Functions

David Gerard and Jane Wall

2019-01-14

Learning Objectives

- Piping
- Creating Functions
- Chapters 17 through 19 in RDS

Magrittr and the %>%

- When doing multiple operations to a dataset, we can:
  - create intermediate objects
  - overwrite the original object
  - nest commands
  - use pipe
- Suppose we want to calculate the average kpg (kilometers per gallon) for cars that are automatic and those that are manual. Since the data are given in mpg (miles per gallon) we need to accomplish the following tasks:
  - Convert the mpg to kpg
  - Group the data by transmission type
  - Calculate average mpg within each type.
- Load data and tidyverse:

```
suppressMessages(library(tidyverse))
data("mtcars")
km_in_mile <- 1.60934
```

- Create intermediate objects

```
mtcars1 <- mutate(mtcars, kpg = mpg * km_in_mile)
mtcars2 <- group_by(mtcars1, am)
summarize(mtcars2, ave_mpg = mean(mpg))
```

```
## # A tibble: 2 x 2
##      am ave_mpg
##   <dbl>   <dbl>
## 1     0    17.1
## 2     1    24.4
```

- Overwrite the original object

```
mtcars <- mutate(mtcars, kpg = mpg * km_in_mile)
mtcars <- group_by(mtcars, am)
summarize(mtcars, ave_mpg = mean(mpg))
```

```
## # A tibble: 2 x 2
##      am ave_mpg
##   <dbl>   <dbl>
## 1     0    17.1
## 2     1    24.4
```

- Nest commands

```
summarize(group_by(mutate(mtcars, kpg = mpg * km_in_mile), am), ave_mpg = mean(mpg))
```

```
## # A tibble: 2 x 2
##      am ave_mpg
##   <dbl>   <dbl>
## 1     0    17.1
## 2     1    24.4
```

- Use pipe

```
mtcars %>%
  mutate(kpg = mpg * km_in_mile) %>%
  group_by(am) %>%
  summarise(ave_mpg = mean(mpg))
```

```
## # A tibble: 2 x 2
##      am ave_mpg
##   <dbl>   <dbl>
## 1     0    17.1
## 2     1    24.4
```

- **Exercise**: In the `mtcars` dataset, use the four different techniques for chaining to calculate the median weight (variable `wt` in lbs) for all cars that have at least 6 cylinders (variable `cyl`) and the median weight for all cars that have less than 6 cylinders.

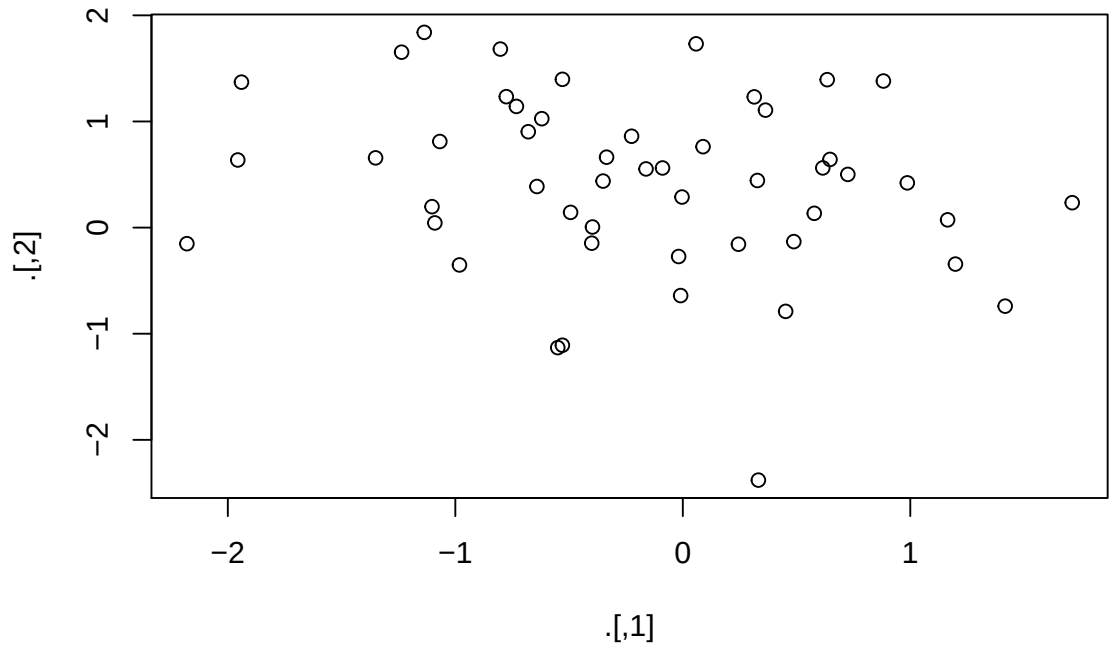
Other magrittr tools

- T-pipe, %T>% returns left-hand side instead of right-hand side

```
library(magrittr)
```

