Week 4 Module 2: Iterate vs. Vectorize

CSCI E-5a: Programming in R

Let's clear the global computing environment:

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

Module Overview and Learning Outcomes

Hello! And welcome to Module 2: Iterate vs. Vectorize.

In this module, we'll compare two methods for automating repetition: iteration and vectorization.

- In section 1, we'll compare the similarities and differences between iteration and vectorization.
- In section 2, we'll learn three principles that will help us to determine when to use vectorization, and when to use iteration.
- In section 3, we'll discuss why it's often not a great idea to try to optimize your code.
- In section 4, we'll see a practical example where iteration can help us to solve a problem.

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

- Explain the three principles that govern the choice between iteration vs. vectorization.
- Explain when to use vectorization.
- Explain when to use iteration.

There are no new built-in R functions in this module.

All right! Let's start out by comparing iteration and vectorization.

Section 1: Comparing the methods

> **Main Idea:** *We can compare iteration with vectorization*

In this section, we'll compare the similarities and differences between iteration and vectorization.

In the previous module, we learned how to automate repetitive computations by using a `for` loop.

We've also seen that repetition can sometimes be automated by using vectorized operations, or even just vector functions.

For instance, let's go back to the code we used to count the number of elements in `animal.vector`:

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

counter <- 0

for( current.animal in animal.vector ) {

    counter <- counter + 1

}

counter
```

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

Of course, we could have just used the built-in R function `length()`:

```
length( animal.vector )
```

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

The iterative approach was more complicated and required more work from us.

Likewise, let's consider the code for calculating the total sum of the values in a numeric vector:

```
numeric.vector <- 1:5

total <- 0

for( current.number in numeric.vector ) {

    total <- total + current.number
}

total
```

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

We could have just used the `sum()` function:

```
sum( numeric.vector )
```

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

Why did we spend so much time learning how to use `for` loops to implement these procedures, when there were already built-in functions in R to do the same thing?

The answer is that for teaching purposes I wanted to find some very simple programs to illustrate various techniques such as how to use an accumulator variable.

But I agree – it's much better to use the `length()` or `sum()` functions than to write an iterative routine to do this.

In fact, it's always better to use a built-in R function than to write your own using iteration.

So those are the similarities and differences between iteration and vectorization.

Now let's consider three principles that can help us to think about iteration vs. vectorization.

Section 2: The Three Principles

> **Main Idea:** *We can use the three principles to help us think about iteration vs. vectorization*

In this section, we'll learn three principles that will help us to determine when to use vectorization, and when to use iteration.

So again we ask the question: why did we learn about iteration, if we can do better without it?

That's an important question, and the answer to it is nuanced.

Many people have the mistaken idea that using `for` loops is a bad idea, and should be avoided.

This is wrong.

There are three basic principles at work here:

- If you can perform a calculation using either vectorized operations or iteration, then it's almost always better to use vectorized operations.
- Not every calculation can be implemented using vectorized operations.
- If you can't implement something using vectorized operations, then you should use iteration with `for` loops.

We can see these principles in action whenever we try to implement a dot product.

For instance, let's suppose that we want to calculate the dot product of two variables X and Y:

X	Y
3	7
1	9
5	2

Let's create two vectors to represent these variables:

```
x.vector <-
    c( 3, 1, 5 )

y.vector <-
    c( 7, 9, 2 )
```

To calculate the dot product using iteration with a `for` loop, we have to work with an accumulator variable to hold the running total.

We also have to iterate over the indices of the vectors, because we are iterating over two vectors at once.

Here's the code:

```
running.total <- 0

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

for( index in index.vector ) {

    running.total <-
```

```
        running.total +
        (x.vector[ index ] * y.vector[ index ])
}

running.total
```

```
## [1] 40
```

Make sure you understand how this code works, and how it calculates the dot product of the two vectors.

Now let's calculate the dot product using vectorized operations and the built-in R function `sum()`:

```
sum( x.vector * y.vector )
```

```
## [1] 40
```

Hey!

That was a *lot* simpler to do.

The iterative version with `for` required a lot of work:

- We had to create the accumulator variable `total`.
- We had to set up the `for` loop.
- We had to access each element of `x.vector` and `y.vector` using positive integer indexing.
- We had to update the accumulator variable `total` in each iteration.

It's easy to see here that vectorized operations are vastly preferable to using iteration.

It turns out that vectorized operations are also faster.

That's because when we use the vectorized operations we are relying on the built-in functions in R, and these have been coded so that they are very fast.

When we use iteration, R has to follow our instructions, and that makes it much slower.

Some people mistakenly think that this means that `for` loops are slow.

This is incorrect, and in general `for` loops are not particularly slow, at least compared to the alternatives.

The problem is the positive integer indexing.

R is optimized to work with vectorized operations, and when you start explicitly indexing the elements like this it slows down the operation.

When you use iteration with a `for` loop, you're almost always going to use positive integer indexing sooner or later, and that's why it appears that the `for` loop is slow.

So this is the first principle:

> If you can perform a calculation using either vectorized operations or iteration, then it's almost always better to use vectorized operations.

So what good is a `for` loop?

This brings us to our second principle: it's not always possible to use vectorized operations.

Let's consider our very first `for` loop, where we simply printed out the elements of the `animal.vector`:

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

for( current.animal in animal.vector ) {

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

}
```

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

How would you do this using vectorized operations?

You might think back to this code:

```
exp( 1:5 )
```

```
## [1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

Here, we mapped the function `exp()` over the vector 1:5, and R created a new vector consisting of the value of `exp()` evaluated at each element of 1:5.

Would this work with our iterative printer program?

Let's try this, but before you run the code try to figure out what it's going to do:

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

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

Hey!

That didn't vectorize the way we wanted it to.

Instead, it just printed out the vector.

Not every function vectorizes like `exp()`.

So for this little bit of code that prints out a report the vectorized approach is not available to us.

This is an example of the second principle:

> Not every calculation can be implemented using vectorized operations.

Vectorized operations work well when we can express a calculation in terms of element-wise or component-wise arithmetic.

They also work well when we have coordinated vectors and we want to perform an element-wise or component-wise calculation.

That's not what's going on with this simple report program that we wrote.

We're not doing arithmetic computations on the elements of the vector – we're printing out messages, and the `cat()` statement doesn't vectorize.

So this is an example where vectorized operations are not going to solve our problem.

There are more advanced methods that we can use to vectorize this process, but they are complicated.

By contrast, it's easy to understand the iterative code.

So those are the three principles that help us to decide whether to use iteration or vectorized operations.

Now let's see why it's often a good idea to avoid optimizing your code.

Section 3: Avoid Optimizing

> **Main Idea:** *Don't always optimize your code*

In this section, we'll discuss why it's often not a great idea to try to optimize your code.

I mentioned in the previous section that positive integer indexing is "slow".

This is a relative statement, and in practice code that uses positive integer indexing is usually perfectly acceptable in terms of performance.

Let's do an experiment.

First, let's make two vectors, each of which has 100,000 elements.

I'm going to use a random number generator to populate these vectors:

```
random.vector.1 <-
    rnorm( 100000 )

random.vector.2 <-
    rnorm( 100000 )
```

First let's calculate the dot product of these two vectors using a vectorized approach:

```
sum( random.vector.1 * random.vector.2 )
```

```
## [1] -112.5597
```

Next, let's calculate the dot product using an iterative approach with a `for` loop:

```
total <- 0

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

for( index in index.vector ) {

    total <-
        total +
        (random.vector.1[ index ] *
             random.vector.2[ index ])
}

total
```

```
## [1] -112.5597
```

If you run these two programs, you'll see that the vectorized code is virtually instantaneous.

The iterative takes a little bit of time, but not more than a second.

So even in this case, where we have vectors with 100000 elements, the iterative code runs very quickly.

It just doesn't run as fast as the vectorized code.

In some situations, that difference of less than a second might be very important.

If you're trying to create real-time high-speed graphics, then yes, that difference in performance might be critical.

But we don't use R for those sorts of applications.

Instead, we typically use R for data analysis, and a difference of less than a second is negligible.

This gives rise to another fundamental principle:

> Don't worry about speed if it's not important.

Worry about the speed of your code when it becomes an issue.

In fact, we can generalize this principle:

> Don't try to optimize your code if you don't have to.

That seems bizarre – why wouldn't I want my code to be the best it could possibly be?

The answer is that optimization frequently has associated costs, and the benefits of the "optimal" code frequently do not justify the costs.

If you spend 2 hours re-writing your code to save 10 milliseconds of execution, that's not a great investment of your time.

Again, there are high-performance situations where you really do need your code to run as fast as possible, but if that's the case then you probably shouldn't be coding in R.

And this doesn't violate the principle, because the principle is explicitly limited to those situations where your code doesn't have to run optimally.

Often, a "good-enough" solution is the more practical choice.

So that's why you shouldn't always try to optimize your code.

Now let's see an example of this when we calculate final grades.

Section 4: Practical example

> **Main Idea:** *Let's see a practical example*

In this section, we'll see a concrete example where iteration can help us to solve a problem.

So far, we've seen how to use vectorized operations to calculate the Preliminary Score 1 and Preliminary Score 2 for the final course grade.

Once we have these scores, the next step is to select the maximum value.

But we haven't seen any code that allows us to do this.

Let's look at an example.

Here are the scores for five students:

Preliminary Score 1	Preliminary Score 2
85	88
92	91
95	92
82	86
91	89

Let's make some vectors to represent these scores:

```
score.1.vector <-
    c( 85, 92, 95, 82, 91 )

score.2.vector <-
    c( 88, 91, 92, 86, 89 )
```

Now there is a built-in R function called `max()`, and this takes a set of numbers and returns the maximum value.

```
max( 4, 7 )
```

```
## [1] 7
```

Hey!

You might think that we can use vectorized operations with the `max()` function: we'll select the maximum of the two vectors on a component-wise or element-wise basis.

Here's the code, but before we run it, try to think what's going to happen:

```
max( score.1.vector, score.2.vector )
```

```
## [1] 95
```

Oh no!

The `max()` function didn't vectorize the way we hoped it would.

Instead, it just calculated the maximum value across both vectors.

So we're not going to be able to vectorize this problem, at least not this way.

Let's try an iterative approach.

Before we do this, we have to create a new vector for storing the results of our calculation.

Here's how we create a numeric vector with storage for 5 elements:

```
graduate.course.score.vector <-
    numeric( 5 )
```

Now let's construct a `for` loop:

```
for( index in 1:length( score.1.vector ) ) {

    graduate.course.score.vector[ index ] <-
        max(
            score.1.vector[ index ],
            score.2.vector[ index ]
        )
}
```

I must point this out: notice in this code that we are actually indexing *three* vectors.

This illustrates the power of looping over indices.

Now let's directly display the `graduate.course.score.vector`:

```
graduate.course.score.vector
```

```
## [1] 88 92 95 86 91
```

Hey!

Just what we wanted.

The important lesson here is that you should use the tool that helps you to solve your problem in the simplest manner possible.

Sometimes that will be a vectorized operation, and when you can use that, that will almost always be the right approach.

But using iterative methods is fine as well.

The important issue is to write your code cleanly so that you can be confident in its correctness.

In all fairness, there is a built-in R function that we can use to perform this calculation, and we'll see it in the next lecture.

But that doesn't change the fundamental principles, because you might not know about that function.

So that's an example of a particular situation where iterating with a `for()` loop can be a good design choice.

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

Module Review

In this module, we compared two methods for automating repetition: iteration and vectorization.

- In section 1, we compared the similarities and differences between iteration and vectorization.
- In section 2, we learned three principles that will help us to determine when to use vectorization, and when to use iteration.
- In section 3, we discussed why it's often not a great idea to try to optimize your code.
- In section 4, we saw a practical example where iteration can help us to solve a problem.

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

- Explain the three principles that govern the choice between iteration vs. vectorization.
- Explain when to use vectorization.
- Explain when to use iteration.

There were no new built-in R functions in this module.

All right! That's it for Module 2: Iterate vs. Vectorize.

Now let's move on to Module 3: Amortizing a Loan.