levels.

Each .R file can have multiple functions in them. You can do something like this:

```
zscoreGated1 <- function(df, colname){
…
}
zscoreGated2 <- function (...){
…
}
```

In general, try to group together related functions into the same .R file (e.g. if you have multiple plotting functions then putting them in `R/plot.R` would be a good idea).

Step 4. Add Documentation

It's not a proper package until you've added documentation.

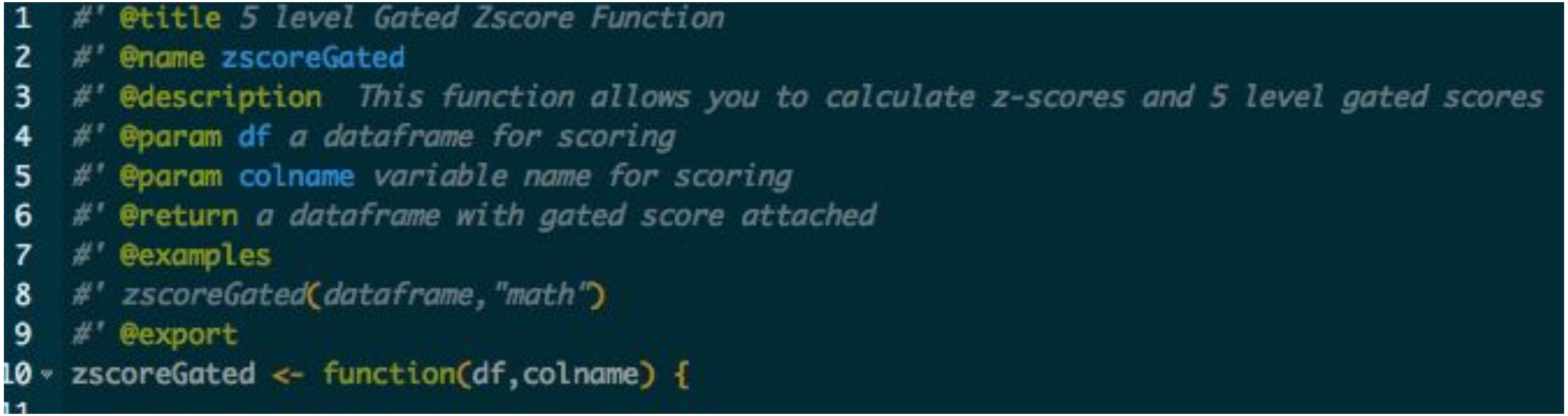
The documentation

- Sits in the `man` subdirectory (`man` for "manual"),
- In `.Rd` (`Rd` for "R documentation") format,
- One `.Rd` file for each function in the package.

Let's checkout the `man` folder for your favourite R package on Github (e.g. usethis)

Roxygen2 is an R package that makes it much easier to create R documentation for your R package.

The comments you need to add at the beginning of the zscoreGated function are, for example, as follows:



```r
#' @title 5 level Gated Zscore Function
#' @name zscoreGated
#' @description  This function allows you to calculate z-scores and 5 level gated scores
#' @param df a dataframe for scoring
#' @param colname variable name for scoring
#' @return a dataframe with gated score attached
#' @examples
#' zscoreGated(dataframe,"math")
#' @export
zscoreGated <- function(df,colname) {
```

The Roxygen2 comments are just R comments (preceded by `#`), but you need to use `#'` to distinguish the Roxygen2 comments from any normal R comments. Special tag @

The `@export` line is critical. This tells Roxygen2 to add this function to the `NAMESPACE` file, so that it will be accessible to users.

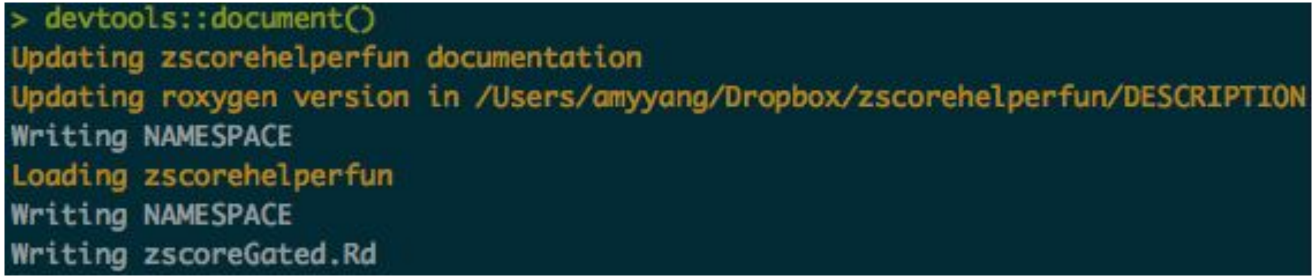
Step 5. Processing the Roxygen2 comments

Now you need to create the documentation from your annotations earlier.

- In R console, run

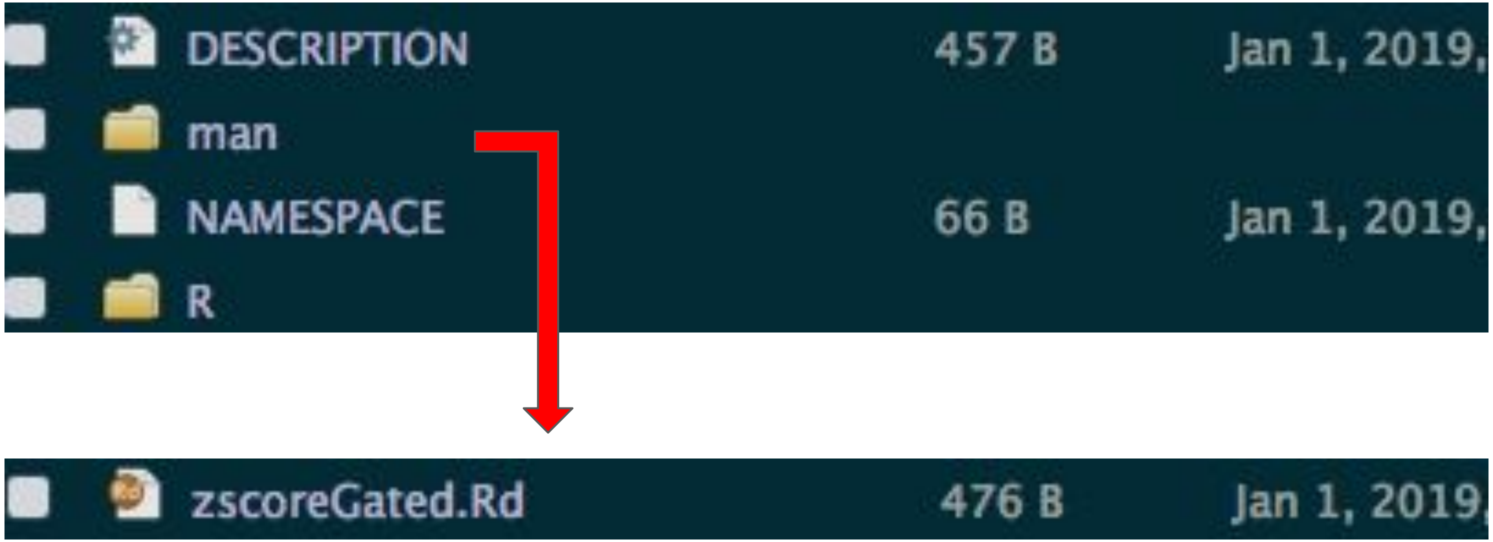
```
devtools::document()
```

You will see something like this:



```
> devtools::document()
Updating zscorehelperfun documentation
Updating roxygen version in /Users/amyyang/Dropbox/zscorehelperfun/DESCRIPTION
Writing NAMESPACE
Loading zscorehelperfun
Writing NAMESPACE
Writing zscoreGated.Rd
```

