## Advance Programming Lab(CS-3095)

# KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

## School of Computer Engineering

*2 Credit*

R Programming

# Lab Contents

| Sr # | Major and Detailed Coverage Area | Lab# |
|---|---|---|
| 1 | Vector | 4 |
| 2 | List | |
| 3 | Matrices | |
| 4 | Arrays | |

## Lab Goals

After the lab, you should:

- Be able to create new vectors from existing vectors
- Understand lengths, dimensions, and names
- Be able to create and manipulate matrices, lists and arrays

# Vectors

So far, you have used the colon operator, :, for creating sequences from one number to another, and the c function for concatenating values and vectors to create longer vectors.

To recap:

```
8.5:4.5 #sequence of numbers from 8.5 down to 4.5
[1] 8.5 7.5 6.5 5.5 4.5
```

```
c(1, 1:3, c(5, 8), 13) #values concatenated into single vector
[1] 1 1 2 3 5 8 13
```

The **vector** function creates a vector of a specified type and length. Each of the values in the result is zero, FALSE, or an empty string, or whatever the equivalent of “nothing” is:

```
vector("numeric", 5)      vector("complex ", 5)
[1] 0 0 0 0 0             [1] 0+0i 0+0i 0+0i 0+0i 0+0i
```

## Lab Work

vector(“logical", 5)
[1] ?

vector("character", 5)
[1] ?

vector("list", 5)
[1] ?

For convenience, wrapper functions exist for each type to save typing when creating vectors in this way. The following commands are equivalent to the previous ones:

```
numeric(5)
[1] 0 0 0 0 0
```

```
complex(5)
[1] 0+0i 0+0i 0+0i 0+0i 0+0i
```

```
character(5)
[1] "" "" "" "" ""
```

```
logical(5)
[1] FALSE FALSE FALSE FALSE FALSE
```

# Sequences

Beyond the colon operator, there are several functions for creating more general sequences. The **seq** function is the most general, and allows to specify sequences in many different ways. In practice, though, one should never need to call it, since there are three other specialist sequence functions that are faster and easier to use, covering specific use cases.

**seq.int** create a sequence from one number to another. With two inputs, it works exactly like the colon operator:

seq.int(3, 12) #same as 3:12

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

seq.int is slightly more general than :, since it lets you specify how far apart intermediate values should be:

seq.int(3, 12, 2) and the output is [1] 3 5 7 9 11

seq.int(0.1, 0.01, -0.01) and the output is [1] 0.10 0.09 0.08 0.07 0.06 0.05 0.04 0.03 0.02 0.01

## Sequences cont'd

**seq_len** creates a sequence from 1 up to its input, so seq_len(5) is just a clunkier way of writing 1:5. However, the function is extremely useful for situations when its input could be zero:

```
n <- 0
seq_len(n)
```

**seq_along** creates a sequence from 1 up to the length of its input.

```
pp <- c("Peter", "Piper", "picked", "a", "peck", "of", "pickled", "peppers")
for(i in seq_along(pp)) {
  print(pp[i])
}
```

# length

All vectors have a length, which tells how many elements they contain. This is a non-negative integer (yes, zero-length vectors are allowed), and value can be accessed with the length function. Missing values still count toward the length.

```
length(1:5)        length(c(TRUE, FALSE, NA))
[1] 5              [1] 3
```

One possible source of confusion is character vectors. With these, the length is the number of strings, not the number of characters in each string. For that, nchar to be used

```
sn <- c("Sheena", "leads", "Sheila", "needs")
length(sn)         nchar(sn)
[1] 4              [1] 6 5 6 5
```

## length cont’d

It is also possible to assign a new length to a vector, but this is an unusual thing to do, and probably indicates bad code. If you shorten a vector, the values at the end will be removed, and if you extend a vector, missing values will be added to the end.

```
poincare <- c(1, 0, 0, 0, 2, 0, 2, 0)

length(poincare) <- 3

poincare

[1] 1 0 0

length(poincare) <- 8

poincare

[1] 1 0 0 NA NA NA NA NA
```

## Names

A great feature of R’s vectors is that each element can be given a name. Labeling the elements can often make the code much more readable. Name can be specified when a vector is created in the form name = value. If the name of an element is a valid variable name, it doesn’t need to be enclosed in quotes. Some elements of a vector can be named and leave others blank.

```
x = c(apple = 1, banana = 2, "kiwi fruit" = 3, 4)
print(x)
```

Element names can be added to a vector after its creation using the names function:

```
x <- 1:4
names(x) <- c("apple", "bananas", "kiwi fruit", "")
print(x)
```

names function can also be used to retrieve the names of a vector i.e. names(x)

