Creating R Packages

James Balamuta

Department of Informatics, Statistics
University of Illinois at Urbana-Champaign

July 20, 2017

On the Agenda

- R Packages
    - Motivation
    - Package Structure
    - Documentation
    - Testing
    - Data

On the Agenda

1. R Packages
   - Background
2. R Package Projects
   - Creation
3. Behind the Scenes
   - Package Structure
   - Licensing
   - R Code
   - Documentation
4. Misc
   - Package Creation CLI

What is an R package?

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.
– Hadley Wickham in R packages

How many R packages are out there?

- At the time of this writing, there are **11098** packages on CRAN!
- This does **not** include packages on GitHub, RForge, or in an additional repository.

```r
# See all R packages on CRAN
available.packages()
```

Why an R Package?

Why not?

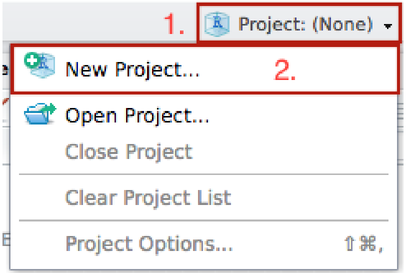
- Code is already organized via functions.
- Nothing already exists (hence why you wrote it!)
- Give back to the community, *pay it forward*
  - Example: All class material is made *freely* available for non-commerical use.
- Portfolio of your coding and development skills.
  - Examples: sitmo, cIRT, balamuta, . . .
- If you track the package via VCS, you can see how much you've grown as a developer and statistician.

On the Agenda

1. R Packages
   - Background
2. R Package Projects
   - Creation
3. Behind the Scenes
   - Package Structure
   - Licensing
   - R Code
   - Documentation
4. Misc
   - Package Creation CLI

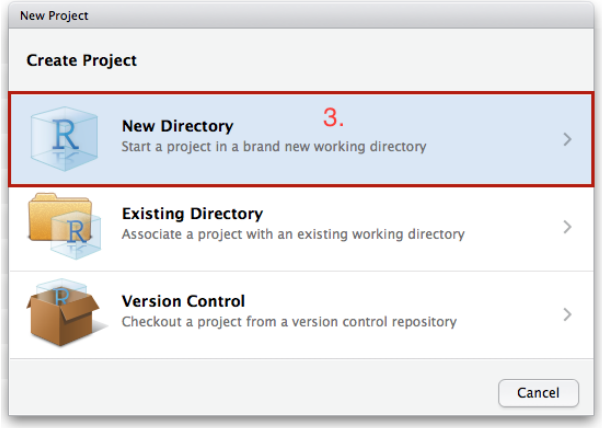
Setting up an R Package Project - Dropdown Menu

- Select the project dropdown menu and press **New Project**



Setting up an R Package Project - New Directory

- Select **New Directory**



New Project

Create Project

New Directory 3.
Start a project in a brand new working directory

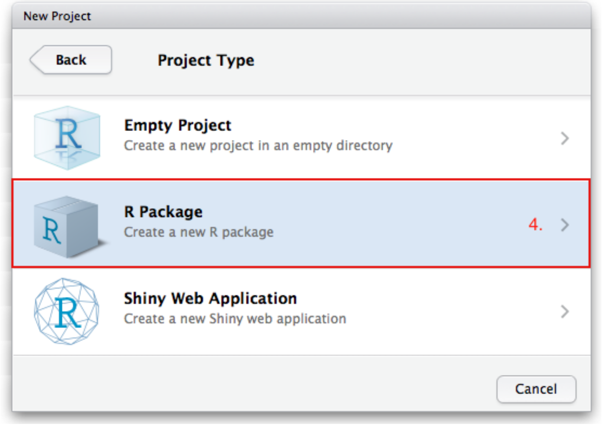
Existing Directory
Associate a project with an existing working directory

Version Control
Checkout a project from a version control repository

Cancel

Setting up an R Package Project - Project Type

- Select **R Package**


New Project

Back

Project Type

Empty Project
Create a new project in an empty directory

R Package
Create a new R package

4.

