CHAPTER 5

Classes of Objects

In R, objects belong to classes as well as modes and types. Classes tell something about how an object is structured. S3 and S4 differ with regard to classes. In S3, there are specific classes into which an R object falls. In S4, the user defines a class for an S4 object. Classes in S3 are called informal classes, whereas classes in S4 are called formal classes. This chapter covers both kinds of classes.

Some Basics on Classes

S3 classes are attributes of S3 objects and are not usually assigned by the user. Given an object, the class of the object can be found by using the function `class()`. If an object has not been given a class in the package to which the object belongs, then the class of the object is just the mode of the object. For example, an object of mode `function` is also of class `function`.

The output from many functions will have a class attribute specific to the function. For example, the class of the output from a linear model fit with the function `lm()` is `lm`. Also, objects can belong to more than one class. An example is a model fit using the generalized linear model function `glm()`. The classes of the output are `glm` and `lm`.

On a more technical side, according to the help page for `class()`, the classes of an object are the classes from which an object inherits. So, the output of `lm()` inherits from `lm`, and the output from `glm()` inherits from both `lm` and `glm`.

M. Tollefson, *R Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-4405-0_5

One useful function for classes is the function `methods()`. Entering **methods(class=*name*)**, where *name* is the name of a class, will show functions specifically written to be applied to objects of the class. For example:

```
> methods(class=lm)
 [1] add1           alias          anova            case.names
 [5] coerce         confint        cooks.distance   deviance
 [9] dfbeta         dfbetas        drop1            dummy.coef
[13] effects        extractAIC     family           formula
[17] hatvalues      influence      initialize       kappa
[21] labels         logLik         model.frame      model.matrix
[25] nobs           plot           predict          print
[29] proj           qr             residuals        rstandard
[33] rstudent       show           simulate         slotsFromS3
[37] summary         variable.names vcov
see '?methods' for accessing help and source code
```

S4 (formal) classes are the starting point for S4 methods. An S4 class contains a user-defined name for the class and the variables to be used by methods associated with the class, along with the classes of the variables.

Entering **?class** at the R prompt or using the R Studio "Help" tab gives more information about S3 and S4 classes and inheritance.

Vectors

Although there is no class `vector`, the vector merits discussion as one of the most basic kinds of objects. For atomic mode vectors, a vector is a collection of elements of only one dimension. The class is just the mode of the vector, except for integer vectors, which take on the class `integer`. Another reason vectors are important is that for the `as.name()` functions, where *name* is the *name* of an atomic mode, except for the mode `NULL`, `as.name()` returns a vector.

The functions `vector()`, `as.vector()`, and `is.vector()` exist and operate somewhat like the similar functions for the modes. The function `vector()` takes the arguments **mode** and **length** and creates a vector of the given mode and length. The acceptable modes are the atomic modes—except `NULL`, the `list` mode, and the `expression` mode. Other modes give an error.

For the atomic modes,

```
vector( mode="name", length=n )
```

behaves the same way as

```
name( length=n ),
```

where ***name*** is the name of the mode and **n** is the length argument. Note that ***name*** must be in quotes in the call to `vector()`. For the `list` mode, `vector()` returns a list of **NULL**s of length given by the length argument. With the mode set equal to **expression**, `vector()` gives an expression with **NULL**s for arguments, where the number of **NULL**s is given by the length argument.

The function `as.vector()` tries to coerce an object to a vector. For some objects, `as.vector()` just passes the object through and does not create a vector. For some other objects, an error is returned if the function `as.vector()` is run.

For matrices and arrays, dimensional information is removed by `as.vector()` (for example, names of columns in a matrix and the number of rows and columns), and a vector of the elements of the matrix or array is returned. The elements of the vector are ordered starting with the first dimension of the matrix or array and continuing through the dimensions. For example:

```
> a=array( 1:8, c( 2, 2, 2 ) )

> dimnames( a )=list( c( "a", "b" ), c( "m", "n" ), c( "y", "z" ) )
>
```

```
> a
, , y

  m n
a 1 3
b 2 4

, , z

  m n
a 5 7
b 6 8
> as.vector( a )
[1] 1 2 3 4 5 6 7 8
```

Here, the `c()` function is used to create the vector of the dimensions for the 2x2x2 `array()` and to create names for the three dimensions of the array.

For objects of mode `list`, `as.vector()` passes the list through. Depending on the structure of the list, `is.vector()` operating on the result can give either **TRUE** or **FALSE**. The mode does not change.

For objects of mode `function`, `as.vector()` returns an error.

For objects of mode `call`, `as.vector()` passes the object through but does not create a vector. The mode does not change.

For objects of mode `environment`, `as.vector()` returns an error.

For objects of mode `expression`, `as.vector()` passes the expression through, and the result gives **TRUE** for `is.vector()`. The mode does not change.

For the `S4` mode, `as.vector()` returns an error.

The function `is.vector()` returns **TRUE** if the object is a vector and **FALSE** otherwise, although some objects that do not look like vectors return **TRUE**.

More information about `vector()`, `as.vector()`, and `is.vector()` can be found by entering **?vector** at the R prompt or by using the R Studio Help tab.

Some Common S3 Classes

Some common S3 classes are `integer`, `numeric`, `matrix`, and `array`. Objects of class `integer` and `numeric` are vectors. Matrices are just that—objects made up of elements in rows and columns, all of the same mode. Arrays are like matrices, but they can have more than two dimensions.

Some other common S3 classes are `ts` and `mts`, for time series; `factor`, for factors; `Date`, for dates; and `POSIXct`, for dates with times, all of which are numeric.

