```
> Womenlf$working.alt.2 <- factor(with(Womenlf,
+     ifelse(partic %in% c("parttime", "fulltime"), "yes", "no")))
> with(Womenlf, all.equal(working, working.alt.2))

[1] TRUE

> Womenlf$fulltime.alt <- factor(with(Womenlf,
+     ifelse(partic == "fulltime", "yes",
+         ifelse(partic == "parttime", "no", NA))))
> with(Womenlf, all.equal(fulltime, fulltime.alt))

[1] TRUE
```

The first argument to `ifelse` is a logical vector, containing the values `TRUE` and `FALSE`; the second argument gives the value assigned where the first argument is `TRUE`; and the third argument gives the value assigned when the first argument is `FALSE`. We used cascading `ifelse` commands to create the variable `fulltime.alt`, assigning the value `"yes"` to those working `"fulltime"`, `"no"` to those working `"parttime"`, and `NA` otherwise (i.e., where `partic` takes on the value `"not.work"`).

We employed the *matching operator*, `%in%`, which returns a logical vector containing `TRUE` if a value in the vector before `%in%` is a member of the vector after the symbol and `FALSE` otherwise. See `help("%in%")` for more information; the quotes are required because of the `%` character. We also used the function `all.equal` to test the equality of the alternative recodings. When applied to numeric variables, `all.equal` tests for *approximate* equality, within the precision of floating-point computations (discussed in Section 2.6.2); more generally, `all.equal` not only reports whether two objects are approximately equal but, if they are not equal, provides information on how they differ.

An alternative to the first test is `all(working == working.alt.2)`, but this approach won't work properly in the second test because of the missing data:

```
> with(Womenlf, all(fulltime == fulltime.alt))

[1] NA
```

We once more clean up before proceeding by removing the copy of `Womenlf` that was made in the working data:

```
> remove(Womenlf)
```

2.3 Matrices, Arrays, and Lists

We have thus far encountered and used several data structures in R:

- *Vectors:* One-dimensional arrays of numbers, character strings, or logical values. Single numbers, character strings, and logical values in R are treated as vectors of length one.

- *Factors:* One-dimensional arrays of levels.
- *Data frames:* Two-dimensional data tables, with the rows defining observations and the columns defining variables. Data frames are heterogeneous, in the sense that some columns may be numeric and others factors and some may even contain character data or logical data.

In this section, we describe three other common data structures in R: *matrices*, *arrays*, and *lists*.

2.3.1 MATRICES

A matrix in R is a two-dimensional array of elements all of which are of the same *mode*—for example, real numbers, integers, character strings, or logical values. Matrices can be constructed using the `matrix` function, which reshapes its first argument into a matrix with the specified number of rows (the second argument) and columns (the third argument): for example,

```
> (A <- matrix(1:12, nrow=3, ncol=4))

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

>(B <- matrix(c("a", "b", "c"), 4, 3, byrow=TRUE))# 4 rows,3 columns

     [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"
```

A matrix is filled by columns, unless the optional argument `byrow` is set to `TRUE`. The second example illustrates that if there are fewer elements in the first argument than are required, then the elements are simply recycled, extended by repetition to the required length.

A defining characteristic of a matrix is that it has a `dim` (dimension) *attribute* with two elements: the number of rows and the number of columns:[16]

```
> dim(A)

[1] 3 4

> dim(B)

[1] 4 3
```

As we have seen before, a vector is a one-dimensional array of numbers. For example, here is a vector containing a random permutation of the first 10 integers:

[16] More correctly, a matrix is a vector with a two-element `dim` attribute.

```
> set.seed(54321) # for reproducibility
> (v <- sample(10, 10))  # permutation of 1 to 10

 [1]  5 10  2  8  7  9  1  4  3  6
```

A vector has a `length` attribute but not a `dim` attribute:

```
> length(v)

[1] 10

> dim(v)

NULL
```

R often treats vectors differently than matrices. You can turn a vector into a one-column matrix using the `as.matrix` *coercion* function:

```
> as.matrix(v)

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

R includes extensive facilities for matrix computation, some of which are described later in this chapter and in Section 8.2.

2.3.2 ARRAYS

Higher-dimensional arrays of homogeneous elements are encountered much less frequently than matrices. If needed, higher-dimensional arrays may be created with the `array` function; here is an example generating a three-dimensional array:

```
> (array.3 <- array(1:24, dim=c(4, 3, 2))) # 4 rows, 3 columns, 2 layers

, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

The order of the dimensions is row, column, and layer. The array is filled with the index of the first dimension changing most quickly—that is, row, then column, then layer.