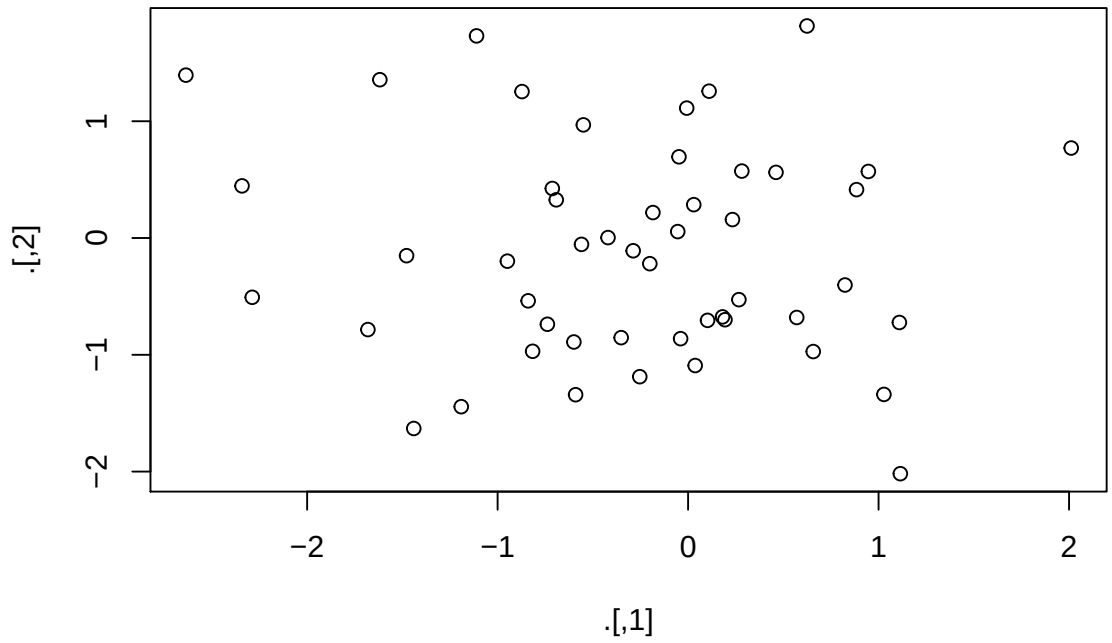
```
##
## Attaching package: 'magrittr'

## The following object is masked from 'package:purrr':
##
##     set_names

## The following object is masked from 'package:tidyr':
##
##     extract
```

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
```




```
##  NULL
```

```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
```




```
##  num [1:50, 1:2] 1.114 0.57 -1.44 -0.351 1.109 ...
```

- Explode to refer to variables without having to attach the dataset using %$%.

```
cor(mtcars$disp, mtcars$mpg)
```

```
## [1] -0.8475514
```

```
mtcars %$%
  cor(disp, mpg)
```

```
## [1] -0.8475514
```

```
attach(mtcars)
```

```
## The following object is masked from package:ggplot2:
##
##     mpg
```

```
cor(disp, mpg)
```

```
## [1] -0.8475514
```

- Assignment with %<>% replaces the object on the left.

- **Exercise**: Try taking the nycflights13 flights dataset (from the library nycflights13) and replacing it with only those flights from JFK that were over 2000 miles long, keeping only the origin, destination, distance, flight time and carrier.
- DCG Opinion: I've never seen anyone use this before. I would avoid it since it is obfuscating the assignment and makes the code harder to read.

Functions

- When to write a function and why to write a function
- Steps to creating a function:
  1. figure out the logic in a simple case
  2. name it something meaningful - usually a verb
  3. list the inputs inside function(x,y,z)
  4. place code for function in a {} block
  5. test your function with some different inputs
  6. add error-checking of inputs
- Example from our book follows.

