# Session Six - Transforming Manipulating Data Objects

By Hellen Gakuruh

June 2, 2016

## Table of Contents

## Session Goal

The main goal of this session is to facilitate you to make necessary pre-analysis activities.

## What we shall cover

By the end of this session you should:

- Be able to index values/elements any R data object
- Know how to use binary operators to make comparison and check conditions
- Understand vectorisation in R
- Have the skills to extract and/or replace parts of any data object in R

## Prerequisite

To appreciate this session, you must be conversant with:

- Making function calls
- Difference between errors and warnings
- R's data types and objects
- Installing and loading packages
- Importing data into R

## Indexing different data objects in R

An index is a position marker or identifier of values/elements in an object. There are two ways to specify an index, with an numerical or a character (for named values/elements). Logical value TRUE, FALSE and NA can also be used to index elements of a vector, but they eventualy become integers (1 for TRUE and 0 for FALSE).

Values in a vector are indexed by the dimensions of the vector, hence, a one dimensional vector like a basic atomic vector or a list would have one index while a two dimensional vector such as a matrix or a data frame, would have two indices. Higher dimensional vectors like arrays, have indices the same length as its dimension, for example, a three dimensional array would have three indices.

### Indexing one dimensional objects

One dimensional vectors include basic vectors and lists. Values or elements are specified with one index.

#### One dimension (1d) Atomic/Homogenous vectors

```
set.seed(6839)
x <- sample(1:100, 10)
x
##  [1] 35 24 78 59 82  9 52 75 80 94
```

Vector x has 10 values, the position of each of these values gives its index as an integer.

| Value | Index |
|---|---|
| 35 | 1 |
| 24 | 2 |
| 78 | 3 |
| 59 | 4 |
| 82 | 5 |
| 9 | 6 |
| 52 | 7 |
| 75 | 8 |
| 80 | 9 |
| 94 | 10 |

If this vector were named, then values can be specified with their names

```
# Character index
names(x) <- letters[1:10]
x
##  a  b  c  d  e  f  g  h  i  j 
## 35 24 78 59 82  9 52 75 80 94
```

| Value | Index |
|---|---|
| 35 | "a" |
| 24 | "b" |
| 78 | "c" |
| 59 | "d" |
| 82 | "e" |
| 9 | "f" |
| 52 | "g" |
| 75 | "h" |
| 80 | "i" |
| 94 | "j" |

### One dimension (1d) Generic/Heterogeneous vectors

Lists are one dimensional objects whose contents can have different types of objects (hence heterogenous). Like basic atomic vectors, they are indexed with one value.

```
library(reshape2)
aList <- list(data = head(tips), metaData = c("This dataset is from Reshape2 
package"))
aList
## $data
##   total_bill  tip    sex smoker day   time size
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3
## 4      23.68 3.31   Male     No Sun Dinner    2
## 5      24.59 3.61 Female     No Sun Dinner    4
## 6      25.29 4.71   Male     No Sun Dinner    4
##
## $metaData
## [1] "This dataset is from Reshape2 package"
```

| Objects in the list | Integer indexing | Character indexing |
|---|---|---|
| First element | 1 | "data" |
| Second element | 2 | "metaData" |

Elements in a list can further be indexed per their dimension. In this case there are two elements; the first is a data frame with two dimensions and the second is a one dimesional character vector both of which can be indexed by their dimensions.

### Indexing One Dimension Character vectors

Charater or string vectors are slightly different when it comes to indexing. Indices are given per character which include spaces and punctuation marks. Words can be indexed with a range of indices from its beginning to its end. This range can easily be determined by a package called "strigi".

```
library(stringi)
char <- "This is session six"
stri_locate_all_words(char)
## [[1]]
##      start end
## [1,]     1   4
## [2,]     6   7
## [3,]     9  15
## [4,]    17  19
```

## Indexing two dimensional vectors

There are two types of two dimensional objects commonly used in R, these are matrices which are atomic or homogeneous vectors and data frames which are generic or heterogeneous objects.

### Two dimensional (2d) Atomic/Homogenous vectors

Matrices are two dimensional objects that contain one data type (homogeneous). Since they have two dimensions, values are specified with two indices; row and column.

```
set.seed(1038)
mat <- matrix(round(rnorm(12, 50, 5), 1), nrow = 4)
dimnames(mat) <- list(letters[1:4], LETTERS[1:3])
mat
```

```
##      A    B    C
## a 45.7 58.6 47.7
## b 52.9 47.2 45.0
## c 50.5 52.7 55.2
## d 42.2 41.5 47.6
```

"Mat" has twelve values. Each value can be specified by its row and column position.

| Value | Integer indices | Character indicies |
|---|---|---|
| 45.7 | 1, 1 | "a, A" |
| 52.9 | 2, 1 | "b, A" |
| 50.5 | 3, 1 | "c, A" |
| 42.2 | 4, 1 | "d, A" |
| 58.6 | 1, 2 | "a, B" |
| 47.2 | 2, 2 | "b, B" |
| 52.7 | 3, 2 | "c, B" |
| 41.5 | 4, 2 | "d, B" |
| 47.7 | 1, 3 | "a, C" |
| 45.0 | 2, 3 | "b, C" |
| 55.2 | 3, 3 | "c, C" |
| 47.6 | 4, 3 | "d, C" |

An index 1,1 means first row, first column and "a, A" means row name "a" and column name "A". A complete row can be indexed with an integer or name of the row followed by a blank column index.

| Row | Integer indices | Character indices |
|---|---|---|
| First | 1, | "a, " |
| Second | 2, | "b, " |
| Third | 3, | "c, " |
| Fourth | 4, | "d, " |

To index a column.

