Vector

A vector is a sequence of data elements of the same basic type. Members in a vector are officially called components. Nevertheless, we will just call them members in this site.

Here is a vector containing three numeric values 2, 3 and 5.

```
> c(2, 3, 5)
[1] 2 3 5
```

And here is a vector of logical values.

```
> c(TRUE, FALSE, TRUE, FALSE, FALSE)
[1]  TRUE FALSE  TRUE FALSE FALSE
```

A vector can contain character strings.

```
> c("aa", "bb", "cc", "dd", "ee")
[1] "aa" "bb" "cc" "dd" "ee"
```

Incidentally, the number of members in a vector is given by the length function.

```
> length(c("aa", "bb", "cc", "dd", "ee"))
[1] 5
```

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 Arithmetics

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

For example, suppose we have 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 memberwise operations.

```
> a - b
[1]  0  1  1 -1

> a * b
[1]  1  6 20 56

> a / b
[1] 1.000 1.500 1.250 0.875
```

Recycling Rule

If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. For example, the following vectors u and v have different lengths, and their sum is computed by recycling values of the shorter vector u.

```
> u = c(10, 20, 30)

> v = c(1, 2, 3, 4, 5, 6, 7, 8, 9)

> u + v
[1] 11 22 33 14 25 36 17 28 39
```

Vector Index

We retrieve values in a vector by declaring an index inside a single square bracket "[]" operator.

For example, the following shows how to retrieve a vector member. Since the vector index is 1-based, we use the index position 3 for retrieving the third member.

```
> s = c("aa", "bb", "cc", "dd", "ee")
> s[3]
[1] "cc"
```

Unlike other programming languages, the square bracket operator returns more than just individual members. In fact, the result of the square bracket operator is another vector, and s[3] is a vector slice containing a single member "cc".

Negative Index

If the index is negative, it would strip the member whose position has the same absolute value as the negative index. For example, the following creates a vector slice with the third member removed.

```
> s[-3]
[1] "aa" "bb" "dd" "ee"
```

Out-of-Range Index

If an index is out-of-range, a missing value will be reported via the symbol NA.

```
> s[10]
[1] NA
```

Numeric Index Vector

A new vector can be sliced from a given vector with a numeric index vector, which consists of member positions of the original vector to be retrieved.

Here it shows how to retrieve a vector slice containing the second and third members of a given vector s.

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

Duplicate Indexes

The index vector allows duplicate values. Hence the following retrieves a member twice in one operation.

```
> s[c(2, 3, 3)]
[1] "bb" "cc" "cc"
```

Out-of-Order Indexes

The index vector can even be out-of-order. Here is a vector slice with the order of first and second members reversed.

```
> s[c(2, 1, 3)]
[1] "bb" "aa" "cc"
```

Range Index

To produce a vector slice between two indexes, we can use the colon operator ":". This can be convenient for situations involving large vectors.

```
> s[2:4]
[1] "bb" "cc" "dd"
```

More information for the colon operator is available in the R documentation.

```
> help(":")
```

Logical Index Vector

A new vector can be sliced from a given vector with a logical index vector, which has the same length as the original vector. Its members are TRUE if the corresponding members in the original vector are to be included in the slice, and FALSE if otherwise.

For example, consider the following vector s of length 5.

```
> s = c("aa", "bb", "cc", "dd", "ee")
```

To retrieve the the second and fourth members of s, we define a logical vector L of the same length, and have its second and fourth members set as TRUE.

```
> L = c(FALSE, TRUE, FALSE, TRUE, FALSE)
> s[L]
[1] "bb" "dd"
```

The code can be abbreviated into a single line.

```
> s[c(FALSE, TRUE, FALSE, TRUE, FALSE)]
[1] "bb" "dd"
```

Named Vector Members

We can assign names to vector members.

