Biostatistics Preparatory Course: Methods and Computing

Lecture 2

Matrix/vector manipulation and flow control

Part I: Vector/Matrix Manipulation

First, we will cover the basics of vector/matrix manipulation in R including:

1. Elementwise Operations
2. Matrix Operations

Vector Elementwise Operations

To begin to understand how R performs vector manipulations, we experiment with some simple operations:

```
### create vectors
set.seed(123)
vec1 <- sample(15,3)
vec2 <- sample(15,3)

### Perform operations
vec1 + 3
vec1^2
vec1 + vec2
vec1*vec2
vec2/vec3
```

Matrix Elementwise Operations

Now, we will run some simple matrix operations:

```
### create matrices
set.seed(123)
mat1 <- matrix(sample(15,6),2,3,byrow=TRUE)
mat2 <- matrix(sample(15,6),2,3,byrow=TRUE)

### perform operations
mat1*3
mat1^2
mat1+mat2
mat1*mat2
mat1/mat2
```

Elementwise Operations: Take-aways

- $+$, $-$, $*$, and / perform elementwise addition, subtraction, multiplication and division on matrices and vectors of the same dimension
- If these operators are applied to matrices/vectors of different dimensions, R will recycle the values and display an error
- Applying a boolean operator to a vector/matrix will return a vector/matrix with the corresponding logical values
- The `any` and `all` functions can be used to evaluate a boolean expression for any/all of the elements of a vector/matrix

Matrix operations

To begin to understand how R performs vector manipulations, we experiment with some simple operations:

```
### Dot product
vec1%*%vec2

### Transpose
t(mat2)

### Matrix multiplication
mat1%*%t(mat2)
mat1%*%vec1

### Inverse
matrix.id <- diag(5)
solve(matrix.id)
```

Matrix operations exercises

Flow Control

- For performing more complex tasks, which may involve repetition or conditional execution of code
- All this can be done with flow control, of which we will focus on:
  - For loop
  - While loop
  - Conditional statements

For Loop

- For loops allow for the repetition of a set of commands
- The basic syntax of a for loop:

```
for (___ in ___){
   # the commands to be repeated
}
```

- The first blank is for a variable, while the second is for a vector

For Loop

- For loops allow for the repetition of a set of commands
- The basic syntax of a for loop:

```
for (___ in ___){
   # the commands to be repeated
}
```

- The first blank is for a variable, while the second is for a vector
- The variable will be set to each element of the vector once, and the commands in the loop will be executed once for each element of the vector
    - The variable may or may not be involved in the commands being repeated

For Loop Examples

```
# here the variable in the first line is not involved
# in the command being repeated
for (i in 1:10){
  ## seq(1, 10) or seq_len(10) could also be used
  print("Hello")
}
```

For Loop Examples

```
# here the variable in the first line is not involved
# in the command being repeated
for (i in 1:10){
  ## seq(1, 10) or seq_len(10) could also be used
  print("Hello")
}

# Here the variable is involved
vec <- c("who", "what", "when", "where", "why")
for(word in vec){
  print(word)
}
```

For Loop Exercises

apply()

- Many questions that can be answered using a for loop can be also be evaluated using a R function in the `apply` family
- `apply()` will execute a function on every row or every column of a matrix

$$\texttt{apply}(\texttt{X}, \texttt{MARGIN}, \texttt{FUN}, \ldots)$$

- `X` is the matrix or array you would like to apply the function `FUN` to
- To apply `FUN` to the rows of `X`, $\texttt{MARGIN} = 1$, and to apply `FUN` to the columns of `X`, $\texttt{MARGIN} = 2$
- If additional arguments need to be passed to the function, they can simply be passed to apply

apply() Examples

- Calculate the column sums of matrix M

```
> M <- matrix(1:15, nrow=3, ncol=5)
> M[2,1] <- M[3,4] <- NA
> M
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]   NA    5    8   11   14
[3,]    3    6    9   NA   15
> apply(M, 2, sum)
[1] NA 15 24 NA 42
```

apply() Examples

- Calculate the column sums of matrix *M*

```
> M <- matrix(1:15, nrow=3, ncol=5)
> M[2,1] <- M[3,4] <- NA
> M
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]   NA    5    8   11   14
[3,]    3    6    9   NA   15
> apply(M, 2, sum)
[1] NA 15 24 NA 42
```

- Remove NA values and calculate column sums of matrix *M*

```
> apply(M, 2, sum, na.rm=TRUE)
[1]  4 15 24 21 42
```

lapply() and sapply()

