Week 4 Module 1: `for` Loops

CSCI E-5a: Introduction to R

Let's clear the global computing environment:

```
rm( list = ls() )
```

Module Overview and Learning Objectives

Hello! And welcome to Module 1: For Loops.

In this module, we'll learn a powerful method for automating repetitive calculations.

- In section 1, we'll learn about `for` loops, and construct some simple examples.
- In section 2, we'll explore the concept of an accumulator variable.
- In section 3, we'll learn how to iterate over indices, which provides much more flexibility.

When you've completed this module, you should be able to:

- Construct a `for` loop to perform iteration.
- Implement and use accumulator variables.
- Iterate over the indices of a vector.

There is one new built-in R functions in this module:

- `for()`

Allright! Let's get started by learning about the basics of `for` loops.

Section 1: Basic `for` Loops

> **Main Idea:** *We can automate repetitive activities by using a `for()` loop*

In this section, we'll learn about `for` loops, and construct some simple examples.

So far, we've seen many instances of repetition in the calculations that we've done.

In Lecture 2, we performed these repetitive calculations by hand.

In Lecture 3, we saw how to automate some of these procedures by using vectorized operations.

Now we're going to learn a new programming construct called a *for loop*, and this will enable us to automate many repetitive processes.

Some students find `for` loops to be puzzling, so we're going to start right from the beginning and proceed step-by-step.

Every `for` loop consists of three components:

- The first component is a *loop variable*.
- The second component is a *loop vector*.
- The third component is the *loop body*, which is just a sequence of R statements.

The loop body is enclosed in a pair of matching curly braces.

When the `for` loop starts, the loop variable is assigned the first value in the loop vector.

Then each statement in the loop body is executed, using the current value of the loop variable.

When the last statement of the loop body has completed, R goes back to the top of the loop, and assigns the second value in the loop vector to the loop variable.

Then all the statements in the loop body are executed using the current value of the loop variable.

When the last statement of the loop body has completed, R returns to the top of the loop and now assigns the third value in the loop vector to the loop variable.

R continues this process, assigning each successive value in the loop vector to the loop variable and then executing the statements in the body of the loop.

When the last value in the loop vector has been assigned to the loop variable and all the statements in the loop body have executed, then the loop comes to an end and R continues to the next statement.

Let's see a very simple example of this process.

First, we'll make a vector:

```r
animal.vector <- c( "mouse", "dog", "bison" )
```

Now we'll construct a `for` loop to print out each animal name in the vector.

We have to use the correct syntax for the `for()` loop:

- First, we have the keyword `for`, followed by an opening parenthesis.
- Next, we have the loop variable.
- Next is the keyword `in`.
- Then there is the loop vector, followed by a closing parenthesis.
- Finally, there is the loop body, which starts with an opening curly brace, followed by a sequence of R statements, and then finally a closing curly brace.

Can you predict what this code will print out before you run it?

```r
for( current.animal in animal.vector ) {

    cat( "Current animal:", current.animal, "\n" )

}
```

```
## Current animal: mouse
## Current animal: dog
## Current animal: bison
```

In this example, `current.animal` is the loop variable.

The vector `animal.vector` is the loop vector.

Don't forget the keyword `in`!

The loop body is contained between the two curly braces, and in this example it consists of only one statement, which is a `cat()` statement.

At the start of the `for` loop, R assigns the first value in the loop vector `animal.vector` to the loop variable `current.animal`.

Since the first element of `animal.vector` is "mouse", the loop variable `current.animal` is assigned this value.

R then executes every statement in the loop body.

In this case, that means that R executes the `cat()` statement with the current value of `current.animal`, which is "mouse".

Once R has printed out the `cat()` statement, the loop body has finished, so R returns to the top of the loop and assigns the second value of the loop vector `animal.vector` to the loop variable.

Now the loop variable `current.animal` has the value "dog".

Then the body of the loop is executed, and again the `cat()` statement is executed.

Finally, once the `cat()` statement has finished, R returns to the top of the loop and assigns the third and last value of `animal.vector` to the loop variable.

Now 'current.animal' has the value "bison".

R then executes the loop body, printing out the `cat()` statement.

At this point, there are no more values to assign to the loop variable, so R exits the loop.

By the way, you might have noticed that strange little string consisting of a backslash character and the letter 'n' in the `cat()` statement.

What's that?

The strange string represents a *newline*.

That is, R will stop printing on the current line, and start at the beginning of the next line.

If we don't include this newline string, then the next time we call the `cat()` function it will continue printing on the same line, which is generally not what we want.

Go back to the `for` loop, remove the newline, run the code and see what happens.

Let's try this again

Remember, once we've assigned a value to the loop variable, it's just a regular variable like any other variable, and we can do anything with the loop variable that we can do with other variables.

