R Tutorial

Brad Setzler

University of Chicago

July, 2018

I. Introduction to 4 Workhorse Objects

II. Data Analysis, Visualization, and Import/Export

III. Style and Structure Rules

IV. Developing a Package

Preliminaries

To complete this tutorial, you will need to download and install:

- R (https://www.r-project.org/)
- RStudio (https://www.rstudio.com/)

Once R is installed, run these commands to install useful packages:

```
install.packages("data.table") # big data management
install.packages("ggplot2") # graphing
install.packages("lfe") # fixed effects models
install.packages("AER") # IV regressions
install.packages("styler") # code style
install.packages("devtools") # package development
install.packages("roxygen2") # package documentation
install.packages("testthat") # testing tools
```

```
## Warning: package 'data.table' was built under R version 3.3.2
```

```
## Warning: package 'ggplot2' was built under R version 3.3.2
```

Frequently Asked Questions

Before getting into R, here are the answers to the questions I have heard the most about it:

1. Is R a full programming language? *Yes.*
2. Is R popular? *Yes, it is the #10 most popular language, per TIOBE.*
3. In R code, do `a=1` and `a<-1` mean the same thing? *Yes.*
4. In R code, do `a='hello'` and `a="hello"` mean the same thing? *Yes.*
5. Can R be used for purposes other than statistical applications? *Yes, this LaTeX presentation was created by R.*

I. Introduction to 4 Workhorse Objects

Overview: 4 Workhorse Objects

This section will demonstrate the basics of working with 4 common objects in R:

1. vector;
2. data.table;
3. function; and,
4. list.

Workhorse Objects in R (1/4): vector

A *vector* is defined with the `c()` operator. Arithmetic operations are applied element-wise:

```
a_vector <- c(1, 2, 8)
a_vector * 3 / 2 + 4
```

```
## [1]  5.5  7.0 16.0
```

Once a vector has a string, all contents are converted to strings:

```
another_vector <- c(1, 2.5, "hello")
print(another_vector)
```

```
## [1] "1"     "2.5"   "hello"
```

Workhorse Objects in R (2/4): data.table

A *data.table* is a powerful data structure in R. We will study its various capabilities in the next section. Here, we only show here how to construct a `data.table` and add a column to it:

```
library(data.table)
a_data_table <- data.table(year = c(1, 1, 2, 2), value = rnorm(4))
a_data_table[, person := c("John", "Mary", "John", "Mary")]
a_data_table
```

```
##    year       value person
## 1:    1  1.5206060   John
## 2:    1 -0.1080404   Mary
## 3:    2  1.8010454   John
## 4:    2  0.8358732   Mary
```

Workhorse Objects in R (3/4): function

A *function* takes inputs (which must be named, separated by commas, and may be assigned default values using the equals sign) and returns an output:

```
a_function <- function(arg1, arg2 = 3.5) {
  intermediate <- arg1 + arg2
  final <- intermediate^2.5
  return(final)
}

a_function(2) # using the default arg2 value
```

```
## [1] 70.94254
```

```
a_function(2, 7) # overriding the default arg2 value
```

```
## [1] 243
```

Workhorse Objects in R (4/4): list

A *list* is the most general type of R object, which may include any other type of R object as an element with a string as the key:

```
a_list <- list(
  "a vector" = a_vector,
  "a data.table" = a_data_table,
  "a function" = a_function
)

a_list
```

```
## $`a vector`
## [1] 1 2 8
##
## $`a data.table`
##    year      value person
## 1:    1  1.5206060   John
## 2:    1 -0.1080404   Mary
## 3:    2  1.8010454   John
## 4:    2  0.8358732   Mary
##
## $`a function`
## function (arg1, arg2 = 3.5)
## {
##     intermediate <- arg1 + arg2
##     final <- intermediate^2.5
##     return(final)
## }
```

