Chapter 39: Subsetting

Given an R object, we may require separate analysis for one or more parts of the data contained in it. The process of obtaining these parts of the data from a given object is called `subsetting`.

Section 39.1: Data frames

Subsetting a data frame into a smaller data frame can be accomplished the same as subsetting a list.

```
> df3 <- data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = FALSE)

> df3
##   x y
## 1 1 a
## 2 2 b
## 3 3 c

> df3[1]   # Subset a variable by number
##   x
## 1 1
## 2 2
## 3 3

> df3["x"]   # Subset a variable by name
##   x
## 1 1
## 2 2
## 3 3

> is.data.frame(df3[1])
## TRUE

> is.list(df3[1])
## TRUE
```

Subsetting a dataframe into a column vector can be accomplished using double brackets [[ ]] or the dollar sign operator $.

```
> df3[[2]]    # Subset a variable by number using [[ ]]
## [1] "a" "b" "c"

> df3[["y"]]  # Subset a variable by name using [[ ]]
## [1] "a" "b" "c"

> df3$x    # Subset a variable by name using $
## [1] 1 2 3

> typeof(df3$x)
## "integer"

> is.vector(df3$x)
## TRUE
```

Subsetting a data as a two dimensional matrix can be accomplished using `i` and `j` terms.

```
> df3[1, 2]    # Subset row and column by number
## [1] "a"
```

```
> df3[1, "y"]  # Subset row by number and column by name
## [1] "a"

> df3[2, ]     # Subset entire row by number
##   x y
## 2 2 b

> df3[ , 1]    # Subset all first variables
## [1] 1 2 3

> df3[ , 1, drop = FALSE]
##   x
## 1 1
## 2 2
## 3 3
```

Note: Subsetting by `j` (column) alone simplifies to the variable's own type, but subsetting by `i` alone returns a **`data.frame`**, as the different variables may have different types and classes. Setting the **`drop`** parameter to `FALSE` keeps the data frame.

```
> is.vector(df3[, 2])
## TRUE

> is.data.frame(df3[2, ])
## TRUE

> is.data.frame(df3[, 2, drop = FALSE])
## TRUE
```

Section 39.2: Atomic vectors

Atomic vectors (which excludes lists and expressions, which are also vectors) are subset using the `[` operator:

```
# create an example vector
v1 <- c("a", "b", "c", "d")

# select the third element
v1[3]
## [1] "c"
```

The `[` operator can also take a vector as the argument. For example, to select the first and third elements:

```
v1 <- c("a", "b", "c", "d")

v1[c(1, 3)]
## [1] "a" "c"
```

Some times we may require to omit a particular value from the vector. This can be achieved using a negative sign(`-`) before the index of that value. For example, to omit to omit the first value from v1, use `v1[-1]`. This can be extended to more than one value in a straight forward way. For example, `v1[-c(1,3)]`.

```
> v1[-1]
[1] "b" "c" "d"
> v1[-c(1,3)]
[1] "b" "d"
```

On some occasions, we would like to know, especially, when the length of the vector is large, index of a particular

value, if it exists:

```
> v1=="c"
[1] FALSE FALSE  TRUE FALSE
> which(v1=="c")
[1] 3
```

If the atomic vector has names (a `names` attribute), it can be subset using a character vector of names:

```
v <- 1:3
names(v) <- c("one", "two", "three")

v
##  one   two three
##    1     2     3

v["two"]
## two
##   2
```

The `[[` operator can also be used to index atomic vectors, with differences in that it accepts a indexing vector with a length of one and strips any names present:

```
v[[c(1, 2)]]
## Error in v[[c(1, 2)]] :
##  attempt to select more than one element in vectorIndex

v[["two"]]
## [1] 2
```

Vectors can also be subset using a logical vector. In contrast to subsetting with numeric and character vectors, the logical vector used to subset has to be equal to the length of the vector whose elements are extracted, so if a logical vector `y` is used to subset `x`, i.e. `x[y]`, if `length(y) < length(x)` then `y` will be recycled to match `length(x)`:

```
v[c(TRUE, FALSE, TRUE)]
##  one three
##    1     3

v[c(FALSE, TRUE)]  # recycled to 'c(FALSE, TRUE, FALSE)'
## two
##   2

v[TRUE]   # recycled to 'c(TRUE, TRUE, TRUE)'
##  one   two three
##    1     2     3

v[FALSE]   # handy to discard elements but save the vector's type and basic structure
## named integer(0)
```

