R code

Sean Raleigh

From "Programming with Data", Chapter 2 of *Data Science for Mathematicians*

Section 2.3.1

The following R code calculates the mean of 10 randomly sampled values from a uniform distribution on [0, 1] (`runif` stands for "random uniform") and then repeats that process 100 times.

```
set.seed (42)
simulated_data <- vector(length = 100)
for(i in 1:100) {
    simulated_data[i] <- mean(runif(10, 0, 1))
}
```

A more flexible way to do this—and one that will usually result in fewer bugs—is to give names to various parameters and assign them values once at the top:

```
set.seed (42)
reps <- 100
n <- 10
lower <- 0
upper <- 1
simulated_data <- vector(length = reps)
for(i in 1:reps) {
    simulated_data[i] <- mean(runif(n, lower , upper))
}
```

Section 2.3.5

In preparation for the example from the text, load the `testthat` package. (If the following command does not work, you may need to `install.packages("testthat")` first.)

```
library(testthat)
```

Define a simple function called `test_parity`:

```
test_parity <- function(int_value) {
    parity <- (int_value) %% 2
    if (parity == 0) print("even")
    if (parity == 1) print("odd")
}
```

Make sure the file `parity_test_file.R` is located in the same directory as this notebook file. The following code will run a unit test to see if the function does the right thing with a few sample values.

```
testthat::test_file("parity_test_file.R")
```

```
##   |  OK F W S | Context
## / |   0       | parity_test_file  |   3       | parity_test_file
##
##   Results
## Duration: 0.2 s
##
## OK:       3
## Failed:   0
## Warnings: 0
## Skipped:  0
```

Figure 2.7

```r
my_list <- list(
  1:10,
  c("Sean", "Raleigh"),
  data.frame(letter = c("a", "b", "c", "d", "e"),
             position = 1:5)
)
```

```r
my_list
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
## [1] "Sean"    "Raleigh"
##
## [[3]]
##   letter position
## 1      a        1
## 2      b        2
## 3      c        3
## 4      d        4
## 5      e        5
```

Section 2.4.3.2

Here is a fake data frame for the "Utah" example:

```r
df <- data.frame(
  lastname = c("Reed", "Reynolds", "Rice", "Richards",
               "Richardson", "Roberts", "Roberts", "Robertson",
               "Rogers", "Ross", "Ross", "Russell"),
  occupation = c("plumber", "clerk", "retail", "food service",
                 "computer engineer", "administrator",
                 "manager", "accountant",
                 "nurse", "server", "teacher", "mechanic"),
  city = c("Salt Lake City", "Salt Lake City", "St. George",
           "West Valley City", "Provo", "Murrary",
           "Orem", "Sandy", "Draper",
           "Cottonwood Heights", "Logan", "Ogden"),
  state = c("UT", "ut", "Ut", "ut", "UT", "ut",
            "UT", "ut", "Ut", "ut", "ut", "Ut")
)
```

```r
df
```

```
##      lastname        occupation               city state
## 1        Reed           plumber     Salt Lake City    UT
## 2    Reynolds             clerk     Salt Lake City    ut
## 3        Rice            retail         St. George    Ut
## 4    Richards      food service   West Valley City    ut
## 5  Richardson computer engineer              Provo    UT
## 6     Roberts     administrator            Murrary    ut
## 7     Roberts           manager               Orem    UT
## 8   Robertson        accountant              Sandy    ut
## 9      Rogers             nurse             Draper    Ut
## 10       Ross            server Cottonwood Heights    ut
## 11       Ross           teacher              Logan    ut
## 12    Russell          mechanic              Ogden    Ut
```

Look at the 12th row:

```r
df[12, ]
```

```
##    lastname occupation  city state
## 12  Russell   mechanic Ogden    Ut
```

If you find an instance of "Ut" in an R data frame that you want to change to "UT," you could just note that it appears in the the 12th row and the 4th column, and fix it with code like the following one-liner.

```r
df[12, 4] <- "UT"
```

Now look at the 12th row again:

```r
df[12, ]
```

```
##    lastname occupation  city state
## 12  Russell   mechanic Ogden    UT
```

Now inspect the whole `state` column:

```r
df['state']
```

```
##    state
## 1     UT
## 2     ut
## 3     Ut
## 4     ut
## 5     UT
## 6     ut
## 7     UT
## 8     ut
## 9     Ut
## 10    ut
## 11    ut
## 12    UT
```