| Column | Integer indices | Character indices |
|---|---|---|
| First | , 1 | " , a" |
| Second | , 2 | " , b" |
| Third | , 3 | " , c" |
| Fourth | , 4 | " , d" |

## Atomic/Homogenous vectors with more than 2 dimensions (>2d)

Arrays are atomic vectors with one or more dimensions, matrices is a special case of an array. For one dimensional array, indexing is like a basic vector while two dimensional arrays would be indexed like a matrix. In the case of arrays with more than two dimensions, values are indexed by the number of dimensions, so for a three dimensional array, values are specified with three indices.

```
ucbArray
## , , Dept = A
##
##           Gender
## Admit      Male Female
##   Admitted  512     89
##   Rejected  313     19
##
## , , Dept = B
##
##           Gender
## Admit      Male Female
##   Admitted  353     17
##   Rejected  207      8

# Dimensions
dim(ucbArray)
## [1] 2 2 2
length(dim(ucbArray))
## [1] 3

# Number of values in the array
length(as.vector(ucbArray))
## [1] 8
```

"ucbArray" is an array with three dimensions and 8 values. The first index is for row, the second is for columns and the third is for subsetction or layer like departments in this case.

| Value | Integer indice | Character Indices |
|---|---|---|
| 512 | 1, 1, 1 | "Admitted, Male, A" |
| 313 | 2, 1, 1 | "Rejected, Male, A" |
| 89 | 1, 2, 1 | "Admitted, Female, A" |
| 19 | 2, 2, 1 | "Rejected, Female, A" |
| 353 | 1, 1, 2 | "Admitted, Male, B" |
| 207 | 2, 1, 2 | "Rejected, Male, B" |
| 17 | 1, 2, 2 | "Admitted, Female, B" |
| 8 | 2, 2, 2 | "Admitted, Female, B" |

An index of 2,2,1 means a value in the second row, second column and first layer (department).

## Two dimensional (2d) Generic/Heterogeneous vectors

Data frames are generic or heterogeneous vectors. They are the most widely used data objects in R. Being two dimensions, data frame values are indexed as matrices; by their row and column position.

```
irisSub
##   Sepal.Length Petal.Length Species
## 1          5.1          1.4  setosa
## 2          4.9          1.4  setosa
## 3          4.7          1.3  setosa
## 4          4.6          1.5  setosa
## 5          5.0          1.4  setosa

# Dimension
dim(irisSub)
## [1] 5 3

# Number of dimensions
length(dim(iris))
## [1] 2

# Total number of values
length(irisSub) * nrow(irisSub)
## [1] 15
```

"irisSub" is a data frame with two dimensions (5 row and 3 columns) and 15 values. Each of these values can be indexed by their row and column position.

| Value | Integer indices | Character indices |
|---|---|---|
| 5.1 | 1, 1 | 1, "Sepal.Length" |
| 1.4 | 1, 2 | 1, "Petal.Lenght" |
| setosa | 1, 3 | 1, "Species" |
| 4.9 | 2, 1 | 2, "Sepal.Length" |
| 1.4 | 2, 2 | 2, "Petal.Length" |
| setosa | 2, 3 | 2, "Species" |
| 4.7 | 3, 1 | 3, "Sepal.Length" |
| 1.3 | 3, 2 | 3, "Petal.Length" |
| setosa | 3, 3 | 3, "Species" |
| 4.6 | 4, 1 | 4, "Sepal.Length" |
| 1.5 | 4, 2 | 4, "Petal.Length" |
| setosa | 4, 3 | 4, "Species" |

|  |  |  |
|---|---|---|
| 5.0 | 5, 1 | 5, "Sepal.Length" |
| 1.4 | 5, 2 | 5, "Petal.Length" |
| setosa | 5, 3 | 5, "Species" |

Notice we can mix both integers and character indices, in this case we had to mix them as rows are not named.

There are two important issues that needs to be discussed before venturing into subsetting and data manipulation, these are binary operators and conditional functions useful for subsetting.

## Binary and vectorisation Operators

### Comparison Operators

Comparison operators also known as relational operators are functions for comparing values of atomic vectors (they have only one data type). R has six comparison operators, these are:

| Operator | Usage |
|---|---|
| == | Equal |
| != | Not Equal |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |

These operators output/produce logical values TRUE, FALSE or NA.

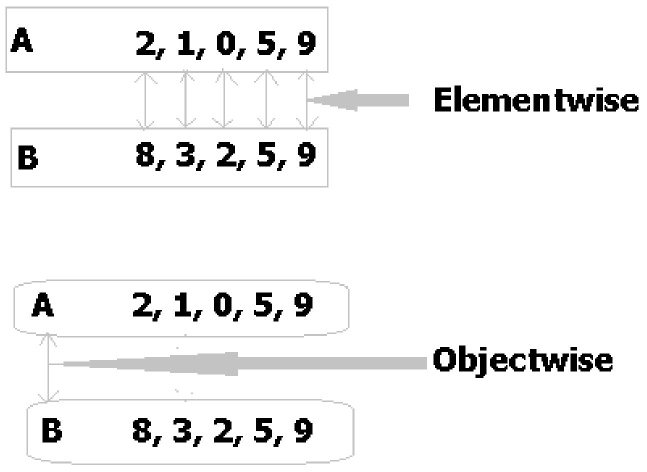
#### Equality

To test equality, we will begin with length-one atomic vectors.

```r
# Comparing integers
2L == 3L
## [1] FALSE
# Comparing doubles
3.56 == 3.6
## [1] FALSE
round(3.56, 1) == 3.6
## [1] TRUE
# Comparing characters (words)
"Hellen" == "Helen"
## [1] FALSE
# Comparing Logical values
TRUE == FALSE
## [1] FALSE
```

Vectorized Comparison

Binary operators can also be used with vectors of length greater than one. In this case comparison is carried out element by element producing a logical vector the same length as the two vectors being compared. This type of comparison; elementwise, is referred to as **vectorized comparison**. Therefore vectorization allows comparison to be done between elements of two (binary) vectors.



