R Data Structures

R.W. Oldford
Data structures in R

The base data structures in R:

<table>
<thead>
<tr>
<th>dimensionality</th>
<th>homogeneous contents</th>
<th>heterogeneous contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d</td>
<td>Atomic vector</td>
<td>List</td>
</tr>
<tr>
<td>2d</td>
<td>Matrix</td>
<td>Data frame</td>
</tr>
<tr>
<td>nd</td>
<td>Array</td>
<td></td>
</tr>
</tbody>
</table>

Note there are no scalar or 0-dimensional data structures. Instead these are 1d data structures having a single element.

There are also three different types of object-oriented programming systems in R (S3, S4, and reference classes) which can be used to construct more complex data types.

The function `str()` can be used to reveal the contents of any R data structure.
Data structures in R – Vectors

The basic data structure is a “vector”
Data structures in R – Vectors

The basic data structure is a “vector”

Two kinds:
Data structures in R – Vectors

The basic data structure is a “vector”

Two kinds: atomic vectors
Data structures in R – Vectors

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.
The basic data structure is a “vector”
Two kinds: atomic vectors and lists.
Three properties:
Data structures in R – Vectors

The basic data structure is a “vector”
Two kinds: atomic vectors and lists.
Three properties:

- its type, `typeof()`
The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

- its type, typeof()
- the number of elements it has, length()
Data structures in R – Vectors

The basic data structure is a “vector”
Two kinds: atomic vectors and lists.

Three properties:

- its type, typeof()
- the number of elements it has, length()
- a place for arbitrary additional properties, attributes()
Data structures in R – Vectors

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

▶ its type, typeof()
▶ the number of elements it has, length()
▶ a place for arbitrary additional properties, attributes()

Elements of an atomic vector must all be of the same type.
Data structures in R – Vectors

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

▶ its type, typeof()
▶ the number of elements it has, length()
▶ a place for arbitrary additional properties, attributes()

Elements of an atomic vector must all be of the same type.

Elements of a list can be of different types.
Data structures in R – Vectors

The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

▶ its type, typeof()
▶ the number of elements it has, length()
▶ a place for arbitrary additional properties, attributes()

Elements of an **atomic vector** must all be of the **same** type.

Elements of a **list** can be of different types.

Constructors: `c()` for atomic vectors, `list()` for lists.
The basic data structure is a “vector”

Two kinds: atomic vectors and lists.

Three properties:

▶ its type, typeof()
▶ the number of elements it has, length()
▶ a place for arbitrary additional properties, attributes()

Elements of an atomic vector must all be of the same type.

Elements of a list can be of different types.

Constructors: c() for atomic vectors, list() for lists.

tests: is.atomic() and is.list().
Data structures in R – `c()` constructing atomic vectors

Atomic vectors are constructed using `c()` (c for “combine”)

```r
x <- c(1, 2, 3)
x
```

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

```r
is.atomic(x)
```

```r
## [1] TRUE
```

```r
is.list(x)
```

```r
## [1] FALSE
```

Atomic vectors are always “flat”

```r
y <- c(x, x, 4, 5, 6)
y
```

```r
## [1] 1 2 3 1 2 3 4 5 6
```

```r
c(x, y, c(7, 8, y, 9, c(10, 11)))
```

```r
## [1] 1 2 3 1 2 3 1 2 3 4 5 6 7 8 1 2 3 1 2 3 1 2 3 4 5 6
## [24] 9 10 11
```
Data structures in R – `c()` constructing atomic vectors

Elements of an atomic vector are accessed using the [] operator (see ?"[")

```r
x <- c("a", "b", "c", "d", "e", "f")
x[3]
```

```r
## [1] "c"
```

```r
x[c(1,3,5)]
```

```r
## [1] "a" "c" "e"
```

