UNIT-2

Lists:

- ✓ In contrast to a vector, in which all elements must be of the same mode, R's list structure can combine objects of different types.
- ✓ For those familiar with Python, an R list is similar to a Python dictionary or, for that matter, a Perl hash.

Creating Lists:

- ✓ A list in R can contain many different data types inside it.
- ✓ A list is a collection of data which is ordered and changeable.
- ✓ To create a list, use the list() function.

Example:

```
# List of characters/strings
  thislist <- list("apple", "banana", "cherry")
# Print the list
  Print(thislist)
```

Output:

```
[[1]]              [[2]]              [[3]]
[1] "apple"        [1] "banana"       [1] "cherry"
```

General List Operations:

- ✓ Now that you've seen a simple example of creating a list, let's look at how to access and work with lists.

Example(simple list):

```
j <- list(name="Joe", salary=55000, union=T)
print(j)
```

Output:

```
$name        $salary      $union
[1] "Joe"    [1] 55000    [1] TRUE
```

i. List Indexing
ii. Adding and Deleting List Elements
iii. Getting the Size of a List

List Indexing:

- ✓ We can refer to list components by their numerical indices, treating the list as a vector.
- ✓ So, there are three ways to access an individual component of a list and return it in the data type.
  - lst_name$individual_element
  - lst_name[["individual_element "]]
  - lst_name[[i]],   where i is the index of individual_element within lst_name

Example:

```
> j$salary
        [1] 55000
> j[["salary"]]
        [1] 55000
> j[[2]]
        [1] 55000
```

Adding and Deleting List Elements:

- ✓ The operations of adding and deleting list elements arise in a surprising number of contexts.

  Example:

  Before adding:

```
> z <- list(a="abc",b=12)
> z
$a
[1] "abc"
$b
[1] 12
```

  After adding:

```
> z$c <- "sailing" # add a c component
> z
$a
[1] "abc"
$b
[1] 12
$c
[1] "sailing"
```

Getting the Size of a List:

- ✓ Since a list is a vector, you can obtain the number of components in a list via length().
  Example:

```
> length(j)
[1] 3
```

Accessing List Components and Values:

- ✓ If the components in a list do have tags, as is the case with name, salary, and union for j in simple list, you can obtain them via names():

```
> names(j)
[1] "name"    "salary"    "union"
```

- ✓ To obtain the values, use unlist():

```
> ulj <- unlist(j)
> ulj
Name        salary          union
"Joe"            "55000"    "TRUE"
> class(ulj)
[1] "character"
```

Applying Functions to Lists:

- ✓ Two functions are handy for applying functions to lists: lapply and sapply

Using the lapply() and sapply() Functions:

- o The function lapply() (for list apply) works like the matrix apply() function, calling the specified function on each component of a list (or vector coerced to a list) and returning another list. Here's an example:

```
> lapply(list(1:3,25:29),median)
[[1]]
[1] 2
[[2]]
[1] 27
```

- o R applied median() to 1:3 and to 25:29, returning a list consisting of 2 and 27.
- o In some cases, such as the example here, the list returned by lapply() could be simplified to a vector or matrix.
- o This is exactly what sapply() (for simplified [l]apply) does. Here's an example:

```
> sapply(list(1:3,25:29),median)
```

[1] 2 27

- Using sapply(), rather than applying the function directly, gave us the desired matrix form in the output.

Extended Example: Text Concordance:

- ✓ We'll write a function called findwords() that will determine which words are in a text file and compile a list of the locations of each word's occurrences in the text.
- ✓ Suppose our input file, testconcord.txt, has the following contents.