Workhorse Objects in R: Putting all 4 objects together

```
combined_function <- function(a.list, flag = F) {
  func1 <- function(arg.1, arg.2) {
    dt <- data.table(results = c(arg.1, arg.2) / 2)
    dt[, arg := c(1, 2)]
    return(dt)
  }

  func2 <- function(arg.1, arg.2) {
    return(arg.1 - arg.2)
  }

  arg1 <- a.list[["first argument"]]
  arg2 <- a.list[["second argument"]]

  output <- NULL
  if (flag) {
    output <- func1(arg1, arg2)
  }
  if (!flag) {
    output <- func2(arg1, arg2)
  }

  return(output)
}

this_list <- list("first argument" = 2.0, "second argument" = 3.5)

print(combined_function(this_list, flag = T))
```

```
##    results arg
## 1:    1.00   1
## 2:    1.75   2
```

Quiz #1: Workhorse Objects in R

Write a function with 3 arguments:

- The first argument is the vector `temperature` (no default);
- The second argument is the boolean `FtoC` (default `FtoC=TRUE`);
- The third argument is the list `options` (no default).

This function does the following:

- Converts temperature from Fahrenheit to Celsius if `FtoC=T`;
- Converts temperature from Celsius to Fahrenheit if `FtoC=F`;
- If there is not an argument in `options` called `option1`, it prints an error to the screen that says `"error: options$option1 must exist"` [hint: use the `stop()` command].
- Returns a `data.table` with Celsius and Fahrenheit columns.

Provide examples that work as expected when `FtoC=TRUE` and when `FtoC=FALSE`, and with/without including `option1` in `options`.

II. Data Analysis, Visualization, and Import/Export

Overview

This section shows how to analyze panel data in R:

1. Simulating panel data from a given DGP;
2. Computing aggregate statistics;
3. Plotting bivariate relationships;
4. OLS and fixed effect regressions;
5. Constructing lags and differences; and,
6. Importing/exporting results.

We will rely on data.table throughout.

Simulation in data.table

Suppose we wish to simulate panel data $(Y_{i,t}, X_{i,t})$ from the DGP:

$$Y_{i,t} = \alpha + \beta X_{i,t} + t\kappa + \mu_i + \epsilon_{i,t}, \quad \epsilon_{i,t} \sim_{iid} \mathcal{N}(0, \sigma^2)$$

where X depends on μ and t (more on this later). It can be simulated as:

```
library(data.table)
set.seed(101) # Always set the seed before simulating data
NI <- 1000 # Number of individuals
NT <- 5 # Number of time periods
aalpha <- 8
bbeta <- 1
kkappa <- 1
ddata <- data.table(expand.grid(id = 1:NI, time = 1:NT))
ddata[, mu := rnorm(n = 1), by = "id" ] # draw one mu for each id.
ddata[, X := -mu - time * .5 + rnorm(n = nrow(ddata))] # draw X on mu,t.
ddata[, epsilon := rnorm(nrow(ddata)) ] # draw one epsilon for each observation.
ddata[, Y := aalpha + bbeta * X + time * kkappa + mu + epsilon] # construct Y.
ddata[c(1:3)] # print the first 3 rows.
```

```
##    id time         mu          X       epsilon        Y
## 1:  1    1 -0.3260365  0.32923230  0.003604515 9.006800
## 2:  2    1  0.5524619 -3.14993253 -0.984257731 5.418272
## 3:  3    1 -0.6749438  0.08045926  0.571400889 8.976916
```

Aggregation in data.table

We now wish to summarize how Y varies across values of time and X. Summarizing Y over time is easy because t is discrete. Let's compute the mean, 25th percentile, maximum, and count (`.N`) by time:

```
obsdata <- ddata[, .(id, time, Y, X)] # data actually observed
Y_time_stats <- obsdata[,
  list(
    Y_mean = mean(Y), # mean
    Y_q25 = quantile(Y, .25), # 25th percentile
    Y_max = max(Y), # max
    count = .N # number of observations used in this statistic
  ),
  by = "time"
]
Y_time_stats <- Y_time_stats[order(time)] # putting data in order of time
Y_time_stats[] # printing
```

```
##    time    Y_mean    Y_q25    Y_max count
## 1:    1  8.502082 7.532084 13.42059  1000
## 2:    2  9.078298 8.071414 14.76886  1000
## 3:    3  9.495968 8.455407 13.75927  1000
## 4:    4 10.011724 9.052344 15.25380  1000
## 5:    5 10.554132 9.582877 14.42392  1000
```

Aggregation in data.table (cont.)