- The lapply and sapply functions in R apply the specified function to each element of a list or vector
- lapply() always returns a list
- sapply() will return a vector or array if appropriate

```
x.list <- list(a = 1:5, b=6:10, c=11:15)

# save result in a list
lapply(x.list, mean)

# save result as a vector
sapply(x.list, mean)
```

User Defined Functions

- Basic syntax:

```
function_name <- function(_______){
# body of function
return(______)
}
```

  - First blank contains argument names (comma-separated)
  - Second blank is what the function outputs

- To use the function, one just calls the function and fills in the corresponding arguments:

```
function_name(______)
```

User Defined Functions: Input and Output

- Function arguments:
  - Can be of any type (eg., character, numeric, boolean, list, matrix, data frame, etc.)
  - Can have defaults

```
myfunction <- function(a, b=2){
return(paste0(a, " + ", b, ''*x"))
}
```

  - Function output:
    - Only one item can be returned, but that item can be of any type/structure.
    - Lists are useful for returning multiple things from one function.

User Defined Function Example

- Here is one example

```
expit <- function(x){
return(exp(x) / (1 + exp(x)))
}
expit(1)
expit(-2)
```

- User defined functions are useful when one wants to perform the same actions but with different numbers or other arguments
- Variables defined inside the function cannot be used outside of it

```
myfn <- function(x){
y <- 2 ## Can't reference this outside function
return(x + y)
}
```

apply() Exercises

Boolean Expressions

Definition (Boolean expressions)

Statements that are evaluated and return a Boolean value

```
## Examples:
3 < 5
3 > 5
3 == 3
3 == 5
```

- They are often used in flow control to assess whether a certain condition is met
- Used in conditional statements and while loops

Boolean Operators: Combining logical expressions

Definition (Logical negation)

Returns the opposite of the expression

```
!(3 < 5)
```

Boolean Operators: Combining logical expressions

Definition (Logical negation)

Returns the opposite of the expression

```
!(3 < 5)
```

Definition (Logical AND)

Returns TRUE only if all expressions are TRUE

```
(3 < 5) & (5 < 3)
```

Boolean Operators: Combining logical expressions

Definition (Logical negation)

Returns the opposite of the expression

```
!(3 < 5)
```

Definition (Logical AND)

Returns TRUE only if all expressions are TRUE

```
(3 < 5) & (5 < 3)
```

Definition (Logical OR)

Returns FALSE only if all expressions are FALSE

```
(3 < 5) | (5 < 3)
```

Boolean Operators: Combining logical expressions

Definition (Logical negation)

Returns the opposite of the expression

```
!(3 < 5)
```

Definition (Logical AND)

Returns TRUE only if all expressions are TRUE

```
(3 < 5) & (5 < 3)
```

Definition (Logical OR)

Returns FALSE only if all expressions are FALSE

```
(3 < 5) | (5 < 3)
```

- To combine Boolean expressions for a vector/matrix, use `any` and `all`

Conditional Statements

- With conditional statements, we can perform different actions contingent on which of several conditions are met, or not met
- Basic syntax:

```
if (______){
# commands 1
} else if (______){
# commands 2
} else {
# commands 3
}
```

- If the first condition is met, `commands 1` are executed
- If the first condition is not met, then we go to the second condition. If the second condition is met, `commands 2` are executed
- If the second condition is also not met, `commands 3` are executed
- You can chain together as many "else if" together as you wish

Conditional Statements Examples

- Basic if-else statement

```
num <- rnorm(1, 0, 5)
if (num < 0){
print("num is negative")
} else {
print("num is positive")
}
```

Conditional Statements Examples

- Basic if-else statement

```
num <- rnorm(1, 0, 5)
if (num < 0){
print("num is negative")
} else {
print("num is positive")
}
```

- Multilevel if-else statement

```
num <- runif(1)
if (num < 0.4){
print("less than 0.4")
} else if(num < 0.8){
print("between 0.4 and 0.8")
} else {
print("greater than 0.8")
}
```

While Loop

- With the for loop, we know exactly how many times the commands will be repeated
- With the while loop, there is a condition that must be met for the commands to be repeated
- The repetition continues until the condition is no longer met
- The basic syntax of a while loop:

```
while (______){
   # the commands to be repeated
}
```

- The blank is for a boolean expression

While Loop Example

- Here are some examples

```
i <- 1
while (i < 10){
  print(i)
  i <- i + 1
}

while ((i < 10) & (runif(1) < 0.5) ){
  print("heads")
  i <- i + 1
}
```

- The code being repeated needs to progress towards the condition being no longer true, or else the loop will continue infinitely

While Loop and Conditional Statement Exercises