Data Wrangling in R: Programming and Automation

Clay Ford

Fall 2018

Agenda

- Intro to writing functions in R
- Using functions to automate tasks

R is comprised of functions

- Just about everything in R is a function.
- Existing functions come in packages. For example:
  - The `read.csv` function is in the `utils` package
  - The `lm` function is in the `stats` package
  - The `+` operator is a function in the `base` package
  - The `ggplot` function is in the `ggplot2` package
- R allows us to easily create our own functions during an R session that are not part of a package.

What is a function?

- In math, a function assigns an output to a given input. For example:

$$f(x) = 2x + 6$$

- An input of $x = 2$ results in an output of 10.

$$f(x = 2) = 2 \cdot 2 + 6 = 10$$

- Functions in R are similar

Writing functions in R

- We can define the function on the previous slide as follows in R:

```
f <- function(x) 2*x + 6
f(x = 2)
```

```
## [1] 10
```

- The `function` function allows you to define a function!

Some basic details on creating functions

- The `function` arguments are the inputs. We can name them whatever we want (following R naming conventions).
- To create a reusable function, we have to assign it to a name. We can name it whatever we want.
- Once created, the function is available to use for the remainder of the R session.
- Same example as before using different argument and name

```
exFunction <- function(value) 2*value + 6
exFunction(value = 2)
```

```
## [1] 10
```

Functions can have multiple arguments

- For example, the formula for BMI has two inputs:

$$\text{BMI} = \frac{\text{weight(lb)}}{\text{height(in)}^2} \times 703$$

- A possible R function with two arguments:

```
bmi <- function(w,h) w/(h^2) * 703
bmi(w = 205, h = 69)
```

```
## [1] 30.2699
```

Functions can have more than one line of code

- Place multiple lines between curcly braces: `{}`
- Example: check to make sure numbers are entered; if not, return an error

```
bmi <- function(w,h){
  if(!is.numeric(w) || !is.numeric(h))
    stop("enter numbers")
  w/(h^2) * 703}
bmi(w = 205, h = "5 ft 9 in")
```

```
## Error in bmi(w = 205, h = "5 ft 9 in"): enter numbers
```

- Functions with more than one line of code return the last line.

Functions can return multiple outputs

- A function can return more than one thing. For example, return original height and weight along with BMI in a data frame:

```
bmi <- function(w,h){
  b <- w/(h^2) * 703
  data.frame(weight = w, height = h, bmi = b)}
bmi(w = 205, h = 69)
```

```
##   weight height      bmi
## 1    205     69 30.2699
```

R functions not limited to math formulas

- Here's a function that returns a difference in means along with a 95% CI on the difference

```
meanDiff <- function(var1, var2){
  tt <- t.test(var1, var2)
  ci <- tt$conf.int  # get CI of diff in means
  dif <- mean(var1) - mean(var2)
  c("CI lower" = ci[1], "Difference" = dif,
    "CI upper" = ci[2])}
```

```
meanDiff(var1 = x1, var2 = x2)
```

