Accelerated Lecture 8: Iteration and Loops

Harris Coding Camp

Summer 2022

Iteration and for-loops

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^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

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)
}
```

The for-loop is equivalent to the following code.

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

Simple for-loop: what is going on?

The variable (here `item`) is a **name** that you pick!

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

- it can be `anything`!!

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

General structure of a `for` loop

The general structure of a `for` loop is as follows:

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

Main components: Sequence, Body

Components of a `for` loop

```
for (z in c(5, 4, 3)) {
  print(z/2)
}
```

1. **Sequence**. Determines what to "loop over"
   - sequence above is `for (z in c(5, 4, 3))`
   - this creates a variable `z`
   - we assign `z` to values `c(5, 4, 3)` *iteratively*
     - in the first iteration, `z` is 5
     - in the second iteration, `z` is 4, etc.
2. **Body**. What to execute as we run through the loop.
   - Body in above loop is `print(z/2)`
   - Each iteration, the body prints the value of `z/2`

The output.

for each value in `c(5,4,3)`, we divide it by 2.

```
for (z in c(5, 4, 3)) {
  print(z/2)
}
```

```
## [1] 2.5
## [1] 2
## [1] 1.5
```

Of course, we have an easier way to divide items in a vector by 2

```
c(5, 4, 3) / 2
```

```
## [1] 2.5 2.0 1.5
```

Components of a `for` loop

1. **Sequence**. Determines what to “loop over”

- often we loop over indices.
- recall, we can refer to items in a vector by their location

```
values <- c(5, 4, 3)

for (i in 1:3) {
  print(values[[i]]/2)
}
```

```
## [1] 2.5
## [1] 2
## [1] 1.5
```

When to write a loop or use an iteration method

Grolemund and Wickham: **don't copy and paste more than twice**

- instead consider **a loop or function**

Broadly, rationale for writing loop:

- Can make changes to code in one place rather than many
- Easier to read

When to write a loop vs a functions

Loops are useful when:

- a similar task is repeated many times in a row
- you *cannot use a vectorized option*

Examples:

```
- read in data sets from individual years; each csv only differs by nam
```

Functions are useful when:

- we anticipate repeating tasks at different points in time
- we require flexible and ad-hoc usage of the code
- code is complex and naming and encapsulation helps clarify code functionality

Often we write functions and then put them in loops or other iterators!

Recipe for how to write loop

The general recipe for writing a loop:

1. Complete the task for one instance outside a loop
2. a) Decide which part(s) of the **body** will change with each iteration
3. b) Write the **sequence**
4. Usually you want to store the output, create an object to store the output outside of the loop
5. Construct the loop

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]  0.182460654  0.356973138 -0.320532177 -0.047843672 -0.001224917
```

Example: find sample means

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

```
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]  0.278181799  0.173378712 -0.216770250 -0.085438369 -0.002919184
```

In the following slides we'll explain each step.

Finding sample means, broken down

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

```
# 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))
```

Why do we make `sample_means`?

Why do this? It makes the code more efficient.

- An alternative is to build up an object as you go (see lab material)
- This requires copying the data over and over again and your loop will be slow

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

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

Reviewing alternative ways to pre-allocate space

```
sample_means <- vector("double", length = 5)
sample_means <- numeric(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 or tibbles.

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

Adding data to a vector, broken down

Determine what sequence to loop over.

- we iterate over the indices!
- Numbers from 1 to length(sample_sizes)

```
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.

Simple Example

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

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

```
seq_along(vec)
```

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

A helper function `seq_along()`

`seq_along()` protects against that moment when `length(x) = 0`

- you might worry about this in a function when you don't have control over the input.

```
seq_along(NULL)
```

```
## integer(0)
```

```
# equivalent to 1:0
1:length(NULL)
```

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

A helper function seq_along()

Back to Our Example

```
sample_sizes <- c(5, 10, 15, 20, 25000)
1:length(sample_sizes)
```

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

```
seq_along(sample_sizes)
```

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

Adding data to a vector, broken down

Add `for`-loop structure:

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

for (i in seq_along(sample_sizes)) {

}
```

When writing loops, it's very common to create a **sequence** from 1 to the length (i.e., number of elements) of an object.

- **sequence**: `for (i in seq_along(sample_sizes))`
  - `i` takes on 1, 2, 3, 4 and 5 *sequentially*
  - Sequence iterates through the *position number* or *index* of each element in `sample_sizes`

Adding data to a vector, broken down

Add `for`-loop body:

```
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]]))
}
```

body: value of `i` refers to the *position number* or *index* of the i^{th} element in `sample_sizes`

- Access element contents using `sample_sizes[[i]]`
- Here, save the output as the i^{th} element in `sample_means`

Adding data to a vector, broken down

Now `sample_means` has stored the results of iteratively running our code!