We can aggregate across X similarly to how we aggregated across t, but first we will need to discretize X into bins. There are many ways to group X values into bins. One way is to group X by nearest decile:

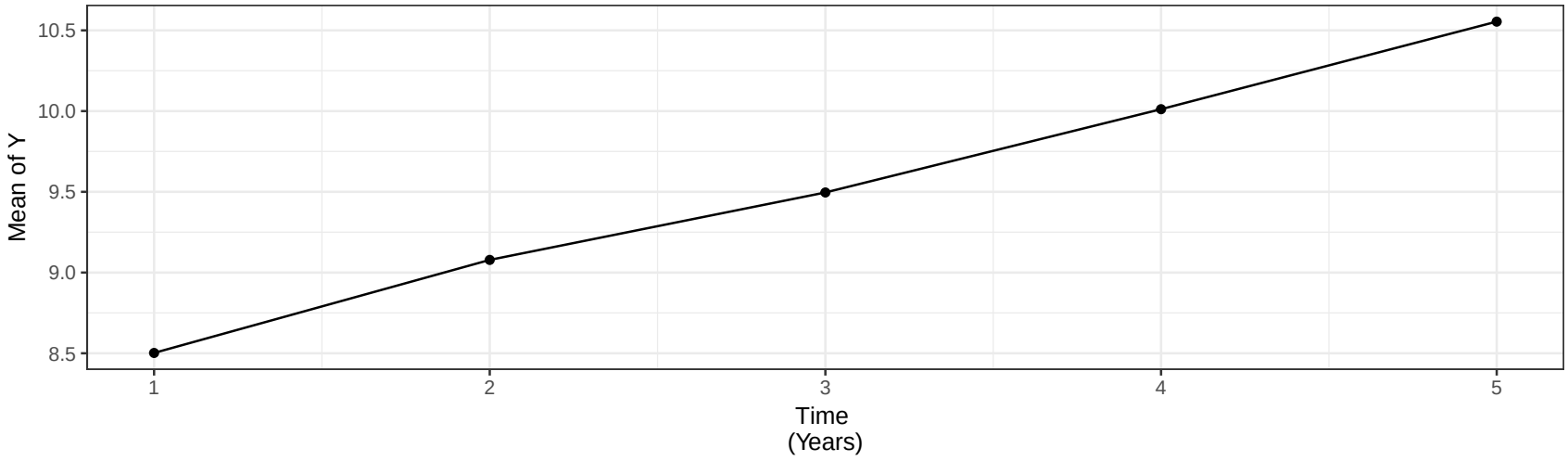
```
obsdata[, X_quantile_bin := round(ecdf(X)(X), 1)]
Y_X_stats <- obsdata[,
  list(
    Y_mean = mean(Y), # mean
    Y_q25 = quantile(Y, .25), # 25th percentile
    Y_max = max(Y), # max
    count = .N # number of observations used in this statistic
  ),
  by = "X_quantile_bin"
]
Y_X_stats <- Y_X_stats[order(X_quantile_bin)] # putting data in order of X
Y_X_stats[1:3] # print first 3 rows
```

```
##    X_quantile_bin   Y_mean    Y_q25    Y_max count
## 1:            0.0 8.687269 7.773793 12.10929   250
## 2:            0.1 9.139086 8.070127 14.09781   499
## 3:            0.2 9.298968 8.288466 13.90786   501
```

Visualization in ggplot2

ggplot2 allows for highly customized plotting in R.