Shiny Web Application
Create a new Shiny web application

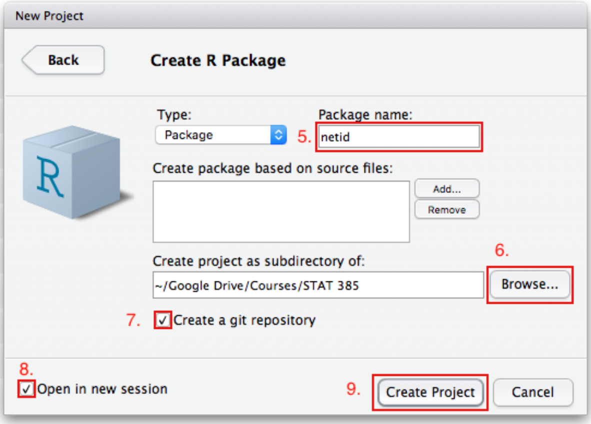
Cancel

Naming an *R* Package

- Three requirements to a package name. It must:
  1. Only contain letters, numbers and periods (.)
  2. Start with a letter
  3. Not end with a period (.)
- Examples:
  - Good: stat385, UIUCstats, AutoGrader, vis.ualize
  - Bad: data-pkg, funny_r, orgk!J@*%, -bootstrappeR, 2r, Rman., 90120r
- **Note:** Avoid using a . in package name due to S3 methods.

Setting up an R Package Project - Initialization Values

- Enter a name for your package (per rules on last slide)
- Check the **Create a git repository** and **Open in a new session**
- Press **Create Project**


New Project

Create R Package

Type: Package

5. Package name: netid

Create package based on source files:

Create project as subdirectory of: ~/Google Drive/Courses/STAT 385

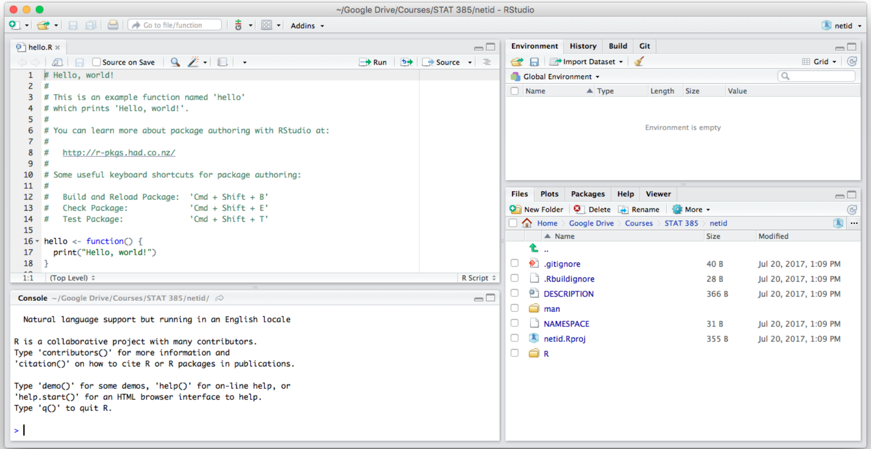
6. Browse...

7. [✓] Create a git repository

8. [✓] Open in new session

9. Create Project

View of an R Package in RStudio



```
# Hello, world!
#
# This is an example function named 'hello'
# which prints 'Hello, world!'.
#
# You can learn more about package authoring with RStudio at:
#
#   http://r-pkgs.had.co.nz/
#
# Some useful keyboard shortcuts for package authoring:
#
#   Build and Reload Package:  'Cmd + Shift + B'
#   Check Package:             'Cmd + Shift + E'
#   Test Package:              'Cmd + Shift + T'

hello <- function() {
  print("Hello, world!")
}
```