For example, the following variable v is a character string vector with two members.

```
> v = c("Mary", "Sue")
> v
[1] "Mary" "Sue"
```

We now name the first member as First, and the second as Last.

```
> names(v) = c("First", "Last")
> v
 First   Last
"Mary"  "Sue"
```

Then we can retrieve the first member by its name.

```
> v["First"]
[1] "Mary"
```

Furthermore, we can reverse the order with a character string index vector.

```
> v[c("Last", "First")]
  Last  First
 "Sue" "Mary"
```

Matrix

A matrix is a collection of data elements arranged in a two-dimensional rectangular layout. The following is an example of a matrix with 2 rows and 3 columns.

$$A = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

We reproduce a memory representation of the matrix in R with the matrix function. The data elements must be of the same basic type.

```
> A = matrix(
+   c(2, 4, 3, 1, 5, 7),      # the data elements
+   nrow=2,                   # number of rows
+   ncol=3,                   # number of columns
+   byrow = TRUE)             # fill matrix by rows

> A                           # print the matrix
     [,1] [,2] [,3]
[1,]    2    4    3
[2,]    1    5    7
```

An element at the mth row, nth column of A can be accessed by the expression A[m, n].

```
> A[2, 3]          # element at 2nd row, 3rd column
[1] 7
```

The entire mth row A can be extracted as A[m,].

```
> A[2, ]           # the 2nd row
[1] 1 5 7
```

Similarly, the entire nth column A can be extracted as A[,n].

```
> A[ ,3]           # the 3rd column
[1] 3 7
```

We can also extract more than one rows or columns at a time.

```
> A[ ,c(1,3)]      # the 1st and 3rd columns
     [,1] [,2]
[1,]    2    3
[2,]    1    7
```

If we assign names to the rows and columns of the matrix, than we can access the elements by names.

```
> dimnames(A) = list(
+   c("row1", "row2"),          # row names
+   c("col1", "col2", "col3"))  # column names

> A                             # print A
     col1 col2 col3
row1    2    4    3
row2    1    5    7

> A["row2", "col3"]             # element at 2nd row, 3rd column
[1] 7
```

Matrix Construction

There are various ways to construct a matrix. When we construct a matrix directly with data elements, the matrix content is filled along the column orientation by default. For example, in the following code snippet, the content of B is filled along the columns consecutively.

```
> B = matrix(
+   c(2, 4, 3, 1, 5, 7),
+   nrow=3,
+   ncol=2)

> B             # B has 3 rows and 2 columns
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
```

Transpose

We construct the transpose of a matrix by interchanging its columns and rows with the function t .

```
> t(B)          # transpose of B
     [,1] [,2] [,3]
[1,]    2    4    3
[2,]    1    5    7
```

Combining Matrices

The columns of two matrices having the same number of rows can be combined into a larger matrix.

For example, suppose we have another matrix C also with 3 rows.

```
> C = matrix(
+   c(7, 4, 2),
+   nrow=3,
+   ncol=1)

> C              # C has 3 rows
     [,1]
[1,]    7
[2,]    4
[3,]    2
```

Then we can combine the columns of B and C with cbind.

```
> cbind(B, C)
     [,1] [,2] [,3]
[1,]    2    1    7
[2,]    4    5    4
[3,]    3    7    2
```

Similarly, we can combine the rows of two matrices if they have the same number of columns with the rbind function.

```
> D = matrix(
+   c(6, 2),
+   nrow=1,
+   ncol=2)

> D             # D has 2 columns
     [,1] [,2]
[1,]    6    2

> rbind(B, D)
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
[4,]    6    2
```

Deconstruction

We can deconstruct a matrix by applying the c function, which combines all column vectors into one.

```
> c(B)
[1] 2 4 3 1 5 7
```

List

A list is a generic vector containing other objects.

