Coding Lab: iteration and loops

Ari Anisfeld

Fall 2020

Iteration and for-loops (Control flow II)

We use for-loops to repeat a task over many different inputs or to repeat a simulation process several times.

- How to write for-loops
- When to use a for-loop vs vectorized code

```r
for(value in c(1, 2, 3, 4, 5)) {
  print(value)
}
```

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

Simple for-loop

```r
for (x in c(3, 6, 9)) {
  print(x)
}
```

```
## [1] 3
## [1] 6
## [1] 9
```

Simple for-loop: what is going on?

```
for (x in c(3, 6, 9)) {
  print(x)
}
```

Our for-loop is equivalent to the following code. For each value in `c(3,6,9)`, we assign the value to `x` and the do the action between the curly brackets in order.

```
x <- 3
print(x)
x <- 6
print(x)
x <- 9
print(x)
```

For loops

The general structure of a for loop is as follows:

```
for (value in list_of_values) {
  do something (based on value)
}

for (index in list_of_indices) {
  do something (based on index)
}
```

Example: find sample means

Suppose we want to find the means of increasingly large samples.

```r
mean1 <- mean(rnorm(5))
mean2 <- mean(rnorm(10))
mean3 <- mean(rnorm(15))
mean4 <- mean(rnorm(20))
mean5 <- mean(rnorm(25000))

means <- c(mean1, mean2, mean3, mean4, mean5)

means
```

```
## [1] -0.91736867  0.21439972 -0.11878419  0.15339323  0.0
```

Example: find sample means

Let's avoid repeating code with a `for` loop.

```r
sample_sizes <- c(5, 10, 15, 20, 25000)
sample_means <- rep(0, length(sample_sizes))

for (i in seq_along(sample_sizes)) {
  sample_means[[i]] <- mean(rnorm(sample_sizes[[i]]))
}

sample_means
```

```
## [1] -0.598897197  0.405766348 -0.066943761 -0.064758312
```

In the following slides we'll explain each step.

Finding sample means, broken down

Assign initial variables **before** starting the for loop.

```r
# determine what to loop over
sample_sizes <- c(5, 10, 15, 20, 25000)

# pre-allocate space to store output
sample_means <- rep(0, length(sample_sizes))
```

To start:

1. create a vector of the `sample_sizes` we want to use
2. create a vector to store the output

What does sample_means currently look like?

```r
sample_means <- rep(0, length(sample_sizes))
sample_means
```

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

Why do this? It makes the code more efficient. An alternative is to build up an object as you go, but this requires copying the data over and over again.

Alternative ways to preallocate space

```r
sample_means <- vector("double", length = 5)
sample_means <- double(5)
```

Each data type has a comparable function e.g. `logical()`, `integer()`, `character()`.

To hold data of different types, we'll use lists.[1]

```r
data_list <- vector("list", length = 5)
```

[1] Lists are vectors that are not "atomic".

Adding data to a vector, broken down

Determine what sequence to loop over.

```r
for (i in 1:length(sample_sizes)) {

}
```

A helper function `seq_along()`

`seq_along(x)` is synonymous to `1:length(x)`

where `x` is a vector.

Example

```r
vec <- c("x", "y", "z")
1:length(vec)
```

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

```r
seq_along(vec)
```

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

```r
sample_sizes <- c(5, 10, 15, 20, 25000)
seq_along(sample_sizes)
```

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

(What if `sample_sizes` is accidentally a 0-length vector? See what happens in R for Data Science.)

Adding data to a vector, broken down

```r
sample_sizes <- c(5, 10, 15, 20, 25000)
sample_means <- rep(0, length(sample_sizes))

for (i in seq_along(sample_sizes)) {

}
```

Use `seq_along()` to be safe!

Adding data to a vector, broken down

```r
sample_sizes <- c(5, 10, 15, 20, 25000)
sample_means <- numeric(length(sample_sizes))

for (i in seq_along(sample_sizes)) {

  sample_means[[i]] <-  mean(rnorm(sample_sizes[[i]]))

}

sample_means
```

```
## [1] 0.133747643 0.078506814 0.174355355 0.098998378 0.00
```

Save the mean of the sample to the ith place of the `sample_means` vector.