And set with the same function

```r
x[1] <- "EH"
x
```

```r
## [1] "EH" "b" "c" "d" "e" "f"
```

```r
x[c(3,5)] <- c("third", "fifth")
x
```

```r
## [1] "EH" "b" "third" "d" "fifth" "f"
```
Data structures in R – double precision numeric vectors

```r
x <- c(1, 2, 3)
length(x)

## [1] 3
typeof(x)

## [1] "double"
attributes(x)

## NULL
is.atomic(x)

## [1] TRUE
is.numeric(x)

## [1] TRUE
is.double(x)

## [1] TRUE
```
Data structures in R – integer numeric vectors

```r
x <- c(1L, 20L, 3L)  # "longs"
length(x)
```

```r
## [1] 3
```

```r
typeof(x)
```

```r
## [1] "integer"
```

```r
attributes(x)
```

```r
## NULL
```

```r
is.atomic(x)
```

```r
## [1] TRUE
```

```r
is.numeric(x)
```

```r
## [1] TRUE
```

```r
is.integer(x)
```

```r
## [1] TRUE
```
Data structures in R – logical vectors

```r
x <- c(T, F, TRUE, T, FALSE, T)
length(x)

## [1] 6

typeof(x)

## [1] "logical"

attributes(x)

## NULL

is.atomic(x)

## [1] TRUE

is.numeric(x)

## [1] FALSE

is.logical(x)

## [1] TRUE
```
Data structures in R – character vectors

```r
x <- c("Now", "is the time", "for", "all")
length(x)

## [1] 4
typeof(x)

## [1] "character"
attributes(x)

## NULL
is.atomic(x)

## [1] TRUE
is.numeric(x)

## [1] FALSE
is.character(x)

## [1] TRUE
```
Data structures in R – type contagion

From least to most flexible vector types are: logical, integer, double, and character.

Elements are coerced to be of the same type (the most flexible).

```r
typeof(c(FALSE, T))
```

```r
## [1] "logical"
```

```r
typeof(c(FALSE, T, 2L))
```

```r
## [1] "integer"
```

```r
typeof(c(FALSE, T, 2L, 3))
```

```r
## [1] "double"
```

```r
typeof(c(FALSE, T, 2L, 3, "four"))
```

```r
## [1] "character"
```

```r
c(FALSE, T, 2L, 3, "four")
```

```r
## [1] "FALSE" "TRUE" "2" "3" "four"
```

All elements are automatically coerced to be strings.
Data structures in R – coercion

Can force the coercion using `as.numeric()`, `as.numeric()`, `as.double()`, `as.integer()`, or `as.logical()`

```r
as.numeric(c(FALSE, T, TRUE, F, F))
```

```r
## [1] 0 1 1 0 0
```

```r
as.double(c(FALSE, T, TRUE, F, F))
```

```r
## [1] 0 1 1 0 0
```

```r
as.integer(c(FALSE, T, TRUE, F, F))
```

```r
## [1] 0 1 1 0 0
```

```r
as.character(c(FALSE, T, TRUE, F, F))
```

```r
## [1] "FALSE" "TRUE" "TRUE" "FALSE" "FALSE"
```

```r
as.logical(c(0, 1, 2.3, 4.5, 6))
```

```r
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Note that many functions will force their argument to the required type.
E.g. `sum()` forces its argument to be numeric, logical operators `&`, `|`, etc. force theirs to be `logical`. 
Data structures in R – coercion

Forcing coercion can result in the loss of information and can give some strange answers:

```r
as.numeric(c(FALSE, T, 2L, 3))
```

```r
## [1] 0 1 2 3
```

```r
as.numeric(c(FALSE, T, 2L, 3, "four"))
```

```r
## Warning: NAs introduced by coercion
## [1] NA NA 2 3 NA
```

```r
as.numeric(c(as.numeric(c(FALSE, T, 2L, 3)), "four"))
```

```r
## Warning: NAs introduced by coercion
## [1] 0 1 2 3 NA
```

Note that warnings are given.
Data structures in R – vectors

Can also produce a vector (possibly to be modified later) by specifying its type (mode) and length:

```r
x <- vector(mode = "double", length = 3)
x
```

```r
## [1] 0 0 0
```

```r
y <- vector(mode = "logical", length = 3)
y
```

```r
## [1] FALSE FALSE FALSE
```

```r
z <- vector(mode = "character", length = 3)
z
```

```r
## [1] "" "" ""
```

And
Data structures in R – Lists

Elements of lists can be of any type:

```r
x <- list("a", c(2, 3, 4), c(T,F), c("b", "c", "d", 56))
x
```

```
[[1]]
[1] "a"
[[2]]
[1] 2 3 4
[[3]]
[1] TRUE FALSE
[[4]]
[1] "b" "c" "d" "56"
```

```r
str(x)
```

```
List of 4
$ : chr "a"
$ : num [1:3] 2 3 4
$ : logi [1:2] TRUE FALSE
$ : chr [1:4] "b" "c" "d" "56"
```

```r
attributes(x)
```

```
NULL
```
Data structures in R – Lists

Elements of lists can accessed in a few ways:

```r
x[2]
```

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

```r
typeof(x[2])
```

```r
## [1] "list"
```

```r
length(x[2])
```

```r
## [1] 1
```

```r
x[[2]]
```

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

```r
typeof(x[[2]])
```

```r
## [1] "double"
```

And can be created using `vector()`, default elements being `NULL` (being an “empty” vector)

```r
vector(mode = "list", length = 3)[[1]]
```

```r
## NULL
```
Data structures in R – Lists are recursive

```
x <- list("one", 2)
is.recursive(x)

## [1] TRUE

Lists can be arbitrarily recursive:

y <- list(x, list(3, x))
is.recursive(y)

## [1] TRUE

str(y)
```

```
## List of 2
## $ :List of 2
## ..$ : chr "one"
## ..$ : num 2
## $ :List of 2
## ..$ : num 3
## ..$ :List of 2
## .. ..$ : chr "one"
## .. ..$ : num 2
```
Data structures in R – Vectors can have named components

Atomic vectors

```r
x <- c(a=3, b=4, 5)
x
```

```r
## a b
## 3 4 5
```

```r
names(x)
```

```r
## [1] "a" "b" ""
```

```r
x["a"]
```

```r
## a
## 3
```

```r
attributes(x)
```

```r
## $names
## [1] "a" "b" ""
```
Data structures in R – Vectors can have named components

Lists (or “recursive vectors”)

```r
x <- list(a=c(1,2,3), b=4, 5)
str(x)

## List of 3
## $ a: num [1:3] 1 2 3
## $ b: num 4
## $ : num 5

x["b"]

## $b
## [1] 4

x$b

## [1] 4

names(x)

## [1] "a" "b" ""
```
Data structures in R – Lists as general data structures

Lists (handy with named components) can be used to create more general structures. For example,

```r
# data frames:
is.list(cars)

## [1] TRUE

names(cars)

## [1] "speed" "dist"

# results of functions
fit <- lm(dist ~ speed, data = cars)
is.list(fit)

## [1] TRUE

names(fit)

## [1] "coefficients" "residuals" "effects" "rank"
## [5] "fitted.values" "assign" "qr" "df.residual"
## [9] "xlevels" "call" "terms" "model"
Data structures in R – Lists to atomic vectors

Sometimes it can be handy to change a list to an atomic vector:

```r
x <- list(a = 1, b = c(2, 3, 4), c = list(e = c(5, 6), f = c(7, 8, 9)))
str(x)
```

```r
## List of 3
## $ a: num 1
## $ b: num [1:3] 2 3 4
## $ c:List of 2
##   ..$ e: num [1:2] 5 6
##   ..$ f: num [1:3] 7 8 9
```

```r
y <- unlist(x)
is.list(y)
```

```r
## [1] FALSE
```

```r
y
```

```r
## a b1 b2 b3 c.e1 c.e2 c.f1 c.f2 c.f3
## 1 2 3 4 5 6 7 8 9
```

```r
names(y)
```

```r
## [1] "a" "b1" "b2" "b3" "c.e1" "c.e2" "c.f1" "c.f2" "c.f3"
```
Data structures in R – attributes

Additional information can be added to any R object as one or more attributes.

▶ these are very much like a property list (or plist) in other languages.

```r
y
>> a  b1  b2  b3  c.e1  c.e2  c.f1  c.f2  c.f3
>> 1   2   3   4   5   6   7   8   9

names(y)

>> [1] "a" "b1" "b2" "b3" "c.e1" "c.e2" "c.f1" "c.f2" "c.f3"

# introduce a specific attribute
attr(y, "originalNames") <- c("a", "b", "c", "d", "e", "f")
attr(y, "originalNames")

>> [1] "a" "b" "c" "d" "e" "f"

which provides a simple cache of the component names from the original list hierarchy.
Data structures in R – attributes

Using `attr()` to set a specific named attribute on the object modifies that object:

```r
tau <- 2 * pi
attr(tau, "description") <- "Twice pi"
attributes(tau)
```

```
## $description
## [1] "Twice pi"
```

`structure()` is similar but returns a copy:

```r
TwoPi <- structure(tau, description = "Two times pi")
attributes(TwoPi)
```

```
## $description
## [1] "Two times pi"
```

# TwoPi is a different object having the same value as tau
# which can be checked here since it is just a numeric vector

```r
tau == TwoPi
```

```
## [1] TRUE
```
Of course, assigning a value to `attributes()` will change all attributes on that object.

```r
attributes(tau)
```

```r
## $description
## [1] "Twice pi"
```

```r
attributes(tau) <- list(constant = "tau")
```

and we have lost the previous attributes.

```r
attributes(tau)
```

```r
## $constant
## [1] "tau"
```
Data structures in R – factors

A factor is a special kind of atomic vector having class `factor` and an attribute called `levels`. A factor is typically used to represent a categorical variate having a fixed, finite, and known set of values. The possible values are assigned to the factor levels.

```r
r
# treatment example

treatment <- factor(c("a", "b", "b", "b", "a", "c", "c", "a", "b"))
levels(treatment)

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

table(treatment)

## treatment
## a b c
## 3 4 2

# levels are fixed

# Warning in '[<-.factor`(`*tmp*`, 3, value = "d")': invalid factor level, NA

# generated

treatment[3] <- "d"

# Warning in `[<-.factor`(`*tmp*`, 3, value = "d")`: invalid factor level, NA

# generated

treatment

## [1] a b <NA> b a c c a b
## Levels: a b c
```
Data structures in R – factors

A factor is an atomic vector,

```r
is.atomic(treatment)
## [1] TRUE

typeof(treatment)
## [1] "integer"

of type integer having a length and attributes

```r
length(treatment)
## [1] 9

```r
attributes(treatment)

## $levels
## [1] "a" "b" "c"

## $class
## [1] "factor"
These attributes can be accessed via two functions

\begin{verbatim}
levels(treatment)

## [1] "a" "b" "c"
\end{verbatim}

\begin{verbatim}
class(treatment)

## [1] "factor"
\end{verbatim}
Data structures in R – factors

All factor levels need to be known, but need not appear:

```r
answer <- factor(c(1, 2, 1, 1, 2, 2, 1, 1, 2), levels = c(1, 2, 3))
table(answer)
```

```r
## answer
## 1 2 3
## 5 4 0
```

Again, `answer` is a factor and not a numeric vector:

```r
is.integer(answer)
```

```r
## [1] FALSE
```
Data structures in R – factors

Though a factor is an atomic vector of type integer, it does not test positive as an integer

```r
is.integer(treatment)
```

## [1] FALSE

Nor does it test positive as a vector

```r
is.vector(treatment)
```

## [1] FALSE

The problem is that `is.vector()` returns TRUE only if its argument is a vector and it has no attributes except (possibly) a names attribute.

```r
names(treatment)
```

## NULL

It is a factor

```r
is.factor(treatment)
```

## [1] TRUE
Data structures in R – factors

A factor could be coerced to a vector

\texttt{as.vector(treatment)}

\begin{verbatim}
## [1] "a" "b" NA  "b" "a" "c" "c" "a" "b"
\end{verbatim}

\texttt{typeof(treatment)}

\begin{verbatim}
## [1] "integer"
\end{verbatim}

but loses the levels information. Combining \texttt{factor}s will perform a similar coercion:

\texttt{c(treatment)}

\begin{verbatim}
## [1] 1 2 NA 2 1 3 3 1 2
\end{verbatim}

\texttt{is.integer(c(treatment))}

\begin{verbatim}
## [1] TRUE
\end{verbatim}
Data structures in R – accidental factors

Often, we read in data from, say, a \texttt{.csv} file where we might be expecting only numerical values. For example, the \texttt{csv} file might have contents like this:

\begin{verbatim}
x, y
5, 3
7, -
8, 2
\end{verbatim}

The dash "−" is a mistake in coding the data, or perhaps codes missing data. Read in (using \texttt{read.csv()}) the result assigned to \texttt{data}:

```r
data <- \texttt{read.csv(file = "some_file_somewhere.csv")}  # for example
```

the dash − will be located in the same place in \texttt{data}.

```r
data
```

```
##     x y
## 1  5 3
## 2  7 -
## 3  8 2
```

Where we might have expected a numeric atomic vector, \texttt{data$y} is a \texttt{factor} (Why?)

```r
data$y
```

```
## [1] 3 - 2
## Levels: - 2 3
```
Data structures in R – matrices and arrays

An atomic vector can be made to act like a matrix (or array) by simply adding an attribute `dim` to record the matrix (or array) dimensions.

```r
x <- 1:12
dim(x) <- c(2,6)
x

## [1,]  1  3  5  7  9 11
## [2,]  2  4  6  8 10 12
class(x)

## [1] "matrix"

attributes(x)

## $dim
## [1] 2 6
typeof(x)

## [1] "integer"

length(x)

## [1] 12
Data structures in R – matrices and arrays

An atomic vector can be made to act like a matrix (or array) by simply adding a attribute `dim` to record the matrix (or array) dimensions.

```r
x <- 1:12
dim(x) <- c(2,3,2)
x
```

```
## , , 1
## [,1] [,2] [,3]
## [1,] 1 3 5
## [2,] 2 4 6

## , , 2
## [,1] [,2] [,3]
## [1,] 7 9 11
## [2,] 8 10 12
```

```r
class(x)
```

```
[1] "array"
```

```r
attributes(x)
```

```
$dim
[1] 2 3 2
```
There are also special constructor functions

```r
# for matrices
x <- matrix(1:4, nrow = 3, ncol = 4)
x
```

## 

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,]</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>[2,]</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>[3,]</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

```r
dim(x)
```

##

| [1] 3 4 |

```r
c(nrow(x), ncol(x))
```

##

| [1] 3 4 |

```r
length(x)
```

##

| [1] 12 |
Data structures in R – matrices and arrays

There are also special constructor functions

```r
# For arrays
y <- array(1:8, dim = c(2,3,2))
y
```

```
## , , 1
## [,1] [,2] [,3]
## [1,] 1 3 5
## [2,] 2 4 6

## , , 2
## [,1] [,2] [,3]
## [1,] 7 1 3
## [2,] 8 2 4
```

```r
dim(y)
```

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

```r
c(nrow(y), ncol(y))
```

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

```r
length(y)
```

```
## [1] 12
```
Data structures in R – matrices and arrays

Again, these are atomic vectors

```r
y <- array(1:6, dim = c(2,3,2))
is.atomic(y)
```

```r
## [1] TRUE
```

```r
length(y)
```

```r
## [1] 12
```

```r
typeof(y)
```

```r
## [1] "integer"
```

```r
attributes(y)
```

```r
## $dim
## [1] 2 3 2
```

```r
is.vector(y) # explain this result
```

```r
## [1] FALSE
```
Data structures in R – matrices and arrays

**Subsetting**

```
x

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

x[c(1,2), c(2,4)]

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

x[c(T,F,T), c(F,T,T,F)]

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

x[-2,]

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

x[2,]

## [1] 2 1 4 3
```
Data structures in R – matrices and arrays

Dimensions are dropped by default:

```r
x
```

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

```r
x[2,]
```

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

```r
dim(x[2,])
```

```r
## NULL
```

```r
x[2,, drop = FALSE]  # preserve the dimensions
```

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

```r
dim(x[2,, drop = FALSE])
```

```r
## [1] 1 4
```
Data structures in R – matrices and arrays

Matrices/arrays are still indexable as vectors:

```r
x

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

x[7]

## [1] 3

x[-7]

## [1] 1 2 3 4 1 2 4 1 2 3 4

x[[5]]

## [1] 1
```

See `help("["`) for details.
Data structures in R – matrices and arrays

Subsetting is same on arrays

```r
y

## , , 1
## [,1] [,2] [,3]
## [1,] 1 3 5
## [2,] 2 4 6

## , , 2
## [,1] [,2] [,3]
## [1,] 1 3 5
## [2,] 2 4 6

y[1,2,1, drop=FALSE]

## , , 1
## [,1]
## [1,] 3

y[1,2,]

## [1] 3 3

y[,-2,1]

## [,1] [,2]
## [1,] 1 5
## [2,] 2 6
```
The transpose \( t() \) switches the rows and columns of a matrix. That is it permutes the indices:

\[
\begin{align*}
\mathbf{x} & \\
\text{## } & [1,] & 1 & 4 & 3 & 2 \\
\text{## } & [2,] & 2 & 1 & 4 & 3 \\
\text{## } & [3,] & 3 & 2 & 1 & 4 \\
\end{align*}
\]

\[
\begin{align*}
\mathbf{t(x)} & \\
\text{## } & [,1] & [,2] & [,3] \\
\text{## } & [1,] & 1 & 2 & 3 \\
\text{## } & [2,] & 4 & 1 & 2 \\
\text{## } & [3,] & 3 & 4 & 1 \\
\text{## } & [4,] & 2 & 3 & 4 \\
\end{align*}
\]
Data structures in R – matrices and arrays

For arrays, `aperm()` can be used to permute indices:

```R
y

### , , 1
###     [,1] [,2] [,3]
### [1,]  1  3  5
### [2,]  2  4  6
### , , 2
###     [,1] [,2] [,3]
### [1,]  1  3  5
### [2,]  2  4  6

aperm(y, perm = c(2,1,3))

### , , 1
###     [,1] [,2]
### [1,]  1  2
### [2,]  3  4
### [3,]  5  6
### , , 2
###     [,1] [,2]
### [1,]  1  2
### [2,]  3  4
### [3,]  5  6
```
Data structures in R – matrices and arrays

Just as vectors are combined with `c()`, conformable matrices can be combined using `cbind()` and `rbind()` (column and row binding).

```r
x
cbind(x, 11:13, x[,3:4])
rbind(x, 11:14)
```
An exception to conformable matrices when using `cbind()` and `rbind()` is the case when one of the arguments is a scalar.

```r
x
```

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

```r
cbind(x, 1000, x)
```

```r
## [1,] 1 4 3 2 1000 1 4 3 2
## [2,] 2 1 4 3 1000 2 1 4 3
## [3,] 3 2 1 4 1000 3 2 1 4
```
Data structures in R – matrices and arrays

Similarly for arrays, the combine constructor `abind()` (from the package `abind`) is available to extend an array along different dimensions

```r
library(abind)
abind(x, 11:13, x[,3:4], along = 2)
```

```
[1,] 1 4 3 2 11 3 2
[2,] 2 1 4 3 12 4 3
[3,] 3 2 1 4 13 1 4
```

```r
abind(y, 100 * x[c(2,3),c(3:4)], along = 2)
```

```
, , 1

[1,] 1 3 5 400
[2,] 2 4 6 100
, , 2

[1,] 1 3 5 300
[2,] 2 4 6 400
```

Note that the default `along` is the last dimension and `along = 0` will create a new dimension at the front.
Data structures in R – matrices and arrays

Can also give names to the rows and columns of a matrix

```r
rownames(x) <- c("A", "B", "C")
colnames(x) <- c("one", "two", "three", "four")
x
```

```
## one two three four
## A 1 4 3 2
## B 2 1 4 3
## C 3 2 1 4
```

```r
x[c("A"), c("one", "three")]
```

```
## one three
## 1 3
```

```r
x[c("A"), c("one", "three"), drop = FALSE]
```

```
## one three
## A 1 3
```

```r
x[-1, c("one", "three")]
```

```
## one three
## B 2 4
## C 3 1
```

Note that `x[-c("A"), c("one", "three")]]` will fail. Negation requires numerical indices (or logicals).
Data structures in R – matrices and arrays

Can also give names to the dimensions of an array

```r
dimnames(y) <- list(c("row 1", "row 2"),
                    c("col 1", "col 2", "col 3"),
                    c("slice 1", "slice 2"))
```

```r
y
```

```r
## , , slice 1
##    col 1 col 2 col 3
## row 1  1   3   5
## row 2  2   4   6
## , , slice 2
##    col 1 col 2 col 3
## row 1  1   3   5
## row 2  2   4   6
```

```r
str(y)
```

```r
## int [1:2, 1:3, 1:2] 1 2 3 4 5 6 1 2 3 4 ... 
## - attr(*, "dimnames")=List of 3
## ..$ : chr [1:2] "row 1" "row 2"
## ..$ : chr [1:3] "col 1" "col 2" "col 3"
## ..$ : chr [1:2] "slice 1" "slice 2"
```

dimnames(x) will also work.
Data structures in R – Data frames

A data frame is a list of equal-length vectors:

```r
data <- data.frame(x = 1:3, y = 4:6, z=c("one", "two", "three"))
str(data)
```

```r
## 'data.frame': 3 obs. of 3 variables:
## $ x: int 1 2 3
## $ y: int 4 5 6
## $ z: Factor w/ 3 levels "one","three",..: 1 3 2
```

Note that the strings are turned into factors. This can be suppressed:

```r
data_nofactor <- data.frame(x = 1:3, y = 4:6, z=c("one", "two", "three"), stringsAsFactors = FALSE)
str(data_nofactor)
```

```r
## 'data.frame': 3 obs. of 3 variables:
## $ x: int 1 2 3
## $ y: int 4 5 6
## $ z: chr "one" "two" "three"
```

A data frame can be thought of as a rectangular structure where each column is a variate and each row an observation. So it is similar to (but not the same as) a matrix.
Data structures in R – Data frames

As a rectangular structure, a data frame behaves much like a matrix and, being a list, behaves much like one of those. Consider selection operators

```R
data

## x  y  z
## 1  1  4  one
## 2  2  5  two
## 3  3  6  three

data[1,]

## x  y  z
## 1  1  4  one

data[,"x"]

## [1] 1 2 3

data$z

## [1] one two three
## Levels: one three two
```
Data structures in R – Data frames

As a rectangular structure, combining (and other matrix operators) should work. Effect of adding observations (rows).

```r
data

## x y z
## 1 1 4 one
## 2 2 5 two
## 3 3 6 three

moredata <- rbind(data, data[2,], c(1:(ncol(data)-1), "four"))

## Warning in `[-.factor`(`*tmp*`, ri, value = "four"): invalid factor level,
## NA generated

moredata

## x y z
## 1 1 4 one
## 2 2 5 two
## 3 3 6 three
## 21 2 5 two
## 5 1 2 <NA>

rownames(moredata)

## [1] "1" "2" "3" "21" "5"
```

Notes: 1. coercion and warning for the last row, and 2. fourth row name.
Data structures in R – Data frames

Adding variates (columns).

```r
data

## x y z
## 1 1 4 one
## 2 2 5 two
## 3 3 6 three

moredata <- cbind(data, 100 * data, new = 1000)

## Warning in Ops.factor(left, right): '*' not meaningful for factors

moredata

## x y z x y z new
## 1 1 4 one 100 400 NA 1000
## 2 2 5 two 200 500 NA 1000
## 3 3 6 three 300 600 NA 1000

colnames(moredata)

## [1] "x" "y" "z" "x" "y" "z" "new"

moredata$x

## [1] 1 2 3
```

Notes: 1. warning for multiplication, and 2. effect on column names.
Data structures in R – Data frames

Data frames are vectors

data

## x   y   z
## 1 1 4   one
## 2 2 5   two
## 3 3 6   three

attributes(data)

## $names
## [1] "x" "y" "z"
##
## $row.names
## [1] 1 2 3
##
## $class
## [1] "data.frame"

length(data)

## [1] 3

typeof(data)

## [1] "list"
Data structures in R – Data frames

Data frames are lists

data
dataframe

# x y z
# 1 1 4 one
# 2 2 5 two
# 3 3 6 three
class(data)

# [1] "data.frame"
is.data.frame(data)

# [1] TRUE
is.list(data)

# [1] TRUE
is.vector(data)  # why?

# [1] FALSE
Data structures in R – Data frames

Data frames are lists

data
data$y

data$x
data[[1]]
Data structures in R – Data frames

Provided they have the same length, each variate/component can be any atomic vector or list.

```r
data_zAsList <- data.frame(x = 1:3, y = 4:6, z=list(1:3, 4:6, 7:9))
str(data_zAsList)
```

```
## 'data.frame': 3 obs. of 5 variables:
## $ x  : int 1 2 3
## $ y  : int 4 5 6
## $ z.1.3: int 1 2 3
## $ z.4.6: int 4 5 6
## $ z.7.9: int 7 8 9
```

But there is a problem.

The data frame has 5 instead of 3 variates.

The problem is that `data.frame()` interprets, as separate variates/components, each and every element of any component appearing as a list argument to `data.frame()`.
Data structures in R – Data frames

The same thing can happen when the component/variates/column is a matrix or array.

```r
data_zAsList <- data.frame(x = 1:3, y = 4:6, z = (matrix(1:12, nrow=3)))
str(data_zAsList)
```

```r
table-
"data.frame": 3 obs. of 6 variables:
## $ x : int 1 2 3
## $ y : int 4 5 6
## $ z.1: int 1 2 3
## $ z.2: int 4 5 6
## $ z.3: int 7 8 9
## $ z.4: int 10 11 12
```

Again, the data frame has 6 instead of 3 variates.
Data structures in R – Data frames

This can get you into trouble if the each element of the list has different lengths.

```r
data_zAsList <- data.frame(x = 1:3, y = 4:6, z=list(1:3, 4:5, 6:9))
```

Error in (function (... , row.names = NULL, 
  check.rows = FALSE, 
  check.names = TRUE,  :
  arguments imply differing number of rows: 3, 2, 4

There are two solutions to this problem.

1. Use the “inhibit” function I() to stop data.frame() from processing the z list
2. Attach the z list to the data.frame() after it has been constructed.
Data structures in R – Data frames

1. The “inhibit” function \( I() \) prepends the “AsIs” class to the object’s class. This stops its argument from being evaluated as its original class.

```r
data_zAsList <- data.frame(x = 1:3, y = 4:6, z = I(list(1:3, 4:5, 6:9)))
str(data_zAsList)
```

```r
# 'data.frame': 3 obs. of 3 variables:
# $ x: int 1 2 3
# $ y: int 4 5 6
# $ z:List of 3
#  ..$: int 1 2 3
#  ..$: int 4 5
#  ..$: int 6 7 8 9
# ..- attr(*, "class")= chr "AsIs"
```
Data structures in R – Data frames

The effect when the component/variante/column is a matrix or array:

```r
data_zAsList <- data.frame(x = 1:3, y = 4:6, z = I(matrix(1:12, nrow=3)))
str(data_zAsList)
```

```r
## 'data.frame': 3 obs. of 3 variables:
## $ x: int 1 2 3
## $ y: int 4 5 6
## $ z: 'AsIs' int [1:3, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
```
Data structures in R – Data frames

Or,

2. first create the data frame, and then add each list as a new component:

```r
data_zAsList <- data.frame(x = 1:3, y = 4:6)
data_zAsList$z <- list(1:3, 4:5, 6:9)
str(data_zAsList)
```

```r
## 'data.frame': 3 obs. of 3 variables:
## $ x: int 1 2 3
## $ y: int 4 5 6
## $ z:List of 3
## ..$ : int 1 2 3
## ..$ : int 4 5
## ..$ : int 6 7 8 9
```
Data structures in R – Data frames

This is more likely to arise when the rows/observations have more structure. For example, suppose each observation is a fitted model:

```r
fit2cars <- data.frame(poly_degree = 1:3)
rownames(fit2cars) <- c("linear", "quadratic", "cubic")
fit2cars$fit_info <- lapply(fit2cars$poly_degree,
    FUN = function(p){
        fit <- lm(dist ~ poly(speed, p), data = cars)
        results <- summary(fit)$coefficients
        df <- data.frame(coefficients = results[,"Estimate"],
            p_values = round(results[,"Pr(>|t|)"],5)
        )
        df}
    )

str(fit2cars)
```

```
## 'data.frame': 3 obs. of 2 variables:
## $ poly_degree: int 1 2 3
## $ fit_info :List of 3
## ..$ : 'data.frame': 2 obs. of 2 variables:
## ...$ coefficients: num 43 146
## ...$ p_values : num 0 0
## ..$ : 'data.frame': 3 obs. of 2 variables:
## ...$ coefficients: num 43 146 23
## ...$ p_values : num 0 0 0.136
## ..$ : 'data.frame': 4 obs. of 2 variables:
## ...$ coefficients: num 43 145.6 23 13.8
## ...$ p_values : num 0 0 0.137 0.369
```
Data structures in R – Data frames

Then the value of the fit_info each observation is a fitted model

```
fit2cars

## poly_degree
## linear       1
## quadratic    2
## cubic        3

## fit_info
## linear       42.9800, 145.5523, 0.0000, 0.0000
## quadratic    42.98000, 145.55226, 22.99576, 0.00000, 0.00000, 0.13640
## cubic        42.98000, 145.55226, 22.99576, 13.79688, 0.00000, 0.00000, 0.13727, 0.36892

fit2cars$fit_info

## [[1]]
## coefficients p_values
## (Intercept) 42.9800 0
## poly(speed, p) 145.5523 0

## [[2]]
## coefficients p_values
## (Intercept) 42.98000 0.0000
## poly(speed, p)1 145.55226 0.0000
## poly(speed, p)2 22.99576 0.1364

## [[3]]
## coefficients p_values
## (Intercept) 42.98000 0.00000
## poly(speed, p)1 145.55226 0.00000
## poly(speed, p)2 22.99576 0.13727
## poly(speed, p)3 13.79688 0.36892
```