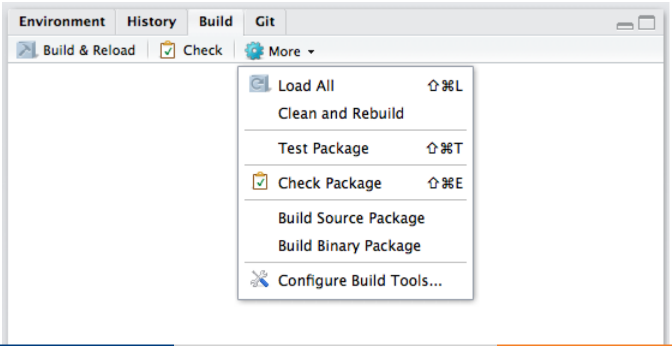
Name	Type	Length	Size	Value

Name	Size	Modified
..		
.gitignore	40 B	Jul 20, 2017, 1:09 PM
.Rbuildignore	28 B	Jul 20, 2017, 1:09 PM
DESCRIPTION	366 B	Jul 20, 2017, 1:09 PM
man		
NAMESPACE	31 B	Jul 20, 2017, 1:09 PM
netid.Rproj	355 B	Jul 20, 2017, 1:09 PM
R		

```
  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

RStudio Build Tab

- Notice the new "`Build`" Tab added in the upper right corner
- This handles:
  - Building your package
  - Creating documentation
  - Checking your package
  - Making package bundles



On the Agenda

1 R Packages
- Background

2 R Package Projects
- Creation

3 Behind the Scenes
- Package Structure
- Licensing
- R Code
- Documentation

4 Misc
- Package Creation CLI

Basic Structure of an R Package

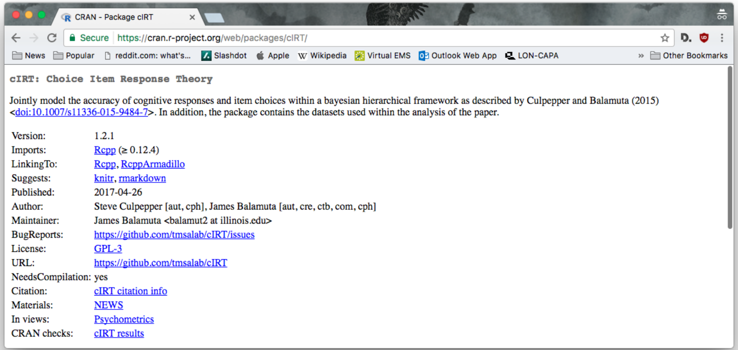
The bare minimum for an R package to exist is the following directory structure:

```
netid
|- DESCRIPTION          # Package meta
|- NAMESPACE            # Register functions
|- R/                   # All R Code
   |- hello.R
|-  man                 # Documentation
   |- hello_me.Rd       # Example Documentation
```

Crafting a DESCRIPTION file

- The DESCRIPTION file specifies the meta data to be associated with your package.
- Meta data covers the:
    - Title
    - Description
    - Authors
    - Version
    - Dependencies
    - and more!

Crafting a DESCRIPTION file - Use on CRAN



cIRT: Choice Item Response Theory

Jointly model the accuracy of cognitive responses and item choices within a bayesian hierarchical framework as described by Culpepper and Balamuta (2015) <doi:10.1007/s11336-015-9484-7>. In addition, the package contains the datasets used within the analysis of the paper.

Version:	1.2.1
Imports:	Rcpp (≥ 0.12.4)
LinkingTo:	Rcpp, RcppArmadillo
Suggests:	knitr, rmarkdown
Published:	2017-04-26
Author:	Steve Culpepper [aut, cph], James Balamuta [aut, cre, ctb, com, cph]
Maintainer:	James Balamuta <balamut2 at illinois.edu>
BugReports:	https://github.com/tmsalab/cIRT/issues
License:	GPL-3
URL:	https://github.com/tmsalab/cIRT
NeedsCompilation:	yes
Citation:	cIRT citation info
Materials:	NEWS
In views:	Psychometrics
CRAN checks:	cIRT results

cIRT CRAN page | DESCRIPTION file source on GitHub

Crafting a DESCRIPTION file - Example