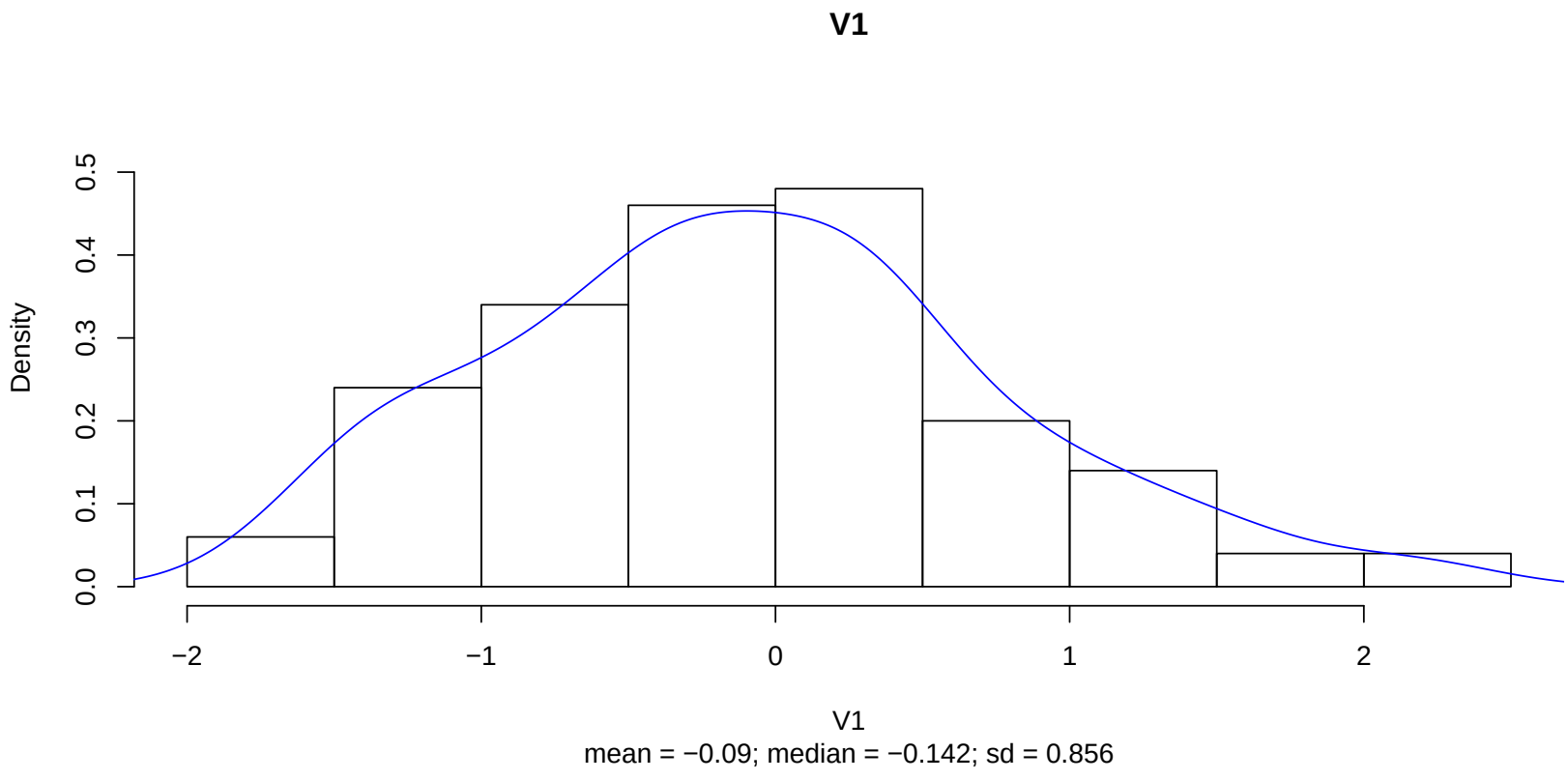
```
##   CI lower Difference   CI upper
##  -2.467935  -1.944420  -1.420906
```

Debugging functions

- There are lots of debugging features in R/RStudio. We'll talk about just one: `debug`
- How it works:
  1. Call `debug` on your function
  2. Run your function with some input. This opens the debugging window.
  3. Click the Next button to step through each line of code. Any objects created in the function will be available to inspect and manipulate
  4. When finished, either fix your code and reassign the function, or call `undebug` on your function.
- We'll give a demo in the R script

Motivating example for creating functions

- Let's say I have 50 columns of numeric data in the data frame `dat` and I want to make the following plot for each column



The code for the plot

- The plot on the previous slide was created with this code.

```
x <- dat[["V1"]]
h <- hist(x, plot = FALSE)
info <- paste0("mean = ", round(mean(x),3),
               "; median = ", round(median(x),3),
               "; sd = ", round(sd(x),3))
plot(h, freq = FALSE,
     ylim = c(0, 1.2*max(h$density)),
     sub = info, main = "V1", xlab = "V1")
lines(density(x), col = "blue")
```

- I don't want to copy-and-paste that code 50 times! Or submit it 50 times, each time changing the variable!

