Lab9

Irlanda Ayon-Moreno

5/2/2018

Learning Objectives

- Introduction to the R package "testthat"
- Write simple functions and their unit tests
- Test your code
- String manipulation
- Base R functions for strings

The purpose of this lab is two-fold: 1) getting started with writing tests for your functions, and 2) getting started with functions to do basic manipulation of strings.

R package `"testthat"`

`"testthat"` is one of the packages in R that helps you write tests for your functions. First, remember to install the package:

```r
# do not include this code in your Rmd
# execute this in your console
install.packages("testthat")
```

- `"testthat"` provides a testing framework for R that is easy to learn and use
- `"testthat"` has a hierarchical structure made up of:
  - expectations
  - tests
  - contexts
- A **context** involves **tests** formed by groups of **expectations**
- Each structure has associated functions:
  - `expect_that()` for expectations
  - `test_that()` for groups of tests
  - `context()` for contexts

```r
# load testthat
library(testthat)
```

List of common expectation functions

Function	Description
`expect_true(x)`	expects that `x` is `TRUE`
`expect_false(x)`	expects that `x` is `FALSE`

Function	Description
`expect_null(x)`	expects that `x` is `NULL`
`expect_type(x, y)`	expects that `x` is of type `y`
`expect_is(x, y)`	expects that `x` is of class `y`
`expect_length(x, y)`	expects that `x` is of length `y`
`expect_equal(x, y)`	expects that `x` is equal to `y`
`expect_equivalent(x, y)`	expects that `x` is equivalent to `y`
`expect_identical(x, y)`	expects that `x` is identical to `y`
`expect_lt(x, y)`	expects that `x` is less than `y`
`expect_gt(x, y)`	expects that `x` is greater than `y`
`expect_lte(x, y)`	expects that `x` is less than or equal to `y`
`expect_gte(x, y)`	expects that `x` is greater than or equal `y`
`expect_named(x)`	expects that `x` has names `y`
`expect_matches(x, y)`	expects that `x` matches `y` (regex)
`expect_message(x, y)`	expects that `x` gives message `y`
`expect_warning(x, y)`	expects that `x` gives warning `y`
`expect_error(x, y)`	expects that `x` throws error `y`

Practice writing simple tests

- To start the practice, create a new directory, e.g. `lab09`
- `cd` to `lab09`
- Create an `R` script to write and document your functions: e.g. `functions.R`
- Create a separate `R` script `tests.R` to write the tests of your functions.

Example with `stat_range()`

Let's start with a basic function `stat_range()` to compute the overall range of a numeric vector. Create this function in the file `functions.R`.

```r
#' @title Range
#' @description Computes overall range: max - min
#' @param x numeric vector
#' @return computed range
stat_range <- function(x) {
  max(x) - min(x)
}
```

- *description:* computes the range of a numeric vector (i.e. max - min)
- *input:* a numeric vector
- *output:* the range value (max - min)

Tests for `stat_range()`

To write the unit tests in `tests.R`, we are going to consider the following testing vectors:

- `x <- c(1, 2, 3, 4, 5)`
- `y <- c(1, 2, 3, 4, NA)`
- `z <- c(TRUE, FALSE, TRUE)`
- `w <- letters[1:5]`

The typical structure of the tests has the following form:

```r
# load the source code of the functions to be tested
source("functions.R")

# context with one test that groups expectations
context("Test for range value")

test_that("range works as expected", {
  x <- c(1, 2, 3, 4, 5)

  expect_equal(stat_range(x), 4)
  expect_length(stat_range(x), 1)
  expect_type(stat_range(x), 'double')
})
```

