functions_solution

Thiyanga Talagala

03/03/2020

```
library(lubridate)
```

```
##
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:base':
##
##     date
```

```
library(tidyverse)
```

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

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.4
## v tidyr   0.8.3     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ------------------------ tidyverse_conflicts() --
## x lubridate::as.difftime() masks base::as.difftime()
## x lubridate::date()        masks base::date()
## x dplyr::filter()          masks stats::filter()
## x lubridate::intersect()   masks base::intersect()
## x dplyr::lag()             masks stats::lag()
## x lubridate::setdiff()     masks base::setdiff()
## x lubridate::union()       masks base::union()
```

19.2.1 Practice Why is TRUE not a parameter to rescale01()? What would happen if x contained a single missing value, and na.rm was FALSE?

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = FALSE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(-1, 0, 5, 20, NA))
```

```
## [1] NA NA NA NA NA
```

Everything is NA! We should set an argument to control the `TRUE` or `FALSE` of `range`

In the second variant of rescale01(), infinite values are left unchanged. Rewrite rescale01() so that -Inf is mapped to 0, and Inf is mapped to 1.

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

```
  x[x == -Inf] <- 0
  x
}
rescale01(c(Inf, -Inf, 0:5, NA))
```

```
## [1]  1  0  0  1  2  3  4  5 NA
```

Practice turning the following code snippets into functions. Think about what each function does. What would you call it? How many arguments does it need? Can you rewrite it to be more expressive or less duplicative?

```
x <- 1:10
mean(is.na(x))
```

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

```
x / sum(x, na.rm = TRUE)
```

```
##  [1] 0.01818182 0.03636364 0.05454545 0.07272727 0.09090909 0.10909091
##  [7] 0.12727273 0.14545455 0.16363636 0.18181818
```

```
sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
```

```
## [1] 0.5504819
```

Implementation

```
prop_miss <- function(x) {
  mean(is.na(x))
}
my_mean <- function(x) {
  x / sum(x, na.rm = TRUE)
}
my_var <- function(x) {
  sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
}

# Follow http://nicercode.github.io/intro/writing-functions.html to write your own functions to compute
```

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.

```
both_na <- function(x, y) {
  stopifnot(length(x) == length(y))
  which(is.na(x) & is.na(y))
}
both_na(c(1, 2, NA, 2, NA), c(1, 2, 3, 4, NA))
```

```
## [1] 5
```

What do the following functions do? Why are they useful even though they are so short?

```
# Checks whether the path is a directory.
is_directory <- function(x) file.info(x)$isdir
# Checks whether a file is readable.
is_readable <- function(x) file.access(x, 4) == 0
```

Read the complete lyrics to "Little Bunny Foo Foo". There's a lot of duplication in this song. Extend the initial piping example to recreate the complete song, and use functions to reduce the duplication.

Exercises 19.3.1

Read the source code for each of the following three functions, puzzle out what they do, and then brainstorm better names.

```
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))
}
```

```
# It tests whether the prefix argument is the same prefix
# as in the string
has_prefix <- function(string, prefix) {
  substr(string, 1, nchar(prefix)) == prefix
}
has_prefix(c("hey_ho", "another_pre"), "hey")
```

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

```
# It removes the last element of x, only of the length greater than 1
# otherwise returns NULL
remove_last <- function(x) {
  if (length(x) <= 1) return(NULL)
  x[-length(x)]
}
remove_last(c(1, 2, 3))
```

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

```
convert_length <- function(x, y) {
  rep(y, length.out = length(x))
}
convert_length(1:10, 1:3)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1
```

Take a function that you've written recently and spend 5 minutes brainstorming a better name for it and its arguments.

-

Compare and contrast rnorm() and MASS::mvrnorm(). How could you make them more consistent?

You could create one function that accepts the same first three arguments and then a logical stating whether you want it to be uni or multivariate.

Make a case for why norm_r(), norm_d() etc would be better than rnorm(), dnorm(). Make a case for the opposite.

Taken from: https://jrnold.github.io/r4ds-exercise-solutions/functions.html#when-should-you-write-a-function

If named norm_r and norm_d, it groups the family of functions related to the normal distribution. If named rnorm, and dnorm, functions related to are grouped into families by the action they perform. r* functions

always sample from distributions: rnorm, rbinom, runif, rexp. d* functions calculate the probability density or mass of a distribution: dnorm, dbinom, dunif, dexp.

