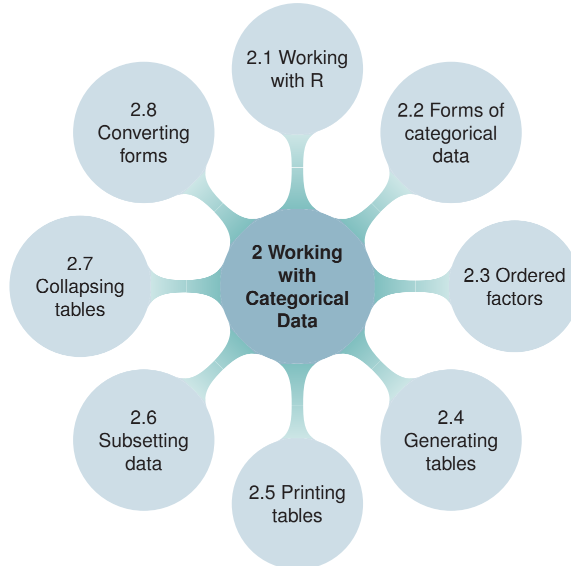

2

Working with Categorical Data

{ch:working}

Creating and manipulating categorical data sets requires some skills and techniques in R beyond those ordinarily used for quantitative data. This chapter illustrates these for the main formats for categorical data: case form, frequency form and table form.

> I'm a tidy sort of bloke. I don't like chaos. I kept records in the record rack, tea in the tea caddy, and pot in the pot box
>
> George Harrison, from `http://www.brainyquote.com/quotes/keywords/tidy.html`

Categorical data can be represented as data sets in various formats: case form, frequency form, and table form. This chapter describes and illustrates the skills and techniques in R needed to input, create, and manipulate R data objects to represent categorical data. More importantly, you also need to be able to convert these from one form to another for the purposes of statistical analysis and visualization, which are the subject of the remainder of the book.

As mentioned earlier, this book assumes that you have at least a basic knowledge of the R language and environment, including interacting with the R console (Rgui for Windows, R.app for Mac OS X) or some other editor/environment (e.g., R Studio), loading and using R functions in packages (e.g., `library(vcd)`) getting help for these from R (e.g., `help(matrix)`), etc. This chapter is therefore devoted to covering those topics needed in the book beyond such basic skills.[1]

[1] Some excellent introductory treatments of R are: Fox and Weisberg (2011, Chapter 2), Maindonald and Braun (2007), and Dalgaard (2008). Tom Short's *R Reference Card*, `http://cran.us.r-project.org/doc/contrib/Short-refcard.pdf`, is a handy 4-page summary of the main functions. The web sites Quick-R `http://www.statmethods.net/` and Cookbook for R `http://www.cookbook-r.com/` provide very helpful examples, organized by topics and tasks.

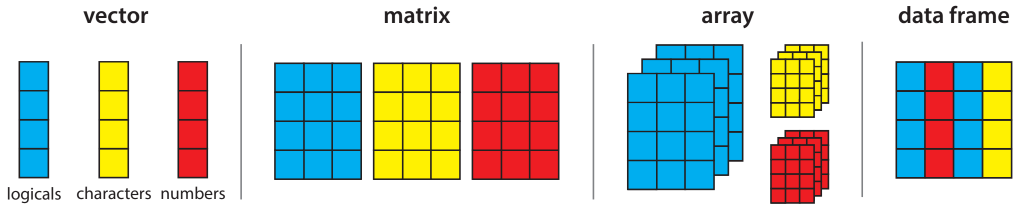

Figure 2.1: Principal data structures and data types in R. Colors represent different data types: numeric, character, logical.

{fig:datatypes}

2.1 Working with R data: vectors, matrices, arrays, and data frames

{sec:Rdata}

R has a wide variety of data structures for storing, manipulating, and calculating with data. Among these, vectors, matrices, arrays, and data frames are most important for the material in this book.

In R, a ***vector*** is a collection of values, like numbers, character strings, or logicals (`TRUE, FALSE`), and often correspond to a variable in some analysis. ***Matrices*** are rectangular arrays like a traditional table, composed of vectors in their columns or rows. ***Arrays*** add additional dimensions, so that, for example, a 3-way table can be represented as composed of rows, columns, and layers. An important consideration is that the values in vectors, matrices, and arrays must all be of the same *mode*, e.g., numbers or character strings. A ***data frame*** is a rectangular table, like a traditional data set in other statistical environments, and composed of rows and columns like a matrix, but allowing variables (columns) of different types. These data structures and the types of data they can contain are illustrated in Figure 2.1. A more general data structure is a *list*, a generic vector that can contain any other types of objects (including lists, allowing for *recursive* data structures). A data frame is basically a list of equally sized vectors, each representing a column of the data frame.

2.1.1 Vectors

The simplest data structure in R is a ***vector***, a one-dimensional collection of elements of the same type. An easy way to create a vector is with the `c()` function, which combines its arguments. The following examples create and print vectors of length 4, containing numbers, character strings, and logical values, respectively:

```
> c(17, 20, 15, 40)

[1] 17 20 15 40

> c("female", "male", "female", "male")

[1] "female" "male"   "female" "male"

> c(TRUE, TRUE, FALSE, FALSE)

[1]  TRUE  TRUE FALSE FALSE
```

To store these values in variables, R uses the assignment operator (<-) or equals sign (=). This creates a variable named on the left-hand side. An assignment doesn't print the result, but a bare expression does, so you can assign and print by surrounding the assignment with `()`.

```
> count <- c(17, 20, 15, 40)                          # assign
> count                                                # print

[1] 17 20 15 40

> (sex <- c("female", "male", "female", "male"))    # both

[1] "female" "male"   "female" "male"

> (passed <- c(TRUE, TRUE, FALSE, FALSE))

[1]  TRUE  TRUE FALSE FALSE
```

Other useful functions for creating vectors are:

- The `:` operator for generating consecutive integer sequences, e.g., `1:10` gives the integers 1 to 10. The `seq()` function is more general, taking the forms `seq(from, to)`, `seq(from, to, by= )`, and `seq(from, to, length.out= )` where the optional argument `by` specifies the interval between adjacent values and `length.out` gives the desired length of the result.
- The `rep()` function generates repeated sequences, replicating its first argument (which may be a vector) a given number of `times`, and individual elements can be repeated with `each` until an optional `length.out` is obtained.

```
> seq(10, 100, by = 10)          # give interval

 [1]  10  20  30  40  50  60  70  80  90 100

> seq(0, 1, length.out = 11)      # give length

 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> (sex <- rep(c("female", "male"), times = 2))

[1] "female" "male"   "female" "male"

> (sex <- rep(c("female", "male"), length.out = 4))   # same

[1] "female" "male"   "female" "male"

> (passed <- rep(c(TRUE, FALSE), each = 2))

[1]  TRUE  TRUE FALSE FALSE
```

2.1.2 Matrices

A **matrix** is a two-dimensional array of elements of the same type composed in a rectangular array of rows and columns. Matrices can be created by the function `matrix(values, nrow, ncol)`, which reshapes the elements in the first argument (`values`) to a matrix with `nrow` rows and `ncol` columns. By default, the elements are filled in columnwise, unless the optional argument `byrow = TRUE` is given.

```
> (matA <- matrix(1:8, nrow = 2, ncol = 4))

     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

```
> (matB <- matrix(1:8, nrow = 2, ncol = 4, byrow = TRUE))

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8

> (matC <- matrix(1:4, nrow = 2, ncol = 4))

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

The last example illustrates that the values in the first argument are recycled as necessary to fill the given number of rows and columns.

All matrices have a dimensions attribute, a vector of length two giving the number of rows and columns, retrieved with the function `dim()`. Labels for the rows and columns can be assigned using `dimnames()`,[2] which takes a list of two vectors for the row names and column names, respectively. To see the structure of a matrix (or any other R object) and its attributes, you can use the `str()` function, as shown in the example below.

```
> dim(matA)

[1] 2 4

> str(matA)

 int [1:2, 1:4] 1 2 3 4 5 6 7 8

> dimnames(matA) <- list(c("M", "F"), LETTERS[1:4])
> matA

  A B C D
M 1 3 5 7
F 2 4 6 8

> str(matA)

 int [1:2, 1:4] 1 2 3 4 5 6 7 8
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:2] "M" "F"
  ..$ : chr [1:4] "A" "B" "C" "D"
```

Additionally, names for the row and column *variables* themselves can also be assigned in the `dimnames` call by giving each dimension vector a name.

```
> dimnames(matA) <- list(sex = c("M", "F"), group = LETTERS[1:4])
> ## or: names(dimnames(matA)) <- c("Sex", "Group")
> matA

   group
sex A B C D
  M 1 3 5 7
  F 2 4 6 8

> str(matA)

 int [1:2, 1:4] 1 2 3 4 5 6 7 8
 - attr(*, "dimnames")=List of 2
  ..$ sex  : chr [1:2] "M" "F"
  ..$ group: chr [1:4] "A" "B" "C" "D"
```

[2] The `dimnames` can also be specified as an optional argument to `matrix()`.

(`LETTERS` is a predefined character vector of the 26 uppercase letters). Matrices can also be created or enlarged by "binding" vectors or matrices together by rows or columns:

- `rbind(a, b, c)` creates a matrix with the vectors `a`, `b`, and `c` as its rows, recycling the elements as necessary to the length of the longest one.
- `cbind(a, b, c)` creates a matrix with the vectors `a`, `b`, and `c` as its columns.
- `rbind(mat, a, b, ...)` and `cbind(mat, a, b, ...)` add additional rows (columns) to a matrix `mat`, recycling or subsetting the elements in the vectors to conform with the size of the matrix.

```
> rbind(matA, c(10, 20))

   A  B  C  D
M  1  3  5  7
F  2  4  6  8
  10 20 10 20

> cbind(matA, c(10, 20))

  A B C D
M 1 3 5 7 10
F 2 4 6 8 20
```

Rows and columns can be swapped (transposed) using `t()`:

```
> t(matA)

     sex
group M F
    A 1 2
    B 3 4
    C 5 6
    D 7 8
```

Finally, we note that basic computations involving matrices are performed *element-wise*:

```
> 2 * matA / 100

   group
sex    A    B    C    D
  M 0.02 0.06 0.10 0.14
  F 0.04 0.08 0.12 0.16
```

Special operators and functions do exist for matrix operations, such as `%*%` for the matrix product.