Some common classes of mode `list` are `data.frame`, for data frames; `POSTXlt`, for dates and times; and most output from higher-level functions in the packages, such as **lm** and **glm**.

The class `formula` contains formulas and is of mode `call`.

The Matrix Class: matrix

Objects of class `matrix` are matrices made up of elements of one of the atomic modes, except `NULL`, or of the modes `list` or `expression`. The three functions `matrix()`, `as.matrix()`, and `is.matrix()` exist and behave similarly to the functions for atomic modes.

The function `matrix()` creates a matrix. The function takes five possible arguments. The first argument is an object of `atomic`, `list`, or `expression` mode. The second argument is **nrow**, the number of rows. The third argument is **ncol**, the number of columns. The fourth argument is **byrow**, which tells R to create the matrix going across rows rather than down columns. The default value is **FALSE**. The **byrow** argument is useful for scanning tabular atomic data into a matrix. The fifth argument is **dimnames**, which assigns names to the rows and columns within the call to `matrix()`. The default value for **dimnames** is **NULL** and if supplied should be a list of two vectors of names. **NULL** can be substituted for either vector.

Using the array **a** from the section on vectors, two examples of creating a matrix follow:

```
> matrix( a, 3, 3 )
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    1
Warning message:
In matrix(a, 3, 3) :
  data length [8] is not a sub-multiple or multiple of the
  number of rows [3]
```

and

```
> matrix(a,3,3, byrow=T, dimnames=list( NULL, c("c1","c2","c3") ) )
     c1 c2 c3
[1,]  1  2  3
[2,]  4  5  6
[3,]  7  8  1
Warning message:
In matrix(a, 3, 3, byrow = T, dimnames = list(NULL, c("c1",
"c2",  :
  data length [8] is not a sub-multiple or multiple of the
  number of rows [3]
```

Note that R gives a warning if the product of the number of rows and columns is not a multiple of the number of elements in the first argument. The warning message does not affect the result.

For the atomic modes, if just the first argument is given, R creates a matrix with the number of rows equal to the number of elements in the object and the number of columns equal to one. If just **nrow** or **ncol** is given, R creates a matrix out of the object in the first argument with the given number of rows or columns, filling out as many of the columns

or rows that it takes to use up all of the elements in the first argument—cycling if necessary. If both **nrow** and **ncol** are present, R will go through the elements of the first argument until the matrix is full, cycling as necessary. The **byrow** argument can be used to cycle the first argument across rows rather than down columns.

For objects of the `list` mode, `matrix()` creates a matrix that describes the contents of each lowest level element of the list. The elements of the list do not need to be of the same mode. The description gives the mode of the element and the size of the element. Sometimes, a **?** is placed in the cell of the matrix. Referencing cells in the matrix returns the contents of the list for the cell. The following code gives an example:

```
> a.list = list( matrix( 1:4, 2, 2 ), c( "abc", "cde" ), 1:3,
function(){ print( 1:3 ) } )

> a.list
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
[1] "abc" "cde"

[[3]]
[1] 1 2 3

[[4]]
function ()
{
    print(1:3)
}
```

```
> matrix( a.list, 2, 2 )
     [,1]        [,2]
[1,] Integer,4   Integer,3
[2,] Character,2 ?

> matrix( a.list, 2, 2 )[2, 2]
[[1]]
function ()
{
    print(1:3)
}
```

Objects of mode `expression` are legal for `matrix()`. The result of `matrix()` is to return the contents of the expression, where the contents cycle to fill in the size of the matrix and are enclosed within an expression function statement.

The function `as.matrix()` attempts to coerce an object to class `matrix` and is mainly used with `data.frames`. If the argument to `as.matrix()` can be coerced to a vector and is not a matrix or `data.frame`, then `as.matrix()` creates a single column matrix of the coerced elements. The class is `matrix`. If the object is a matrix, `as.matrix()` just returns the matrix and maintains row and column names.

If the object is a `data.frame`, then `as.matrix()` coerces the data frame to a matrix. (A `data.frame` is a special kind of list for which the elements of the list all have the same length and the elements in a column of the list are all of the same atomic mode, but the modes are not necessarily the same between columns.) If there is a column in the `data.frame` that contains character data or raw data, then the entire `data.frame` is coerced to character. Otherwise, the `data.frame` is coerced to a logical matrix if all of the columns are logical, to an integer matrix if an integer column is present but no numeric or complex columns are present, to a numeric matrix if a numeric column is present and no complex columns are present, and to a complex matrix if a complex column is present.

Data frames can also be converted to a matrix using the `data.matrix()` function. The function `data.matrix()` converts a data frame to a matrix by coercing all of the elements in the data frame to numeric. For complex elements, the imaginary part is discarded. The function coerces character columns to **NA**s and factor columns to integers, starting with **1**. (When a data frame is created, columns of mode `character` are changed to factors by default. See the section on `data.frame()` for how `data.frame()` can handle columns of mode character.)