```
df <- tibble::tibble(
  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$b, na.rm = TRUE) - min(df$a, 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))
```

- How many inputs does each line have?

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

```
##  [1] 0.53091514 0.15787327 0.46528458 0.30430354 0.00000000 0.03813980
##  [7] 0.08216206 0.42584589 1.00000000 0.03141179
```

```
# get rid of duplication
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])
```

```
##  [1] 0.53091514 0.15787327 0.46528458 0.30430354 0.00000000 0.03813980
##  [7] 0.08216206 0.42584589 1.00000000 0.03141179
```

```r
# make it into a function and test it
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10))
```

```
## [1] 0.0 0.5 1.0
```

```r
rescale01(c(-10, 0, 10))
```

```
## [1] 0.0 0.5 1.0
```

```r
rescale01(c(1, 2, 3, NA, 5))
```

```
## [1] 0.00 0.25 0.50   NA 1.00
```

```r
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

- Now, if we have a change in requirements, we only have to change it in one place. For instance, perhaps we want to handle columns that have Inf as one of the values.

```r
x <- c(1:10, Inf)
rescale01(x)
```

```
##  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

```r
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)
```

```
##  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
##  [8] 0.7777778 0.8888889 1.0000000       Inf
```

- Do's and do not's of function naming:
  - pick either snake_case or camelCase but don't use both
  - meaningful names (preferably verbs)
  - for a family of functions, start with the same word
  - try not to overwrite common functions or variables
  - use lots of comments in your code, particularly to explain the "why" of your code or to break up your code into sections using something like `# load data ----------`

Function Documentation

- This is a DCG opinion section.
- It is good practice to always precede a function with documentation on

  1. What does it do?
  2. What are the inputs?
  3. What are the outputs?

- Use comments to do this.
- Here is an example of me re-writing the `sum()` function.

```
# Sums up the elements of a numeric vector.
#
# x: a vector of numerics to be summed.
#
# returns: The sum of x
sum2 <- function(x) {
  stopifnot(is.numeric(x))

  sum_val <- 0
  for (index in seq_along(x)) {
    sum_val <- sum_val + x[index]
  }

  return(sum_val)
}

sum2(c(1, 5, 21))
```

```
## [1] 27
```

- **Exercise**: Re-write the the function `range()`, including documentation at the top of the function.
- **Exercise**: Write `both_na()`, a function that takes two vectors of the same length and returns the number of positions that have an `NA` in both vectors. Include documentation.
- **Exercise**: Read the source code for each of the following three functions, puzzle out what they do, and then brainstorm better names. Then add documentation to each function.

```
f1 <- function(string, prefix) {
  substr(string, 1, nchar(prefix)) == prefix
}

f2 <- function(x) {
  if (length(x) <= 1) return(NULL)
  x[-length(x)]
}

f3 <- function(x, y) {
  rep(y, length.out = length(x))
}
```

Conditional Execution

- Conditional if-then statements are useful to evaluate code only if certain conditions are met. The syntax is:

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

```
# note that a function returns the last value computed
has_name <- function(x) {
  nms <- names(x)
  if (is.null(nms)) {
    rep(FALSE, length(x))
  } else {
    !is.na(nms) & nms != ""
  }
}
x1 <- c(1,2,3)
x2 <- c(a = 1, 2,3)
has_name(x1)
```

```
## [1] FALSE FALSE FALSE
```

```
has_name(x2)
```

```
## [1]  TRUE FALSE FALSE
```

- The condition for an if statement must be a single logical element, not a vector. For this reason, if combining them, use `||` which returns `TRUE` at first occurrence of `TRUE` or `&&` which return `FALSE` at first occurrence of `FALSE`. Can use `any()` or `all()` to collapse a logical vector. Similarly, for equality, use `identical()` or `dplyr::near()`. You can use `==`, but it is vectorized and might give you errors if you use it with a vector.

Multiple conditions

- You may chain multiple if statements or use switch or cut

```
do_op <- function(x, y, op) {
  switch(op,
     plus = x + y,
     minus = x - y,
     times = x * y,
     divide = x / y,
     stop("Unknown op!")
  )
}
do_op(2,4,"plus")
```

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