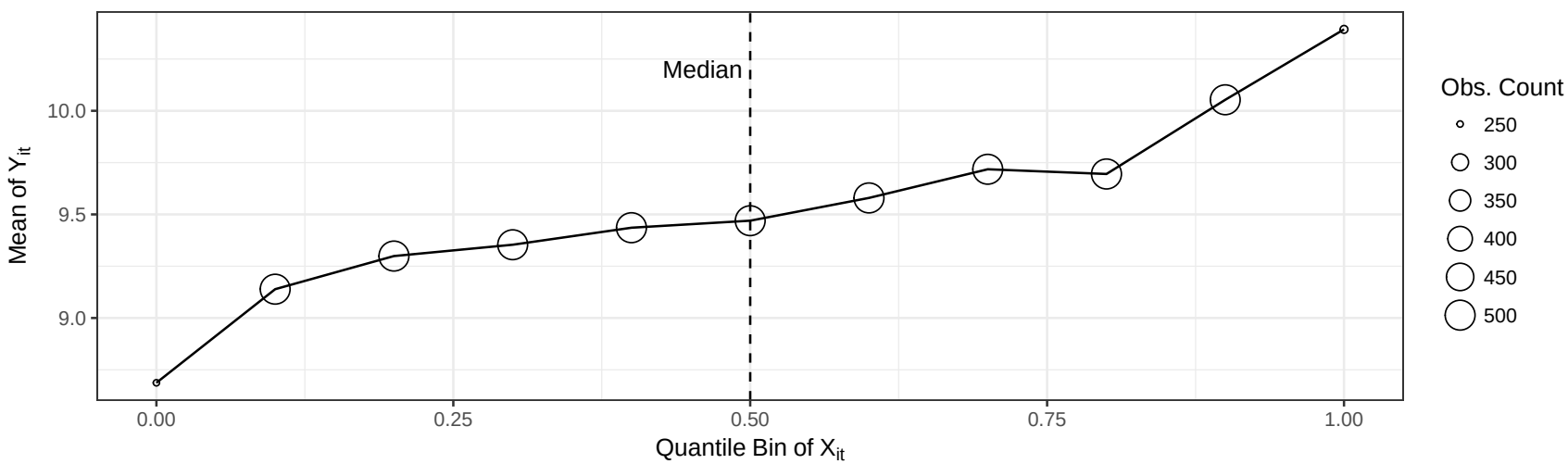
```
library(ggplot2)
ggp_time <- ggplot(aes(x = time, y = Y_mean), # x- and y-axis variables
  data = Y_time_stats
) + # set the data source
  geom_point() + # this says to display points
  geom_line() + # this says to display a line
  theme_bw() + # attractive black-and-white color scheme
  ylab("Mean of Y") + xlab("Time \n(Years)") # axis labels
ggp_time # print figure
```



Visualization in ggplot2 (cont.)

We will now visualize the relationship between Y and X, using it as an opportunity to show off some of the customization available in ggplot2:

```
ggp_X <- ggplot(aes(x = X_quantile_bin, y = Y_mean), data = Y_X_stats) +
  geom_point(aes(size = count), shape = 1) + # circles that scale with count
  geom_line() + # this says to display a line
  geom_vline(xintercept = .5, linetype = "dashed") + # vertical line at median
  annotate("text", x = .46, y = 10.2, label = "Median") + # annotate median
  theme_bw() + # attractive black-and-white color scheme
  labs(size = "Obs. Count") + # set legend title
  ylab(expression(Mean ~ of ~ Y[it])) + # math label
  xlab(expression(Quantile ~ Bin ~ of ~ X[it])) # math label
ggp_X # print figure
```



Regression Analysis: OLS using lm

Now suppose we wish to estimate β, which is 1 in the simulation. We will use `lm` ("linear model") and R's regression formula notation:

```
OLS_formula_base <- as.formula("Y ~ X") # define the formula
OLS_result_base <- lm(formula = OLS_formula_base, data = obsdata) # regression
OLS_coef_base <- coef(summary(OLS_result_base)) # extract results as matrix
OLS_coef_base # print results
```

```
##               Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) 9.8629647 0.03010966 327.56815 0.000000e+00
## X           0.2314779 0.01417932  16.32503 2.078508e-58
```