The following example shows the results for `as.matrix()` and `data.matrix()`, using a `data.frame` called **a.df**:

```
> a.df = data.frame( c( T, F ), 1:2, 1:2+.5, 1:2+1i, c( as.raw
( 1 ), as.raw( 10 ) ), c( "a", "b" ) )

> dimnames(a.df)=list( 1:2, c( "logical",  "integer", "double",
"complex", "raw", "character" ))

> a.df
  logical integer double complex raw character
1    TRUE       1    1.5    1+1i  01         a
2   FALSE       2    2.5    2+1i  0a         b

> as.matrix( a.df )
  logical integer double complex raw  character
1 " TRUE" "1"     "1.5"  "1+1i"  "01" "a"
2 "FALSE" "2"     "2.5"  "2+1i"  "0a" "b"

> as.matrix( a.df[,1:5] )
  logical integer double complex raw
1 " TRUE" "1"     "1.5"  "1+1i"  "01"
2 "FALSE" "2"     "2.5"  "2+1i"  "0a"

> as.matrix( a.df[,1:4] )
  logical integer double complex
1    1+0i    1+0i 1.5+0i    1+1i
2    0+0i    2+0i 2.5+0i    2+1i
```

```
> as.matrix( a.df[,1:3] )
  logical integer double
1       1       1    1.5
2       0       2    2.5

> as.matrix( a.df[,1:2] )
  logical integer
1       1       1
2       0       2

> as.matrix( a.df[,1] )
      [,1]
[1,]  TRUE
[2,] FALSE

> data.matrix( a.df )
  logical integer double complex raw character
1       1       1    1.5       1   1         1
2       0       2    2.5       2  10         2
Warning message:
In data.matrix(a.df) : imaginary parts discarded in coercion
```

The function `is.matrix()` tests whether an object is of class `matrix`. The function returns **TRUE** if the class of the argument is `matrix` and **FALSE** otherwise. If an object of mode and class `expression` is used to create a matrix or is coerced to a matrix, the result will have class `matrix`, even though the structure of the result is not `matrix`like.

More information on `matrix()`, `as.matrix()`, and `is.matrix()` can be found by entering **?matrix** at the R prompt. More information about `data.matrix()` can be found by entering **?data.matrix** at the R prompt. You can also use the Help tab in R Studio.

The Array Class: array

The `array` class is a class of data that is organized using dimensions, such as a multidimensional contingency table. Matrices can be set up as two-dimensional arrays, and vectors can be set up as one-dimensional arrays. A vector created by array() will be of class array; however, a two-dimensional array will have class `matrix`, even though `array()` creates the object.

The function `array()` creates an array out of an object. The function takes three arguments. The first argument is any object that can be coerced to a vector. The second argument is a vector that contains the size of each dimension and is of length equal to the number of dimensions of the array. The third argument is a list of names for each of the dimensions and can be omitted. The default value is **NULL**.

The following is an example of setting up an array:

```
> array( 1:12, c(  2, 3, 2 ), dimnames=list( c( "", "" ), c( "a",
"b", "c" ), NULL )  )
, , 1

 a b c
 1 3 5
 2 4 6

, , 2

 a  b  c
 7  9 11
 8 10 12
.
```

Other than there being more than two dimensions, `array()` behaves the same as `matrix()`.

The function `as.array()` attempts to coerce an object to class `array`. The object must be of the atomic modes—except for the `NULL` mode—or of the `list` or `expression` modes. Otherwise, `as.array()` returns an error. For the legal modes, `as.array()` behaves like `as.matrix()`.

The function `is.array()` tests an object to see if the class of the object is `array`. The function returns **TRUE** if the class is array and **FALSE** otherwise. Matrices return **TRUE**, independently of how the matrix was created.

More information about `array()`, `as.array()`, and `is.array()` can be found by entering **?array** at the R prompt or under the Help tab in R Studio.

The Time Series Classes: ts and mts

Classes `ts` and `mts` refer to objects that have a starting point, an end point, and a frequency or period defined, and for which observations are assumed to be at equal intervals. The default time series class for a vector of time series observations is `ts`. For a matrix of concurrent time series observations, the default classes are `mts`, `ts`, and `matrix`. The class of the time series can be changed when the time series object is created.

Time series objects can be created out of `vector`, `matrix`, some `list`, and `expression` objects—as well as some other classes of objects such as `factor` and `Date`—using the function `ts()`. Objects of mode `array` give an error. All of the atomic modes are legal as arguments for the function `ts()`, except the `NULL` mode. For `list` objects, depending on the contents and structure of the list, the `ts()` function will create a, sometimes strange, time series object. Similarly, operating on an object of mode expression with `ts()` does not give an error but does give strange results.

If the argument to `ts()` is a data frame, then the data frame is coerced to a matrix by the function `data.matrix()`. For matrix arguments, the different time series go across the columns and time goes down the rows.

The function `ts()` takes eight arguments. The first argument is the object to be changed into a time series. The second argument is **start** and gives a value for the start of the series. The third argument is **end** and gives a value for the end of the series. The fourth argument is **frequency**, which give the periodic frequency for the series. The fifth argument is **deltat**, which is the inverse of the frequency. Either **frequency** or **deltat** is supplied, not both.

The sixth argument is **ts.eps**, which gives the acceptable tolerance for comparing frequencies between different time series. The seventh argument is **class**, which tells R what class to assign to the time series object. The eighth argument is **names** and gives names to the time series for time series matrices. If no names are given, R assigns the names **Series 1**, **Series 2**, and so forth.

The second, third, fourth, and fifth arguments can be confusing. R treats monthly or quarterly data as a special case when regarding printing and plotting. Other types of periodic data have to be treated specially. For monthly data, setting **start** equal to

```
start = c('year', 'month number')
```

and **frequency** equal to

```
frequency = 12
```

or **deltat** equal to

```
deltat = 1/12,
```

where **year** is the starting year and **month number** is the number of the starting month (**1** for January, **2** for February, and so on), assigns months and years to the points in the object being converted to a time series.

To generate a monthly time series, include **end** with

```
end = c('year', 'month number'),
```

where **year** is the ending year and **month number** is the number of the ending month. The function **ts()** will cycle the first argument until the time series is filled out. (For any time series, supplying start, end, and frequency will create a time series out of the first argument by cycling. If the first argument is a matrix, each column cycles independently.)

