CSCE 120: Learning To Code

Module 8.0: Processing Data I
Introduction to Loops

This module is designed to get you started working with loops which are control structures that allow blocks of code to be repeatedly executed multiple times.

Introduction

Computers are really good at automation. A key aspect of automation is that we be able to repeat a process over and over on different pieces of data until some condition is met. For example, if we have a collection of numbers and we want to find their sum we would *iterate* over each number, adding it to a total, until we have examined every number. Another example may include sending an email message to each student in a course. To automate the process, we could iterate over each student record and *for each* student we would generate and send the email.

Automated repetition is where *loops* come in handy. Computers are perfectly suited for performing such repetitive tasks. We can write a single block of code that performs some action or processes a single piece of data, then we can write a loop around that block of code to execute it a number of times.

Loops provide a much better alternative than repeating (cut-paste-cut-paste) the same code over and over with different variables. Indeed, we wouldn't even do this in real life. Suppose that you took a 100 mile trip. How would you describe it? Likely, you wouldn't say, "I drove a mile, then I drove a mile, then I drove a mile, ..." repeated 100 times. Instead, you would simply state "I drove 100 miles" or maybe even, "I drove until I reached my destination."

Loops allow us to write concise, repeatable code that can be applied to each element in a collection or perform a task over and over again until some condition is met. When writing a loop, there are three essential components:

- An *initialization* statement that specifies how the loop begins

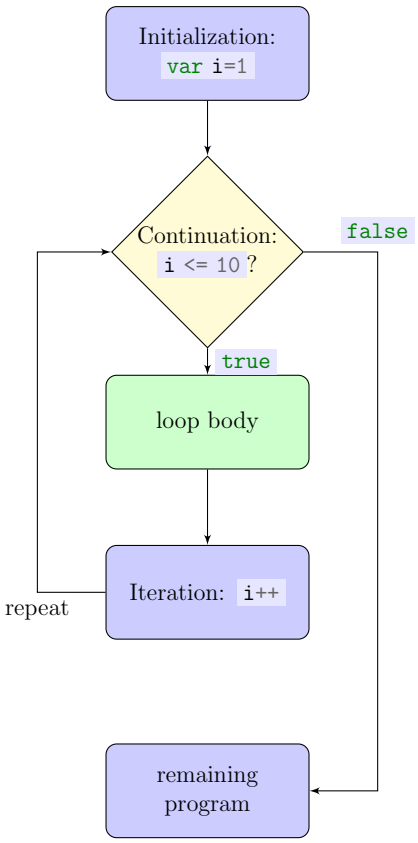

Figure 1: A Typical Loop Flow Chart

- A *continuation* (or *termination*) condition that specifies whether the loop should continue to execute or terminate
- An *iteration* statement that makes progress toward the termination condition

The initialization statement is executed before the loop begins and serves as a way to set the loop up. Typically, the initialization statement involves setting the initial value of some variable.

The continuation statement is a logical statement (that evaluates to *true* or *false*) that specifies if the loop should continue (if the value is *true*) or should terminate (if the value is *false*). Upon termination, code returns to a sequential control flow and the program continues.

The iteration statement is intended to update the state of a program to make progress toward the termination condition. If we didn't make such progress, the loop would continue on forever as the termination condition would never be satisfied. This is known as an *infinite loop* , and results in a program that never terminates.

As a simple example, consider the following outline.

1. Initialize the value of a variable i to 1...

2. While the value of i is less than or equal to 10 . . . (continuation condition). . .
3. Perform some action (this is sometimes referred to as the *loop body*)
4. Iterate the variable i by adding one to its value
5. Return to step 2 to determine if we should stop or continue

The process outlined above is depicted as a *flow chart* in Figure 1. It specifies that some action is to be performed once for each value: $i = 1, i = 2, \ldots, i = 10$, after which the loop terminates and the normal *flow of control* continues with the rest of the program. Overall, the loop executes a total of 10 times. Prior to each of the 10 executions, the value of i is checked; as it is less than or equal to 10, the action is performed. At the end of each of the 10 iterations, the variable i is incremented by 1 and the termination condition is checked again, repeating the process.

There are several different types of loops that vary in syntax and style but they all have the same three basic components.

For Loops

A for-loop allows you to specify the three components on the same line. An implementation of the pseudocode above in JavaScript would look something like the following.

```
1 for(var i=1; i<=10; i++) {
2   console.log(i);
3 }
```

This snippet of code will print the values 1, 2, . . . , 10 to the console.