```
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] -0.16515550  0.45082350  0.14683349  0.23165429  0.00276221
```

To belabor the point

Our code is equivalent to . . .

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

i <- 1
sample_means[[i]] <- mean(rnorm(sample_sizes[[i]]))
i <- 2
sample_means[[i]] <- mean(rnorm(sample_sizes[[i]]))
i <- 3
sample_means[[i]] <- mean(rnorm(sample_sizes[[i]]))
i <- 4
sample_means[[i]] <- mean(rnorm(sample_sizes[[i]]))
i <- 5
sample_means[[i]] <- mean(rnorm(sample_sizes[[i]]))


sample_means
```

```
## [1]  0.34862368 -0.05109775  0.07100906  0.38975346 -0.01302429
```

You try: Why doesn't this work?

- Run the code
- Try to fix it.

```
sample_sizes <- 1:5
sample_means <- rep(0, length(sample_sizes))

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

sample_means
```

Aside: Common errors

This code falls, why?

- It's not *not* running!

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

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

```
## [1] -0.6481123
## [1] 0.6777631
## [1] 0.08803558
## [1] -0.026407
## [1] 0.0007925727
```

```
sample_means
```

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

Aside: Common errors

This code falls, why?

- It's running!
- But we didn't save the output!

```
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] -0.620382231  0.074542587  0.255263461  0.297927083  0.002077501
```

Aside: Common errors

What's wrong with this code?

```
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[[1]]))
}

sample_means
```

```
## [1]  0.94194026 -0.64252404  0.02064714  0.26091892 -0.16567702
```

Aside: Debugging with cat()

cat() also prints out the output - and can handle variables!

```
for (i in seq_along(sample_sizes)) {

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

  cat("mean of sample", i, "is", sample_means[[i]],
      fill = TRUE)
}
```

```
## mean of sample 1 is 0.0304767
## mean of sample 2 is 0.2253846
## mean of sample 3 is -0.3087565
## mean of sample 4 is -0.09129111
## mean of sample 5 is 0.001660984
```

Aside: Debugging with cat()

What does fill=TRUE do?

```
for (i in seq_along(sample_sizes)) {

  sample_means[[i]] <- mean(rnorm(sample_sizes[[i]]))
  cat("mean of sample", i, "is", sample_means[[i]])
}
```

```
## mean of sample 1 is 0.4646038mean of sample 2 is 0.1712121mean of sa
```

Aside: Debugging with `cat()`

We can explicitly make a new line with "`\n`" or use `fill = TRUE`.

```
for (i in seq_along(sample_sizes)) {

  sample_means[[i]] <- mean(rnorm(sample_sizes[[i]]))
  cat("mean of sample", i, "is", sample_means[[i]], "\n")
}
```

```
## mean of sample 1 is -0.3538945
## mean of sample 2 is -0.3669583
## mean of sample 3 is -0.372049
## mean of sample 4 is -0.09865239
## mean of sample 5 is -0.004259006
```

Aside: Debugging with `cat()`

`cat()` prints vectors directly

- i.e. it's not vectorized

```
names <- c("Sam", "Yuchen", "Yunjoo")
cat("Our TA", names, "is great!", fill=TRUE)
```

```
## Our TA Sam Yuchen Yunjoo is great!
```

```
# paste separates input with " " by default
# paste0 does no separation.
print(paste("Our TA", names, "is great!"))
```

```
## [1] "Our TA Sam is great!"    "Our TA Yuchen is great!"
## [3] "Our TA Yunjoo is great!"
```

```
# What about print by itself?
# ... throws an ERROR
print("Our TA", names, "is great!")
```

Try it yourself

1. Create a numeric vector that has year of birth of members of your family
   - you decide who to include
   - e.g., `birth_years <- c(1944, 1950, 1981, 2016)`
2. Write a loop that calculates the age of each member of you family.
3. Print the output in sentences
   - e.g. The age of family member `i` is...