For quarterly data, follow the same steps but use a frequency of four. For example:

```
> ts( 1:12, start=c( 2019, 2 ), freq=12 )
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2019       1   2   3   4   5   6   7   8   9  10  11
2020  12
```

```
> ts( 1:12, start=c( 2019, 2 ), freq=4 )
     Qtr1 Qtr2 Qtr3 Qtr4
2019         1    2    3
2020    4    5    6    7
2021    8    9   10   11
2022   12
```

On a more general level, say there is daily data for one week and three days and the starting week is number 32. Let **d.data** be the data. Then, the time series can be created as follows:

```
> ts( 1:12, start=c( 3, 2 ), freq=7 )
Time Series:
Start = c(3, 2)
End = c(4, 6)
Frequency = 7
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
> print( ts( 1:12, start=c( 3, 2 ), freq=7 ), calendar=T )
  p1 p2 p3 p4 p5 p6 p7
3     1  2  3  4  5  6
4  7  8  9 10 11 12
```

Note that the default for printing the time series is not in periods—except for frequencies of 4 and 12, for which R assumes that the data is monthly or quarterly. The printing of periods can be turned on and off with the **calendar** argument to `print()`.

If one number, instead of two, is used for each of **start** and **end**, then only the quantities (n+i/f) can be used as the starting and end points, where **n** is the integer of the first period, **f** is the frequency, and **i** can take integer values between zero and (f-1). The quantity (n+i/f) must be taken out to at least five decimal places if entered manually unless the argument **ts.eps** is changed from the default value of 1.0E-5. The value of **ts.eps** is set in `options()`. R is very picky here.

The function `as.ts()` attempts to coerce an object to class `ts`. Objects that are `vector`—or `matrix`like—will coerce. Arrays will not, functions will not, calls will not, and environments will not; expressions and lists will.

The function `is.ts()` tests if an object is of class `ts` and returns **TRUE** if so and **FALSE** otherwise.

More information about `ts()`, `as.ts()`, and `is.ts()` can be found by entering **?ts** at the R prompt or by using the Help tab in R Studio.

The Factor Classes: factor and ordered

The class `factor` is the class of objects that are factor levels. Factors with ordered factor levels belong to two classes, `ordered` and `factor`. Factors and ordered factors are used in modeling for which at least some categorical data is present. The mode of factors and ordered factors is numeric, and the levels are associated with integers that increase in value from one. However, when printed, the nominal levels are given.

The factor levels are usually ordered alphabetically or numerically by default, depending on the mode of the argument, but can be assigned a different order.

The three functions `factor()`, `as.factor()`, and `is.factor()` exist, as well as `ordered()`, `as.ordered()`, and `is.ordered()`. The second set of functions behaves the same as the first set with regard to creating and testing factor objects, so we only discuss the first set of functions here.

The function `factor()` creates a vector of factor levels and an associated list of levels. The function has six arguments. The first argument is the object from which the factors will be generated. The argument must be of an atomic mode or a list. Not all lists will form factors. The second argument is **levels** and sets the order of the factor levels. The **levels** argument is optional.

The third argument is **labels** and assigns labels to the levels. The third argument is optional and defaults to the values of the elements of the object. The fourth argument is **exclude** and gives any levels to be excluded in the result. Excluded levels are set to <NA>. The argument is optional and defaults to **NA**.

The fifth argument is **ordered**, which is in `factor()`, but not in `ordered()`. The argument **ordered** tells `factor()` to create a factor with ordered levels. The function `factor()` with **ordered** set to **TRUE** gives the same result as the function `ordered()` (which is only included in the current version of R for backward compatibility with S.) The sixth argument is **nmax** and is described as the maximum number of levels to use, where many values are present in the object to be made into a factor. The argument does not appear to work too well.

Converting between factors and the original data is sometimes of interest. If labels have not been assigned in `factor()`,

```
as.mode( levels( fac.obj ) )[ fac.obj ],
```

returns the original values of the object, where ***mode*** is the mode of the original object and **fac.obj** is the factor object. Note that the function,

```
as.numeric( fac.obj ),
```

returns the integers associated with the levels, even if the original object was not of mode numeric. If labels have been assigned, then usually the original data cannot be extracted.

An example follows:

```
> a.log = c( T, T, F, T )

> a.log
[1]  TRUE  TRUE FALSE  TRUE

> af1 = factor( a.log )

> af1
[1] TRUE  TRUE  FALSE TRUE
Levels: FALSE TRUE

> as.logical( levels( af1 ) )[ af1 ]
[1]  TRUE  TRUE FALSE  TRUE

> as.numeric( af1 )
[1] 2 2 1 2

> af2 = factor( a.log, levels=c( T, F ) )

> af2
[1] TRUE  TRUE  FALSE TRUE
Levels: TRUE FALSE

> as.logical( levels( af2 ) )[ af2 ]
[1]  TRUE  TRUE FALSE  TRUE

> as.numeric( af2 )
[1] 1 1 2 1

> af3 =factor( a.log, labels=c( "flab", "tlab" ) )
```

```
> af3
[1] tlab tlab flab tlab
Levels: flab tlab

> as.logical( levels( af3 ) )[ af3 ]
[1] NA NA NA NA

> as.numeric( af3 )
[1] 2 2 1 2

> as.character( levels( af3 )) [ af3 ]
[1] "tlab" "tlab" "flab" "tlab"
```

The `as.factor()` function operates the same way as `factor()`, but only takes one argument, an object to be made into a factor.