For example, the following variable x is a list containing copies of three vectors n, s, b, and a numeric value 3.

```
> n = c(2, 3, 5)

> s = c("aa", "bb", "cc", "dd", "ee")

> b = c(TRUE, FALSE, TRUE, FALSE, FALSE)

> x = list(n, s, b, 3)                    # x contains copies of n, s, b
```

List Slicing

We retrieve a list slice with the single square bracket "[]" operator. The following is a slice containing the second member of x, which is a copy of s.

```
> x[2]
[[1]]
[1] "aa" "bb" "cc" "dd" "ee"
```

With an index vector, we can retrieve a slice with multiple members. Here a slice containing the second and fourth members of x.

```
> x[c(2, 4)]
[[1]]
[1] "aa" "bb" "cc" "dd" "ee"

[[2]]
[1] 3
```

Member Reference

In order to reference a list member directly, we have to use the double square bracket "[[]]" operator.

The following object x[[2]] is the second member of x. In other words, x[[2]] is a copy of s, but is not a slice containing s or its copy.

```
> x[[2]]
[1] "aa" "bb" "cc" "dd" "ee"
```

We can modify its content directly.

```
> x[[2]][1] = "ta"
> x[[2]]
[1] "ta" "bb" "cc" "dd" "ee"
```

```
> s
[1] "aa" "bb" "cc" "dd" "ee"   # s is unaffected
```

Named List Members

We can assign names to list members, and reference them by names instead of numeric indexes.

For example, in the following, v is a list of two members, named "bob" and "john".

```
> v = list(bob=c(2, 3, 5), john=c("aa", "bb"))
> v
$bob
[1] 2 3 5

$john
[1] "aa" "bb"
```

List Slicing

We retrieve a list slice with the single square bracket "[]" operator. Here is a list slice containing a member of v named "bob".

```
> v["bob"]
$bob
[1] 2 3 5
```

With an index vector, we can retrieve a slice with multiple members. Here is a list slice with both members of v. Notice how they are reversed from their original positions in v.

```
> v[c("john", "bob")]
$john
[1] "aa" "bb"

$bob
[1] 2 3 5
```

Member Reference

In order to reference a list member directly, we have to use the double square bracket "[[]]" operator. The following references a member of v by name.

```
> v[["bob"]]
[1] 2 3 5
```

A named list member can also be referenced directly with the "$" operator in lieu of the double square bracket operator.

```
> v$bob
[1] 2 3 5
```

Search Path Attachment

We can attach a list to the R search path and access its members without explicitly mentioning the list. It should to be detached for cleanup.

```
> attach(v)
> bob
[1] 2 3 5

> detach(v)
```

Data Frame

A data frame is used for storing data tables. It is a list of vectors of equal length. For example, the following variable df is a data frame containing three vectors n, s, b.

```
> n = c(2, 3, 5)
> s = c("aa", "bb", "cc")
> b = c(TRUE, FALSE, TRUE)
> df = data.frame(n, s, b)       # df is a data frame
```

Build-in Data Frame

We use built-in data frames in R for our tutorials. For example, here is a built-in data frame in R, called mtcars.

```
> mtcars
              mpg cyl disp  hp drat   wt ...
Mazda RX4     21.0   6  160 110 3.90 2.62 ...
Mazda RX4 Wag 21.0   6  160 110 3.90 2.88 ...
Datsun 710    22.8   4  108  93 3.85 2.32 ...
              ............
```

The top line of the table, called the header, contains the column names. Each horizontal line afterward denotes a data row, which begins with the name of the row, and then followed by the actual data. Each data member of a row is called a cell.

To retrieve data in a cell, we would enter its row and column coordinates in the single square bracket "[]" operator. The two coordinates are separated by a comma. In other words, the coordinates begins with row position, then followed by a comma, and ends with the column position. The order is important.

