R PACKAGE WRITING

Operationalizing your algorithms

WHAT IS A PACKAGE

- The fundamental unit of shareable code
- An R package is a collection of functions that is available to the user once the package is installed on the user's device.
- Will often include function help files, vignettes, sample data, and more.
- Either way, a package is a convenient way to organize and automate commands that you use frequently.

WHY WE USE PACKAGES

- Organizing your commonly used code
- Standardized conventions lead to standardized tools
- Save time
- Sharing code (inside or outside your org)

PREREQUISITES

PACKAGES REQUIRED

- Regardless of your operating system, you will need to install some packages to begin your package-writing career:

```
# make sure that you have the necessary package-writing packages installed
install.packages(c("devtools","roxygen2","testthat","knitr"))
```

PROPER ENVIRONMENT

- If you run Windows, you will need to install `RTools`
- After installation, you will need to verify that your system can find `Rtools`. If your system is set up correctly, the `has_devel()` function should return `TRUE`

```
# make sure that your system knows where to find everything you'll need
devtools::find_rtools()
# make sure that your set up is ready to construct R packages
devtools::has_devel()
```

THINGS TO PACKAGE!

- To create a package we need custom functions
- We will use the **`pv`** and **`rescale`** functions in the package-functions.R script
- We will also use the `pkg_data.csv` data set

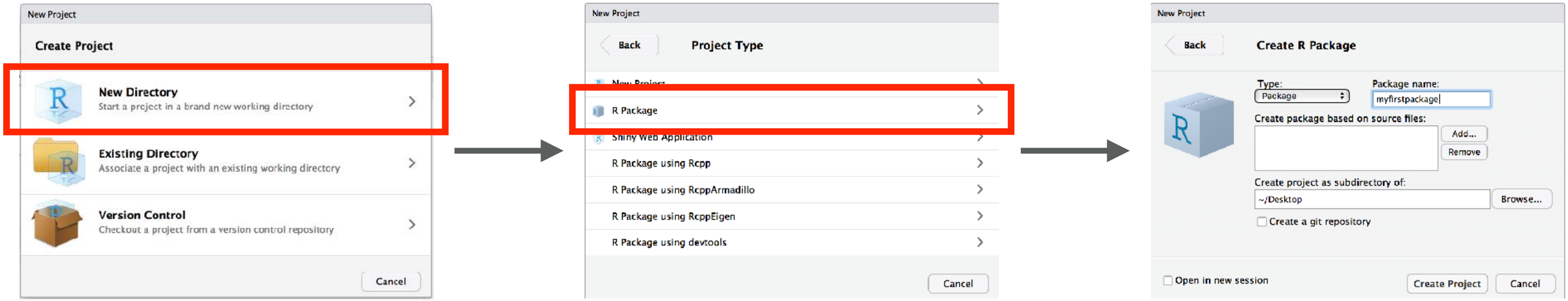
If you are not familiar with creating functions, refer to the back-up slides to get started!

Creating the framework

Every R package has the same basic shell

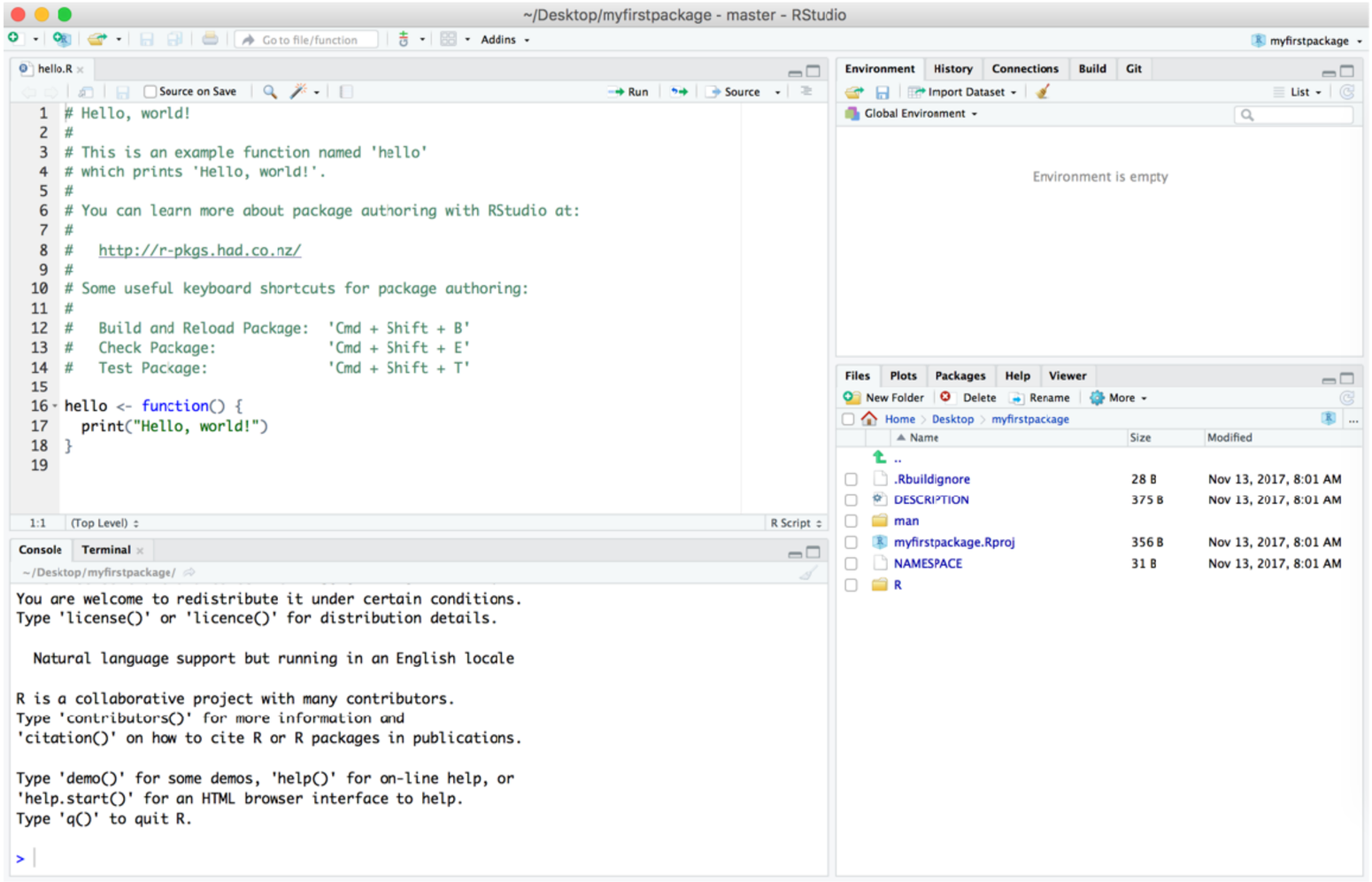
SETTING UP AN R PACKAGE PROJECT

`File >> New Project`



Alternatively: `devtools::create("path/to/package/myfirstpackage")`

INITIAL STRUCTURE



Every R package has at least 3 main components:

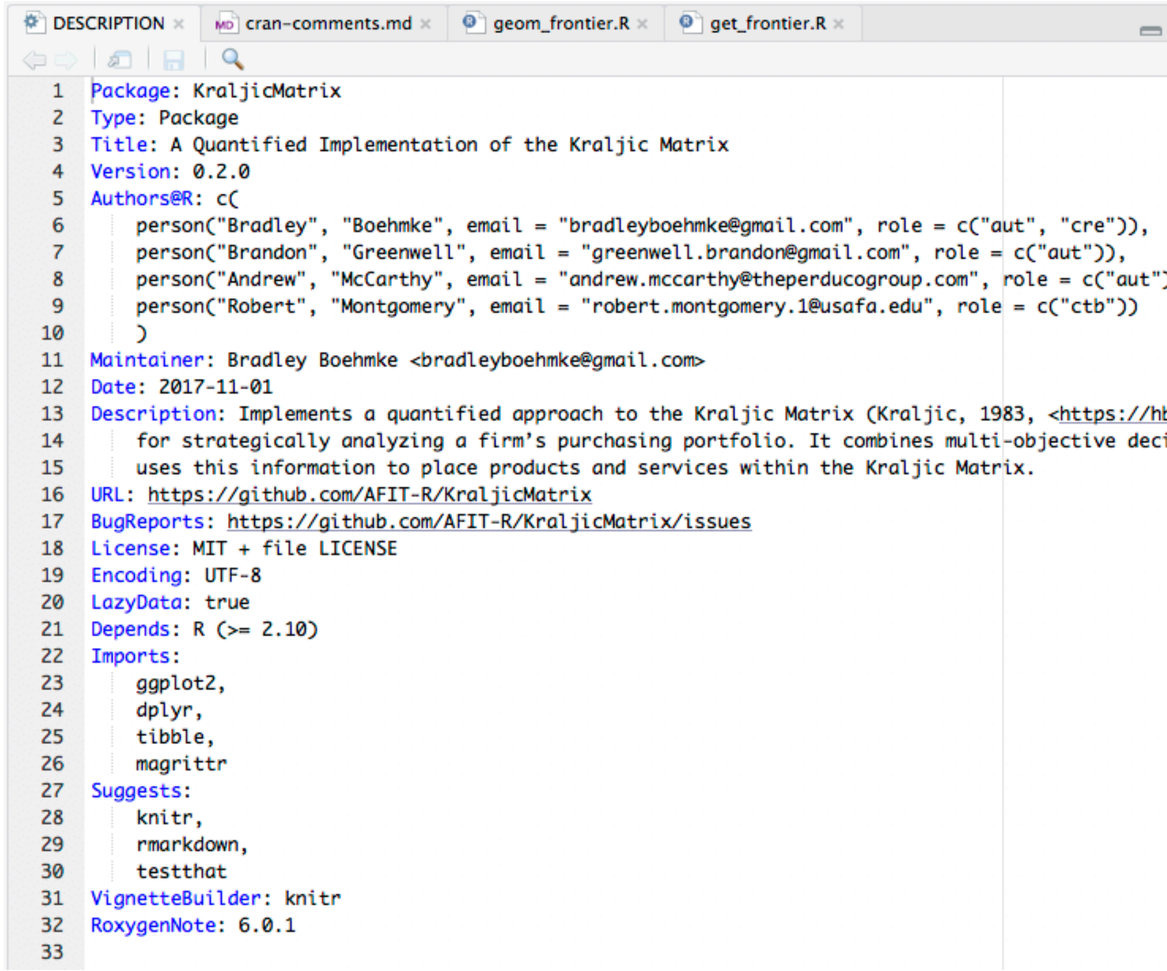
1. DESCRIPTION file
2. R directory
3. NAMESPACE file
    - Go ahead and delete this NAMESPACE file; we'll make a new one later.

DESCRIPTION file

Creating your package metadata

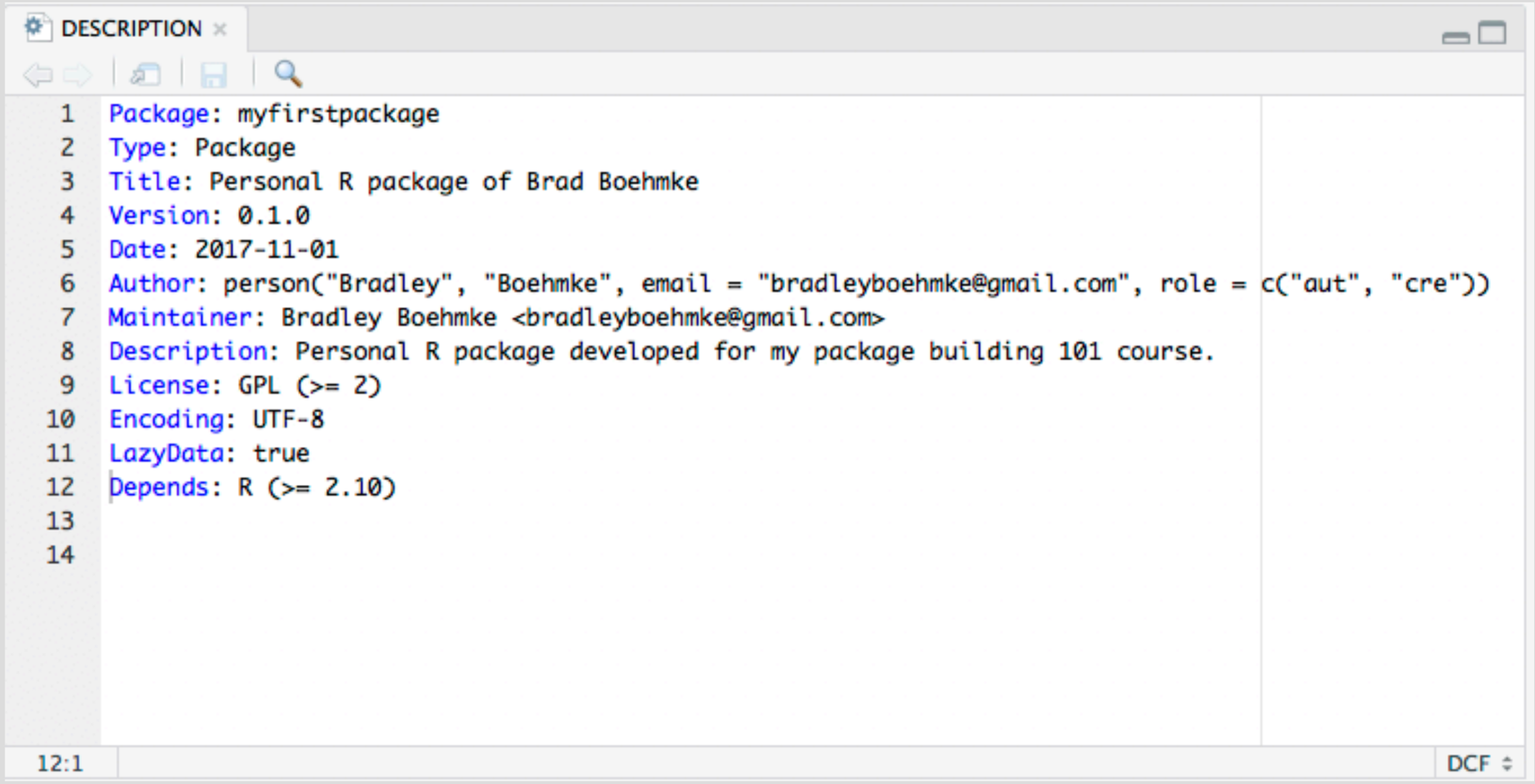
DESCRIPTION FILE

- The DESCRIPTION file is important if you want to distribute your package
- This file is where you make a title for the package, describe briefly what it does, and indicate the author
- Imports must be added; lists packages that are essential for using your functions. These will be automatically installed (but not attached) when your package is installed
- Suggests lists packages that will make your set of functions more useful and effective, but aren't necessary



YOUR TURN!