# Indexing Vectors

Oftentimes we may want to access only part of a vector, or perhaps an individual element. This is called **indexing** and is accomplished with square brackets, []. (Some also call it **subsetting** or **subscripting** or **slicing**. All these terms refer to the same thing.) R has a very flexible system that gives us several choices of index:

- Positive Index: Passing a vector of positive numbers returns the slice of the vector containing the elements at those locations. The first position is 1 (not 0, as in some other languages).
  ```
  s = c("aa", "bb", "cc", "dd", "ee")
  s[3]
  ```
- Negative Index: Passing a vector of negative numbers returns the slice of the vector containing the elements everywhere except at those locations.
  ```
  s = c("aa", "bb", "cc", "dd", "ee")
  s[-3]
  ```
- Out-of-Range Index:
  ```
  s = c("aa", "bb", "cc", "dd", "ee")
  s[10]
  ```

## Vectors Arithmetic

Arithmetic operations of vectors are performed member-by-member, i.e., member-wise.

For example, suppose two vectors a and b.

```
a = c(1, 3, 5, 7)
b = c(1, 2, 4, 8)
```

Then, if we multiply a by 5, we would get a vector with each of its members multiplied by 5.

```
5 * a
[1]  5 15 25 35
```

And if we add a and b together, the sum would be a vector whose members are the sum of the corresponding members from a and b.

```
a + b
[1]  2  5  9 15
```

Similarly for subtraction, multiplication and division, we get new vectors via member-wise operations.

# Combining Vectors

Vectors can be combined via the function c. For examples, the following two vectors n and s are combined into a new vector containing elements from both vectors.

```
n = c(2, 3, 5)
s = c("aa", "bb", "cc", "dd", "ee")
c(n, s)
[1] "2"  "3"  "5"  "aa" "bb" "cc" "dd" "ee“
```

**Value Coercion**

In the code snippet above, notice how the numeric values are being coerced into character strings when the two vectors are combined. This is necessary so as to maintain the same primitive data type for members in the same vector.

## Vector Recycling

So far, all the vectors that we have added together have been the same length. Wonder, “What happens if we try to do arithmetic on vectors of different lengths?”

If we try to add a single number to a vector, then that number is added to each element of the vector:

```
1:5 + 1
[1] 2 3 4 5 6
1 + 1:5
[1] 2 3 4 5 6
```

When adding two vectors together, R will recycle elements in the shorter vector to match the longer one:

```
1:5 + 1:15 # length of the longer vector is a multiple of the length of the shorter one
[1] 2 4 6 8 10 7 9 11 13 15 12 14 16 18 20
```

## Vector Recycling cont'd and Vector Repetition

If the length of the longer vector isn't a multiple of the length of the shorter one, a warning will be given:

```
1:5 + 1:7
Warning: longer object length is not a multiple of shorter object length
[1] 2 4 6 8 10 7 9
```

It must be stressed that just because we can do arithmetic on vectors of different lengths, it doesn't mean that we should. Adding a scalar value to a vector is okay, but otherwise we are liable to get ourselves confused. It is much better to explicitly create equal-length vectors before we operate on them.

The rep function is very useful for this task, letting us create a vector with repeated elements:

```
rep(1:5, 3)
## [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```
rep(1:5, each = 3)
[1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

## List

Lists are the R objects which contain elements of different types like – numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using list() function.

**Creating a List:**

Following is an example to create a list containing strings, numbers, vectors and a logical values.

```
# Create a list containing strings, numbers, vectors and a logical  values.
list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)
```

# List cont’d

**Naming List Elements:**

The list elements can be given names and they can be accessed using these names.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
  list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Show the list.
print(list_data)
```

## Accessing List Elements

Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Access the first element of the list.
print(list_data[1])

# Access the third element. As it is also a list, all its elements will be printed.
print(list_data[3])

# Access the list element using the name of the element.
print(list_data$A_Matrix)
```

## Manipulating List Elements

We can add, delete and update list elements as shown below. We can add and delete elements only at the end of a list. But we can update any element.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),   list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Add element at the end of the list.
list_data[4] <- "New element"
print(list_data[4])

# Remove the last element.
list_data[4] <- NULL

# Print the 4th Element.
print(list_data[4])

# Update the 3rd Element.
list_data[3] <- "updated element"
print(list_data[3])
```

## Merging Lists

You can merge many lists into one list by placing all the lists inside one list() function.