*Elemenwise and Vectorized comparison*

Do take note, if two vectors are of different lengths, the shorter vector is recycled until it reaches the same length as the longer vector before comparison is carried out, this will result in an output and a warning.

```
# elementwise comparison
a <- c(2, 1, 0, 5, 9)
b <- c(8, 3, 2, 5, 9)
a == b
## [1] FALSE FALSE FALSE  TRUE  TRUE

# Comparison with a shorter vector
c <- c(2, 2)
a == c
## Warning in a == c: longer object length is not a multiple of shorter
object
## length
## [1]  TRUE FALSE FALSE FALSE FALSE
b == c
## Warning in b == c: longer object length is not a multiple of shorter
object
```

```
## length
## [1] FALSE FALSE  TRUE FALSE FALSE
```

To check equality between two vectors as an objects rather that its elements only, function "identical" and "all.equal" are more appropriate. Objectwise comparison also involves checking type and attributes of the object for equality.

```
a == b
## [1] FALSE FALSE FALSE  TRUE  TRUE
identical(a, b)
## [1] FALSE
all.equal(a, b)
## [1] "Mean relative difference: 3.333333"
```

There are subtle difference between "identical" and "all.equal". All.equal checks for "nearness" but identical checks for complete equality.

```
e <- c(round(1.2938), round(3.2903), round(6.3958))
f <- c(1L, 3L,  6L)
e == f
## [1] TRUE TRUE TRUE
class(e); class(f)
## [1] "numeric"
## [1] "integer"
typeof(e); typeof(f)
## [1] "double"
## [1] "integer"
identical(e, f)
## [1] FALSE
all.equal(e, f)
## [1] TRUE
```

These two functions (identical and all.equal) will become useful when discussing control structures.

### Non - Equality

Non-equality between two (binary) objects is done using negation, that is "!=".

```
# Length-one vectors
2 != 3
## [1] TRUE
2L != 3L
## [1] TRUE
"Hellen" != "Helen"
## [1] TRUE
TRUE != FALSE
## [1] TRUE

# Vectors with length > 1
a; b
```

```
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
a != b
## [1]  TRUE  TRUE  TRUE FALSE FALSE
# They look the same but are not the same
e; f
## [1] 1 3 6
## [1] 1 3 6
e != f
## [1] FALSE FALSE FALSE
```

### Greater than operator

This operator typically asks if value(s) on the left hand side is(are) greater than the value(s) on the right hand side.

```
2 > 3
## [1] FALSE
2L > 3L
## [1] FALSE
# Comparison for words is by each character
"Hellen" > "Helen"
## [1] TRUE
# Logical values are interally stored as 1 for TRUE and 0 for FALSE
TRUE > FALSE
## [1] TRUE
# Element comprison for vectors > 1
a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
a > b
## [1] FALSE FALSE FALSE FALSE FALSE
# Comparison for double and integer vectors (taking account of decimal point)
e; f
## [1] 1 3 6
## [1] 1 3 6
e > f
## [1] FALSE FALSE FALSE
```

### Greater than or equal operator

This operator checks if the left value is greater than or equal to the value on the right.

```
2 >= 3
## [1] FALSE
2L >= 3
## [1] FALSE
"Hellen" >= "Helen"
## [1] TRUE
TRUE >= FALSE
## [1] TRUE
```

```
a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
a >= b
## [1] FALSE FALSE FALSE  TRUE  TRUE
e; f
## [1] 1 3 6
## [1] 1 3 6
e >= f
## [1] TRUE TRUE TRUE
```

### Less than operator

This operator compares two values to see if the first is smaller than the second value.

```
2 < 3
## [1] TRUE
2L < 3
## [1] TRUE
"Hellen" < "Helen"
## [1] FALSE
TRUE < FALSE
## [1] FALSE

# Vectorized
a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
a < b
## [1]  TRUE  TRUE  TRUE FALSE FALSE
e; f
## [1] 1 3 6
## [1] 1 3 6
e < f
## [1] FALSE FALSE FALSE
```

### Less than or equal to operator

Finally this operator compares the two values to see if the first is equal or less than the second value.

```
2 <= 3
## [1] TRUE
2L <= 3
## [1] TRUE
"Hellen" <= "Helen"
## [1] FALSE
TRUE <= FALSE
## [1] FALSE

# Vectorized
```

```
a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
a <= b
## [1] TRUE TRUE TRUE TRUE TRUE
e; f
## [1] 1 3 6
## [1] 1 3 6
e <= f
## [1] TRUE TRUE TRUE
```

## Logical Operators

Logical operators are used to compare two values for commonality, combination or negation.

In R there are two operators that test for commonality, these are "&" and "&&" (meaning **and**); two for combination that is "|" and "||" (meaning **or**); and one operator for negation "!" (meaning not).