Update and save your DESCRIPTION file to look something like the following:



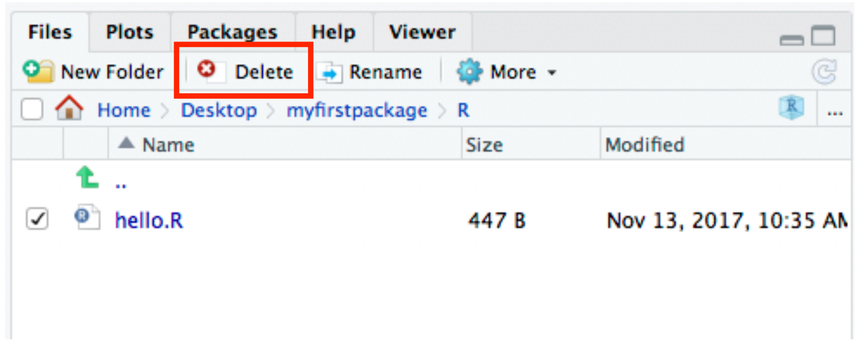
Packaging Functions

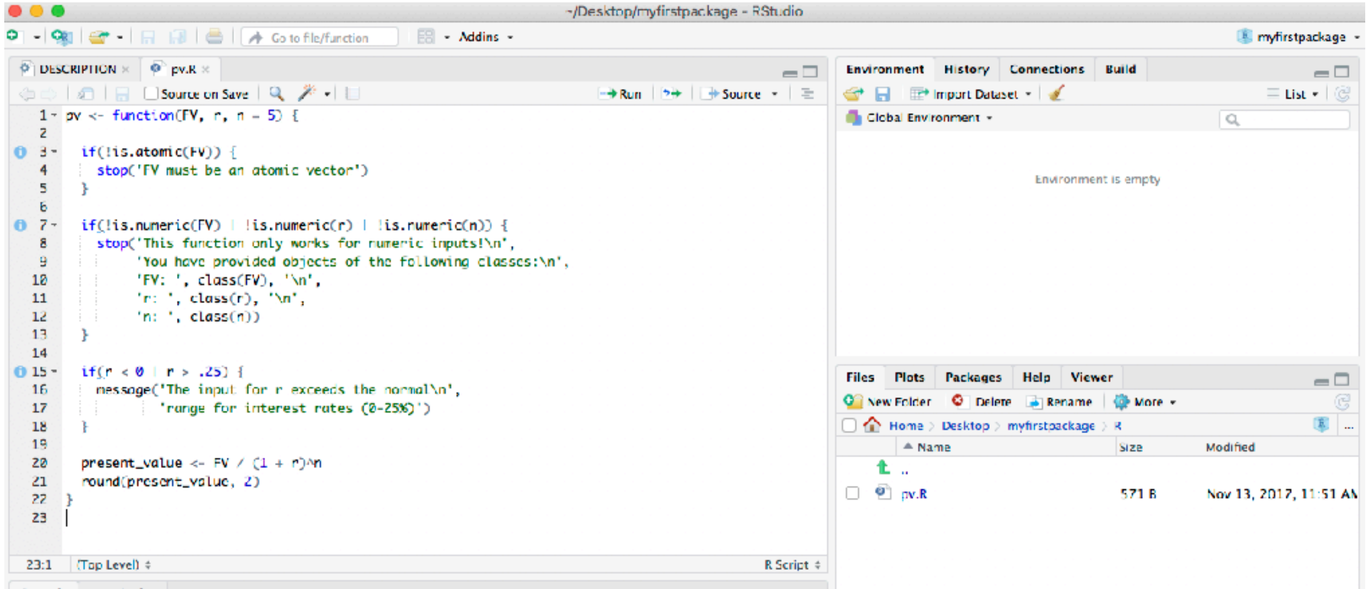
.R Files and the capabilities of your package

WRITING A FUNCTION

- To have a working R Package you need a set of functions (or data)
- Let's work with the **`pv`** and **`rescale`** functions in the package-functions.R script

1. Delete the hello.R file in the R directory & the hello.Rd file in the man directory
2. Create a new .R script, copy the **`pv`** function, and save it as pv.R in the R directory

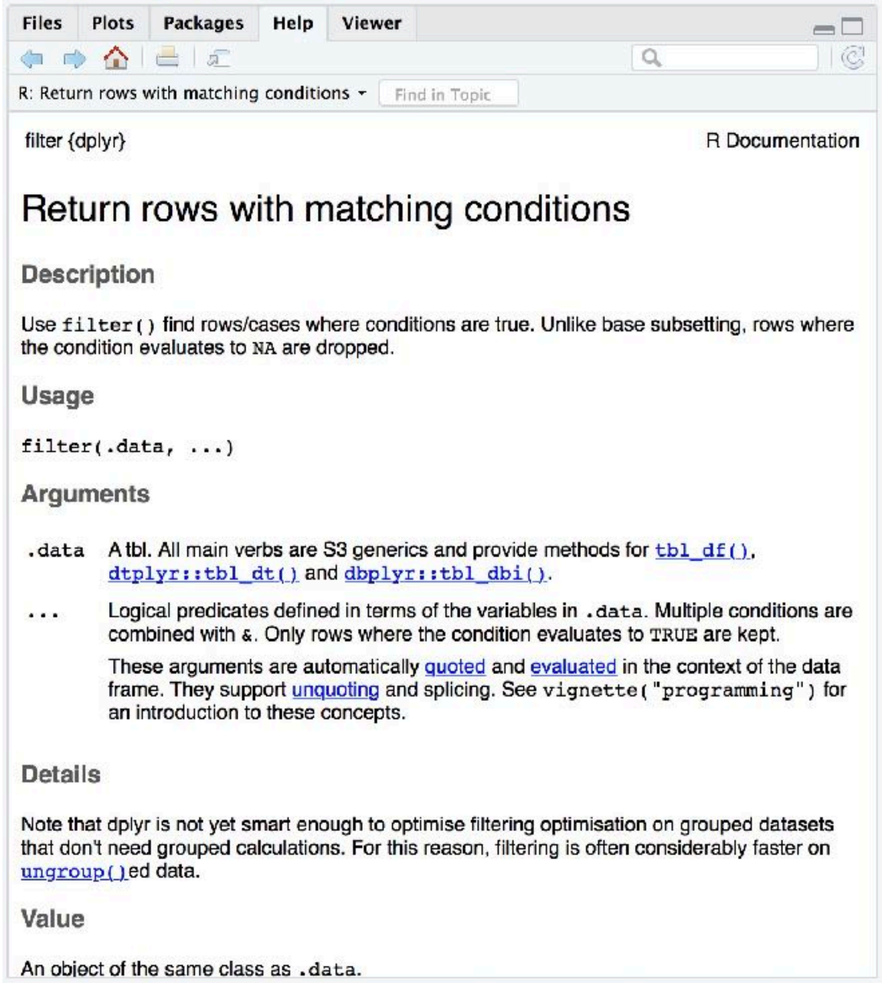




DOCUMENTING FUNCTIONS

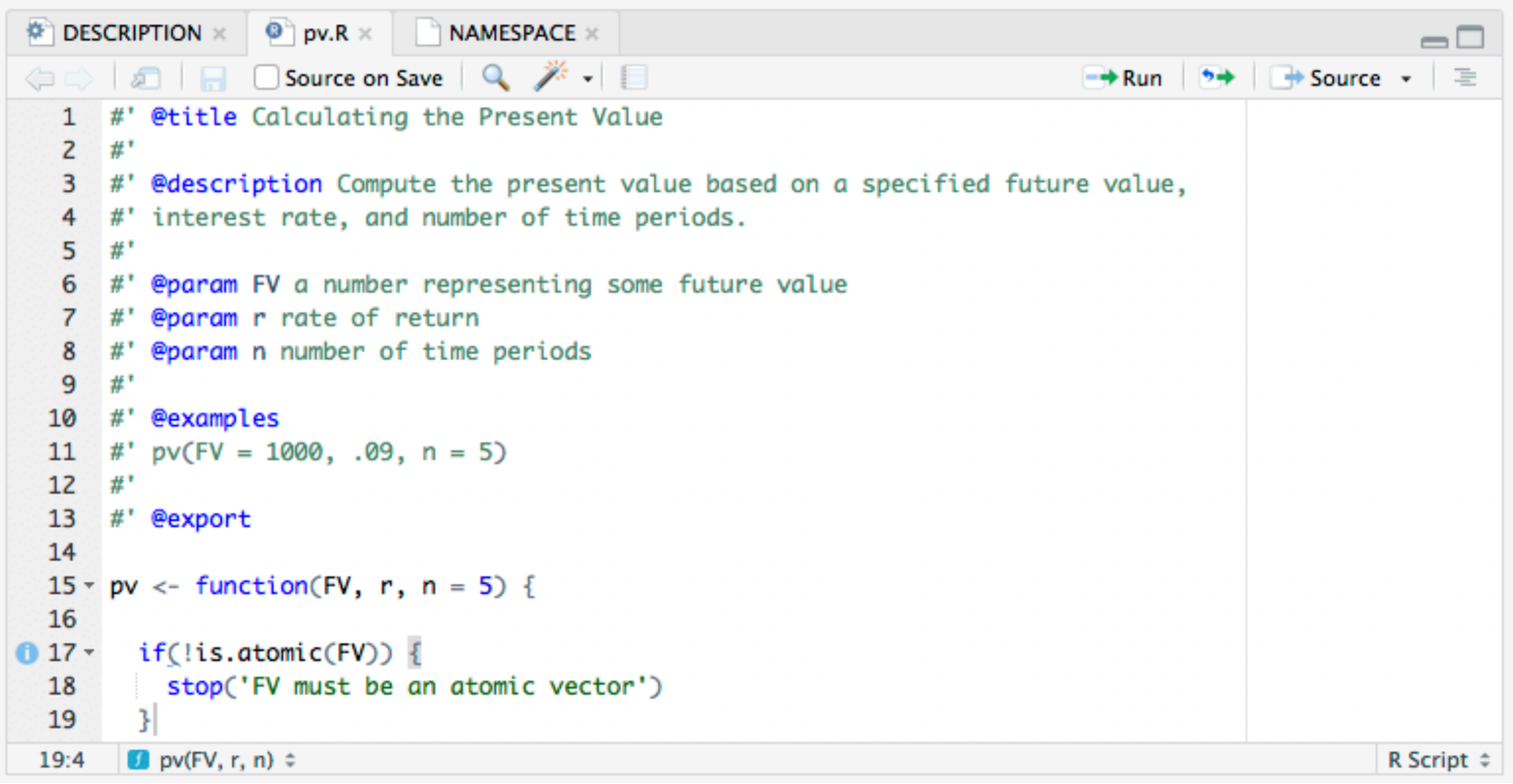
- All functions in R come with help menus so that other users (or even yourself in the future) can figure out how to use various functions
- We will be using the `roxygen2` package to add documentation files
- Documentation files are saved as **.Rd** files in the `/man` folder of your project

```
?dplyr::filter
```



CONSTRUCTING HELP FILES

- Directly above the function in the .R file, add various categories with a leading `#'`
- You should always include `@title`, `@description`, `@param`, `@examples`, and `@export`

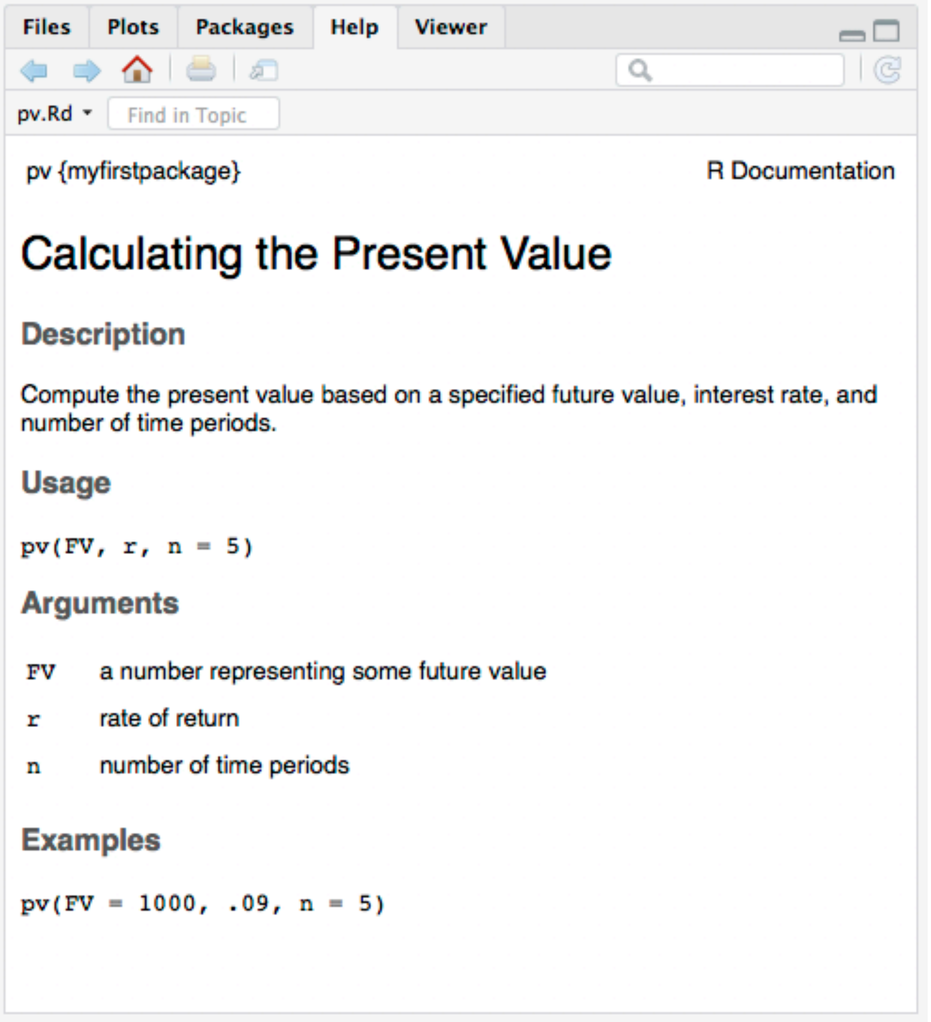


- Once you have incorporated these comments, type `devtools::document()` in your console.
- This creates:
  - .Rd file in man directory
  - Adds `pv` function to package NAMESPACE