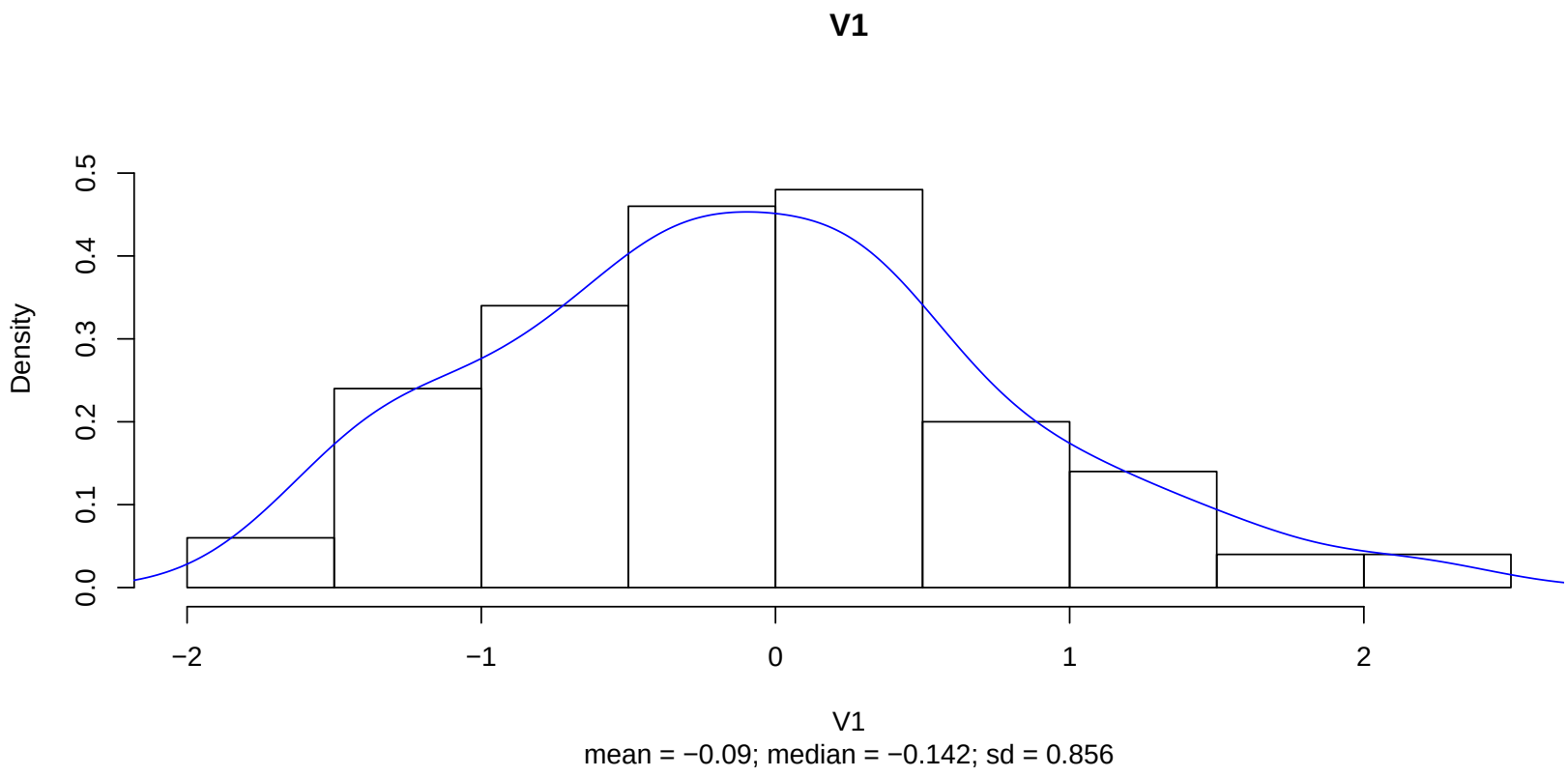
The first step: create a function

```
histDensity <- function(v){
  x <- dat[[v]]
  h <- hist(x, plot = FALSE)
  info <- paste0("mean = ", round(mean(x),3),
                 "; median = ", round(median(x),3),
                 "; sd = ", round(sd(x),3))
  plot(h, freq = FALSE,
       ylim = c(0, 1.2*max(h$density)),
       sub = info, main = v, xlab = v)
  lines(density(x), col = "blue")
}
```

- Notice that we basically took our existing code, inserted it between `function(v){` and `}`, replaced `"V1"` with `v`, and assigned it the name `histDensity`

