Piping and Functions

David Gerard

January 23, 2019

Calculating a Geometric Mean

Create some random data

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

```
## [1] 0.8294 0.9943 1.4024 0.4421 0.1621 0.4641
```

Create a new object after each function

```
log_x <- log(x)
mean_logx <- mean(log_x)
exp(mean_logx)
```

```
## [1] 0.412
```

Nest functions

```
exp(mean(log(x)))
```

```
## [1] 0.412
```

Overight original object

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

Use the pipe %>%

```
library(tidyverse)
```

```
## -- Attaching packages -- tidyverse 1.2.1 --
```

```
## v ggplot2 3.1.0     v purrr   0.2.5
## v tibble  2.0.1     v dplyr   0.7.8
## v tidyr   0.8.2     v stringr 1.3.1
## v readr   1.3.1     v forcats 0.3.0
```

```
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
x %>%
  log() %>%
  mean() %>%
  exp()
```

```
## [1] 0.412
```

Piping always inserts the output into the first argument of the next function

```r
x[1] <- NA
x %>%
  log() %>%
  mean(na.rm = TRUE) %>%
  exp()
```

```
## [1] 0.4091
```

exercise solutions

```r
rnorm(n = 100, sd = 10) %>%
  sort(decreasing = FALSE) %>%
  diff() %>%
  mean() %>%
  round(digits = 1)
```

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

```r
rnorm(100) %>%
  sort()
```

```
##   [1] -2.90090 -2.11750 -2.00286 -1.81862 -1.81090 -1.56869 -1.44951
##   [8] -1.39331 -1.35129 -1.24939 -1.20011 -1.19408 -1.18180 -1.17203
##  [15] -1.15507 -1.11632 -1.06064 -1.02052 -1.01217 -0.99334 -0.97741
##  [22] -0.96525 -0.90063 -0.88789 -0.83108 -0.80094 -0.78991 -0.78040
##  [29] -0.77806 -0.77294 -0.68202 -0.67551 -0.59222 -0.58551 -0.54804
##  [36] -0.53896 -0.51476 -0.49317 -0.48020 -0.45286 -0.43329 -0.36720
##  [43] -0.34558 -0.30414 -0.26969 -0.20931 -0.19930 -0.13298 -0.06567
##  [50] -0.06452 -0.04790 -0.02996  0.01750  0.02230  0.03120  0.05305
##  [57]  0.10303  0.11075  0.12665  0.14210  0.16489  0.17145  0.17640
##  [64]  0.19835  0.24050  0.24209  0.28978  0.31234  0.32742  0.34449
##  [71]  0.52859  0.60359  0.61438  0.64336  0.66301  0.69345  0.70703
##  [78]  0.74657  0.74701  0.76130  0.79188  0.81992  0.82273  0.88927
##  [85]  1.02199  1.03326  1.06103  1.16587  1.18060  1.19263  1.24619
##  [92]  1.28894  1.38886  1.41441  1.55149  1.59374  1.71939  1.91795
##  [99]  1.95062  2.32579
```

```r
sort(rnorm(100))
```

```
##   [1] -2.34726 -2.01772 -1.92093 -1.85237 -1.61404 -1.56923 -1.50231
##   [8] -1.36368 -1.32001 -1.28425 -1.23541 -1.21947 -1.21817 -1.19260
##  [15] -1.18037 -1.16869 -1.16820 -1.11069 -1.10871 -1.10402 -1.09757
##  [22] -1.00617 -0.96740 -0.96708 -0.96093 -0.93438 -0.91852 -0.83635
##  [29] -0.81458 -0.81424 -0.78171 -0.73065 -0.72157 -0.63521 -0.62064
##  [36] -0.62008 -0.56510 -0.56279 -0.50604 -0.46071 -0.45753 -0.45605
##  [43] -0.45241 -0.44940 -0.42420 -0.36918 -0.35914 -0.32952 -0.24646
##  [50] -0.23534 -0.19531 -0.14924 -0.11776 -0.11415 -0.05792 -0.01594
##  [57]  0.02506  0.03467  0.03710  0.04187  0.11456  0.11533  0.14171
##  [64]  0.14257  0.14314  0.14798  0.15740  0.26719  0.27243  0.27780
##  [71]  0.31680  0.40650  0.41574  0.41682  0.45567  0.49734  0.50464
##  [78]  0.51341  0.55388  0.62305  0.73706  0.82900  0.88840  0.96975
##  [85]  1.00934  1.04530  1.05521  1.06649  1.15337  1.16481  1.18453
##  [92]  1.19699  1.48007  1.50020  1.68530  1.73584  1.81377  2.05834
##  [99]  2.13861  2.59577
```

Functions and Function Creation

A very basic function

```
add_two <- function(a, b) {
  ## tons of code goes here
  c <- a + b
  return(c)
}
add_two(a = 2, b = 5)
```

```
## [1] 7
```

The book example

```
df <- data.frame(
  a = rnorm(100),
  b = rnorm(100),
  c = rnorm(100),
  d = rnorm(100)
)
head(df)
```

```
##         a       b        c        d
## 1  1.8291 -1.1056 -1.74077  1.02329
## 2 -0.7598  0.4003  1.14011  0.04564
## 3 -1.0405  1.5169 -0.01836  0.23579
## 4 -0.9783 -0.9541  1.99346  0.18172
## 5 -0.1687  0.3324  0.66745  0.60737
## 6  0.3156  2.7165 -0.79239 -0.94581
```

Suppose the goal is to transform these variables so that all elements are between 0 and 1

Copy code

- Obnoxious
- Subject to error
- If you have to change later, it is obnoxious and subject to error

```
df$a <- ((df$a - min(df$a))) / (max(df$a) - min(df$a))
df$b <- ((df$b - min(df$b))) / (max(df$a) - min(df$b))
df$c <- ((df$c - min(df$c))) / (max(df$c) - min(df$c))
df$d <- ((df$d - min(df$d))) / (max(df$d) - min(df$d))
```

Create a function!

```
x <- df$a

rescale1 <- function(x) {

  ## Calculate minimum -------------------
  min_x <- min(x, na.rm = TRUE)

  ## Rescale -----------------------------
  rescale_vec <- (x - min_x) /
    (max(x, na.rm = TRUE) - min_x)

  return(rescale_vec)
```

```
}
rescale1(c(1, 2, 3))
```

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

```
df$a <- rescale1(df$a)
df$b <- rescale1(df$b)
df$c <- rescale1(df$c)
df$d <- rescale1(df$d)
```

```
rescale1(c(NA, 1, 3, 4))
```

```
## [1]     NA 0.0000 0.6667 1.0000
```

```
x <- c(NA, NA, 2, 3, NA)
y <- c(NA, NA, 5, 6, 7)
both_na <- function(x, y) {
  na_x <- is.na(x)
  na_y <- is.na(y)
  index_vector <- 1:length(x)
  sum(na_x & na_y)
}


sum(c(TRUE, TRUE, TRUE))
```

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

If-then statements

```
x <- 6
if (x > 5) {
  ## code
  "hurray!"
} else {
  ## other code
  "oh well"
}
```

```
## [1] "hurray!"
```

The has-name function

Check if a vector has any names attached if not then it will return FALSE for all elements

If yes, then it will return TRUE in the positions where the vector has a name

```
x <- c(a = 1, 2, 3)
names(x)
```

```
## [1] "a" ""  ""
```

```
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
```

Multiple conditional execution

```
x <- -5
if (x > 5) {
  "Hurray!"
} else if (x > 0) {
  "oh well"
} else if (x > -10) {
  "oh boy"
} else {
  "oh no"
}
```

```
## [1] "oh boy"
```