Section 39.3: Matrices

For each dimension of an object, the `[` operator takes one argument. Vectors have one dimension and take one argument. Matrices and data frames have two dimensions and take two arguments, given as `[i, j]` where `i` is the row and `j` is the column. Indexing starts at 1.

```
## a sample matrix
mat <- matrix(1:6, nrow = 2, dimnames = list(c("row1", "row2"), c("col1", "col2", "col3")))
```

```
mat
#      col1 col2 col3
# row1    1    3    5
# row2    2    4    6
```

`mat[i, j]` is the element in the `i`-th row, `j`-th column of the matrix `mat`. For example, an `i` value of `2` and a `j` value of `1` gives the number in the second row and the first column of the matrix. Omitting `i` or `j` returns all values in that dimension.

```
mat[ , 3]
## row1 row2
##    5    6

mat[1, ]
# col1 col2 col3
#    1    3    5
```

When the matrix has row or column names (not required), these can be used for subsetting:

```
mat[ , 'col1']
# row1 row2
#    1    2
```

By default, the result of a subset will be simplified if possible. If the subset only has one dimension, as in the examples above, the result will be a one-dimensional vector rather than a two-dimensional matrix. This default can be overridden with the `drop = FALSE` argument to `[`:

```
## This selects the first row as a vector
class(mat[1, ])
# [1] "integer"

## Whereas this selects the first row as a 1x3 matrix:
class(mat[1, , drop = F])
# [1] "matrix"
```

Of course, dimensions cannot be dropped if the selection itself has two dimensions:

```
mat[1:2, 2:3]  ## A 2x2 matrix
#      col2 col3
# row1    3    5
# row2    4    6
```

Selecting individual matrix entries by their positions

It is also possible to use a Nx2 matrix to select N individual elements from a matrix (like how a coordinate system works). If you wanted to extract, in a vector, the entries of a matrix in the `(1st row, 1st column), (1st row, 3rd column), (2nd row, 3rd column), (2nd row, 1st column)` this can be done easily by creating a index matrix with those coordinates and using that to subset the matrix:

```
mat
#      col1 col2 col3
# row1    1    3    5
# row2    2    4    6

ind = rbind(c(1, 1), c(1, 3), c(2, 3), c(2, 1))
ind
#      [,1] [,2]
```

```
# [1,]    1    1
# [2,]    1    3
# [3,]    2    3
# [4,]    2    1

mat[ind]
# [1] 1 5 6 2
```

In the above example, the 1st column of the `ind` matrix refers to rows in `mat`, the 2nd column of `ind` refers to columns in `mat`.

Section 39.4: Lists

A list can be subset with `[`:

```
l1 <- list(c(1, 2, 3), 'two' = c("a", "b", "c"), list(10, 20))
l1
## [[1]]
## [1] 1 2 3
##
## $two
## [1] "a" "b" "c"
##
## [[3]]
## [[3]][[1]]
## [1] 10
##
## [[3]][[2]]
## [1] 20

l1[1]
## [[1]]
## [1] 1 2 3

l1['two']
## $two
## [1] "a" "b" "c"

l1[[2]]
## [1] "a" "b" "c"

l1[['two']]
## [1] "a" "b" "c"
```

Note the result of `l1[2]` is still a list, as the `[` operator selects elements of a list, returning a smaller list. The `[[` operator extracts list elements, returning an object of the type of the list element.

Elements can be indexed by number or a character string of the name (if it exists). Multiple elements can be selected with `[` by passing a vector of numbers or strings of names. Indexing with a vector of `length > 1` in `[` and `[[` returns a "list" with the specified elements and a recursive subset (if available), *respectively*:

```
l1[c(3, 1)]
## [[1]]
## [[1]][[1]]
## [1] 10
##
## [[1]][[2]]
## [1] 20
##
```

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

Compared to:

```
l1[[c(3, 1)]]
## [1] 10
```

which is equivalent to:

```
l1[[3]][[1]]
## [1] 10
```

The `$` operator allows you to select list elements solely by name, but unlike `[` and `[[`, does not require quotes. As an infix operator, `$` can only take a single name:

```
l1$two
## [1] "a" "b" "c"
```

Also, the `$` operator allows for partial matching by default:

```
l1$t
## [1] "a" "b" "c"
```

in contrast with `[[` where it needs to be specified whether partial matching is allowed:

```
l1[["t"]]
## NULL
l1[["t", exact = FALSE]]
## [1] "a" "b" "c"
```

Setting `options(warnPartialMatchDollar = TRUE)`, a "warning" is given when partial matching happens with `$`:

```
l1$t
## [1] "a" "b" "c"
## Warning message:
## In l1$t : partial match of 't' to 'two'
```

Section 39.5: Vector indexing

For this example, we will use the vector:

```
> x <- 11:20
> x
 [1] 11 12 13 14 15 16 17 18 19 20
```