- use `context()` to describe what the test are about
- use `test_that()` to group expectations:
  - output equal to 4
  - output of length one
  - output of type `double`
- to run the tests from the R console, use the function `test_file()` by passing the path of the file `tests.R`

```r
# assuming that your working directory is "lab09/"
library(testthat)
test_file("tests.R")
```

```
## √ | OK F W S | Context
##
/ |  0       | Test for range value
- |  1       | Test for range value
\ |  2       | Test for range value
| |  3       | Test for range value
/ |  4       | Test for range value
- |  5       | Test for range value
\ |  6       | Test for range value
| |  7       | Test for range value
/ |  8       | Test for range value
- |  9       | Test for range value
√ |  9       | Test for range value [0.1 s]
```

```
##
/ |  0       | Test for center values
- |  1       | Test for center values
\ |  2       | Test for center values
| |  3       | Test for center values
√ |  3       | Test for center values
##
/ |  0       | Test for spread values
- |  1       | Test for spread values
\ |  2       | Test for spread values
√ |  2       | Test for spread values
##
## == Results ===========================================================
## Duration: 0.2 s
##
## OK:       14
## Failed:   0
## Warnings: 0
## Skipped:  0
```

You could actually include a code chunk in your Rmd with the code above.

Your Turn

Write more groups of tests—`test_that()`—to test `stat_range()` with the rest of the testing vectors `y`, `z`, `w`:

- Using `y`, write expectations for:
  - output is of length one
  - output is equal to `NA_real_`

```r
test_that("range value for numeric vectors with NAs", {
  y <- c(1, 2, 3, 4, NA)

  expect_length(stat_range(y), 1)
  expect_equal(stat_range(y), NA_real_)
})
```

- Using `z`, write expectations for:
  - output is of length one
  - output is of type `integer`
  - output is equal to `1L`

```r
test_that("range value for logical vectors", {
  z <- c(TRUE, FALSE, TRUE)

  expect_length(stat_range(z), 1)
  expect_type(stat_range(z), "integer")
  expect_equal(stat_range(z), 1L)
```

```
})
```

- Using `w`, write expectations for:
  - throws an error

```
test_that("range value for non-numeric vectors", {
  w <- letters[1:5]

  expect_error(stat_range(z))
})
```

Try writing the following functions and come up with unit tests:

- `stat_centers()`
  - *description:* computes measures of center such as Median and Mean
  - *input:* a numeric vector
  - *output:* a numeric vector with median and mean
  - use `mean()` and `median()` to write `stat_centers()`

```
#' @title Centers measures
#' @description Computes measures of center such as Median and Mean
#' @param x numeric vector
#' @return a numeric vector with median and mean
stat_centers <- function(x) {
  y <- c(0, 0)
  y[1] <- median(x)
  y[2] <- mean(x)
  names(y) <- c("Median", "Mean")
  return(y)
}
```

```
context("Test for center values")

test_that("centers works as expected", {
  x <- c(1, 2, 3, 4, 5)

  expect_equal(stat_centers(x), c("Median" = 3, "Mean" = 3))
  expect_length(stat_centers(x), 2)
  expect_type(stat_centers(x), 'double')
})
```

- `stat_spreads()`
  - *description:* computes measures of spread such as Range, IQR, Standard Deviation
  - *input:* a numeric vector
  - *output:* a numeric vector with range, iqr, and stdev
  - use `stat_range()`, `IQR()`, and `sd()` to write `stat_spreads()`

```
#' @title Spread measures
#' @description Computes measures of spread: range, IQR, and SD
#' @param x numeric vector
#' @return vector with range, iqr, and stdev
```

```r
stat_spreads <- function(x) {
  y <- c('range' = stat_range(x),
         'iqr' = IQR(x),
         'stdev' = sd(x))
  return(y)
}
```

```r
stat_spreads(x)
```

```
## Error in stat_range(x): object 'x' not found
```

```r
context("Test for spread values")

test_that("centers works as expected", {
  x <- c(1, 2, 3, 4, 5)

  expect_length(stat_spreads(x), 3)
  expect_type(stat_spreads(x), 'double')
})
```

Basics of String Manipulation

In this second part of the lab, you will be using the row names of the data frame `USArrests` (this data comes already in R):

```r
head(USArrests)
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