Since there are other instances of "Ut" in the data, it would make a lot more sense to write code to fix every instance of "Ut." In R, that code looks like the following.

```r
df$state[df$state == "Ut"] <- "UT"
```

Here's the 4th column again:

```r
df['state']
```

```
##    state
## 1     UT
## 2     ut
## 3     UT
## 4     ut
## 5     UT
## 6     ut
## 7     UT
## 8     ut
## 9     UT
## 10    ut
## 11    ut
## 12    UT
```

Going a step further, the following code uses the `toupper` function to convert all state names to uppercase first, which would also fix any instances of "ut."

```r
df$state[toupper(df$state) == "UT"] <- "UT"
```

```r
df['state']
```

```
##    state
## 1     UT
## 2     UT
## 3     UT
## 4     UT
## 5     UT
## 6     UT
## 7     UT
## 8     UT
## 9     UT
## 10    UT
## 11    UT
## 12    UT
```

Figure 2.9

```r
student_test_data <- data.frame(
  student = c("A", "B"),
  test1 = c(72, 90),
  test2 = c(75, 92),
  test3 = c(69, 98)
)
```

```r
student_test_data
```

```
##   student test1 test2 test3
## 1       A    72    75    69
## 2       B    90    92    98
```

Making this data "long" can be done using the `pivot_longer` function from the `tidyr` package. (If the following command does not work, you may need to `install.packages("tidyr")` first.)

```r
library("tidyr")
```

```
##
## Attaching package: 'tidyr'
## The following object is masked from 'package:testthat':
##
##     matches
```

```
student_test_data_long <-
    pivot_longer(student_test_data,
                 cols = c("test1", "test2", "test3"),
                 names_to = "test",
                 values_to = "score")
```

```
student_test_data_long
```

```
## # A tibble: 6 x 3
##   student test  score
##   <fct>   <chr> <dbl>
## 1 A       test1    72
## 2 A       test2    75
## 3 A       test3    69
## 4 B       test1    90
## 5 B       test2    92
## 6 B       test3    98
```

The pivot_wider function transforms back to the "wide" version:

```
pivot_wider(student_test_data_long,
            id_cols = "student",
            names_from = "test",
            values_from = "score")
```

```
## # A tibble: 2 x 4
##   student test1 test2 test3
##   <fct>   <dbl> <dbl> <dbl>
## 1 A          72    75    69
## 2 B          90    92    98
```

Figure 2.10

```
obs_color_data <- data.frame(
  observation = factor(c("A", "B", "C", "D", "E", "F")),
  color = factor(c("Red", "Red", "Blue", "Green", "Red", "Green"),
                 levels = c("Red", "Blue", "Green"))
)
```

```
obs_color_data
```

```
##   observation color
## 1           A   Red
## 2           B   Red
## 3           C  Blue
## 4           D Green
## 5           E   Red
## 6           F Green
```

Generally speaking, categorical encoding is done behind the scenes: the functions you use to analyze data will either do it under the hood automatically when needed, or will allow you to specify that you want a certain kind of encoding as an argument to some function in your pipeline. It is rare that you would need to perform the encoding manually and store it in a data frame, as illustrated in Figure 2.10.

Nevertheless, we can use the `model.matrix` function to peek under the hood at part of the process that prepares data sets for regression tasks.

Here is an example of dummy encoding. Ignore the column labeled `(Intercept)`; that is part of a linear regression model that doesn't concern us here.

```
model.matrix(~ color, data = obs_color_data)
```

```
##   (Intercept) colorBlue colorGreen
## 1           1         0          0
## 2           1         0          0
## 3           1         1          0
## 4           1         0          1
## 5           1         0          0
## 6           1         0          1
## attr(,"assign")
## [1] 0 1 1
## attr(,"contrasts")
## attr(,"contrasts")$color
## [1] "contr.treatment"
```

This output is similar to the rightmost panel in Figure 2.10.

If we tell R to remove the intercept term, the encoding scheme (often called a "contrast" in R and other places), becomes one-hot encoding.

```
model.matrix(~ 0 + color, data = obs_color_data)
```

```
##   colorRed colorBlue colorGreen
## 1        1         0          0
## 2        1         0          0
## 3        0         1          0
## 4        0         0          1
## 5        1         0          0
## 6        0         0          1
## attr(,"assign")
## [1] 1 1 1
## attr(,"contrasts")
## attr(,"contrasts")$color
## [1] "contr.treatment"
```

This is like the output in the center panel of Figure 2.10.