  "the here means that the first item in this line of output is item in this case our output consists of only one line and one item so this is redundant but this notation helps to read voluminous output that consists of many items spread over many lines for example if there were two rows of output with six items per row the second row would be labeled"

- ✓ Here is an excerpt from the list that is returned when our function findwords() is called on this file:

```
> findwords("testconcorda.txt")
Read 68 items
$the
[1] 1 5 63
$here
[1] 2
$means
[1] 3
$that
[1] 4 40
$first
[1] 6
$item
[1] 7 14 27
```

Recursive Lists:

- ✓ Lists can be recursive, meaning that you can have lists within lists. Here's an example:

```
> b <- list(u = 5, v = 12)
> c <- list(w = 13)
> a <- list(b,c)
```

```
> a
[[1]]
[[1]] $u
[1] 5
[[1]] $v
[1] 12
[[2]]
[[2]] $w
[1] 13
> length(a)
[1] 2
```

Data Frames:

- ✓ On an intuitive level, a data frame is like a matrix, with a two-dimensional rows-and□columns structure.
- ✓ In this sense, just as lists are the heterogeneous analogs of vectors in one dimension, data frames are the heterogeneous analogs of matrices for two-dimensional data.

Creating Data Frames:

- ✓ Data Frames are data displayed in a format as a table.
- ✓ Data Frames can have different types of data inside it. While the first column can be character, the second and third can be numeric or logical. However, each column should have the same type of data.
- ✓ Use the data.frame() function to create a data frame:

  Example:

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
> d            # matrix-like viewpoint
               kids   ages
          1    jack   12
          2    jill   10
```

Merging Data Frames:

- ✓ In the relational database world, one of the most important operations is that of a join, in which two tables can be combined according to the values of a common variable.
- ✓ In R, two data frames can be similarly combined using the merge() function.

Example:

```
>d1
        kids   ages
   1    aju    19
   2    alan   10
>d2
        kids   height
   1    aju    198
   2    alan   108
> d <- merge(d1,d2)
>d
        kids   ages   height
   1    aju    19     198
   2    alan   10     108
```

Applying Functions to Data Frames:

- ✓ As with lists, you can use the lapply and sapply functions with data frames.

  Using lapply() and sapply() on Data Frames:

```
>d
        kids   ages
   1    jack   12
   2    jill   10
> dl <- lapply(d,sort)
> dl
$kids
[1] "Jack" "Jill"
$ages
[1] 10 12
```

- ✓ And use the cbind() function to combine two or more data frames in R horizontally.
- ✓ Use the rbind() function to combine two or more data frames in R vertically.
- ✓ Use the length() function to find the number of columns in a Data Frame (similar to ncol()).

Factors and Tables:

- ✓ Factors are used to categorize data. Examples of factors are:
  - Demography: Male/Female
  - Music: Rock, Pop, Classic, Jazz
  - Training: Strength, Stamina

To create a factor, use the `factor()` function and add a vector as argument:

Example:

```
# Create a factor
music_genre <-
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"))

# Print the factor
music_genre
```

Output:

```
[1] Jazz   Rock   Classic   Classic   Pop   Jazz   Rock   Jazz
Levels: Classic   Jazz   Pop   Rock
```

Common Functions used with Factors:

- ✓ With factors, we have yet another member of the family of apply functions, tapply.
- ✓ We'll look at that function, as well as two other functions commonly used with factors: split() and by().

  The tapply() Function:
  - The operation performed by tapply() is to (temporarily) split x into groups, each group corresponding to a level of the factor (or a combination of levels of the factors in the case of multiple factors), and then apply g() to the resulting subvectors of x.
  - Here's a little example:

```
> ages <- c(25,26,55,37,21,42)
> affils <- c("R","D","D","R","U","D")
> tapply(ages,affils,mean)
D    R    U
41   31   21
```

The Split() Function:

- In contrast to tapply(), which splits a vector into groups and then applies a specified function on each group, split() stops at that first stage, just forming the groups.

  Example:

```
> d
  gender age income over25
1      M  47  55000      1
2      M  59  88000      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0
> split(d$income,list(d$gender,d$over25))
$F.0
[1] 32450 45650

$M.0
numeric(0)

$F.1
[1] 123000

$M.1
[1] 55000 88000 76500
```

Working with Tables:

To begin exploring R tables, consider this example:

> u <- c(22,8,33,6,8,29,-2)

> fl <- list(c(5,12,13,12,13,5,13),c("a","bc","a","a","bc","a","a"))

> tapply(u,fl,length)

a b c

5 2 NA

12 1 1

13 2 1