Here's the code, but before I run it, see if you can figure out in advance what it prints out:

```
numeric.vector <- 1:3

for( current.number in numeric.vector ) {
    double.current.number <- 2 * current.number

    cat(
        "double.current.number:",
        double.current.number,
        "\n"
    )
}
```

```
## double.current.number: 2
## double.current.number: 4
## double.current.number: 6
```

So that's how to construct simple `for()` loops.

Now let's see how to keep a running total during a `for()` loop.

Exercise 1.1: Even numbers

Write a `for()` loop that prints out the first 10 even numbers. (Hint: use the `seq()` function to create the loop vector).

Solution

Section 2: Accumulator Variables

> **Main Idea:** *We can use an accumulator variable to keep a running total through a `for()` loop*

In this section, we'll explore the concept of an accumulator variable.

An *accumulator* variable is just a variable that keeps a running total over all the iterations of a `for()` loop.

We can update the accumulator variable as we run the iterations of the `for` loop.

The code for updating a variable might look a little strange at first.

Let's first make a variable, and assign it the value 0.

```
accumulator.variable <- 0
```

Now we'll update this variable:

```
accumulator.variable <-
    accumulator.variable + 1

accumulator.variable
```

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

Hey – the `accumulator.variable` is on *both* the left-hand and right-hand side of the assignment statement.

How is that even possible?

Let's think about how the assignment operation works.

First, R evaluates all the terms on the right-hand side of the assignment statement.

At this time, `accumulator.variable` has the value 0, because we just initialized it with that value.

Thus, the right-hand side evaluates to 0 + 1, or just 1.

The value 1 is then assigned to `accumulator.variable`.

So at the end of the statement, the `accumulator.variable` now has the value 1.

```
accumulator.variable
```

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

Thus, we've "updated" this variable.

What happens if we run this code again?

```
accumulator.variable <-
    accumulator.variable + 1
```

At the start of this statement, the `accumulator.variable` has the value 1.

Thus, the right-hand side will evaluate to 1 + 1, or 2.

This is then assigned to the `accumulator.variable`.

Thus, at the end of this statement, the `accumulator.variable` has the value 2.

```
accumulator.variable
```

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

So we've "updated" the `accumulator.variable` again.

We can use this method along with a `for()` loop to count things.

Let's count the number of elements in the `animal.vector` variable.

Since we're using the accumulator variable to count the number of elements, I'm just going to use the name `counter` for this variable.

At the beginning, we haven't counted anything yet, so the value of the `counter` variable should be 0:

```
counter <- 0
```

Now let's count the number of elements in `animal.vector`:

```
counter <- 0

for( current.animal in animal.vector ) {

    counter <- counter + 1

}

counter
```

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

So we counted 3 elements in the `animal.vector`, which is correct.

You might have noticed that we didn't actually do anything with the loop variable.

Instead, we just used the looping process to control how many times we updated the `counter` variable.

That's fine – we're not required to use the loop variable inside the loop body, although we usually do.

We can add up all the values in a vector using a similar approach.

Let's work with the first 5 positive integers:

```
numeric.vector <- 1:5
```

I'm going to call the accumulator variable `running.total`, because we're trying to calculate the total or sum of the values in the `numeric.vector`.

Now we can add these numbers together using the accumulator variable:

```
running.total <- 0

for( current.number in numeric.vector ) {

    running.total <- running.total + current.number
}

running.total
```

```
## [1] 15
```

Here's a more complicated example.

In this example, we'll calculate the total sum of the values of a vector, but we'll also print out a series of messages as we perform the computation.

I'm going to call the accumulator variable `running.total`, because we're interested in the intermediate values of this variable as we're doing the calculation.

I'm also going to use another special printing character, which is indicated by a backslash followed by a 't'; this will produce a tab stop.

```
running.total <- 0

numeric.vector <- 1:3

for( current.number in numeric.vector ) {

    cat(
        "current.number:",
        current.number,
        "\n"
    )

    cat(
        "\trunning.total before assignment:",
        running.total,
        "\n"
    )

    running.total <-
        running.total + current.number

    cat(
        "\trunning.total after assignment:",
        running.total,
        "\n\n"
    )
}
```

```
## current.number: 1
##  running.total before assignment: 0
##  running.total after assignment: 1
##
## current.number: 2
##  running.total before assignment: 1
##  running.total after assignment: 3
##
## current.number: 3
##  running.total before assignment: 3
##  running.total after assignment: 6
```

Now we're going to do a little thought experiment.

We're going to run the exact same code as before (I actually copied and pasted the code chunk) but instead of a vector of the numbers 1, 2, and 3, we'll use a vector with the numbers 2, 7, and -5.