Note: multiple correct ways to complete this task

4. Write the same code vectorized.

Review: Vectorized operations

When possible, take advantage of vectorization!

```
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

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

out
```

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

Use vectorized operations when you can.

- easier to read code
- easier to write code (eventually!)

What happens when we loop over a tibble?

```
df <- tibble(a = rnorm(4), b = rnorm(4))
df
```

```
## # A tibble: 4 x 2
##         a      b
##     <dbl>  <dbl>
## 1 -0.0800  0.644
## 2 -1.04   -0.185
## 3 -0.469  -0.370
## 4 -0.395  -1.12
```

```
for (i in seq_along(df)) {
  cat("value of object", i, "=", df[[i]], "\n")
}
```

```
## value of object 1 = -0.0799726 -1.040806 -0.4685387 -0.3954239
## value of object 2 = 0.643723 -0.1851511 -0.3696833 -1.115587
```

We loop over columns, *not* rows!

It unnatural to loop over *rows*

We have vectorized functions with mutate or $<-

```
data %>%
  mutate(new_col = something_vectorized(old_col))

data$new_col <- something_vectorized(old_col1, old_col2)
```

and `rowwise()` for unvectorized code

```
data %>%
  rowwise() %>%
  mutate(new_col = something_unvectorized(old_col))
```

And, of course, you can pull out a column as a vector and iterate over it.

An example of iterating over columns

Task: calculates z-scores for a set of variables in a data frame

First, create sample data

```
# matrix is like a 2d atomic vector
set.seed(4)
df <- as_tibble(matrix(runif(40), ncol = 4))
names(df) <- c("a", "b", "c", "d")
head(df)
```

```
## # A tibble: 6 x 4
##         a     b     c     d
##     <dbl> <dbl> <dbl> <dbl>
## 1 0.586   0.755 0.715 0.567
## 2 0.00895 0.286 0.997 0.239
## 3 0.294   0.100 0.506 0.878
## 4 0.277   0.954 0.490 0.655
## 5 0.814   0.416 0.649 0.482
## 6 0.260   0.455 0.831 0.971
```

An Example of iterating over columns

The z-score for observation i is the number of standard deviations from mean:

$z_i = \frac{x_i - \bar{x}}{sd(x)}$

Let's calculate z-score for first 4 observations of `df$a`:

```
(df$a[1] - mean(df$a, na.rm=TRUE))/sd(df$a, na.rm=TRUE)
```

```
## [1] 0.2768789
```

```
(df$a[2] - mean(df$a, na.rm=TRUE))/sd(df$a, na.rm=TRUE)
```

```
## [1] -1.377454
```

```
(df$a[3] - mean(df$a, na.rm=TRUE))/sd(df$a, na.rm=TRUE)
```

```
## [1] -0.5607078
```

Hmm ... Maybe we need a function!

```
calc_z_score <- function(x, i, na.rm = TRUE) {
  (x[i] - mean(x, na.rm = na.rm)) / sd(x, na.rm = na.rm)
}

calc_z_score(df$a, 1)
```

```
## [1] 0.2768789
```

```
calc_z_score(df$a, 2)
```

```
## [1] -1.377454
```

```
calc_z_score(df$a, 3)
```

```
## [1] -0.5607078
```

Hmm . . . Maybe we can vectorize

```
calc_z_score <- function(x) {
  (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
}

calc_z_score(df$a)
```

```
##  [1]  0.2768789 -1.3774542 -0.5607078 -0.6076392  0.930
##  [7]  0.6743792  1.1954284  1.3185971 -1.1933419
```

Example of modifying an object: z-score loop

Our Task: write loop that replaces variables with z-scores of those variables

- **sequence**
  - data frame `df` has 4 variables and all are quantitative
  - operate on each column
    - `for (i in seq_along(df))`
- **body**
  - Take z-score function:
    - `calc_z_score(x)`
  - Replace `x` with `df[[i]]`:
    - `calc_z_score(df[[i]])`
  - Assign or replace each column:
    - overwrite: `df[[i]] <- calc_z_score(df[[i]])`
    - make new object: `out_df[[i]] <- calc_z_score(df[[i]])`

