# Lecture 3.4 – Looping

## Specific Learning Objectives:

**1.2.1 – Understand the way computers execute commands.**

**1.2.6 – Understand and successfully execute a while loop.**

**1.2.8 – Understand and successfully execute repeat and for loops.**

**3.5 – Think and work independently with code.**

# Continuing Flow Control in Computing

- Looping is the other main method of flow control available for helping computers run without intervention.

    - **Loops** are sections of code that runs repeatedly until a stop condition is met.

    - Loops allow code to run while changing things automatically and running sections of code.

    - Functions, conditional statements, and loops make for a extremely powerful trifecta!

# Loops


https://www.youtube.com/watch?v=wxds6MAtUQ0

# `repeat` loops

- Repeat loops simply repeat a section of code. They will continue to repeat until you break.

**Please repeat the following line(s).**

```
repeat print("This is the song that never ends.")
```

**This is what you should repeat.**

- Put multiple lines inside curly braces.

```
repeat {
    print("This is the song that never ends.")
    print("Yes it goes on and on, my friends.")
}
```

# Stopping `repeat` loops

- **`break`** will stop a repeat loop (and any other type of loop).

```
repeat {
    print("This is the song that never ends.")
    print("Yes it goes on and on, my friends.")
    break
}
```

This will only allow the loop to run once.

- Setting up a condition to meet before breaking is a little more useful.

```
x <- 1
repeat {
    x <- x + 1
    print("This is the song that never ends.")
    print("Yes it goes on and on, my friends.")
    if (x == 10) break
}
```

This will break the loop when x = 10.

# Check Your Understanding

**Write a `repeat` loop that prints the number of the repetition and stops after 100 repetitions.**

# `while` loops

- While loops look very similar to repeat loops, but they take a condition for stopping as an argument.

  - Here we can use x as a counter to stop the loop when it reaches 11.

```
x <- 1
while (x < 11) {
    print("This will keep printing while x is less than 11.")
    x <- x + 1
}
```

- Don't forget to include a statement that turns `FALSE`, or you will wait forever for the loop to stop!

# `while` loops

- It's easy to substitute the x to use the value of x in any calculation.

```
x <- 1
while (x < 11) {
   print(paste("x is", x, "out of 10."))
   x <- x + 1
}
```

```
[1] "The value of x is 1 out of 10."
[1] "The value of x is 2 out of 10."
[1] "The value of x is 3 out of 10."
[1] "The value of x is 4 out of 10."
[1] "The value of x is 5 out of 10."
[1] "The value of x is 6 out of 10."
[1] "The value of x is 7 out of 10."
[1] "The value of x is 8 out of 10."
[1] "The value of x is 9 out of 10."
[1] "The value of x is 10 out of 10."
```

# `for` loops

- `for` loops repeat for a predetermined number of times that you set as an argument: a variable that will cycle through each number in a vector.

    - Each time the loop starts again, the value of the variable will change to the next value in the vector.

**variable will run at every number in a sequence from 1 to 10**

```
for (x in 1:10) {
   print(paste("x is", x, "out of 10."))
}
```

**This example produces the same output as the repeat and while loops!**

# `for` loops

- It doesn't need to be a sequence of whole numbers.

```
for (x in seq(1, 5, by = 0.1)) {
   print(paste("The value of x is", x, "out of 5."))
}
```

- It doesn't even need to be a sequence, any vector will do!

```
for (x in runif(10)) {
   print(paste0("The value of x is", x, "."))
}
```

# `for` loops

- You can also nest for loops! Just be sure to define your variables separately and keep them straight!

```
for (x in 1:5) {
    for (y in 6:10) {
        print(paste0("The value of x is ", x,
                     " and y is ", y, "."))
    }
}
```

```
[1] "The value of x is 1 and y is 6."
[1] "The value of x is 1 and y is 7."
[1] "The value of x is 1 and y is 8."
[1] "The value of x is 1 and y is 9."
[1] "The value of x is 1 and y is 10."
[1] "The value of x is 2 and y is 6."
[1] "The value of x is 2 and y is 7."
[1] "The value of x is 2 and y is 8."
[1] "The value of x is 2 and y is 9."
[1] "The value of x is 2 and y is 10."
[1] "The value of x is 3 and y is 6."
[1] "The value of x is 3 and y is 7."
[1] "The value of x is 3 and y is 8."
[1] "The value of x is 3 and y is 9."
[1] "The value of x is 3 and y is 10."
[1] "The value of x is 4 and y is 6."
```

**The inner loop will do a full cycle every repetition of the outer loop!**

# Implicit looping with `apply()`

- **`apply()`** applies a function over the margins of a matrix. It's an 'implicit' loop because it has the same output as an explicit loop without using **`for`**, **`while`**, or **`repeat`**.

**Set up an example matrix:**

```
my.example <- matrix(1:15, nrow = 5)
```

**Computing column means with a for loop:**

**Preallocating space for answers!**

```
col.means1 <- rep(NA, length = ncol(my.example))
for (x in 1:ncol(my.example)){
  col.means1[x] <- mean(my.example[,x])
}
```

**Computing column means with `apply()`:**

```
col.means2 <- apply(my.example, MARGIN = 2, FUN = mean)
```

# What Good are Loops?

- What good are loops? They are really useful! Especially if you need to do repeated calculations, they will step through any number of calculations.

  - An example: you want to do an element-wise calculation on a vector. Here, we'll add 1.3 to each element of **`vec1`**.

**Defining elements of `vec1`**

**Preallocating space for `vec2`**

```
vec1 <- c(2, 1, 4, 5, 7, 2)
vec2 <- rep(1, length = length(vec1))
for (x in 1:length(vec1)) {
  vec2[x] <- vec1[x] + 1.3
}
```

**For loop will cycle through every element of `vec1`**

**Performing calculation and storing the result in `vec2`**

**This is a simple calculation, but you can do much more complicated stuff!**

# When should you use loops? And which one?

- Loops are great for repeating calculations, so whenever you need to repeat a calculation a bunch of times, you will want to put it into some kind of loop!

  - All of these looping options are very similar, and can be coded to produce the same output (so it essentially doesn't matter in most cases)

  - For loops are probably the most common, because they are easiest to understand in terms of starting and stopping.

  - Many people also use **`apply()`** or similar functions to implicitly loop in R. These are more restrictive because of the types of data they accept and output. You can always default to a for loop!

# Check Your Understanding

**The Fibonacci sequence is a series of numbers in which each number in the series is defined by adding the previous two numbers in the sequence. The first two Fibonacci numbers are 0 and 1, then 1, then 2, then 3, etc.**

**Write a loop that will calculate the first 500 Fibonacci numbers. I've gotten you started with the first four numbers in the Lecture Notebook!**

```
Fib.nums <- rep(NA, length = 500)
Fib.nums[1:2] <- c(0, 1)
Fib.nums[3] <- Fib.nums[1] + Fib.nums[2]
Fib.nums[4] <- Fib.nums[2] + Fib.nums[3]
```

# Action Items

1. **Complete Assignment 3.4.**

2. **Read Davies Ch. 10.3 for next time.**