Week 5 Module 5: Conditional Branching

CSCI E-5a: Introduction to R

Let's clear the global environment:

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

Let's load in the objects for this module:

```
load( file = "Module 5 R Objects.Rdata" )
```

Module Overview and Learning Objectives

Hello! And welcome to Module 5: Conditional Branching.

In this module, we'll learn about *conditional branching*.

- In section 1, we'll learn the basics of conditional branching.
- In section 2, we'll find that the `if()` function does *not* vectorize.
- In section 3, we'll see another function that can sometimes operate as a vectorized `if` statement.

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

- Explain how conditional branching works.
- Explain how the `if()` function does not vectorize.
- Use the `ifelse()` function as a vectorized version of `if()`.

There are two new built-in R functions in this module:

- `if()`
- `ifelse()`

All right! Let's get started by defining the concept of conditional branching.

Section 1: Conditional Branching

> **Main Idea:** *We can use conditional branching to control program flow*

In this section, we'll learn the basics of conditional branching.

We've seen how to construct loops, in which we control the flow of program execution so that it repeats a code block.

Now we'll consider the other important flow-of-control operation, called *conditional branching*.

In this operation, we perform some sort of test (that's the "condition"), and then execute certain code depending on whether or not the test was `TRUE` or `FALSE` (that's the "branching").

Let's start with the most basic form of conditional branching:

- First, we use the keyword `if` to indicate that we are performing conditional branching, followed by an open parenthesis.
- Next, we perform a test, which is just an expression that evaluates to `TRUE` or `FALSE`, and we follow this with a closing parenthesis.
- Finally, we have a code block, which is delimited using curly braces.

If the expression in the test evaluates to `TRUE`, then we execute the code in the code block, but if the code evaluates to `FALSE`, then we do not execute the code in the code block.

For instance, in this example, the test is `TRUE`, so R executes the code in the following code block:

```
x <- 6

if( x == 6 ) {
    cat( "We got into the code block!\n" )
}
```

```
## We got into the code block!
```

On the other hand, if the test is `FALSE`, then the following code block is *not* executed:

```
x <- 5

if( x == 6 ) {
    cat( "We got into the code block!\n" )
}
```

We can extend this process by using the keyword `else`, which is followed by a second code block.

In this approach, if the conditional test evaluates to `TRUE`, then the first code block is executed, but if it evaluates to `FALSE` then the second code block following the keyword `else` is executed.

The keyword `else` has to occur on the same line as the closing curly brace for the initial code block:

```
animal <- "dog"

if( animal == "hedgehog" ) {

    cat( "The animal is a hedgehog\n" )

} else {

    cat( "The animal is not a hedgehog\n" )

}
```

```
## The animal is not a hedgehog
```

We can chain together `if` and `else` statements to allow for multi-way branching:

```
animal <- "bunny rabbit"

if( animal == "hedgehog" ) {
    cat( "The animal is a hedgehog\n" )
} else if( animal == "armadillo" ) {
    cat( "The animal is an armadillo\n" )
} else if( animal == "bunny rabbit" ) {
    cat( "The animal is a bunny rabbit\n" )
} else {
    cat( "Some other thing\n" )
}
```

```
## The animal is a bunny rabbit
```

Conditional branching with the `if()` function will be very important in Lecture 8: Functions, when we learn how to write our own user-defined functions.

So that's how to use conditional branching to control the flow of execution in a program.

Now let's see what happens when we try to vectorize the `if()` function.

Section 2: `if` doesn't vectorize

> **Main Idea:** *The `if()` function doesn't vectorize*

In this section, we'll find that the `if()` function does *not* vectorize.

The `if()` function doesn't vectorize, at least not in the way that we often want it to.

Let's create a test vector:

```
test.vector <- c( 4, 2, 9, 3, 1, 6, 12)
```

We want to scan through the test vector and print out a short message each time an element is strictly greater than 3.

Let's see if we can implement this using a vectorized approach:

```
if( test.vector > 3 ) {
    cat( "Hi Mom!\n" )
}
```

```
## Warning in if (test.vector > 3) {: the condition has length > 1 and only the
## first element will be used
```

```
## Hi Mom!
```

Notice that although `test.vector` has 7 elements, R printed out only one statement, so this statement did not vectorize.

