Programming Basics II
Loops
Abstraction

- Copying and pasting code is something you never want to do!
- **Functions** are one form of abstraction that can be used in place of copy and paste. **Loops** are another.
What is a loop?

- Loops are used to execute repetitive tasks.
- Loops use predicates to decide when to stop. They keep going, as long as the predicate is TRUE, and stop when it is FALSE.
- There are many types of loops: e.g.,
  - `for` loops
  - `while` loops
  - `do while` loops
  - `repeat until` loops
While Loop

```java
while (loop condition) {
    loop body
}
```

A **while** loop executes its body **while** the condition is **TRUE**.

A **while** loop continues, until the condition becomes **FALSE**.
Infinite Loops

while (loop condition) {
    loop body
}

A while loop never executes its body if the condition is initially FALSE.

A while loop executes forever if the condition is always TRUE. Beware!
Example

maxCapacity <- 150
currCapacity <- 100
while (currCapacity < maxCapacity) {
    currCapacity <- currCapacity + 1
}

At the start of this while loop, the room is not at its maximum capacity.

It continues to execute until the room is full.
Example

maxCapacity <- 150
currCapacity <- 200
while (currCapacity < maxCapacity) {
    currCapacity <- currCapacity + 1
}

What happens now?
Example

maxCapacity <- 150
currCapacity <- 100
while (currCapacity < maxCapacity) {
    currCapacity <- currCapacity - 1
}

And now?
For loop

A for loop runs for a **prespecified** number of times, namely the length of a vector. Here is an example:

```r
for (i in 1:3) {
  print(i)
}
```

1
2
3
Sequences

You can use the `seq` function to create a vector:

```r
> seq(0, 1, by = 0.2) # step by 0.2
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

You can also generate the same sequence by specifying the number of elements:

```r
> seq(length = 6, from = 0, by = .2)
[1] 0.0 0.2 0.4 0.6 0.8 1.0
```

You can also simply use `:`, to step by the default step size of 1:

```r
> 0:5
[1] 0 1 2 3 4 5
```
Example using a sequence

```r
for (i in seq(1, 10, by = 2)) {
    print(i)
}
```
1
3
5
7
9
Example using a vector of strings

```r
presidents <- c("Washington", "Adams", "Jefferson")
for (p in presidents) {
  print(p)
}
Washington
Adams
Jefferson
For loop

A for loop runs for a prespecified number of times.
A for loop looks like this:

```javascript
for (variable in vector) {
  loop body
}
```

A for loop runs as many times as the length of the vector.
Variables

What if you want to compute the sum of a vector of numbers? You can use a variable and a for loop!

```r
sum <- 0
for (i in 1:5) {
  sum <- sum + i
}
```

Initially, the value of `sum` is 0. After running through the loop once, the value of `sum` is 1. And after a second run through, it becomes 3. By the end, it is 15.
Variables

What’s wrong with this version of the code?

```r
for (i in 1:5) {
    sum <- 0
    sum <- sum + i
}
```

What is the final value of `sum`?
In-class Activity

Write a while loop that sums the numbers from 1 to 5.
In-class Activity

Write a while loop that sums the numbers from 1 to 5.

```r
sum <- 0
i <- 1
while (i <= 5) {
  sum <- sum + i
}
print(sum)
```

Yikes!!! Infinite loop!!! The `print` statement never executes.
In-class Activity

Write a while loop that sums the numbers from 1 to 5.

```plaintext
sum <- 0
i <- 1
while (i <= 5) {
    sum <- sum + i
    i <- i + 1
}
print(sum)
15
```
Which looping structure to use when?

<table>
<thead>
<tr>
<th>for loop</th>
<th>while loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you know in advance how many times you need to iterate (i.e., repeat)</td>
<td>When you do not know in advance how many times you need to iterate</td>
</tr>
<tr>
<td>When the condition is fixed in advance</td>
<td>When the condition can change</td>
</tr>
</tbody>
</table>
Looking under the hood
Indices

You can reference the elements of a vector using their index.

> presidents <- c("Washington", "Adams", "Jefferson")
> presidents[1]
[1] "Washington"
> presidents[2]
[1] "Adams"
> presidents[3]
[1] "Jefferson"
Indices

You can also access the elements of a vector using their *index* within a `for` loop.

```r
presidents <- c("Washington", "Adams", "Jefferson")
for (i in 1:length(presidents)) {
    print(presidents[i])
}
Jefferson
Adams
Washington
```
Looping backwards

What if you want to loop backwards? You reverse the order of the sequence:

```r
presidents <- c("Washington", "Adams", "Jefferson")
for (i in length(presidents):1) {
  print(presidents[i])
}

Jefferson
Adams
Washington
```
Looping backwards

A better way, in this particular example, would be to use the `rev` function:

```r
presidents <- c("Washington", "Adams", "Jefferson")
for (i in rev(presidents)) {
  print(i)
}

Jefferson
Adams
Washington
```
Matrices

Vectors are only one-dimensional collections of data (a single row, or a column). Matrices are two-dimensional, like databases.

```
> m <- matrix(1:9, nrow = 3, ncol = 3)
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

So they have two-dimensional indices:

```
> m[1, 2]
[1] 4
```
Another way to create matrices

Matrices can also be created by binding vectors together. Below is an example of binding columns using `cbind`.

```
> x <- cbind(1, 1:4)
> x

bind two column vectors

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

```
> x <- cbind(x, 5:8)[, c(1, 3, 2)]
> x

bind a matrix and a vector

[,1] [,2] [,3]
[1,]  1  5  1
[2,]  1  6  2
[3,]  1  7  3
[4,]  1  8  4
```

rearrange the order of the columns
Another way to create matrices

Analogously, you can also use \texttt{rbind} to bind rows.

\begin{verbatim}
> x <- rbind(1, 1:4)
> x

[1,]  1   1   1   1
[2,]  1   2   3   4

> x <- rbind(x, 5:8)[c(1, 3, 2), ]
> x

[1,]  1   1   1   1
[2,]  5   6   7   8
[3,]  1   2   3   4
\end{verbatim}
Nested Loops

You can use nested loops to loop over matrices, with two different variables, one looping over each dimension.

This is a topic for a more advanced computer science course.
Now that you've learned the basics of variables, functions, conditionals, loops, and a few data structures, we'll do some in-class activities to master these concepts on Friday!