Exercises 19.4.4

What's the difference between if and ifelse()? Carefully read the help and construct three examples that illustrate the key differences.

`ifelse` tests the conditions in a vectorizes way, meaning it returns a vector that can be of length 1 or > 1. `if` tests only one logical statement for an object of length 1.

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.)

```
greeter <- function(now = now()) {
  if (between(hour(now), 8, 13)) {
    print("Good morning")
  } else if (between(hour(now), 13, 18)) {
    print("Good afternoon")
  } else {
    print("Good evening")
  }
}
greeter(now())
```

```
## [1] "Good evening"
```

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.

```
x <- 9
fizzbuzz <- function(x) {
  by_three <- x %% 3 == 0
  by_five <- x %% 5 == 0

  if (by_three && by_five) {
    return("fizzbuzz")
  } else if (by_three) {
    return("fizz")
  } else if (by_five) {
    return("buzz")
  } else {
    return(x)
  }
}
sapply(1:20, fizzbuzz)
```

```
##  [1] "1"        "2"        "fizz"     "4"        "buzz"     "fizz"
##  [7] "7"        "8"        "fizz"     "buzz"     "11"       "fizz"
## [13] "13"       "14"       "fizzbuzz" "16"       "17"       "fizz"
## [19] "19"       "buzz"
```

How could you use cut() to simplify this set of nested if-else statements?

```
temp <- c(27, 30)
if (temp <= 0) {
```

```
  "freezing"
} else if (temp <= 10) {
  "cold"
} else if (temp <= 20) {
  "cool"
} else if (temp <= 30) {
  "warm"
} else {
  "hot"
}
```

```
## Warning in if (temp <= 0) {: the condition has length > 1 and only the first
## element will be used
```

```
## Warning in if (temp <= 10) {: the condition has length > 1 and only the first
## element will be used
```

```
## Warning in if (temp <= 20) {: the condition has length > 1 and only the first
## element will be used
```

```
## Warning in if (temp <= 30) {: the condition has length > 1 and only the first
## element will be used
```

```
## [1] "warm"
```

```
cut(temp, breaks = seq(-10, 40, 10),
    labels = c("freezing", "cold", "cool", "warm", "hot"))
```

```
## [1] warm warm
## Levels: freezing cold cool warm hot
```

How would you change the call to cut() if I'd used < instead of <=? What is the other chief advantage of cut() for this problem? (Hint: what happens if you have many values in temp?)

First, that it's vectorized, so I can recode a vector of values with the single function and second that it controls the closing points of what to recode such as:

```
cut(temp, breaks = seq(-10, 40, 10),
    right = FALSE,
    labels = c("freezing", "cold", "cool", "warm", "hot"))
```

```
## [1] warm hot
## Levels: freezing cold cool warm hot
```

What happens if you use switch() with numeric values?

```
x = 2
switch(x, 1 = "No", 2 = "Yes")
switch(x, `1` = "No", `2` = "Yes")
```

What does this switch() call do? What happens if x is "e"?

```
x <- "d"
switch(x,
  a = ,
  b = "ab",
  c = ,
  d = "cd"
)
```

```
## [1] "cd"
```

When a slot is empty it finds the next slot that has values.

```
x <- "a"
switch(x,
  a = ,
  z = ,
  b = "ab",
  c = ,
  d = "cd"
)
```

```
## [1] "ab"
```

Exercises 19.5.5

What does commas(letters, collapse = "-") do? Why?

```
commas <- function(..., collapse = ",") {
  str_c(..., collapse = collapse)
}
commas(letters, collapse = "-")
```

```
## [1] "a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"
```

You get them - separated!

It'd be nice if you could supply multiple characters to the pad argument, e.g. rule("Title", pad = "-+"). Why doesn't this currently work? How could you fix it?

```
rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  pad_char <- nchar(pad)
  cat(title, " ", stringr::str_dup(pad, width / pad_char), "\n", sep = "")
}
rule("my title", pad = "-+")
```

```
## my title -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

What does the trim argument to mean() do? When might you use it?

```
mean(c(99, 1:10))
```

```
## [1] 14
```

```
mean(c(99, 1:10), trim = 0.1)
```

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

In this case, it trimes 10% of the the vector on both sides.

The default value for the method argument to cor() is c("pearson", "kendall", "spearman"). What does that mean? What value is used by default?

The first value is used by default and you have the option of specifying any of the three values.