The "&" and "|" perform elementwise comparison while "&&", "||" perform comparison for only the first element of each vector.

```
# For length-one vectors
#-----------------------
# Is there a common value between 2 AND 8. Yes, 2
2 & 8
## [1] TRUE
# How about 0 AND 2? There is none
0 & 2
## [1] FALSE

# Is there a 2 OR a 5 in a vector with just 2 and another with 5? Yes
2 | 5
## [1] TRUE
# How about a 0 OR a 2 in a vector with just 0 and another with just 5? Yes
0 | 2
## [1] TRUE

# Vectors with length > 1 (Same logic as length-one vector)
#------------------------------------------------------------
a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
a & b
## [1]  TRUE  TRUE FALSE  TRUE  TRUE
a | b
## [1] TRUE TRUE TRUE TRUE TRUE
e; f
## [1] 1 3 6
## [1] 1 3 6
e & f
## [1] TRUE TRUE TRUE
e | f
## [1] TRUE TRUE TRUE
```

To sum up, when we use `AND (&)` we are checking for common value(s) and when we use `OR (|)`, then we are looking at all values on the left and on the right of the operator (a combination).

To check only the first values of a vector with length greater than one, we use "&&" and "||". This is mostly suitable for control structures like "if" which we will discuss later.

```
a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
a && b
## [1] TRUE
a || b
## [1] TRUE
e; f
## [1] 1 3 6
## [1] 1 3 6
e && f
## [1] TRUE
e || f
## [1] TRUE

# What about this, where first values (0 and 1) have nothing in common
0:5 && 1:6
## [1] FALSE
```

Logical statements work with all types of data except character vectors which will produce an error.

```
char1 <- c("this", "is", "a", "character", "vector")
char2 <- c("Yet", "another", "character", "vector")
char1 & char2
## Error in c("this", "is", "a", "chracter", "vector") & c("Yet", "another",
:
##  operations are possible only for numeric, logical or complex types
```

**Negating operator**

An exclamation mark (!) is used to get the opposite of a logical operation.

```
# There is no common value between 2 AND 8. False there is 2
!2 & 8
## [1] FALSE
# There is no common value between 0 and 2. True, there is no common value
```

```
!0 & 2
## [1] TRUE
```

## Checking conditions

There are three functions that will become quite handy when assessing or subseting data. These are the `all()`, `any()`, and `which()` functions. They are used together with comparison and logical operators.

### "All" function

This function is used to assesses whether all values in a logical vector are TRUE.

```
logVec1 <- c(TRUE, TRUE, TRUE, TRUE)
logVec2 <- c(TRUE, TRUE, FALSE, TRUE)
all(logVec1)
## [1] TRUE
all(logVec2)
## [1] FALSE
```

all() is ideal when used with comparison operators to give an overall assessment of logical values.

```
a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
# Are all values in "a" and "b" the same? No, index 1:4 are different
all(a == b)
## [1] FALSE
# Are all the values in "a" different from "b"? No, index 4 and 5 are the
same
all(a != b)
## [1] FALSE
```

### "Any" function

"any()" is used to check if there is at least one TRUE value in a logical vector.

```
# Checking if there is at least one TRUE value
any(logVec1)
## [1] TRUE
any(logVec2)
## [1] TRUE

a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
any(a == b)
## [1] TRUE
any(a != b)
## [1] TRUE
```

```
any(a > b)
## [1] FALSE

# Other ways to use any (As long as ... argument is a logical vector)
any(a == FALSE)
## [1] TRUE
any(a == 0)
## [1] TRUE

# any() is not suitable to check for missing values directly
any(a == NA)
## [1] NA
# Instead use is.na() first
any(is.na(a))
## [1] FALSE
```

### "Which" function

This function is used to get the indices of all TRUE values in a logical vector.

```
logVec1
## [1] TRUE TRUE TRUE TRUE
which(logVec1)
## [1] 1 2 3 4
logVec2
## [1]  TRUE  TRUE FALSE  TRUE
which(logVec2)
## [1] 1 2 4

a; b
## [1] 2 1 0 5 9
## [1] 8 3 2 5 9
which(a == b)
## [1] 4 5
which(a != b)
## [1] 1 2 3
which(a & b)
## [1] 1 2 4 5
which(a | b)
## [1] 1 2 3 4 5
```

## Subsetting Data Objects

R provides three subsetting operators, these are single square bracker ([ ]), double square brackects ([[ ]]) and the dollar sign ($). With these operators, one can extract or replace parts of a data object. Some operators are applicable to only data object while others are applicable to all data objects though they might have different usage.

## Subsetting one dimensional atomic vectors

Single square brackets are used to subset or extract data from a one dimensional atomic vector. Within the brackets are the indices of data to be extracted or replaced. Recall indices can either be integers or characters, logical values can also be used.

```
x
##  a  b  c  d  e  f  g  h  i  j
## 35 24 78 59 82  9 52 75 80 94

# Using integer indices
#----------------------
# Subsetting the first six values (sequenced numeric/factors use :)
x[1:6]
##  a  b  c  d  e  f
## 35 24 78 59 82  9
# This can also be achieved with head function
head(x)
##  a  b  c  d  e  f
## 35 24 78 59 82  9
# Geting last six values
x[(length(x) - 5):length(x)]
##  e  f  g  h  i  j
## 82  9 52 75 80 94
# This can be achieved by
tail(x)
##  e  f  g  h  i  j
## 82  9 52 75 80 94
# Subsetting unsequenced values
x[c(1, 5, 9)]
##  a  e  i
## 35 82 80
# Negative indices excluded them from subset vector
x[-c(1:6)]
##  g  h  i  j
## 52 75 80 94
x[-c(1, 5, 9)]
##  b  c  d  f  g  h  j
## 24 78 59  9 52 75 94

# Using character indices (names)
#--------------------------------
x[c("a", "e", "i")]
##  a  e  i
## 35 82 80

# Negative indices only works for numeric values
x[-c("a", "e", "i")]
## Error in -c("a", "e", "i") : invalid argument to unary operator
```

```r
# Using logical vectors
#-----------------------
# Logical vector the same length as x
x[c(TRUE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)]
##  a  d  e  f  g  i
## 35 59 82  9 52 80
# Logical vector with values less than length of x (it will be recycled)
x[c(TRUE, FALSE, FALSE, TRUE, TRUE)]
##  a  d  e  f  i  j
## 35 59 82  9 80 94
# Conditional logical vector
x[x > 50]
##  c  d  e  g  h  i  j
## 78 59 82 52 75 80 94
x[x < 10]
## f
## 9
# Negative logical indices works because logical values are coerced to
integers
x[-c(TRUE, FALSE, FALSE, TRUE, TRUE)]
##  b  c  d  e  f  g  h  i  j
## 24 78 59 82  9 52 75 80 94
# For comparison operators, instead of negative values, use negation
x[!x > 50]
##  a  b  f
## 35 24  9
x[!x < 10]
##  a  b  c  d  e  g  h  i  j
## 35 24 78 59 82 52 75 80 94

# An empty bracket returns original vector
x[]
##  a  b  c  d  e  f  g  h  i  j
## 35 24 78 59 82  9 52 75 80 94
```