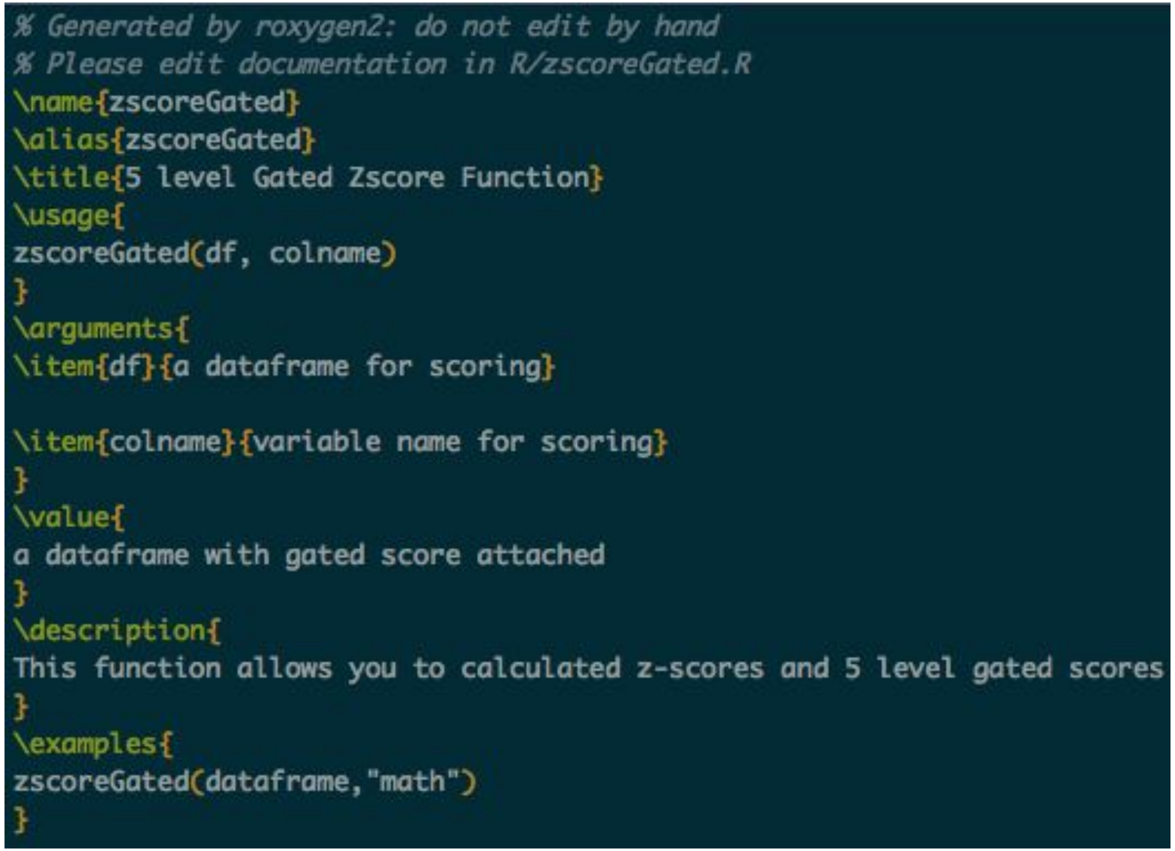
In the `man/` subdirectory you will see an `zscoreGated.Rd` file for the documented function.



```
DESCRIPTION        457 B     Jan 1, 2019,
man
NAMESPACE          66 B      Jan 1, 2019,
R

zscoreGated.Rd     476 B     Jan 1, 2019,
```

zscoreGated.Rd file



```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/zscoreGated.R
\name{zscoreGated}
\alias{zscoreGated}
\title{5 level Gated Zscore Function}
\usage{
zscoreGated(df, colname)
}
\arguments{
\item{df}{a dataframe for scoring}

\item{colname}{variable name for scoring}
}
\value{
a dataframe with gated score attached
}
\description{
This function allows you to calculated z-scores and 5 level gated scores
}
\examples{
zscoreGated(dataframe,"math")
}
```

That's it! Your package is now documented.

You will get one `.Rd` file for each function in your R package.

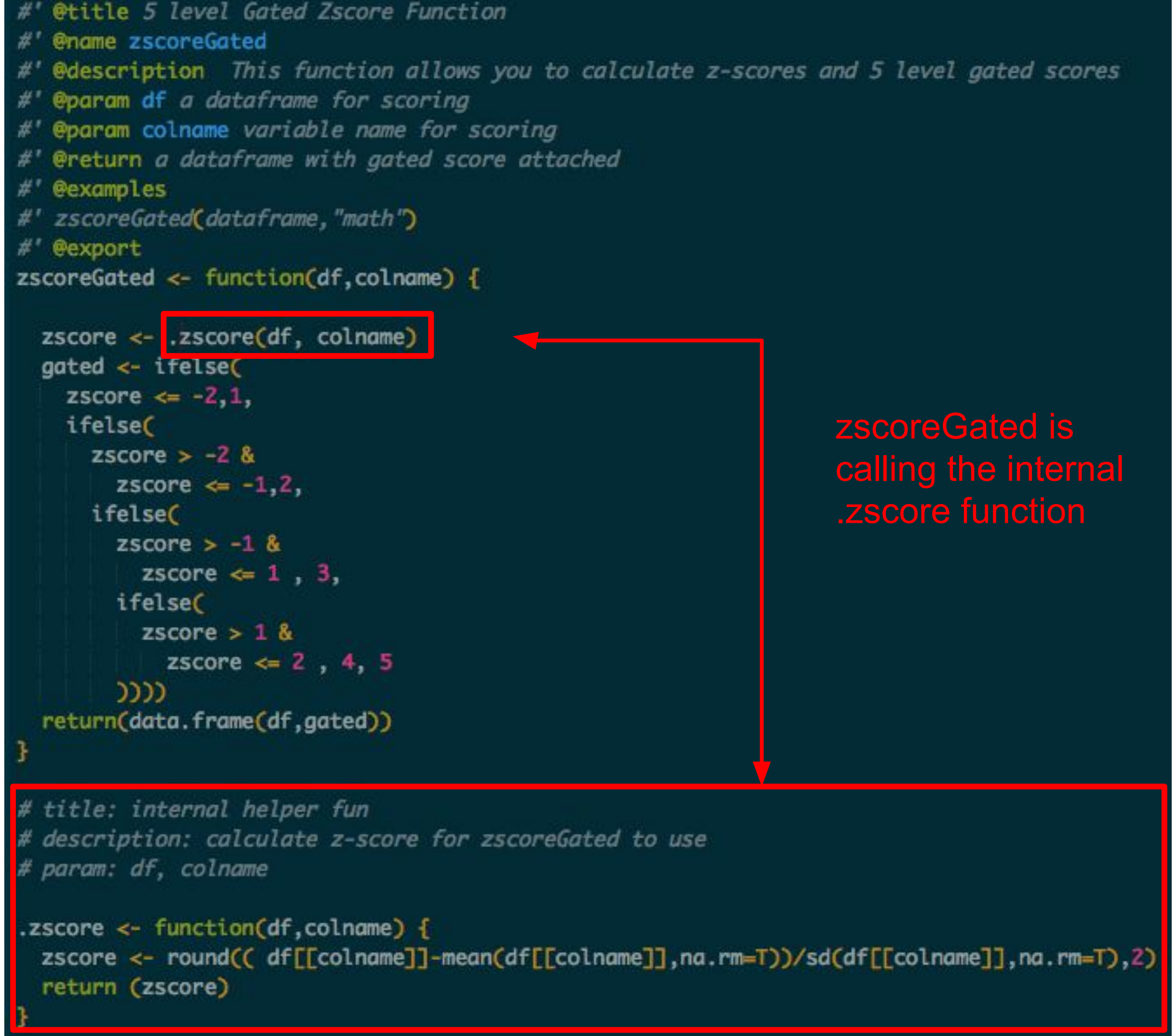
Each time you add new function to your R folder, you need to run `devtools::document()` again to re-generate the .Rd files and refresh NAMESPACE.

What if you have internal functions that are called by your main function which you don't want the user to have access to?

- Use regular R comment # instead of #'
- Start the function name with a dot → .zscore
- Recommend: Rerun devtools::document() to update the .Rd file

Next try add

- @title
- @title and @export



```
#' @title 5 level Gated Zscore Function
#' @name zscoreGated
#' @description  This function allows you to calculate z-scores and 5 level gated scores
#' @param df a dataframe for scoring
#' @param colname variable name for scoring
#' @return a dataframe with gated score attached
#' @examples
#' zscoreGated(dataframe,"math")
#' @export
zscoreGated <- function(df,colname) {

  zscore <- .zscore(df, colname)
  gated <- ifelse(
    zscore <= -2,1,
    ifelse(
      zscore > -2 &
        zscore <= -1,2,
      ifelse(
        zscore > -1 &
          zscore <= 1 , 3,
        ifelse(
          zscore > 1 &
            zscore <= 2 , 4, 5
        ))))
  return(data.frame(df,gated))
}

# title: internal helper fun
# description: calculate z-score for zscoreGated to use
# param: df, colname

.zscore <- function(df,colname) {
  zscore <- round(( df[[colname]]-mean(df[[colname]],na.rm=T))/sd(df[[colname]],na.rm=T),2)
  return (zscore)
}
```

zscoreGated is calling the internal .zscore function

External Dependencies

What if your package depends on other packages e.g. `dplyr`, `data.table` ...