2.1.3 Arrays

Higher-dimensional arrays are less frequently encountered in traditional data analysis, but they are of great use for categorical data, where frequency tables of three or more variables can be naturally represented as arrays, with one dimension for each table variable.

The function `array(values, dim)` takes the elements in `values` and reshapes these into an array whose dimensions are given in the vector `dim`. The number of dimensions is the length of `dim`. As with matrices, the elements are filled in with the first dimension (rows) varying most rapidly, then by the second dimension (columns) and so on for all further dimensions, which can be considered as layers. A matrix is just the special case of an array with two dimensions.

```
> dims <-  c(2, 4, 2)
> (arrayA <- array(1:16, dim = dims))       # 2 rows, 4 columns, 2 layers

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

, , 2

     [,1] [,2] [,3] [,4]
[1,]    9   11   13   15
[2,]   10   12   14   16

> str(arrayA)

 int [1:2, 1:4, 1:2] 1 2 3 4 5 6 7 8 9 10 ...

> (arrayB <- array(1:16, dim = c(2, 8)))    # 2 rows, 8 columns

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    3    5    7    9   11   13   15
[2,]    2    4    6    8   10   12   14   16

> str(arrayB)

 int [1:2, 1:8] 1 2 3 4 5 6 7 8 9 10 ...
```

In the same way that we can assign labels to the rows and columns in matrices, we can assign these attributes to `dimnames(arrayA)`, or include this information in a `dimnames=` argument to `array()`.

```
> dimnames(arrayA) <- list(sex = c("M", "F"),
+                          group = letters[1:4],
+                          time = c("Pre", "Post"))
> arrayA

, , time = Pre

   group
sex a b c d
  M 1 3 5 7
  F 2 4 6 8

, , time = Post

   group
sex  a  b  c  d
  M  9 11 13 15
  F 10 12 14 16

> str(arrayA)

 int [1:2, 1:4, 1:2] 1 2 3 4 5 6 7 8 9 10 ...
 - attr(*, "dimnames")=List of 3
  ..$ sex  : chr [1:2] "M" "F"
  ..$ group: chr [1:4] "a" "b" "c" "d"
  ..$ time : chr [1:2] "Pre" "Post"
```

Arrays in R can contain any single type of elements— numbers, character strings, logicals. R also has a variety of functions (e.g., `table()`, `xtabs()`) for creating and manipulating "table" objects, which are specialized forms of matrices and arrays containing integer frequencies in a contingency table. These are discussed in more detail below (Section 2.4).

2.1.4 Data frames

{sec:data-frames}

Data frames are the most commonly used form of data in R and more general than matrices in that they can contain columns of different types. For statistical modeling, data frames play a special role, in that many modeling functions are designed to take a data frame as a `data=` argument, and then find the variables mentioned within that data frame. Another distinguishing feature is that discrete variables (columns) like character strings `("M", "F")` or integers `(1, 2, 3)` in data frames can be represented as ***factor***s, which simplifies many statistical and graphical methods.

A data frame can be created using keyboard input with the `data.frame()` function, applied to a list of objects, `data.frame(a, b, c, ...)`, each of which can be a vector, matrix, or another data frame, but typically all containing the same number of rows. This works roughly like `cbind()`, collecting the arguments as columns in the result.

The following example generates `n = 100` random observations on three discrete factor variables, `A, B, sex`, and a numeric variable, `age`. As constructed, all of these are statistically independent, since none depends on any of the others. The function `sample()` is used here to generate `n` random samples from the first argument allowing replacement (`replace = TRUE`). The `rnorm()` function produces a vector of `n` normally distributed values with mean 30 and standard deviation 5. The call to `set.seed()` guarantees the reproducibility of the resulting data. Finally, all four variables are combined into the data frame `mydata`.

```
> set.seed(12345)   # reproducibility
> n <- 100
> A <- factor(sample(c("a1", "a2"), n, replace = TRUE))
> B <- factor(sample(c("b1", "b2"), n, replace = TRUE))
> sex <- factor(sample(c("M", "F"), n, replace = TRUE))
> age <- round(rnorm(n, mean = 30, sd = 5))
> mydata <- data.frame(A, B, sex, age)
> head(mydata, 5)

   A  B sex age
1 a2 b1   F  22
2 a2 b2   F  33
3 a2 b2   M  31
4 a2 b2   F  26
5 a1 b2   F  29

> str(mydata)

'data.frame': 100 obs. of  4 variables:
 $ A  : Factor w/ 2 levels "a1","a2": 2 2 2 2 1 1 1 2 2 2 ...
 $ B  : Factor w/ 2 levels "b1","b2": 1 2 2 2 2 2 2 2 1 1 ...
 $ sex: Factor w/ 2 levels "F","M": 1 1 2 1 1 1 2 2 1 1 ...
 $ age: num  22 33 31 26 29 29 38 28 30 27 ...
```

Rows, columns, and individual values in a data frame can be manipulated in the same way as a matrix, using subscripting (`[,]`). Additionally, variables can be extracted using the `$` operator:

```
> mydata[1,2]

[1] b1
Levels: b1 b2

> mydata$sex
```

```
  [1] F F M F F F M M F F M F M M F M M F F M M M M F F F F M M F
 [31] M F M F F F F F M M F F F F F F F F M F M F M M F F M M M M
 [61] F F F F F M M F F F M M M F F M F M M F M F M M M M M F F F
 [91] F F M M F M F M F M
Levels: F M

> ##same as: mydata[,"sex"] or mydata[,3]
```

Values in data frames can also be edited conveniently using, e.g., `fix(mydata)`, opening a simple, spreadsheet-like editor.

For real data sets, it is usually most convenient to read these into R from external files, and this is easiest using plain text (ASCII) files with one line per observation and fields separated by commas (or tabs), and with a first header line giving the variable names—called *comma-separated* or CSV format. If your data is in the form of Excel, SAS, SPSS, or other file format, you can almost always export that data to CSV format first.[3]

The function `read.table()` has many options to control the details of how the data are read and converted to variables in the data frame. Among these some important options are:

`header` indicates whether the first line contains variable names. The default is `FALSE` unless the first line contains one fewer field than the number of columns;
`sep` (default: `""`, meaning white space, i.e., one or more spaces, tabs or newlines) specifies the separator character between fields;
`stringsAsFactors` (default: `TRUE`) determines whether character string variables should be converted to factors;
`na.strings` (default: `"NA"`) refers to one or more strings that are interpreted as missing data values (`NA`);

For delimited files, `read.csv()` and `read.delim()` are convenient wrappers to `read.table()`, with default values `sep=","` and `sep="\t"` respectively, and `header=TRUE`.

{ex:ch2-arth-csv}

EXAMPLE 2.1: Arthritis treatment

The file `Arthritis.csv` contains data in CSV format from Koch and Edwards (1988), representing a double-blind clinical trial investigating a new treatment for rheumatoid arthritis with 84 patients.[4] The first ("header") line gives the variable names. Some of the lines in the file are shown below, with `...` representing omitted lines:

```
ID,Treatment,Sex,Age,Improved
57,Treated,Male,27,Some
46,Treated,Male,29,None
77,Treated,Male,30,None
17,Treated,Male,32,Marked
  ...
42,Placebo,Female,66,None
15,Placebo,Female,66,Some
71,Placebo,Female,68,Some
1,Placebo,Female,74,Marked
```

We read this into R using `read.table()` as shown below:

---

[3]The foreign (R Core Team, 2015) package contains specialized functions to *directly* read data stored by Minitab, SAS, SPSS, Stata, Systat, and other software. There are also a number of packages for reading (and writing) Excel spreadsheets directly (gdata (Warnes et al., 2014), XLConnect (Mirai Solutions GmbH, 2015), xlsx (Dragulescu, 2014)). The R manual, *R Data Import/Export* covers many other variations, including data in relational data bases.

[4]This data set can be created using: `library(vcd); write.table(Arthritis, file = "Arthritis.csv", quote = FALSE, sep = ",")`.

```
> path <- "ch02/Arthritis.csv" ## set path
> ## for convenience, use path <- file.choose() to retrieve a path
> ## then, use file.show(path) to inspect the data format
> Arthritis <- read.table(path, header = TRUE, sep = ",")
> str(Arthritis)

'data.frame': 84 obs. of  5 variables:
 $ ID       : int  57 46 77 17 36 23 75 39 33 55 ...
 $ Treatment: Factor w/ 2 levels "Placebo","Treated": 2 2 2 2 2 2 2 2 2 2 ...
 $ Sex      : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 2 2 2 2 ...
 $ Age      : int  27 29 30 32 46 58 59 59 63 63 ...
 $ Improved : Factor w/ 3 levels "Marked","None",..: 3 2 2 1 1 1 2 1 2 2 ...
```

Note that the character variables `Treatment`, `Sex`, and `Improved` were converted to factors, and the levels of those variables were ordered *alphabetically*. This often doesn't matter much for binary variables, but here, the response variable `Improved` has levels that should be considered *ordered*, as `c("None", "Some", "Marked")`. We can correct this here by re-assigning `Arthritis$Improved` using `ordered()`. The topic of re-ordering variables and levels in categorical data is considered in more detail in Section 2.3.

```
> levels(Arthritis$Improved)

[1] "Marked" "None"   "Some"

> Arthritis$Improved <- ordered(Arthritis$Improved,
+                                    levels = c("None", "Some", "Marked"))
```

△

2.2 Forms of categorical data: case form, frequency form, and table form

{sec:forms}

As we saw in Chapter 1, categorical data can be represented as ordinary data sets in case form, but the discrete nature of factors or stratifying variables allows the same information to be represented more compactly in summarized form with a frequency variable for each cell of factor combinations, or in tables. Consequently, we sometimes find data created or presented in one form (e.g., a spreadsheet data set, a two-way table of frequencies) and want to input that into R. Once we have the data in R, it is often necessary to manipulate the data into some other form for the purposes of statistical analysis, visualizing results, and our own presentation. It is useful to understand the three main forms of categorical data in R and how to work with them for our purposes.

