Pipes, Functions, Conditionals

David Gerard and Jane Wall

2019-08-30

Learning Objectives

- Piping.
- Creating Functions.
- Chapters 17 through 19 in RDS.
- Magrittr Overview.

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 geometric mean of a vector of positive numbers. Then the numerically stable steps are
  1. Take the log of all of the elements of the vector.
  2. Average these logged elements.
  3. Exponentiate this average.
- We'll create some random data using `rnorm()` to look at these steps:

```
x <- abs(rnorm(100))
head(x)
```

```
## [1] 1.1094 1.1638 2.0769 0.8728 1.1639 1.8938
```

- Create intermediate objects

```
log_x <- log(x)
mean_lx <- mean(log_x)
geo_mean <- exp(mean_lx)
geo_mean
```

```
## [1] 0.5261
```

- Nest functions

```
geo_mean <- exp(mean(log(x)))
geo_mean
```

```
## [1] 0.5261
```

- Use pipe

```
library(tidyverse)
x %>%
  log() %>%
  mean() %>%
  exp() ->
  geo_mean
geo_mean
```

```
## [1] 0.5261
```

- The term on the left of the pipe becomes the *first* argument of the function on the right of the pipe. Thus, the following two expressions are the exact same:

```
4 %>%
  log(base = 2)
```

```
## [1] 2
```

```
log(4, base = 2)
```

```
## [1] 2
```

- Note that the output is only placed in the *first* argument of the next function. We can use the pipe while specifying other arguments..

```
x %>%
  log() %>%
  mean(na.rm = TRUE) %>%
  exp() ->
  geo_mean
geo_mean
```

```
## [1] 0.5261
```

- **Exercise**: Using the pipe, create a vector of 100 random draws from the normal distribution with standard deviation 10, sort this vector in increasing order, calculate the lagged differences between these elements, average these lagged differences, then round this average to the nearest tenth decimal place.

  Useful functions: `diff()`, `rnorm()`, `sort()`, `round()`, `mean()`.

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

- Coding standards

1. Save as text file
2. Indent code
3. Limit width of code (80 columns?)
4. Limit the length of individual functions
5. Frequent comments

```
add_two <- function(a, b) {
  return(a + b)
}
add_two(2, 4)
```

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

- 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] 0.38495 0.60719 0.76868 1.00000 0.40916 0.43060 0.00000 0.61478
##  [9] 0.74254 0.07127
```

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

```
##  [1] 0.38495 0.60719 0.76868 1.00000 0.40916 0.43060 0.00000 0.61478
##  [9] 0.74254 0.07127
```

```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] 0.0000 0.1111 0.2222 0.3333 0.4444 0.5556 0.6667 0.7778 0.8889 1.0000
## [11]    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 `diff()` function.

```
# Calculates a vector of lags from a numeric vector
#
# x: a vector of numerics to be lagged
#
# returns: The lags of x
diff2 <- function(x) {
  stopifnot(is.numeric(x))
  lagvec <- x[2:length(x)] - x[1:(length(x) - 1)]
  return(lagvec)
}

diff2(c(1, 5, 21))
```

```
## [1]  4 16
```

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

```
## [1]  4 16
```

- **Exercise**: Re-write the the function `range()`, including documentation at the top of the function. Use functions: `min()`, `max()`
- **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. Useful functions: `is.na()`, `sum()`, logical operators.
- **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
}
```

```
# input: x a numeric
# output: TRUE if x > 0, FALSE if x <= 0
is_positive <- function(x) {
  if (x > 0) {
    pos_logic <- TRUE
  } else {
    pos_logic <- FALSE
  }
  return(pos_logic)
}

is_positive(10)
```

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

```
is_positive(-1)
```

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

```
is_positive(0)
```

```
## [1] 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

```
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**: 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
wt_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"): is.logical(na.rm) is not TRUE
```

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.