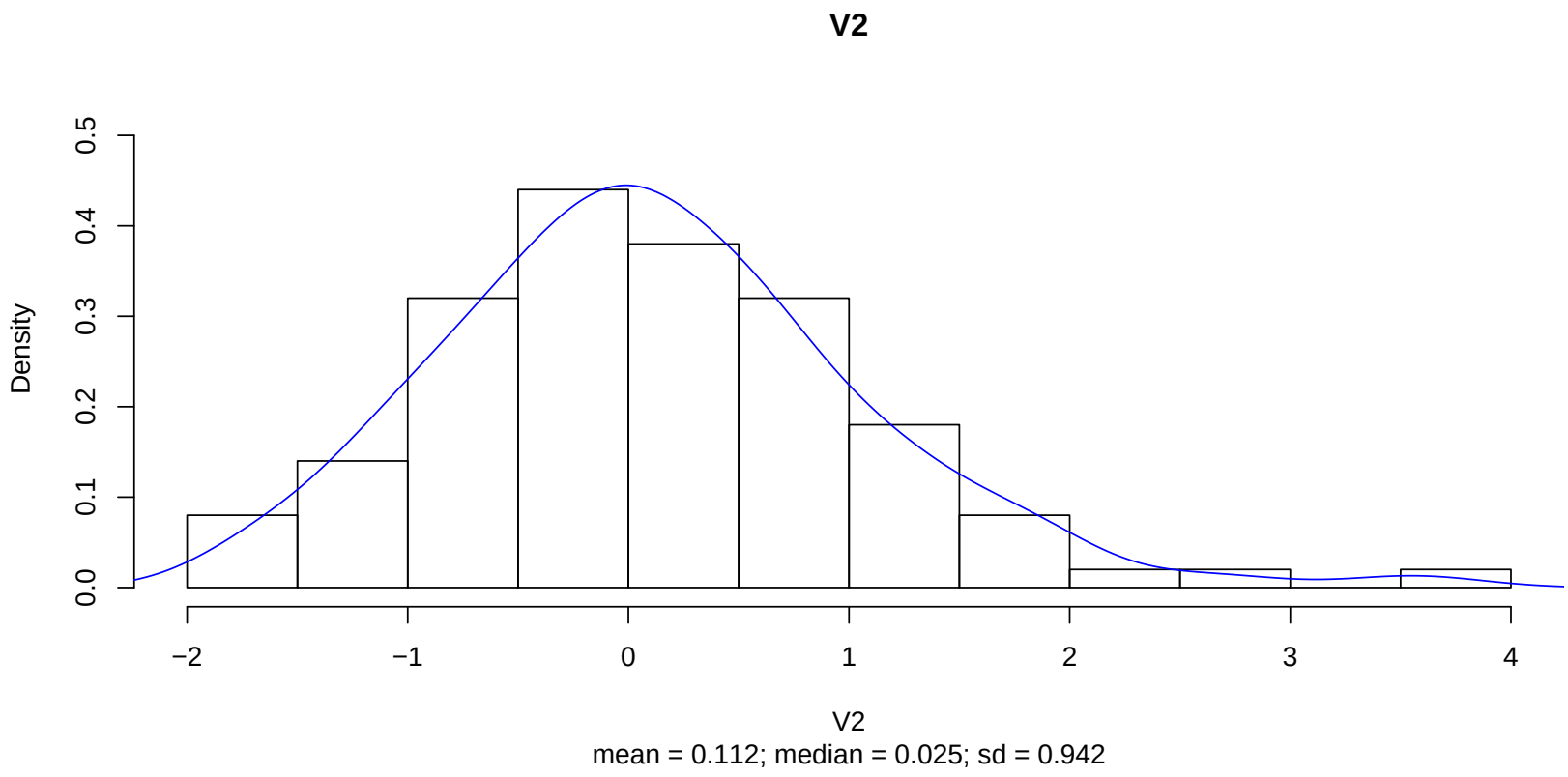
The function at work for V1

```
histDensity("V1")
```



The function at work for V2

```
histDensity("V2")
```



But this leads to another problem

- Do we want to do this?

```
histDensity("V1")
histDensity("V2")
histDensity("V3")
histDensity("V4")
histDensity("V5")
 # etc. all the way to "V50"
```

- No.

Second step: *apply* the function to the variable names

- R provides facilities for applying functions to vectors of values.
- In this example we could create a vector of variables names using the `names` function

```
vars <- names(dat)
head(vars)
```

```
## [1] "V1" "V2" "V3" "V4" "V5" "V6"
```

- Then we could apply the function to the `vars` vector to generate 50 plots

```
lapply(vars, histDensity)
```

Another motivating example

- Let's say I have 50 columns of numeric data in the data frame `dat` and I want to generate the following summary for each column

```
##             V1
## max      2.182
## 90%      1.046
## mean    -0.090
## 50%     -0.142
## 10%     -1.249
## min     -1.699
```

The code for the summary

```
x <- dat[["V1"]]
qn <- quantile(x, probs = c(0.1, 0.5, 0.9))
s <- rev(round(c(min = min(x), qn[1], qn[2],
                 mean = mean(x), qn[3],
                 max = max(x)),3))
m <- matrix(data = s)
rownames(m) <- names(s)
colnames(m) <- "V1"
m
```

- I don't want to copy-and-paste that code 50 times, or submit it 50 times each time changing the variable

The first step: create a function

```
mySummary <- function(v){
  x <- dat[[v]]
  qn <- quantile(x, probs = c(0.1, 0.5, 0.9))
  s <- rev(round(c(min = min(x), qn[1], qn[2],
                   mean = mean(x), qn[3],
                   max = max(x)),3))
  m <- matrix(data = s)
  rownames(m) <- names(s)
  colnames(m) <- v
  m
}
```