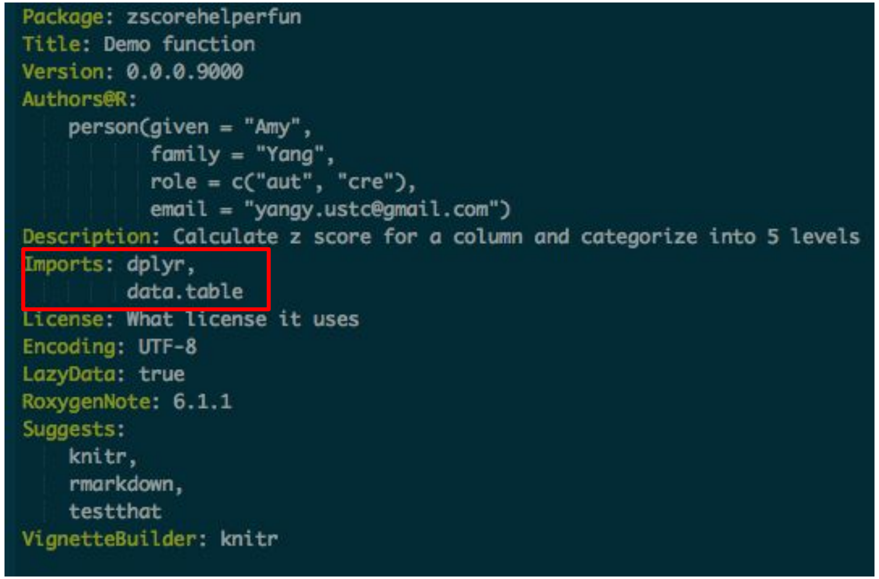
If your R functions require functions from external packages, the way to call them is to use the "double colon" approach.

E.g. `dplyr::select()`

Using `library()` in an R function can globally affect the availability of functions. To reiterate: Do not use `library()` or `require()` in your package.

You also need to indicate this information in your DESCRIPTION file under the **Imports** content.

DESCRIPTION file **Imports examples**



```
Package: zscorehelperfun
Title: Demo function
Version: 0.0.0.9000
Authors@R:
    person(given = "Amy",
           family = "Yang",
           role = c("aut", "cre"),
           email = "yangy.ustc@gmail.com")
Description: Calculate z score for a column and categorize into 5 levels
Imports: dplyr,
         data.table
License: What license it uses
Encoding: UTF-8
LazyData: true
RoxygenNote: 6.1.1
Suggests:
    knitr,
    rmarkdown,
    testthat
VignetteBuilder: knitr
```

- Specify the version of the package e.g. `data.table(>=1.9.4)`
- Multiple external dependencies can be added in a new line
- There is a comma between each package

Step 6 Install

Now that we have:

1. Our functions (.R files) the `R` folder
2. Documentation (.Rd) files in in the `man` folder
3. DESCRIPTION file

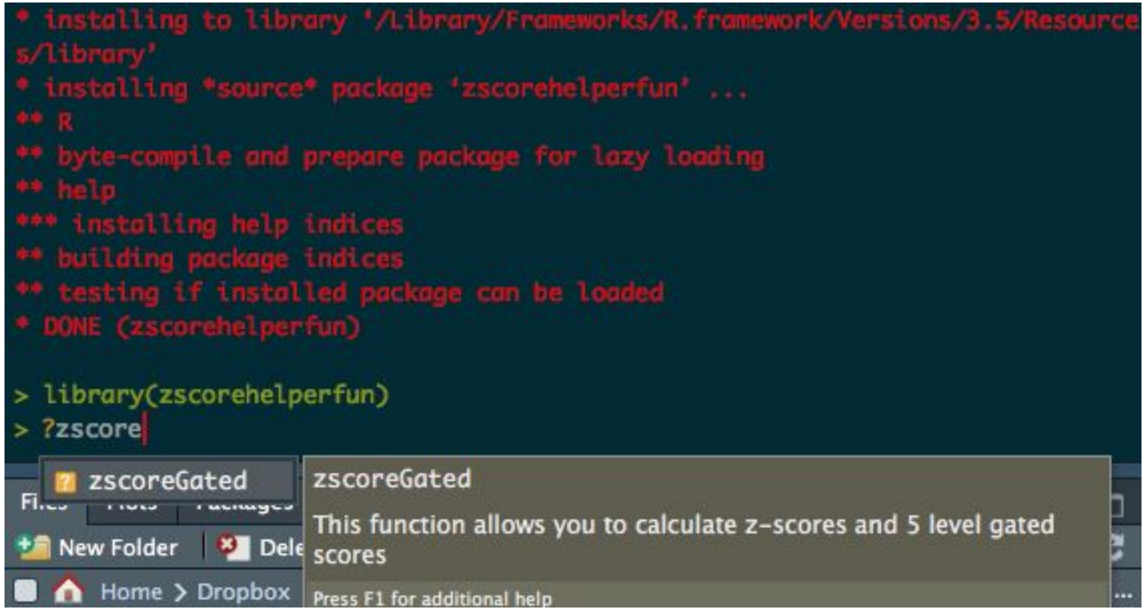
How do we actually install and use our package?

Run `devtools::install()` function in your project directory

This installs your R package into your R system library.

Then you will be able to load up your package with:

```
library(zscorehelperfun)
```



```
* installing to library '/Library/Frameworks/R.framework/Versions/3.5/Resource
s/library'
* installing *source* package 'zscorehelperfun' ...
** R
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (zscorehelperfun)

> library(zscorehelperfun)
> ?zscore
```

zscoreGated

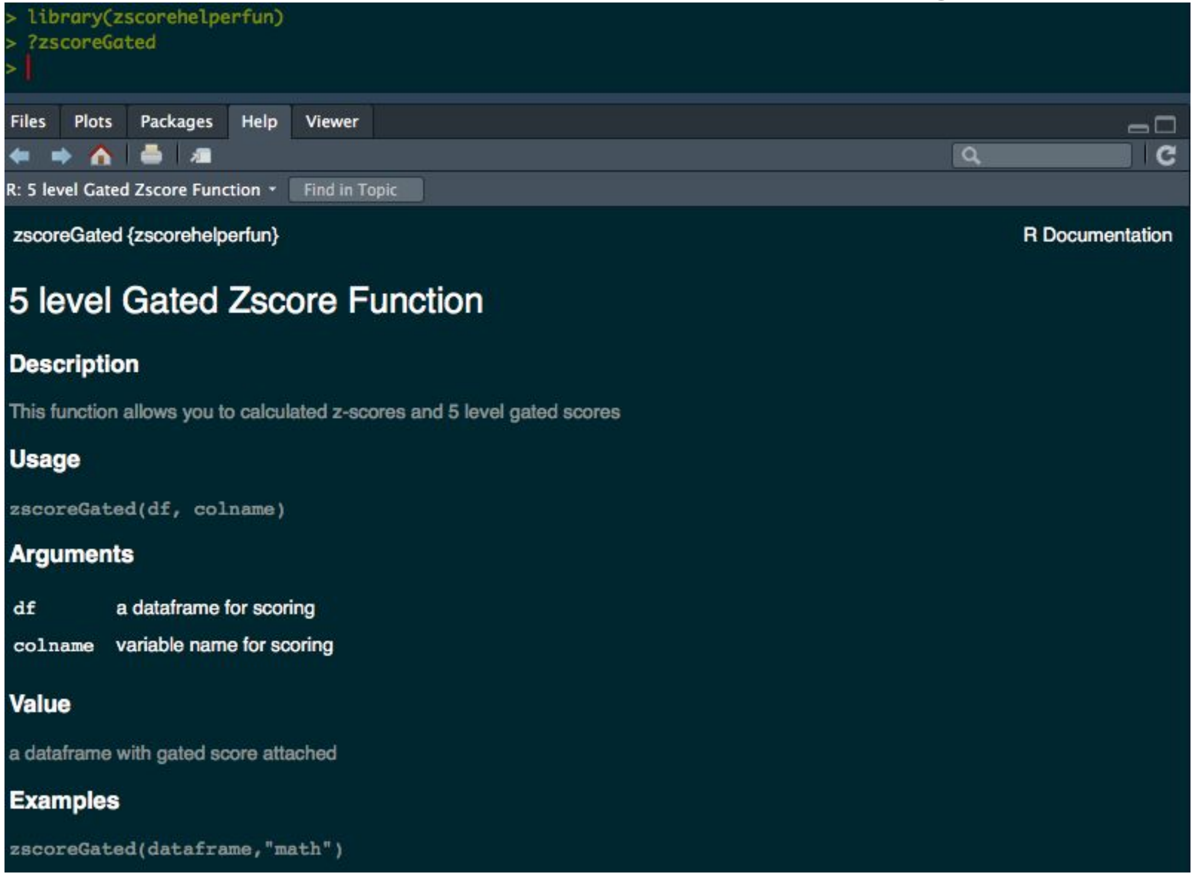
This function allows you to calculate z-scores and 5 level gated scores