TEST DRIVE YOUR FUNCTION

- Once you've created your function and added it to the NAMESPACE you can:

```
# Load your development package functions
devtools::load_all()

# Look at help file documentation
?pv

# Test your function out
pv(FV = 1000, .09, n = 5)
[1] 649.93

# Test that your current package meets CRAN
# requirements
devtools::check()
```



COMMON WORKFLOW

```
# Document your package functions
devtools::document()

# Load your package functions
devtools::load_all()

# Evaluate your package against requirements
devtools::check()
```

Shortcuts:

Ctrl/Cmd + Shift + D

Ctrl/Cmd + Shift + L

Ctrl/Cmd + Shift + E

Rinse and repeat

SOME THINGS TO REMEMBER

- All functions can be written in a single .R script, each function can be in a different script, or some other configuration in between these two extremes
- All of these .R scripts should be saved in the `/R` folder of your project
- Functions can either be internal or exported
  - *Internal functions*, or "helper" functions can only be accessed within the project, unless the user calls `:::` to specify an internal function
  - *Exported functions* are the typical functions that can be accessed anytime once the package has been loaded
- DO NOT call libraries within functions

YOUR TURN!

Add the `rescale` *function to your package:*

1. *Create new .R script*
2. *Add rescale function*
3. *Add roxygen comments to document help file*
4. *Run* `devtools::document()` *to add to NAMESPACE*
5. *Load function with* `devtools::load_all()` *(check our help file, test function)*
6. *Run* `devtools::check()`

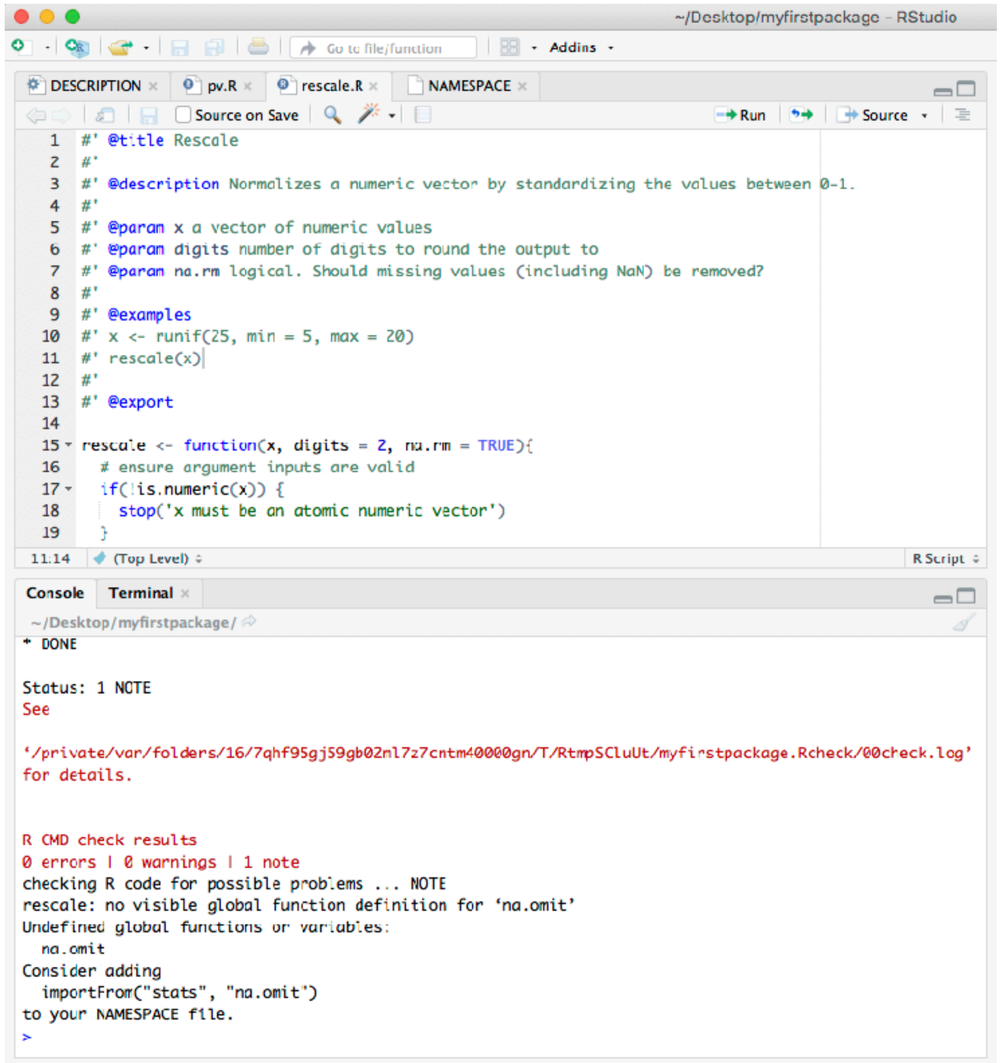
A FUNCTION WITH DEPENDENCIES

- 2 approaches

  1. Add `stats` package to dependencies with `devtools::use_package("stats")`. Then refer to function explicitly (`stats::na.omit`)

  2. Add `stats::na.omit` to NAMESPACE with `@importFrom stats na.omit`

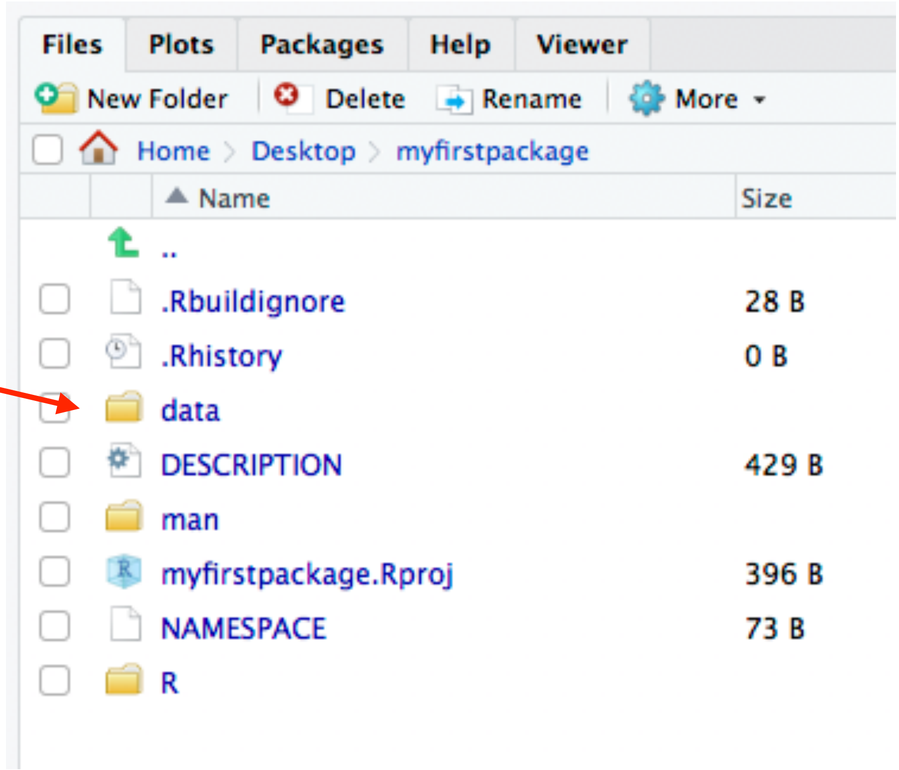
Add this dependency, rerun
`devtools::document()` and `devtools::check()`
and you should be clear!



Adding example data

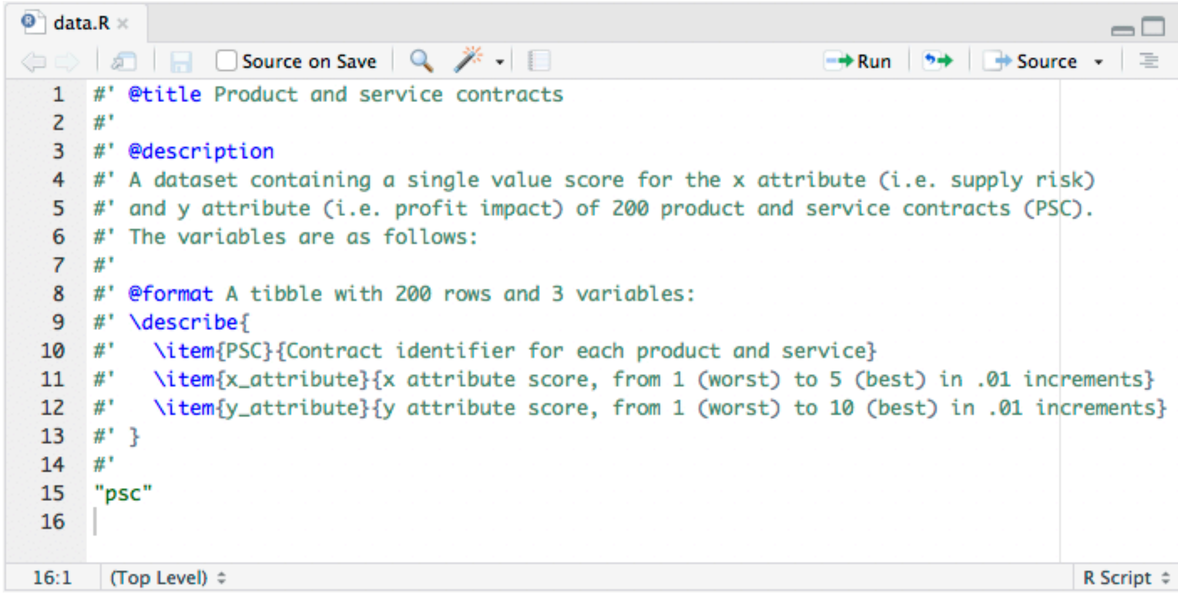
SETTING UP THE DATA

```
# 1. Import the data you want to use
psc <- readr::read_csv("path/to/pkg_data.csv")

# 2. Add the data object to your package
devtools::use_data(psc)
```



SETTING UP THE DATA

```
# 1. Import the data you want to use
psc <- readr::read_csv("path/to/pkg_data.csv")

# 2. Add the data object to your package
devtools::use_data(psc)

# 3. Document data using roxygen comments then run
devtools::document()
devtools::load_all()
?psc
psc
devtools::check()
```

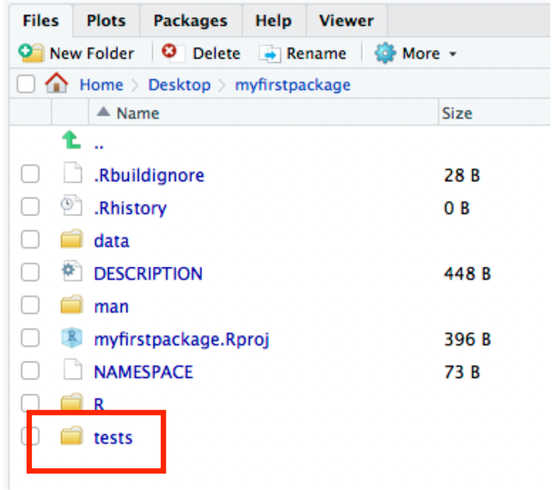


Integrated testing

Know when your code breaks!