- Notice that we basically took our existing code, inserted it between `function(v){` and `}`, replaced `"V1"` with `v`, and assigned it the name `mySummary`

The function at work for V1

```
mySummary("V1")
```

```
##            V1
## max     2.182
## 90%     1.046
## mean   -0.090
## 50%    -0.142
## 10%    -1.249
## min    -1.699
```

The function at work for V2

```
mySummary("V2")
```

```
##            V2
## max     3.561
## 90%     1.245
## mean    0.112
## 50%     0.025
## 10%    -1.013
## min    -1.729
```

Second step: *apply* the function to the variable names

- Create a vector of variables names using the `names` function

```
vars <- names(dat)
head(vars)
```

```
## [1] "V1" "V2" "V3" "V4" "V5" "V6"
```

- Then apply the function to the `vars` vector to generate 50 summaries

```
lapply(vars, mySummary)
```

- The printed result will be a list of 50 summaries

The basic recipe to automate repetitive tasks

1. Write a function
2. Apply function to data (a vector, columns of a data frame, elements in a list)

The many ways of applying functions

- `lapply` returns a list
- `sapply` attempts to simplify results into a vector or matrix
- `apply` is for applying functions to rows/columns of a matrix (or an array)
- `map` from the `purrr` package is similar to `lapply` but with some extra functionality
- `purrr` provides several versions of `map` to control output, such as `map_chr`, `map_dbl`, `map_df`

What about `for` loops?

- Traditional programming languages will often use `for` loops to repeat tasks
- `for` loops can also be used in R. Example:

```
for(i in 1:ncol(dat)){
  histDensity(names(dat)[i])
}
```

- `for` loops in R can be slow if lots of memory allocation is happening with each iteration
- Some R users frown on loops
- My advice: if `for` loops make sense to you and they run fast enough for you, use them

for loops vs apply/map

One argument for using apply/map is they make storing output easier. Compare the two methods below:

- Storing output of mySummary with lapply

```
s.out <- lapply(names(dat),mySummary)
```

- Storing output of mySummary with for loop

```
s.out <- vector(mode = "list", length = ncol(dat))
for(i in 1:ncol(dat)){
  s.out[[i]] <- mySummary(names(dat)[i])
}
```

Anonymous functions

- Functions don't always have to be created and stored in memory
- `apply` and `map` functions work with *anonymous* functions, which are simply functions created on-the-fly
- Example: calculate standard error for random draws from a N(0,1) distribution, with increasing sample size

```
n <- seq(20,80,by=20)
sapply(n, function(x)sd(rnorm(n = x))/sqrt(x))
```

```
## [1] 0.2766387 0.1441126 0.1055953 0.1272554
```

Know when to apply/map

- Many functions in R are already vectorized, which mean they essentially have a built-in looping mechanism
- Example: do not need to "apply" `sqrt` to `x`; the `sqrt` function is vectorized

```
x <- c(4,9,16,25)
sqrt(x)
```

```
## [1] 2 3 4 5
```

- The help page for `sqrt` states that it works on a numeric vector
- A function that is not vectorized is `read.csv`; it works on a single file

Viewing output in R Markdown

- The output of a function applied to a large data frame or vector can overwhelm the console or plotting window
- One option is to run the code in R Markdown
- R Markdown allows you to combine R code, output, plots and exposition in one document
- Easiest way to get started:
  1. File. . . New File. . . R Markdown. . .
  2. Add a Title, leave "Document" and "HTML" selected, click OK
  3. Review the document with sample code/text and click Knit
  4. All code, output, plots and text are combined into one HTML document

Let's go to the R script

For the remainder of the workshop we'll work through examples and exercises for writing and applying R functions.

References

- Wickham, H. and Grolemund, G. (2017). *R for Data Science*, O'Reilly: http://r4ds.had.co.nz/ (chs 17 - 21)
- Wickham, H. (2014) *Advanced R*, Chapman & Hall: http://adv-r.had.co.nz/
- Grolemund, G. (2014). *Hands-On Programming with R*, O'Reilly.
- Matloff, N. (2011). *The Art of R Programming*, Starch Press.

Thanks for coming

- For statistical consulting: statlab@virginia.edu
- Sign up for more workshops or see past workshops: http://data.library.virginia.edu/training/
- Register for the Research Data Services newsletter to be notified of new workshops: http://data.library.virginia.edu/newsletters/