```
do_op(2,4,"divide")
```

```
## [1] 0.5
```

```
do_op(2,4,1)
```

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

```
do_op(2,4,"mod")
```

```
## Error in do_op(2, 4, "mod"): Unknown op!
```

```
# note that we had to change the chunk option to error=TRUE in
# order to test our error output
```

- Some code style standards:
  - Put function code and if statement code inside `{}`
  - opening `{` never on own line and always followed by a new line
  - closing `}` on own line unless followed by else
  - use indentation
- **Exercise**: Write a greeting function that says `"good morning"`, `"good afternoon"`, or `"good evening"`, depending on the time of day. (Hint: use a time argument that defaults to `lubridate::now()`. That will make it easier to test your function.)
- **Exercise**: Implement a `fizzbuzz()` function. It takes a single number as input. If the number is divisible by three, it returns `"fizz"`. If it's divisible by five it returns `"buzz"`. If it's divisible by three and five, it returns `"fizzbuzz"`. Otherwise, it returns the number. Make sure you first write working code before you create the function.

Function inputs

- First inputs listed should be the data to compute on. Later arguments should be parameters that control the details of the computation and should generally have defaults specified. The default should be the most often used value unless there is a safety reason to do differently. Look at some functions and see what the defaults are. Try `log()`, `mean()`, `t.test()`, `cor()`.
- When calling a function, omit the names of the data arguments, but specify the names of the parameters. White space: use space after `,` and around `=` or other operators.
- Use descriptive names for your arguments with the following exceptions:
  - `x`, `y`, `z`: vectors.
  - `w`: a vector of weights.
  - `df`: a data frame.
  - `i`, `j`: numeric indices (typically rows and columns).
  - `n`: length, or number of rows.
  - `p`: number of columns.

Idiot proof your functions

- Check validity of inputs and use `stop()` to output an error message

```r
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }
  sum(w * x) / sum(w)
}
wt_mean(1:4,1:2)
```

```
## Error: `x` and `w` must be the same length
```

- Another option is to use `stopifnot()` which checks that each argument is true and issues a generic error message if there is a problem

```r
t_mean <- function(x, w, na.rm = FALSE) {
  stopifnot(is.logical(na.rm), length(na.rm) == 1)
  stopifnot(length(x) == length(w))

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(w)
}
wt_mean(1:6, 6:1, na.rm = "foo")
```

```
## Error in wt_mean(1:6, 6:1, na.rm = "foo"): unused argument (na.rm = "foo")
```

Taking an arbitrary number of inputs

```r
commas <- function(...) {
  stringr::str_c(..., collapse = ", ")
}
commas(letters[1:10])
```

```
## [1] "a, b, c, d, e, f, g, h, i, j"
```

```r
rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
rule("Important output")
```

```
## Important output ----------------------------------------------------------
```

Return values (function output)

Explicit return values

- R returns the last value calculated unless you specify otherwise. You can use a `return()` command to break out of the program early and return a value. Perhaps you want to check for a trivial condition and return early if that is the case. Or you have an if statement with one simple and one complex case.
- DCG Opinion: Hadley says to only use `return()` for early command breaks. But I always use `return()` to indicate what the function is returning. I prefer to be more explicit.

Writing Pipeable Functions

- Two types of functions:
  - transformation functions - primary object is the first argument and a modified version is returned by the function
  - side-effect functions - called to perform an action; should invisibly return the first argument so they are not printed by default

```
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")

  invisible(df)
}
# calling interactively
show_missings(mtcars)
```

```
## Missing values: 0
```

```
#  assigning output
x <- show_missings(mtcars)
```

```
## Missing values: 0
```

```
class(x)
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

```
dim(x)
```

```
## [1] 32 12
```

```
#  allows for use in a pipe
mtcars %>%
  show_missings() %>%
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
  show_missings()
```

```
## Missing values: 0
## Missing values: 18
```

- What happens if we leave out the `invisible(df)` line?

```r
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")
}
# calling interactively
show_missings(mtcars)
```

```
## Missing values: 0
```

```r
#  assigning output
x <- show_missings(mtcars)
```

```
## Missing values: 0
```

```r
class(x)
```

```
## [1] "NULL"
```

```r
dim(x)
```

```
## NULL
```

```r
# pipe no longer works
mtcars %>%
  show_missings() %>%
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
  show_missings()
```

```
## Missing values: 0
```

```
## Error in UseMethod("mutate_"): no applicable method for 'mutate_' applied to an object of class
```

Environment

- R will look for functions and variables in the current environment before it looks elsewhere. This allows you to use variables in the environment that are not explicitly in your function or to overwrite things that may be general functions in R with local versions. However, in general, this is not a good idea.