The coefficient on X, 0.23, should have been 1, so OLS is downward-biased. Now let's try controlling for a linear time trend:

```
OLS_formula_time <- as.formula("Y ~ X + time") # define the formula
OLS_result_time <- lm(formula = OLS_formula_time, data = obsdata) # regression
OLS_coef_time <- coef(summary(OLS_result_time)) # extract results as matrix
OLS_coef_time # print results
```

```
##               Estimate Std. Error   t value Pr(>|t|)
## (Intercept) 7.9839016 0.04012428 198.97930        0
## X           0.5558655 0.01235384  44.99534        0
## time        0.7826190 0.01359122  57.58269        0
```

Regression Analysis: Fixed Effects using lm and felm

The coefficient estimate on X is downward-biased because $X_{i,t}$ is negatively correlated with the fixed effect μ_i. We can control for the fixed effect in OLS by including a factor indicator for each id:

```
OLS_formula_fe <- as.formula("Y ~ X + time + as.factor(id)") # define the formu
OLS_result_fe <- lm(formula = OLS_formula_fe, data = obsdata) # regression
OLS_coef_fe <- coef(summary(OLS_result_fe)) # extract results as matrix
OLS_coef_fe[1:3, ] # too much output to show everything
```

```
##                Estimate Std. Error  t value     Pr(>|t|)
## (Intercept) 7.797695 0.43736076 17.82898 1.768675e-68
## X           1.038548 0.01535736 67.62544 0.000000e+00
## time        1.024771 0.01243031 82.44134 0.000000e+00
```

This is the right answer! However, it requires a variable for each individual, so it is infeasible for big data. Use `felm` from *lfe* for big data:

```
library(lfe)
felm_formula <- as.formula("Y ~ X + time | id") # define the formula
felm_result <- felm(formula = felm_formula, data = obsdata) # regression
felm_result # use felm_result$STATS to see all the output
```

```
##     X  time
## 1.039 1.025
```

Importing and Exporting

R has many available file formats for reading and writing data.

- In general, always try to use CSV. CSV files are lightweight and can be written and read by any other language (Stata, Python, Excel, etc).
- In the rare event that you need to save a full list structure, use RDS files to save lists one at a time. These can only be openend by R.
- Try to never use RData or save a workspace. This is prone to generate conflicts in your workspace since it can include any objects, including objects you did not mean to save or load.

Here is an example of writing and then reading the observed data as CSV:

```
# Write to CSV. Always omit row names and replace missings with a space.
write.csv(obsdata, file = "observed_data.csv", row.names = F, na = " ")
# Read data from CSV and immediately set it as a data.table.
obsdata <- setDT(read.csv(file = "observed_data.csv"))
```

Tip: always use setDT() instead of as.data.table() to convert an object into a data.table, since setDT saves on memory by not making a copy.

Here is how to save a ggplot2 object as a PDF figure:

```
ggsave(ggp_time, file = "ggplot_time.pdf", height = 5, width = 8)
```

Quiz #2: Data Analysis and Visualization

Simulate data again from the model:

$$Y_{i,t} = \alpha + \beta X_{i,t} + t\kappa + \mu_i + \epsilon_{i,t}.$$

Instead of being iid, let $X_{i,t}$ also depends on $\epsilon_{i,t}$. Furthermore, draw an observed variable $Z_{i,t}$ which is independent of $\epsilon_{i,t}$, and have $X_{i,t}$ also depend on $Z_{i,t}$. The observed data is now $(Y_{i,t}, X_{i,t}, Z_{i,t})$.

Demonstrate how to estimate β using the command `ivreg` from package *AER*, where we treat $X_{i,t}$ as the endogenous variable and $Z_{i,t}$ as the instrumental variable. Compare the `ivreg` result to using`lm` ingoring $Z_{i,t}$.