Why did this happen?

The answer is that `if()` doesn't vectorize; instead, it just takes the first element of the vector and applies the `if()` statement to that.

Actually, the warning message that R prints out is unusually helpful and descriptive.

This is a situation where a `for()` loop works great:

```
for( element in test.vector ) {

    if( element > 3 ) {
        cat( "Element:", element, "\n" )
    }

}
```

```
## Element: 4
## Element: 9
## Element: 6
## Element: 12
```

The important point is that you have to be careful about vectorization, and to check and make sure that R actually does what you thought it would do.

So the `if()` function doesn't vectorize.

Now let's explore another function that performs conditional branching and does vectorize in the way that we might want it to.

Section 3: The `ifelse()` function

> **Main Idea:** *We can use the `ifelse()` function to perform vectorized conditional branching*

In this section, we'll see another function that can sometimes operate as a vectorized `if` statement.

The `ifelse()` function can sometimes operate as a vectorized `if()` statement (sort of).

The `ifelse()` function takes three input arguments:

- A logical value, called `test`.

- A value to be returned if the logical value is `TRUE`, called `yes`.
- A value to be returned if the logical value is `FALSE`, called `no`.

The important point about `ifelse()` is that it can be vectorized.

Let's see how this works.

First, we'll create a logical test vector:

```
logical.test.vector <-
  c( TRUE, TRUE, FALSE, TRUE, FALSE )
```

Let's create a new vector based on `logical.test.vector`:

- If a particular element of `logical.test.vector` is `TRUE`, then the corresponding element of the new vector is "More Kale!".
- If a particular element of `logical.test.vector` is `FALSE`, then the corresponding element of the new vector is "More Tofu!".

To create this vector, we can use the `ifelse()` function:

- The `test` input argument is just the `logical.test.vector`.
- The `yes` input argument is the character string value "More Kale!"
- The `no` input argument is the character string value "More Tofu!"

```
ifelse(
  test = logical.test.vector,
  yes = "More Kale!",
  no = "More Tofu!"
)
```

```
## [1] "More Kale!" "More Kale!" "More Tofu!" "More Kale!" "More Tofu!"
```

Make sure you understand how this function works, and how this output vector was constructed.

Just for reference, we could also construct this vector using a combination of a `for()` loop and an `if()` statement:

```
number.of.elements <-
  length( logical.test.vector )

output.vector <-
  numeric( number.of.elements )

for( i in 1:number.of.elements ) {

  if( logical.test.vector[ i ] == TRUE ) {
    output.vector[ i ] <- "More Kale!"
  } else {
    output.vector[ i ] <- "More Tofu!"
  }
}

output.vector
```

```
## [1] "More Kale!" "More Kale!" "More Tofu!" "More Kale!" "More Tofu!"
```

Notice that in the end both approaches produce the same output.

However, the iterative approach is more complicated to implement, because we have to specify every step of the computation.

By contrast, the `ifelse()` function handles a lot of this work, and so the code is much simpler when we use this approach.

I must emphasize – the `ifelse()` function is perhaps a little more abstract, and if you don't feel comfortable with this and prefer the iterative approach, that's perfectly fine.

As always, the most important thing is to write clear and correct code that works and does what it's supposed to do.

You might wonder if the `ifelse()` function is particularly useful, and I admit that it took me a long to appreciate its merits.

The point is that we can construct the logical test vector using a vectorized operation.

This enables us to effectively implement a conditional test for each element of a vector and then branch on the result, so that this can function as a sort of vectorized `if()` statement.

Let's see how this works.

Let's recall our `test.vector`:

```
test.vector
```

```
## [1]  4  2  9  3  1  6 12
```

Now let's consider an example of how to use the `ifelse()` function (I admit this is a little artificial).

We want to create a new vector:

- If an element of `test.vector` is greater than or equal to 3, then the corresponding element of the new vector has the same value.
- If an element of `test.vector` is strictly less than 3, then the corresponding element of the new vector is 0.

In other words, to create the new vector we make a copy of `test.vector` where any elements strictly less than 3 are converted to 0.