## Subsetting a Matrix

A matrix is an atomic vector with dimensions, and like one dimensional atomic vectors, they use single brackets to subset elements, in addition, double square brackets are also used to subset specific values.

Dimensions are used inside the brackets to index values to be extracted or replaced.

```r
USPersonalExpenditure
##                       1940   1945  1950 1955  1960
## Food and Tobacco    22.200 44.500 59.60 73.2 86.80
## Household Operation 10.500 15.500 29.00 36.5 46.20
## Medical and Health   3.530  5.760  9.71 14.0 21.10
## Personal Care        1.040  1.980  2.45  3.4  5.40
## Private Education    0.341  0.974  1.80  2.6  3.64
```

```
# Using integers for both row and column indices
#-------------------------------------------------
# Food and tobacco expenditure for 1940
USPersonalExpenditure[1, 1]
## [1] 22.2
# Medical and health expenditure from 1940 upto 1960
USPersonalExpenditure[3, ]
##  1940  1945  1950  1955  1960
##  3.53  5.76  9.71 14.00 21.10
# 1950 total expenditure
USPersonalExpenditure[, 3]
##    Food and Tobacco Household Operation  Medical and Health
##               59.60               29.00                9.71
##       Personal Care   Private Education
##                2.45                1.80
# All expenditure for 1940, 1950 and 1960
USPersonalExpenditure[, c(1, 3, 5)]
##                       1940  1950  1960
## Food and Tobacco    22.200 59.60 86.80
## Household Operation 10.500 29.00 46.20
## Medical and Health   3.530  9.71 21.10
## Personal Care        1.040  2.45  5.40
## Private Education    0.341  1.80  3.64
# All expediture excluding 1945 and 1955
USPersonalExpenditure[, -c(2, 4)]
##                       1940  1950  1960
## Food and Tobacco    22.200 59.60 86.80
## Household Operation 10.500 29.00 46.20
## Medical and Health   3.530  9.71 21.10
## Personal Care        1.040  2.45  5.40
## Private Education    0.341  1.80  3.64

# Indexing using row and column names
#--------------------------------------
# Household Operation expenditure for 1940, 1950 and 1960
USPersonalExpenditure["Household Operation", c("1940", "1950", "1960")]
## 1940 1950 1960
## 10.5 29.0 46.2
# Personal expenditure from 1940 through to 1960
USPersonalExpenditure["Personal Care",]
## 1940 1945 1950 1955 1960
## 1.04 1.98 2.45 3.40 5.40
# All expenditure for 1940 and 1960
USPersonalExpenditure[, c("1940", "1960")]
##                       1940  1960
## Food and Tobacco    22.200 86.80
## Household Operation 10.500 46.20
## Medical and Health   3.530 21.10
## Personal Care        1.040  5.40
## Private Education    0.341  3.64

# Using logical vectors
#-----------------------
logVec3 <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
USPersonalExpenditure[logVec3, logVec3]
##                       1940  1950  1960
## Food and Tobacco    22.200 59.60 86.80
## Medical and Health   3.530  9.71 21.10
## Private Education    0.341  1.80  3.64
# Expenditure above 10 billion
USPersonalExpenditure[which(USPersonalExpenditure > 10)]
##  [1] 22.2 10.5 44.5 15.5 59.6 29.0 73.2 36.5 14.0 86.8 46.2 21.1

# Using double square brackets
#------------------------------
# Getting the first value
USPersonalExpenditure[[1]]
## [1] 22.2
```

## Subsetting a dataframe

Data frames are two dimensional generic vectors. To subset a data frame, single and double square bracket as well as a dollar sign are used.

```
library(reshape2)
str(tips)
## 'data.frame':    244 obs. of  7 variables:
##  $ total_bill: num  17 10.3 21 23.7 24.6 ...
##  $ tip       : num  1.01 1.66 3.5 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3
3 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  2 3 3 2 4 4 2 4 2 2 ...

# Using integer indices
#-------------------------
# Subsetting first 6 rows of the first 3 columns
tips[1:6,1:3]
##   total_bill  tip    sex
## 1      16.99 1.01 Female
## 2      10.34 1.66   Male
## 3      21.01 3.50   Male
## 4      23.68 3.31   Male
## 5      24.59 3.61 Female
## 6      25.29 4.71   Male
# First 6 rows similar to head(tips)
```

```
tips[1:6,]
##   total_bill  tip    sex smoker day   time size
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3
## 4      23.68 3.31   Male     No Sun Dinner    2
## 5      24.59 3.61 Female     No Sun Dinner    4
## 6      25.29 4.71   Male     No Sun Dinner    4
head(tips)
##   total_bill  tip    sex smoker day   time size
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3
## 4      23.68 3.31   Male     No Sun Dinner    2
## 5      24.59 3.61 Female     No Sun Dinner    4
## 6      25.29 4.71   Male     No Sun Dinner    4
```