```r
states <- rownames(USArrests)
head(states)
```

```
## [1] "Alabama"    "Alaska"     "Arizona"    "Arkansas"   "California"
## [6] "Colorado"
```

Here are some functions that you may need to use in this lab:

- `nchar()`
- `tolower()`
- `toupper()`
- `casefold()`
- `paste()`

- `paste0()`
- `substr()`

Number of characters

`nchar()` allows you to count the number of characters in a string. Use it on `states` to get the number of characters of each state:

```
# number of characters
nchar(states)
```

```
##  [1]  7  6  7  8 10  8 11  8  7  7  6  5  8  7  4  6  8  9  5  8 13  8  9
## [24] 11  8  7  8  6 13 10 10  8 14 12  4  8  6 12 12 14 12  9  5  4  7  8
## [47] 10 13  9  7
```

Case folding

There are 3 functions to do case-folding: `tolower()`, `toupper()`, and `casefold()`. Apply each function on `states` to see what happens:

```
# to lower case
head(
  tolower(states)
)
```

```
## [1] "alabama"    "alaska"     "arizona"    "arkansas"   "california"
## [6] "colorado"
```

```
# to upper case
head(
  toupper(states)
)
```

```
## [1] "ALABAMA"    "ALASKA"     "ARIZONA"    "ARKANSAS"   "CALIFORNIA"
## [6] "COLORADO"
```

```
# case folding (upper = TRUE)
head(
  casefold(states, upper = T)
)
```

```
## [1] "ALABAMA"    "ALASKA"     "ARIZONA"    "ARKANSAS"   "CALIFORNIA"
## [6] "COLORADO"
```

```
# case folding (upper = FALSE)
head(
  casefold(states, upper = F)
)
```

```
## [1] "alabama"    "alaska"     "arizona"    "arkansas"   "california"
## [6] "colorado"
```

Length of State Names

We just saw how to use `nchar()` to count the number of characters in each state name:

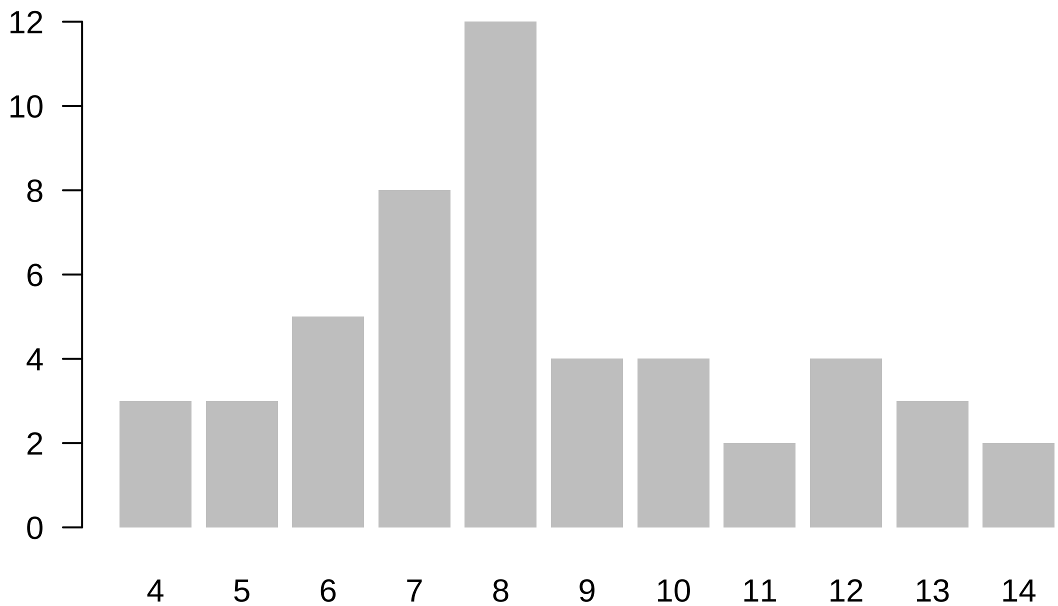
```
# number of charcaters
num_chars <- nchar(states)
```

Use the vector `num_chars` to obtain a frequency table called `char_freqs`, and then plot the frequencies with a bar chart:

```
# frequency table
char_freqs <- table(num_chars)
char_freqs
```

```
## num_chars
##  4  5  6  7  8  9 10 11 12 13 14
##  3  3  5  8 12  4  4  2  4  3  2
```

```
# barchart of number-of-characters
barplot(char_freqs, las = 1, border = NA)
```



Pasting strings

R provides the `paste()` function. This function allows you to **paste** (i.e. append, concatenate) character vectors separated by a blank space:

```
paste('Pumpkin', 'Pie')
```

```
## [1] "Pumpkin Pie"
```

You can give it any number of vector inputs

```
paste('a', 'b', 'c', 'd', 'e')
```

```
## [1] "a b c d e"
```

You can change the separator with `sep`

```
paste('a', 'b', 'c', 'd', 'e', sep = '-')
```

```
## [1] "a-b-c-d-e"
```

`paste()` is vectorized:

```
paste('a', 1:5, sep = '.')
```

```
## [1] "a.1" "a.2" "a.3" "a.4" "a.5"
```

There's a special wrapper around `paste()` called `paste0()` which is equivalent to `paste(..., sep = "")`

```
# paste0() -vs- paste(..., sep = "")
paste0('Pumpkin', 'Pie')
```

```
## [1] "PumpkinPie"
```

```
paste('Pumpkin', 'Pie', sep = '')
```

```
## [1] "PumpkinPie"
```

```
# paste0() is also vectorized
paste0('a', 1:5)
```

```
## [1] "a1" "a2" "a3" "a4" "a5"
```

Your Turn: Use `paste()` to form a new vector with the first five states and their number of characters like this:

```
"Alabama = 7"  "Alaska = 6"  "Arizona = 7"  "Arkansas = 8"  "California = 10"
```

```
# paste names with their num-of-chars
paste(states[1:5], "=", num_chars[1:5])
```

```
## [1] "Alabama = 7"     "Alaska = 6"      "Arizona = 7"     "Arkansas = 8"
## [5] "California = 10"
```

Now use `paste()`'s argument `collapse = ''` to *collapse* the first five states like this:

```
"AlabamaAlaskaArizonaArkansasCalifornia"
```

```
# collapse first 5 states
paste(states[1:5], collapse = '')
```

```
## [1] "AlabamaAlaskaArizonaArkansasCalifornia"
```

Substrings

R provieds the function `substr()` to extract substrings in a character vector:

```r
# extract first 3 characters
substr('Berkeley', 1, 3)
```

```
## [1] "Ber"
```

Use `substr()` to shorten the state names using the first 3-letters:

```r
# shorten state names with first 3 characters
substr(states, 1, 3)
```

```
##  [1] "Ala" "Ala" "Ari" "Ark" "Cal" "Col" "Con" "Del" "Flo" "Geo" "Haw"
## [12] "Ida" "Ill" "Ind" "Iow" "Kan" "Ken" "Lou" "Mai" "Mar" "Mas" "Mic"
## [23] "Min" "Mis" "Mis" "Mon" "Neb" "Nev" "New" "New" "New" "New" "Nor"
## [34] "Nor" "Ohi" "Okl" "Ore" "Pen" "Rho" "Sou" "Sou" "Ten" "Tex" "Uta"
## [45] "Ver" "Vir" "Was" "Wes" "Wis" "Wyo"
```

Use `substr()` to shorten the state names using the last 3-letters:

```r
# shorten state names with last 3 characters
substr(states, num_chars - 2, num_chars)
```

```
##  [1] "ama" "ska" "ona" "sas" "nia" "ado" "cut" "are" "ida" "gia" "aii"
## [12] "aho" "ois" "ana" "owa" "sas" "cky" "ana" "ine" "and" "tts" "gan"
## [23] "ota" "ppi" "uri" "ana" "ska" "ada" "ire" "sey" "ico" "ork" "ina"
## [34] "ota" "hio" "oma" "gon" "nia" "and" "ina" "ota" "see" "xas" "tah"
## [45] "ont" "nia" "ton" "nia" "sin" "ing"
```

How would you shorten the state names using the first letter and the last 3-letters? For instance: `"Aama" "Aska" "Aona" "Asas"` etc.