Think about the code, and try to predict what the output will be *before* you run the code chunk.

```
running.total <- 0

numeric.vector <- c(2, 7, -5)

for( current.number in numeric.vector ) {

    cat(
        "current.number:",
        current.number,
        "\n"
    )

    cat(
        "\trunning.total before assignment:",
        running.total,
        "\n"
    )

    running.total <-
        running.total + current.number
    cat(
        "\trunning.total after assignment:",
        running.total,
        "\n\n"
    )
}
```

```
## current.number: 2
##  running.total before assignment: 0
##  running.total after assignment: 2
##
## current.number: 7
##  running.total before assignment: 2
##  running.total after assignment: 9
##
## current.number: -5
##  running.total before assignment: 9
##  running.total after assignment: 4
```

So that's how we can keep a running total by using an accumulator variable.

Now let's see how to iterate over the indices of a vector.

Exercise 1.2: Even numbers

Write a `for` loop that prints out the total sum of the first 10 even numbers. For each iteration, print out the value of the current running total.

Solution

Section 3: Looping over Indices

> **Main Idea:** *We can iterate over the indices of a vector*

In this section, we'll learn how to iterate over indices, which provides much more flexibility.

So far, all of our iterations have involved iterating over the elements of the vector itself.

There's another approach that's a little more indirect, but can be very flexible.

In this approach, we don't iterate over the vector, but instead iterate over the indices of the vector.

Remember, "indices" is just the plural of the word "index".

In this approach, we iterate over a sequence of numbers, and then use positive integer indexing to obtain the actual elements of the vector.

Let's use this approach to print out the values in `animal.vector`:

```
animal.vector <- c( "mouse", "dog", "bison" )

index.vector <- 1:length( animal.vector )

for( current.index in index.vector ) {
    current.animal <- animal.vector[ current.index ]

    cat( "Current animal:", current.animal, "\n" )
}
```

```
## Current animal: mouse
## Current animal: dog
## Current animal: bison
```

One advantage of iterating over the indices is that we can print out an "enumeration" of the elements of the vector.

That just means that we number the elements.

```
animal.vector <- c( "mouse", "dog", "bison" )

index.vector <- 1:length( animal.vector )

for( current.index in index.vector ) {

    current.animal <- animal.vector[ current.index ]

    cat(
        current.index,
        ". Current animal:",
        current.animal,
        "\n"
    )
}
```

```
## 1 . Current animal: mouse
## 2 . Current animal: dog
## 3 . Current animal: bison
```

You might have noticed that the printout is correct, but some of the formatting is a little weird.

In particular, there is a space between the number and the period.

That's because `cat()` by default separates each element using a space character.

We can control this by using the `sep` option.

If we set `sep = ""`, then `cat()` will not place a separator between elements.

```
animal.vector <- c( "mouse", "dog", "bison" )

index.vector <- 1:length( animal.vector )

for( current.index in index.vector ) {

    current.animal <- animal.vector[ current.index ]

    cat(
        current.index,
        ". Current animal:",
        current.animal,
        "\n",
        sep = ""
    )
}
```

```
## 1. Current animal:mouse
## 2. Current animal:dog
## 3. Current animal:bison
```

That look better, except that now we need a space after the colon:

```
animal.vector <- c( "mouse", "dog", "bison" )

index.vector <- 1:length( animal.vector )

for( current.index in index.vector ) {

    current.animal <- animal.vector[ current.index ]

    cat(
        current.index,
        ". Current animal: ",
        current.animal,
        "\n",
        sep = ""
    )
}
```

```
## 1. Current animal: mouse
## 2. Current animal: dog
## 3. Current animal: bison
```

By the way, notice how we constructed the `index.vector`:

```
index.vector <-
    1:length(animal.vector)

index.vector
```

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

Another way to do this is to use the `seq_along()` function:

```
seq_along( animal.vector )
```

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

The seq_along() function has some technical features that make it slightly preferable to constructing the sequence with the length() function and the colon operator method, but for our purposes there isn't a difference.

The seq_along() function is a common construct that's used by many R programmers, so it's a good idea to make sure that you understand what it actually does.

For our course, I prefer to use the approach with the colon operator, because that makes it very clear what's going on, whereas with seq_along() if you don't understand the underlying process then the whole thing seems mysterious.

By the way, notice that the function name "seq_along" doesn't conform to the standard naming conventions of R.

This is because R is an evolved language, not a designed language.

Whoever originally implemented seq_along() choose this name, and now it's too late to change it and we're stuck with it.

Iterating over an index variable like this is very useful when we want to iterate over two or more vectors at once.