Here is the cell value from the first row, second column of mtcars.

```
> mtcars[1, 2]
[1] 6
```

Moreover, we can use the row and column names instead of the numeric coordinates.

```
> mtcars["Mazda RX4", "cyl"]
[1] 6
```

Lastly, the number of data rows in the data frame is given by the nrow function.

```
> nrow(mtcars)    # number of data rows
[1] 32
```

And the number of columns of a data frame is given by the ncol function.

```
> ncol(mtcars)     # number of columns
[1] 11
```

Further details of the mtcars data set is available in the R documentation.

```
> help(mtcars)
```

Preview

Instead of printing out the entire data frame, it is often desirable to preview it with the head function beforehand.

```
> head(mtcars)
               mpg cyl disp  hp drat   wt ...
Mazda RX4     21.0   6  160 110 3.90 2.62 ...  ............
```

Data Frame Column Vector

We reference a data frame column with the double square bracket "[[]]" operator.

For example, to retrieve the ninth column vector of the built-in data set mtcars, we write mtcars[[9]].

```
> mtcars[[9]]
 [1]  1 1 1 0 0 0 0 0 0 0 0 ...
```

We can retrieve the same column vector by its name.

```
> mtcars[["am"]]
 [1]  1 1 1 0 0 0 0 0 0 0 0 ...
```

We can also retrieve with the "$" operator in lieu of the double square bracket operator.

```
> mtcars$am
 [1]  1 1 1 0 0 0 0 0 0 0 0 ...
```

Yet another way to retrieve the same column vector is to use the single square bracket "[]" operator. We prepend the column name with a comma character, which signals a wildcard match for the row position.

```
> mtcars[,"am"]
 [1]  1 1 1 0 0 0 0 0 0 0 0 ...
```

Data Frame Column Slice

We retrieve a data frame column slice with the single square bracket "[]" operator.

Numeric Indexing

The following is a slice containing the first column of the built-in data set mtcars.

```
> mtcars[1]
                    mpg
Mazda RX4          21.0
Mazda RX4 Wag      21.0
Datsun 710         22.8
             ............
```

Name Indexing

We can retrieve the same column slice by its name.

```
> mtcars["mpg"]
                   mpg
Mazda RX4         21.0
Mazda RX4 Wag     21.0
Datsun 710        22.8
            ............
```

To retrieve a data frame slice with the two columns mpg and hp, we pack the column names in an index vector inside the single square bracket operator.

```
> mtcars[c("mpg", "hp")]
                  mpg  hp
Mazda RX4         21.0 110
Mazda RX4 Wag     21.0 110
Datsun 710        22.8  93
            ............
```

Data Frame Row Slice

We retrieve rows from a data frame with the single square bracket operator, just like what we did with columns. However, in additional to an index vector of row positions, we append an extra comma character. This is important, as the extra comma signals a wildcard match for the second coordinate for column positions.

Numeric Indexing

For example, the following retrieves a row record of the built-in data set mtcars. Please notice the extra comma in the square bracket operator, and it is not a typo. It states that the 1974 Camaro Z28 has a gas mileage of 13.3 miles per gallon, and an eight cylinder 245 horse power engine, ..., etc.

```
> mtcars[24,]
            mpg cyl disp  hp drat   wt  ...
Camaro Z28 13.3   8  350 245 3.73 3.84  ...
```

To retrieve more than one rows, we use a numeric index vector.

```
> mtcars[c(3, 24),]
            mpg cyl disp  hp drat   wt  ...
Datsun 710 22.8   4  108  93 3.85 2.32  ...
Camaro Z28 13.3   8  350 245 3.73 3.84  ...
```

Name Indexing

We can retrieve a row by its name.

```
> mtcars["Camaro Z28",]
           mpg cyl disp  hp drat   wt  ...
Camaro Z28 13.3   8  350 245 3.73 3.84  ...
```

And we can pack the row names in an index vector in order to retrieve multiple rows.

```
> mtcars[c("Datsun 710", "Camaro Z28"),]
           mpg cyl disp  hp drat   wt  ...
Datsun 710 22.8   4  108  93 3.85 2.32  ...
Camaro Z28 13.3   8  350 245 3.73 3.84  ...
```

