CSSS508, Lecture 6

Loops

Michael Pearce
(based on slides from Chuck Lanfear)

May 3, 2022

Topics

Last time, we learned about,

1. Importing and exporting data
2. Cleaning and reshaping data
3. Dates and times

Today, we will cover,

1. Why Loops?
2. `for()` loops
3. `while()` loops

1. Why Loops?

Bad Repetition

If someone doesn't know better, they might find the means of variables in the `swiss` data by typing in a line of code for each column:

```
mean1 <- mean(swiss$Fertility)
mean2 <- mean(swiss$Agriculture)
mean3 <- mean(swissExamination)
mean4 <- mean(swiss$Fertility)
mean5 <- mean(swiss$Catholic)
mean5 <- mean(swiss$Infant.Mortality)
c(mean1, mean2 mean3, mean4, mean5, man6)
```

Can you spot the problems?

How upset would they be if the `swiss` data had 200 columns instead of 6?

Good Repetition

You will learn a better way to calculate column means today using loops!

```
means <- rep(NA, ncol(swiss))
for(i in 1:ncol(swiss)){
  means[i] <- mean(swiss[,i])
}
data.frame(Variable=names(swiss),Mean=means)
```

```
##              Variable Mean
## 1           Fertility 70.1
## 2         Agriculture 50.7
## 3         Examination 16.5
## 4           Education 11.0
## 5            Catholic 41.1
## 6 Infant.Mortality 19.9
```

Don't worry about the details yet!

Don't Repeat Yourself (DRY)!

The **DRY** idea: Computers are much better at doing the same thing over and over again than we are.

- Writing code to repeat tasks for us reduces the most common human coding mistakes.
- It also *substantially* reduces the time and effort involved in processing large volumes of data.
- Lastly, compact code is more readable and easier to troubleshoot.

2. `for()` Loops

The `for()` Loop

`for()` loops are the most general kind of *loop*, found in pretty much every programming language.

"**For** each of these values -- in order -- **do this**"

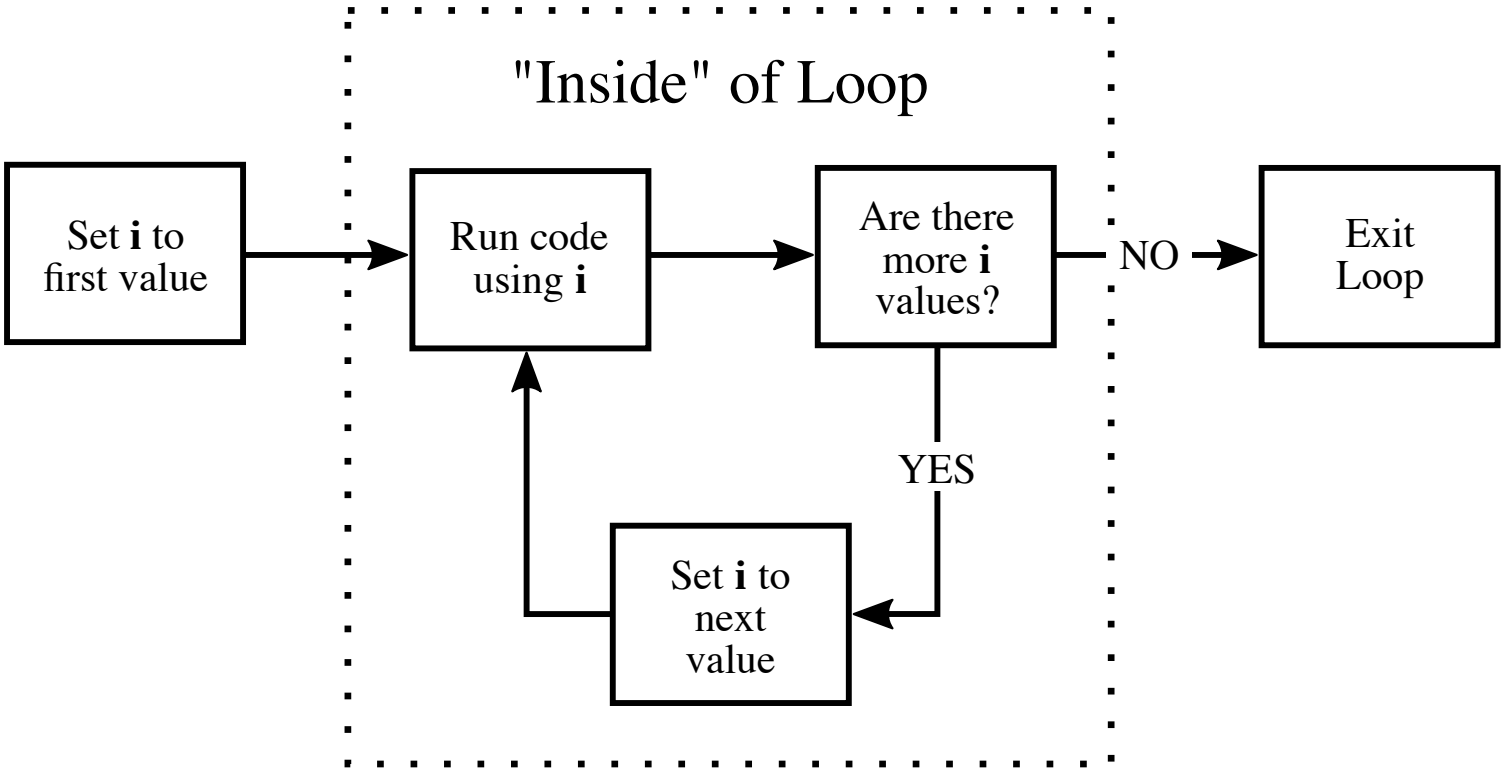
Given a set of values...