Logical vectors can be produced by checking certain conditions. Here we will combine logical and variable names indices to subset using the dollar sign.

```
# Tipping in relation to smoking habits
#-------------------------------------
# Non smoking tippers
nonSmokers <- tips[tips$smoker == "No", ]
str(nonSmokers)
## 'data.frame':    151 obs. of  7 variables:
##  $ total_bill: num  17 10.3 21 23.7 24.6 ...
##  $ tip       : num  1.01 1.66 3.5 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3
3 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  2 3 3 2 4 4 2 4 2 2 ...
# Female Non-smoking tippers
femNonSmokers <- tips[tips$smoker == "No" & tips$sex == "Female", ]
str(femNonSmokers)
## 'data.frame':    54 obs. of  7 variables:
##  $ total_bill: num  17 24.6 35.3 14.8 10.3 ...
##  $ tip       : num  1.01 3.61 5 3.02 1.67 3.5 2.75 2.23 3 3 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 1 1 1 1 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 2 2 2
2 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  2 4 4 2 3 3 2 2 2 2 ...
# Male Non smoking tippers
malNonSmokers <- tips[tips$smoker == "No" & tips$sex == "Male", ]
```

```
str(malNonSmokers)
## 'data.frame':    97 obs. of  7 variables:
##  $ total_bill: num  10.34 21.01 23.68 25.29 8.77 ...
##  $ tip       : num  1.66 3.5 3.31 4.71 2 3.12 1.96 3.23 1.71 1.57 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 2 2 2 2 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3
3 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  3 3 2 4 2 4 2 2 2 2 ...
# Tippers who smoked
smokers <- tips[tips$smoker == "Yes", ]
str(smokers)
## 'data.frame':    93 obs. of  7 variables:
##  $ total_bill: num  38 11.2 20.3 13.8 11 ...
##  $ tip       : num  3 1.76 3.21 2 1.98 3.76 1 2.09 3.14 5 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 1 2 1 1 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 2 2 2 2 2 2 2 2 2
2 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  4 2 2 2 2 4 1 2 2 2 ...
# Female tippers who smoked
femSmokers <- tips[tips$smoker == "Yes" & tips$sex == "Female", ]
str(femSmokers)
## 'data.frame':    33 obs. of  7 variables:
##  $ total_bill: num  3.07 26.86 25.28 5.75 16.32 ...
##  $ tip       : num  1 3.14 5 1 4.3 2.5 3 2.5 3.48 4 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 1 1 1 1 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 2 2 2 1 1 1 1 2 2
2 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  1 2 2 2 2 2 2 3 2 2 ...
# Male tippers who smoked
malSmokers <- tips[tips$smoker == "Yes" & tips$sex == "Male", ]
```

To extract the first vector of a data frame simplified as a vector, double square brackets are appropriate.

```
# Subsetting first column ("total bill")
str(tips[[1]])
##  num [1:244] 17 10.3 21 23.7 24.6 ...
# Same as
str(tips[, 1])
##  num [1:244] 17 10.3 21 23.7 24.6 ...
```

A function called "subset" can also be used, it is more efficient though it is based on the same notion as brackets and dollar sign.

```
# Female tippers
fem <- subset(tips, sex == "Female")
# Male tippers
mal <- subset(tips, sex == "Male")
# Female tippers who smoked
fem.smoker <- subset(tips, smoker == "Yes" & sex == "Female")
fem.nonsmoker <- subset(tips, smoker == "No" & sex == "Female")
mal.smoker <- subset(tips, smoker == "Yes" & sex == "Male")
mal.nonsmoker <- subset(tips, smoker == "No" & sex == "Male")
# Number of female smokers and non-smokers
nrow(fem.smoker); nrow(fem.nonsmoker)
## [1] 33
## [1] 54
# Number of male smokers and non-smokers
nrow(mal.smoker); nrow(mal.nonsmoker)
## [1] 60
## [1] 97
```

## Subsetting a List

List are one dimensional vectors containing one or more type of object (not necessarily data objects). Contents of a list are referred to as elements. Subsetting elements of a list can involve either a single or double bracket, or a dollar sign.

Key guiding issue in subsetting lists is whether to preserve original data structure or simply it, that is, return a subset as a list or in another simpler data structure (data frame, matrix, array or vector). To preserve data structure, single bracket is used, to simply double brackets or the dollar sign can be used.

```
# Creating a list
subsetList <- list(Females = fem, Males = mal, FemaleSmokers = fem.smoker,
MaleSmokers = mal.smoker, FemaleSmokers = fem.nonsmoker, MaleSmokers =
mal.nonsmoker)

# Subsetting first element
#-------------------------
# Returning a list (preserves data structure)
elementAsList <- subsetList[1]
class(elementAsList)
## [1] "list"
# Returning a dataframe (simplifies data structure)
elementAsDf <- subsetList[[1]]
class(elementAsDf)
## [1] "data.frame"
# $ is the same as using [[]]
elementAsDf2 <- subsetList$Females
```

```
class(elementAsDf2)
## [1] "data.frame"
```