Then, use felm to estimate the same instrumental variables regression more efficiently. [Hint: In felm, the instrumental variables notation looks like `felm_formula <- as.formula("Y ~ time | id | (X ~ Z)")`.]

Export the results on coefficient estimates, standard errors, and p-values from `lm`, `ivreg`, and `felm` as CSV files. [Note that each of these organizes results somewhat differently.]

III. Style and Structure Rules

Overview

The tidyverse coding style, which is derived from Google's R style guide, is available here: `http://style.tidyverse.org/`. However, it does not cover issues of scoping or when to check for errors, which are key.

Note: The *styler* package can re-write your code to satisfy the style guide. It does things like automatically replace equals (a = 3) with arrows (a <- 3), and replace single quotes ('hello') with double quotes ("hello"). It has an RStudio plug-in for ease of use. I highly recommend using styler, especially when you are new to R and still learning its style. (All of my code here was edited by styler.)

On the next few slides, I will cover what I consider to be the 3 most important rules of R style and organization:

1. Use informative names;
2. Manually control function scoping; and,
3. Fill functions with manual error checks.

Rule #1: Use informative names

Consider the function that calculates the rents collected by workers in the Lamadon, Mogstad, and Setzler (2018) model using the wages and the preference parameter β. You could write this as:

```
function1 <- function(x, y) {
  return(x / (1 + y))
}
```

It is difficult to see how this relates to the worker rents formula. Try:

```
compute_worker_rents <- function(sum_of_wages, beta_preference) {
  return(sum_of_wages / (1 + beta_preference))
}
```

Two other notes on names:

- Multiple words in a name should be separate by the underscore (worker_rents) rather than the dot (worker.rents).
- Do not re-use names. For example, "c" is used to make vectors, so don't call any of your variables c.

Rule #2: Manually control function scoping

When a function requires a variable that is not passed into the function as an argument, it will search outside of the function into the next highest scope. This can be very dangerous. Here is an example (courtesy of Darren Wilkinson's blog):

```
a <- 1
f <- function(x) {
  return(a * x)
}
g <- function(x) {
  a <- 2
  return(f(x))
}
g(3)
```

```
## [1] 3
```

I wanted a=2 to be true when g(3) is evaluated on the last line, but f(x) searched for a in the wrong scope and used a=1 instead.

Rule #2: Manually control function scoping (cont.)

There are a number of ways to prevent the mistake on the previous slide, but the simplest and most reliable way is to pass 100% of objects used by the function as function arguments:

```
a <- 1
f <- function(x, a) {
  return(a * x)
}
g <- function(x) {
  a <- 2
  return(f(x, a))
}
g(3)
```

```
## [1] 6
```

Now that f includes a as an argument, there is no problem. 100% of objects used in a function should be passed into it as arguments.

Rule #3: Fill functions with manual error checks

Here, I will make two mistakes:

```
f <- function(data, a) {
  data[, var2 := a]
  return(data)
}
dd <- data.table(var1 = c(1, 2))
f(data, a = 2) # I meant to write data=dd here!
```

```
## Error in `:=`(var2, a): Check that is.data.table(DT) == TRUE. Otherwise, :=
```

```
f(data = dd, a = c(2, 3, 4)) # a has too many numbers here!
```

```
## Warning in `[.data.table`(data, , `:=`(var2, a)): Supplied 3 items to be
## assigned to 2 items of column 'var2' (1 unused)
```

- Why doesn't the first error just say "data does not exist"? It turns out this is because data is a function in base R, so data actually already exists! (If I had known this, I would not have called the object data, per Rule #1 above.)
- Why does the second give a warning instead of an error? This is an example of R trying to help but causing a disaster!