```
Package: netid
Title: What The Package Does (Title Case)
Version: 0.1.0
Authors@R: c(person("First", "Last",
                    email = "first.last@somewhere.net",
                    role = c("aut", "cre")),
             person("J", "B",
                    email = "J.B@somewhere.net",
                    role = c("aut","ctb"))
Description: More about what it does
    (maybe more than one line)
    with the following line four indented inwards
Depends: R (>= 3.4.1)
License: What license is it under?
LazyData: true
```

Crafting a `DESCRIPTION` file - Example Extended

- To add package dependencies, use the `Imports:` field.
  - Package will be automatically downloaded when `install.packages()` is run.
- To suggest, but **not require**, the use of another package use `Suggests:`
  - Packages must *previously* be installed by user.

```
Imports: rvest
Suggests: testthat,
    knitr,
    rmarkdown
VignetteBuilder: knitr
```

In the case presented above, `rvest` must be present. However, users do not necessarily need `knitr` or `rmarkdown` unless they want to be able to generate the extended documentation via our vignettes.

Crafting a `DESCRIPTION` file - Specific Versions

We can also specify a specific package version by adding:

- `pkgname (>= 0.2.1)`
- `pkgname (== 0.2.1)`
- `pkgname (<= 0.2.1)`

Note, we did this to force users to have *R* greater than or equal to 3.4.1 via

```
Depends: R (>= 3.4.1)
```

Protect against `Suggests:` packages

If you do use a feature of a `Suggests:` package, make sure to protect against the call via:

```
if (!requireNamespace("suggestpkg", quietly = TRUE)) {
  stop("`suggestpkg` is required to use this function.",
       " It must be installed.")
}
```

or

```
if (requireNamespace("suggestpkg", quietly = TRUE)) {
  # Call function in suggestpkg via namespace
  suggestpkg::function_name()
} else {
  # Custom code
}
```

Crash into Licensing

- Before you begin uploading code, make sure to choose an open source license that is compatible with *R*.
- The rationale behind licensing code is to relieve you of any liabities that may arise should someone use your code.
- Furthermore, it enables you to retain rights to the work that you have done.

Licensing within R

See **Licensing** within **Writing R Extensions**

View differences between licenses at:

- **choosealicense.com**
- **tldrlegal.com**

If you want to make money or do commercial endavors, make sure to **contact a lawyer** before choosing a license!

General License

The MIT license should be sufficient for your present needs as it enables the code to be embedded in multiple projects without the worry of needing to be developed in the open. The downside is the loss of commerical ability and changes are not propogated back to the initial source. If this doesn't appeal to you consider looking at the GPL 3 license.

To use this in *R* add the `DESCRIPTION` file:

```
License: MIT + file LICENSE
```

Now, create a new file called `LICENSE` and add:

```
YEAR: <Current Year>
COPYRIGHT HOLDER: <Name of Copyright Holder>
```

Writing R Code

To use code within an R package, the code must be in function-facing form.

```
multiply =  function(x, y) {
  x * y
}
```

Not

```
x*y
```

Writing R Code - /R/

All *R* code in function form must be placed into the /R/ directory.

- /R/binary-ops.R

```
multiply =  function(x, y) {
  x * y
}
divide =  function(x, y) {
  x / y
}
```

- Try to group functions of families *together* to avoid hunting in codefiles

Adding documentation - Stone Age

- Previously, one had to manually make documentation to use within *R*'s help system.
    - e.g. `?func` or `help(func)`
- To do so, one would write `funcName.Rd` or `dataset.Rd` per function/dataset and place it in the `/man/` directory.
- Documentation was completely detached from source.
- Worse yet, the documentation had a LaTeX structure to it that was not necessarily kind.

Adding documentation - Stone Age Example