Example of modifying an object: z-score loop

Creating an object to capture output is a bit more involved with data frames.

```
# we can use the old object
out_df <- df

# OR we can make an empty dataframe
out_df <- as_tibble(matrix(rep(NA, 40), ncol = ncol(df)))
names(out_df) <- names(df)

head(out_df)
```

```
## # A tibble: 6 x 4
##   a     b     c     d    
##   <lgl> <lgl> <lgl> <lgl>
## 1 NA    NA    NA    NA   
## 2 NA    NA    NA    NA   
## 3 NA    NA    NA    NA   
## 4 NA    NA    NA    NA   
## 5 NA    NA    NA    NA   
## 6 NA    NA    NA    NA   
```

The whole loop

```
out_df <- as_tibble(matrix(rep(0, 40), ncol = ncol(df)))
names(out_df) <- names(df)

for (i in seq_along(df)) {
  # modify values
  out_df[[i]] <- calc_z_score(df[[i]])
}
str(out_df)
```

```
## tibble [10 x 4] (S3: tbl_df/tbl/data.frame)
##  $ a: num [1:10] 0.277 -1.377 -0.561 -0.608 0.93 ...
##  $ b: num [1:10] 0.426 -1.106 -1.714 1.078 -0.683 ...
##  $ c: num [1:10] 0.3229 1.8655 -0.8158 -0.905 -0.0344 ..
##  $ d: num [1:10] 0.141 -1.013 1.236 0.449 -0.157 ...
```

Modifying an object in place

We can also change df in place!

- Useful if df is very large (relative to your RAM)
- Theoretically, can do this in other loops we've seen, but dangerous!
  - We might change underlying data that we operate on in the next iteration!

```
for (i in seq_along(df)) {
  # modify values
  df[[i]] <- calc_z_score(df[[i]])
}
str(df)
```

```
## tibble [10 x 4] (S3: tbl_df/tbl/data.frame)
##  $ a: num [1:10] 0.277 -1.377 -0.561 -0.608 0.93 ...
##  $ b: num [1:10] 0.426 -1.106 -1.714 1.078 -0.683 ...
##  $ c: num [1:10] 0.3229 1.8655 -0.8158 -0.905 -0.0344 ...
##  $ d: num [1:10] 0.141 -1.013 1.236 0.449 -0.157 ...
```

map or apply

Many R coders prefer the `map()` family functions from `purrr` or base R `apply` family.

- See iteration in R for Data Science

```
# map(.x, .f)
map(df, calc_z_score)
# sapply(X, FUN, ..., simplify = TRUE)
sapply(df, calc_z_score, simplify = FALSE)
```

This says "apply" the function to the columns of the `df` or "map" the columns of `df` to the function `calc_z_score`.

Output is a list – here, a list of modified columns.

In action

```
# map_<output type>(.x, .f)
map(df, calc_z_score) %>% bind_cols() %>% head(4)
```

```
## # A tibble: 4 x 4
##        a      b      c      d
##    <dbl>  <dbl>  <dbl>  <dbl>
## 1  0.277  0.426  0.323  0.141
## 2 -1.38  -1.11   1.87  -1.01
## 3 -0.561 -1.71  -0.816  1.24
## 4 -0.608  1.08  -0.905  0.449
```

```
sapply(df, calc_z_score, simplify = FALSE) %>%
  bind_cols() %>% head(4)
```

```
## # A tibble: 4 x 4
##        a      b      c      d
##    <dbl>  <dbl>  <dbl>  <dbl>
## 1  0.277  0.426  0.323  0.141
## 2 -1.38  -1.11   1.87  -1.01
## 3 -0.561 -1.71  -0.816  1.24
## 4 -0.608  1.08  -0.905  0.449
```

map makes a list, but we usually want vectors or tibbles.

The map family has the form

- `map_<output type>(.x, .f)`

```
map(.x, .f) %>% as.integer()
map_int(.x, .f)

map(.x, .f) %>% as.character()
map_chr(.x, .f)
```

```
# dfc = data.frame columns
# map(df, calc_z_score) %>% bind_cols()
map_dfc(df, calc_z_score) %>% head()
```

```
## # A tibble: 6 x 4
##         a      b       c      d
##     <dbl>  <dbl>   <dbl>  <dbl>
## 1  0.277  0.426  0.323   0.141
## 2 -1.38  -1.11   1.87   -1.01
```

map functions can feel like magic

- imagine writing a loop
  - the sequence is `.x`
  - the body is `.f`
    - often you'll write new functions or even use "anonymous functions".

```
sample_means <- map_dbl(c(1, 10, 100, 1000),
                        function(x) mean(rnorm(x)))
sample_means
```

```
## [1]  1.54081498  0.57023416 -0.05061468 -0.03883378
```