To do this using `ifelse()`, we first perform a vectorized comparison of `test.vector` with the numeric value 3:

```
logical.test.vector <-
  test.vector >= 3

logical.test.vector
```

```
## [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
```

Now we can use `ifelse()` with this logical test vector:

```
ifelse(
    test = logical.test.vector,
    yes = test.vector,
    no = 0
)
```

```
## [1]  4  0  9  3  0  6 12
```

Notice here that we're vectorizing over both `logical.test.vector` and `test.vector`.

By the way, we didn't really need to store the result of the comparison operation in the dedicated variable `logical.test.vector`, because the only time we use it is in the `ifelse()` statement.

So we could have just done this:

```
ifelse(
    test = test.vector >= 3,
    yes = test.vector,
    no = 0
)
```

```
## [1]  4  0  9  3  0  6 12
```

So that's how the `ifelse()` function allows us to perform a vectorized conditional branching operation.

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

Exercise 1: Vectorized conditional branching

The R object `exercise.1.data` is a numeric vector.

Create a new character string vector:

- If an element of `exercise.1.data` is less than 0, then the corresponding element in the new vector should be the character string "Cold".
- If an element of `exercise.1.data` is greater than or equal to 0, then the corresponding element in the new vector should be the character string "Hot".

Directly display the vector `exercise.1.data`, and try to determine what the new vector will be, based on this data.

Then write R code to create the new vector.

Ideally, you should do this using two approaches:

- Using vectorized operations with `ifelse()`.
- Using a `for()` loop and an `if()` statement.

Which method do you prefer?

Solution

Module Review

In this module, we learned about *conditional branching*.

- In section 1, we learned the basics of conditional branching.
- In section 2, we found that the `if()` function does *not* vectorize.
- In section 3, we saw another function that can sometimes operate as a vectorized `if` statement.

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

- Explain how conditional branching works.
- Explain how the `if()` function does not vectorize.
- Use the `ifelse()` function as a vectorized version of `if()`.

There were two new built-in R functions in this module:

- `if()`
- `ifelse()`

All right! That's it for Module 5: Conditional Branching.

Now let's move on to Module 6: Histograms and Curves.

Solution to the Exercise

Exercise 1: Vectorized conditional branching

The R object `exercise.1.data` is a numeric vector.

Create a new character string vector:

- If an element of `exercise.1.data` is less than 0, then the corresponding element in the new vector should be the character string "Cold".
- If an element of `exercise.1.data` is greater than or equal to 0, then the corresponding element in the new vector should be the character string "Hot".

Directly display the vector `exercise.1.data`, and try to determine what the new vector will be, based on this data.

Then write R code to create the new vector.

Ideally, you should do this using two approaches:

- Using vectorized operations with `ifelse()`.
- Using a `for()` loop and an `if()` statement.

Which method do you prefer?

Solution

First, let's directly display the vector:

```
exercise.1.data
```

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

In `exercise.1.data`, positive numbers occur in positions 1, 2, 4, 5, and 7, while negative numbers occur in positions 3, 6, and 8.

Thus, the new vector should have the character string "Hot" in positions 1, 2, 4, 5, and 7, and the character string "Cold" in positions 3, 6, and 8.

We can do this with `ifelse()`:

```
new.vector <-
  ifelse(
    exercise.1.data >= 0,
    "Hot",
    "Cold"
  )

new.vector
```

```
## [1] "Hot"  "Hot"  "Cold" "Hot"  "Hot"  "Cold" "Hot"  "Cold"
```

This is exactly what we had predicted.

To do this using a `for()` and an `if()` statement is a little more work:

```
number.of.elements <-
  length( exercise.1.data )

new.vector <-
  character( number.of.elements )

for( i in 1:number.of.elements ) {

  if( exercise.1.data[ i ] >= 0 ) {
    new.vector[ i ] <- "Hot"
  } else {
    new.vector[ i ] <- "Cold"
  }
}

new.vector
```

```
## [1] "Hot"  "Hot"  "Cold" "Hot"  "Hot"  "Cold" "Hot"  "Cold"
```

This is exactly the same result as before, but it was a little bit more work to get there using a `for()` loop and an `if()` statement, compared to doing this with an `ifelse()` statement.