Elements of lists can also be subset depending on the type of object returned after simplification.

```
# First column of female tippers subset as a data frame (preserving)
subElementAsDf <- subsetList[[1]][2]
# First column of female tippers subset as a vector
subsetList[[1]][[2]]
##  [1] 1.01 3.61 5.00 3.02 1.67 3.50 2.75 2.23 3.00 3.00 2.45 3.07 2.60 5.20
## [15] 1.50 2.47 1.00 3.00 3.14 5.00 2.20 1.83 5.17 1.00 4.30 3.25 2.50 3.00
## [29] 2.50 3.48 4.08 4.00 1.00 4.00 3.50 1.50 1.80 2.92 1.68 2.52 4.20 2.00
## [43] 2.00 2.83 1.50 2.00 3.25 1.25 2.00 2.00 2.75 3.50 5.00 2.30 1.50 1.36
## [57] 1.63 5.14 3.75 2.61 2.00 3.00 1.61 2.00 4.00 3.50 3.50 4.19 5.00 2.00
## [71] 2.01 2.00 2.50 3.23 2.23 2.50 6.50 1.10 3.09 3.48 3.00 2.50 2.00 2.88
## [85] 4.67 2.00 3.00
```

For named lists, it is much easier to subset using names of the list and if the components are also named, then they can also be used.

```
# Female tippers who tipped above average
femTipsHigh <- subsetList$Females[subsetList$Females$tip >
mean(subsetList$Females$tip), ]
# Male tippers who tipped above average
malTipsHigh <- subsetList$Males[subsetList$Males$tip >
mean(subsetList$Males$tip), ]

# Comparison
nrow(femTipsHigh)/nrow(subsetList$Females) * 100
## [1] 47.12644
nrow(malTipsHigh)/nrow(subsetList$Males)  * 100
## [1] 40.76433
```

There are two special cases of one dimensional objects we should discuss further, these are character and factor vectors.

## Subsetting character vectors

To subset words in a sentense or line, we need to extract the content of "stri_locate_all_words()" which comes in a form of a list.

```
char
## [1] "This is session six"

# Extracting first (and only) element of the list
charMat <- stri_locate_all_words(char)[[1]]
cat("Extracted element simplified to a", class(charMat), "by using double
brackets.\n")
## Extracted element simplified to a matrix by using double brackets.
charMat
```

```
##      start end
## [1,]     1   4
## [2,]     6   7
## [3,]     9  15
## [4,]    17  19

# This vector extraction method would not extract a word or a letter
char[1]
## [1] "This is session six"

# Instead use stri_sub function from package stringi
# Extracting the word "session"
stri_sub(char, from = 9L, to = 15L)
## [1] "session"
# Extracting second "s" in session
stri_sub(char, 11, length = 1)
## [1] "s"
```

### Subsetting factor vectors

Factor vectors can be subset like any other atomic vector but with an option of returning all the levels or only the subset levels.

```
# A factor vector
tipsDay <- tips$day

# Subsetting weekend (Saturday and Sunday) but mantaining original levels
weekend1 <- tipsDay[tipsDay == "Sat" | tipsDay == "Sun"]
str(weekend1)
##  Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3 3 ...
# Subsetting weekdays but with only relevant levels
weekend2 <- tipsDay[tipsDay == "Sat" | tipsDay == "Sun", drop = TRUE]
str(weekend2)
##  Factor w/ 2 levels "Sat","Sun": 2 2 2 2 2 2 2 2 2 2 ...
```

## Transformation/Manipulation of datasets

Before undertaking any data transformation or manipulation, the first thing to do is save a copy of raw data set in one or more different locations. This will give a backup copy of the data file in case of unexpected data transformation or manipulation.

One of the first thing to do with any data set is to inspect it's quality, in particular you would be interested in assessing completeness (no missing data).

```
# Short description of data
str(tips)
## 'data.frame':    244 obs. of  7 variables:
##  $ total_bill: num  17 10.3 21 23.7 24.6 ...
##  $ tip       : num  1.01 1.66 NA 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3
3 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  2 3 3 2 4 4 2 4 2 2 ...
```

To check for missing values, we will use a function that looks at each variable for missing data, this function is called "sapply". Sapply is part of R's looping functions (do repetitive tasks) and we shall discuss them in level 2. But for now we will used sapply() to produce a data frame with logical values TRUE for indices with missing data and FALSE for indices with values.

From sapply's output, we will further determine which variables have any missing values using the "which" function and lastly produce its name.

```
varMissing <- names(which(sapply(tips, function(x) any(is.na(x)))))
cat("Variable", paste0('"', varMissing, '"'), "has some missing data\n")
## Variable "tip" has some missing data
```

Let's determine the magnitude of missing data.

```
numMissing <- length(which(is.na(tips$tip)))
percMissing <- round(numMissing/length(tips$tip) * 100)
cat("Missing data accounts for", paste0(percMissing, "%"))
## Missing data accounts for 3%
```

To have a complete dataset, we will impute its average.

```
averageTip <- mean(tips$tip, na.rm = TRUE)
tips[is.na(tips$tip), "tip"] <- averageTip
str(tips)
## 'data.frame':    244 obs. of  7 variables:
##  $ total_bill: num  17 10.3 21 23.7 24.6 ...
##  $ tip       : num  1.01 1.66 3 3.31 3.61 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3
3 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  2 3 3 2 4 4 2 4 2 2 ...

# Confirm no missing data
names(which(sapply(tips, function(x) any(is.na(x)))))
## character(0)
```

One other common pre-analyis activity is handling categorical data, for example grouping numerical data or changing value labels.

```r
# Categorizing varable "tip" as above and below average
averageTip <- mean(tips$tip)
tipsCat <- cut(x = tips$tip, breaks = c(0, averageTip, max(tips$tip)), labels
= c("Below average", "Above average"))
tipsCat[1:5]
## [1] Below average Below average Below average Above average Above average
## Levels: Below average Above average

# Converting variable day to a binary vector (re-labeling)
tipsDay <- tips$day
levels(tipsDay)[2:3] <- "Weekend"
levels(tipsDay)[c(1, 3)] <- "Weekday"
str(tipsDay)
##  Factor w/ 2 levels "Weekday","Weekend": 2 2 2 2 2 2 2 2 2 2 ...
```