R vectors are 1-indexed, so for example `x[1]` will return `11`. We can also extract a sub-vector of `x` by passing a vector of indices to the bracket operator:

```
> x[c(2,4,6)]
[1] 12 14 16
```

If we pass a vector of negative indices, R will return a sub-vector with the specified indices excluded:

```
> x[c(-1,-3)]
[1] 12 14 15 16 17 18 19 20
```

We can also pass a boolean vector to the bracket operator, in which case it returns a sub-vector corresponding to the coordinates where the indexing vector is `TRUE`:

```
> x[c(rep(TRUE,5),rep(FALSE,5))]
[1] 11 12 13 14 15 16
```

If the indexing vector is shorter than the length of the array, then it will be repeated, as in:

```
> x[c(TRUE,FALSE)]
[1] 11 13 15 17 19
> x[c(TRUE,FALSE,FALSE)]
[1] 11 14 17 20
```

Section 39.6: Other objects

The `[` and `[[` operators are primitive functions that are generic. This means that any *object* in R (specifically `isTRUE(is.object(x))` --i.e. has an explicit "class" attribute) can have its own specified behaviour when subsetted; i.e. has its own *methods* for `[` and/or `[[`.

For example, this is the case with "data.frame" (`is.object(iris)`) objects where `[.data.frame` and `[[.data.frame` methods are defined and they are made to exhibit both "matrix"-like and "list"-like subsetting. With forcing an error when subsetting a "data.frame", we see that, actually, a function `[.data.frame` was called when we -just- used `[`.

```
iris[invalidArgument, ]
## Error in `[.data.frame`(iris, invalidArgument, ) :
##   object 'invalidArgument' not found
```

Without further details on the current topic, an example`[` method:

```
x = structure(1:5, class = "myClass")
x[c(3, 2, 4)]
## [1] 3 2 4
'[.myClass' = function(x, i) cat(sprintf("We'd expect '%s[%s]' to be returned but this a custom `[`
method and should have a `?[.myClass` help page for its behaviour\n", deparse(substitute(x)),
deparse(substitute(i))))

x[c(3, 2, 4)]
## We'd expect 'x[c(3, 2, 4)]' to be returned but this a custom `[` method and should have a
`?[.myClass` help page for its behaviour
## NULL
```

We can overcome the method dispatching of `[` by using the equivalent non-generic `.subset` (and `.subset2` for `[[`). This is especially useful and efficient when programming our own "class"es and want to avoid work-arounds (like `unclass(x)`) when computing on our "class"es efficiently (avoiding method dispatch and copying objects):

```
.subset(x, c(3, 2, 4))
## [1] 3 2 4
```

Section 39.7: Elementwise Matrix Operations

Let A and B be two matrices of same dimension. The operators `+,-,/,*,^` when used with matrices of same dimension perform the required operations on the corresponding elements of the matrices and return a new

matrix of the same dimension. These operations are usually referred to as element-wise operations.

Operator	A op B	Meaning
+	A + B	Addition of corresponding elements of A and B
-	A - B	Subtracts the elements of B from the corresponding elements of A
/	A / B	Divides the elements of A by the corresponding elements of B
*	A * B	Multiplies the elements of A by the corresponding elements of B
^	A^(-1)	For example, gives a matrix whose elements are reciprocals of A

For "true" matrix multiplication, as seen in *Linear Algebra*, use `%*%`. For example, multiplication of A with B is: `A %*% B`. The dimensional requirements are that the `ncol()` of `A` be the same as `nrow()` of `B`

Some Functions used with Matrices

Function	Example	Purpose
nrow()	nrow(A)	determines the number of rows of A
ncol()	ncol(A)	determines the number of columns of A
rownames()	rownames(A)	prints out the row names of the matrix A
colnames()	colnames(A)	prints out the column names of the matrix A
rowMeans()	rowMeans(A)	computes means of each row of the matrix A
colMeans()	colMeans(A)	computes means of each column of the matrix A
upper.tri()	upper.tri(A)	returns a vector whose elements are the upper triangular matrix of square matrix A
lower.tri()	lower.tri(A)	returns a vector whose elements are the lower triangular matrix of square matrix A
det()	det(A)	results in the determinant of the matrix A
solve()	solve(A)	results in the inverse of the non-singular matrix A
diag()	diag(A)	returns a diagonal matrix whose off-diagnal elemts are zeros and diagonals are the same as that of the square matrix A
t()	t(A)	returns the the transpose of the matrix A
eigen()	eigen(A)	retuens the eigenvalues and eigenvectors of the matrix A
is.matrix()	is.matrix(A)	returns TRUE or FALSE depending on whether A is a matrix or not.
as.matrix()	as.matrix(x)	creates a matrix out of the vector x