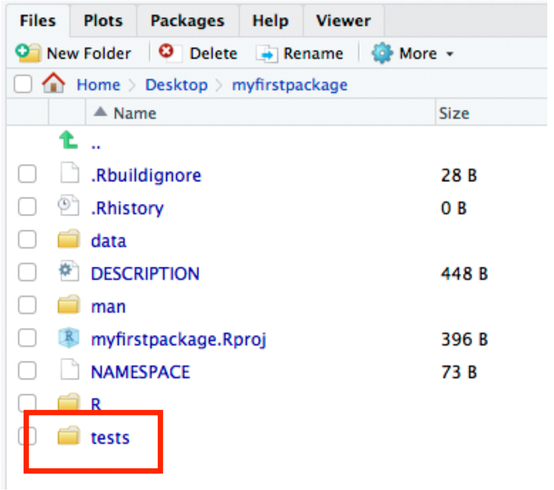
DESCRIPTION FILE

- Automated testing reduces the time to find and fix bugs
- Formalizes testing to ensure consistency
- To get started, run `devtools::use_testthat()`

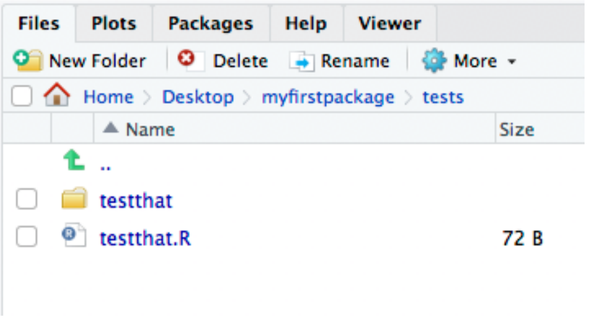


DESCRIPTION FILE

- Automated testing reduces the time to find and fix bugs
- Formalizes testing to ensure consistency
- To get started, run `devtools::use_testthat()`

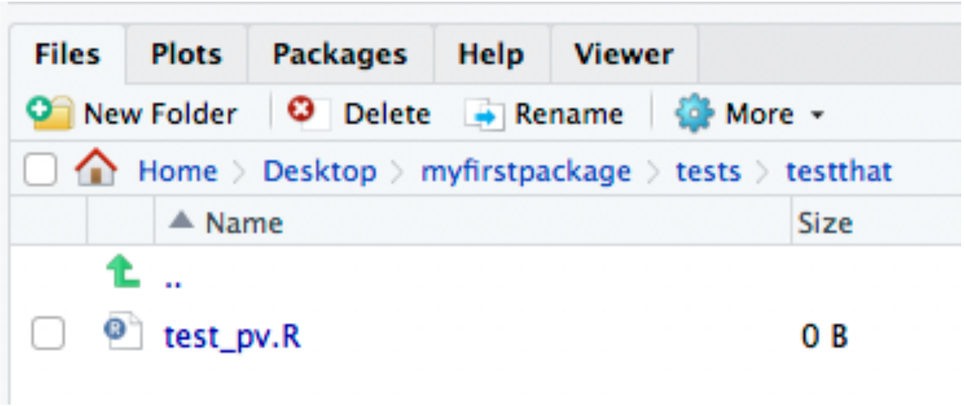


- Inside the tests directory, there are two things:
  1. testthat.R file that runs tests (don't touch!)
  2. testthat folder where you save your tests



CREATING A TEST

1. Create a new .R script
2. Title it `test_pv` (all test files must start with "test" and I typically have one test file per function).
3. Save it in `tests/testthat` directory



CONSTRUCTING TESTS

```
test_that(“text here to describe test”, {
  expect_xxx(function(x), ___)
})
```

CONSTRUCTING TESTS

```
test_that("pv provides proper messages and warnings", {
  expect_error(pv(FV = "1000", .09, n = 5))
  expect_error(pv(FV = list(1, 2, 3), .09, n = 5))
  expect_message(pv(FV = 1000, .36, n = 5))
})
```

The common types of tests include…

- Testing for warnings & messages

CONSTRUCTING TESTS

```
test_that("pv provides proper messages and warnings", {
  expect_error(pv(FV = "1000", .09, n = 5))
  expect_error(pv(FV = list(1, 2, 3), .09, n = 5))
  expect_message(pv(FV = 1000, .36, n = 5))
})

test_that("pv has correct dimensions and output type", {
  expect_is(pv(FV = 1000, .09, n = 5), "numeric")
  expect_true(is.vector(pv(FV = c(1000, 2000), .09, n = 5)))
  expect_length(pv(FV = c(1000, 2000), .09, n = 5), 2)
})
```

The common types of tests include…

- Testing for warnings & messages
- Testing for correct type of outputs

CONSTRUCTING TESTS

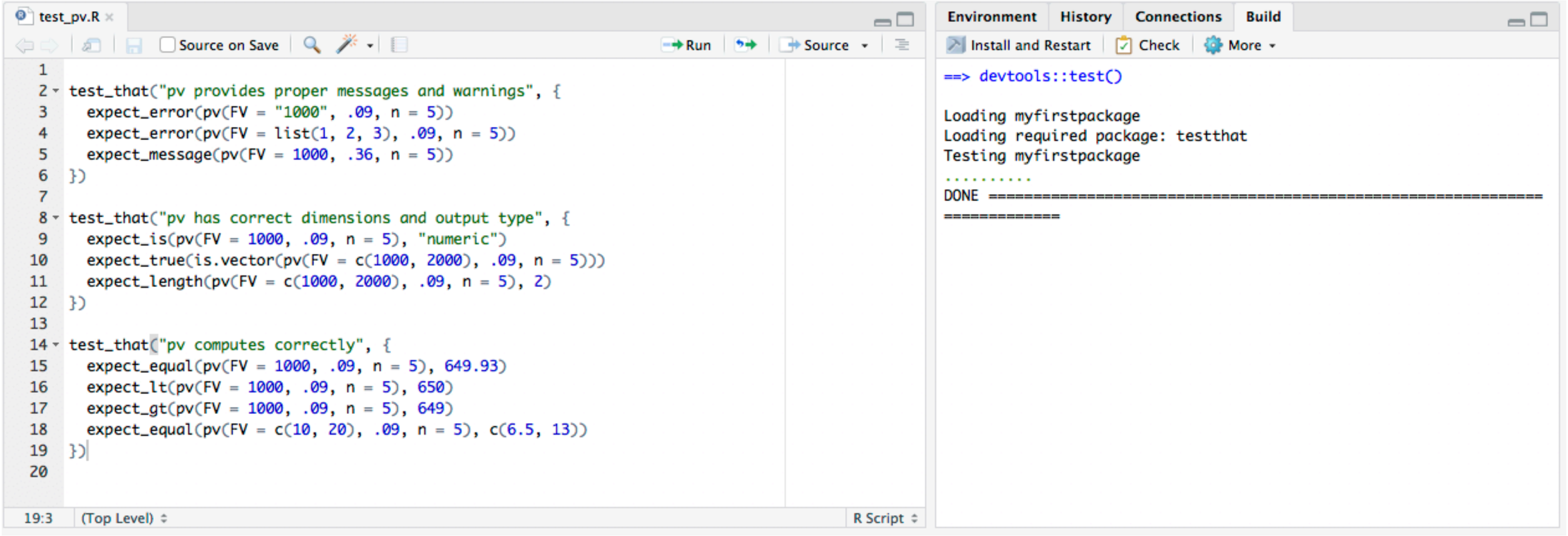
```
test_that("pv provides proper messages and warnings", {
  expect_error(pv(FV = "1000", .09, n = 5))
  expect_error(pv(FV = list(1, 2, 3), .09, n = 5))
  expect_message(pv(FV = 1000, .36, n = 5))
})

test_that("pv has correct dimensions and output type", {
  expect_is(pv(FV = 1000, .09, n = 5), "numeric")
  expect_true(is.vector(pv(FV = c(1000, 2000), .09, n = 5)))
  expect_length(pv(FV = c(1000, 2000), .09, n = 5), 2)
})

test_that("pv computes correctly", {
  expect_equal(pv(FV = 1000, .09, n = 5), 649.93)
  expect_lt(pv(FV = 1000, .09, n = 5), 650)
  expect_gt(pv(FV = 1000, .09, n = 5), 649)
  expect_equal(pv(FV = c(10, 20), .09, n = 5), c(6.5, 13))
})
```

The common types of tests include…

- Testing for warnings & messages
- Testing for correct type of outputs
- Testing for correct computation

RUNNING TESTS

- Once you've incorporated your tests, you can run them anytime with **`devtools::test()`** (Ctrl/Cmd + Shift + T)
- Now, every time you update your functions or make changes to your package you can test quickly and consistently



YOUR TURN!

Integrate testing for our rescale function:

1. Create test_rescale.R script and save in testthat folder
2. Create a test to check for:
   (I) Creating proper warnings or messages
   (II) Check for proper dimensions or output type
   (III) Computes correctly
3. Save and run tests with `devtools::test()` (Ctrl/Cmd + Shift + T)

SOLUTION

test_pv.R | test_rescale.R

Source on Save | Run | Source

```
1  set.seed(123)
2  x <- runif(25, min = 5, max = 20)
3  y <- c("12", x)
4
5  test_that("rescale provides proper messages and warnings", {
6    expect_error(rescale(y))
7  })
8
9  test_that("rescale has correct dimensions and output type", {
10   expect_is(rescale(x), "numeric")
11   expect_length(rescale(x), 25)
12 })
13
14 test_that("rescale computes correctly", {
15   expect_equal(rescale(x)[1:2], c(0.26, 0.78))
16 })
17
```

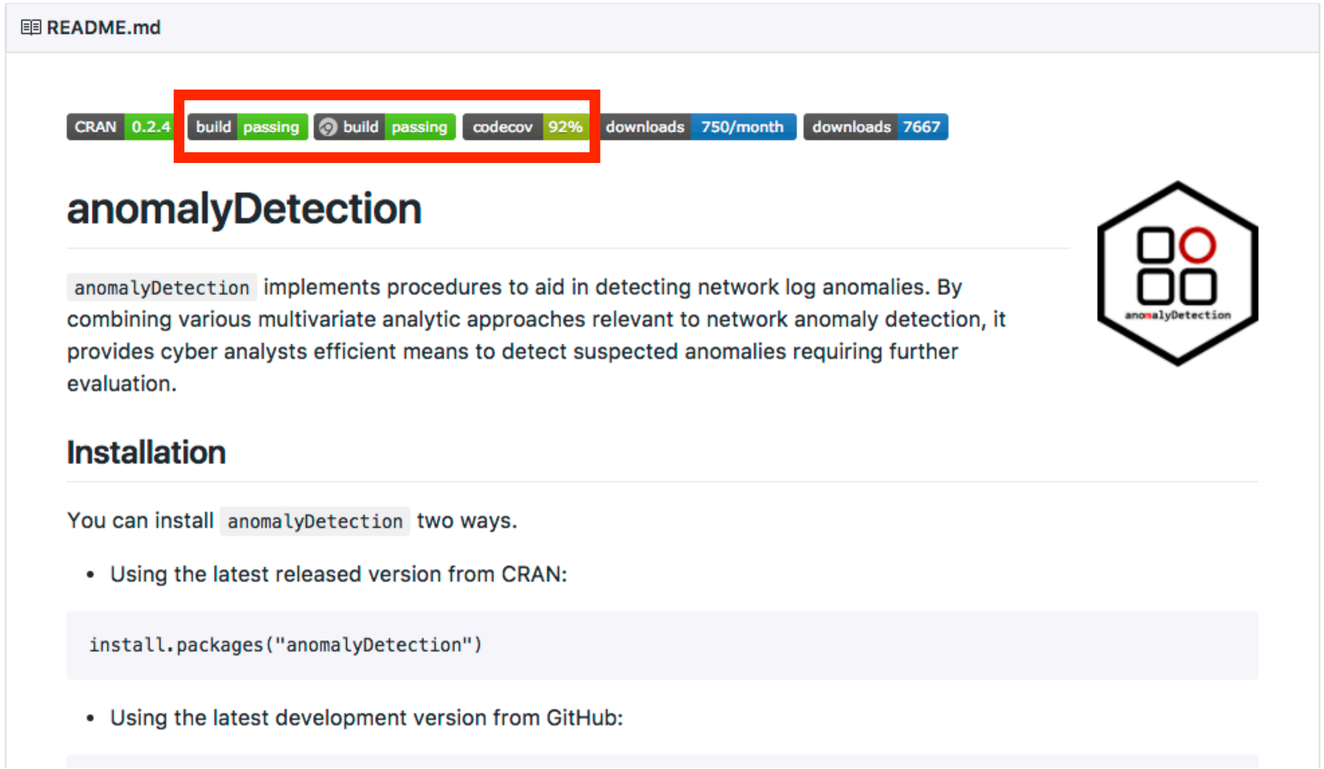
Environment | History | Connections | Build

Install and Restart | Check | More

```
==> devtools::test()

Loading myfirstpackage
Loading required package: testthat
Testing myfirstpackage
.............
DONE ================================================================
================
```

TAKE-AWAY

- Once you get the general pattern down, testing is not too complicated
- You can create very elaborate tests
- You can automate testing online with:
  - travis.ci
  - appveyor
  - codecov


README.md

CRAN 0.2.4 | build passing | build passing | codecov 92% | downloads 750/month | downloads 7667

anomalyDetection

`anomalyDetection` implements procedures to aid in detecting network log anomalies. By combining various multivariate analytic approaches relevant to network anomaly detection, it provides cyber analysts efficient means to detect suspected anomalies requiring further evaluation.

Installation

You can install `anomalyDetection` two ways.