The `is.factor()` function tests if an object is a factor and returns **TRUE** if so and **FALSE** otherwise.

There is also a related function, `addNA()`. The function creates a factor object with a level for missing data (NAs). The function takes on two arguments. The first argument is an object from which an object of class `factor` can be created. The second argument is **ifany**. The **ifany** argument is logical and takes on the value **TRUE** if the extra level is only added when **NA**s are present and the value **FALSE** if the extra level is to always be included.

More information about the seven functions can be found by entering **?factor** at the R prompt or by using the Help tab in R Studio.

The Data Frame Class: data.frame

The class `data.frame` is a matrixlike class of mode `list`. Data frames and how to use them are important. Many of the data sets that are available for R are data frames. When data is read from external sources, many of the functions that do the reading create data frames. Learning how to work with and create data frames pays high dividends.

Data frames contain atomic data in rows and columns. Within a column, all of the data must be of the same mode. Across columns, the mode can change. Because data frames do not have to be of just one mode, data frames are a special kind of list.

Accessing elements of the data frame can be done like matrices or like lists, which makes data frames more versatile than the usual list. By default, the columns take names that reflect what is or is not in the original objects making up the data frame.

The functions `data.frame()`, `as.data.frame()`, and `is.data.frame()` all exist in R. In `data.frame()`, the objects to be included in the data frame are listed first, separated by commas. The objects can be any object of atomic mode or lists made up of atomic columns. If an object is made up of more than one column, like some matrices and lists, then each column in the original object becomes a column in the data frame. Otherwise, each object becomes a column. If the columns had names in the original objects, the names are brought into the data frame by default.

The objects used to make up the data frame do not have to be of the same length (or number of rows for matrices) but must be multiples of each other in length. The number of rows in the data frame will equal the length of the longest column. The data in the other columns will cycle until the column has the right number of rows. For example:

```
> a.list
[[1]]
     a1 a2
[1,]  1  7
[2,]  2  8
[3,]  3  9
[4,]  4 10
[5,]  5 11
[6,]  6 12
```

```
[[2]]
[1] "abc" "cde"

>
> data.frame( a.list, 1:3 )
  a1 a2 c..abc....cde.. X1.3
1  1  7             abc    1
2  2  8             cde    2
3  3  9             abc    3
4  4 10             cde    1
5  5 11             abc    2
6  6 12             cde    3
```

Note that R has created names for the third and fourth columns and that the third and fourth columns both cycle.

The function `data.frame()` has four arguments in addition to the objects that will make up the data frame. The first argument is **row.names**, which assigns names to the rows and by default is **NULL**, that is, no names are assigned. The second argument is **check.rows**, which is a logical argument and will check for consistency of row lengths and row names if set to **TRUE**. The default value is **FALSE**. The third argument is **check.names**, which is also logical and which checks that column names are syntactically correct and corrects names that are not. The default for **check.names** is **TRUE**.

The last argument is **stringsAsFactors**. By default, `data.frame()` converts any column containing character data into a factor. The argument **stringsAsFactors** is a logical variable. If set to **TRUE**, factors are created. If set to **FALSE**, character columns remain columns of mode `character`. The actual default value is generated by `default.stingsAsFactors()`. The value from `default.stringsAsFactors()` is set in `options()` (Chapter 15) and by default is **TRUE** but can be changed in `options()`.

The function I() can be used in the setting up of data frames. The function is another way to stop data.frame() from converting a character vector to factors. Also, I() ensures that for a matrix the column structure is maintained in the data frame. An object in the data.frame() call enclosed in I() will be treated as one element of the data frame, even if the object contains more than one column. Objects enclosed in I() do not cycle. For example:

```
> mat
     one two
row1   1   3
row2   2   4

> a.char
[1] "a1" "a2" "a3" "a4"

> a.df1 = data.frame( mat, a.char )
Warning message:
In data.frame( mat, a.char ) :
  row names were found from a short variable and have been
  discarded

> a.df1
  one two a.char
1   1   3     a1
2   2   4     a2
3   1   3     a3
4   2   4     a4

> a.df1[[ 3 ]]
[1] a1 a2 a3 a4
Levels: a1 a2 a3 a4

> a.df2 = data.frame( I( mat  ), I( a.char ) )
```

```
Error in data.frame( I(mat), I(a.char) ) :
  arguments imply differing number of rows: 2, 4

> a.df2 = data.frame( I( mat ), I( a.char[ 1:2 ] )  )

> a.df2
     mat.one mat.two a.char.1.2.
row1       1       3          a1
row2       2       4          a2

> a.df2[[ 1 ]]
     one two
row1   1   3
row2   2   4

> a.df2[[ 2 ]]
[1] "a1" "a2"
```

If row names are not entered in the call to `data.frame()`, row names are taken from the first column if the first column has row labels and does not cycle. Otherwise, row names are set to **1**, **2**, **3**, and so forth. See the above example.

The function `as.data.frame()` attempts to coerce an object to a data frame. If the object is a list made up of atomic elements (and some other simple lists) or is an object of an atomic mode, then `as.data.frame()` creates a data frame out of the object. Otherwise, `as.data.frame()` gives an error.

The function takes four arguments: the object to be coerced, **row.names**, **optional**, and **stringsAsFactors**. The arguments **row.names** and **stringsAsFactors** behave the same way as in `data.frame()`. The argument **optional** is a logical variable that, if set to **TRUE**, tells `as.data.frame()` that setting column names is optional. If set to **TRUE**, and no column names have been set in the original object, column names are not present in the result. The default value for **optional** is **FALSE**.