1. Set an index variable (often `i`) equal to the first value
2. Do something (perhaps depending on `i`)
3. Is there a next value?
    - *YES*: Update to next value, go back to 2.
    - *NO*: Exit loop

We are *looping* through values and repeating some actions.

`for()` Loop: Diagram

Given a set of values...



`for()` Loop: Example

```
for(i in 1:5) {
    # inside for, output won't show up without print()
    print(i^2)
}
```

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

Note this runs *5 separate print commands*, which is why each line starts with `[1]`.

These Do the Same Thing

```
for(i in 1:3) {
    print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
```

```
i <- 1
print(i^2)
i <- 2
print(i^2)
i <- 3
print(i^2)
```

```
## [1] 1
## [1] 4
## [1] 9
```

Iteration Conventions

- We call what happens in the loop for a particular value one **iteration**.
- Iterating over indices `1:n` is *very* common. `n` might be the length of a vector, the number of rows or columns in a matrix or data frame, or the length of a list.
- Common notation: `i` is the object that holds the current value inside the loop.
  - If loops are nested, you will often see `j` and `k` used for the inner loops.
  - This notation is similar to indexing in mathematical symbols (e.g $\sum_{i=1}^{n}$)
- Note `i` (and `j`,`k`, etc) are just normal objects. You can use any other names you want.
  - Ex: When iterating over rows and/or columns, I often use `row` and/or `col`!

Iterate Over Characters

What we iterate over doesn't have to be numbers `1:n` or numbers at all! You can also iterate over a character vector in R:

```
for(i in letters[1:3]) {
    print(i)
}
```

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

```
i # in R, this will exist outside of the loop!
```

```
## [1] "c"
```

Pre-Allocation

Usually in a `for()` loop, you aren't just printing output, but want to store results from calculations in each iteration somewhere.

To do that, figure out what you want to store, and **pre-allocate** an object of the right size as a placeholder (typically with missing values as placeholders).

Examples of what to pre-allocate based on what you store:

- Single numeric value per iteration:
  - `rep(NA, num_iter_iters)`
- Numeric vector per iteration:
  - `matrix(NA, nrow = num_of_iters, ncol = length_of_vector)`

Pre-Allocation: Numeric

```
iters <- 10 # Set number of interations
output <- rep(NA,iters) # Pre-allocate numeric vector
output
```

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

```
for(i in 1:iters) { # Run code below iters times
    output[i] <- (i-1)^2 + (i-2)^2
}
output # Display output
```

```
##  [1]   1   1   5  13  25  41  61  85 113 145
```

Steps:

1. Set a number of iterations
2. Pre-allocate a numeric vector of that length
3. Run ten iterations where the output is a mathematical function of each iteration number.

Pre-Allocation: Numeric Vector per Iteration Matrix

```
rownums <- 3
colnums <- 6
output <- matrix(NA,nrow=rownums,ncol=colnums)

for(i in 1:rownums){
  for(j in 1:colnums){
    output[i,j] <- i + j
  }
}
output
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2    3    4    5    6    7
## [2,]    3    4    5    6    7    8
## [3,]    4    5    6    7    8    9
```

Aside: If/Else Statements

To test a logical statement and then conditionally execute a set of actions, use `if()` and `else`. The structure is:

```
if(CONDITION){
  SOME CALCULATION
} else{
  A DIFFERENT CALCULATION
}
```

Warning! `else` needs to be on same line as the closing brace `}` of previous `if()`.

If/Else Simple Example

```
if(8  < 10){
  print("Less than 10!")
}else{
  print("Not less than 10!")
}
```

```
## [1] "Less than 10!"
```

More Complex If/Else

We can nest together multiple if/elses! if we wish:

```
i <- 13
if(i <= 10) {
  print("i is less than or equal to 10!")
} else if(i <= 14) {
  print("i is greater than 10, less than or equal to 14")
} else {
  print("i is greater than or equal to 15")
}
```

```
## [1] "i is greater than 10, less than or equal to 14"
```

Loops with If/Else Statements

Suppose we want to take the numbers between 1 and 5, and divide the evens by 2 and multiply the odds by 2. We could do that using a loop with if/else statements!

```
for(i in 1:5){
  if(i %% 2 == 0){ #check for even numbers
    print(i / 2)
  }else{
    print(i * 2)
  }
}
```

```
## [1] 2
## [1] 1
## [1] 6
## [1] 2
## [1] 10
```

Handling Special Cases

Aside from the previous toy example, `if()` statements are useful when you have to handle special cases.