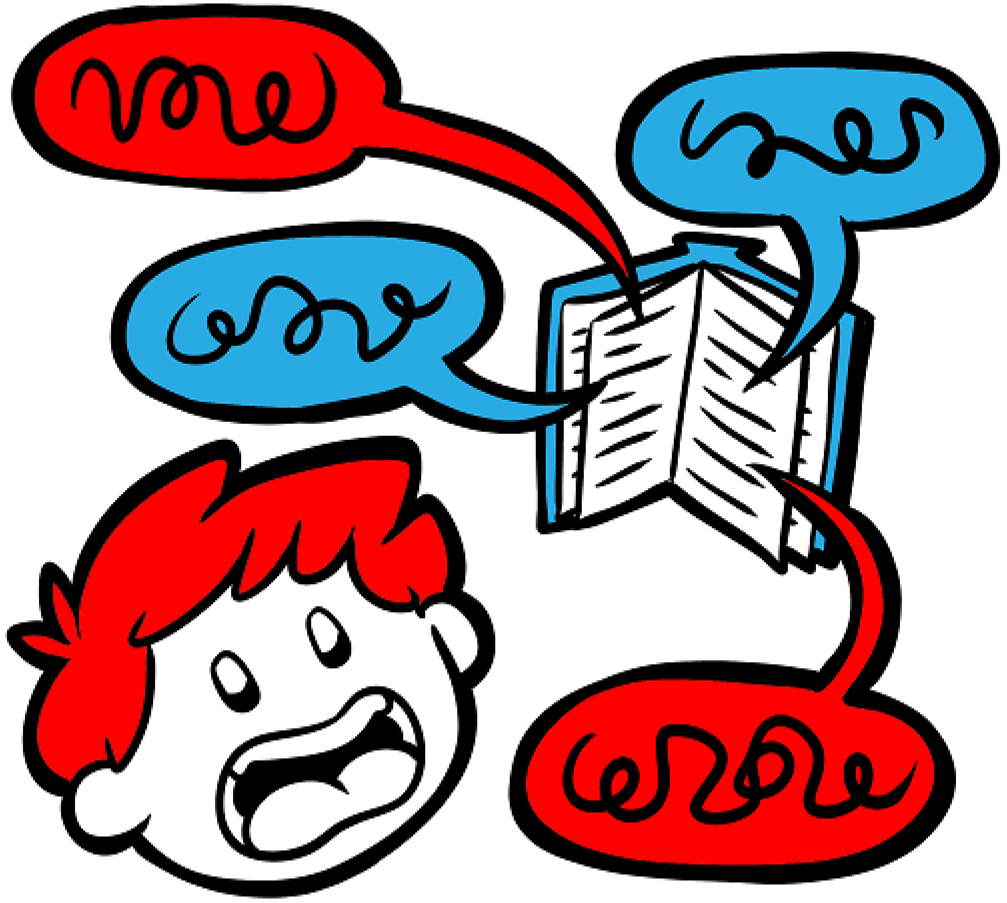
```
# Create two lists.
list1 <- list(1,2,3)
list2 <- list("Sun","Mon","Tue")

# Merge the two lists.
merged.list <- c(list1,list2)

# Print the merged list.
print(merged.list)
```

## Converting List to Vector

A list can be converted to a vector so that the elements of the vector can be used for further manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors. To do this conversion, we use the unlist() function. It takes the list as input and produces a vector.

```
# Create lists.
list1 <- list(1:5)
print(list1)

list2 <-list(10:14)
print(list2)

# Convert the lists to vectors.
v1 <- unlist(list1)
v2 <- unlist(list2)

print(v1)
print(v2)

# Now add the vectors
result <- v1+v2
print(result)
```

## Too Much?



## Matrices

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations. A Matrix is created using the matrix() function.

**Syntax**: matrix(data, nrow, ncol, byrow, dimnames)

Following is the description of the parameters used –

- data is the input vector which becomes the data elements of the matrix.
- nrow is the number of rows to be created.
- ncol is the number of columns to be created.
- byrow is a logical clue. If TRUE then the input vector elements are arranged by row.
- dimname is the names assigned to the rows and columns.

# Matrices cont'd

```
# Elements are arranged sequentially by row.
M <- matrix(c(3:14), nrow = 4, byrow = TRUE)
print(M)

# Elements are arranged sequentially by column.
N <- matrix(c(3:14), nrow = 4, byrow = FALSE)
print(N)

# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))
print(P)
```

# Accessing Elements of a Matrix

Elements of a matrix can be accessed by using the column and row index of the element.

```
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

# Create the matrix.
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))

# Access the element at 3rd column and 1st row.
print(P[1,3])

# Access the element at 2nd column and 4th row.
print(P[4,2])
```

1. Write the code to access the 2nd row
2. Write the code to access the 2nd column

# Matrix Computations

Various mathematical operations are performed on the matrices using the R operators. The result of the operation is also a matrix.  The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