- There's much less overhead.

sapply can feel like magic

- imagine writing a loop
  - the sequence is `X`
  - the body is `FUN`
    - often you'll write new functions or even use "anonymous functions".

```
sample_means <- sapply(c(1, 10, 100, 1000),
                        function(x) mean(rnorm(x)))
sample_means
```

```
## [1] -1.46709849  0.03112029 -0.02484604  0.01790692
```

- There's much less overhead.
- By default `simplify = TRUE`, so `sapply` outputs a double vector.
  - This "simplification" can make for confusing code since you might be surprised by the output.

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)
  - Pre-allocate space for your output
  - 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 `cat()` or `print()`)

Next steps

Lab:

- *Today:* Learning Loops

I can write loops, but know when to vectorize

Final project:

- Deadline for guaranteed feedback September 24.
  - Recommend setting a personal deadline of the 21st.
  - Optional but worth trying!

Thank you!

Additional Material

Creating multiple plots with a loop

Another good use of a loop is to create multiple graphs easily. Let's use a loop to create 4 plots representing data from an exam containing 4 questions. Here are how the first few rows of the data look:

```
head(examscores)
```

```
## # A tibble: 6 x 4
##       a     b     c     d
##   <dbl> <dbl> <dbl> <dbl>
## 1  57.8  78.4  61.8  68.5
## 2  33.7  69.9  68.9  62.9
## 3  71.7  41.9  84.7  78.5
## 4 118.   32.0  58.5  88.2
## 5  38.7  40.5  76.6  65.8
## 6  62.6  60.3  84.6  79.7
```

Creating multiple plots with a loop

Let's loop over the columns and create a histogram of the data in each column:

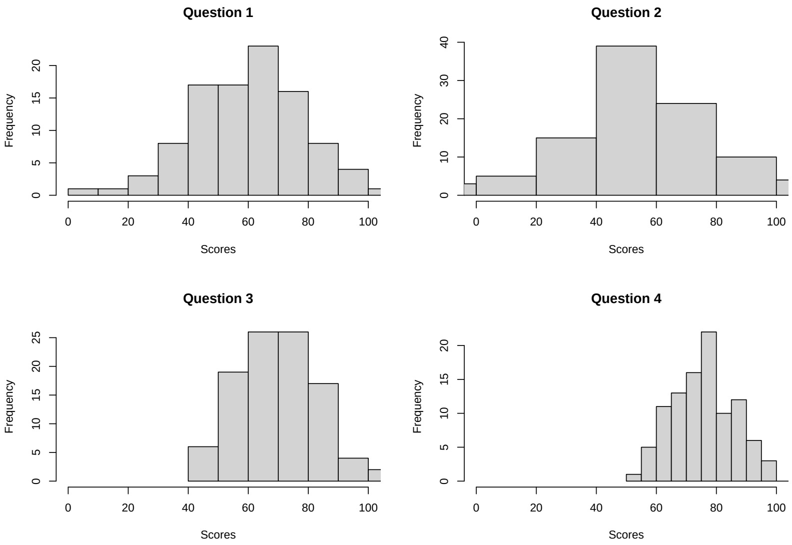
```r
# Set up a 2 x 2 plotting space
par(mfrow = c(2, 2))

# Create the loop.vector (sequence)
each.question <- 1:4

for (i in each.question) {

  # Plot histogram of each question
  hist(examscores[[i]],
       main = paste("Question", i),
       xlab = "Scores",
       xlim = c(0, 100))
}
```

Creating multiple plots with a loop



Try it yourself

We'll use `midwest` data and use a loop to create 4 plots representing data from `midwest` containing 4 key columns: `poptotal`, `percpovertyknown`, `percollege` and `percbelowpoverty`. Loop over these selected columns and create a histogram of the data in each column by completing the following code.

```
# Set up a 2 x 2 plotting space
par(mfrow = c(2, 2))

# Create the sequence
selected.column <- c("poptotal", "percpovertyknown",
                     "percollege", "percbelowpoverty")

for (...) {

  # Plot histogram of each column
  hist(...,
       main = paste(...),
       xlab = ...)
}
```

Try it yourself

You should get the histograms below: