

Chapter 1

R Objects

In R, objects can have one or more classes, consisting of the class of the scalar value and the class of the data structure holding the scalar value. Use the `is()` command to determine what an object *is*. If you are already familiar with R objects, you may skip to Section ?? for loading data, or Section ?? for a description of Zelig commands.

1.1 Scalar Values

R uses several classes of scalar values, from which it constructs larger data structures. R is highly class-dependent: certain operations will only work on certain types of values or certain types of data structures. We list the three basic types of scalar values here for your reference:

1. **Numeric** is the default value type for most numbers. An **integer** is a subset of the **numeric** class, and may be used as a **numeric** value. You can perform any type of math or logical operation on numeric values, including:

```
> log(3 * 4 * (2 + pi))      # Note that pi is a built-in constant,
[1] 4.122270                 #   and log() the natural log function.
> 2 > 3                      # Basic logical operations, including >,
[1] FALSE                   #   <, >= (greater than or equals),
                             #   <= (less than or equals), == (exactly
                             #   equals), and != (not equals).
> 3 >= 2 && 100 == 1000/10    # Advanced logical operations, including
[1] TRUE                     #   & (and), && (if and only if), | (or),
                             #   and || (either or).
```

Note that `Inf` (infinity), `-Inf` (negative infinity), `NA` (missing value), and `NaN` (not a number) are special numeric values on which most math operations will fail. (Logical operations will work, however.)

2. **Logical** operations create logical values of either **TRUE** or **FALSE**. To convert logical values to numerical values, use the `as.integer()` command:

```
> as.integer(TRUE)
[1] 1
> as.integer(FALSE)
[1] 0
```

3. **Character** values are text strings. For example,

```
> text <- "supercalafragilisticxpaladocious"
> text
[1] "supercalafragilisticxpaladocious"
```

assigns the text string on the right-hand side of the `<-` to the named object in your workspace. Text strings are primarily used with data frames, described in the next section. R always returns character strings in quotes.

1.2 Data Structures

1.2.1 Arrays

Arrays are data structures that consist of only one type of scalar value (e.g., a vector of character strings, or a matrix of numeric values). The most common versions, one-dimensional and two-dimensional arrays, are known as *vectors* and *matrices*, respectively.

Ways to create arrays

1. Common ways to create **vectors** (or one-dimensional arrays) include:

```
> a <- c(3, 7, 9, 11)    # Concatenates numeric values into a vector
> a <- c("a", "b", "c") # Concatenates character strings into a vector
> a <- 1:5               # Creates a vector of integers from 1 to 5 inclusive
> a <- rep(1, times = 5) # Creates a vector of 5 repeated '1's
```

To manipulate a vector:

```
> a[10]                # Extracts the 10th value from the vector `a`
> a[5] <- 3.14          # Inserts 3.14 as the 5th value in the vector `a`
> a[5:7] <- c(2, 4, 7)  # Replaces the 5th through 7th values with 2, 4, and 7
```

Unlike larger arrays, vectors can be extended without first creating another vector of the correct length. Hence,

```
> a <- c(4, 6, 8)
> a[5] <- 9      # Inserts a 9 in the 5th position of the vector,
                  # automatically inserting an `NA' values position 4
```

2. A **factor vector** is a special type of vector that allows users to create j indicator variables in one vector, rather than using j dummy variables (as in Stata or SPSS). R creates this special class of vector from a pre-existing vector \mathbf{x} using the `factor()` command, which separates \mathbf{x} into levels based on the discrete values observed in \mathbf{x} . These values may be either integer value or character strings. For example,

```
> x <- c(1, 1, 1, 1, 1, 2, 2, 2, 2, 9, 9, 9, 9)
> factor(x)
[1] 1 1 1 1 1 2 2 2 2 9 9 9 9
Levels: 1 2 9
```

By default, `factor()` creates unordered factors, which are treated as discrete, rather than ordered, levels. Add the optional argument `ordered = TRUE` to order the factors in the vector:

```
> x <- c("like", "dislike", "hate", "like", "don't know", "like", "dislike")
> factor(x, levels = c("hate", "dislike", "like", "don't know"),
+       ordered = TRUE)
[1] like    dislike   hate     like     don't know  like    dislike
Levels: hate < dislike < like < don't know
```

The `factor()` command orders the levels according to the order in the optional argument `levels`. If you omit the `levels` command, R will order the values as they occur in the vector. Thus, omitting the `levels` argument sorts the levels as `like < dislike < hate < don't know` in the example above. If you omit one or more of the levels in the list of levels, R returns levels values of `NA` for the missing level(s):

```
> factor(x, levels = c("hate", "dislike", "like"), ordered = TRUE)
[1] like    dislike hate     like     <NA>    like    dislike
Levels: hate < dislike < like
```

Use factored vectors within data frames for plotting (see Section ??), to set the values of the explanatory variables using `setx` (see Section ??) and in the ordinal logit and multinomial logit models (see Section ??).

3. Build **matrices** (or two-dimensional arrays) from vectors (one-dimensional arrays). You can create a matrix in two ways:
 - (a) From a vector: Use the command `matrix(vector, nrow = k, ncol = n)` to create a $k \times n$ matrix from the vector by filling in the columns from left to right. For example,

```

> matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)
      [,1] [,2] [,3]      # Note that when assigning a vector to a
[1,]    1    3    5      # matrix, none of the rows or columns
[2,]    2    4    6      # have names.

