Programming Basics II

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.

Loops

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

```
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`.

Example

```
maxCapacity <- 100
currCapacity <- 50
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 <- 100
currCapacity <- 200
while (currCapacity < maxCapacity) {
    currCapacity <- currCapacity + 1
}
```

What happens now?

Example

```
maxCapacity <- 100
currCapacity <- 200
while (currCapacity < maxCapacity) {
    currCapacity <- currCapacity - 1
}
```

And now?

Example

```
maxCapacity <- 100
currCapacity <- 50
while (currCapacity < maxCapacity) {
    currCapacity <- currCapacity - 1
}
```

And now?

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!

For loop

A for loop runs for a **prespecified** number of times, namely the length of a vector.

Here is an example:

```
for (i in 1:3) { # 1:3 is a vector [1] 1 2 3
   print(i)
}
```

```
1
2
3
```

Sequences

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

```
> 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:

```
> 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:

```
> 0:5
[1] 0 1 2 3 4 5
```

Example using a sequence

```
for (i in seq(1, 10, by = 2)) {
    print(i)
}
```

```
1
3
5
7
9
```

Example using a vector of strings

```
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:

```
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!

```
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?

```
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.

```
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.

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

```
15
```

Which looping structure to use when?

for loop	while loop
When you know in advance how many times you need to iterate (i.e., repeat)	When you do not know in advance how many times you need to iterate
When the condition is fixed in advance	When the condition can change

Looking under the hood

Review

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

```
Washington
Adams
Jefferson
```

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.

```
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:

```
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:

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

```
Jefferson
Adams
Washington
```

Matrices

Vectors are only one-dimensional data tables (a single row, or a column).
Matrices are two-dimensional data tables.

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

bind two column vectors

rearrange the order of the columns

```
> y <- cbind(x, 5:8)[, c(1, 3, 2)]
> y
     [,1] [,2] [,3]
[1,]    1    5    1
[2,]    1    6    2
[3,]    1    7    3
[4,]    1    8    4
```

bind a matrix and a vector

Another way to create matrices

Analogously, you can also use `rbind` to bind rows.

```
> x <- rbind(1, 1:4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    2    3    4
```

```
> y <- rbind(x, 5:8)[c(1, 3, 2), ]
> y
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    5    6    7    8
[3,]    1    2    3    4
```

Arrays

Arrays are multidimensional data tables: i.e., they can store tensors.

A two-dimensional array is a matrix. A one-dimensional array is a vector.

To construct an array, you need data (e.g., `1:20`) and a dimension vector (e.g., `c(4, 5)`).

```
> x <- array(1:20, dim = c(4, 5)) # Generate a 4 by 5 array.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Arrays

```
> x <- array(1:20, dim = c(2, 4, 5))

> x
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]    9   11   13   15
[2,]   10   12   14   16

, , 3

     [,1] [,2] [,3] [,4]
[1,]   17   19    1    3
[2,]   18   20    2    4

, , 4

     [,1] [,2] [,3] [,4]
[1,]    5    7    9   11
[2,]    6    8   10   12

, , 5

     [,1] [,2] [,3] [,4]
[1,]   13   15   17   19
[2,]   14   16   18   20
```

Matrices vs. Arrays

```
> x <- matrix(1:10, 2)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> y <- array(1:10, c(2, 5))
> y
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> identical(x, y)
[1] TRUE
```

Nested Loops

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

This is a topic for another day.

An alternative: Lists

A sequence can only hold numerics.
An array should hold only one type of data.

A list (in R) is a collection of **components** of possibly varying types:

```
> my_list <- list("Fred", "Wilma", -1, c(1,3,5,7,9))
> my_list
[[1]]
[1] "Fred"
[[2]]
[1] "Wilma"
...
```

An alternative: Lists

As you can see list components are indexed with double brackets:

```
> my_list[[1]]
[1] "Fred"

> my_list[[4]]
[1] 1 3 5 7 9

> my_list[[4]][1]
[1] 1
```

An alternative: Lists

You can loop through a list, just like you might loop through a sequence or a vector.

```
my_list <- list("Fred", "Wilma", -1, c(1,3,5,7,9))
for (i in lst) {
    print(i)
}
```

```
Fred
Wilma
-1
1 3 5 7 9
```

The Truth about Data Frames

Now we've learned about **lists**, we can see that data frame is actually a list:

I.e., rows that comprise variables of different types in each column.

```
> state <- c("ri", "ny", "nj", "ct", "ma")
> incomes <- c(40, 49, 45, 61, 64)
> accountant <- data.frame(home = state, income = incomes)
> accountant
  home income
1   ri     40
2   ny     49
3   nj     45
4   ct     61
5   ma     64
```

Extras

Nested Loops

Just as we nested if-statements, we can nest loops!

Nested loops are particularly useful for looking through multidimensional data structures, like vectors, matrices, and arrays.

The `dim` function

```
> x1 <- # a vector of length 100
> x2 <- # ditto
> x3 <- # ditto

> data_rows <- rbind(x1, x2, x3)
> dim(data_rows)
[1] 3 100
> dim(data_rows)[1]
[1] 3
> dim(data_rows)[2]
[1] 100

> data_cols <- cbind(x1, x2, x3)
> dim(data_cols)
[1] 100 3
```

Nested Loops

Recall that an array (in R) is a monomorphic matrix: i.e., all data are the same type!

Let's start by creating an empty array.

```
> x <- array(numeric(), dim = c(5, 5))
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]   NA   NA   NA   NA   NA
[2,]   NA   NA   NA   NA   NA
[3,]   NA   NA   NA   NA   NA
[4,]   NA   NA   NA   NA   NA
[5,]   NA   NA   NA   NA   NA
```

Nested Loops

We first loop through this array by row indices.

```
x <- array(numeric(), dim = c(5, 5))
for (i in 1:dim(x)[1]) {          # dim(x)[1] returns the
                                    number of rows
     code goes here
}
```

Nested Loops

Then, we loop through each row by column indices.

```
x <- array(numeric(), dim = c(5, 5))
for (i in 1:dim(x)[1]) {
  for (j in 1:dim(x)[2]) {            # dim(x)[2] returns the
                                        number of columns
      code goes here
  }
}
```

Nested Loops

Here, we fill in each entry in the matrix with the product of the row and column indices.

```
x <- array(numeric(), dim = c(5, 5))
for (i in 1:dim(x)[1]) {
  for (j in 1:dim(x)[2]) {
    x[i, j] <- i * j         # do something for each entry
  }
}
```

Nested Loops

The result is:

```
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

Nested Loops

Similarly you can loop through a three-dimensional array three times.

```
x <- array(numeric(), dim = c(5, 5, 5))
for (i in 1:dim(x)[1]) {
  for (j in 1:dim(x)[2]) {
    for (k in 1:dim(x)[3]) {
      x[i, j ,k] <- i * j * k
    }
  }
}
```