The function `is.data.frame()` tests if an object is of class `data.frame` and, if so, returns **TRUE**. Otherwise, `is.data.frame()` returns **FALSE**.

The functions `as.matrix()` and `data.matrix()` can be used to convert a data frame to a matrix. See the section on the `matrix` class for more information about the two kinds of conversions.

For more information about `data.frame()`, enter **?data.frame** at the R prompt. For more information about `as.data.frame()` and `is.data.frame()`, enter **?as.data.frame** at the R prompt. For more information about `I()`, enter **?I** at the R prompt. Or, use the Help tab in R Studio to access the help pages.

The Date and Time Classes: Date, POSIXct, POSIXlt, and difftime

Sometimes, working with dates and times is useful, as when printing and plotting against time. R provides classes for dates and for dates and times. The classes are `Date`, `POSIXct`, `POSIXlt`, and `difftime`. Objects of class `Date`, `POSIXct`, or `difftime` are of mode `numeric` and objects of class `POSIXlt` are of mode `list`. `Date` is the date class, and `POSIXlt` and `POSIXct` are the date and time classes. The class `difftime` contains objects formed by taking the difference between two date or two date and time objects. Of the three types of functions usual for the classes given above, only the functions `as.Date()`, `as.POSIXct()`, and `as.POSIXlt()` exist for date and date and time objects. Both `difftime()` and `as.difftime()` exist.

POSIX stands for Portable Operating System Interface and is a family of standards used by the IEEE Computer Society. The formats used in the Date, POSIXct, and POSIXlt classes are based on the POSIX standards, but the standards are not universal across platforms.

To just get a date and time stamp in R, enter **date()** at the R prompt, which returns the day of the week, date, and time. The result is of mode and class `character`. The system date function `Sys.Date()` returns the

system date and is of `numeric` mode and class `Date`. The system date and time function is `Sys.time()` and returns the system date, time, and time zone and is of mode `numeric` and classes `POSIXct` and `POSIXlt`. By default, dates are read and returned in the format "Year-Month-Day" and times are returned in the format "hour:minute:second."

There are a number of functions that operate on the date and time classes, including `weekdays()`, which returns the day of the week of objects of class `Date`, `POSIXlt`, or `POSIXct;` the function `difftime()` takes two date or date and time objects and finds the difference in time elementwise between the two objects. For class Date objects the difference between the dates are measured in days. For `POSIXlt` and `POXITct` objects the difierences are measured in seconds.

The functions `strftime()` and `strptime()` lets the user convert to or from any date time format.

More information about the date and time classes can be found at the help page for `DateTimeClasses` by entering **?DateTimeClasses** at the R prompt or by using the Help tab in R Studio to access the help pages. Information about the various date and time functions can be found at their help pages.

The function `as.Date()` creates a date object. The arguments to `as.Date()` are the object to be converted to a date; **format**, which gives the format of the object in terms of year, month, and day; **tryFormats,** which is a character string of formats to try if **format** is not given; **optional**, which is logical and, when set to **TRUE,** causes `as.Date()` to return an **NA** if format matching returns an error; **origin**, which is an origin for the first argument and must be of class `Date` or `POSIXct`; and **tz** for the time zone name.

If **origin** is used, the object to be converted can be any numeric object. If **origin** is given, the function adds or subtracts the values of the object to or from the date given by the **origin** argument and converts the result to a date. An example of weekly spacing is

```
 > as.Date( 0:1*7, origin="2019-1-1" )
[1] "2019-01-01" "2019-01-08"
```

If dates are used as the object and the dates are not in a "year-month-day" format, then the format of the dates must be given. The format is a character variable, where the POSIX standard for the year is **%Y**, the day is **%d**, and the month is **%m**, such as

```
> as.Date( "1/20/2000", format="%m/%d/%Y" )
[1] "2000-01-20"
```

Note that the format is the format of the object to be converted, not the format of the result.

The argument **tz** is for the time zone name. Some time zones are recognized, some are not. See the help page for `as.Date()` for more information.

The functions `as.POSIXct()` and `as.POSIXlt()` take the same arguments as `Date()` except that the dates can contain time, too. The default format for time is **%H:%M:%S** for hours, minutes, and seconds. For example:

```
> as.POSIXct( "1/13/2000 00:30:00", format="%m/%d/%Y %H:%M:%S" )
[1] "2000-01-13 00:30:00 CST"
```

Dates and dates and times can be operated on by addition and subtraction. Decimals for times are converted correctly. Dates in function `Date()` are incremented by days; times in the two date time functions are incremented by seconds. Examples follow:

```
> as.POSIXct( Sys.time() + 1:2*3600 )
[1] "2018-11-01 14:59:27 CDT"
[2] "2018-11-01 15:59:27 CDT"

> mode( as.POSIXct( Sys.time() + 1:2*3600 ) )
[1] "numeric"

> as.POSIXlt( Sys.time() + 1:2*3600 )
[1] "2018-11-01 15:02:53 CDT"
[2] "2018-11-01 16:02:53 CDT"
```

```
> mode( as.POSIXlt( Sys.time() + 1:2*3600 ) )
[1] "list"

> as.POSIXlt( Sys.time() ) + 1:2*3600
[1] "2018-11-01 15:07:43 CDT"
[2] "2018-11-01 16:07:43 CDT"

> mode( as.POSIXlt( Sys.time() ) + 1:2*3600 )
[1] "numeric"
```

The functions `difftime()` and `as.difftime()` are not covered here. An example of a date difference is

```
> ( Sys.Date() - as.Date( "2000-1-1" ) )
Time difference of 5125 days

> mode( Sys.Date() - as.Date( "2000-1-1" ) )
[1] "numeric"

> class( Sys.Date() - as.Date( "2000-1-1" ) )
[1] "difftime"
```

More information about date and time functions can be found by entering **?as.Date**, **?as.POSIXct**, **?as.POSIXlt**, **?difftime,** or **?as.difftime** at the R prompt or by using the Help tab in R Studio to access the help pages.