2.3.3 LISTS

Lists are data structures composed of potentially heterogeneous elements. The elements of a list may be complex data structures, including other lists. Because a list can contain other lists as elements, each of which can also contain lists, lists are *recursive* structures. In contrast, the elements of an ordinary vector—such as an individual number, character string, or logical value—are *atomic* objects.

Here is an example of a list, constructed with the `list` function:

```
> (list.1 <- list(mat.1=A, mat.2=B, vec=v))  # a 3-item list

$mat.1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

$mat.2
     [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"

$vec
 [1]  5 10  2  8  7  9  1  4  3  6
```

This list contains a numeric matrix, a character matrix, and a numeric vector. We named the arguments in the call to the `list` function; these are arbitrary names that we chose, not standard arguments to `list`. The argument names supplied became the names of the list elements.

Because lists permit us to collect related information regardless of its form, they provide the foundation for the class-based S3 object system in R.[17] Data frames, for example, are lists with some special properties that permit them to behave somewhat like matrices.

2.3.4 INDEXING

A common operation in R is to extract some of the elements of a vector, matrix, array, list, or data frame by supplying the indices of the elements to be extracted. Indices are specified between square brackets, "`[`" and "`]`". We have already used this syntax on several occasions, and it is now time to consider indexing more systematically.

[17] Classes are described in Sections 2.6 and 8.7.

INDEXING VECTORS

A vector can be indexed by a single number or by a vector of numbers; indeed, indices may be specified out of order, and an index may be repeated to extract the corresponding element more than once:

```
> v

 [1]  5 10  2  8  7  9  1  4  3  6

> v[2]

[1] 10

> v[c(4, 2, 6)]

[1]  8 10  9

> v[c(4, 2, 4)]

[1]  8 10  8
```

Specifying *negative* indices suppresses the corresponding elements of the vector:

```
> v[-c(2, 4, 6, 8, 10)]

[1] 5 2 7 1 3
```

If a vector has a `names` attribute, then we can also index the elements by name:[18]

```
> names(v) <- letters[1:10]
> v

 a  b  c  d  e  f  g  h  i  j
 5 10  2  8  7  9  1  4  3  6

> v[c("f", "i", "g")]

f i g
9 3 1
```

Finally, a vector may be indexed by a logical vector of the same length, retaining the elements corresponding to `TRUE` and omitting those corresponding to `FALSE`:

```
> v < 6

    a     b     c     d     e     f     g     h     i     j
 TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE

> v[v < 6]  # all entries less than 6
```

[18] The vector `letters` contains the 26 lowercase letters from `"a"` to `"z"`; `LETTERS` similarly contains the uppercase letters.

```
a c g h i
5 2 1 4 3
```

Any of these forms of indexing may be used on the left-hand side of the assignment operator to replace the elements of a vector—an unusual and convenient feature of the R language: for example,

```
> (vv <- v)  # make copy of v

 a  b  c  d  e  f  g  h  i  j
 5 10  2  8  7  9  1  4  3  6

> vv[c(1, 3, 5)] <- 1:3
> vv

 a  b  c  d  e  f  g  h  i  j
 1 10  2  8  3  9  1  4  3  6

> vv[c("b", "d", "f", "h", "j")] <- NA
> vv

 a  b  c  d  e  f  g  h  i  j
 1 NA  2 NA  3 NA  1 NA  3 NA

> remove(vv)
```

INDEXING MATRICES AND ARRAYS

Indexing extends straightforwardly to matrices and to higher-dimensional arrays. Indices corresponding to the different dimensions of an array are separated by commas; if the index for a dimension is left unspecified, then all the elements along that dimension are selected. We demonstrate with the matrix A:

```
> A

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> A[2, 3]  # element in row 2, column 3

[1] 8

> A[c(1, 2), 2]  # rows 1 and 2, column 2

[1] 4 5

> A[c(1, 2), c(2, 3)] # rows 1 and 2, columns 2 and 3

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

```
> A[c(1, 2), ]  # rows 1 and 2, all columns

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
```

The second example above, `A[2, 3]`, returns a single-element vector rather than a 1×1 matrix; likewise, the third example, `A[c(1, 2), 2]`, returns a vector with two elements rather than a 2×1 matrix. *More generally, in indexing a matrix or array, dimensions of extent one are automatically dropped.* In particular, if we select elements in a single row or single column of a matrix, then the result is a vector, not a matrix with a single row or column, a convention that will occasionally give an R programmer headaches. We can override this default behavior with the argument `drop=FALSE`:

```
> A[ , 1]

[1] 1 2 3

> A[ , 1, drop=FALSE]

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

In both of these examples, the row index is missing and is therefore taken to be all rows of the matrix.