Logical Indexing

Lastly, we can retrieve rows with a logical index vector. In the following vector L, the member value is TRUE if the car has automatic transmission, and FALSE if otherwise.

```
> L = mtcars$am == 0
> L
 [1]   FALSE FALSE FALSE  TRUE ...
```

Here is the list of vehicles with automatic transmission.

```
> mtcars[L,]
                    mpg cyl  disp  hp drat    wt  ...
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215  ...
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440  ...
               ............
```

And here is the gas mileage data for automatic transmission.

```
> mtcars[L,]$mpg
 [1] 21.4 18.7 18.1 14.3 24.4 ...
```

Data Import

It is necessary to import the sample textbook data into R before you start working on your homework.

Excel File

Quite often, the sample data is in Excel format, and needs to be imported into R prior to use. For this, we use the read.xls function from the gdata package. It reads from an Excel spreadsheet and returns a data frame. The following shows how to load an Excel spreadsheet named "mydata.xls". As the package is not in the core R library, it has to be installed and loaded into the R workspace.

```
> library(gdata)                  # load the gdata package
> help(read.xls)                  # documentation
> mydata = read.xls("mydata.xls")  # read from first sheet
```

Minitab File

If the data file is in Minitab Portable Worksheet format, it can be opened with the read.mtp function from the foreign package. It returns a list of components in the Minitab worksheet.

```
> library(foreign)                 # load the foreign package
> help(read.mtp)                   # documentation
> mydata = read.mtp("mydata.mtp")  # read from .mtp file
```

SPSS File

For the data files in SPSS format, it can be opened with the read.spss function from the foreign package. There is a "to.data.frame" option for choosing whether a data frame is to be returned.

```
> library(foreign)                 # load the foreign package
> help(read.spss)                  # documentation
> mydata = read.spss("myfile", to.data.frame=TRUE)
```

Table File

A data table can resides in a text file. The cells inside the table are separated by blank characters. Here is an example of a table with 4 rows and 3 columns.

```
100   a1   b1
200   a2   b2
300   a3   b3
400   a4   b4
```

Now copy and paste the table above in a file named "mydata.txt" with a text editor. Then load the data into the workspace with the read.table function.

```
> mydata = read.table("mydata.txt")  # read text file
> mydata                             # print data frame
   V1 V2 V3
1 100 a1 b1
2 200 a2 b2
3 300 a3 b3
4 400 a4 b4
```

For further detail of the read.table function, please consult the R documentation.

```
> help(read.table)
```

CSV File

The sample data can also be in comma separated values (CSV) format. Each cell inside such data file is separated by a special character, which usually is a comma, although other characters can be used as well.

The first row of the data file should contain the column names instead of the actual data. Here is a sample of the expected format.

```
Col1,Col2,Col3
100,a1,b1
200,a2,b2
300,a3,b3
```

After we copy and paste the data above in a file named "mydata.csv" with a text editor, we can read the data with the read.csv function.

```
> mydata = read.csv("mydata.csv")  # read csv file
> mydata                           # print data frame
  Col1 Col2 Col3
1  100   a1   b1
2  200   a2   b2
3  300   a3   b3
```

In various European locales, as the comma character serves as decimal point, the read.csv2 function should be used instead. For further detail of the read.csv and read.csv2 functions, please consult the R documentation.

```
> help(read.csv)
```

Working Directory

Finally, the code samples above assume the data files are located in the R working directory, which can be found with the getwd() function.

```
> getwd()             # get the current working directory
```

You can select a different working directory with the setwd() function, and avoid entering the full path of the data files.

```
> setwd("<new path>")   # set the working directory to "<new path>"
```

The forward slash should be used as the path separator even on Windows platform.

```
> setwd("C:/MyDoc")     # set the working directory to "C:\MyDoc"
```