Take note of the syntax:

- We use the keyword `for` along with parenthesis enclosing our three conditions
- The initialization statement, `var i=1;` declares a variable and sets it to one
- The continuation condition, `i<=10;` will continue the loop until the variable `i` exceeds 10
- The iteration statement, `i++` increments the variable `i` by one
- The code inside the opening/closing curly brackets is executed for each iteration of the loop

Iterating Over Array Elements

Another general use case for loops is to iterate over elements in an array. Recall that the elements in an array are referenced by an *index* which starts at zero. A for-loop

can be used to iterate over the elements in an array by defining an index variable and incrementing it until it exceeds the size of the array. Recall that the size of an array can be determined using the `.length` property. A full example:

```
var myNumbers = [2, 3, 10, 5, 2, 19, 12];
var sum = 0;
for(var i=0; i<myNumbers.length; i++) {
  sum = sum + myNumbers[i];
}
console.log("Total: " + sum); //prints "Total: 53"
```

The code above *iterates* over the elements in the array `myNumbers`. It uses an index variable, `i` which is initialized to zero, corresponding to the index of the first element in the array. Each iteration increments `i` by one, advancing the index to the next element in the array. The loop stops short of `myNumbers.length` (which has a value of 7) by using a strictly-less-than comparison operator. This is because the last element has an index that is one less than the length (the element 12 is at index 6).

Inside the loop, the i-th element is accessed and added to the `sum` variable. After the loop terminates, the total is logged to the console.

While Loops

Another type of loop is the while loop. The syntax is similar, but the location of the three components is different.

```
var i = 1;
while(i<=10) {
  console.log(i);
  i++;
}
```

In this example, the initialization statement comes *before* the loop; the continuation condition is still attached to the loop's keyword (which is now `while`) but the iteration statement comes at the end of the loop.

In general, a while loop is used if the number of iterations is not known (that is, the loop should continue to execute until some condition that does not involve a counter variable is satisfied). However, any while loop can be rewritten as a for-loop and vice versa.

Nested Loops

Within a loop, you can place any code you want to be executed each time the loop iterates. This *includes* conditionals and even *other loops*. When you have a loop within

a loop, they are called *nested* loops. An example:

```
for(var i=0; i<5; i++) {
  for(var j=0; j<3; j++) {
    console.log("i, j = " + i + ", " + j);
  }
}
```

In this example, the "outer" for loop will execute 5 times with i running from 0 up to 4. On *each* iteration of this loop, the "inner" for loop will fully execute 3 times (j running from 0 to 2). Thus, in total, these nested loops will run $5 \times 3 = 15$ times. When using nested loops, take care that the variables and conditions for each loop are separate (you cannot reuse the counter variable i in the inner for loop for example). The output of the above code would look something like the following.

```
i, j = 0, 0
i, j = 0, 1
i, j = 0, 2
i, j = 1, 0
i, j = 1, 1
i, j = 1, 2
i, j = 2, 0
i, j = 2, 1
i, j = 2, 2
i, j = 3, 0
i, j = 3, 1
i, j = 3, 2
i, j = 4, 0
i, j = 4, 1
i, j = 4, 2
```

jQuery's Each Loop

jQuery allows you to process elements in an array or the result of a selector query by using a special `each()` function. The syntax and approach to using this function is fundamentally different than directly using a for or while loop. The `each()` function takes, as an argument, *another function*. This is known as a *callback* function.

Essentially what we are passing the function is an array of elements `arr` and a function `f()` that will be *applied* to each element in that function. To think of it another way: we could write a simple for-loop that iterated over each element `e` in the array and then passed each individual element to the function, so `f(e)`. This alternative way of writing a loop allows us to write code in a more *functional* programming way by just worrying about the actual function.

There are two versions of the `each()` function. The first one can be applied to any array type. An example:

```
1 var arr = [10, 20, 30, 40];
2 $.each(arr, function(index, value) {
3   console.log("index " + index + " has the value " + value);
4 });
```

In the example above, the `$.each()` function takes two arguments: the first is the array and the second is a function that we want applied to each element in the array. The callback function itself is defined to take two arguments, an `index` and a `value`. The `$.each()` function applies this function to *each* element in the array. Each element has a corresponding index and value which get passed to this function.

The other version of the `each()` function allows you to apply a function to elements that are the result of a jQuery selector call. An example:

```
1 $("li").each(function(index, element) {
2   console.log( index + ": " + $( element ).text() );
3 });
```

In this example, the selector will get all `<li>` elements in the HTML document and then apply the function to each of them. As with the other version, the function is passed the index of the element as well as the DOM element itself.

Additionally, note that in line 3 we "wrap" the `element` in a call to jQuery. Recall that the `text()` function is one that is provided by jQuery and is not part of standard JavaScript. By wrapping the `element` inside a call to jQuery, `$(element)` it transforms it into a jQuery object which *does* have the `text()` function available.

Since ECMAScript version 5 (the official specification for the language we call JavaScript), the `forEach` function has been supported for arrays that essentially does the same thing as the jQuery `$.each()` function (see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach for details). We prefer jQuery's version as it is more flexible for how we'll typically use it.