CSSS508, Lecture 6

Loops

Michael Pearce
(based on slides from Chuck Lanfear)

November 3, 2022

UW CS&SS

Reminders!

- Mid-quarter feedback survey is on Canvas now!
- Homework 5 is due *now* (Key to be posted soon!)
- Homework 6 will be posted tonight (Includes HW 5 Key!)
- If you are worried about your grade, please come and talk to me!

  +*Reminder:* Class is pass/fail, 60% to pass!

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

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 <- numeric(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.

`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**"

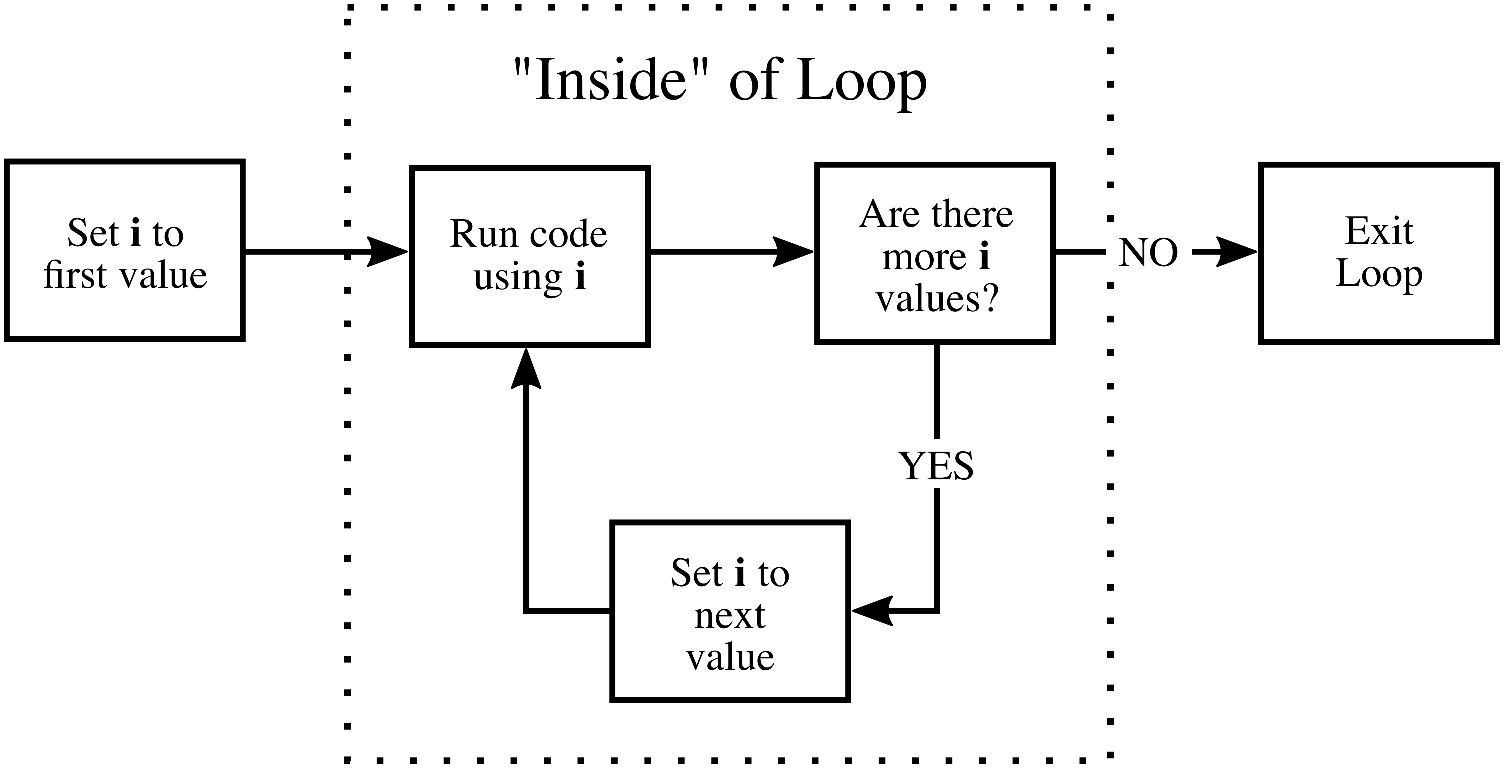
Given a set of values…

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

```r
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

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

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

```r
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:

```r
some_letters <- letters[4:6] # Vector of letters d,e,f
for(i in some_letters) {
    print(i)
}
```

```
## [1] "d"
## [1] "e"
## [1] "f"
```

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

```
## [1] "f"
```

`seq_along()` and Messages

`seq_along(x)` creates an integer vector equal to `1:length(x)`.