2.2.1 Case form

Categorical data in case form are simply data frames, with one or more discrete classifying variables or response variables, most conveniently represented as factors or ordered factors. In case form, the data set can also contain numeric variables (covariates or other response variables) that cannot be accommodated in other forms.

As with any data frame, `X`, you can access or compute with its attributes using `nrow(X)` for the number of observations, `ncol(X)` for the number of variables, `names(X)` or `colnames(X)` for the variable names, and so forth.

{ex:ch2-arth}

EXAMPLE 2.2: Arthritis treatment

The *`Arthritis`* data is available in case form in the vcd (Meyer et al., 2015) package. There are two explanatory factors: `Treatment` and `Sex`. `Age` is a numeric covariate, and `Improved` is the response—an ordered factor, with levels `"None" < "Some" < "Marked"`. Excluding `Age`, we would have a $2 \times 2 \times 3$ contingency table for `Treatment`, `Sex`, and `Improved`.

```
> data("Arthritis", package = "vcd")  # load the data
> names(Arthritis)       # show the variables

[1] "ID"        "Treatment" "Sex"       "Age"       "Improved"

> str(Arthritis)         # show the structure

'data.frame': 84 obs. of  5 variables:
 $ ID       : int  57 46 77 17 36 23 75 39 33 55 ...
 $ Treatment: Factor w/ 2 levels "Placebo","Treated": 2 2 2 2 2 2 2 2 2 2 ...
 $ Sex      : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 2 2 2 2 ...
 $ Age      : int  27 29 30 32 46 58 59 59 63 63 ...
 $ Improved : Ord.factor w/ 3 levels "None"<"Some"<..: 2 1 1 3 3 3 1 3 1 1 ...

> head(Arthritis, 5)     # first 5 observations, same as Arthritis[1:5,]

  ID Treatment  Sex Age Improved
1 57   Treated Male  27     Some
2 46   Treated Male  29     None
3 77   Treated Male  30     None
4 17   Treated Male  32   Marked
5 36   Treated Male  46   Marked
```

△

2.2.2 Frequency form

Data in frequency form is also a data frame, containing one or more discrete factor variables and a frequency variable (often called `Freq` or `count`) representing the number of basic observations in that cell.

This is an alternative representation of a table form data set considered below. In frequency form, the number of cells in the equivalent table is `nrow(X)`, and the total number of observations is the sum of the frequency variable, `sum(X$Freq)`, `sum(X[,"Freq"])` or a similar expression.

{ex:ch2-GSS}

EXAMPLE 2.3: General social survey

For small frequency tables, it is often convenient to enter them in frequency form using `expand.grid()` for the factors and `c()` to list the counts in a vector. The example below, from Agresti (2002), gives results for the 1991 General Social Survey, with respondents classified by sex and party identification. As a table, the data look like this:

	party		
sex	dem	indep	rep
female	279	73	225
male	165	47	191

We use `expand.grid()` to create a 6×2 matrix containing the combinations of `sex` and `party` with the levels for `sex` given first, so that this varies most rapidly. Then, input the frequencies in the table by columns from left to right, and combine these two results with `data.frame()`.

```
> # Agresti (2002), table 3.11, p. 106
> tmp <- expand.grid(sex = c("female", "male"),
+                    party = c("dem", "indep", "rep"))
> tmp

     sex party
1 female   dem
2   male   dem
3 female indep
```

```
4   male indep
5 female   rep
6   male   rep

> GSS <- data.frame(tmp, count = c(279, 165, 73, 47, 225, 191))
> GSS

     sex party count
1 female   dem   279
2   male   dem   165
3 female indep    73
4   male indep    47
5 female   rep   225
6   male   rep   191

> names(GSS)

[1] "sex"   "party" "count"

> str(GSS)

'data.frame': 6 obs. of  3 variables:
 $ sex  : Factor w/ 2 levels "female","male": 1 2 1 2 1 2
 $ party: Factor w/ 3 levels "dem","indep",..: 1 1 2 2 3 3
 $ count: num  279 165 73 47 225 191

> sum(GSS$count)

[1] 980
```

The last line above shows that there are 980 cases represented in the frequency table. △

2.2.3 Table form

Table form data is represented as a matrix, array, or table object whose elements are the frequencies in an n-way table. The number of dimensions of the table is the length, `length(dim(X))`, of its `dim` (or `dimnames`) attribute, and the sizes of the dimensions in the table are the elements of `dim(X)`. The total number of observations represented is the sum of all the frequencies, `sum(X)`. {ex:ch2-hec}

EXAMPLE 2.4: Hair color and eye color

A classic data set on frequencies of hair color, eye color, and sex is given in table form in *HairEyeColor* in the **datasets** package, reporting the frequencies of these categories for 592 students in a statistics course.

```
> data("HairEyeColor", package = "datasets")    # load the data
> str(HairEyeColor)                  # show the structure

 table [1:4, 1:4, 1:2] 32 53 10 3 11 50 10 30 10 25 ...
 - attr(*, "dimnames")=List of 3
  ..$ Hair: chr [1:4] "Black" "Brown" "Red" "Blond"
  ..$ Eye : chr [1:4] "Brown" "Blue" "Hazel" "Green"
  ..$ Sex : chr [1:2] "Male" "Female"

> dim(HairEyeColor)                  # table dimension sizes

[1] 4 4 2

> dimnames(HairEyeColor)            # variable and level names
```

```
$Hair
[1] "Black" "Brown" "Red"   "Blond"

$Eye
[1] "Brown" "Blue"  "Hazel" "Green"

$Sex
[1] "Male"   "Female"

> sum(HairEyeColor)                  # number of cases

[1] 592
```

Three-way (and higher-way) tables can be printed in a more convenient form using `structable()` and `ftable()` as described below in Section 2.5. △

Tables are often created from raw data in case form or frequency form using the functions `table()` and `xtabs()` described in Section 2.4. For smallish frequency tables that are already in tabular form, you can enter the frequencies in a matrix, and then assign `dimnames` and other attributes.

To illustrate, we create the GSS data as a table below, entering the values in the table by rows (`byrow=TRUE`), as they appear in printed form.

```
> GSS.tab <- matrix(c(279, 73, 225,
+                     165, 47, 191),
+                   nrow = 2, ncol = 3, byrow = TRUE)
> dimnames(GSS.tab) <- list(sex = c("female", "male"),
+                           party = c("dem", "indep", "rep"))
> GSS.tab

        party
sex       dem indep rep
  female 279    73 225
  male   165    47 191
```

`GSS.tab` is a matrix, not an object of `class("table")`, and some functions are happier with tables than matrices.[5] You should therefore coerce it to a table with `as.table()`,

```
> GSS.tab <- as.table(GSS.tab)
> str(GSS.tab)

 table [1:2, 1:3] 279 165 73 47 225 191
 - attr(*, "dimnames")=List of 2
  ..$ sex  : chr [1:2] "female" "male"
  ..$ party: chr [1:3] "dem" "indep" "rep"
```

{ex:jobsat1}

EXAMPLE 2.5: Job satisfaction

Here is another similar example, entering data on job satisfaction classified by `income` and level of `satisfaction` from a 4×4 table given by Agresti (2002, Table 2.8, p. 57).

```
> ## A 4 x 4 table  Agresti (2002, Table 2.8, p. 57) Job Satisfaction
> JobSat <- matrix(c(1, 2, 1, 0,
+                    3, 3, 6, 1,
+                    10, 10, 14, 9,
+                    6, 7, 12, 11),
```

[5] There are quite a few functions in R with specialized methods for "table" objects. For example, `plot(GSS.tab)` gives a mosaic plot and `barchart(GSS.tab)` gives a divided bar chart.

```
+                    nrow = 4, ncol = 4)
> dimnames(JobSat) <-
+   list(income = c("< 15k", "15-25k", "25-40k", "> 40k"),
+        satisfaction = c("VeryD", "LittleD", "ModerateS", "VeryS"))
> JobSat <- as.table(JobSat)
> JobSat

        satisfaction
income   VeryD LittleD ModerateS VeryS
  < 15k      1       3        10     6
  15-25k     2       3        10     7
  25-40k     1       6        14    12
  > 40k      0       1         9    11
```

△

2.3 Ordered factors and reordered tables

{sec:ordered}

As we saw above (Example 2.1), the levels of factor variables in data frames (case form or frequency form) can be re-ordered (and the variables declared as ordered factors) using `ordered()`. As well, the order of the factor values themselves can be rearranged by sorting the data frame using `sort()`.

However, in table form, the values of the table factors are ordered by their position in the table. Thus in the *JobSat* data, both `income` and `satisfaction` represent ordered factors, and the *positions* of the values in the rows and columns reflect their ordered nature, but only implicitly.