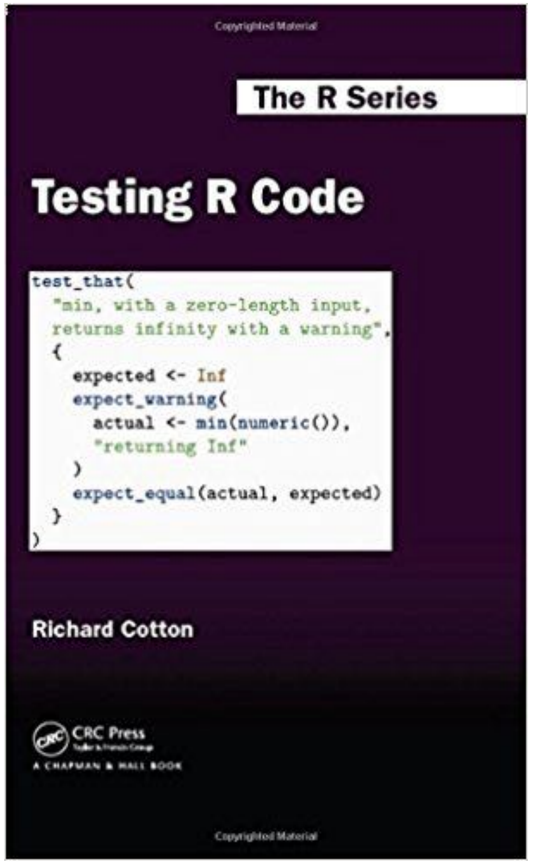
- Using the latest released version from CRAN:

```
install.packages("anomalyDetection")
```

- Using the latest development version from GitHub:

TAKE-AWAY

- Once you get the general pattern down, testing is not too complicated
- You can create very elaborate tests
- You can automate testing online with:
  - travis.ci
  - appveyor
  - codecov
- Great book for digging into testing



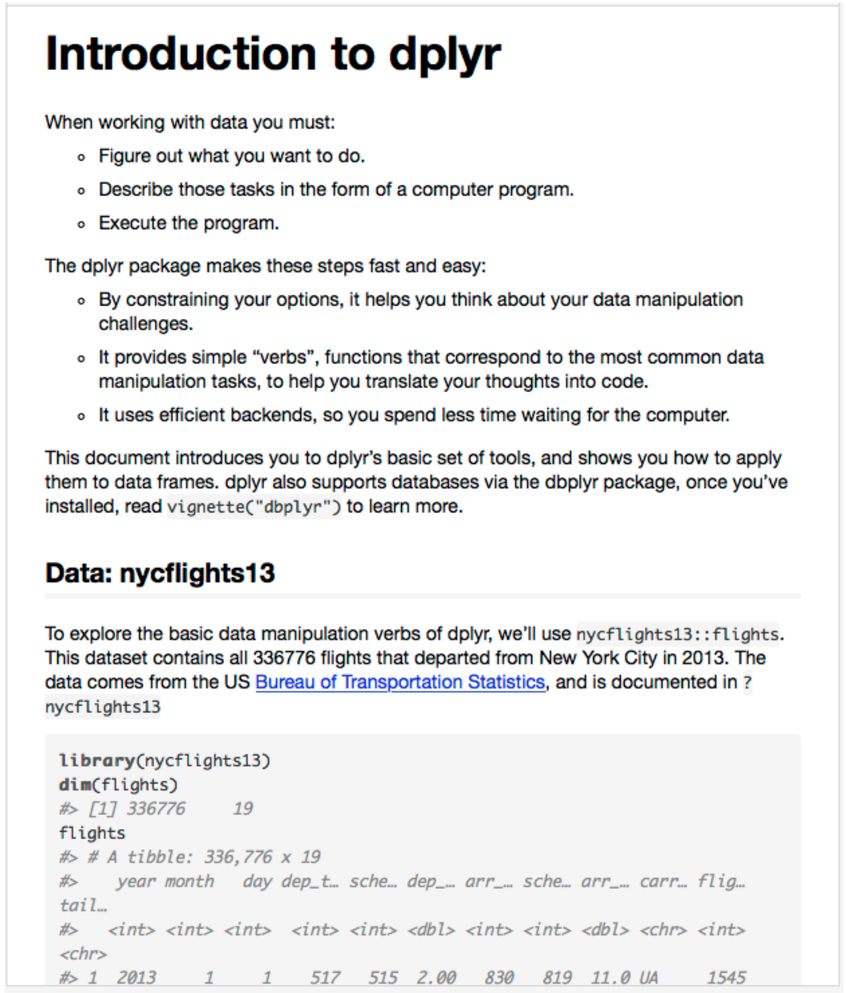
Vignette

Make it easy to learn about your packages!

LEARNING ABOUT YOUR PACKAGE

- Most modern packages include a vignette which:
  - Introduce the purpose for your package
  - Discuss the functions supplied by your package
  - Illustrates how someone can use your package

```
vignette("dplyr")
```



Introduction to dplyr

When working with data you must:

- Figure out what you want to do.
- Describe those tasks in the form of a computer program.
- Execute the program.

The dplyr package makes these steps fast and easy:

- By constraining your options, it helps you think about your data manipulation challenges.
- It provides simple "verbs", functions that correspond to the most common data manipulation tasks, to help you translate your thoughts into code.
- It uses efficient backends, so you spend less time waiting for the computer.

This document introduces you to dplyr's basic set of tools, and shows you how to apply them to data frames. dplyr also supports databases via the dbplyr package, once you've installed, read `vignette("dbplyr")` to learn more.

Data: nycflights13

To explore the basic data manipulation verbs of dplyr, we'll use `nycflights13::flights`. This dataset contains all 336776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics, and is documented in `?nycflights13`

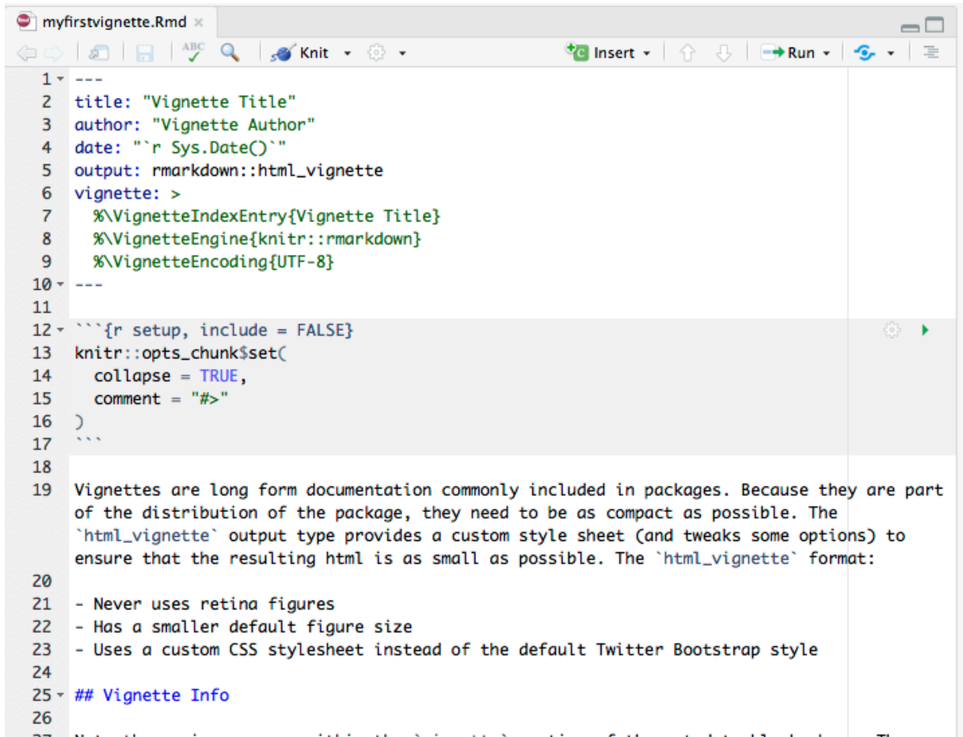
```
library(nycflights13)
dim(flights)
#> [1] 336776     19
flights
#> # A tibble: 336,776 x 19
#>    year month   day dep_t… sche… dep_… arr_… sche… arr_… carr… flig…
tail…
#>   <int> <int> <int>  <int> <int> <dbl> <int> <int> <dbl> <chr> <int>
<chr>
#> 1  2013     1     1    517   515  2.00   830   819  11.0 UA     1545
```

CREATING A VIGNETTE

Initialize a vignette with `devtools::use_vignette()`:

1. Creates a vignettes/ directory
2. Adds necessary dependencies to DESCRIPTION file
3. Creates a draft vignette

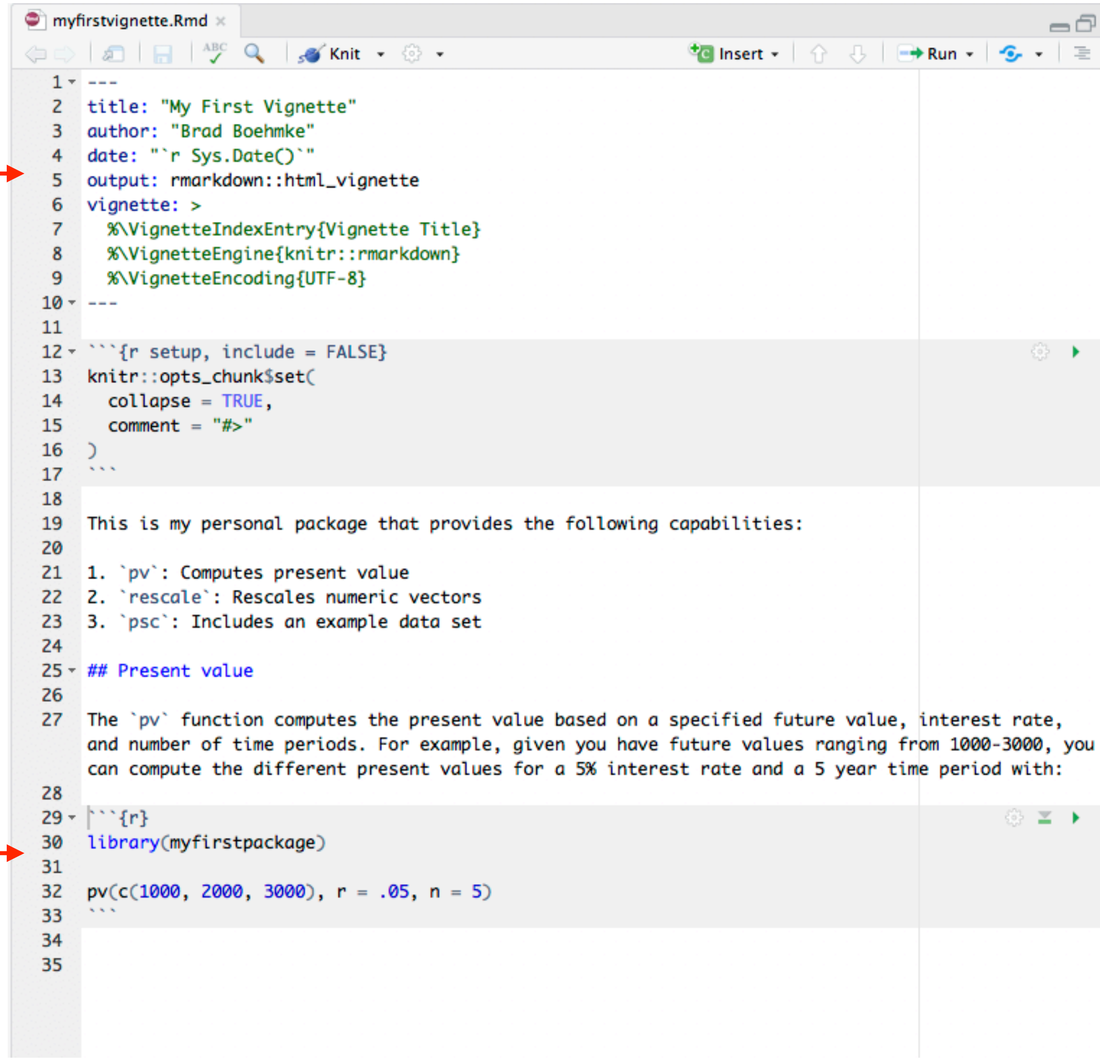
```
devtools::use_vignette("myfirstvignette")
```



CREATING A VIGNETTE

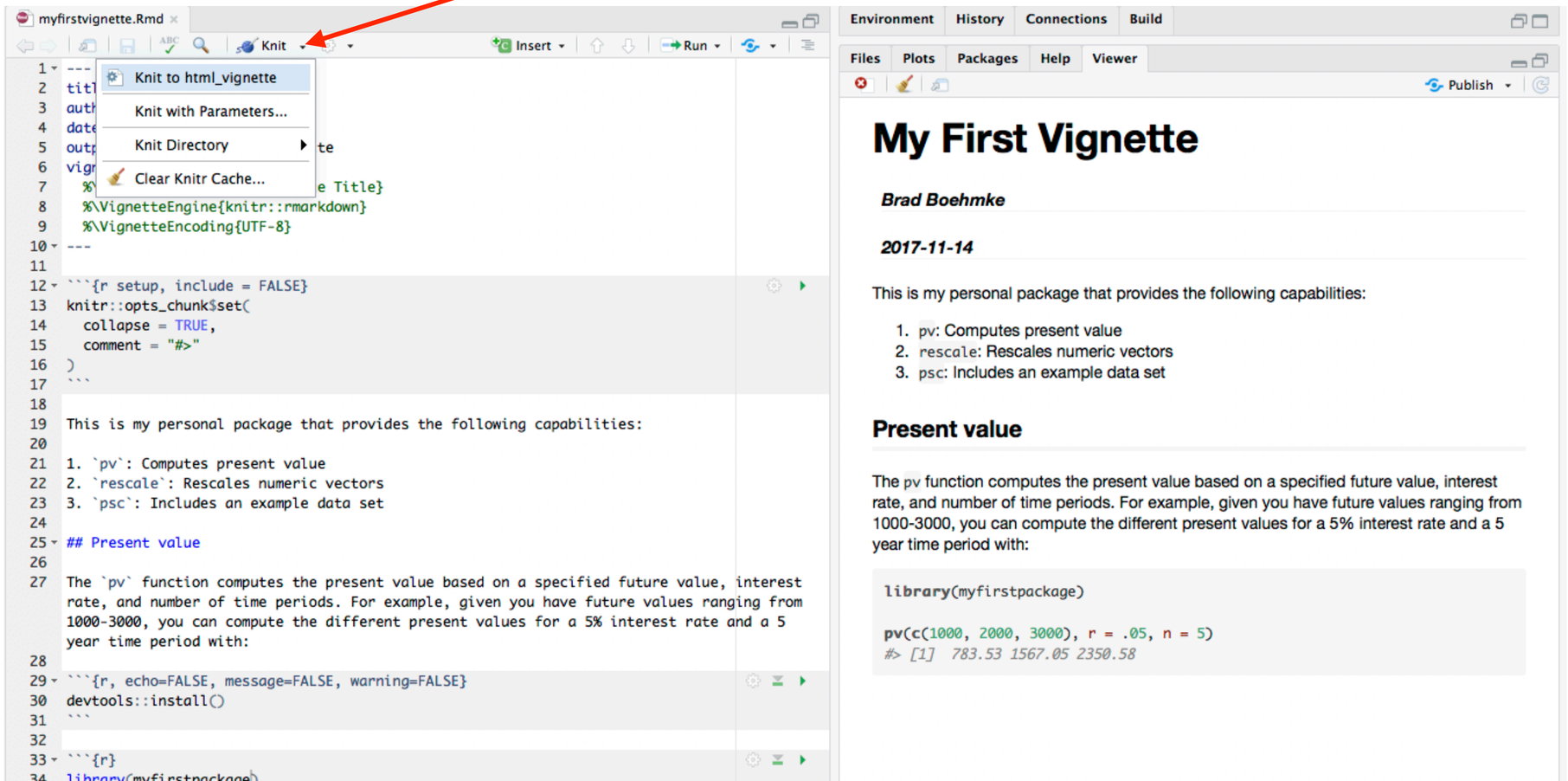
If you are familiar with R Markdown then vignettes are a piece of cake:

1. Run **`devtools::install()`** in your console to install development version of package on your computer.
2. Update YAML
3. Add your own text and headers
4. Include code chunks (load your package like any other package - **`library(myfirstpackage)`**)
5. Anything else you'd want to include to explain this package



CREATING A VIGNETTE

When you are ready just knit to html_vignette (Ctrl/Cmd + Shift + K)



YOUR TURN!