**Matrix Addition, Subtraction, Multiplication and Division:**

```
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2) # creating 2X3 matrix
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2) # creating 2X3 matrix
```

```
# Add the matrices.
result <- matrix1 + matrix2
cat("Result of addition","\n")
print(result)
```

```
# Subtract the matrices
result <- matrix1 - matrix2
cat("Result of subtraction","\n")
print(result)
```

```
# Multiply the matrices
result <- matrix1 * matrix2
cat("Result of subtraction","\n")
print(result)
```

```
# Divide the matrices.
result <- matrix1 / matrix2
cat("Result of multiplication","\n")
print(result)
```

1. Write the code to transpose the matrix
2. Write the code to add the transpose of the matrix with original matrix

# Arrays

Arrays are the R data objects which can store data in more than two dimensions. For example – If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. An array is created using the array() function. It takes vectors as input and uses the values in the dim parameter to create an array.

**Example:**

The following example creates an array of two 3x3 matrices each with 3 rows and 3 columns.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2))
print(result)
```

## Arrays – Naming Rows and Columns

We can give names to the rows, columns and matrices in the array by using the dimnames parameter.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames = list(row.names,column.names, matrix.names))
print(result)
```

## Accessing Array Elements

```
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2))

# Print the third row of the second matrix of the array.
print(result[3,,2])

# Print the element in the 1st row and 3rd column of the 1st matrix.
print(result[1,3,1])

# Print the 2nd Matrix.
print(result[,,2])
```

# Manipulating Array Elements

As array is made up matrices in multiple dimensions, the operations on elements of array are carried out by accessing elements of the matrices.

```
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
array1 <- array(c(vector1,vector2),dim = c(3,3,2))

vector3 <- c(9,1,0)
vector4 <- c(6,0,11,3,14,1,2,6,9)
array2 <- array(c(vector1,vector2),dim = c(3,3,2))

# create matrices from these arrays.
matrix1 <- array1[,,2]
matrix2 <- array2[,,2]

# Add the matrices.
result <- matrix1+matrix2
print(result)
```

## Thank You
## End of Lab 4

# Assignment

1. Write an R-script to initialize 5 elements to a vector and then find out square root of each and every element.
2. Write an R-script to initialize two vectors and then check both vectors are equal or not?
3. Write an R-script to initialize two vectors with 1s and 0s. Perform AND operation with two vectors and display the result.
4. Write an R-script to enter the elements of a vector from the keyboard and sort the elements of vector in ascending order.
5. Write an R-script to initialize two vectors and then evaluate the 1st vector raised to the power of 2nd vector.
6. Write an R-script to initialize a vector and then find out minimum value and maximum value, also evaluate the sum of all the elements.
7. Write an R-script to initialize a vector and then search a specific element from that vector.
8. Write an R-script to create a list with different types of data set. Now display each data set separately according to the data type.

# Assignment cont'd

9. Write an R-script to create a list having vector, matrix and a list. Now display only the 2nd data set of the list.
10. Write an R-script to add a new data set to the previous list and also remove the 2nd data set from that list.
11. Write an R-script to create two lists- one contains the integers from 1 to 5 and another contains the name of 5 months. Now merge two lists and display that.
12. Write an R-script to create a 4*3 matrix. Now display the elements of row1, row3 and column2 of that matrix.
13. Write an R-script to create two matrix and then perform addition, subtraction, multiplication and division of them.
14. Write an R-script to create a 3*3 matrix and update that matrix by adding 4 to each and every element, also display the updated matrix.
15. Write an R-script to set those elements of the created matrix to 0, whose value are less than 5.
16. Write an R-script to check the given matrix is symmetric matrix or not?

# Assignment cont'd

17. Write an R-script to create a matrix and evaluate the sum of the elements row-wise.
18. Write an R-script to create an array having 3 dimensions. Now retrieve the elements of 2nd row of 3rd matrix.
19. Write an R-script to create an array having 3 dimensions. Now calculate the sum of the rows across all the matrices.
20. The nth triangular number is given by n * (n + 1) / 2. Create a sequence of the first 20 triangular numbers. R has a built-in constant, letters, that contains the lowercase letters of the Roman alphabet. Name the elements of the vector that you just created with the first 20 letters of the alphabet. Select the triangular numbers where the name is a vowel
21. The **diag** function has several uses, one of which is to take a vector as its input and create a square matrix with that vector on the diagonal. Create a 21-by-21 matrix with the sequence 10 to 0 to 11 (i.e., 11, 10, … , 1, 0, 1, …, 11)
22. Write an R-script to demonstrate Wilkinson matrix