Press F1 for additional help

When you type help() or ? with the function name, you will see the function description pop up while you type

Notice the hidden .zscore function is not in the list of available functions (unless you export it)

Now you have a real, live, functioning R package. For example, try typing `?zscoreGated`. You should see the standard help page pop up!



```
> library(zscorehelperfun)
> ?zscoreGated
>
```

zscoreGated {zscorehelperfun} R Documentation

5 level Gated Zscore Function

Description

This function allows you to calculated z-scores and 5 level gated scores

Usage

```
zscoreGated(df, colname)
```

Arguments

`df`	a dataframe for scoring
`colname`	variable name for scoring

Value

a dataframe with gated score attached

Examples

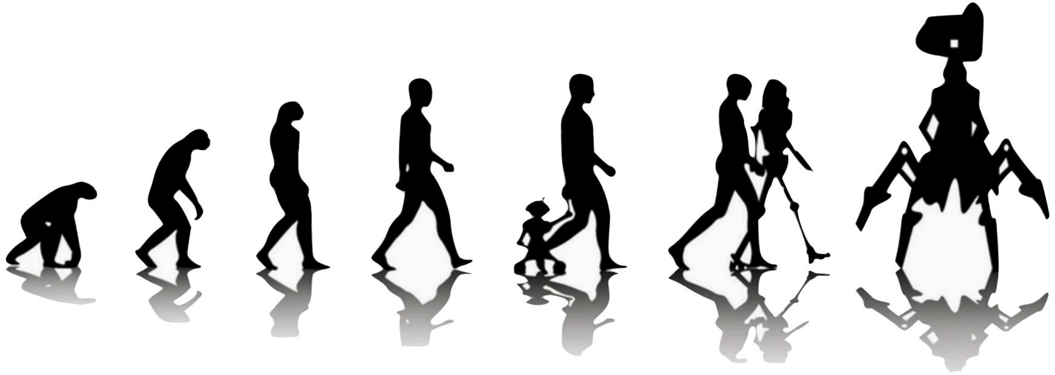
```
zscoreGated(dataframe,"math")
```

Bonus: The following are important but not essential.

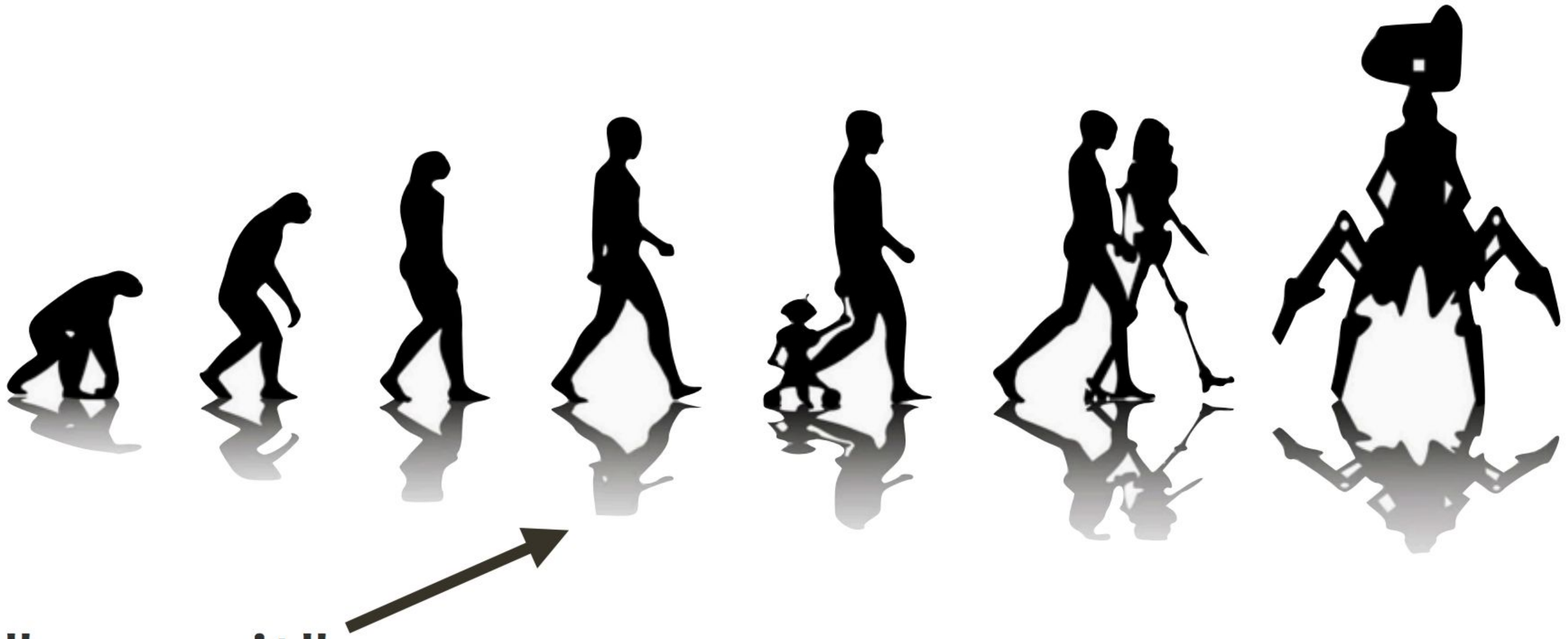
- Putting it on GitHub/Distribute your package
- Writing vignettes
- Writing tests
- Including datasets

Putting your R package on GitHub

If you are developing an R package, or for almost everything you do on a computer, you need a system for keeping track of the changes to the source code.



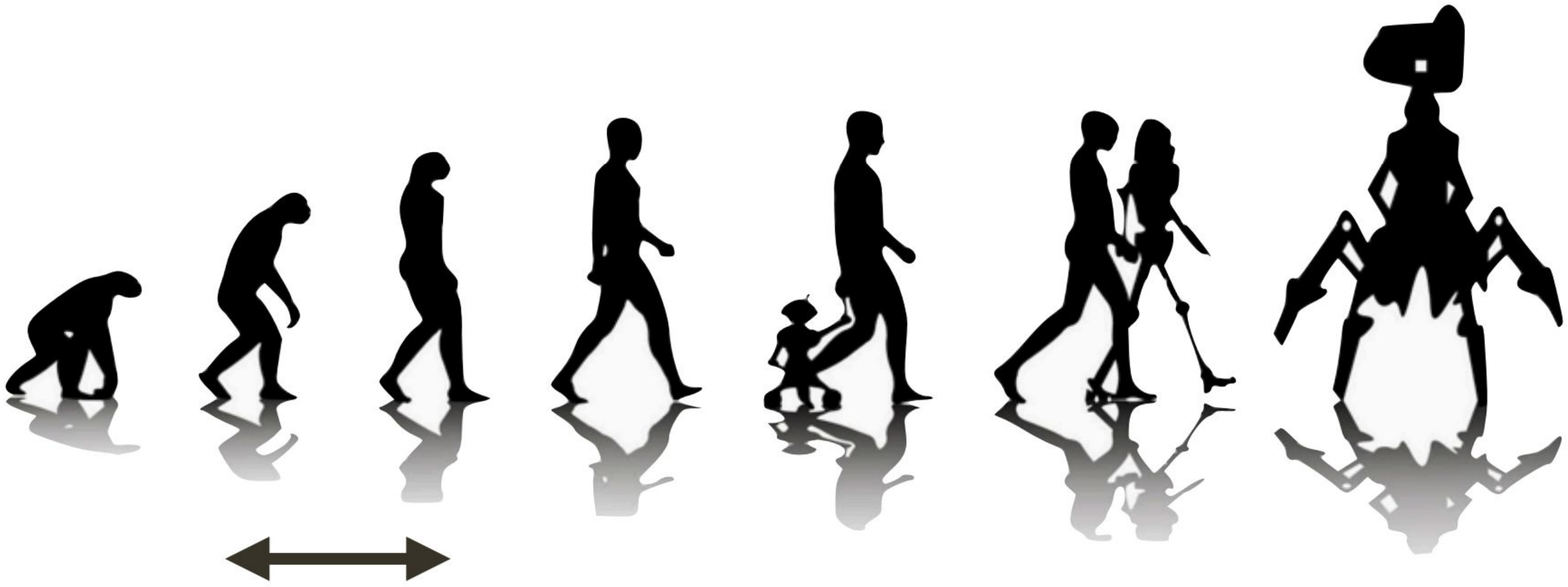
use version control



"commit"

a file or project state that is **meaningful to you**

for inspection, comparison, restoration

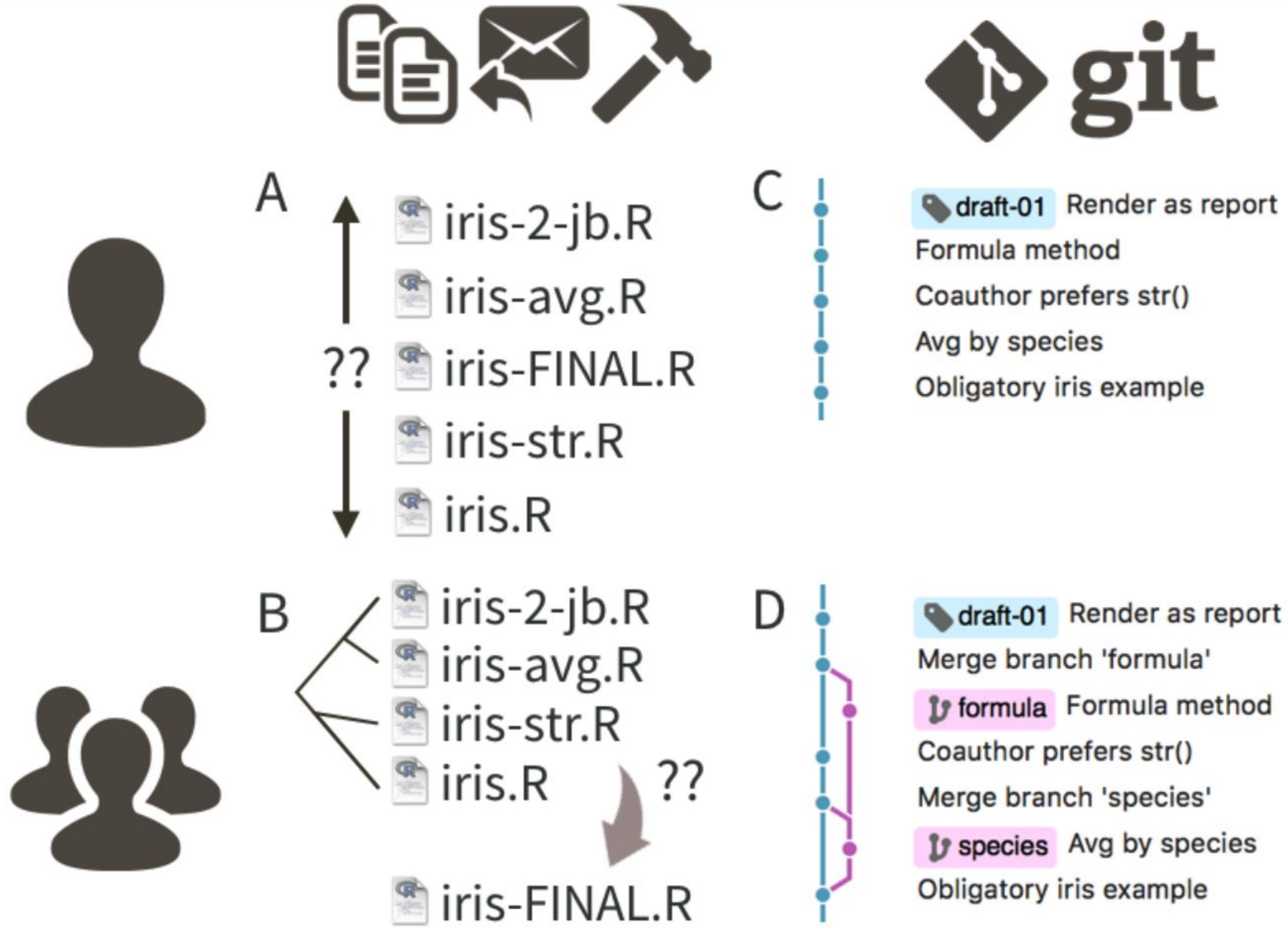


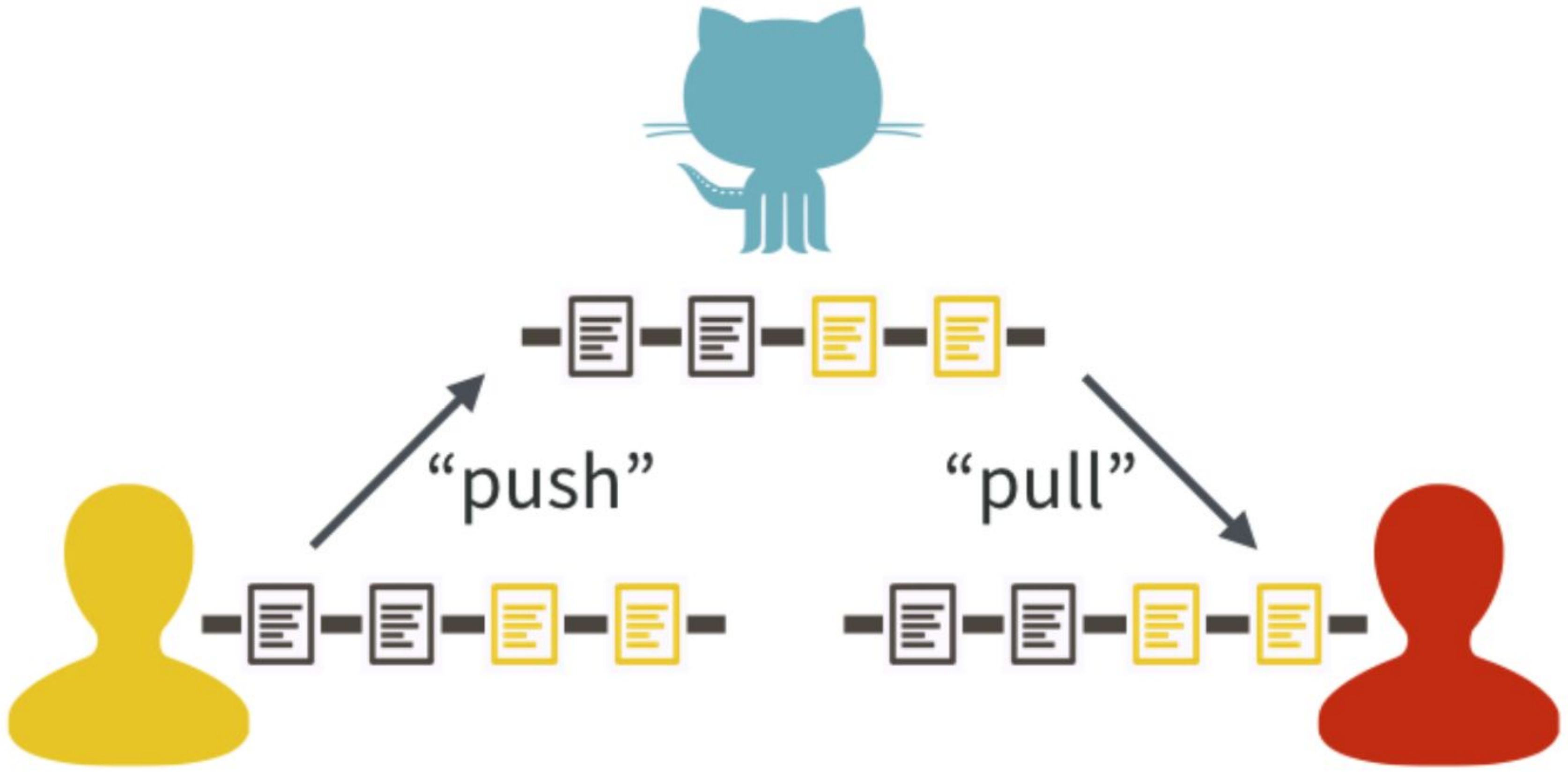
Δ

"diff"

What changed here?

Why?


git
A
iris-2-jb.R
iris-avg.R
??
iris-FINAL.R
iris-str.R
iris.R
B
iris-2-jb.R
iris-avg.R
iris-str.R
iris.R
??
iris-FINAL.R
C
draft-01
Render as report
Formula method
Coauthor prefers str()
Avg by species
Obligatory iris example
D
draft-01
Render as report
Merge branch 'formula'
formula
Formula method
Coauthor prefers str()
Merge branch 'species'
species
Avg by species
Obligatory iris example



collaboration

happygitwithr.com

Why put your R package on GitHub?

- GitHub includes issue tracking: people (including yourself) can note problems they're having or suggestions for improvements they'd like you to make. [First step to Open Source]
- In addition to just pointing out problems, people can actually fix the problem and send you a patch, which you can easily test and then incorporate into your package.
- With the `install_github()` function in devtools package, it's easy for people to install your package directly from GitHub. It doesn't have to be on CRAN.

How to install a package from GitHub

1. First, you need to install the devtools package

2. Load the devtools package. `library(devtools)`

3. In most cases, you just use `install_github("author/package")`.

Exercise: Pick a package from Github and install the dev version of the package

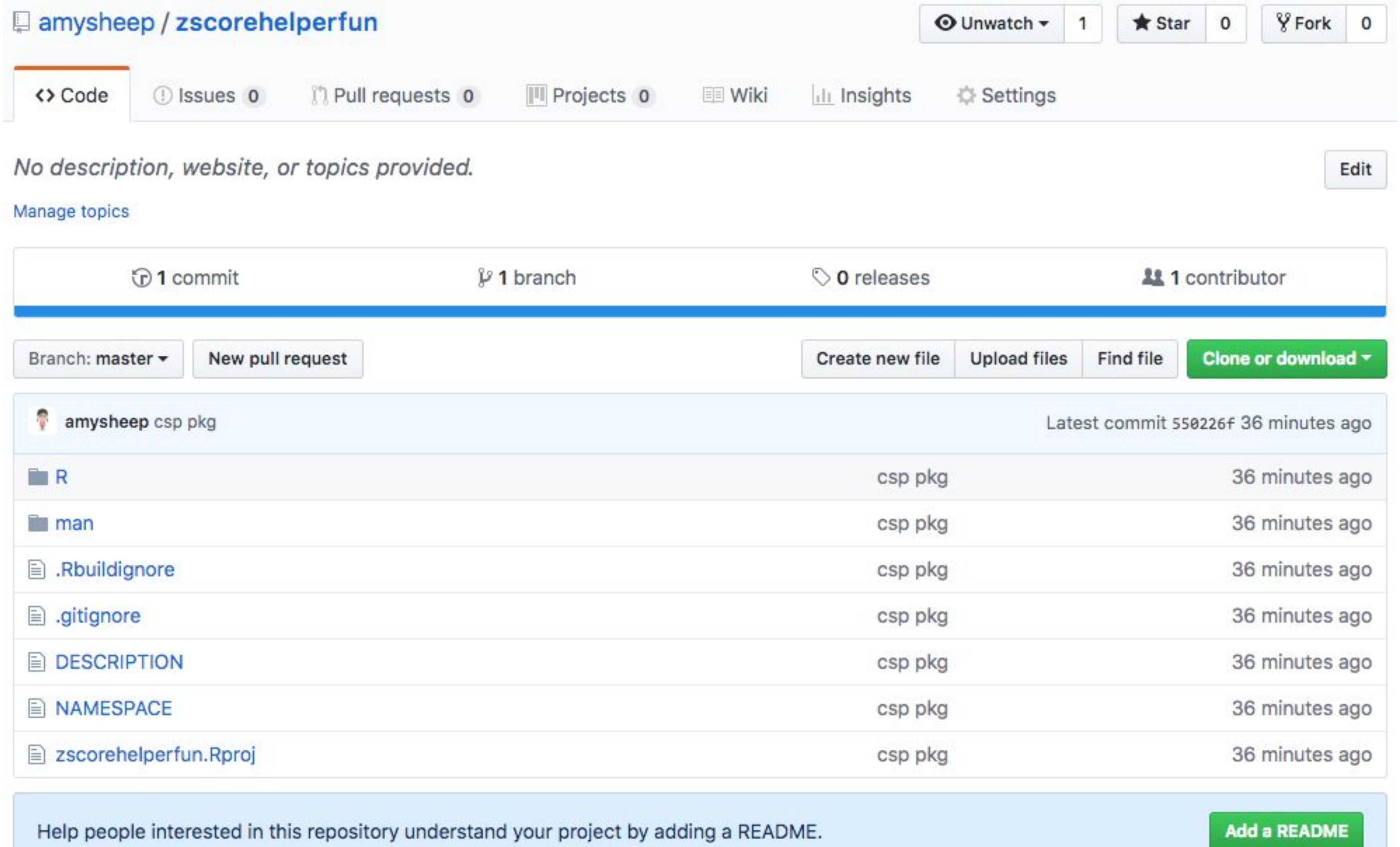
Put your R package on GitHub

To put your package on GitHub, you'll need to get a GitHub account. Then follow the instructions in git/github guide on creating a new git repository. In brief:

- Navigate to the package directory
- Initialize the repository with `git init (from command line)`
- Add and commit everything with `git add .` and `git commit`
- Create a new repository on GitHub
- Connect your local repository to the GitHub one
- `git remote add origin https://github.com/username/reponame`
- Push everything to github
- `git push -u origin master`

Now anyone can install your package using the following command:

```
devtools::install_github("yourusername/myfirstpackage")
```


amysheep / zscorehelperfun

No description, website, or topics provided.

1 commit | 1 branch | 0 releases | 1 contributor

amysheep csp pkg — Latest commit 550226f 36 minutes ago

R	csp pkg	36 minutes ago
man	csp pkg	36 minutes ago
.Rbuildignore	csp pkg	36 minutes ago
.gitignore	csp pkg	36 minutes ago
DESCRIPTION	csp pkg	36 minutes ago
NAMESPACE	csp pkg	36 minutes ago
zscorehelperfun.Rproj	csp pkg	36 minutes ago

Help people interested in this repository understand your project by adding a README.

Writing vignettes

It’s important to write good and clear documentation, but users don’t often read it

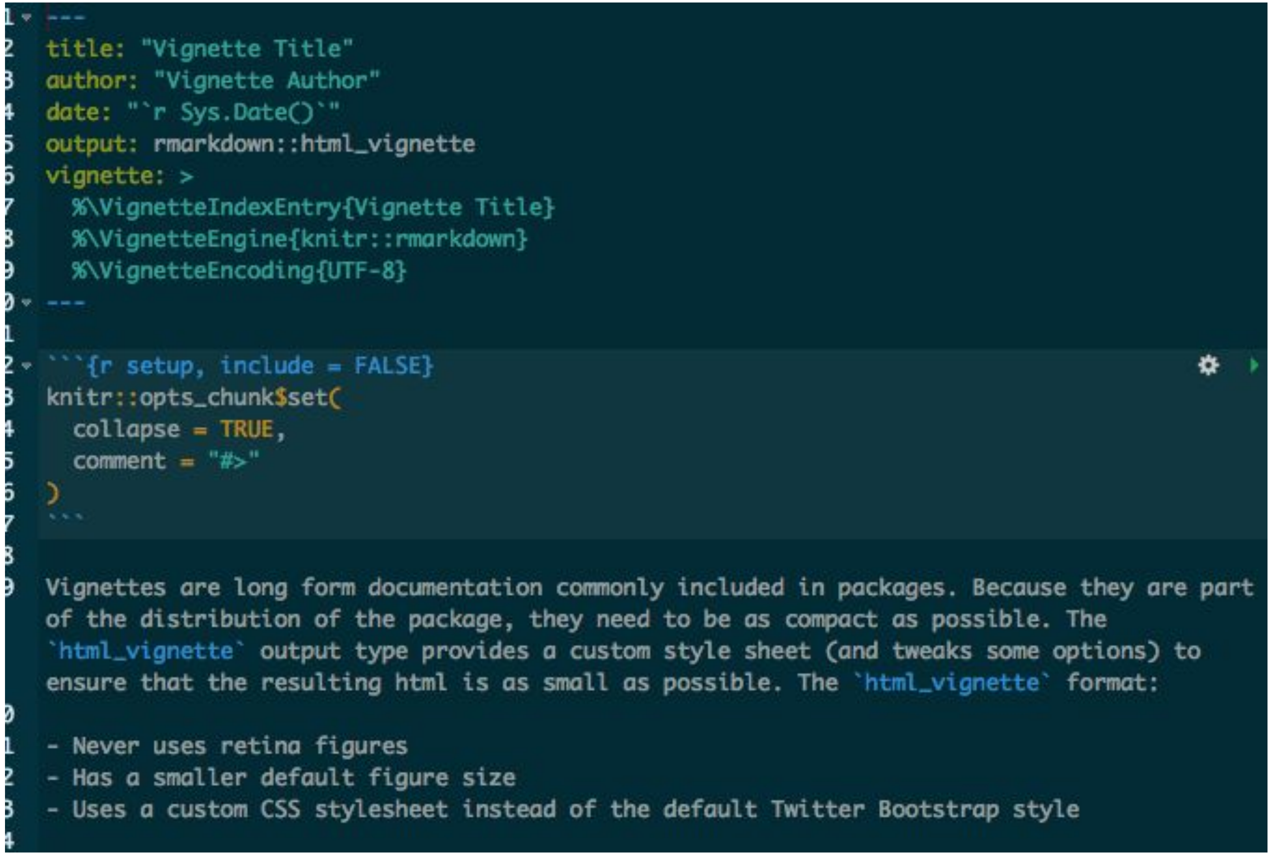
what users want are instructive tutorials demonstrating practical uses of the package with discussion of the interpretation of the results.