Take 5 minutes to edit your vignette

Local Deployment

Create a local source package to share

BEFORE SHARING!

Be sure to run your local tests and checks prior to sharing!

```
devtools::test()
devtools::check()
```

Assuming everything checks out fine, let's proceed with sharing our package

CREATING THE SOURCE FILE

To share locally, we need to save our packages as a .tar.gz files

```
devtools::build()
```

Console | Terminal | R Markdown

~/Desktop/myfirstpackage/

```
> devtools::build()
'/Library/Frameworks/R.framework/Resources/bin/R' --no-site-file --no-environ --no-save  \
  --no-restore --quiet CMD build '/Users/bradleyboehmke/Desktop/myfirstpackage' --no-resave-data  \
  --no-manual

* checking for file ‘/Users/bradleyboehmke/Desktop/myfirstpackage/DESCRIPTION’ ... OK
* preparing ‘myfirstpackage’:
* checking DESCRIPTION meta-information ... OK
* installing the package to build vignettes
* creating vignettes ... OK
* checking for LF line-endings in source and make files and shell scripts
* checking for empty or unneeded directories
* looking to see if a ‘data/datalist’ file should be added
* building ‘myfirstpackage_0.1.0.tar.gz’

[1] "/Users/bradleyboehmke/Desktop/myfirstpackage_0.1.0.tar.gz"
```

You can now pass along this .tar.gz file

LOADING A SOURCE FILE

Load this package with `install.packages("path", repos = NULL, type = "source")`

```
# install package from a raw source file
install.packages("/Users/bradleyboehmke/Desktop/myfirstpackage_0.1.0.tar.gz", repos = NULL, type = "source")
```

YOUR TURN!

1. *Create your source file.*
2. *Email to your neighbor.*
3. *Load your neighbor's package.*

Public Deployment

Host your package on GitHub or CRAN

HOSTING ON GITHUB

- Git & GitHub provide a free version control capability
- Allows for easy collaboration and beta development of packages
- Allows you to tie into travis.ci, appveyor, codecov and other integrated testing platforms
- RStudio allows you to easily integrate GitHub interactions within the IDE

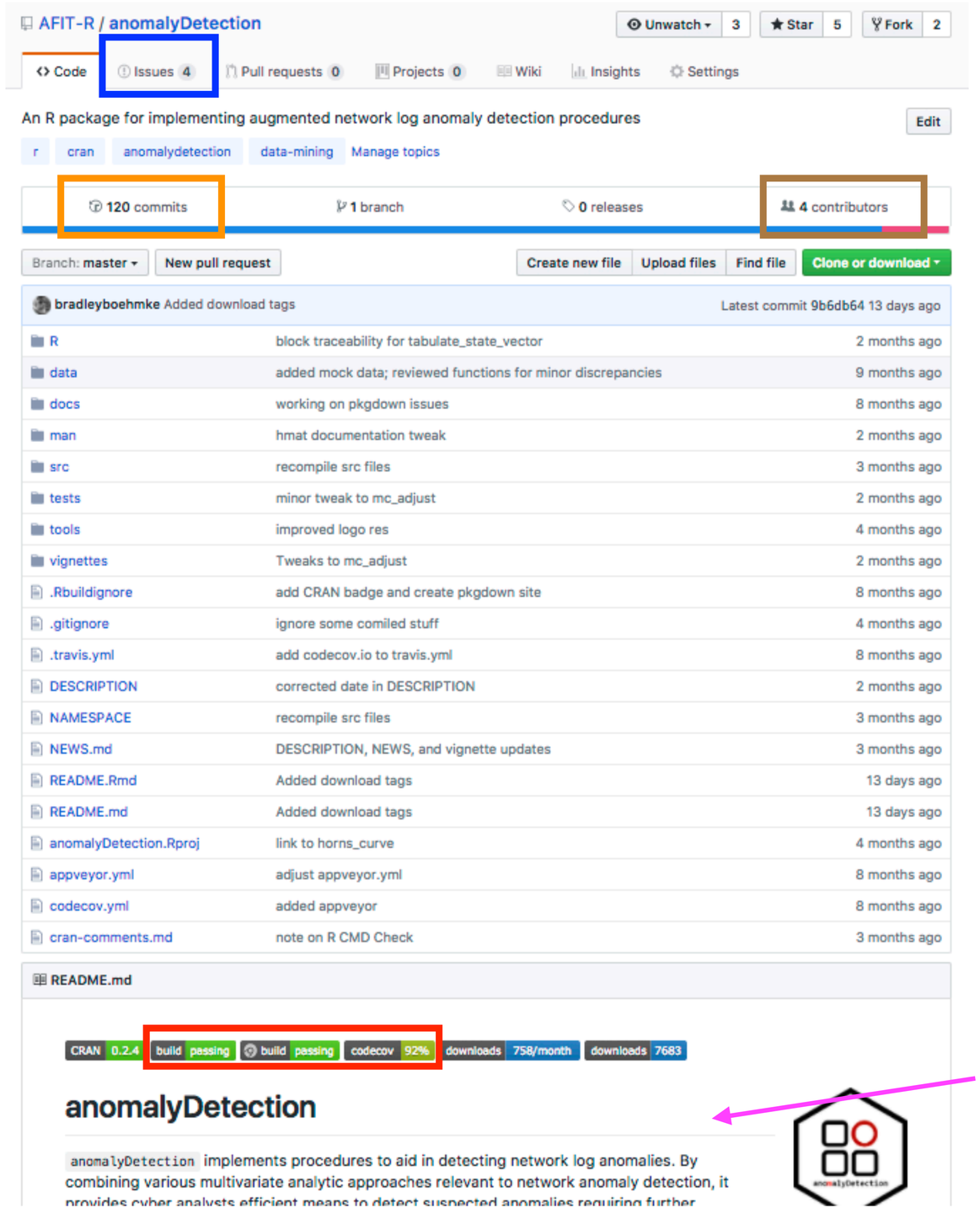
Bugs, suggested improvements, etc.

History

Integrated testing

Collaborators

Package info for public



AFIT-R / anomalyDetection

Unwatch 3 | Star 5 | Fork 2

Code | Issues 4 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

An R package for implementing augmented network log anomaly detection procedures

Edit

r | cran | anomalydetection | data-mining | Manage topics

120 commits | 1 branch | 0 releases | 4 contributors

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

bradleyboehmke Added download tags — Latest commit 9b6db64 13 days ago

R	block traceability for tabulate_state_vector	2 months ago
data	added mock data; reviewed functions for minor discrepancies	9 months ago
docs	working on pkgdown issues	8 months ago
man	hmat documentation tweak	2 months ago
src	recompile src files	3 months ago
tests	minor tweak to mc_adjust	2 months ago
tools	improved logo res	4 months ago
vignettes	Tweaks to mc_adjust	2 months ago
.Rbuildignore	add CRAN badge and create pkgdown site	8 months ago
.gitignore	ignore some comiled stuff	4 months ago
.travis.yml	add codecov.io to travis.yml	8 months ago
DESCRIPTION	corrected date in DESCRIPTION	2 months ago
NAMESPACE	recompile src files	3 months ago
NEWS.md	DESCRIPTION, NEWS, and vignette updates	3 months ago
README.Rmd	Added download tags	13 days ago
README.md	Added download tags	13 days ago
anomalyDetection.Rproj	link to horns_curve	4 months ago
appveyor.yml	adjust appveyor.yml	8 months ago
codecov.yml	added appveyor	8 months ago
cran-comments.md	note on R CMD Check	3 months ago

README.md

CRAN 0.2.4 | build passing | build passing | codecov 92% | downloads 758/month | downloads 7683

anomalyDetection

anomalyDetection implements procedures to aid in detecting network log anomalies. By combining various multivariate analytic approaches relevant to network anomaly detection, it provides cyber analysts efficient means to detect suspected anomalies requiring further

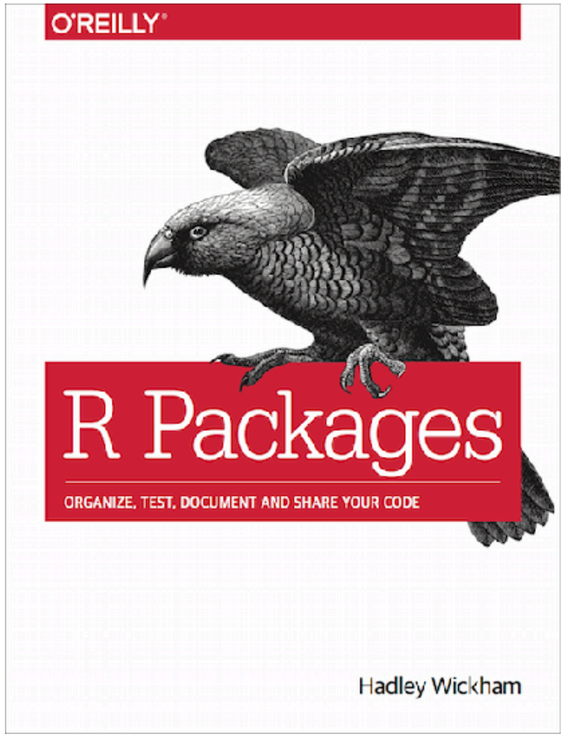
HOSTING ON GITHUB

- To install a package already on GitHub, you must know the **Username** and **Repository** that the package resides in
- Once you have this information, the `install_github` command will install the package on your computer.
- Because of the size of most vignettes, `devtools` does not automatically download them from GitHub. If you want to include the vignette, you must add the argument `build_vignettes = TRUE`
- To start posting to GitHub, you will first need to install and set up Git, which is what you'll use to communicate with GitHub. Start by visiting http://git-scm.com/download and installing Git for your system.

```
# download package from GitHub
devtools::install_github("Username/Repository", build_vignettes = TRUE)
```

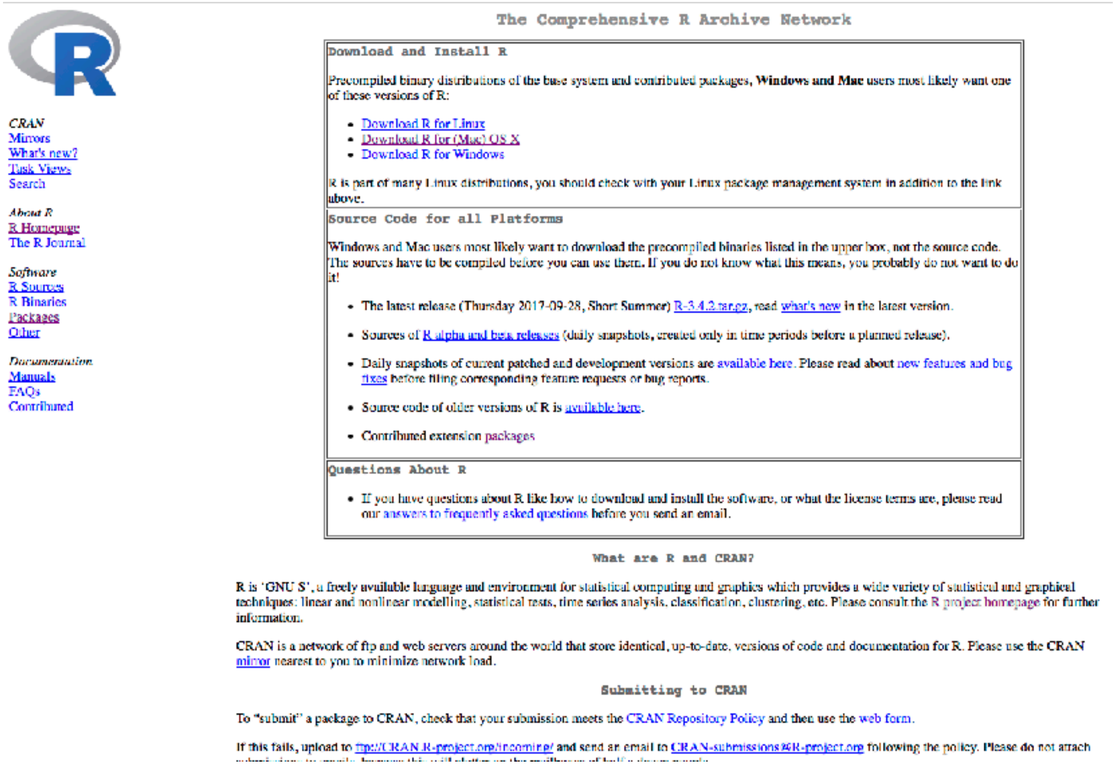
CONNECTING TO GIT & GITHUB

Not enough time!



HOSTING ON CRAN

- The centralized R package-hosting website
- The `install.packages` command automatically downloads packages from CRAN
- Designed to ensure high package quality and functionality on all systems and versions of R, including legacy, current, and devel



CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (Thursday 2017-09-28, Short Summer) R-3.4.2.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to minimize network load.

Submitting to CRAN

To "submit" a package to CRAN, check that your submission meets the CRAN Repository Policy and then use the web form.