Negative indices, row or column names (if they are defined), and logical vectors of the appropriate length may also be used to index a matrix or a higher-dimensional array:

```
> A[ , -c(1, 3)]  # omit columns 1 and 3

     [,1] [,2]
[1,]    4   10
[2,]    5   11
[3,]    6   12

> A[-1, -2]     # omit row 1 and column 2

     [,1] [,2] [,3]
[1,]    2    8   11
[2,]    3    9   12

> rownames(A) <- c("one", "two", "three")  # set row names
> colnames(A) <- c("w", "x", "y", "z")     # set column names
> A

      w x y  z
one   1 4 7 10
two   2 5 8 11
three 3 6 9 12
```

```
> A[c("one", "two"), c("x", "y")]

    x y
one 4 7
two 5 8

> A[c(TRUE, FALSE, TRUE), ]

      w x y  z
one   1 4 7 10
three 3 6 9 12
```

Used on the left of the assignment arrow, we may replace indexed elements in a matrix or array:

```
> (AA <- A)  # make a copy of A

      w x y  z
one   1 4 7 10
two   2 5 8 11
three 3 6 9 12

> AA[1, ] <- 0  # set first row to zeros
> AA

      w x y  z
one   0 0 0  0
two   2 5 8 11
three 3 6 9 12

> remove(AA)
```

INDEXING LISTS

Lists may be indexed in much the same way as vectors, but some special considerations apply. Recall the list that we constructed earlier:

```
> list.1

$mat.1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

$mat.2
     [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"

$vec
 [1]  5 10  2  8  7  9  1  4  3  6
```

```
> list.1[c(2, 3)]  # elements 2 and 3

$mat.2
     [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"

$vec
 [1]  5 10  2  8  7  9  1  4  3  6

> list.1[2]  # returns a one-element list

$mat.2
     [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"
```

Even when we select a single element of the list, as in the last example, we get a *single-element list* rather than (in this case) a matrix. To extract the matrix in position 2 of the list, we can use double-bracket notation:

```
> list.1[[2]]  # returns a matrix

     [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"
```

The distinction between a one-element list and the element itself is subtle but important, and it can trip us up if we are not careful.

If the list elements are named, then we can use the names in indexing the list:

```
> list.1["mat.1"]  # produces a one-element list

$mat.1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> list.1[["mat.1"]]  #  extracts a single element

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

An element name may also be used—either quoted or, if it is a legal R name, unquoted—after the $ (dollar sign) to extract a list element:

```
> list.1$mat.1
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Used on the left-hand side of the assignment arrow, dollar-sign indexing allows us to replace list elements, define new elements, or delete an element (by assigning NULL to the element):

```
> list.1$mat.1 <- matrix(1, 2, 2)       # replace element
> list.1$title <- "an arbitrary list"   # new element
> list.1$mat.2 <- NULL                  # delete element
> list.1

$mat.1
     [,1] [,2]
[1,]    1    1
[2,]    1    1

$vec
 [1]  5 10  2  8  7  9  1  4  3  6

$title
[1] "an arbitrary list"
```

Setting a list element to NULL is trickier:

```
> list.1["title"] <- list(NULL)
> list.1

$mat.1
     [,1] [,2]
[1,]    1    1
[2,]    1    1

$vec
 [1]  5 10  2  8  7  9  1  4  3  6

$title
NULL
```

Once a list element is extracted, it may itself be indexed: for example,

```
> list.1$vec[3]

[1] 2

> list.1[["vec"]][3:5]

[1] 2 8 7
```

Finally, extracting a nonexistent element returns NULL:

```
> list.1$foo

NULL
```

This behavior is potentially confusing because it is not possible to distinguish by the value returned between a NULL element, such as `list.1$title` in the example, and a nonexistent element, such as `list.1$foo`.

