Iterations: For and While Loops

Stat 133, Fall 2024

Lectures 15 & 16, 10/10/2024

1. Introduction to Loops in R

Loops are fundamental structures in programming that allow repetitive execution of a block of code. R supports two main types of loops:

- `for` loops
- `while` loops

Each loop type has its own strengths and is suitable for different use cases. In this document, we will explore the syntax, usage, and examples for both `for` and `while` loops.

2. `for` Loops

A `for` loop iterates over a sequence, executing a block of code for each element in that sequence. It's useful when the number of iterations is known beforehand.

2.1. Syntax of `for` Loops

```
for (variable in sequence) {
    # code to execute on each iteration
}
```

Here, `variable` takes each value in `sequence` in turn, and the loop body is executed for each value.

2.2. Example: Basic `for` Loop

```
for (i in 1:5) {
    print(i)
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

2.3. Iterating Over Vectors

`for` loops can iterate over vectors or other data structures in R. For example:

```
numbers <- c(2, 4, 6, 8)
for (num in numbers) {
    print(num^2)
}
```

Output:

```
[1] 4
[1] 16
[1] 36
[1] 64
```

2.4. Nested `for` Loops

`for` loops can be nested to perform repeated operations on multi-dimensional structures like matrices.

```
matrix_data <- matrix(1:9, nrow = 3)
for (i in 1:nrow(matrix_data)) {
    for (j in 1:ncol(matrix_data)) {
        print(matrix_data[i, j] * 2)
    }
}
```

3. `while` Loops

A `while` loop continues to execute as long as a specified condition is `TRUE`. It is especially useful when the number of iterations is not predetermined.

3.1. Syntax of `while` Loops

```
while (condition) {
    # code to execute as long as condition is TRUE
}
```

The loop will stop executing once `condition` evaluates to `FALSE`.

3.2. Example: Basic `while` Loop

```
count <- 1
while (count <= 5) {
    print(count)
    count <- count + 1
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

3.3. Avoiding Infinite Loops

A common pitfall in `while` loops is accidentally creating an infinite loop if the condition never becomes `FALSE`. It's crucial to ensure that there is a mechanism to break out of the loop.

```
count <- 1
while (count <= 3) {
    print(count)
    # Ensure condition changes to avoid infinite loop
    count <- count + 1
}
```

4. Loop Control Statements

R provides additional control statements that allow fine-tuning the behavior within loops:

- `break` - immediately exits the loop
- `next` - skips the current iteration and proceeds to the next

4.1. Example: Using `break`

```
for (i in 1:10) {
    if (i == 5) {
        break
    }
    print(i)
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
```

4.2. Example: Using `next`

```
for (i in 1:5) {
    if (i == 3) {
        next
    }
    print(i)
}
```

Output:

```
[1] 1
[1] 2
[1] 4
[1] 5
```

5. Loop Alternatives: `apply` Functions

In R, the `apply` family of functions provides alternatives to loops, allowing for vectorized operations which are generally faster and more concise.

5.1. The `lapply()` Function

```
data <- list(a = 1, b = 2, c = 3)
result <- lapply(data, function(x) x^2)
print(result)
```

Output:

```
$a
[1] 1

$b
[1] 4

$c
[1] 9
```

6. Summary

Both `for` and `while` loops provide essential tools for repeated execution of blocks of code. `for` loops are useful when the number of iterations is known, while `while` loops are suitable for conditions that change dynamically. Loop control statements such as `break` and `next` offer finer control over loop execution. Additionally, the `apply` function in R allows for vectorized code which is often more efficient and provides alternatives to loops.