If this fails, upload to ftp://CRAN.R-project.org/incoming/ and send an email to CRAN-submissions@R-project.org following the policy. Please do not attach

HOSTING ON CRAN

- Before uploading to CRAN, your package must be checked for bugs
- The check function from devtools will simulate the CRAN check for your system
- Goal is to have: 0 errors | 0 warnings | 0 notes

Often, notes can be justifiable (i.e. *"R under development (unstable)"*)

```
devtools::test()
devtools::check()
devtools::build_win()
```

HOSTING ON CRAN

- If your checks pass, create a cran-comments.md file

```
devtools::use_cran_comments()
```

Initial Comments

cran-comments.md

Preview | Insert | Run

```
## Test environments
* local OS X install, R 3.4.2
* ubuntu 12.04 (on travis-ci), R 3.4.2
* win-builder (devel and release)
* Windows (on apveyor)

## R CMD check results

0 errors | 0 warnings | 0 note


## Reverse dependencies

There are currently no downstream dependencies for this package
```

Version Update Comments

cran-comments.md

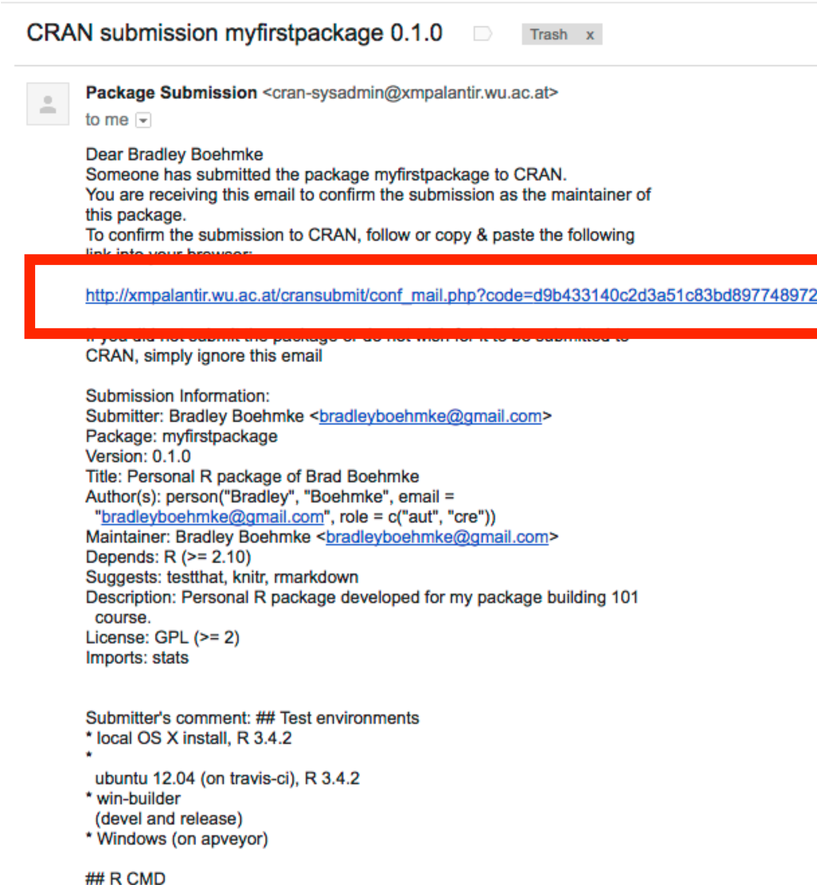
Preview

```
## Changes

This is a resubmission. In this version I have:

- Allowed for better tolerance when inverting covariance matrices
- Rewritten code to improve speed (and accuracy), both in R and C++
- Added new function to display histogram matrices

## Test environments

* local Windows install, R 3.4.1
* ubuntu 12.04 (on travis-ci), R 3.4.1
* Windows (on apveyor)
* win-builder (devel and release)

## R CMD check results

0 errors | 0 warnings | 1 note

We believe that the note is a false positive.


## Reverse dependencies

There are currently no downstream dependencies for this package.
```

HOSTING ON CRAN

- When you are ready to submit, run one of the following:

```
devtools::release()
devtools::submit()
```



CRAN submission myfirstpackage 0.1.0 Trash x

Package Submission <cran-sysadmin@xmpalantir.wu.ac.at>
to me

Dear Bradley Boehmke
Someone has submitted the package myfirstpackage to CRAN.
You are receiving this email to confirm the submission as the maintainer of this package.
To confirm the submission to CRAN, follow or copy & paste the following

http://xmpalantir.wu.ac.at/cransubmit/conf_mail.php?code=d9b433140c2d3a51c83bd897748972

CRAN, simply ignore this email

Submission Information:
Submitter: Bradley Boehmke <bradleyboehmke@gmail.com>
Package: myfirstpackage
Version: 0.1.0
Title: Personal R package of Brad Boehmke
Author(s): person("Bradley", "Boehmke", email = "bradleyboehmke@gmail.com", role = c("aut", "cre"))
Maintainer: Bradley Boehmke <bradleyboehmke@gmail.com>
Depends: R (>= 2.10)
Suggests: testthat, knitr, rmarkdown
Description: Personal R package developed for my package building 101 course.
License: GPL (>= 2)
Imports: stats

Submitter's comment: ## Test environments
* local OS X install, R 3.4.2
*
ubuntu 12.04 (on travis-ci), R 3.4.2
* win-builder
(devel and release)
* Windows (on apveyor)

R CMD

- You will receive an email asking to confirm submission
- Click link and wait for feedback!

Programming Exercise

Expand the capability of your package

YOUR TURN!

1. Add any custom functions you have to your personal package.

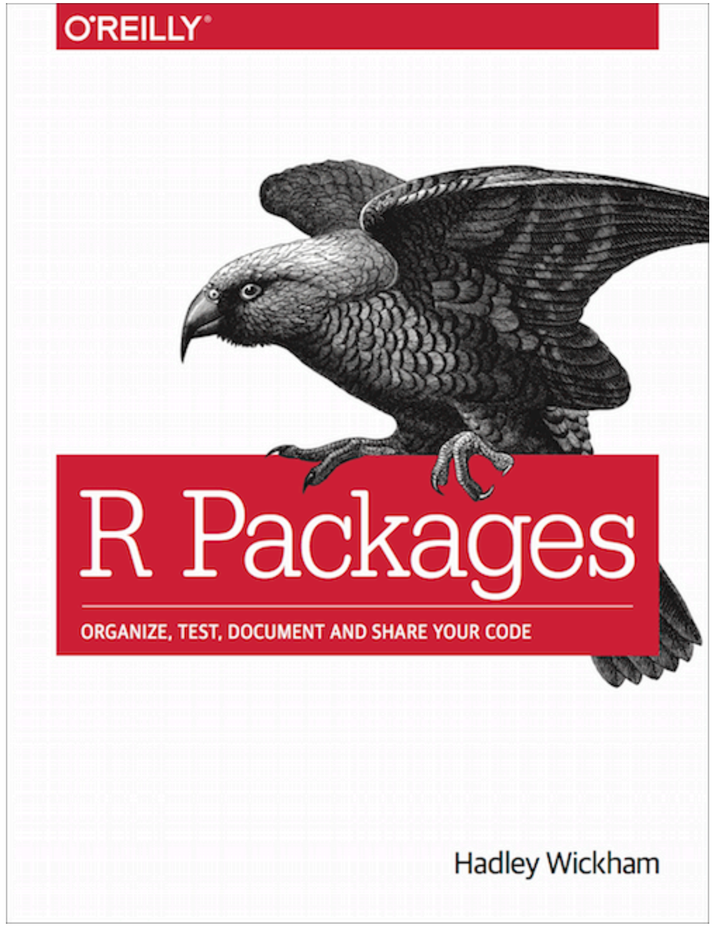
2. If you don't have any, practice by incorporating the stat functions in the package-functions.R file.

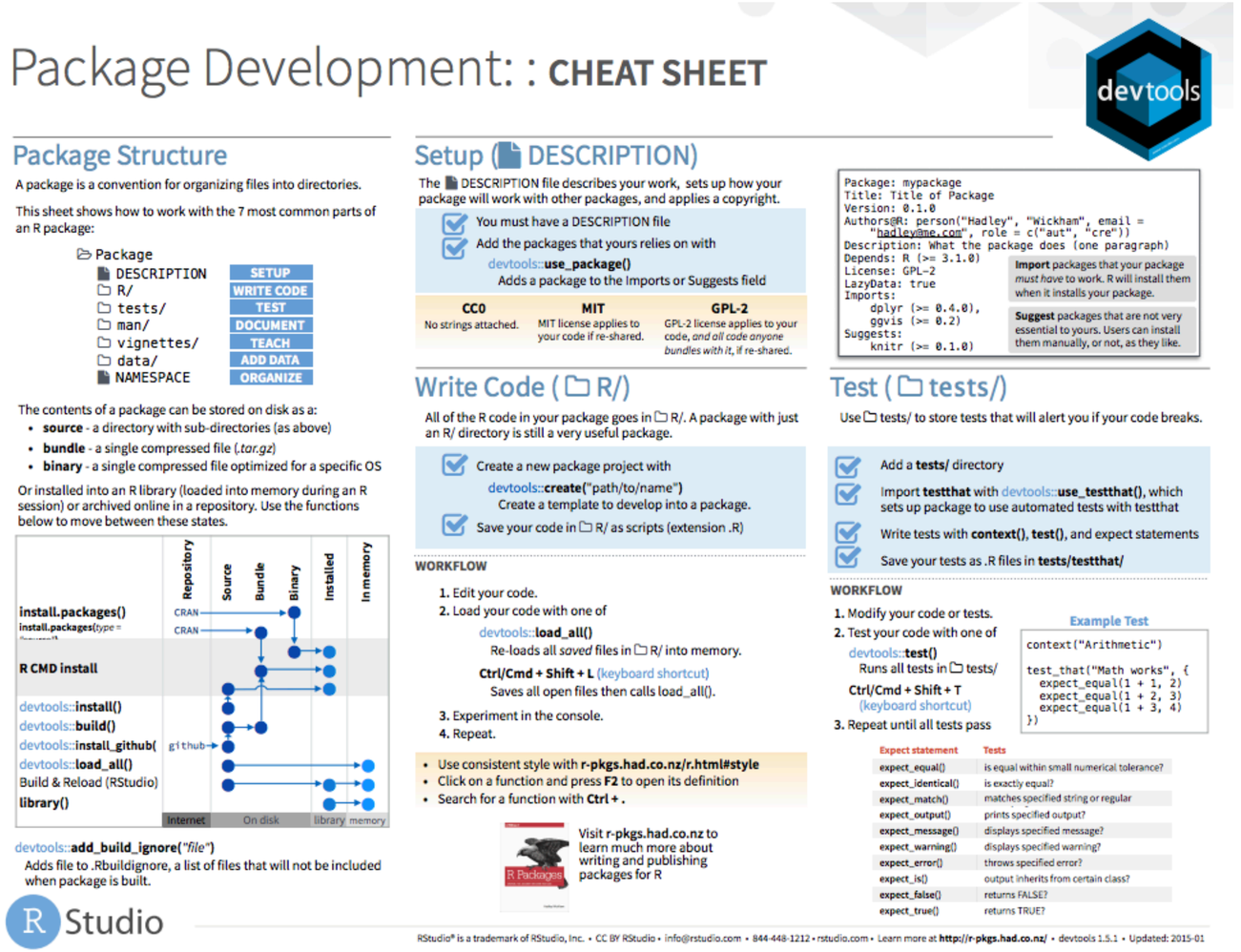
WHAT TO REMEMBER

PACKAGE CREATION PROCESS

Phase	Description
`Start New Project`	Initialize container for holding functions and documentation files
`Write Functions`	Write code for what the package will do
`Add Documentation`	Describe how to use the functions
`Add Testing`	Check that changes and updates don't affect other functions
`Write a Vignette`	Describe in more detail the vision for the package and how it could be utilized
`Deploy`	Host your package on your machine, GitHub, or CRAN

LEARN MORE


O'REILLY
R Packages
ORGANIZE, TEST, DOCUMENT AND SHARE YOUR CODE
Hadley Wickham


Package Development: : CHEAT SHEET
devtools
Package Structure
A package is a convention for organizing files into directories.
This sheet shows how to work with the 7 most common parts of an R package:
Package
DESCRIPTION
R/
tests/
man/
vignettes/
data/
NAMESPACE
SETUP
WRITE CODE
TEST
DOCUMENT
TEACH
ADD DATA
ORGANIZE
The contents of a package can be stored on disk as a:
source - a directory with sub-directories (as above)
bundle - a single compressed file (.tar.gz)
binary - a single compressed file optimized for a specific OS
Or installed into an R library (loaded into memory during an R session) or archived online in a repository. Use the functions below to move between these states.
Repository
Source
Bundle
Binary
Installed
In memory
install.packages()
R CMD install
devtools::install()
devtools::build()
devtools::install_github(
devtools::load_all()
Build & Reload (RStudio)
library()
CRAN
github
Internet
On disk
library
memory
devtools::add_build_ignore("file")
Adds file to .Rbuildignore, a list of files that will not be included when package is built.
Setup (DESCRIPTION)
The DESCRIPTION file describes your work, sets up how your package will work with other packages, and applies a copyright.
You must have a DESCRIPTION file
Add the packages that yours relies on with
devtools::use_package()
Adds a package to the Imports or Suggests field
CC0
No strings attached.
MIT
MIT license applies to your code if re-shared.
GPL-2
GPL-2 license applies to your code, and all code anyone bundles with it, if re-shared.
Package: mypackage
Title: Title of Package
Version: 0.1.0
Authors@R: person("Hadley", "Wickham", email = "hadley@me.com", role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: GPL-2
LazyData: true
Imports:
dplyr (>= 0.4.0),
ggvis (>= 0.2)
Suggests:
knitr (>= 0.1.0)
Import packages that your package must have to work. R will install them when it installs your package.
Suggest packages that are not very essential to yours. Users can install them manually, or not, as they like.
Write Code (R/)
All of the R code in your package goes in R/. A package with just an R/ directory is still a very useful package.
Create a new package project with
devtools::create("path/to/name")
Create a template to develop into a package.
Save your code in R/ as scripts (extension .R)
WORKFLOW
1. Edit your code.
2. Load your code with one of
devtools::load_all()
Re-loads all saved files in R/ into memory.
Ctrl/Cmd + Shift + L (keyboard shortcut)
Saves all open files then calls load_all().
3. Experiment in the console.
4. Repeat.
Use consistent style with r-pkgs.had.co.nz/r.html#style
Click on a function and press F2 to open its definition
Search for a function with Ctrl + .
Visit r-pkgs.had.co.nz to learn much more about writing and publishing packages for R
Test (tests/)
Use tests/ to store tests that will alert you if your code breaks.
Add a tests/ directory
Import testthat with devtools::use_testthat(), which sets up package to use automated tests with testthat
Write tests with context(), test(), and expect statements
Save your tests as .R files in tests/testthat/
WORKFLOW
1. Modify your code or tests.
2. Test your code with one of
devtools::test()
Runs all tests in tests/
Ctrl/Cmd + Shift + T
(keyboard shortcut)
3. Repeat until all tests pass
Example Test
context("Arithmetic")
test_that("Math works", {
expect_equal(1 + 1, 2)
expect_equal(1 + 2, 3)
expect_equal(1 + 3, 4)
})
Expect statement
Tests
expect_equal()
is equal within small numerical tolerance?
expect_identical()
is exactly equal?
expect_match()
matches specified string or regular
expect_output()
prints specified output?
expect_message()
displays specified message?
expect_warning()
displays specified warning?
expect_error()
throws specified error?
expect_is()
output inherits from certain class?
expect_false()
returns FALSE?
expect_true()
returns TRUE?
RStudio
RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at http://r-pkgs.had.co.nz/ • devtools 1.5.1 • Updated: 2015-01

QUESTIONS

Writing Functions

Before we can develop packages we need to understand how to write functions

FUNCTION FUNDAMENTALS

WHEN TO WRITE FUNCTIONS

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$b, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

You should consider writing a function whenever you've copied and pasted a block of code more than twice.