The Formula Class: formula

Formulas are used by various functions in R. For example: `lm(), glm(), nls(), plot(), coplot()`, and `boxplot()`. Formulas have their own class and are created by either setting an object equal to a formula, by using the function `formula()`, or by using the function `as.formula()`. Formulas are of mode call.

If a data frame is specified in the function using the formula, then the function looks first in the data frame for the variables in the formula. If there is no data frame assigned or if the variable is not in the data frame, where to look depends on the function used to create the formula. The difference between the three methods is in which environment R searches for the variables in the formula. Formulas that are just entered are evaluated in the environment within which the formula is used. Formulas created using `formula()` are evaluated in the environment in which the formula was created. Formulas created by `as.formula()` have an environment assigned by the **env** argument, which by default is the parent environment. For each of the functions, the formula must be quoted for the environment assignment to occur. For example:

```
> a.fun
function() {

# at the first function level
# formulas defined using the expression and formula()

  cat( "\nlevel a \n\n" )

  print( parent.frame() )
  print( environment() )

    x=1:10
    y=11:20
  cat( "\nx=", x )
  cat( "\ny=", y, "\n" )

    a.formula="y~x"
    b.formula=formula("y~x")
```

```
  b.fun=function() {

# at the second function level
# lm() is run for the formulas defined at the first level

    cat( "\nlevel b \n\n" )

      x=1:10
      y=21:30

    print( parent.frame() )
    print( environment() )

    print( lm( a.formula ) )

    cat( "\nx=", x )
    cat( "\ny=", y, "\n" )

    print( lm( b.formula ) )

# the cc environment is defined at the second level
# the formula from as.formula() is run

    cat( "\nenvironment cc \n\n" )

      cc=new.env()
        assign( "x", 1:10, env=cc )
        assign( "y", 31:40, env=cc )

    c.formula=as.formula( "y~x", env=cc )

    cat( "\nx=", cc$x )
    cat( "\ny=", cc$y, ”\n” )
    print( lm( c.formula ) )
  }
```

```
 # the second function is run at the first level

  b.fun()
}
<bytecode: 0x10a767448>
```

The function `a.fun()` is run.

```
> a.fun()

level a

<environment: R_GlobalEnv>
<environment: 0x10d3d28c8>

x= 1 2 3 4 5 6 7 8 9 10
y= 11 12 13 14 15 16 17 18 19 20

level b

<environment: 0x10d3d28c8>
<environment: 0x10d39e388>

Call:
lm(formula = a.formula)

Coefficients:
(Intercept)            x
         20            1

x= 1 2 3 4 5 6 7 8 9 10
y= 21 22 23 24 25 26 27 28 29 30

Call:
lm(formula = b.formula)

Coefficients:
(Intercept)            x
         10            1
```

```
environment cc

x= 1 2 3 4 5 6 7 8 9 10
y= 31 32 33 34 35 36 37 38 39 40

Call:
lm(formula = c.formula)

Coefficients:
(Intercept)            x
         30            1
```

The formula object **formula.a** uses the data at level b, where it is run. The formula object **formula.b** uses the data from level a, where it was created. The formula object **formula.c** uses the data in the environment cc, which was created for this example.

The formula function uses some specialized notation. The symbol ~ separates the left side of the formula from the right side. The symbol + tells R to include the variables on either side of the + in the model**.** The symbol – tells R not to use the variable to the right of the –. (Use **-1** to not use an intercept.) The symbol **:** tells R to use the interaction between the variables on either side of **:**. The symbol * tells R to use all of the levels of interaction between the variables on either side of *. If a data.frame is present in the call, the symbol **.** on the right side tells R to use all of the variables in the data frame not already used in the model. The symbol ^ tells R to use all interactions up to the level of the ^**.** The operator %in% can be used to nest variables. Functions of variables can be used within the formula, but functions involving arithmetic expressions need to be enclosed in an I() function to avoid confusing R, since the symbols have special meanings inside of the formula statement.

For more information on formulas, enter `?formula` at the R prompt or use the Help tab in R Studio.

The S4 Class

In S4, data objects have a user-defined S4 (formal) class. There are several functions associated with S4 classes, including `setClass()`, `removeClass()`, `getClass()`, `getClasses()`, and `isClass()`. S4 classes are used with S4 methods, to be covered in Chapter 7.

The function `setClass()` sets up a class and takes the arguments **Class, representation, prototype, contains, validity, access, where, version, sealed, package, S3methods,** and **slots**. The argument **Class** is a character string containing the class name. There should be no blank spaces in the string. The most important argument after the name is **slots**, which is the only extra argument that must be included. The argument **slots** is a vector with each element of the vector taking on a name and an S4 class (most S3 classes have an S4 version.) For example:

```
> setClass( "example", slots=c( x="numeric",
y="numeric",  z="matrix" ) )

> getClass( "example" )
Class "example" [in ".GlobalEnv"]

Slots:

Name:        x       y       z
Class: numeric numeric  matrix
```

The second important argument is **contains**. The argument consists of the names of other classes to be included in the class being defined. The slots in the classes listed in the **contains** argument are included in the new class. The names are a vector of character strings. For example:

```
> setClass( "example.2", slots=c( xx="numeric", yy="numeric",
zz="matrix" ), contains="example" )
```

```
> getClass( "example.2" )
Class "example.2" [in ".GlobalEnv"]

Slots:

Name:       xx      yy      zz
Class: numeric numeric  matrix

Name:        x       y       z
Class: numeric numeric  matrix

Extends: "example"
```

According to the authors at CRAN, the arguments **where**, **sealed**, and **package** are redundant and need not be included. The argument **prototype**, which gives default values for the slots, is better implemented using the function `initialize()`, and the argument **validity,** which sets restrictions on the values in the slots, is better implemented using the function `setValidity()`.