Common error.

This code falls, because we do not store the output in `sample_means` in the for loop! (Compare to previous slide).

```r
sample_sizes <- c(5, 10, 15, 20, 25000)
sample_means <- rep(0, length(sample_sizes))

for (i in seq_along(sample_sizes)) {
  mean(rnorm(sample_sizes[[i]]))
}

sample_means
```

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

Right now, we're calculating the mean, but it's not being saved anywhere.

Another example

You get data stored in split over several `csv` files.

We can read the data into R and store it store it as a single data set.

```r
setwd("../data/loops")

file_1 <- read_csv("data_1999.csv")
file_2 <- read_csv("data_2000.csv")
...
file_22 <- read_csv("data_2020.csv")

data <- bind_rows(file_1, file_2, ..., file_22)
```

Aside: how to make the data

The data used for this exercise is fake data which I made with a for-loop. Run the code bellow (*choose your own working directory*) to follow along.

```
setwd('../data/loops')

file_list <- paste0("data_", 1999:2020, ".csv")

for (file in file_list) {
  data <-
    tibble(id = 1:100,
           employed = sample(c(0, 1, 1, 1),
                             100, replace = TRUE),
             happy = sample(c(0,1),
                            100, replace = TRUE))

  write_csv(data, file)
}
```

Aside: bind_rows()?

`bind_rows()` stacks two dataframe, or combines two vectors into a dataframe:

```r
df_1 <- tibble(col1 = 1, col2 = "A")
df_2 <- tibble(col1 = 2:3, col2 = c("B", "C"))

bind_rows(df_1, df_2)
```

```
## # A tibble: 3 x 2
##    col1 col2
##   <dbl> <chr>
## 1     1 A
## 2     2 B
## 3     3 C
```

Aside: `list.files()`

`?list.files()`:

These functions produce a character vector of the names of files ... in the named directory.

- `pattern` ensures we only take the csv files.
- It uses *regular expressions* where `*` in `*.csv$` matches any string and `.csv` ensures the string ends in `csv`.

```r
list.files("../data/loops", pattern = "*.csv$")
```

```
##  [1] "data_1999.csv" "data_2000.csv" "data_2001.csv" "da
##  [5] "data_2003.csv" "data_2004.csv" "data_2005.csv" "da
##  [9] "data_2007.csv" "data_2008.csv" "data_2009.csv" "da
## [13] "data_2011.csv" "data_2012.csv" "data_2013.csv" "da
## [17] "data_2015.csv" "data_2016.csv" "data_2017.csv" "da
## [21] "data_2019.csv" "data_2020.csv"
```

Let's use a loop to read in the data

```
setwd('../data/loops')

# by default, reads files in working directory
file_list <- list.files(pattern = "*.csv$")

out <- tibble()

for (file in file_list) {
  temp <- read_csv(file)

  out <- bind_rows(out, temp)
}

nrow(out)
```

```
## [1] 2200
```

Review: Vectorized operations

When possible, take advantage of the fact that R is vectorized.

```r
a <- 7:11
b <- 8:12
out <- rep(0L, 5)

for (i in seq_along(a)) {
  out[[i]] <- a[[i]] + b[[i]]
}

out
```

```
## [1] 15 17 19 21 23
```

This is a bad example of a for loop!

The better alternative: vectorized addition

```
a <- 7:11
b <- 8:12
out <- a + b

out
```

```
## [1] 15 17 19 21 23
```

Use vectorized operations and tidyverse functions like `mutate()` when you can.

Key points: iteration

- Iteration is useful when we are repeatedly calling the same block of code or function while changing one (or two) inputs.
- If you can, use vectorized operations.
- Otherwise, for loops work for iteration
  - Clearly define what you will iterate over (values or indicies)
  - Preallocate space for your output (if you can)
  - The body of the for-loop has parametrized code based on thing your iterating over
  - Debug as you code by testing your understanding of what the for-loop should be doing (e.g. using `print()`)

Further study: Many R coders prefer the `map()` family functions from `purrr` or base R `apply` family. See iteration in R for Data Science