```r
# shorten state names with first 3 characters
paste0(substr(states, 1, 1),
       substr(states, num_chars - 2, num_chars))
```

```
##  [1] "Aama" "Aska" "Aona" "Asas" "Cnia" "Cado" "Ccut" "Dare" "Fida" "Ggia"
## [11] "Haii" "Iaho" "Iois" "Iana" "Iowa" "Ksas" "Kcky" "Lana" "Mine" "Mand"
## [21] "Mtts" "Mgan" "Mota" "Mppi" "Muri" "Mana" "Nska" "Nada" "Nire" "Nsey"
## [31] "Nico" "Nork" "Nina" "Nota" "Ohio" "Ooma" "Ogon" "Pnia" "Rand" "Sina"
## [41] "Sota" "Tsee" "Txas" "Utah" "Vont" "Vnia" "Wton" "Wnia" "Wsin" "Wing"
```

Challenge

We already obtained a frequency table `char_freqs` with the counts of state names by number of characters. You can use those frequencies to get those state-names with 4-characters or 10-characters:

```r
# 4-char states
states[num_chars == 4]
```

```
## [1] "Iowa" "Ohio" "Utah"
```

```r
# 10-char states
states[num_chars == 10]
```

```
## [1] "California" "New Jersey" "New Mexico" "Washington"
```

You can use `paste()` to join the 4-character states in one single string (i.e. *collapsing*) like this—separated by a comma and space—:

```r
# collapse 4-char states
paste(states[num_chars == 4], collapse = ", ")
```

```
## [1] "Iowa, Ohio, Utah"
```

Here's one challenge for you: write code (using a for-loop) to obtain a list `states_list` containing the collapsed names by number of characters. If the number of characters is an even number, then the state names should be in capital letters. Otherwise, they should be in lower case letters.

Each list element of `states_list` must be named with the number of characters, followed by a dash, followed by the word `chars`: e.g. `'4-chars'`, `'5-chars'`, etc. In total, `states_list` should have the same length as `char_freqs`.

Here's what `states_list` should look like for the first three elements:

```
$`4-chars`
[1] "IOWA, OHIO, UTAH"

$`5-chars`
[1] "idaho, maine, texas"

$`6-chars`
[1] "ALASKA, HAWAII, KANSAS, NEVADA, OREGON"
```

```r
char_freqs <- table(num_chars)
char_classes <- as.numeric(names(char_freqs))
states_list <- vector("list", length(char_classes))
```

```r
char_classes
```

```
##  [1]  4  5  6  7  8  9 10 11 12 13 14
```

```r
states_list
```

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

```
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
##
## [[7]]
## NULL
##
## [[8]]
## NULL
##
## [[9]]
## NULL
##
## [[10]]
## NULL
##
## [[11]]
## NULL
```

```r
for (i in 1:length(char_freqs)) {
  collapsed <- paste0(states[num_chars == char_classes[i]], collapse = ", ")
  if (char_classes[i] %% 2 == 0) {
    states_list[[i]] <- toupper(collapsed)
  } else {
    states_list[[i]] <- tolower(collapsed)
  }
}

names(states_list) <- paste0(char_classes, "-chars")
```

Converting from Fahrenheit Degrees

Here are four functions that convert from Fahrenheit degrees to other temperature scales:

```r
to_celsius <- function(x = 1) {
  (x - 32) * (5/9)
}

to_kelvin <- function(x = 1) {
  (x + 459.67) * (5/9)
}
```

```r
to_reaumur <- function(x = 1) {
  (x - 32) * (4/9)
}

to_rankine <- function(x = 1) {
  x + 459.67
}
```

We can use the previous functions to create a more general function `temp_convert()`:

```r
temp_convert <- function(x = 1, to = "celsius") {
  switch(to,
         "celsius" = to_celsius(x),
         "kelvin" = to_kelvin(x),
         "reaumur" = to_reaumur(x),
         "rankine" = to_rankine(x))
}

temp_convert(30, 'celsius')
```

```
## [1] -1.111111
```

`temp_convert()` works fine when the argument `to = 'celsius'`. But what happens if you try `temp_convert(30, 'Celsius')` or `temp_convert(30, 'CELSIUS')`?

Your turn. Rewrite `temp_convert()` such that the argument `to` can be given in upper or lower case letters. For instance, the following three calls should be equivalent:

```r
temp_convert(30, 'celsius')
temp_convert(30, 'Celsius')
temp_convert(30, 'CELSIUS')
```

```r
# your function temp_convert
temp_convert <- function(x = 1, to = "celsius") {
  to <- tolower(to)
  switch(to,
         "celsius" = to_celsius(x),
         "kelvin" = to_kelvin(x),
         "reaumur" = to_reaumur(x),
         "rankine" = to_rankine(x))
}
```

Names of files

Imagine that you need to generate the names of 10 data `.csv` files. All the files have the same prefix name but each of them has a different number: `file1.csv`, `file2.csv`, … , `file10.csv`.

How can you generate a character vector with these names in R? Come up with at least three different ways to get such a vector:

```
# vector of file names
paste0("file", 1:10, ".csv")
paste("file", 1:10, ".csv", sep = "")
sprintf("file%s.csv", 1:10)
```

Now imagine that you need to rename the characters `file` into `dataset`. In other words, you want the vector of file names to look like this: `dataset1.csv`, `dataset2.csv`, … , `dataset10.csv`. Take the previous vector of file names and rename its elements:

```
# rename vector of file names
files <- sprintf("file%s.csv", 1:10)
datasets <- gsub(pattern = "file", replacement = "dataset", files)
datasets
```

```
##  [1] "dataset1.csv"  "dataset2.csv"  "dataset3.csv"  "dataset4.csv"
##  [5] "dataset5.csv"  "dataset6.csv"  "dataset7.csv"  "dataset8.csv"
##  [9] "dataset9.csv"  "dataset10.csv"
```

Using function `cat()`

Run the following code:

```
# name of output file
outfile <- "output.txt"

# writing to 'outfile.txt'
cat("This is the first line", file = outfile)
# insert new line
cat("\n", file = outfile, append = TRUE)
cat("A 2nd line", file = "output.txt", append = TRUE)
# insert 2 new lines
cat("\n\n", file = outfile, append = TRUE)
cat("\nThe quick brown fox jumps over the lazy dog\n",
    file = outfile, append = TRUE)
```

After running the previous code, look for the file `output.txt` in your working directory and open it. One of the uses of `cat()` is to write contents to a text file. Note that the first call to `cat()` does not include the argument `append`. The rest of the calls do include `append = TRUE`.

Your turn. Modify the script such that the content of `output.txt` looks like the *yaml* header of an `.Rmd` file with your information:

```
---
title: "Some title"
author: "Your name"
date: "today's date"
output: html_document
```

```
---

This is the first line
A 2nd line


The quick brown fox jumps over the lazy dog
```

Valid Color Names

The function `colors()` returns a vector with the names (in English) of 657 colors available in R. Write a function `is_color()` to test if a given name—in English—is a valid R color. If the provided name is a valid R color, `is_color()` returns `TRUE`. If the provided name is not a valid R color `is_color()` returns `FALSE`.

```r
is_color <- function(str) {
  str %in% colors()
}




# test it:
is_color('yellow')  # TRUE

is_color('blu')     # FALSE

is_color('turkuoise') # FALSE
```

Plot with a valid color

Use `is_color()` to create the function `colplot()` that takes one argument `col` (the name of a color) to produce a simple scatter plot with random numbers (e.g. use `runif()` or `rnorm()` to get point coordinates).

If `col` is a valid name—say "blue"—, the scatterplot should show a title `"Testing color blue"`.

If the provided `col` is not a valid color name, e.g. "blu", then the function must stop, showing an **error message** `"invalid color blu"`.

```r
colplot <- function(col){
  if(is_color(col) == FALSE) {
    stop(paste0("invalid color ", col))
  } else {
    x <- rnorm(10)
    y <- rnorm(10)
    plot(x, y, main = sprintf("Testing color %s", col), col = col)
```

```
  }
}
```

```
# this should plot
colplot('tomato')

# this stops with error message
colplot('tomate')
```

Counting number of vowels

Consider the following vector `letrs` which contains various letters randomly generated:

```
# random vector of letters
set.seed(1)
letrs <- sample(letters, size = 100, replace = TRUE)
head(letrs)
```

```
## [1] "g" "j" "o" "x" "f" "x"
```

If you were to count the number of vowels in `letrs` you would get the following counts:

- a: 2
- e: 2
- i: 6
- o: 2
- u: 8

Write code in R to count the number of vowels in vector `letrs`. Test your code with `letrs` and verify that you get the same counts for each vowel.

```
# count number of vowels
vowels <- c("a", "e", "i", "o", "u")

for (i in 1:length(vowels)) {
  counts <- sum(vowels[i] == letrs)
  print(paste0(vowels[i], ": ", counts))
}
```

```
## [1] "a: 2"
## [1] "e: 2"
## [1] "i: 6"
## [1] "o: 2"
## [1] "u: 8"
```

```
vowels <- c('a', 'e', 'i', 'o', 'u')
letrs %in% vowels
```

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

```
##  [12]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
##  [34]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [45] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [56] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
##  [78] FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
## [100] FALSE
```

```
letrs[letrs %in% vowels]
```

```
##  [1] "o" "e" "u" "u" "a" "i" "e" "u" "u" "o" "u" "a" "i" "i" "i" "i" "u"
## [18] "i" "u" "u"
```

```
table(letrs[letrs %in% vowels])
```

```
##
## a e i o u
## 2 2 6 2 8
```

Now do the same but for the consonants, that is, count the frequencies of consonants in `letrs`. You should get the following counts:

```
b c d f g h j k l m n p q r s t v w x y z
3 3 3 4 6 1 5 6 4 7 2 2 5 4 5 3 4 5 4 3 1
```

```
consonants <- letters[!(letters %in% vowels)]
table(letrs[letrs %in% consonants])
```

```
##
## b c d f g h j k l m n p q r s t v w x y z
## 3 3 3 4 6 1 5 6 4 7 2 2 5 4 5 3 4 5 4 3 1
```

Number of letters, vowels, and consonants

Write a function `count_letters()` that takes a vector of letters (e.g. `letrs`) and computes the total number of letters, the total number of vowels, and the total number of consonants. For instance, given the vector `letrs`, the function will print on console:

```
"letters: 100"
"vowels: 20"
"consonants: 80"
```

```
# counting total letters,vowels and consonants
count_letters <- function(x) {
  # auxiliar vectors
  vowels <- c('a', 'e', 'i', 'o', 'u')
  consonants <- letters[!(letters %in% vowels)]
  # count letters, vowels, and consonants
```

```r
  print(paste('letters:', sum(nchar(letrs))))
  print(paste('vowels:', sum(letrs %in% vowels)))
  print(paste('consonants:', sum(letrs %in% consonants)))
}

count_letters(letrs)
```

```
## [1] "letters: 100"
## [1] "vowels: 20"
## [1] "consonants: 80"
```