`if()` statements can be used to make a loop ignore or fix problematic cases.

They are also useful for producing error messages, by generating a message *if* an input value is not what is expected.

3. `while()` Loops

`while()`

A lesser-used looping structure is the `while()` loop.

Rather than iterating over a predefined vector, the loop keeps going until some condition is no longer true.

Here is the structure:

```
while(COND IS MET){
  RUN CODE
}
```

If you're not careful, the while loop will run **forever!!**

Simple `while()` loop example:

```
x <- 0
while(x < 3){
  x <- x + 1
  print(x)
}
```

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

What happened in each iteration?

These Do the Same Thing

```
x <- 0
while(x < 3){
  x <- x + 1
  print(x)
}
```

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

```
x <- 0
x <- x+1
print(x)
x <- x+1
print(x)
x <- x+1
print(x)
```

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

More Complex Example

Let's see how many times we need to flip a coin to get 4 heads:

```
num_heads <- 0
num_flips <- 0

while(num_heads < 4) {
  # simulating a coin flip
  coin_flip <- rbinom(n = 1, size = 1, prob = 0.5)

  # keep track of heads
  if (coin_flip == 1) {
    num_heads <- num_heads + 1
  }

  # update number of coin flips
  num_flips <- num_flips + 1
}

num_flips # follows negative binomial distribution
```

```
## [1] 10
```

Summary

1. Why Loops?
    - To make our lives easier!
2. `for()` loops:
    - For iterating over a fixed number of items
3. `while()` loops:
    - For iterating until some condition is met

Let's take a 10 minute break, then return for some activities!

Activity 1

1. Create a `for` loop to calculate the maximum value of each variable in the `swiss` data. What are the maximum values of each variable?
2. Using your previous answer as a starting point, create a nested `for` loop to calculate the maximum value for each variable in the `swiss` data (outer loop), and then divide that maximum by 1, 2, and 4 (inner loop). Print the output after each step.
3. Using your previous answer as a starting point, write a loop that does the same calculations as before but stores the values in a matrix with `ncol(swiss)` columns and 3 rows. How will you "pre-allocate" space for the results?

Activity 1: My Answers!

Question 1:

```
for(i in 1:ncol(swiss)) {
  curr_max <- max(swiss[,i])
  print(curr_max)
}
```

```
## [1] 92.5
## [1] 89.7
## [1] 37
## [1] 53
## [1] 100
## [1] 26.6
```

Activity 1: My Answers!

Question 2:

```
for(i in 1:ncol(swiss)) {
  curr_max <- max(swiss[,i])
  for(j in c(1,2,4)){
    print(curr_max/j)
  }
}
```

```
## [1] 92.5
## [1] 46.2
## [1] 23.1
## [1] 89.7
## [1] 44.9
## [1] 22.4
## [1] 37
## [1] 18.5
## [1] 9.25
## [1] 53
## [1] 26.5
## [1] 13.2
## [1] 100
```

Activity 1: My Answers!

Question 3:

```
results <- matrix(NA, ncol=ncol(swiss),nrow=3)
for(i in 1:ncol(swiss)) {
  curr_max <- max(swiss[,i])
  for(j in 1:3){
    curr_divisor <- c(1,2,4)[j]
    results[j,i] <- curr_max/curr_divisor
  }
}
results
```

```
##      [,1] [,2]  [,3] [,4] [,5]  [,6]
## [1,] 92.5 89.7 37.00 53.0  100 26.60
## [2,] 46.2 44.9 18.50 26.5   50 13.30
## [3,] 23.1 22.4  9.25 13.2   25  6.65
```

Activity 2

1. Consider the vector `vec <- c(1,2,NA,3,NA)`. Write a `for` loop that includes an `if`/`else` function so that for each value `x` in `vec`, we print "Missing!" if `x` is NA, and `x^3` otherwise.

2. What will happen if I run the following loop:

```
x <- 1
while(x < 10){
  print(x + 1)
}
```

3. Write a `while()` loop that starts with `x <- 1` and doubles x each iteration, while `x < 100`. Print `x` after each iteration.

Activity 2: My Answers

1.

```
for(x in c(1,2,NA,3,NA)){
  if(is.na(x)){
    print("Missing!")
  } else{
    print(x^3)
  }
}
```

```
## [1] 1
## [1] 8
## [1] "Missing!"
## [1] 27
## [1] "Missing!"
```

Activity 2: My Answers

2. What will happen if I run the following loop:

```
x <- 1
while(x < 10){
  print(x + 1)
}
```

- **Answer:** The while loop will run forever printing `1`, since we are not updating `x`!!

Activity 2: My Answers

3. Write a `while()` loop that starts with `x <- 1` and doubles x each iteration, while `x < 100`. Print `x` after each iteration.

```
x <- 1
while(x <100){
  x <- x * 2
  print(x)
}
```

```
## [1] 2
## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
## [1] 128
```

Why does `x` reach 128?!

Homework

HW 6 will be posted on the website shortly! Remember that it is a continuation of HW 5!