Yet, for analysis or graphing, there are occasions when you need *numeric* values for the levels of ordered factors in a table, e.g., to treat a factor as a quantitative variable. In such cases, you can simply re-assign the `dimnames` attribute of the table variables. For example, here, we assign numeric values to `income` as the middle of their ranges, and treat `satisfaction` as equally spaced with integer scores.

```
> dimnames(JobSat)$income <- c(7.5, 20, 32.5, 60)
> dimnames(JobSat)$satisfaction <- 1:4
```

A related case is when you want to preserve the character labels of table dimensions, but also allow them to be sorted in some particular order. A simple way to do this is to prefix each label with an integer index using `paste()`.

```
> dimnames(JobSat)$income <-
+     paste(1:4, dimnames(JobSat)$income, sep = ":")
> dimnames(JobSat)$satisfaction <-
+     paste(1:4, dimnames(JobSat)$satisfaction, sep = ":")
```

A different situation arises with tables where you want to *permute* the levels of one or more variables to arrange them in a more convenient order without changing their labels. For example, in the *HairEyeColor* table, hair color and eye color are ordered arbitrarily.

For visualizing the data using mosaic plots and other methods described later, it turns out to be more useful to assure that both hair color and eye color are ordered from dark to light. Hair colors are actually ordered this way already: `"Black", "Brown", "Red", "Blond"`. But eye colors are ordered as `"Brown", "Blue", "Hazel", "Green"`. It is easiest to re-order the eye colors by indexing the columns (dimension 2) in this array to a new order, `"Brown", "Hazel", "Green", "Blue"`, giving the indices of the old levels in the new order (here: 1,3,4,2). Again `str()` is your friend, showing the structure of the result to check that the result is what you want.

```
> data("HairEyeColor", package = "datasets")
> HEC <- HairEyeColor[, c(1, 3, 4, 2), ]
> str(HEC)

 num [1:4, 1:4, 1:2] 32 53 10 3 10 25 7 5 3 15 ...
 - attr(*, "dimnames")=List of 3
  ..$ Hair: chr [1:4] "Black" "Brown" "Red" "Blond"
  ..$ Eye : chr [1:4] "Brown" "Hazel" "Green" "Blue"
  ..$ Sex : chr [1:2] "Male" "Female"
```

Finally, there are situations where, particularly for display purposes, you want to re-order the *dimensions* of an n-way table, and/or change the labels for the variables or levels. This is easy when the data are in table form: `aperm()` permutes the dimensions, and assigning to `names` and `dimnames` changes variable names and level labels, respectively.

```
> str(UCBAdmissions)

 table [1:2, 1:2, 1:6] 512 313 89 19 353 207 17 8 120 205 ...
 - attr(*, "dimnames")=List of 3
  ..$ Admit : chr [1:2] "Admitted" "Rejected"
  ..$ Gender: chr [1:2] "Male" "Female"
  ..$ Dept  : chr [1:6] "A" "B" "C" "D" ...

> # vary along the 2nd, 1st, and 3rd dimension in UCBAdmissions
> UCB <- aperm(UCBAdmissions, c(2, 1, 3))
> dimnames(UCB)$Admit <- c("Yes", "No")
> names(dimnames(UCB)) <- c("Sex", "Admitted", "Department")
> str(UCB)

 table [1:2, 1:2, 1:6] 512 89 313 19 353 17 207 8 120 202 ...
 - attr(*, "dimnames")=List of 3
  ..$ Sex       : chr [1:2] "Male" "Female"
  ..$ Admitted  : chr [1:2] "Yes" "No"
  ..$ Department: chr [1:6] "A" "B" "C" "D" ...
```

2.4 Generating tables with `table()` and `xtabs()`

{sec:table}

With data in case form or frequency form, you can generate frequency tables from factor variables in data frames using the `table()` function; for tables of proportions, use the `prop.table()` function, and for marginal frequencies (summing over some variables) use `margin.table()`. The examples below use the same case-form data frame `mydata` used earlier (Section 2.1.4).

```
> set.seed(12345)   # reproducibility
> n <- 100
> A <- factor(sample(c("a1", "a2"), n, replace = TRUE))
> B <- factor(sample(c("b1", "b2"), n, replace = TRUE))
> sex <- factor(sample(c("M", "F"), n, replace = TRUE))
> age <- round(rnorm(n, mean = 30, sd = 5))
> mydata <- data.frame(A, B, sex, age)
```

2.4.1 `table()`

{sec:table2}

`table(...)` takes a list of variables interpreted as factors, or a data frame whose columns are so interpreted. It does not take a `data=` argument, so either supply the names of columns in the data frame (possibly using `with()` for convenience), or select the variables using column indexes:

```
> # 2-Way Frequency Table
> table(mydata$A, mydata$B)          # A will be rows, B will be columns


     b1 b2
  a1 18 30
  a2 22 30

> ## same: with(mydata, table(A, B))
> (mytab <- table(mydata[,1:2]))      # same

    B
A    b1 b2
  a1 18 30
  a2 22 30
```

We can use `margin.table(X, margin)` to sum a table `X` for the indices in `margin`, i.e., over the dimensions not included in `margin`. A related function is `addmargins(X, margin, FUN = sum)`, which extends the dimensions of a table or array with the marginal values calculated by `FUN`.

```
> margin.table(mytab)       # sum over A & B

[1] 100

> margin.table(mytab, 1)    # A frequencies (summed over B)

A
a1 a2
48 52

> margin.table(mytab, 2)    # B frequencies (summed over A)

B
b1 b2
40 60

> addmargins(mytab)         # show all marginal totals

     B
A      b1  b2 Sum
  a1   18  30  48
  a2   22  30  52
  Sum  40  60 100
```

The function `prop.table()` expresses the table entries as a fraction of a given marginal table.

```
> prop.table(mytab)         # cell proportions

    B
A      b1   b2
  a1 0.18 0.30
  a2 0.22 0.30

> prop.table(mytab, 1)      # row proportions

    B
A         b1      b2
  a1 0.37500 0.62500
  a2 0.42308 0.57692
```

```
> prop.table(mytab, 2)      # column proportions

    B
A      b1   b2
  a1 0.45 0.50
  a2 0.55 0.50
```

`table()` can also generate multidimensional tables based on 3 or more categorical variables. In this case, use the `ftable()` or `structable()` function to print the results more attractively as a “flat” (2-way) table.

```
> # 3-Way Frequency Table
> mytab <- table(mydata[,c("A", "B", "sex")])
> ftable(mytab)

      sex  F  M
A  B
a1 b1      9  9
   b2     15 15
a2 b1     12 10
   b2     19 11
```

`table()` ignores missing values by default, but has optional arguments `useNA` and `exclude` that can be used to control this. See `help(table)` for the details.

2.4.2 `xtabs()`

{sec:xtabs}

The `xtabs()` function allows you to create cross tabulations of data using formula style input. This typically works with case-form or frequency-form data supplied in a data frame or a matrix. The result is a contingency table in array format, whose dimensions are determined by the terms on the right side of the formula. As shown below, the `summary` method for tables produces a simple χ^2 test of independence of all factors, and indicates the number of cases and dimensions.

```
> # 3-Way Frequency Table
> mytable <- xtabs(~ A + B + sex, data = mydata)
> ftable(mytable)     # print table

      sex  F  M
A  B
a1 b1      9  9
   b2     15 15
a2 b1     12 10
   b2     19 11

> summary(mytable)    # chi-squared test of independence

Call: xtabs(formula = ~A + B + sex, data = mydata)
Number of cases in table: 100
Number of factors: 3
Test for independence of all factors:
Chisq = 1.54, df = 4, p-value = 0.82
```

When the data have already been tabulated in frequency form, include the frequency variable (usually `count` or `Freq`) on the left side of the formula, as shown in the example below for the GSS data.

```
> (GSStab <- xtabs(count ~ sex + party, data = GSS))
```

```
        party
sex      dem indep rep
  female 279    73 225
  male   165    47 191

> summary(GSStab)

Call: xtabs(formula = count ~ sex + party, data = GSS)
Number of cases in table: 980
Number of factors: 2
Test for independence of all factors:
Chisq = 7, df = 2, p-value = 0.03
```

For "table" objects, the `plot` method produces basic mosaic plots using the `mosaicplot()` function from the graphics package.

2.5 Printing tables with `structable()` and `ftable()`

{sec:structable}

2.5.1 Text output

For 3-way and larger tables, the functions `ftable()` (in the stats package) and `structable()` (in vcd) provide a convenient and flexible tabular display in a "flat" (2-way) format.

With `ftable(X, row.vars=, col.vars=)`, variables assigned to the rows and/or columns of the result can be specified as the integer numbers or character names of the variables in the array `X`. By default, the last variable is used for the columns. The formula method, in the form `ftable(colvars ~ rowvars, data)` allows a formula where the left- and right-hand side of formula specify the column and row variables, respectively.

```
>  ftable(UCB)                      # default

                Department   A   B   C   D   E   F
Sex    Admitted
Male   Yes                 512 353 120 138  53  22
       No                  313 207 205 279 138 351
Female Yes                  89  17 202 131  94  24
       No                   19   8 391 244 299 317

> #ftable(UCB, row.vars = 1:2)      # same result
>  ftable(Admitted + Sex ~ Department, data = UCB)    # formula method

           Admitted  Yes         No
           Sex      Male Female Male Female
Department
A                    512     89  313     19
B                    353     17  207      8
C                    120    202  205    391
D                    138    131  279    244
E                     53     94  138    299
F                     22     24  351    317
```

The `structable()` function is similar, but more general, and uses recursive splits in the vertical or horizontal directions (similar to the construction of mosaic displays). It works with both data frames and table objects.

```
> library(vcd)
> structable(HairEyeColor)                    # show the table: default

             Eye Brown Blue Hazel Green
```

```
Hair  Sex
Black Male          32   11    10     3
      Female        36    9     5     2
Brown Male          53   50    25    15
      Female        66   34    29    14
Red   Male          10   10     7     7
      Female        16    7     7     7
Blond Male           3   30     5     8
      Female         4   64     5     8

> structable(Hair + Sex ~ Eye, HairEyeColor) # specify col ~ row variables

      Hair Black        Brown        Red        Blond
      Sex   Male Female  Male Female Male Female  Male Female
Eye
Brown         32     36    53     66   10     16     3      4
Blue          11      9    50     34   10      7    30     64
Hazel         10      5    25     29    7      7     5      5
Green          3      2    15     14    7      7     8      8
```

It also returns an object of class `"structable"` for which there are a variety of special methods. For example, the transpose function `t()` interchanges rows and columns, so that a call like `t(structable(HairEyeColor))` produces the second result shown just above. There are also plot methods: for example, `plot()` produces mosaic plots from the vcd package.

2.6 Subsetting data

{sec:subsettingdata}

Often, the analysis of some data set is focused on a subset only. For example, the *HairEyeColor* data set introduced above tabulates frequencies of hair and eye colors for male and female students—the analysis could concentrate on one group only, or compare both groups in a stratified analysis. This section deals with extracting subsets of data in tables, structables, or data frames.

2.6.1 Subsetting tables

{sec:subsettingtables}

If data are available in tabular form created with `table()` or `xtabs()`, resulting in `table` objects, subsetting is done via indexing, either with integers or character strings corresponding to the factor levels. The following code extracts the female data from the `HairEyeColor` data set:

```
> HairEyeColor[,,"Female"]

        Eye
Hair     Brown Blue Hazel Green
  Black     36    9     5     2
  Brown     66   34    29    14
  Red       16    7     7     7
  Blond      4   64     5     8

> ##same using index: HairEyeColor[,,2]
```

Empty indices stand for taking all data of the corresponding dimension. The third one (Sex) is fixed at the second (“`Female`”) level. Note that in this case, the dimensionality is reduced to a two-way table, since dimensions with only one level are dropped by default. Functions like `apply()` can iterate through all levels of one or several dimensions and apply a function to each subset. The following calculates the total amount of male and female students:

```
> apply(HairEyeColor, 3, sum)

  Male Female
   279    313
```

It is of course possible to select more than one level:

```
> HairEyeColor[c("Black", "Brown"), c("Hazel", "Green"),]

, , Sex = Male

       Eye
Hair    Hazel Green
  Black    10     3
  Brown    25    15

, , Sex = Female

       Eye
Hair    Hazel Green
  Black     5     2
  Brown    29    14
```

2.6.2 Subsetting structables

{sec:subsettingstructables}

Structables work in a similar way, but take into account the hierarchical structure imposed by the "flattened" format, and also distinguish explicitly between subsetting levels and subsetting tables. In the following example, compare the different effects of applying the `[` and `[[` operators to the structable:

```
> hec <- structable(Eye ~ Sex + Hair, data = HairEyeColor)
> hec

             Eye Brown Blue Hazel Green
Sex    Hair
Male   Black        32   11    10     3
       Brown        53   50    25    15
       Red          10   10     7     7
       Blond         3   30     5     8
Female Black        36    9     5     2
       Brown        66   34    29    14
       Red          16    7     7     7
       Blond         4   64     5     8

> hec["Male",]

           Eye Brown Blue Hazel Green
Sex  Hair
Male Black        32   11    10     3
     Brown        53   50    25    15
     Red          10   10     7     7
     Blond         3   30     5     8

> hec[["Male",]]

      Eye Brown Blue Hazel Green
Hair
Black        32   11    10     3
Brown        53   50    25    15
Red          10   10     7     7
Blond         3   30     5     8
```

The first form keeps the dimensionality, whereas the second conditions on the "Male" level and returns the corresponding subtable. The following does this twice, once for Sex, and once for Hair (restricted to the Male level):

```
> hec[[c("Male", "Brown"),]]

Eye Brown Blue Hazel Green

       53   50    25    15
```

2.6.3 Subsetting data frames

{sec:subsettingdf}

Data available in data frames (frequency or case form) can also be subsetted, either by using indexes on the rows and/or columns, or, more conveniently, by applying the subset() function. The following statement will extract the Treatment and Improved variables for all female patients older than 68:

```
> rows <- Arthritis$Sex == "Female" & Arthritis$Age > 68
> cols <- c("Treatment", "Improved")
> Arthritis[rows, cols]

   Treatment Improved
39   Treated     None
40   Treated     Some
41   Treated     Some
84   Placebo   Marked
```

Note the use of the single & for the logical expression selecting the rows. The same result can be achieved more conveniently using the subset() function, first taking the data set, followed by an expression for selecting the rows (evaluated in the context of the data frame), and then an expression for selecting the columns:

```
> subset(Arthritis, Sex == "Female" & Age > 68,
+        select = c(Treatment, Improved))

   Treatment Improved
39   Treated     None
40   Treated     Some
41   Treated     Some
84   Placebo   Marked
```

Note the non-standard evaluation of c(Treatment, Improved): the meaning of c() is not "combine the two columns into a single vector," but "select both from the data frame." Likewise, columns can be removed using - on column names, which is not possible using standard indexing in matrices or data frames:

```
> subset(Arthritis, Sex == "Female" & Age > 68,
+        select = -c(Age, ID))

   Treatment    Sex Improved
39   Treated Female     None
40   Treated Female     Some
41   Treated Female     Some
84   Placebo Female   Marked
```

2.7 Collapsing tables

{sec:collapsetables}

2.7.1 Collapsing over table factors: aggregate(), margin.table(), and apply()

{sec:collapse}

It sometimes happens that we have a data set with more variables or factors than we want to analyze, or else, having done some initial analyses, we decide that certain factors are not important, and so should be excluded from graphic displays by collapsing (summing) over them. For example, mosaic plots and fourfold displays are often simpler to construct from versions of the data collapsed over the factors that are not shown in the plots.

The appropriate tools to use again depend on the form in which the data are represented—a case-form data frame, a frequency-form data frame (`aggregate()`), or a table-form array or table object (`margin.table()` or `apply()`).

When the data are in frequency form, and we want to produce another frequency data frame, `aggregate()` is a handy tool, using the argument `FUN = sum` to sum the frequency variable over the factors *not* mentioned in the formula. {ex:dayton1}

Example 2.6: Dayton survey

The data frame *DaytonSurvey* in the vcdExtra (Friendly, 2015) package represents a 2^5 table giving the frequencies of reported use ("ever used?") of alcohol, cigarettes, and marijuana in a sample of 2276 high school seniors, also classified by sex and race.

```
> data("DaytonSurvey", package = "vcdExtra")
> str(DaytonSurvey)

'data.frame': 32 obs. of  6 variables:
 $ cigarette: Factor w/ 2 levels "Yes","No": 1 2 1 2 1 2 1 2 1 2 ...
 $ alcohol  : Factor w/ 2 levels "Yes","No": 1 1 2 2 1 1 2 2 1 1 ...
 $ marijuana: Factor w/ 2 levels "Yes","No": 1 1 1 1 2 2 2 2 1 1 ...
 $ sex      : Factor w/ 2 levels "female","male": 1 1 1 1 1 1 1 1 2 2 ...
 $ race     : Factor w/ 2 levels "white","other": 1 1 1 1 1 1 1 1 1 1 ...
 $ Freq     : num  405 13 1 1 268 218 17 117 453 28 ...

> head(DaytonSurvey)

  cigarette alcohol marijuana    sex  race Freq
1       Yes     Yes       Yes female white  405
2        No     Yes       Yes female white   13
3       Yes      No       Yes female white    1
4        No      No       Yes female white    1
5       Yes     Yes        No female white  268
6        No     Yes        No female white  218
```

To focus on the associations among the substances, we want to collapse over sex and race. The right-hand side of the formula used in the call to `aggregate()` gives the factors to be retained in the new frequency data frame, `Dayton_ACM_df`. The left-hand side is the frequency variable (`Freq`), and we aggregate using the `FUN = sum`.

```
> # data in frequency form: collapse over sex and race
> Dayton_ACM_df <- aggregate(Freq ~ cigarette + alcohol + marijuana,
+                            data = DaytonSurvey, FUN = sum)
> Dayton_ACM_df

  cigarette alcohol marijuana Freq
1       Yes     Yes       Yes  911
2        No     Yes       Yes   44
3       Yes      No       Yes    3
```

```
4        No       No        Yes    2
5       Yes      Yes         No  538
6        No      Yes         No  456
7       Yes       No         No   43
8        No       No         No  279
```

△

When the data are in table form, and we want to produce another table, `apply()` with `FUN = sum` can be used in a similar way to sum the table over dimensions not mentioned in the `MARGIN` argument. `margin.table()` is just a wrapper for `apply()` using the `sum()` function.

{ex:dayton2}

EXAMPLE 2.7: Dayton survey

To illustrate, we first convert the *DaytonSurvey* to a 5-way table using `xtabs()`, giving `Dayton_tab`.

```
> # convert to table form
> Dayton_tab <- xtabs(Freq ~ cigarette + alcohol + marijuana + sex + race,
+                     data = DaytonSurvey)
> structable(cigarette + alcohol + marijuana ~ sex + race,
+            data = Dayton_tab)

             cigarette Yes              No
             alcohol   Yes      No      Yes      No
             marijuana Yes  No Yes  No Yes  No Yes  No
sex    race
female white           405 268   1  17  13 218   1 117
       other            23  23   0   1   2  19   0  12
male   white           453 228   1  17  28 201   1 133
       other            30  19   1   8   1  18   0  17
```

Then, use `apply()` on `Dayton_tab` to give the 3-way table `Dayton_ACM_tab` summed over sex and race. The elements in this new table are the column sums for `Dayton.tab` shown by `structable()` just above.

```
> # collapse over sex and race
> Dayton_ACM_tab <- apply(Dayton_tab, MARGIN = 1:3, FUN = sum)
> Dayton_ACM_tab <- margin.table(Dayton_tab, 1:3)   # same result
> structable(cigarette + alcohol ~ marijuana, data = Dayton_ACM_tab)

          cigarette Yes     No
          alcohol   Yes  No Yes  No
marijuana
Yes                 911   3  44   2
No                  538  43 456 279
```

△

(Note that `structable()` would do the collapsing job for us anyway.)

Many of these operations can be performed using the `**ply()` functions in the plyr (Wickham, 2014) package. For example, with the data in a frequency form data frame, use `ddply()` to collapse over unmentioned factors, and `summarise()` as the function to be applied to each piece.

```
> library(plyr)
> Dayton_ACM_df <- ddply(DaytonSurvey, .(cigarette, alcohol, marijuana),
+                        summarise, Freq = sum(Freq))
```

2.7.2 Collapsing table levels: `collapse.table()`

{sec:collapse-levels}

A related problem arises when we have a table or array and for some purpose we want to reduce the number of levels of some factors by summing subsets of the frequencies. For example, we may have initially coded Age in 10-year intervals, and decide that, either for analysis or display purposes, we want to reduce Age to 20-year intervals. The `collapse.table()` function in vcdExtra was designed for this purpose.

{ex:collapse-cat}

EXAMPLE 2.8: Collapsing categories

Create a 3-way table, and collapse Age from 10-year to 20-year intervals and Education from three levels to two. To illustrate, we first generate a $2 \times 6 \times 3$ table of random counts from a Poisson distribution with mean of 100, with factors `sex`, `age`, and `education`.

```
> # create some sample data in frequency form
> set.seed(12345)   # reproducibility
> sex <- c("Male", "Female")
> age <- c("10-19", "20-29",  "30-39", "40-49", "50-59", "60-69")
> education <- c("low", "med", "high")
> dat <- expand.grid(sex = sex, age = age, education = education)
> counts <- rpois(36, 100)   # random Poisson cell frequencies
> dat <- cbind(dat, counts)
> # make it into a 3-way table
> tab1 <- xtabs(counts ~ sex + age + education, data = dat)
> structable(tab1)

                 age 10-19 20-29 30-39 40-49 50-59 60-69
sex    education
Male   low             105    98   123    97    95   105
       med              74   113   114    82    95    85
       high            121   116   104   103    89   100
Female low             107    95   105   116   103    92
       med              96    88    93   118    99   108
       high            120   102    96   103   127    84
```

Now collapse `age` to 20-year intervals, and `education` to 2 levels. In the arguments to `collapse.table()`, levels of `age` and `education` given the same label are summed in the resulting smaller table.

```
> # collapse age to 3 levels, education to 2 levels
> tab2 <- collapse.table(tab1,
+          age = c("10-29", "10-29",  "30-49", "30-49", "50-69", "50-69"),
+          education = c("<high", "<high", "high"))
> structable(tab2)

                 age 10-29 30-49 50-69
sex    education
Male   <high           390   416   380
       high            237   207   189
Female <high           386   432   402
       high            222   199   211
```

△

2.8 Converting among frequency tables and data frames

{sec:convert}

As we've seen, a given contingency table can be represented equivalently in case form, frequency form, and table form. However, some R functions were designed for one particular representation. Table 2.1 gives an overview of some handy tools (with sketched usage) for converting from one form to another, discussed below.

Table 2.1: Tools for converting among different forms for categorical data {tab:convert}

	To this		
From this	Case form	Frequency form	Table form
Case form	—	`Z <- xtabs(~ A+B)` `as.data.frame(Z)`	`table(A, B)`
Frequency form	`expand.dft(X)`	—	`xtabs(count ~ A+B)`
Table form	`expand.dft(X)`	`as.data.frame(X)`	—

2.8.1 Table form to frequency form

A contingency table in table form (an object of class "table") can be converted to a data frame in frequency form with `as.data.frame()`.[6] The resulting data frame contains columns representing the classifying factors and the table entries (as a column named by the `responseName` argument, defaulting to `Freq`). The function `as.data.frame()` is the inverse of `xtabs()`, which converts a data frame to a table.

{ex:GSS-convert}

EXAMPLE 2.9: General social survey

Convert the `GSStab` object in table form to a data.frame in frequency form. By default, the frequency variable is named `Freq`, and the variables `sex` and `party` are made factors.

```
> as.data.frame(GSStab)

     sex party Freq
1 female   dem  279
2   male   dem  165
3 female indep   73
4   male indep   47
5 female   rep  225
6   male   rep  191
```

△

In addition, there are situations where numeric table variables are represented as factors, but you need to convert them to numerics for calculation purposes.

{ex:horse.df}

EXAMPLE 2.10: Death by horse kick

For example, we might want to calculate the weighted mean of `nDeaths` in the *`HorseKicks`* data. Using `as.data.frame()` won't work here, because the variable `nDeaths` becomes a factor.

```
> str(as.data.frame(HorseKicks))

'data.frame': 5 obs. of  2 variables:
 $ nDeaths: Factor w/ 5 levels "0","1","2","3",..: 1 2 3 4 5
 $ Freq   : int  109 65 22 3 1
```

One solution is to use `data.frame()` directly and `as.numeric()` to coerce the table names to numbers.

[6]Because R is object-oriented, this is actually a shorthand for the function `as.data.frame.table()`, which is automatically selected for objects of class "table".

```
> horse.df <- data.frame(nDeaths = as.numeric(names(HorseKicks)),
+                        Freq = as.vector(HorseKicks))
> str(horse.df)

'data.frame': 5 obs. of  2 variables:
 $ nDeaths: num  0 1 2 3 4
 $ Freq   : int  109 65 22 3 1

> horse.df

  nDeaths Freq
1       0  109
2       1   65
3       2   22
4       3    3
5       4    1
```

Then, `weighted.mean()` works as we would like:

```
> weighted.mean(horse.df$nDeaths, weights=horse.df$Freq)

[1] 2
```

△

2.8.2 Case form to table form

Going the other way, we use `table()` to convert from case form to table form. {ex:Arth-convert}

EXAMPLE 2.11: Arthritis treatment

Convert the *Arthritis* data in case form to a 3-way table of `Treatment` × `Sex` × `Improved`. We select the desired columns with their names, but could also use column numbers, e.g., `table(Arthritis[,c(2,3,5)])`.

```
> Art.tab <- table(Arthritis[,c("Treatment", "Sex", "Improved")])
> str(Art.tab)

 'table' int [1:2, 1:2, 1:3] 19 6 10 7 7 5 0 2 6 16 ...
 - attr(*, "dimnames")=List of 3
  ..$ Treatment: chr [1:2] "Placebo" "Treated"
  ..$ Sex      : chr [1:2] "Female" "Male"
  ..$ Improved : chr [1:3] "None" "Some" "Marked"

> ftable(Art.tab)

                 Improved None Some Marked
Treatment Sex
Placebo   Female            19    7      6
          Male              10    0      1
Treated   Female             6    5     16
          Male               7    2      5
```

△

2.8.3 Table form to case form

There may also be times that you will need an equivalent case form data frame with factors representing the table variables rather than the frequency table. For example, the `mca()` function in package MASS (Ripley, 2015) (for multiple correspondence analysis) only operates on data in this format. The function `expand.dft()`[7] in vcdExtra does this, converting a table into a case form. {ex:Arth-convert2}

EXAMPLE 2.12: Arthritis treatment

Convert the *Arthritis* data in table form (`Art.tab`) back to a `data.frame` in case form, with factors `Treatment`, `Sex`, and `Improved`.

```
> library(vcdExtra)
> Art.df <- expand.dft(Art.tab)
> str(Art.df)

'data.frame': 84 obs. of  3 variables:
 $ Treatment: Factor w/ 2 levels "Placebo","Treated": 1 1 1 1 1 1 1 1 1 1 ...
 $ Sex      : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 1 1 1 1 1 ...
 $ Improved : Factor w/ 3 levels "Marked","None",..: 2 2 2 2 2 2 2 2 2 2 ...
```

△

2.8.4 Publishing tables to LaTeX or HTML

OK, you've read your data into R, done some analysis, and now want to include some tables in a LaTeX document or in a web page in HTML format. Formatting tables for these purposes is often tedious and error-prone.

There are a great many packages in R that provide for nicely formatted, publishable tables for a wide variety of purposes; indeed, most of the tables in this book are generated using these tools. See Leifeld (2013) for a description of the texreg (Leifeld, 2014) package and a comparison with some of the other packages.

Here, we simply illustrate the xtable (Dahl, 2014) package, which, along with capabilities for statistical model summaries, time-series data, and so forth, has a `xtable.table` method for one-way and two-way table objects.

The *HorseKicks* data is a small one-way frequency table described in Example 3.4 and contains the frequencies of 0, 1, 2, 3, 4 deaths per corps-year by horse-kick among soldiers in 20 corps in the Prussian army.

```
> data("HorseKicks", package = "vcd")
> HorseKicks

nDeaths
  0   1   2   3   4
109  65  22   3   1
```

By default, `xtable()` formats this in LaTeX as a vertical table, and prints the LaTeX markup to the R console. This output is shown below.

```
> library(xtable)
> xtable(HorseKicks)

% latex table generated in R 3.2.1 by xtable 1.8-0 package
% Wed Nov 18 09:12:23 2015
\begin{table}[ht]
\centering
\begin{tabular}{rr}
  \hline
 & nDeaths \\
  \hline
0 & 109 \\
  1 &  65 \\
```

[7] The original code for this function was provided by Marc Schwarz on the R-Help mailing list.

```
  2 &   22 \\
  3 &    3 \\
  4 &    1 \\
   \hline
\end{tabular}
\end{table}
```

When this is rendered in a LaTeX document, the result of `xtable()` appears as shown in the table below.

```
> xtable(HorseKicks)
```

	nDeaths
0	109
1	65
2	22
3	3
4	1

The table above isn't quite right, because the column label "nDeaths" belongs to the first column, and the second column should be labeled "Freq." To correct that, we convert the *HorseKicks* table to a data frame (see Section 2.8 for details), add the appropriate `colnames`, and use the `print.xtable` method to supply some other options.

```
> tab <- as.data.frame(HorseKicks)
> colnames(tab) <- c("nDeaths", "Freq")
> print(xtable(tab), include.rownames = FALSE,
+        include.colnames = TRUE)
```

nDeaths	Freq
0	109
1	65
2	22
3	3
4	1

There are many more options to control the LaTeX details and polish the appearance of the table; see `help(xtable)` and `vignette("xtableGallery", package = "xtable")`.

Finally, in Chapter 3, we display a number of similar one-way frequency tables in a transposed form to save display space. Table 3.3 is the finished version we show there. The code below uses the following techniques, giving the version shown in Table 2.2: (a) `addmargins()` is used to show the sum of all the frequency values; (b) `t()` transposes the data frame to have 2 rows; (c) `rownames()` assigns the labels we want for the rows; (d) using the `caption` argument provides a table caption, and a numbered table in LaTeX; (e) column alignment (`"r"` or `"l"`) for the table columns is computed as a character string used for the `align` argument.

```
> horsetab <- t(as.data.frame(addmargins(HorseKicks)))
> rownames(horsetab) <- c( "Number of deaths", "Frequency" )
> horsetab <- xtable(horsetab, digits = 0, label="tab:xtable5",
+       caption = "von Bortkiewicz's data on deaths by horse kicks",
+       align = paste0("l|", paste(rep("r", ncol(horsetab)),
+                                       collapse = "")))
+       )
> print(horsetab, include.colnames=FALSE, caption.placement="top")
```

For use in a web page, blog, or Word document, you can use `type="HTML"` in the call to `print()` for "xtable" objects.

Table 2.2: von Bortkiewicz's data on deaths by horse kicks {tab:xtable5}

Number of deaths	0	1	2	3	4	Sum
Frequency	109	65	22	3	1	200

2.9 A complex example: TV viewing data*

{sec:working-complex}

If you have followed so far, congratulations! You are ready for a more complicated example that puts together a variety of the skills developed in this chapter: (a) reading raw data, (b) creating tables, (c) assigning level names to factors and (d) collapsing levels or variables for use in analysis.

For an illustration of these steps, we use the dataset `tv.dat`, supplied with the initial implementation of mosaic displays in R by Jay Emerson. In turn, they were derived from an early, compelling example of mosaic displays (Hartigan and Kleiner, 1984) that illustrated the method with data on a large sample of TV viewers whose behavior had been recorded for the Neilsen ratings. This data set contains sample television audience data from Neilsen Media Research for the week starting November 6, 1995.

The data file, `tv.dat`, is stored in frequency form as a file with 825 rows and 5 columns. There is no header line in the file, so when we use `read.table()` below, the variables will be named `V1` – `V5`. This data represents a 4-way table of size $5 \times 11 \times 5 \times 3 = 825$ where the table variables are `V1` – `V4`, and the cell frequency is read as `V5`.

The table variables are:
`V1`— values 1:5 correspond to the days Monday–Friday;
`V2`— values 1:11 correspond to the quarter-hour times 8:00 pm through 10:30 pm;
`V3`— values 1:5 correspond to ABC, CBS, NBC, Fox, and non-network choices;
`V4`— values 1:3 correspond to transition states: turn the television Off, Switch channels, or Persist in viewing the current channel.

2.9.1 Creating data frames and arrays

The file `tv.dat` is stored in the `doc/extdata` directory of vcdExtra; it can be read as follows:

```
> tv_data <- read.table(system.file("doc", "extdata", "tv.dat",
+                                    package = "vcdExtra"))
> str(tv_data)

'data.frame': 825 obs. of  5 variables:
 $ V1: int  1 2 3 4 5 1 2 3 4 5 ...
 $ V2: int  1 1 1 1 1 2 2 2 2 2 ...
 $ V3: int  1 1 1 1 1 1 1 1 1 1 ...
 $ V4: int  1 1 1 1 1 1 1 1 1 1 ...
 $ V5: int  6 18 6 2 11 6 29 25 17 29 ...

> head(tv_data, 5)

  V1 V2 V3 V4 V5
1  1  1  1  1  6
2  2  1  1  1 18
3  3  1  1  1  6
4  4  1  1  1  2
5  5  1  1  1 11
```

To read such data from a local file, just use `read.table()` in this form:

```
> tv_data <- read.table("C:/R/data/tv.dat")
```

or, to select the path using the file-chooser tool,

```
> tv_data <- read.table(file.choose())
```

We could use this data in frequency form for analysis by renaming the variables, and converting the integer-coded factors `V1` – `V4` to R factors. The lines below use the function `within()` to avoid having to use `TV.dat$Day <- factor(TV.dat$Day)` etc., and only supply labels for the first variable.

```
> TV_df <- tv_data
> colnames(TV_df) <- c("Day", "Time", "Network", "State", "Freq")
> TV_df <- within(TV_df, {
+             Day <- factor(Day,
+                           labels = c("Mon", "Tue", "Wed", "Thu", "Fri"))
+             Time <- factor(Time)
+             Network <- factor(Network)
+             State <- factor(State)
+           })
```

Alternatively, we could just reshape the frequency column (`V5` or `tv_data[,5]`) into a 4-way array. In the lines below, we rely on the facts that (a) the table is complete—there are no missing cells, so `nrow(tv_data)` = 825; (b) the observations are ordered so that `V1` varies most rapidly and `V4` most slowly. From this, we can just extract the frequency column and reshape it into an array using the `dim` argument. The level names are assigned to `dimnames(TV)` and the variable names to `names(dimnames(TV))`.

```
> TV <- array(tv_data[,5], dim = c(5, 11, 5, 3))
> dimnames(TV) <-
+     list(c("Mon", "Tue", "Wed", "Thu", "Fri"),
+          c("8:00", "8:15", "8:30", "8:45", "9:00", "9:15",
+            "9:30", "9:45", "10:00", "10:15", "10:30"),
+          c("ABC", "CBS", "NBC", "Fox", "Other"),
+          c("Off", "Switch", "Persist"))
> names(dimnames(TV)) <- c("Day", "Time", "Network", "State")
```

More generally (even if there are missing cells), we can use `xtabs()` to do the cross-tabulation, using `V5` as the frequency variable. Here's how to do this same operation with `xtabs()`:

```
> TV <- xtabs(V5 ~ ., data = tv_data)
> dimnames(TV) <-
+     list(Day = c("Mon", "Tue", "Wed", "Thu", "Fri"),
+          Time = c("8:00", "8:15", "8:30", "8:45", "9:00", "9:15",
+                   "9:30", "9:45", "10:00", "10:15", "10:30"),
+          Network = c("ABC", "CBS", "NBC", "Fox", "Other"),
+          State = c("Off", "Switch", "Persist"))
```

Note that in the lines above, the variable names are assigned directly as the names of the elements in the `dimnames` list.

2.9.2 Subsetting and collapsing

For many purposes, the 4-way table `TV` is too large and awkward to work with. Among the networks, Fox and Other occur infrequently, so we will remove them. We can also cut it down to a 3-way table by considering only viewers who persist with the current station.[8]

[8]This relies on the fact that indexing an array drops dimensions of length 1 by default, using the argument `drop = TRUE`; the result is coerced to the lowest possible dimension.

```
> TV <- TV[,,1:3,]      # keep only ABC, CBS, NBC
> TV <- TV[,,,3]        # keep only Persist -- now a 3 way table
> structable(TV)

            Time 8:00 8:15 8:30 8:45 9:00 9:15 9:30 9:45 10:00 10:15 10:30
Day Network
Mon ABC           146  151  156   83  325  350  386  340   352   280   278
    CBS           337  293  304  233  311  251  241  164   252   265   272
    NBC           263  219  236  140  226  235  239  246   279   263   283
Tue ABC           244  181  231  205  385  283  345  192   329   351   364
    CBS           173  180  184  109  218  235  256  250   274   263   261
    NBC           315  254  280  241  370  214  195  111   188   190   210
Wed ABC           233  161  194  156  339  264  279  140   237   228   203
    CBS           158  126  207   59   98  103  122   86   109   105   110
    NBC           134  146  166   66  194  230  264  143   274   289   306
Thu ABC           174  183  197  181  187  198  211   86   110   122   117
    CBS           196  185  195  104  106  116  116   47   102    84    84
    NBC           515  463  472  477  590  473  446  349   649   705   747
Fri ABC           294  281  305  239  278  246  245  138   246   232   233
    CBS           130  144  154   81  129  153  136  126   138   136   152
    NBC           195  220  248  160  172  164  169   85   183   198   204
```

Finally, for some purposes, we might also want to collapse the 11 `Time`'s into a smaller number. Here, we use `collapse.table()` (see Section 2.7.2), which was designed for this purpose.

```
> TV2 <- collapse.table(TV,
+                       Time = c(rep("8:00-8:59", 4),
+                                rep("9:00-9:59", 4),
+                                rep("10:00-10:44", 3)))
> structable(Day ~ Time + Network, TV2)

                    Day  Mon  Tue  Wed  Thu  Fri
Time        Network
8:00-8:59   ABC          536  861  744  735 1119
            CBS         1167  646  550  680  509
            NBC          858 1090  512 1927  823
9:00-9:59   ABC         1401 1205 1022  682  907
            CBS          967  959  409  385  544
            NBC          946  890  831 1858  590
10:00-10:44 ABC          910 1044  668  349  711
            CBS          789  798  324  270  426
            NBC          825  588  869 2101  585
```

Congratulations! If you followed the operations described above, you are ready for the material described in the rest of the book. If not, try working through some of exercises below.

2.10 Lab exercises

{sec:ch02-exercises} {lab:2.1}

Exercise 2.1 The packages vcd and vcdExtra contain many data sets with some examples of analysis and graphical display. The goal of this exercise is to familiarize yourself with these resources.

You can get a brief summary of these using the function `datasets()` from vcdExtra. Use the following to get a list of these with some characteristics and titles.

```
> ds <- datasets(package = c("vcd", "vcdExtra"))
> str(ds, vec.len = 2)

'data.frame': 74 obs. of  5 variables:
 $ Package: chr  "vcd" "vcd" ...
 $ Item   : chr  "Arthritis" "Baseball" ...
```

```
 $ class  : chr  "data.frame" "data.frame" ...
 $ dim    : chr  "84x5" "322x25" ...
 $ Title  : chr  "Arthritis Treatment Data" "Baseball Data" ...
```

(a) How many data sets are there altogether? How many are there in each package?
(b) Make a tabular display of the frequencies by `Package` and `class`.
(c) Choose one or two data sets from this list, and examine their help files (e.g., `help(Arthritis)` or `?Arthritis`). You can use, e.g., `example(Arthritis)` to run the R code for a given example.

{lab:2.2}

Exercise 2.2 For each of the following data sets in the vcdExtra package, identify which are response variable(s) and which are explanatory. For factor variables, which are unordered (nominal) and which should be treated as ordered? Write a sentence or two describing substantitive questions of interest for analysis of the data. (*Hint*: use `data(foo, package="vcdExtra")` to load, and `str(foo)`, `help(foo)` to examine data set `foo`.)

(a) Abortion opinion data: *Abortion*
(b) Caesarian Births: *Caesar*
(c) Dayton Survey: *DaytonSurvey*
(d) Minnesota High School Graduates: *Hoyt*

{lab:2.3}

Exercise 2.3 The data set *UCBAdmissions* is a 3-way table of frequencies classified by `Admit`, `Gender`, and `Dept`.

(a) Find the total number of cases contained in this table.
(b) For each department, find the total number of applicants.
(c) For each department, find the overall proportion of applicants who were admitted.
(d) Construct a tabular display of department (rows) and gender (columns), showing the proportion of applicants in each cell who were admitted relative to the total applicants in that cell.

{lab:2.4}

Exercise 2.4 The data set *DanishWelfare* in vcd gives a 4-way, $3 \times 4 \times 3 \times 5$ table as a data frame in frequency form, containing the variable `Freq` and four factors, `Alcohol`, `Income`, `Status`, and `Urban`. The variable `Alcohol` can be considered as the response variable, and the others as possible predictors.

(a) Find the total number of cases represented in this table.
(b) In this form, the variables `Alcohol` and `Income` should arguably be considered *ordered* factors. Change them to make them ordered.
(c) Convert this data frame to table form, `DanishWelfare.tab`, a 4-way array containing the frequencies with appropriate variable names and level names.
(d) The variable `Urban` has 5 categories. Find the total frequencies in each of these. How would you collapse the table to have only two categories, `City`, `Non-city`?
(e) Use `structable()` or `ftable()` to produce a pleasing flattened display of the frequencies in the 4-way table. Choose the variables used as row and column variables to make it easier to compare levels of `Alcohol` across the other factors.

{lab:2.5}

Exercise 2.5 The data set *UKSoccer* in vcd gives the distributions of number of goals scored by the 20 teams in the 1995/96 season of the Premier League of the UK Football Association.

```
> data("UKSoccer", package = "vcd")
> ftable(UKSoccer)
```

```
     Away  0  1  2  3  4
Home
0         27 29 10  8  2
1         59 53 14 12  4
2         28 32 14 12  4
3         19 14  7  4  1
4          7  8 10  2  0
```

This two-way table classifies all $20 \times 19 = 380$ games by the joint outcome (Home, Away), the number of goals scored by the `Home` and `Away` teams. The value `4` in this table actually represents 4 or more goals.

(a) Verify that the total number of games represented in this table is 380.
(b) Find the marginal total of the number of goals scored by each of the home and away teams.
(c) Express each of the marginal totals as proportions.
(d) Comment on the distribution of the numbers of home-team and away-team goals. Is there any evidence that home teams score more goals on average?

{lab:2.6}

Exercise 2.6 The one-way frequency table *Saxony* in vcd records the frequencies of families with 0, 1, 2, ... 12 male children, among 6115 families with 12 children. This data set is used extensively in Chapter 3.

```
> data("Saxony", package = "vcd")
> Saxony

nMales
   0    1    2    3    4    5    6    7    8    9   10   11   12
   3   24  104  286  670 1033 1343 1112  829  478  181   45    7
```

Another data set, *Geissler*, in the vcdExtra package, gives the complete tabulation of all combinations of `boys` and `girls` in families with a given total number of children (`size`). The task here is to create an equivalent table, `Saxony12` from the *Geissler* data.

```
> data("Geissler", package = "vcdExtra")
> str(Geissler)

'data.frame': 90 obs. of  4 variables:
 $ boys : int  0 0 0 0 0 0 0 0 0 0 ...
 $ girls: num  1 2 3 4 5 6 7 8 9 10 ...
 $ size : num  1 2 3 4 5 6 7 8 9 10 ...
 $ Freq : int  108719 42860 17395 7004 2839 1096 436 161 66 30 ...
```

(a) Use `subset()` to create a data frame, `sax12` containing the *Geissler* observations in families with `size==12`.
(b) Select the columns for `boys` and `Freq`.
(c) Use `xtabs()` with a formula, `Freq ~ boys`, to create the one-way table.
(d) Do the same steps again to create a one-way table, `Saxony11`, containing similar frequencies for families of `size==11`.

{lab:2.7}

Exercise 2.7 * *Interactive coding of table factors*: Some statistical and graphical methods for contingency tables are implemented only for two-way tables, but can be extended to 3+-way tables by recoding the factors to interactive combinations along the rows and/or columns, in a way similar to what `ftable()` and `structable()` do for printed displays.

For the *UCBAdmissions* data, produce a two-way table object, `UCB.tab2`, that has the combinations of `Admit` and `Gender` as the rows, and `Dept` as its columns, to look like the result below:

```
                     Dept
Admit:Gender          A   B   C   D   E   F
  Admitted:Female    89  17 202 131  94  24
  Admitted:Male     512 353 120 138  53  22
  Rejected:Female    19   8 391 244 299 317
  Rejected:Male     313 207 205 279 138 351
```

(a) Try this the long way: convert *UCBAdmissions* to a data frame (`as.data.frame()`), manipulate the factors (e.g., `interaction()`), then convert back to a table (`as.data.frame()`).
(b) Try this the short way: both `ftable()` and `structable()` have `as.matrix()` methods that convert their result to a matrix.

{lab:2.8}

Exercise 2.8 The data set *VisualAcuity* in vcd gives a $4 \times 4 \times 2$ table as a frequency data frame.

```
> data("VisualAcuity", package = "vcd")
> str(VisualAcuity)

'data.frame': 32 obs. of  4 variables:
 $ Freq  : num  1520 234 117 36 266 ...
 $ right : Factor w/ 4 levels "1","2","3","4": 1 2 3 4 1 2 3 4 1 2 ...
 $ left  : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 2 2 2 2 3 3 ...
 $ gender: Factor w/ 2 levels "male","female": 2 2 2 2 2 2 2 2 2 2 ...
```

(a) From this, use `xtabs()` to create two 4×4 frequency tables, one for each gender.
(b) Use `structable()` to create a nicely organized tabular display.
(c) Use `xtable()` to create a LaTeX or HTML table.

References

Agresti, A. (2002). *Categorical Data Analysis*. Wiley Series in Probability and Statistics. New York: Wiley-Interscience [John Wiley & Sons], 2nd edn.

Dahl, D. B. (2014). *xtable: Export tables to LaTeX or HTML.* R package version 1.7-4.

Dalgaard, P. (2008). *Introductory Statistics with R.* Springer, 2nd edn.

Dragulescu, A. A. (2014). *xlsx: Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files*. R package version 0.5.7.

Fox, J. and Weisberg, S. (2011). *An R Companion to Applied Regression*. Thousand Oaks CA: SAGE Publications, 2nd edn.

Friendly, M. (2015). *vcdExtra: vcd Extensions and Additions*. R package version 0.6-7.

Hartigan, J. A. and Kleiner, B. (1984). A mosaic of television ratings. *The American Statistician*, 38, 32–35.

Koch, G. and Edwards, S. (1988). Clinical efficiency trials with categorical data. In K. E. Peace, ed., *Biopharmaceutical Statistics for Drug Development*, (pp. 403–451). New York: Marcel Dekker.

Leifeld, P. (2013). texreg: Conversion of statistical model output in R to LaTeX and HTML tables. *Journal of Statistical Software*, 55(8), 1–24.

Leifeld, P. (2014). *texreg: Conversion of R regression output to LaTeX or HTML tables.* R package version 1.34.

Maindonald, J. and Braun, J. (2007). *Data Analysis and Graphics Using R*. Cambridge: Cambridge University Press, 2nd edn.

Meyer, D., Zeileis, A., and Hornik, K. (2015). *vcd: Visualizing Categorical Data*. R package version 1.3-3.

Mirai Solutions GmbH (2015). *XLConnect: Excel Connector for R*. R package version 0.2-11.

R Core Team (2015). *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ...* R package version 0.8-63.

Ripley, B. (2015). *MASS: Support Functions and Datasets for Venables and Ripley's MASS*. R package version 7.3-40.

Warnes, G. R., Bolker, B., Gorjanc, G., Grothendieck, G., Korosec, A., Lumley, T., MacQueen, D., Magnusson, A., Rogers, J., and others (2014). *gdata: Various R programming tools for data manipulation*. R package version 2.13.3.

Wickham, H. (2014). *plyr: Tools for splitting, applying and combining data*. R package version 1.8.1.