Rule #3: Fill functions with manual error checks (cont.)

```
f <- function(data, a) {
  if (!is.data.table(data)) {
    stop(sprintf("Input `data` must be data.table but is %s.\n", class(data)))
  }
  if (!is.numeric(a)) {
    stop(sprintf("Input `a` must be numeric but is %s.\n", class(a)))
  }
  if (!(length(a) == 1 | length(a) == nrow(data))) {
    stop(sprintf(
      "Input `a` is of length %i \nwhile input `data` is of length %i.\n",
      length(a), nrow(data)
    ))
  }
  data[, var2 := a]
  return(data)
}
dd <- data.table(var1 = c(1, 2))
f(data, a = 2)
```

```
## Error in f(data, a = 2): Input `data` must be data.table but is function.
```

```
f(data = dd, a = c(2, 3, 4))
```

```
## Error in f(data = dd, a = c(2, 3, 4)): Input `a` is of length 3
## while input `data` is of length 2.
```

Quiz #3: Style and Structure

Rewrite the codes you wrote for Quiz #1 and #2 to satisfy all of the style and structure rules. [Hint: the `style_active_file()` function from the *styler* package can do a lot of the work.]

Save these as separate files so you can compare before/after.

Note that coming up with useful error checks is as much an art as a science, as you need to anticipate possible mistakes you or another user could make.

IV. Developing a Package

Why Use Packages?

- Objects are organized and updated to the latest version;
- Objects are documented so you know what they do;
- Clear separation of the tools (what you use) from the application (what you're analyzing with those tools);
- Reproducibility since anyone can install/test the package;
- Nice structure/methods for testing that the tools work.

Creating a .RProj and Organizing Package

This is easy with RStudio. Just go to:

File -> New Project -> New Directory -> R Package

Then, give your directory and R package a name, and it will create a skeleton package in the appropriate structure.

The main directories of an R package are:

- R: this contains your .R scripts run when your package is loaded;
- man: this contains your documentation .md manuals; and,
- inst: this contains tests and other things that should NOT be run when the package is loaded (note: you will have to create this folder yourself since RStudio does not do this automatically).

To launch the package, double-click on the .RProj icon.

Creating a .RProj and Organizing Package (cont.)

Two files contain essential information:

- DESCRIPTION: You need to add a line that lists the other packages that your package uses, for example:

```
Imports: data.table, lfe
```

- NAMESPACE: this file lists objects that will be exported (that is, available in the workspace) when your package is loaded. For example, if you have a function called `compute_worker_rents`, and your package relies on data.table, you should add lines in NAMESPACE that say:

```
import(data.table)

export(compute_worker_rents)
```

Writing Documentation

Example of a function with documentation:

```
#' Function to compute worker rents
#'
#' @description
#' This is a function that computes worker rents using the formula
#'  derived by Lamadon, Mogstad, and Setzler (2018).
#'  It returns the rents that go to the worker as a scalar.
#'
#' @param sum_of_wages Sum of all wages in the sample (numeric).
#' @param beta_preference The beta parameter estimate (numeric).
#'
compute_worker_rents <- function(sum_of_wages, beta_preference) {
  return(sum_of_wages / (1 + beta_preference))
}
```

When you run the command `roxygenize()` from the package *roxygen2*, it compiles all documentation into help manuals. Then, if you need to see the documentation for this function, you simply use the command `?compute_worker_rents` and a help manual appears.

Building, checking, compiling, and installing

- To build your package, just click the Build & Reload button in RStudio.
- To check your package for errors (even missing documentation), use the Check button in RStudio. Read the log to see what mistakes you have made.
- To compile your package, go to the command line (outside of R) and run `R CMD BUILD name_of_package`, where name_of_package is whatever your package is called. This will create a compressed file called name_of_package.tar.gz.
- To install your compiled package (which should be called name_of_package.tar.gz), use `R CMD INSTALL name_of_package`.

Quiz on Package Development

Find quiz4_materials.R. This is a set of functions that I have written to make highly customized LaTeX tables in R. Your assignment is to convert these functions into a package called *textables*. This will require the following:

- organize the functions into a package with appropriate DESCRIPTION and NAMESPACE (author: Bradley Setzler);
- functions need correct style, structure, error checks, comments, and documentation;
- add examples and tests to the inst/ folder;
- make sure it passes Check and can be compiled/installed.