Chapter 2
Data Structures

Abstract Programming with R objects and data structures.

Keywords Objects · Data · Indexing · Data manipulation · Missing values

2.1 Data Structures

Data structures store your data, and functions process it. You might think of functions as actions and data structures as the objects acted upon. You might think of functions as operators on data structures. A function's input arguments, and the object it returns, are data structures.

A data structure is the programmer's interface to data organised in computer memory. R provides several kinds of data structure each designed to optimise some aspect of storage, access, or processing. The five main data structures are called: vectors, matrices, arrays, data frames, and lists.

2.1.1 Vectors, Matrices, and Arrays

Vectors, matrices, and arrays are all based on a contiguous sequence of cells. They are designed to enable fast access to a particular layout of data. A vector[1] is an ordered row of cells. A matrix is a rectangular two-dimensional layout of cells like a grid. An array is a layout of cells that allows more than two dimensions.

[1] A vector is the simplest data structure in R. Scalar values are treated as single-cell vectors. A vector can be thought of either as a row or column since under matrix multiplication in R it is interpreted in whichever way makes it conformable with the other argument.

M. Allerhand, *A Tiny Handbook of R*, SpringerBriefs in Statistics,
DOI: 10.1007/978-3-642-17980-8_2, © Springer-Verlag Berlin Heidelberg 2011

Each cell contains an item of data. The three main types of data[2] in R are called: "numeric", "character", and "logical".

Numeric data includes integers and decimal numbers.[3] Character data consists of strings[4] of keyboard characters. Logical data consists of "truth values"[5] denoted `TRUE` and `FALSE`.

The cells of a vector, matrix, or array must contain the same type of data. For example a numeric vector must contain numeric data, a character vector must contain character data, and a logical vector must contain logical data. If you try to combine data of different types within the same vector, matrix, or array it will automatically be "coerced"[6] to one data type.

Numeric vectors are generally used to store continuous data, often as the columns of a data frame. Character vectors are generally used to store names and labels. Factors (see below) are generally used to store categorical data and grouping indicators. Logical vectors are generally used during programming temporarily to store the results of applying some condition to data, which can then be used programmatically, (see the section on Indexing below).

2.1.2 Data Frames and Lists

Data frames and lists are collections of data structures linked together. They are designed as general-purpose containers for data.

---

[2] R also provides a "complex" type for complex numbers, and a "raw" type for bits. Cells with missing values may contain the special value "Not Available" (`NA`).

[3] Sometimes decimal numbers may be printed in exponential form, such as: `1.5e-08`. This notation is used to print very small or very large numbers. The number `1.5e-08` is "1.5 times 10 to the −8", in other words 1.5/100000000.

[4] A string is a sequence of one or more keyboard characters, including spaces, and control characters such as \n (newline) and \t (tab), enclosed within quote marks, (double- or single-quotes). See `help(backquote)` for the list of control characters and further information about quote marks. Each cell of a character vector contains a string. For example: `"apple", "apple", "orange"`.

[5] Logical values are types of data returned by conditional expressions such as $3 > 2$, (`TRUE` as 3 is greater than 2), and `"apple" > "orange"`, (`FALSE` as `"a"` is alphabetically less than `"o"`). `TRUE` and `FALSE` can be abbreviated to `T` and `F`. Note: don't make variables with these names as they will mask the abbreviations.

[6] The rules for type coercion are as follows: `logical => numeric => character`. For example mixtures of `numeric` and `character` data are forced to character data, in which case all numbers become quoted strings, such as: `"3.14"`. The one exception to this is the special value `NA` (Not Available) used to signify a missing value. Functions with names that begin `"is."` are provided to test the type, and functions with names that begin `"as."` can be used to coerce the type. See `apropos("^is\\.")` and `apropos("^as\\.")`.
`as.numeric`: Number strings => numbers. Non-number strings =>`NA`. `TRUE=>1`, `FALSE=>0`.
`as.character`: Numbers => number strings. `TRUE=>"TRUE"`, `FALSE=>"FALSE"`.
`as.logical`: `0=>FALSE`, all non-zero numbers =>`TRUE`. Character strings =>`NA`.

A data frame is a rectangular layout of cells organised by columns. It is a collection of columns, all the same length, but which may be different types of vector. A data frame is most often used to store columns of raw data, some of which may be numbers, and some character data. Typically, but not necessarily, the columns are variables (and the column names are the names of the variables), and the rows are cases or observations. There is no particular requirement for one row per subject or case. Some columns may hold scores or names, others may be grouping factors used to indicate subjects, blocks, conditions or treatments, waves of repeated measures, and so forth.

A list is a collection of objects. The components of a list can be different types of data structure and can be of different lengths. A list is often used to pass a structured argument to a function and to return a multi-valued object from a function.

2.1.3 Creating Data

Data structures are created by reading data from some external source such as a file, database, or website. You can also create data within R for simulation and to create patterned vectors for use as a programming tool. This idea of "programming with data" occurs throughout R. For example a vector passed as an argument to a graph plotting function may be used to control the colours or shapes of individual points on a graph.

Creating Vectors

The simplest ways to create a vector are by combining, sequencing, or repeating vectors. The combine function `c` takes multiple vectors as arguments and combines them into one vector:

```
> x1 = c(2, 6, -1, 3.14, 0)          # Combine 5 single-cell numeric vectors
> x2 = c("apple", "apple", "orange") # Combine 3 single-cell character vectors
> x3 = c(x1, x2)                     # Combine 2 vectors
```

The sequence operator `:` makes a numeric vector that is a sequence in integer[7] steps between its arguments. For example:

```
> x4 = 1:5
> x5 = 5:-5
```

[7] See the `seq` function for making a sequence in fractional steps.

The repeat function `rep` makes a vector by repeating its argument, optionally as a whole or by repeating each cell. For example:

```
> rep(x4, times=3)   # Repeat as a whole
> rep(x4, each=3)    # Repeat each cell
```

Creating Matrices and Arrays

The simplest way to turn a vector into a matrix or an array is to use the functions `matrix` and `array`. For example:

```
> x1 = 1:16                                          # Vector
> x2 = matrix(x1, nrow=4, ncol=4)                    # 4×4 matrix
> x3 = matrix(x1, nrow=4, ncol=4, byrow=TRUE)        # 4×4 matrix by rows
> x4 = array(x1, dim=c(4, 2, 2))                     # 4×2×2 array
> x5 = array(x1, dim=c(2, 2, 2, 2))                  # 2×2×2×2 array
```

Alternatively bind vectors together as the rows or columns of a matrix using functions `rbind` and `cbind`:

```
> x1 = 1:3
> x2 = c("apple", "apple", "orange")
> x3 = rbind(x1, x2)
> x4 = cbind(x1, x2)
```

Matrices and arrays are vectors with an attribute[8] named `dim` that describes the dimensionality of the layout of cells. The `dim` is a numeric vector that stores the number of number of rows and columns of a matrix, or the dimensions of each "margin" of an array. There is also a function named `dim` that is used to get or set this attribute:

```
> dim(x3)   # Dimensions (number of rows and number of columns)
> dim(x4)
```

[8] An "attribute" is a named piece of additional information attached to a data structure. Some attributes are part of the language in the sense that R functions understand them, notably `names` (named elements or columns) and `dim` (dimensions of matrices and data frames). Functions called `names` and `dim` are provided to get and set these attributes. However an attribute can also be any information you like, such as a comment or note attached to a data structure. Function `attr` is provided to get or set any individual attribute, and function `attributes` to get or set an object's list of attributes. Function `as.numeric` has the side effect of removing attributes. Function `c` also removes attributes except for `names`.

Creating Data Frames and Lists

The simplest way to create a data frame (besides reading data from a file) is to use the `data.frame` function to add column vectors to the data frame. The simplest way to create a list is to use the `list` function to add objects to the list:

```
> x5 = data.frame(x1, x2)
> x6 = list(x1, x2, x4, x5)
```

R has a simple GUI editor for data frames:

```
> fix(x5)                  # Edit a data frame
> x7 = edit(data.frame())  # Create a new data frame
```

R provides many data sets[9] as data frames.

A data frame can have column and row names. These are stored in attributes called `names` and `row.names` respectively.[10] For example the provided dataset named `swiss`:

```
> swiss              # The data frame
> help(swiss)        # Its help page
> dim(swiss)         # The dimensions (rows, columns)
> nrow(swiss)        # Number of rows
> ncol(swiss)        # Number of columns
> names(swiss)       # The column names
> rownames(swiss)    # The row names
> summary(swiss)     # A summary of each column
```

Creating Character Data

Some useful character vectors are provided for convenience[11]:

```
> letters   # Lower-case alphabet
> LETTERS   # Upper-case alphabet
```

[9] Most R packages provide some data sets as well as functions. Use function `data()` to see the data sets that are loaded by default. Data sets have help pages. For example the page describing the structure and variables of the data set named `swiss` is displayed by `help(swiss)`.

[10] The row names are just row numbers by default, but the attribute may be assigned a character vector of names using function `rownames`. Use `names` or `colnames` to get or set the columns names.

[11] See also: `month.name` and `month.abb` for the full and abbreviated names of the months; `date`, `Sys.Date`, and `Sys.time` for date and time strings.

Creating and formatting numeric strings:

```
> x = c(3.1, 0.05, 99)
> as.character(x)
> format(x)
> format(x, width=8)
> format(x, width=8, nsmall=3)  # Aligning the decimal point
```

Formatting character vectors:

```
> x = c("Apple", "Orange")
> toupper(x)                              # Convert to upper-case
> tolower(x)                              # Convert to lower-case
> format(x)                               # Left justify (default)
> format(x, width=8)                      # Field width
> format(x, width=8, justify="right")     # Right justify
```

Pasting strings together[12]:

```
> paste("x", 3.14, sep="=")               # Paste strings with a given separator
> paste(c("x", "y"), c(3.14, 99), sep="=") # Paste vectors element-wise
> paste("Item", 1:20, sep="")             # No separator
```

Searching and replacing strings[13]:

```
> grep("^J", month.name, value=TRUE)  # Match strings beginning with "J"
> sub("^J", "j", month.name)          # Replace "J" at start of string with "j"
```

Printing strings:

```
> x = c(3.14, 0, NA, 99)
> cat("Some numbers:", x, "\n")  # Concatenate and print
> print(x)                      # The default print method
> print(x, na.print=" ")        # Print missing values (NA) as blanks
```

2.1.4 Sampling Data

The `sample` function draws a random sample[14] of the cells of a vector, and returns the sample as a vector. The size of the sample is specified by the `size` argument.

---

[12] See function: `strsplit` for splitting strings into vectors.

[13] String pattern matching uses regular expressions. See: `help(regex)`. See function `substr` for extracting and replacing characters at a given position within the string.

[14] A different sample is drawn each time a sampling function is called because the seed of the internal random number generator is updated automatically. If you want to draw the same sample you can set the seed. See `help(set.seed)`.

Without this the default sample is the same size as the argument, so the default behaviour is to make a vector by permuting or shuffling the argument. For example:

```
> sample(1:10)
> sample(c("a", "b", "c", "d", "e"))
> sample(1:100, size=50)              # Random sample (n=50) from 1:100
> sample(100, size=50)                # ... the same thing (allowed for convenience)
> sample(0:1, size=50, replace=T)     # Sampling with replacement
```

R provides functions to draw a random sample from several distribution families. For example the `rnorm` function draws a random sample from a normal[15] distribution. Its first argument specifies the sample size, and there are optional arguments `mean` and `sd` to specify the mean and standard deviation of the distribution to be sampled. The default parameters are `mean=0` and `sd=1`. For example:

```
> rnorm(50)                           # Random sample (n=50) from N(0, 1)
> rnorm(50, mean=10, sd=2)            # Random sample (n=50) from N(10, 4)
```

A careful reading of `help(rnorm)` suggests that `mean` and `sd` take vector values. When the meaning of this is not immediately clear it is often best simply to try it and see. For example:

```
> rnorm(4, mean=c(1, 100))    # Sampling from N(1, 1) and N(100, 1)
```

The vector passed to the `mean` argument is "recycled" (repeated) until its length matches the requested sample size, and then used to specify the mean of the distribution from which each observation is drawn. This idea of vector-valued arguments controlling individual observations is used in several places in R. (For example is to control the appearance of individual plotted points in graphics). A single valued argument is recycled so each observation in the sample is drawn from a distribution with the same mean. A multiple valued argument enables sampling from a normal mixture.

2.1.5 Reading Data

R provides several functions[16] for reading data from an external source such as a file, database, or website, and returning it as a single object. For example the `read.table` function is a general-purpose tool for reading tables of data from a plain text file,[17] and returning the data in a data frame.

---

[15] Equivalent functions for other distribution families include: `rt`, `rf`, `rbinom`, `rpois`, `rchisq`, `rexp`, `rweibull`, `rgeom`, `rhyper`, `rlogis`, `rbeta`, `rgamma`.

[16] See: `read.table`, `readLines`, `scan`, and related functions.

[17] Functions are provided in the `foreign` library to read data in several proprietary formats, including SAS, Stata, and SPSS. It is also possible to read data from certain databases such as MySQL. See the `R Data Import/Export` link from the HTML documentation displayed by `help.start()`.

These functions have a mandatory `file` argument to specify the data source. This may be a filename that is expected to be in the current working folder,[18] or may be a URL for data downloaded from a web site. Under Windows the `file` argument can be a call to the function `file.choose` which pops-up a Windows file locator, or may the special value `"clipboard"` to paste data from the Windows clipboard.[19] For example:

```
> read.table("clipboard")                          # Paste from the clipboard
> read.table("http://www.some.web.site/myfile.txt") # Download over http
> read.table(file.choose())                        # File locator
> read.table("myfile.txt")                         # Read a file in the working folder
```

The `read.table` function has several optional arguments to specify the file format. The most useful are: `header` to specify whether the first line is a column header or data, and `sep` to specify the column separator. The default values[20] of these arguments specify no header, and columns separated by blanks (one or more spaces or tabs). If for example the text file contains column headings on the first line and has comma-separated columns the options should be as follows:

```
> read.table("myfile.txt", header=TRUE, sep=", ")
```

Blank lines and comment lines (with the comment sign `#` at the left) in the file are ignored by `read.table`. Columns of numbers become numeric vectors[21] in the data frame. Columns of character data are by default converted to factors[22] in the data frame.

---

[18] If R displays a message that there is `"No such file or directory"` the most likely explanation is that your working directory is not pointing to the folder that contains the file. Set your working directory, or provide an absolute pathname to the file. Pathnames should either use forward slashes as in `"C:/path/to/file"`, or double backslashes as in `"C:\\path\\to\\file"`.

[19] See: `help(connections)` and `help(clipboard)`.

[20] Several variants of `read.table` are provided with slightly different defaults. See `help(read.table)`.

[21] If a column of numbers is intended to be a grouping indicator then it is necessary explicitly to convert the column to a factor, for example using function `as.factor`.

[22] Set argument `stringsAsFactors=FALSE` to override the default behaviour and keep columns of character data as character vectors. Use command `options(stringsAsFactors=FALSE)` if you wish to set this default behaviour globally. If a column of numbers contains characters such as `'.'` to signify a missing value, then the whole column is taken to be characters and converted to a factor. Set argument `na.strings` to `'.'` to override this and recode all `'.'` as `NA`, (see the section on Missing Values).

2.2 Operations on Vectors and Matrices

2.2.1 Arithmetic Functions

Arithmetic functions take a data structure argument and return the data structure after applying an operation element-wise to each cell. For example:

```
> x = rnorm(10, mean=5, sd=2)   # A numeric vector
> round(x, 2)                   # Round to 2 decimal places
> sqrt(abs(x))                  # Square root. Absolute value to avoid sqrt of negative
> log(x + 1)                    # Log (base e). Add 1 to avoid log(0)
> scale(x)                      # Standardize as z-scores
```

The arithmetic functions have no effect on the input data structure `x`. If you want to transform `x` you must explicitly assign the returned value back to `x`. For example:

```
> x = scale(x)   # Standardize x
```

Some arithmetic functions	
round	Round to given number of decimal places
trunc	Truncate down to nearest whole number
ceiling, floor	Round values in a vector up (ceiling) or down (floor)
zapsmall	Replace values in a vector or matrix that are close to zero with 0
abs	Absolute (unsigned) value
sqrt	Square root
exp	Exponential
log, log10, log2	Log to base e, 10, and 2
sin, cos, tan	Trigonometric functions
asin, acos, atan	Inverse (arc) trigonometric functions
scale	Centering and scaling

2.2.2 Descriptive Functions

Descriptive functions take a data structure argument and return a summary. For example a single-valued summary of a vector:

```
> x = rnorm(10, mean=5, sd=2)   # A numeric vector
> length(x)                     # Length (number of elements in x)
> mean(x)                       # Mean of the elements in x
> sd(x)                         # Standard deviation of the elements in x
```

Some summary descriptive functions	
length	Number of elements in a vector
sum	Sum of the values in a vector
min, max, range	Minimum, maximum, and range (min, max) of a vector
mean, median	Mean and median of the values in a vector
sd, var	Standard deviation and variance
cov, cor	Covariance and Pearson correlation

Functions `cov` and `cor` return either a single value or a matrix, depending upon the arguments.[23] If the arguments are two vectors a single value is returned. If the argument is a matrix a matrix is returned. The `i,j`'th element of the matrix is the covariance (using `cov`) or correlation (using `cor`) between the `i`'th and `j`'th column vectors.

```
> x = rnorm(100)
> y = rnorm(100)
> var(x)
> var(y)              # Variance
> cov(x, y)           # Scalar covariance
> cor(x, y)           # Pearson correlation coefficient
> cov(cbind(x, y))    # Covariance matrix
> cor(cbind(x, y))    # Correlation matrix
```

2.2.3 Operators and Expressions

R has conventional arithmetic expression syntax with the usual arithmetic and conditional operators.[24]

Arithmetic and conditional operators			
`x+y`	add	`x == y`	x equal to y?
`x−y`	subtract	`x != y`	x not equal to y?
`x*y`	multiply	`x < y`	x less than y?
`x/y`	divide	`x <= y`	x less than or equal to y?

(continued)

[23] Functions are designed where possible to allow different kinds of input data, and to implement the generic meaning of the function in whatever way makes best sense for the kind of data they are given. For example the description of the arguments in `help(cor)` suggests they can be a numeric vector, matrix or data frame.

[24] See: `help(Arithmetic)`, `help(Comparison)`, and `help(Syntax)`.

(continued)

Arithmetic and conditional operators			
x^y	raise to power	x > y	x greater than y?
x%%y	remainder	x >= y	x greater than or equal to y?
−x	negate		

Vector Arithmetic

The operators are "vectorized" as follows. Unary operators apply element-wise to each cell:

```
> x = c(0, 2, 4, 6, 8, 10)
> -x   # Negate each element
> x^2   # Square each element
```

Binary operators apply element-wise to corresponding pairs of cells:

```
> x = c(0, 2, 4, 6, 8, 10)
> y = c(1, 5, 3, 7, 11, 8)
> x + y    # Add corresponding elements
> x<y    # Test corresponding elements
```

If the vectors are different lengths the shorter vector is "recycled" to match the length of the longer, by concatenating it end-on-end with itself until it is at least as long, trimming excess off the end if necessary.[25] The operation is then applied pair-wise to corresponding cells. The result vector is the same length as the longer argument.

```
> x = c(0, 2, 4, 6, 8, 10)
> y = c(4, 8)
> x + y
> x<y
> x * 4   # Scalar 4 is recycled to match length(x)
> x<4
```

Matrix Arithmetic

Unary operations and arithmetic functions are applied element-wise. Binary operations are applied between corresponding pairs of elements. Matrices must have the same dimensions to be "conformable" for arithmetic, but if one argument

[25] A warning message about fractional recycling is displayed if the length is not an exact multiple. This can safely be ignored, or turned off by the command: `options(warn = -1)`.

is a vector it is recycled to match the length of the matrix. For conventional matrix multiplication[26] use the special operator: %*%. For example:

```
> x = matrix(1:16, nrow=4, ncol=4)
> x^2          # Square each element
> sqrt(x)      # sqrt each element
> x + 0.1      # 0.1 is recycled
> x * x        # Multiply corresponding elements
> x %*% x      # Conventional matrix multiplication
```

Some matrix functions	
t	Transpose
diag	Diagonal
%*%	Inner (dot) product of two vectors $x^t y$, and conventional matrix multiplication
%o%	Outer product of two vectors xy^t
crossprod, tcrossprod	Cross products $x^t y$ and xy^t of matrices
det	Determinant
solve	Inverse
eigen	Eigenvalues and eigenvectors
svd	Singular value decomposition
qr	QR decomposition
chol	Choleski decomposition

Conditional Expressions

Conditional expressions result in logical vectors. The main purpose of these is to represent the vectorized result of a conditional expression so that it can subsequently be applied in conditional indexing.

Conditional operators compare values and return truth values: TRUE or FALSE. For example if x=1 then the conditional expression x > 0 returns the value TRUE.

Conditional operations are defined for numeric, logical, and character vectors. The conditional operators == and != are defined for factors.

Under conditional operations both numerical and logical values are compared numerically.[27] Logical values are treated numerically in this context as: TRUE=>1,

[26] Matrix multiplication x %*% y is conformable if ncol(x) == nrow(y). If one argument is a vector it is interpreted as a row or column to suit so a transpose is unnecessary.

[27] Annoyance: when testing for negative numbers an expression like x<-1 will unexpectedly modify x because the <- is interpreted as the assignment operator. The workaround is to include space: x < −1.

FALSE=>0. Character values are compared alphabetically and case-sensitively: the alphabet is in increasing order, for example "ant" < "bee", and upper-case is greater than lower-case so "Ant" > "ant".

Conditional operators are vectorized as follows. When two vectors are compared the shorter is recycled if necessary to make the vectors the same length, then the conditional operation is applied between pairs of corresponding cells. The result is a logical vector the same length as the longer argument. For example:

```
> x = c(2, 4, 6, 8, 10)
> x >= 6      # 6 is recycled to match the length of x
> y = c("apple", "apple", "orange", "apple", "orange")
> y == "apple"
```

Logical vectors can be combined using the operators[28]: & (AND), | (OR), and ! (NOT) to form composite conditions. For example:

```
> x >= 6 & y == "apple"  # TRUE where x >= 6 AND y == "apple"
```

Arithmetic With Logical Vectors

Arithmetic operations are conventional for numeric data. They are not defined for character data. They are defined for logical data by treating logical values as numeric in an arithmetic context as: TRUE=>1 and FALSE=>0.

Consequently the sum function counts TRUE in a logical vector, and the mean function calculates the proportion TRUE. This enables descriptive functions to summarise how a vector meets a given condition. For example

```
> x = c(2, 4, 6, 8, 10)
> y = c("apple", "apple", "orange", "apple", "orange")
> sum(x)                          # Sum of the elements of x
> sum(y)                          # Error! (arithmetic not defined for character data)
> sum(x < 6)                      # How many elements of x are less than 6 (x<6 TRUE)?
> sum(y == "apple")               # How many elements of y are "apple"?
> sum(x = 6 & y == "apple")       # How many x>=6 AND y=="apple"?
> mean(x < 6)                     # What proportion of x is less than 6?
> mean(y == "apple")              # What proportion of y is "apple"?
> mean(x >= 6 & y == "apple")     # What proportion x>=6 AND y=="apple"?
```

[28] & and | are vectorized for combining logical index vectors. Corresponding operators && and || result in single truth values, usually for purposes of flow control in a program. See help(Logic, package="base"), and the examples therein which show how to construct truth tables defining the logical operations.

2.3 Factors

Factors represent categorical variables and are used as grouping indicators. The categories are stored internally as numeric codes, with labels to provide meaningful names for each code.

For example a sequence of categorical observations: `"apple"`, `"apple"`,`"orange"`,`"apple"`,`"orange"` is efficiently stored as numeric codes `1,1,2,1,2`. The values of the codes are always restricted to `1,2,...,k`, to represent `k` discrete categories. The labels, here `"apple"`,`"orange"`, are a character vector stored with the factor as an attribute named `levels`. Whenever the factor is used, such as when its value it printed, the labels are mapped onto the codes internally. The order of the labels is important: the first label is mapped to each code `1`, the second to each code `2`, (and so on if there are more levels). Here `"apple"` is mapped to each code `1`, and `"orange"` to each code `2`.

Use function `as.factor` to create the factor from a vector, and functions `as.numeric` and `levels` to get the factor's internal numeric codes[29] and labels:

```
> x = as.factor(c("apple", "apple", "orange", "apple", "orange"))
> as.numeric(x)   # The internal numeric codes
> levels(x)       # The labels (a character vector)
> x               # Print the value of the factor
```

The factor can be used to indicate which group[30] observations belong to:

```
  score condition
1    39     apple
2    14     apple
3     2    orange
4     7     apple
5    44    orange
```

[29] To coerce a factor with numeric labels as a numeric vector first coerce as character using `as.character` and then as numeric using `as.numeric`.

[30] There are programming advantages to abstracting the grouping information in this way: it provides a device for manipulating groupings that does not depend upon a particular shape or layout for data, that can accommodate missing observations, and that enables programming easily to express conditions on the groupings, such as to extract and summarise the observations in a particular group.

2.3.1 Making Factors

The `read.table` function that reads an external text file into a data frame converts columns of non-numeric characters to factors by default. The factor levels depend upon the number of different character strings found in the data.

Functions `as.factor` and `as.ordered` make factors from vectors. Ordered factors[31] differ from factors only in their class. When a vector is converted to a factor the factor labels are the unique values found in the vector by default. Their default order is their natural sort order: alphabetical for character vectors, numerical for numeric vectors.

```
> x = c("orange", "orange", "orange", "orange", "apple")
> as.factor(x)  # Levels in alphabetical order
> x = rep(3:1, each=2)
> as.factor(x)  # Levels in numerical order
```

Function `gl` (generate levels) is used to make grouping factors for balanced experimental designs. Use function `rep` to add replications to the grouping factor if necessary.

```
> gl(3, 10, labels=c("low", "med", "high"))  # Grouping factor
```

Function `cut` makes a factor from a numeric vector by dividing the numeric range into intervals to group the values. The interval boundaries are specified using an argument named `breaks`. By default the intervals are defined as "open on the left", (or equivalently "closed on the right"), and are indicated by labels like `(...]`. This means values that fall on a break between intervals will be grouped to the left of the break. Any value on the left break of the left-most interval is classified as `NA`, (since there is no information about the interval to the left of that). Set argument `include.lowest=TRUE` to override this behaviour and group such values within the lowest interval. Set argument `right=FALSE` to re-define the intervals so that values on a break are grouped to the right of the break. (In that case `include.lowest=TRUE` groups right-most values within the highest interval). For example:

[31] An ordered factor does not refer to the order of the level labels, but simply marks the factor as "ordered" so it can be handled appropriately by functions where the distinction between nominal and ordinal data is relevant. For example the contrast coding used in linear modelling functions has different defaults for unordered and ordered factors. Unordered factors get comparisons between group means. Ordered factors get trend analysis.

```
> x = c(20, 21, 30, 39, 40)
> cut(x, breaks=c(20, 30, 40))                    # Open on the left
> cut(x, breaks=c(20, 30, 40), inc=TRUE)
> cut(x, breaks=c(20, 30, 40), right=FALSE)   # Open on the right
> cut(x, breaks=c(20, 30, 40), right=FALSE, inc=TRUE)
```

The breaks can be specified as a single number (greater than 1) giving the number of intervals. The `quantile` function is useful for calculating breaks so that the intervals will contain equal frequencies of values. For example:

```
> x = round(rexp(1000)*100)
> hist(x)
> g1 = cut(x, breaks=4, inc=TRUE)   # Breaks at equal intervals
> g2 = cut(x, breaks=quantile(x), inc=TRUE)   # Breaks at quantiles
> table(g1)
> table(g2)   # Quantile intervals try to contain equal frequencies
```

2.3.2 Operations on Factors

Some functions for working with factors	
as.factor, factor	Make a factor from a vector
as.ordered, ordered	Make an ordered factor from a vector
gl	Make a factor by generating (equal sized) levels
cut	Make a factor by cutting a vector at given break points
relevel	Set the first (reference) level of a factor
as.numeric	Get a factor's numeric codes
levels	Get the labels of a factor's levels
nlevels	Get the number of levels of a factor
:, interaction	Cross-classify factors

Arithmetic operators are not valid for factors. The conditional operators `==` and `!=` are valid for factors and typically are used for conditional indexing in a vector or data frame. The `:` operator between two factors of the same length returns a factor by cross-classifying its arguments. The result is a factor with levels made from all combinations of the levels of the arguments.

```
> g1 = gl(2, 6, labels=c("+", "-"))
> g2 = gl(3, 4, labels=c("A", "B", "C"))
> g1:g2                 # Cross-classify the factors
> nlevels(g1:g2)        # Number of levels in the crossing (2×3)
```

2.3.3 Re-ordering and Re-labelling

Functions `factor` and `ordered` make factors from vectors, in the same way as `as.factor` and `as.ordered`. They have additional arguments to control the labels which, given a factor instead of a vector as input, can be used to re-order and re-label the factor's levels.

The `levels` argument specifies an order[32] for the labels in terms of the current labels. When the labels are re-ordered the corresponding numeric codes are changed as well to preserve the mapping between labels and codes. The `labels` argument specifies new values for the labels, which are re-labelled in the order given. When both `levels` and `labels` are given the new labels re-label the old in their order specified by `levels`.

```
> x = as.factor(c("orange", "orange", "apple", "orange", "apple", "pear"))
> factor(x, levels=c("orange", "pear", "apple"))   # Re-order
> factor(x, labels=c("A", "O", "P"))               # Re-label
> factor(x, levels=c("orange", "pear", "apple"), labels=c("O", "P", "A"))
```

2.4 Indexing

There are functions that return the first and last few parts of a data structure, (`head` and `tail`), and there is a rudimentary editor GUI for data frames (`fix` and `edit`). But indexing is a more powerful approach to data manipulation in general.

Objects have distinct parts: the rows and columns of a data frame, the components of a list, the individual cells within a vector, matrix, array, or data frame. These parts are ordered and numbered, and may also have names associated with them. Indexing means accessing parts by name or by number to extract or replace values.[33] The index can be conditional and derived programmatically, (for example to extract or replace "scores for males aged over 50 and in employment").

---

[32] The order is relevant to the default appearance of tables and graphs, and also in modelling functions when a reference level is used for comparisons. See also function `relevel` to re-order levels so a given level is the first (reference) level.

[33] See: `help(Extract)`, `help("[.data.frame")`, and `help(subset)`.

2.4.1 Indexing by Name

Objects may have a `names` attribute to associate a label with each part.[34] For example the `names` of a data frame are its column names, most often representing the names of variables. The `names` of a list are the names of the objects collected in the list.

An object's names can be accessed using the `names` function, and then its parts can be accessed by name using a `$` syntax. For example, the provided data set named `sleep` consists of two variables in a data frame:

```
> names(sleep)          # Names of the variables in data frame 'sleep'
> sleep$extra           # Access variable 'extra' in data frame'sleep'
> mean(sleep$extra)     # Mean of 'extra' in data frame 'sleep'
```

Many functions return multi-valued results as a single object with named components. Individual parts of the results can be extracted by name. For example:

```
> x = t.test(extra~group, data=sleep, paired=TRUE)   # Paired t-test
> names(x)    # The names of the components returned by 't.test'
> x$p.value   # The p value from the t test
```

Attaching Data Frames

In situations where the `$` syntax is cumbersome, functions `attach` and `detach` can be used to enable access to the columns of a data frame as if they are variables outside the data frame.[35]

```
> attach(sleep)   # Attach a data frame
> mean(extra)     # Access variables by name
> detach(sleep)   # Detach the data frame
```

Functions `with` and `transform` provide temporary access to data frame columns by name. Function `with` enables evaluation with data frame variables. Function `transform` enables evaluation in the data frame and returns the transformed data frame. For example:

---

[34] See: `help(names)`, and also `help(colnames)`, `help(dimnames)`, and `help(row.names)`.

[35] Caveat: assignments to attached variables are not assignments to the data frame. Attaching a data frame works by inserting the object on the search path so the variables within the data frame can be found by name. See `help(search)`. However the global environment is always searched first, so variables in the data frame may be masked by variables in the global environment. Any assignment to variables of the same name assigns in the global environment and not in the data frame. So assignments to attached variables have no effect on the data frame.

```
> with(sleep, mean(extra))                  # Mean of 'extra' in data frame 'sleep'
> transform(sleep, ctime=scale(extra))      # Add a derived variable
```

2.4.2 Indexing by Number

The parts of an object, (the rows and columns of a data frame, the components of a list, the individual cells within a vector, matrix, array, or data frame), are ordered and numbered. The numbering always starts with 1, (not 0), and ends at the length or dimensions of the object.

The length of an object, (for example the number of cells in a vector, or the number of components in a list), is returned by function `length`:

```
> length(c("x", "y", "z"))   # Length of a vector
```

The dimensions of an object, (such as the number of rows and columns in a matrix or data frame), are returned by functions `dim`, `nrow` and `ncol`. For example, using the provided data set named `swiss`:

```
> dim(swiss)    # Dimensions (rows, columns) of data frame 'swiss'
> nrow(swiss)   # Number of rows in data frame 'swiss'
> ncol(swiss)   # Number of columns in data frame 'swiss'
```

The syntax for indexing by number uses square brackets. For example to index the cells of a vector by number:

```
> x = c("x", "y", "z")
> x[1]                    # Get the first cell (x[0] is undefined)
> x[length(x)]            # Get the last cell
> x[1] = "a"              # Set the first cell
```

A feature of R is that the indices are vectors, enabling simultaneous access to a slice or subset of cells. The result is the same length as the index vector, (which may be longer than the object being indexed). The index addresses particular cells by number and these are returned in the order given in the index. This enables repetition, re-ordering, and sorting.

```
> x[1:2]                      # Get the first two cells
> x[c(3, 1)]                  # Get cells 3 and 1 in that order
> x[c(rep(3, 4), rep(1, 3))]  # Get cell 3 (4 times) and 1 (3 times)
> x[1:2] = c("p", "q")        # Set the first two cells
```

Vectors have a single index, as in: `x[i]`. Matrix and data frame cells have pairs of indices, respectively for rows and columns, as in: `x[i,j]`. Arrays have tuples of indices depending on the dimensionality of the array, as in: `x[i,j,k]`

(for a 3-dimensional array). An empty index is shorthand for a complete index. For example:

```
> swiss[c(10, 15), 2:3]   # Index rows 10 and 15, and columns 2 to 3
> swiss[c(10, 15), ]      # Rows 10, 15 and all columns (empty column index)
> swiss[, 2:3]            # Columns 2:3 and all rows (empty row index)
```

If the object has named parts an index vector can be character or numeric:

```
> swiss[c(10, 15), c("Agriculture", "Examination")]   # Same as: swiss[c(10, 15), 2:3]
```

A single index to a data frame accesses columns by number or by name. Names can for example be derived using `grep` to find names matching a pattern, (useful when the data frame has hundreds of columns). The columns are returned as a data frame, or a single column can be returned in its native type using double brackets. For example:

```
> swiss[2:3]              # Columns 2:3, (same as swiss[, 2:3])
> swiss[3]                # Column 3 returned as a data frame
> swiss[[3]]              # Column 3 returned as a vector
> swiss["Examination"]    # Column 3 accessed by name
```

2.4.3 Inserting and Deleting Rows or Columns

Numeric indices in R can be positive or negative, (the elements of an index vector must either be all positive or all negative). A negative index accesses all the cells NOT indexed. This can be used to delete cells, rows, or columns. For example to delete rows or columns of a data frame (assign the results if you wish to save them):

```
> swiss[-c(3, 5, 7), ]    # Drop rows 3, 5, and 7
> swiss[-6]               # Drop column 6
```

A data frame can dynamically be increased in size with new rows or columns,[36] either by name (using $ syntax) or by index number. For example:

```
> swiss$zFertility = scale(swiss$Fertility)   # Append a derived variable
> sleep$zFertility = NULL                     # Drop a variable by name
> swiss[nrow(swiss)+1, ] = NA                 # Append a row (of NA values)
> swiss = swiss[-nrow(swiss), ]               # Delete a row
```

[36] Columns may only be appended. A new column is not allowed to leave "holes" after existing columns.

To insert a row or column use `rbind` and `cbind` to construct the data frame around the insert. For example:

```
> rbind(swiss[1:6,], NA, swiss[7:nrow(swiss),])   # Insert 7th row
> cbind(swiss[1:2], NA, swiss[3:ncol(swiss)])     # Insert 3rd column
```

2.4.4 Indexing with Factors

Factors can be indexed[37] by numeric or logical vectors, and can be assigned values provided these are amongst the factor's labels:

```
> g = as.factor(c("orange", "orange", "apple", "orange", "apple", "pear"))
> g[4]
> g[4] = "apple"
```

A factor can be used as an index vector, and then its numeric codes are the index.

```
> x = c("red", "orange", "green", "yellow")
> x[g]
```

The conditionals `==` and `!=` are valid for factors and the resulting logical vector is typically used to index a vector or data frame. For example:

```
> x = 1:6
> x[g=="apple"]   # Get values of x where factor g is "apple"
```

Indexing can be used to merge or drop factor levels

```
> factor(c("apple", "orange", "orange")[g])   # Merge levels
> g[g!="pear"]                                # Subset the factor
> g[g!="pear", drop=TRUE]                     # Drop a level
> factor(g, levels=c(levels(g), "lemon"))     # Add a level
```

2.4.5 Conditional Indexing

Conditioning by String Pattern Matching

The `grep` function is used to extract elements of character vectors by string pattern matching using regular expressions. By default it returns the numeric index of each match, or with argument `value=TRUE` it returns the matching elements

[37] See: `help("[.factor")`.

themselves. For example to extract a subset of variables from a data frame by matching name patterns:

```
> swiss[grep("^E", names(swiss))]  # Variables with names beginning "E..."
```

Indexing with Logical Vectors

Conditional indexing means indexing with a logical vector that has been derived from a conditional expression. For example:

```
> i = swiss$Agriculture > 50 & swiss$Education > 8
```

A conditional expression returns a logical vector. Logical index vectors are different from other kinds of index. Numeric and character index vectors address particular cells by number or by name, and these are returned in the order given in the index. Logical index vectors specify a pattern of cells to be addressed. If necessary the pattern is recycled so that the index vector matches the length of the object being indexed. The index is then applied element-wise and addresses cells corresponding to cells of the logical vector that are `TRUE`.

```
> swiss[i,]                      # Index rows where the condition is TRUE
> swiss[!i,]                     # Index rows where the condition is NOT TRUE
> mean(swiss[i, "Fertility"])    # Mean 'Fertility' within condition
```

Some functions for logical and set operations	
any	TRUE if any of the arguments contains TRUE
all	TRUE if all of the arguments are TRUE
which	Get numeric indices where a logical vector is TRUE
grep, agrep	Get indices or values that match regular expression patterns
union	Set union, (elements in vector x OR y or both)
intersect	Set intersection, (elements in vector x AND y)
x %in% y	Get vector containing TRUE for each element of x that is also in y
unique	Get vector of unique values by dropping duplicate values
duplicated	Get vector containing TRUE to indicate duplicated values
upper.tri	TRUE for the upper-triangle of a matrix
lower.tri	TRUE for the lower-triangle of a matrix

2.4.6 Sorting

There is a function `rev` that reverses a vector's order, and a function `sort` sorts a vector numerically or alphabetically into ascending or descending order. A more flexible approach is to sort by deriving and applying an appropriate index vector.

The benefit of this two-step approach is that it enables you sort one vector by another or, for example, to sort the rows of a data frame by one or more of the columns.

The function `order` is used to derive the index vector that would sort a vector. For example the first element of `order(x)` is the index of the lowest-valued element in `x`. The next is the index of the next lowest value, and so on:

```
> x = c(2, 4, 1, 3)
> order(x)
> x[order(x)]   # Sort x using an ordered index vector
```

An appropriately ordered index vector can be used to sort the rows or columns of a data frame:

```
> # Sort rows of 'swiss' by 'Examination'
> swiss[order(swiss$Examination), ]
> # Sort rows of 'swiss' by 'Examination' and within that by 'Education'
> swiss[order(swiss$Examination, swiss$Education), ]
> # Sort columns of 'swiss' by name
> swiss[, order(names(swiss))]
```

2.5 Reshaping

Reshaping data frames	
stack, unstack	Stack or unstack columns to reshape into long or wide format
reshape	Reshape a data frame into long or wide format
merge	Merge data frames by common column or row names

One purpose of reshaping a data frame is to convert between "long format" and "wide format". A "long format" data frame has one column for each variable. A "wide format" data frame has one row for each subject or case. These formats are different when there are repeated measures of subjects in time.

In long format a time-varying variable is a single column containing all the subjects measures of that variable at all time-points. The data frame must then also have factors to indicate which subject and which time-point each measure belongs to.

In wide format a time-varying variable is spread over several columns, one for each time-point. In that case it is clear which subject and time-point each measure

belongs to from its location in the data frame. However the data must be balanced in the sense that all subjects have a complete set of measures at all time-points, otherwise it is necessary to pad the data with missing values to preserve the rectangular shape of the data frame. One advantage of long format is that grouping factors can represent unbalanced data.

2.5.1 Stacking and Unstacking

The basic operation of reshaping is stacking and unstacking. The `stack` function takes a data frame and stacks its columns into a single column, returning this in a data frame as a column named `values`, with a factor named `ind` to indicate which of the original columns each value belongs to. For example, stack the four `iris` measures:

```
> stack(iris[1:4])
```

The `unstack` function takes a column and unstacks it into several columns by splitting it on a factor. It returns a data frame if the factor is balanced, or a list if not.[38] To check balance the `replications` function counts the number of replications, and it too returns a list if the groups are unbalanced. For both functions the name of the column and the grouping factor are specified using a formula. For example with the `sleep` data:

```
> replications(extra~group, sleep)   # Balanced data (10 reps per group)
> unstack(sleep, extra~group)        # Data frame (unstack 'extra' by 'group')
```

As an example of unbalanced data, the `ChickWeight` data contains repeated measures of 50 chickens `weight` at several time-points. Different chickens were measured different numbers of times:

```
> replications(weight~Chick, ChickWeight)   # Unbalanced data
> unstack(ChickWeight, weight~Chick)        # List (unstack 'weight' by 'Chick')
```

2.5.2 Reshaping: Wide and Long

The `reshape` function is a more general reshaping tool[39] designed for data that are repeated measures of subjects in time. It has arguments to specify the time-varying

[38] See also functions `split` and `unsplit` which split a vector or data frame into a list, or combine list components.

[39] See also functions `melt` and `cast` in package `reshape`.

variables in a data frame that are to be stacked or unstacked. It handles unbalanced data by inserting NA as appropriate, and also treats time-invariant variables appropriately.

For example to reshape the ChickWeight data from long to wide, use argument v.names to specify the time-varying variable to be unstacked, and arguments idvar and timevar to specify the subject and time-point factors:

```
> dat = reshape(ChickWeight, direction="wide", v.names="weight",
+               idvar="Chick", timevar="Time")
```

The result contains NA because the data are not balanced. The result also includes the Diet variable, a time-invariant variable that was simply repeated at each time-point in long format.

To reshape from wide to long, use argument varying to specify the columns to be stacked, and argument v.names to specify a name for the stacked column. For example if the four iris measures are taken as four time-points with measures of one time-varying variable:

```
> reshape(iris, direction="long", varying=1:4, v.names="value")
```

The reshape function allows more than one time-varying variable. For example if the four iris measures are taken as two time-varying variables, each with measures at two time-points iris[1:2] and iris[3:4]:

```
> reshape(iris, direction="long", varying=list(1:2, 3:4),
+         v.names=c("value1", "value2"))
```

2.5.3 Merging

The merge function performs a database "join" operation between two data frames based on columns that contain values common to both data frames. For example if two data frames both have a column named id containing some values common to both data frames, the merge function returns a data frame by joining rows with matching id:

```
> x = data.frame(id=1:10, var1=1:10)
> y = data.frame(id=3:7, var2=letters[3:7])
> merge(x, y, by="id")
```

The default behaviour is to return a data frame that contains just the rows that are common to both input data frames, that is rows where the input data frames have the same value in the by variable. This can be overridden using logical arguments all.x and all.y. For example if all.x=TRUE the returned data frame will have all the rows from x, (the first input data frame), whether they match in y

or not. Rows that don't match in y are then given NA values in the columns merged from y. For example:

```
> merge(x, y, by="id", all.x=TRUE)   # All rows from x
> merge(x, y, by="id", all.y=TRUE)   # All rows from y
```

2.6 Missing Values

The special value NA ("Not Available") is used to represent a missing value. The NA value can appear in a vector of any type without coercion.

2.6.1 Recoding Missing Values

Many R functions understand NA in the sense that they have built-in methods for handling NA values. Missing values should be coded as NA to take advantage of these.

Function read.table recodes blanks as NA by default. Other missing value codes can be passed to argument na.strings. For example to recode all . or −999 as NA:

```
> read.table("data.txt", na.strings=c(".", "-999"))
```

2.6.2 Operations with Missing Values

If any data are missing the general principle is that it is better by default to propagate a missing value code than an incorrect summary or result. A missing value can be caught and handled by subsequent processes. An incorrect result might slip through un-noticed. This principle applies logically to arithmetic and conditional operations. Any operation involving NA is undecidable and so the only sensible result is NA. Descriptive functions by default will propagate NA values rather than a potentially incorrect summary. For example:

```
> x = c(3.14, NA, 2.72, 1.96)
> y = c(NA, 2.72, 1.96, 3.14)
> mean(x)
> cor(x, y)
> cor(cbind(x, y))   # The default use="everything"
> sum(x==NA)         # NOT the way to count NA
```

2.6.3 Counting and Sorting Missing Values

A special function `is.na` is needed to test for `NA`. The function returns `TRUE` where its argument is `NA`.

```
> x = c(3.14, 0, NA, 1)
> is.na(x)              # Test for NA
> sum(is.na(x))         # Count NA
```

Function `complete.cases` is provided for testing for `NA` along data frame rows. Other functions provide arguments to accommodate `NA`. For example:

```
> dat = data.frame(A=c(1,0,NA,0,NA), B=c(NA,1,0,1,0))   # Data frame with some NA
> sum(complete.cases(dat))          # Count complete cases
> sum(!complete.cases(dat))         # Count incomplete cases
> summary(dat)                      # Summary counts NA by column
> table(dat, useNA="always")        # Cross-tabulate including NA
> dat[order(dat$A),]                # Sort by column A with NA last
> dat[order(dat$A, na.last=F),]     # Sort by column A with NA first
```

2.6.4 Handling Missing Values

Cases with missing values can be dropped altogether:

```
> dat = dat[complete.cases(dat),]   # Drop cases (rows) with any NA
```

It is possible to impute[40] values that are missing. Descriptive and modelling functions have default methods for handling `NA` values and generally provide arguments to override the default behaviour and specify how `NA` is handled.

Descriptive functions (such as `sum`, `mean`, `sd`, and so forth) propagate `NA` by default but provide an argument named `na.rm` ("NA remove") that can be set to `TRUE` to instruct the function to omit all `NA` from the calculation. Functions `cov` and `cor` have a `use` argument that can be set as `use="complete.obs"` for casewise deletion (aka listwise deletion), and `use="pairwise.complete.obs"` to use all complete pairs of observations in the calculation.

[40] See package `mvnmle` for maximum likelihood imputation, and various packages for multiple imputation such as `mitools` and `mice` listed at http://cran.r-project.org/web/views/Multivariate.html.

```
> x = c(3.14, NA, 2.72, 1.96)
> y = c(NA, 2.72, 1.96, 3.14)
> mean(x, na.rm=TRUE)
> cor(x,y, use="pairwise.complete.obs")
> cor(cbind(x,y), use="pairwise.complete.obs")
```

Modelling functions (such as `lm`, `aov`, `glm`, and so forth), and other functions for multivariate analysis (such as `princomp`, `prcomp`, and `factanal`), have an argument named `na.action` which is set to the name of a function to handle incomplete data. The default setting is the function named `na.omit` which handles missing values by casewise deletion. If instead you wish to propagate `NA` values to fitted values, residuals, predicted values, or factor scores, preserving the length of these data,[41] then it is necessary to set `na.action=na.exclude`.

```
> fitted(lm(y~x))                          # NA omitted from result
> fitted(lm(y~x, na.action=na.exclude))    # NA preserved in result
```

2.7 Mapping Functions

R provides the usual facilities for programming loops for iterations,[42] but in practice you seldom need to use them. Use mapping functions instead because they are faster and easier to program. These are functions that apply (map) a given function over parts of a data structure. Several mapping functions are provided for different kinds of data structure and groupings.

Mapping functions	
apply	Map a function to the margins (rows or columns) of an array
sapply, lapply	Map a function to each cell of a vector, column of a data frame, or component of a list
replicate	Repeated evaluation of an expression
tapply	Map a function to a vector grouped by one or more factors
mapply	Map a multivariate function to corresponding cells of multiple vectors, columns of multiple data frames, or components of multiple lists
outer	Map a binary function to corresponding elements of two matrices
aggregate, by	Map a function to data frame columns grouped by one or more factors

[41] See: `help(na.fail)` and `help(naresid)`. Note that `na.exclude` works by passing a message, via the result's `na.action` attribute, to functions that subsequently process the result. To take advantage of `na.exclude` it is necessary to process the result with an appropriate function. For example `fitted(fit)` and `residuals(fit)` propagate `NA` values, but `fit$fitted` and `fit$residuals` do not.

[42] See: `help(Control)`.

2.7.1 Repeated Evaluation

The `replicate` function is often used in conjunction with `sample` for resampling and permutations[43]:

```
> x = c("bob", "carol", "ted", "alice")
> replicate(24, sample(x))   # Permute x 24 times
```

2.7.2 Applying Functions

Functions are provided for calculating row and column sums and means of a data frame. Functions `colSums` and `colMeans` calculate column sums and means, and functions `rowSums` and `rowMeans` calculate row sums and means. Function `rowsum` calculates column sums over rows grouped by a factor. Mapping functions[44] allow arbitrary functions to be applied.

Use `sapply` to map a function to each column of a data frame. For example the provided `iris` data set:

```
> sapply(iris, class)       # Apply class to columns of iris
> sapply(iris[1:4], mean)   # Apply mean to columns 1:4
```

Use `tapply` to map a function to subsets of a vector grouped by one or more factors. Pass multiple factor arguments within a list. For example the provided `warpbreaks` data set:

```
> attach(warpbreaks)
> tapply(breaks, tension, mean)
> tapply(breaks, list(wool, tension), mean)   # Factor arguments within a list
> detach(warpbreaks)
```

Instead of attaching the data frame to facilitate access to the variables you may prefer to use `with`:

```
> with(warpbreaks, tapply(breaks, list(wool, tension), mean))   # Using with
```

Functions `by` and `aggregate` both map a given function to subsets of data frame columns grouped by one or more factors. They differ in that function `by` returns a list, and function `aggregate` returns a data frame. Function `by` is oriented towards extracting more complicated information from each group, such as

[43] See also: `combn` to generate combinations of elements, `choose` for the number of combinations, and `factorial` for the size of a full permutation. See `permutations` in package `gtools` for enumerating a full permutation.

[44] See: `apropos("apply")`.

model fits, and returning these in a form that can subsequently be processed using `sapply`.[45] Function `aggregate` is oriented towards computing summary statistics for each group, and returning these in a form that can more easily be combined into a table using `cbind`.

```
> by(iris[1:4], list(iris$Species), mean)
> aggregate(iris[1:4], list(iris$Species), mean)
```

Passing Arguments to the Mapped Function

Mapping functions that have an argument named `"..."` allow additional optional arguments to be passed to the mapped function. For example it is often necessary to pass the argument `na.rm=TRUE` to a summary descriptive function to omit missing values from the calculation.

```
> sapply(iris[1:4], mean, na.rm=TRUE)
> sapply(iris[1:4], mean, na.rm=TRUE, trim=0.2)  # Trimmed mean
```

2.8 Writing Functions

Custom functions to extend the R language can be created using the `function` keyword.[46] For example a function to calculate the standard error of a sample vector `x` could be defined as follows:

```
> se = function(x) {
+       sd(x)/sqrt(length(x))
+ }
```

The function arguments are declared as the arguments to the `function` keyword. Here there is just one argument, named `x`. The calculations performed by the function is defined in the function's body between $\{\ldots\}$. The value returned by the function is the value of its final line.[47] The name of the function is the name of the variable you assign it to. Here the function is named `se`. This function could then be used as follows:

```
> y = rnorm(100)
> se(y)     # Call the se function, passing argument y
```

[45] See the examples in `help(by)`.

[46] See: `help("function")`.

[47] See also: `help(return)`.

The function can be mapped to a data structure in the same way as any provided function, for example:

```
> sapply(iris[1:4], se)   # Apply function se to columns 1:4
```

2.8.1 Anonymous Functions

If the custom function is a small one-off function it is typically defined at the place it is used as an "anonymous" function, (a function with no name). For example:

```
> sapply(iris[1:4], function(x) sd(x)/sqrt(length(x)))   # Anonymous se
> sapply(iris[1:4], function(x) sum(is.na(x)))           # Count NA
```

2.8.2 Optional Arguments

Optional arguments and their default values are declared as `name=value` in the argument list. Argument checking within the function can be reported by function `stop` (for fatal errors) or `warning` (for non-fatal warning messages).

For example a function to calculate the small sample confidence interval for the mean, and return a list containing the lower and upper limits:

```
> ci = function(x, conf=0.95) {
+     if(length(x)<2) stop("Not enough \'x\' observations")
+     y = t.test(x, conf.level=conf)$conf.int
+     list(lower=y[1], upper=y[2])
+ }
> # Using the function:
> y = rnorm(100)
> ci(y)                          # 95% CI
> ci(y, conf=0.99)               # 99% CI
```

A special argument '...' is used to pass arguments on to function calls within a function. For example:

```
> ci = function(x,...) {
+     if(length(x) < 2) stop("Not enough \'x\' observations")
+     y = t.test(x,...)$conf.int
+     list(lower=y[1],upper=y[2])
+ }
> ci(y, conf.level=0.99)   # 99% CI
```

http://www.springer.com/978-3-642-17979-2

A Tiny Handbook of R
Allerhand, M.
2011, IX, 83 p. 10 illus., Softcover
ISBN: 978-3-642-17979-2