```
\name{multiply}
\alias{multiply}
\title{Multiple Two Numbers}
\usage{ multiply(x, y) }
\arguments{
\item{x}{A \code{numeric}.}
\item{y}{A \code{numeric}.}
}
\value{ Product of \code{x} and \code{y}.}
\description{
  Takes two numbers and returns their product
}
\examples{
multiply(1, 1)
multiply(4, 6)
}
```

Adding Documentation - Modern Age

- Now, a much better way to accomplish this is via: `roxygen2`.
- `roxygen2` is *R*'s take on `doxygen` and `javadoc` inline documentation.
- Uses `@keyword value` structure to write documentation
- Templating options and inherit options are available as well!
- Leads to more cohesive documentation as the worry about **Don't Repeat Yourself** decreases.

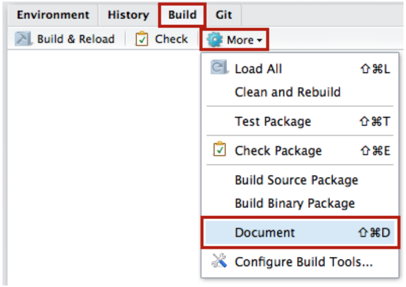
Example roxygen2

```
#' Multiple Two Numbers
#'
#' Takes two numbers and returns their product
#' @param x A \code{numeric}.
#' @param y A \code{numeric}.
#' @return Product of \code{x} and \code{y}.
#' @examples
#' multiply(1, 1)
#' multiply(4, 6)
#' \dontrun{
#' multiple(2, 3) # Avoid running expensive calls
#' }
multiply =  function(x, y) {
  x * y
}
```

Example roxygen2 - Generate Documentation

Three options:

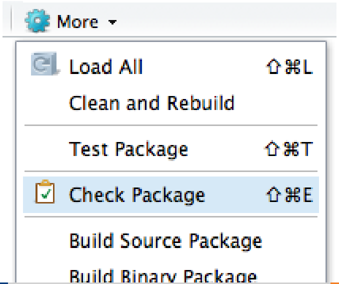
1. In RStudio, use a keyboard shortcut!
   - macOS: `Command` + `Shift` + `D`
   - Windows: `Cntrl` + `Shift` + `D`
2. Press the Document Button!
3. Via *R* console run `devtools::document()`



Check a Package

Before submitting to CRAN, make sure to check your package! To check a package use one of the **three** options:

1. In RStudio, use a keyboard shortcut!
   - macOS: `Command` + `Shift` + `E`
   - Windows: `Cntrl` + `Shift` + `E`
2. Press the `Check` button in the `Build` tab!
3. Via *R* console run `devtools::check()`



Advanced Structure of an R Package

Previously, we discussed the ***bare*** minimum for an *R* package. Now, here are a few other options to include within the *R* package structure.

```
|- .gitignore          # Ignore list for VCS
|- .Rbuildignore       # Hide files
|- .travis.yml         # Enable Continuous Integration
|- data-raw/           # Data Manipulation & Raw data
|- data/               # Cleaned data
|- test/               # Unit Tests
   |- testthat/
        |- test-hi.R   # Test file for hi
   |-  testthat.R      # Run unit tests
|-  vignettes          # Extended Documentation
    |- hello.Rmd
|-  helloRworld.Rproj  # RStudio Project File
|-  man                # Documentation
    |- hello.Rd        # Example Documentation
```

Ignore files

There are two types of ignore files. Each ignore file targets a separate system.

- `.gitignore`: avoids files from being added into the `git`.
  - Useful for preventing large data files or personal information
- `.Rbuildignore` prevents files from being included within the R package.
  - Helpful to avoid contaminating the R package structure with files or folders that would cause `R CMD check` to error.

/data-raw

The /data-raw directory is where beginning data sets (e.g. .csv, .xls) reside alongside of their cleaning script. The folder is *not* meant to ship with the package. To add this to your project, use:

```
devtools::use_data_raw()
```

Once the data has been processed and cleaned, the final version of the data set should be saved using:

```
devtools::use_data(dataset_name)
```

This compresses the data set and saves it as an R binary object via the .rda file format.

data

The `data` directory is where the final data products reside. This data will be included with your package.

The total file size of the data should be relatively small. If the data is large in size (∼1 - 2 mb), you should consider making a separate package to house the data. These kinds of packages are called **Data Packages**.

Unit Tests

- The idea with a *unit* test is to isolate one specific feature and test to make sure it works.
- For example, one might want to test to see if `multiply()` appropriately works by taking `multiply(2, 2)` and checking that it is 4.

Using a Unit Test

- Unit tests are around in manner different forms in *R*. We'll focus on using the `testthat` packages framework for unit testing.
- To setup the unit test, run:

```r
# install.packages("devtools")
devtools::use_test("binary-ops")
```

This gives:

```
# * Adding testthat to Suggests
# * Creating `tests/testthat`.
# * Creating `tests/testthat.R` from template.
# * Creating `tests/testthat/test-binary-ops.R` from template.
# * Modify `tests/testthat/test-binary-ops.R`.
```

Adding a unit test

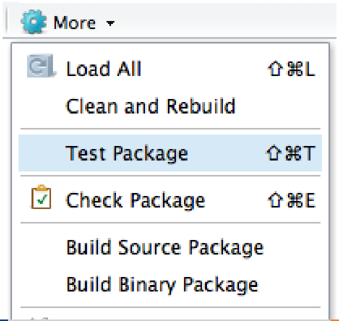
- Open: `tests/testthat/test-binary-ops.R`
- Add:

```r
context("binary-ops")

# Expect exact equality
test_that("Multiply Function", {
  expect_equal(multiply(2, 2), 4)
})

# Expect exact equality
test_that("Division Function", {
  expect_equal(divide(2, 2), 1)
})
```

Running *all* unit tests

Three options:

1. In RStudio, use a keyboard shortcut!
   - macOS: `Command` + `Shift` + `T`
   - Windows: `Cntrl` + `Shift` + `T`
2. Press the `Test` button in the `Build` tab!
3. Via *R* console run `devtools::test()`



testthat - Unit Test Checks

The following functions provide unit test checks

Functions	Description
`expect_equal()`	Equivalent to `all.equal()` (tolerance)
`expect_identical()`	Requires exact equivalence (no tolerance)
`expect_match()`	Regex match string output
`expect_is()`	Has the correct `class()` attribute
`expect_true()`	Function returns `TRUE`
`expect_false()`	Function returns `FALSE`
`expect_output()`	Matches specific `STDOUT`
`expect_message()`	Matches specific `STDERR` **messsage**
`expect_warning()`	Matches specific `STDERR` **warning**
`expect_error()`	Matches specific `STDERR` **error**

On the Agenda

1. R Packages
   - Background
2. R Package Projects
   - Creation
3. Behind the Scenes
   - Package Structure
   - Licensing
   - R Code
   - Documentation
4. Misc
   - Package Creation CLI

`package.skeleton()` - Base R

To generate a package via command line, one can use:

```r
package.skeleton(name="pkgname")
```

This function automatically setups the minimal files required for an *R* package. In addition, the function will also retrieve all active *R* functions and data sets that exist in memory and place them within `/R` in their own **separate** files as well as generate `.Rd` documentation form in `/man`.

devtools - Third Party

devtools by Wickham and Chang offers a similar functionality afforded by package.skeleton() to create an *R* package except it avoids retrieving anything in memory. The functionality to build *R* packages is split into two different functions. . .

- `devtools::create("path/to/name")`: Requires the directory to be empty and creates the package within it.
- `devtools::setup("path/to/name")`: Assumes a pre-exisiting directory and infers package name from directory space.

Both of these methods automatically creates an RStudio .rproj file with the directory name.