INDEXING DATA FRAMES

Data frames may be indexed either as lists or as matrices. Recall the `Guyer` data frame:

```
> head(Guyer)  # first 6 rows

  cooperation condition    sex perc.coop      coop.4 coop.groups
1  -0.3708596    public   male  40.83333 (33.3,44.2]         med
2   0.1335314    public   male  53.33333   (44.2,55]        high
3  -0.8079227    public   male  30.83333 (22.5,33.3]         low
4  -0.2682640    public   male  43.33333 (33.3,44.2]         med
5   0.2682640    public   male  56.66667   (55,65.9]        high
6  -0.2006707    public female  45.00000   (44.2,55]        high
  coop.2
1      1
2      2
3      1
4      1
5      2
6      1
```

Because no row names were specified when we entered the data, the row names are simply the character representation of the row numbers. Indexing `Guyer` as a matrix:

```
> Guyer[ , 1]  # first column

 [1] -0.37085958  0.13353139 -0.80792270 -0.26826399  0.26826399
 [6] -0.20067070  0.03333642  0.65587579  0.13353139 -1.14356368
[11] -1.23676263 -0.06669137 -0.26826399 -0.65587579 -1.09861229
[16] -0.69314718 -0.73088751 -0.54654371 -0.92798677 -0.54654371

> Guyer[ , "cooperation"]  # equivalent

 [1] -0.37085958  0.13353139 -0.80792270 -0.26826399  0.26826399
 [6] -0.20067070  0.03333642  0.65587579  0.13353139 -1.14356368
[11] -1.23676263 -0.06669137 -0.26826399 -0.65587579 -1.09861229
[16] -0.69314718 -0.73088751 -0.54654371 -0.92798677 -0.54654371

> Guyer[c(1, 2), ]  # rows 1 and 2

  cooperation condition  sex perc.coop      coop.4 coop.groups
1  -0.3708596    public male  40.83333 (33.3,44.2]         med
2   0.1335314    public male  53.33333   (44.2,55]        high
  coop.2
1      1
2      2

> Guyer[c("1", "2"), "cooperation"] # by row and column names

[1] -0.3708596  0.1335314

> Guyer[-(6:20), ]  # drop rows 6 through 20

  cooperation condition  sex perc.coop      coop.4 coop.groups
1  -0.3708596    public male  40.83333 (33.3,44.2]         med
2   0.1335314    public male  53.33333   (44.2,55]        high
3  -0.8079227    public male  30.83333 (22.5,33.3]         low
```

```
4  -0.2682640    public male  43.33333 (33.3,44.2]         med
5   0.2682640    public male  56.66667   (55,65.9]        high
  coop.2
1      1
2      2
3      1
4      1
5      2
```

```
> with(Guyer, Guyer[sex == "female" & condition == "public", ])
```

```
   cooperation condition    sex perc.coop      coop.4 coop.groups
6  -0.20067070    public female  45.00000   (44.2,55]        high
7   0.03333642    public female  50.83333   (44.2,55]        high
8   0.65587579    public female  65.83333   (55,65.9]        high
9   0.13353139    public female  53.33333   (44.2,55]        high
10 -1.14356368    public female  24.16667 (22.5,33.3]         low
   coop.2
6       1
7       2
8       2
9       2
10      1
```

We require `with` in the last example to access the variables `sex` and `condition`, because the data frame `Guyer` is not attached to the search path. More conveniently, we can use the `subset` function to perform this operation:

```
> subset(Guyer, sex == "female" & condition == "public")
```

```
   cooperation condition    sex perc.coop      coop.4 coop.groups
6  -0.20067070    public female  45.00000   (44.2,55]        high
7   0.03333642    public female  50.83333   (44.2,55]        high
8   0.65587579    public female  65.83333   (55,65.9]        high
9   0.13353139    public female  53.33333   (44.2,55]        high
10 -1.14356368    public female  24.16667 (22.5,33.3]         low
   coop.2
6       1
7       2
8       2
9       2
10      1
```

Alternatively, indexing the data frame `Guyer` as a list:

```
> Guyer$cooperation
```

```
 [1] -0.37085958  0.13353139 -0.80792270 -0.26826399  0.26826399
 [6] -0.20067070  0.03333642  0.65587579  0.13353139 -1.14356368
[11] -1.23676263 -0.06669137 -0.26826399 -0.65587579 -1.09861229
[16] -0.69314718 -0.73088751 -0.54654371 -0.92798677 -0.54654371
```

```
> Guyer[["cooperation"]]
```

```
 [1] -0.37085958  0.13353139 -0.80792270 -0.26826399  0.26826399
 [6] -0.20067070  0.03333642  0.65587579  0.13353139 -1.14356368
[11] -1.23676263 -0.06669137 -0.26826399 -0.65587579 -1.09861229
[16] -0.69314718 -0.73088751 -0.54654371 -0.92798677 -0.54654371
```