When you want to loop over something that isn't numeric but want to use a numeric index of where you are in the loop, `seq_along` is useful:

```r
some_letters <- letters[4:6]
for(a in seq_along(some_letters)) {
    print(paste0("Letter ", a, ": ", some_letters[a]))
}
```

```
## [1] "Letter 1: d"
## [1] "Letter 2: e"
## [1] "Letter 3: f"
```

```r
a # The object `a` contains the number of the last iteration
```

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

Activity!

Work in pairs on the following questions:

1. Suppose you want the maximum value of each variable in the `swiss` data. *Without* writing down any code, what are the (a) *indices* you will iterate over and (b) *computation* you will apply to each index?
2. Create the `for` loop and print out the results. What are the maximum values for each variable?
3. Do Question 2, but this time using the `seq_along()` function to specify your indices. Ensure you obtain the same result!

Activity! My Answers

1. What are the (a) *indices* and (b) *computation*?

   - **Answer:** Indices are 1,2,3,4,5, and 6, as there are 6 columns in `swiss`. Computation is `max( )`, which will be applied to each column.

2. Create the `for` loop and print out the results. What are the maximum values for each variable?

```
for(i in 1:6){
  maximum <- max(swiss[,i])
  print(maximum)
}
```

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

Activity! My Answers

3.Do Question 2, but this time using the `seq_along()` function to specify your indices. Ensure you obtain the same result!

```
for(i in seq_along(swiss)){
  maximum <- max(swiss[,i])
  print(maximum)
}
```

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

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: `numeric(num_of_iters)`
- Numeric vector per iteration: `matrix(NA, nrow = num_of_iters, ncol = length_of_vector)`
- Single character value per iteration: `character(num_of_iters)`
- Single true/false value per iteration: `logical(num_of_iters)`

Pre-Allocation: Numeric

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

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

```r
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-allocated a numeric vector of that length
3. Ran ten iterations where the output is a mathematical function of each iteration number.

Pre-Allocation: Numeric Vector per Iteration Matrix

```r
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
```

Think-Pair-Share

1. Suppose you want to calculate the maximum value for each variable in `swiss` and then divide the maximum by 1, 2, and 4 (separate operations). How could this be done using nested `for` loops?

2. How could one "pre-allocate" space for the calculations?

3. Write a nested `for` loop to answer Question 1 and print the results.

Think-Pair-Share: My Answers

1. Suppose you want to calculate the maximum value for each variable in `swiss` and then divide the maximum by 1, 2, and 4 (separate operations). How could this be done using nested `for` loops?

    - **Answer:** The first loop could calculate the maximum. Within the loop, you could have a loop to divide by the three numbers (1, 2, and 4).

2. How could one "pre-allocate" space for the calculations?

    - **Answer:** Using a matrix with 6 rows (one for each variable) and 3 columns (one for each value).

Think-Pair-Share: My Answers

1. Write a nested `for` loop to answer Question 1 and print the results.

```
output <- matrix(NA,nrow=6,ncol=3)
divisors <- c(1,2,4)
for(i in 1:6){
  maximum <- max(swiss[,i])
  for(j in 1:3){
    value <- divisors[j]
    output[i,j] <- maximum/value
  }
}
output
```

```
##        [,1] [,2]  [,3]
## [1,]   92.5 46.2 23.12
## [2,]   89.7 44.9 22.43
## [3,]   37.0 18.5  9.25
## [4,]   53.0 26.5 13.25
## [5,]  100.0 50.0 25.00
## [6,]   26.6 13.3  6.65
```

Aside: If/Else Statements

You've seen `ifelse()` before for logical checks on a whole vector.

For checking whether a *single* logical statement holds and then conditionally executing 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

```r
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!

```r
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
```

Activity!

1. What function checks for if values are `NA`?
2. Consider the vector `vec <- c(1,2,NA,3,NA)`. For each value `x`, print "Missing!" is the value is NA, and `x^3` otherwise.

Activity! My Answers

1. What function checks for if values are `NA`?

   - **Answer:** `is.na()`

2. Consider the vector `vec <- c(1,2,NA,3,NA)`. For each value `x`, print "Missing!" is the value is NA, and `x^3` otherwise.

```r
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!"
```

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.

`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:

```r
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

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

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

```r
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:

```r
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] 6
```

Activity!

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

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

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

Activity! My Answers

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

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

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

Activity! My Answers

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

```r
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!