In R packages, such tutorials are called “vignettes.”

To get started with generating a vignette, you can use the `usethis::use_vignette()` function for this. For instance,

```
usethis::use_vignette("introduction")
```

This will create a `vignette/introduction.Rmd` file. This is a vignette template Rmarkdown file that you can then use to fill out steps on how one can use the package.



~~~
---
title: "Vignette Title"
author: "Vignette Author"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  %\VignetteEncoding{UTF-8}
---

```{r setup, include = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>"
)
```

Vignettes are long form documentation commonly included in packages. Because they are part
of the distribution of the package, they need to be as compact as possible. The
`html_vignette` output type provides a custom style sheet (and tweaks some options) to
ensure that the resulting html is as small as possible. The `html_vignette` format:

- Never uses retina figures
- Has a smaller default figure size
- Uses a custom CSS stylesheet instead of the default Twitter Bootstrap style

~~~

Writing tests

Hadley Wickham said it best, in his 2011 paper on his testthat package:

> It’s not that we don’t test our code, it’s that we don’t store our tests so they can be re-run automatically.

How do you know that your code works? You try it out.

How do you know that your code still works a year later?

Well, ideally you saved those initial tests.

Even better, you didn’t just save them, but you structured them in a way that you could run them regularly. This gives you confidence that later changes haven’t broken things that worked.

Test workflow

For proper tests, in which you actually assess whether your code is giving the correct answers, use the testthat package.

Install testthat first-- install.packages("testthat")

To set up your package to use testthat, run:

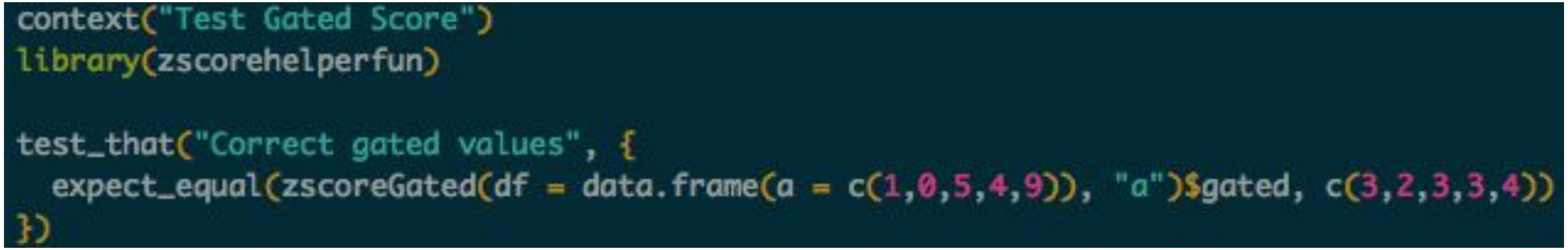
```
usethis::use_testthat()
```

This will:

1. Create a tests/testthat directory.
2. Adds testthat to the Suggests field in the DESCRIPTION.
3. Creates a file tests/testthat.R that runs all your tests when R CMD check runs.

Test structure

A test file lives in `tests/testthat/`. Its name must start with `test`. For example: test-zscoreGated.R



```
context("Test Gated Score")
library(zscorehelperfun)

test_that("Correct gated values", {
  expect_equal(zscoreGated(df = data.frame(a = c(1,0,5,4,9)), "a")$gated, c(3,2,3,3,4))
})
```

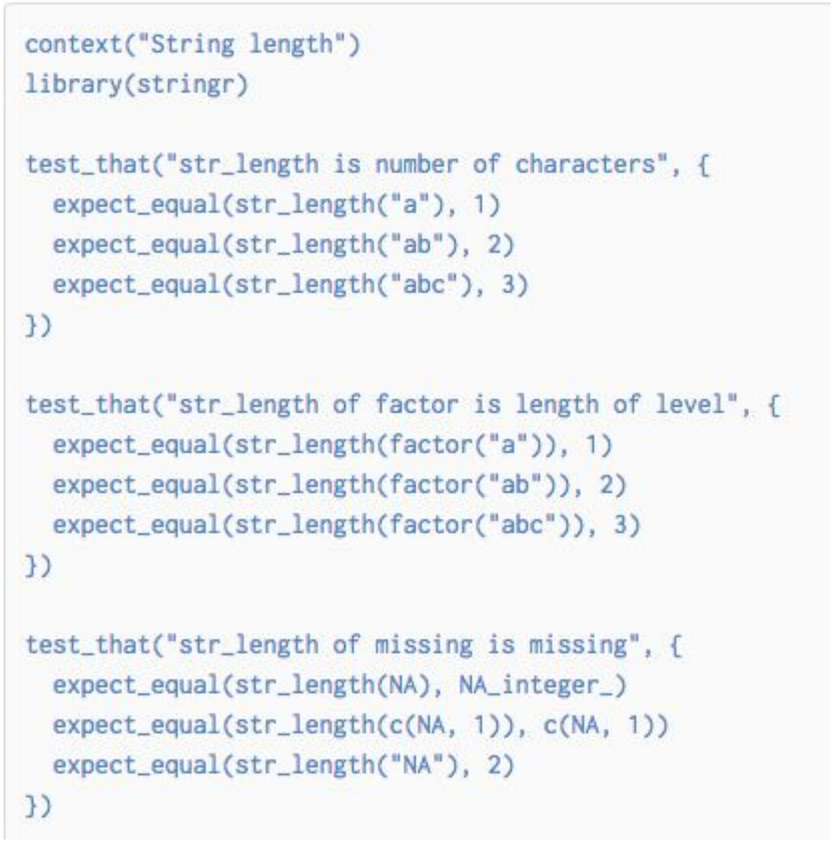
Begin each `test-blah.R` file with context( ); the string here will be printed in the output of `test()`.

Follow this with a series of calls to `test_that()` , each containing a group of related tests. Each of these calls has a character string and then code within `{ }`.

That code will be much like what one always writes in typical informal tests: run a bit of code with known result, and then check if the result actually matches that expected.

More testthat examples

Tests in the stringr package



```
context("String length")
library(stringr)

test_that("str_length is number of characters", {
  expect_equal(str_length("a"), 1)
  expect_equal(str_length("ab"), 2)
  expect_equal(str_length("abc"), 3)
})

test_that("str_length of factor is length of level", {
  expect_equal(str_length(factor("a")), 1)
  expect_equal(str_length(factor("ab")), 2)
  expect_equal(str_length(factor("abc")), 3)
})

test_that("str_length of missing is missing", {
  expect_equal(str_length(NA), NA_integer_)
  expect_equal(str_length(c(NA, 1)), c(NA, 1))
  expect_equal(str_length("NA"), 2)
})
```

The testthat package includes a number of helper functions for checking whether results match expectation.