Can you spot the error?

WHEN TO WRITE FUNCTIONS

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$b, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

You should consider writing a function whenever you've copied and pasted a block of code more than twice.

Can you spot the error?

DEFINING YOUR OWN FUNCTION

```
my_fun <- function(arg1, arg2) {
  body
}
```

Functions have 3 parts:

1. formals (aka arguments)

2. body (code inside the function)

3. environment

DEFINING YOUR OWN FUNCTION

```
pv <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}
```

Functions have 3 parts:

1. formals (aka arguments)
2. body (code inside the function)
3. environment

ANATOMY OF A FUNCTION

```
pv <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}

formals(pv)
$FV
$r
$n

body(pv)
{
    present_value <- FV/(1 + r)^n
    round(present_value, 2)
}

environment(pv)
<environment: R_GlobalEnv>
```

Functions have 3 parts:

1. formals (aka arguments)
2. body (code inside the function)
3. environment

FUNCTION OUTPUT

```
pv <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}

pv(FV = 1000, r = .08, n = 5)
[1] 680.58

pv2 <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  return(present_value)
  round(present_value, 2)
}

pv2(1000, .08, 5)
[1] 680.5832
```

What gets returned from a function is either:

1. The last expression evaluated

2. return(value), which forces the function to stop execution and return value

FUNCTION OUTPUT

```
pv <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}

pv(FV = 1000, r = .08, n = 5)
[1] 680.58

pv2 <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  return(present_value)
  round(present_value, 2)
}

pv2(1000, .08, 5)
[1] 680.5832
```

What gets returned from a function is either:

1. The last expression evaluated

2. return(value), which forces the function to stop execution and return value

Note the differences in how we call these functions. Why do both cases work?

YOUR TURN!

- *Define a function titled* **`ratio`** *that takes arguments* **`x`** *and* **`y`** *and returns their ratio,* **`x / y`**

- *Call* **`ratio()`** *with arguments 3 and 4*

SOLUTION

```
ratio <- function(x, y) {
  x / y
}

ratio(3, 4)
[1] 0.75
```

HANDLING ARGUMENTS

CALLING ARGUMENTS IN DIFFERENT WAYS

```
pv(FV = 1000, r = .08, n = 5)
[1] 680.58
```

Using argument names

```
pv(1000, .08, 5)
[1] 680.58
```

positional matching

```
pv(r = .08, FV = 1000, n = 5)
[1] 680.58
```

must use names if you change order otherwise…

```
pv(.08, 1000, 5)
[1] 0
```

error or incorrect computation will occur

```
pv(1000, .08)
Error in pv(1000, 0.08) : argument "n" is missing,
with no default
```

missing arguments results in error

SETTING DEFAULT ARGUMENTS

```
pv <- function(FV, r, n = 5) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}

pv(1000, .08)
[1] 680.58

PV(1000, .08, n = 3)
[1] 793.83
```

We can set default argument values

now if we do not call the argument the default is used

and we can change the default simply by specifying an n value

ORDERING ARGUMENTS

Ordering arguments in your functions is important:

- positional matching
- pipe (%>%) operator

```
my_fun <- function(data, arg2, arg3 = 5) {
  body
}
```

General rules:

- Data argument first
- First couple arguments require specifying
- Later arguments have defaults

YOUR TURN!

Earlier in these slides you saw the following code duplicated:

```
(df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

Can you write a function called **`rescale`** *that takes argument* **`x`** *and executes this code?*

Test it on the vector provided in your .R script

SOLUTION

```
rescale <- function(x){
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

rescale(vec1)
[1] 0.2704415 0.8299695 0.4060968 0.9358038 1.0000000 0.0000000 0.5392146
[8] 0.9463095 0.5652837 0.4593287
```

YOUR TURN!

Now add an argument to **`rescale`** *that allows you to round the output to a specified decimal. Set the default to 2.*

SOLUTION

```
rescale <- function(x, digits = 2){
  rng <- range(x, na.rm = TRUE)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}

rescale(vec1)
[1] 0.27 0.83 0.41 0.94 1.00 0.00 0.54 0.95 0.57 0.46

rescale(vec1, 3)
[1] 0.270 0.830 0.406 0.936 1.000 0.000 0.539 0.946 0.565 0.459
```

YOUR TURN!

Now let's move the **`na.rm = TRUE`** *argument into the functions formals so that the user can specify whether or not they want to remove NAs. Set the default to* **TRUE**.

SOLUTION

Showing how many missing values were removed

```
rescale <- function(x, digits = 2, na.rm = TRUE){
  if(isTRUE(na.rm)) x <- na.omit(x)
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}

vec1 <- c(NA, vec1)
rescale(vec1)
[1] 0.27 0.83 0.41 0.94 1.00 0.00 0.54 0.95 0.57 0.46
attr(,"na.action")
[1] 1
attr(,"class")
[1] "omit"
```

SOLUTION

Hiding how many missing values were removed

```
rescale <- function(x, digits = 2, na.rm = TRUE){
  if(isTRUE(na.rm)) x <- x[!is.na(x)]
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}

rescale(vec1)
[1] 0.27 0.83 0.41 0.94 1.00 0.00 0.54 0.95 0.57 0.46
```

YOUR TURN!

Now try to apply the **`rescale`** *function across each variable in the* **`mtcars`** *data set.*

Hint: try using one of the **`apply`** *or* **`map`** *functions.*

SOLUTION

You can now apply this function over a data frame, list, matrix with the map function

```
library(purrr)

mtcars %>%
  map_df(rescale)

# A tibble: 32 × 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1   0.45   0.5  0.22  0.20  0.53  0.28  0.23     0     1   0.5  0.43
2   0.45   0.5  0.22  0.20  0.53  0.35  0.30     0     1   0.5  0.43
3   0.53   0.0  0.09  0.14  0.50  0.21  0.49     1     1   0.5  0.00
4   0.47   0.5  0.47  0.20  0.15  0.44  0.59     1     0   0.0  0.00
5   0.35   1.0  0.72  0.43  0.18  0.49  0.30     0     0   0.0  0.14
6   0.33   0.5  0.38  0.19  0.00  0.50  0.68     1     0   0.0  0.00
7   0.17   1.0  0.72  0.68  0.21  0.53  0.16     0     0   0.0  0.43
```

INVALID PARAMETERS

INVALID PARAMETERS

For functions that will be re-used, and especially for those used by someone other than the creator, it is good to check the validity of the arguments.

```
my_fun <- function(data, arg2, arg3 = 5) {

  if(condition) {
    message or warning
  }

  body
}
```

Common issues:

- Making sure data is in the right structure (i.e. df, list, vector)
- Are the argument inputs the right class (i.e. numeric, character)
- Are the argument inputs within the proper boundary limits

INVALID PARAMETERS

Our pv function works on a vector of future values, not data frames, lists, or matrices. Let's add a warning in case a user tries to feed it a non-atomic vector.

INVALID PARAMETERS

Our pv function works on a vector of future values, not data frames, lists, or matrices. Let's add a warning in case a user tries to feed it a non-atomic vector.

```
pv <- function(FV, r, n = 5) {

  if(!is.atomic(FV) {
    stop('FV must be an atomic vector')
  }

  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}
```

- Check if class of FV is something other than a vector (be careful with is.vector - use is.atomic instead)
- If so, stop, return an error, and the specified message

INVALID PARAMETERS

Our pv function works on a vector of future values, not data frames, lists, or matrices. Let's add a warning in case a user tries to feed it a non-vector.

```
fv_l <- list(fv1 = 800,
             fv2 = 900,
             fv3 = 1100)


pv(fv_l, 0.08)
Error in pv(fv_l, 0.08) : FV must be an
atomic vector
```

- Now when we execute pv on a non-atomic vector we get an error output

INVALID PARAMETERS

Now let's add tests for the type of class input.

```
pv <- function(FV, r, n = 5) {

  if(!is.atomic(FV)) {
    stop('FV must be an atomic vector')
  }

  if(!is.numeric(FV) | !is.numeric(r) | !is.numeric(n)){
    stop('This function only works for numeric inputs!\n',
         'You have provided objects of the following classes:\n',
         'FV: ', class(FV), '\n',
         'r: ', class(r), '\n',
         'n: ', class(n))
  }

  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}
```

Now we test for

- data type
- argument class

and both of these will provide warnings if violated

INVALID PARAMETERS

Now let's add tests for the type of class input.

```
pv(FV = "1000", .08, n = 5)
Error in pv(FV = "1000", 0.08, n = 5) :
  This function only works for numeric inputs!
You have provided objects of the following classes:
FV: character
r: numeric
n: numeric
```

Now we test for

- data type
- argument class

and both of these will provide warnings if violated

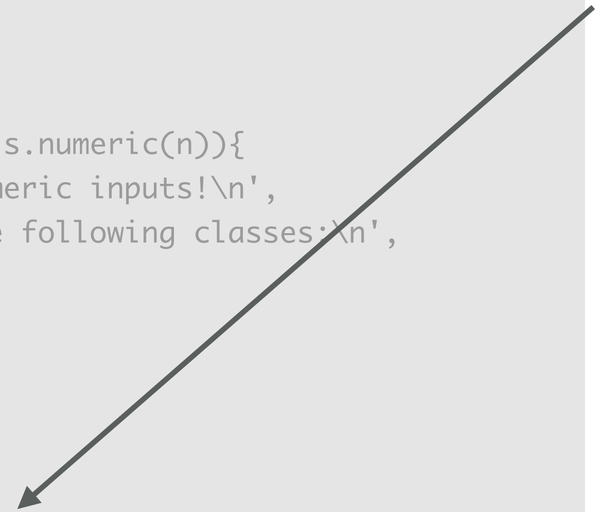
INVALID PARAMETERS

What else can you think of?

INVALID PARAMETERS

What else can you think of? What about abnormal interest rate ranges?

```
pv <- function(FV, r, n = 5) {

  if(!is.atomic(FV)) {
    stop('FV must be an atomic vector')
  }

  if(!is.numeric(FV) | !is.numeric(r) | !is.numeric(n)){
    stop('This function only works for numeric inputs!\n',
         'You have provided objects of the following classes:\n',
         'FV: ', class(FV), '\n',
         'r: ', class(r), '\n',
         'n: ', class(n))
  }

  if(r < 0 | r > .25) {
    message('The input for r exceeds the normal\n',
            'range for interest rates (0-25%)')
  }

  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}
```



If we add a message() this allows us to:

- notify the user of something
- while still executing the code

INVALID PARAMETERS

What else can you think of? What about abnormal interest rate ranges?

```
pv(FV = 1000, r = .28, n = 5)
The input for r exceeds the normal
range for interest rates (0-25%)
[1] 1292.36
```

If we add a message() this allows us to:

- notify the user of something
- while still executing the code

YOUR TURN!

Going back to the rescale function:

```
rescale <- function(x, digits = 2, na.rm = TRUE){
  if(isTRUE(na.rm)) x <- x[!is.na(x)]
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}
```

YOUR TURN!

Going back to the rescale function add conditional statements to check and provide appropriate errors or messages for:

- *making sure* `x` *input is a numeric vector*
- `digits` *input is a numeric vector of one element*
- `na.rm` *input is a single logical input*

SOLUTION

```
rescale <- function(x, digits = 2, na.rm = TRUE){
  # ensure argument inputs are valid
  if(!is.numeric(x)) {
    stop('x must be an atomic numeric vector')
  }
  if(!is.numeric(digits) | length(digits) > 1) {
    stop('digits must be a numeric vector of one element')
  }
  if(!is.logical(na.rm)) {
    stop('na.rm must be logical input (TRUE or FALSE)')
  }

  if(isTRUE(na.rm)) x <- x[!is.na(x)]
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}
```

```
rescale <- function(x, digits = 2, na.rm = TRUE){
  # ensure argument inputs are valid
  if(!is.numeric(x)) {
    stop('x must be an atomic numeric vector')
  }
  if(!is.numeric(digits) | length(digits) > 1) {
    stop('digits must be a numeric vector of one element')
  }
  if(!is.logical(na.rm)) {
    stop('na.rm must be logical input (TRUE or FALSE)')
  }
```

SOLUTION

```
rescale(c(letters))

rescale(vec1, digits = c(1, 2))

rescale(vec1, na.rm = "false")
```