According to the authors at CRAN, the arguments **representation, access, version,** and **S3methods** are deprecated and should not be used.

The function `removeClass()` removes a class. It takes two arguments. The first argument is the name of the class to be removed, in quotes. The second argument is **where**—the environment in which to start looking for the class. The default value is the environment where `removeClass()` is run. (To make changes to a class, remove the class and redefine it.)

The function `getClass()` returns the contents of a class. The function takes three arguments, the name of the class in quotes, **.Force**, and **where**. The argument **.Force** is a logical variable. If set to **TRUE**, a NULL rather than an error is returned if the class does not exist. The default value is **FALSE.** The argument **where** is as described in the last paragraph. The two examples given above use `getClass()`.

The function `getClasses()` gets the classes in an environment. The function takes two arguments, **where** and **inherits**. The argument **where** tells R the specific environment to search. The argument **inherits** is a logical variable which, when set to **TRUE**, tells the function to look in all of the parent environements. By default, **inherits** equals **TRUE** if **where** is not used and **FALSE** otherwise. An example:

```
> getClasses( .GlobalEnv )
[1] "example.2" "example"
```

Running `getClasses()` without an argument returns every class in the parent environments, which can be many.

The function `isClass()` tests if a class is an S4 (formal) class. The function takes on three arguments, **Class**, **formal**, and **where**. The argument **Class** is the name of the class, enclosed in quotes. The argument formal is always set to **TRUE**, indicating that the test is for S4 (formal) classes. The argument **where** tells R in which environment to look for the class. By default, the level of the calling environment is used. For example:

```
> isClass( "example" )
[1] TRUE

> isClass( "numeric" )
[1] TRUE
```

Here, **numeric** is both an S3 and an S4 class.

More information about S4 (formal) classes can be found by entering **?setClass, ?getClass**, and **?getClasses** at the R prompt, or by using the Help tab in R Studio.

Names for Vectors, Matrices, Arrays, and Lists

A chapter on objects would not be complete without information on how to set names for vectors, matrices, arrays, and lists. Dimension names are always of `character` mode. For objects of more than one dimension, the name objects are put together in a `list`.

To see what names a vector has or to assign names to a vector, the `names()` function is used. The function just has one argument, the object. For example:

```
> cde
 [1] 21 22 23 24 25 26 27 28 29 30

> names( cde )
NULL

> names( cde ) = paste( "v", 1:10, sep="" )

> cde
 v1  v2  v3  v4  v5  v6  v7  v8  v9 v10
 21  22  23  24  25  26  27  28  29  30

> names( cde )
 [1] "v1"  "v2"  "v3"  "v4"  "v5"  "v6"  "v7"  "v8"  "v9"  "v10"

> mode( names( cde ) )
[1] "character"

> class( names( cde ) )
[1] "character"
```

You can also assign names directly to vectors at the time the vector is created. For example:

```
> a.vec = c( a=1, b=2, c=3 )
> a.vec
a b c
1 2 3
```

Some objects of mode `list` are vectors. For such lists, assigning names to the lowest level of the list is done with `names()` or by direct assignment.

For matrices, there are three possible functions used to see the names or to assign names: `rownames()`, `colnames()`, and `dimnames()`. The functions `rownames()` and `colnames()` have three arguments, the R object, **do.NULL**, and **prefix**. The argument **do.NULL** is logical with default value **TRUE**, which tells the function to do nothing if the row or column names are NULL. If **do.NULL** is **FALSE**, the row or column names are indexed with the prefix equal to the value of the argument **prefix**. For example:

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

> colnames( mat )
NULL

> colnames( mat ) = colnames( mat, do.NULL=F, prefix="cl" )


> mat
     cl1 cl2
[1,]   1   3
[2,]   2   4
```

Note that the right-hand side of the third expression only returns the names of the columns and does not do the assignment.

The function `dimnames()` can be used to see or assign names to matrices and arrays. If `dimnames()` operates on an object, then the names of the dimensions in the object are returned as a list. If names are assigned using `dimnames()`, the object on the right side of the assignment must be a list with the same number of lowest level elements as there are dimensions in the object and with each lowest level element either being **NULL** or of the same length as there are elements in each dimension of the matrix or array. For example:

```
> a
, , d31

    d21 d22
d11   1   3
d12   2   4

, , d32

    d21 d22
d11   5   7
d12   6   8

>
> dimnames( a )
[[1]]
[1] "d11" "d12"

[[2]]
[1] "d21" "d22"

[[3]]
[1] "d31" "d32"

>
> dimnames( a ) = list( c( "11", "12") ,c( "21", "22" ),
c( "31", "32" ) )
```

```
>
> a
, , 31

   21 22
11  1  3
12  2  4

, , 32

   21 22
11  5  7
12  6  8
```

More information about names can be found by entering **?names**, **?rownames**, or **?dimnames** at the R prompt or by using the Help tab in R Studio.