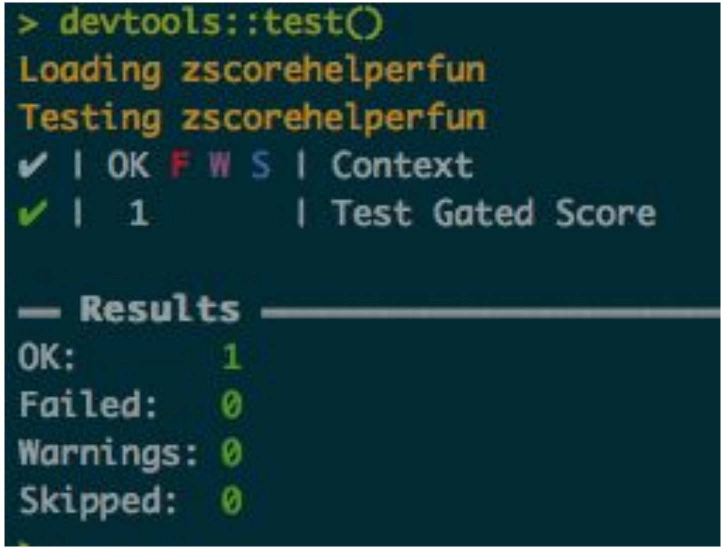
- `expect_error()` if the code should throw an error
- `expect_warning()` if the code should throw a warning
- `expect_equal(a, b)` if `a` and `b` should be the same (up to numeric tolerance)
- `expect_equivalent(a, b)` if `a` and `b` should contain the same values but may have different attributes (e.g., `names` and `dimnames`)

There are a number of others. More details can be found here (http://r-pkgs.had.co.nz/tests.html)

Once you're set up, the workflow is simple:

1. Modify your code or tests.
2. Test your package with `devtools::test()`.
3. Repeat until all tests pass.

The testing output looks like this:



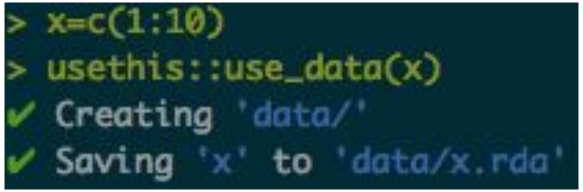
```
> devtools::test()
Loading zscorehelperfun
Testing zscorehelperfun
✔ | OK F W S | Context
✔ |  1       | Test Gated Score

— Results ————————————————————
OK:       1
Failed:   0
Warnings: 0
Skipped:  0
```

Including datasets

It can be useful to include example datasets in your R package, to use in examples or vignettes or to illustrate a data format.

The example data should be stored in the `data/` folder in `.rda` format

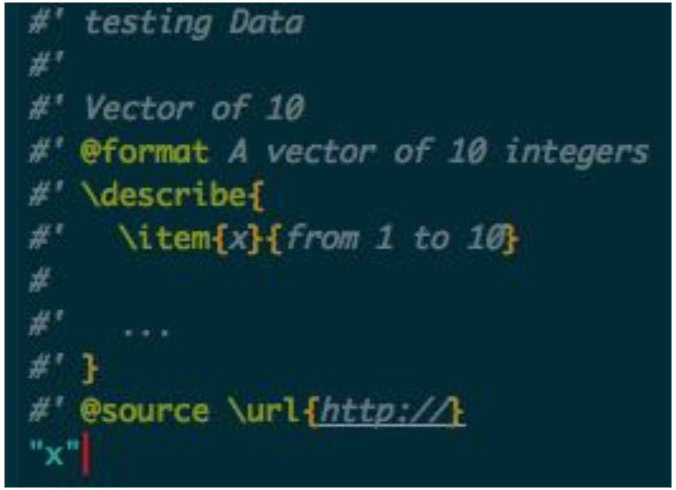
use the `usethis::use_data(yourdata)` function on any R object you have.
E.g.: 

```
> x=c(1:10)
> usethis::use_data(x)
✔ Creating 'data/'
✔ Saving 'x' to 'data/x.rda'
```

This ends up creating and saving the x object into `data/x.rda`. When you load up your package, the x object will be available for you to use.

Documenting datasets

Documenting data is like documenting a function with a few minor differences. Instead of documenting the data directly, you document the name of the dataset and save it in R/. For example, the roxygen2 block used to document the x vector is saved as `R/data.R` and looks something like this:

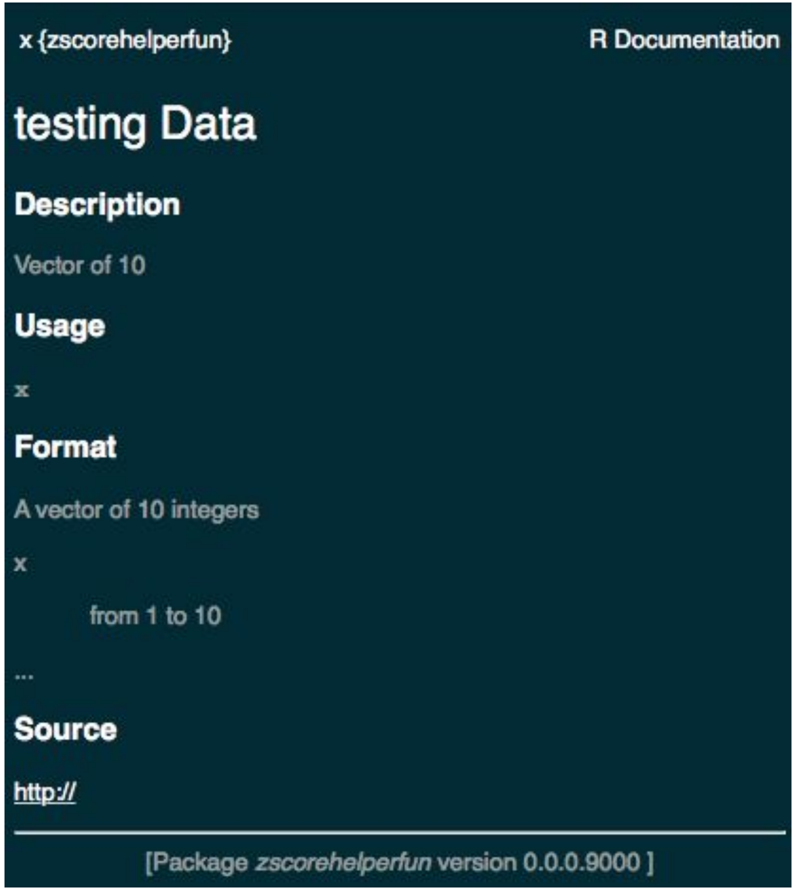


```
#' testing Data
#'
#' Vector of 10
#' @format A vector of 10 integers
#' \describe{
#'   \item{x}{from 1 to 10}
#
#'   ...
#' }
#' @source \url{http://}
"x"
```

There are two additional tags that are important for documenting datasets:

- @format gives an overview of the dataset. For data frames, you should include a definition list that describes each variable. It's usually a good idea to describe variables' units here.
- @source provides details of where you got the data, often a \url{}.

Never @export a data set.


x {zscorehelperfun} R Documentation

testing Data

Description

Vector of 10

Usage

```
x
```

Format

A vector of 10 integers

x

from 1 to 10

...

Source

http://

[Package *zscorehelperfun* version 0.0.0.9000]

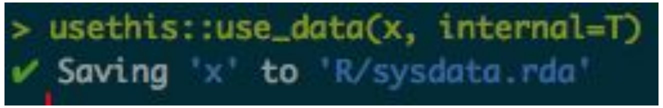
You can use ?x to view the help page of the data

Other types of data

- Internal Data

    Sometimes functions need pre-computed data tables. If you put these in `data/` they'll also be available to package users, which is not appropriate. Instead, you can save them in R/sysdata.rda.

    You can use usethis::use_data() to create this file with the argument internal = TRUE



```
> usethis::use_data(x, internal=T)
✔ Saving 'x' to 'R/sysdata.rda'
```

    Objects in `R/sysdata.rda` are not exported, so they don't need to be documented. They're only available inside your package.

Last Step: infinity Iterate

- Flesh out the documentation as you use and share the package.

- Add new functions the moment you write them, rather than waiting to see if you'll reuse them.

- Divide up the functions into new packages. The possibilities are endless!

R packages can seem like a big, intimidating feat, and they really shouldn't be. The minimum viable R package is a package with just one function!

Further resources

- Hadley Wickham's book about R packages
- Friedrich Leisch's Tutorial on creating R packages (pdf)
- Jeff Leek on developing R packages
- Stat 545 guide to writing an R package
- The official Writing R extensions manual
- A lecture on writing R packages
- Developing R packages with RStudio
- Introduction to roxygen2
- knitr documentation and book
- How to build package vignettes with knitr
- knitr in a knutshell tutorial
- devtools package
- Rd2roxygen package
- testthat package
- A lecture with discussion of software licenses
- Rtools, for building R packages on Windows
- A web service for building and checking R packages for Windows
- R-package-devel email list