In some instances new variables can be created from computation of existing variables. Here we will create a variable which will compute tips as a percentage of total bill.

```r
tipsPerc <- round(tips$tip/tips$total_bill * 100)
str(tipsPerc)
##  num [1:244] 6 16 14 14 15 19 23 12 13 22 ...
```

Variables can also be added to data frames.

```r
# Adding grouped, re-labeled and computed variables
tips$TipAverage <- tipsCat
tips$TypeofDay <- tipsDay
tips$TipPerc <- tipsPerc
str(tips)
## 'data.frame':    244 obs. of  10 variables:
##  $ total_bill: num  17 10.3 21 23.7 24.6 ...
##  $ tip       : num  1.01 1.66 3 3.31 3.61 ...
##  $ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
##  $ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3
3 ...
##  $ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1
...
##  $ size      : int  2 3 3 2 4 4 2 4 2 2 ...
##  $ TipAverage: Factor w/ 2 levels "Below average",..: 1 1 1 2 2 2 1 2 1 2
...
##  $ TypeofDay : Factor w/ 2 levels "Weekday","Weekend": 2 2 2 2 2 2 2 2 2 2
...
##  $ TipPerc   : num  6 16 14 14 15 19 23 12 13 22 ...
```

Finally, it might be of interest to order a data frame according to some conditions.

```r
# Sort data frame according to percentage of tip to total bill
tips[order(tips$TipPerc, decreasing = TRUE), 5:10]
# Sort data frame according to percentage of tip to total bill and size of
```

```r
party
tips[order(tips$TipPerc, tips$size, decreasing = TRUE),]
```

## Gauge yourself

**Do you have the expected tools and skills from this session?**

1. Can you mix numerical(integer) and character indices, explain with an example
2. Is this a valid vector of indices for a one dimensional atomic vector; c(TRUE, FALSE, TRUE, NA, TRUE)
3. What do expect from 2i == 2; an error, an output with a warning, a valid comparison, explain.
4. Why does this "x == NA" produce "NA", how else can you get missing values?
5. What is your understanding of vectorization
6. What is the difference between all and any functions.
7. How is "&" and "&&" different
8. Why would this produce an error for a one dimensional atomic vector; x[1, 4, 5]
9. What do you anticipate from subsetting a matrix with "$columnname" e.g. "USPersonalExpenditure$1940"
10. Would this "iris[[1]]" output and error, a warning with an results or simply its output. If the latter, what do you expect it to produce.
11. How are "[]", "[[]]" and "$" different when subsetting lists?
12. How can you subset a word in a character vector
13. For factor variables, after subsetting specific levels, how can you ensure you get only labels for subset levels.
14. What function is ideal for small but comprehensive inspection of an object
15. What function would you use to convert numerical values into factors
16. How would you change the labels of a factor variable?
17. Between sort() and order(), which is ideal for data frames.

**You are ready for the seventh session if you know these**

1. Yes, you can mix numerical values and character indices for example "row1, 2" for row 1, second column.
2. Yes, it is valid, the "NA" index will output "NA"
3. Complex numbers can be compared to any data type, output will be a result of coercion on one to another. In this case 2 was coerced to complex.
4. x == NA will output NA because any operation involving NA will always be NA (its simply a placeholder, nothing to compare with). Instead, "is.na()" is used.
5. Vectorization involves vectors of length > 1 and operations carried out elementwise.
6. all() is used to determing if a logical vector has all its values as TRUE while any() checks for at least one TRUE value.
7. "&" is used for elementwise comparison while "&&" is used to compare only the first elements.

8. x[1, 4, 5] will produce an error as x expects one index being a one dimensional vector. If intension is to get values at index 1, 4, and 5, then these indices have to be vectorized with the function "c".
9. Subsetting a matrix with a "$" sign is an error as matrices can only be subset with brackets (single and double brackets)
10. "iris[[1]]" will result in an output with no warning as it will extract the first vector simplified as an unnamed one dimensional atomic vector.
11. "[]" will always return a list, while "[[]]" and "$" will return a simplified data object which will be the same class as the element or content being extracted.
12. Parts of a string are best extracted or replaced with functions from other packages like "stringi"
13. To get labels of only the subset levels, set argument "drop" to TRUE.
14. "str()" is a handy function for quick inspection of data object.
15. A function called "cut" is suitable for converting numerical values to categorical .
16. Function "levels", is used to change the labels of a factor vector
17. Order() is suitable for data frames while sort() is appropriate for one dimensional vectors.

## References

### Quick reference on functions

1. ?logic
2. ?all
3. ?any
4. ?which
5. ?":"
6. ?USPersonalExpenditure
7. ?subset
8. ?cut
9. ?levels

### Packages to read further on

1. String processing package: help(package = "stringi")

### R Manuals

#### An Introcduction to R sections:

- 2.3 Generating Regular Sequence
- 2.4 Logical vectors
- 2.5 Missing values
- 2.6 Character vectors
- 2.7 Index vectors: selecting and modifying subset of a data set
- 5.2 Array indexing, Susbsetting of array

- 5.3 Index matrices

## R Language Definition section

- 3.4 Indexing

## R FAQ sections

- 7.1 How can I set components of a list to NULL
- 7.10 How do I convert factors to numeric
- 7.21 How can I turn a string into a variable
- 7.23 How can I sort the rows of a data frame?
- 7.37 Why do backslash behave strangely inside strings?