Suppose each animal has a name:

Animal	Name
Mouse	"Ashley"
Dog	"Bob"
Bison	"Taylor"

Let's print out a report, stating the animal and its name.

```
animal.vector <-
    c( "Mouse", "Dog", "Bison" )

name.vector <-
    c( "Ashley", "Bob", "Taylor" )

for( index in seq_along( animal.vector ) ) {

    cat(
        index,
        ". Animal: ",
        animal.vector[ index ],
        ", Name: ",
        name.vector[ index ],
        "\n",
        sep = ""
    )
}
```

```
## 1. Animal: Mouse, Name: Ashley
## 2. Animal: Dog, Name: Bob
## 3. Animal: Bison, Name: Taylor
```

So that's how to iterate over the indices of a vector.

Now let's review what we've learned in this module.

Exercise 1.3: Sales report

Suppose a store has three sales:

Price per item	Number of items
10.99	4
3.99	12
8.99	7

Create two vectors to represent the Price per item column and the Number of items column. Then write a `for` loop that prints out the price per item and number of items for each transaction, as well as the total for that transaction. Finally, when you've finished, report the total sales for all three transactions.

Solution

Module Review

In this module, we learned a powerful method for automating repetitive calculations.

- In section 1, we learned about `for` loops, and construct some simple examples.
- In section 2, we explored the concept of an accumulator variable.
- In section 3, we learned how to iterate over indices, which provides much more flexibility.

Not that you've completed this module, you should be able to:

- Construct a `for` loop to perform iteration.
- Implement and use accumulator variables.
- Iterate over the indices of a vector.

There is one new built-in R functions in this module:

- `for()`

All right! That's it for Module 1: For Loops.

Now let's move on to Module 2: Iterate vs. Vectorize

Solutions to the Exercises

Exercise 1.1: Even numbers

Write a `for` loop that prints out the first 10 even numbers. (Hint: use the `seq()` function to create the loop vector).

Solution

First, to construct the vector, we can use the `seq()` function:

```
even.number.vector <-
    seq( from = 2, by = 2, length.out = 10)

even.number.vector
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

We could also use some vectorized trickery:

```
even.number.vector <-
    2 * 1:10

even.number.vector
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

Now we can print out the even numbers:

```
for( current.even.number in even.number.vector ) {

    cat(
        "Current even number:",
        current.even.number,
        "\n"
    )

}
```

```
## Current even number: 2
## Current even number: 4
## Current even number: 6
## Current even number: 8
## Current even number: 10
## Current even number: 12
## Current even number: 14
## Current even number: 16
## Current even number: 18
## Current even number: 20
```

Exercise 1.2: Even numbers

Write a `for` loop that prints out the total sum of the first 10 even numbers. For each iteration, print out the value of the current running total.

Solution

```
running.total <- 0

even.number.vector <-
    seq( from = 2, by = 2, length.out = 10)
```

```
for( current.even.number in even.number.vector ) {

    running.total <-
        running.total + current.even.number

    cat(
        "\tRunning.total:",
        running.total,
        "\n\n"
    )
}
```

```
##  Running.total: 2
##
##  Running.total: 6
##
##  Running.total: 12
##
##  Running.total: 20
##
##  Running.total: 30
##
##  Running.total: 42
##
##  Running.total: 56
##
##  Running.total: 72
##
##  Running.total: 90
##
##  Running.total: 110
```

Exercise 1.3: Sales report

Suppose a store has three sales:

Price per item	Number of items
10.99	4
3.99	12
8.99	7

Create two vectors to represent the Price per item column and the Number of items column. Then write a `for` loop that prints out the price per item and number of items for each transaction, as well as the total for that transaction. Finally, when you've finished, report the total sales for all three transactions.

Solution

```
price.per.item.vector <-
    c( 10.99, 3.99, 8.99 )

number.of.items.vector <-
```

```
    c( 4, 12, 7 )

index.vector <-
    1:length( price.per.item.vector )

total.sales.amount <- 0

for( index in index.vector ) {

    cat(
        "Transaction:",
        index,
        "\n"
    )

    cat(
        "\tPrice per item:",
        price.per.item.vector[ index ],
        "\n"
    )

    cat(
        "\tNumber of items:",
        number.of.items.vector[ index ],
        "\n"
    )

    cat(
        "\tSales amount:",
        price.per.item.vector[ index ] *
            number.of.items.vector[ index ],
        "\n"
    )

    cat( "\n" )

    total.sales.amount <-
        total.sales.amount +
        price.per.item.vector[ index ] *
            number.of.items.vector[ index ]
}
```

```
## Transaction: 1
##  Price per item: 10.99
##  Number of items: 4
##  Sales amount: 43.96
##
## Transaction: 2
##  Price per item: 3.99
##  Number of items: 12
##  Sales amount: 47.88
##
## Transaction: 3
##  Price per item: 8.99
##  Number of items: 7
##  Sales amount: 62.93
```

```r
cat(
    "Total sales amount:",
    total.sales.amount
)
```

```
## Total sales amount: 154.77
```