```

- (b) From two or more vectors of length k : Use `cbind()` to combine n vectors vertically to form a $k \times n$ matrix, or `rbind()` to combine n vectors horizontally to form a $n \times k$ matrix. For example:

```

> x <- c(11, 12, 13)      # Creates a vector `x' of 3 values.
> y <- c(55, 33, 12)      # Creates another vector `y' of 3 values.
> rbind(x, y)             # Creates a 2 x 3 matrix. Note that row
      [,1] [,2] [,3]      # 1 is named x and row 2 is named y,
x    11   12   13        # according to the order in which the
y    55   33   12        # arguments were passed to rbind().
> cbind(x, y)             # Creates a 3 x 2 matrix. Note that the
      x y                # columns are named according to the
[1,] 11 55                # order in which they were passed to
[2,] 12 33                # cbind().
[3,] 13 12

```

R supports a variety of matrix functions, including: `det()`, which returns the matrix's determinant; `t()`, which transposes the matrix; `solve()`, which inverts the the matrix; and `%*%`, which multiplies two matrices. In addition, the `dim()` command returns the dimensions of your matrix. As with vectors, square brackets extract specific values from a matrix and the assignment mechanism `<-` replaces values. For example:

```

> loo[,3]                # Extracts the third column of loo.
> loo[1,]                # Extracts the first row of loo.
> loo[1,3] <- 13          # Inserts 13 as the value for row 1, column 3.
> loo[1,] <- c(2,2,3)     # Replaces the first row of loo.

```

If you encounter problems replacing rows or columns, make sure that the `dims()` of the vector matches the `dims()` of the matrix you are trying to replace.

4. An **n-dimensional array** is a set of stacked matrices of identical dimensions. For example, you may create a three dimensional array with dimensions (x, y, z) by stacking z matrices each with x rows and y columns.

```

> a <- matrix(8, 2, 3)    # Creates a 2 x 3 matrix populated with 8's.
> b <- matrix(9, 2, 3)    # Creates a 2 x 3 matrix populated with 9's.
> array(c(a, b), c(2, 3, 2)) # Creates a 2 x 3 x 2 array with the first
, , 1                      # level [,1] populated with matrix a (8's),
, , 2                      # and the second level [,2] populated

```

```

      [,1] [,2] [,3]      # with matrix b (9's).
[1,]      8      8      8
[2,]      8      8      8      # Use square brackets to extract values. For
                                # example, [1, 2, 2] extracts the second
                                # value in the first row of the second level.
, , 2                                # You may also use the <- operator to
                                # replace values.
      [,1] [,2] [,3]
[1,]      9      9      9
[2,]      9      9      9

```

If an array is a one-dimensional vector or two-dimensional matrix, R will treat the array using the more specific method.

Three functions especially helpful for arrays:

- `is()` returns both the type of scalar value that populates the array, as well as the specific type of array (vector, matrix, or array more generally).
- `dims()` returns the size of an array, where

```

> dims(b)
[1] 33 5

```

indicates that the array is two-dimensional (a matrix), and has 33 rows and 5 columns.

- The single bracket `[]` indicates specific values in the array. Use commas to indicate the index of the specific values you would like to pull out or replace:

```

> dims(a)
[1] 14
> a[10]      # Pull out the 10th value in the vector `a`
> dims(b)
[1] 33 5
> b[1:12, ]  # Pull out the first 12 rows of `b`
> c[1, 2]    # Pull out the value in the first row, second column of `c`
> dims(d)
[1] 1000 4 5
> d[ , 3, 1] # Pulls out a vector of 1,000 values

```

1.2.2 Lists

Unlike arrays, which contain only one type of scalar value, lists are flexible data structures that can contain heterogeneous value types and heterogeneous data structures. Lists are so flexible that one list can contain another list. For example, the list `output` can contain `coef`,

a vector of regression coefficients; **variance**, the variance-covariance matrix; and another list **terms** that describes the data using character strings. Use the **names()** function to view the named elements in a list, and to extract a named element, use

```
> names(output)
[1] coefficients  variance  terms
> output$coefficients
```

For lists where the elements are not named, use double square brackets **[[]]** to extract elements:

```
> L[[4]]      # Extracts the 4th element from the list `L'
> L[[4]] <- b # Replaces the 4th element of the list `L' with a matrix `b'
```

Like vectors, lists are flexible data structures that can be extended without first creating another list of with the correct number of elements:

```
> L <- list()                # Creates an empty list
> L$coefficients <- c(1, 4, 6, 8) # Inserts a vector into the list, and
                                # names that vector `coefficients'
                                # within the list
> L[[4]] <- c(1, 4, 6, 8)     # Inserts the vector into the 4th position
                                # in the list. If this list doesn't
                                # already have 4 elements, the empty
                                # elements will be `NULL' values
```

Alternatively, you can easily create a list using objects that already exist in your workspace:

```
> L <- list(coefficients = k, variance = v) # Where `k' is a vector and
                                             # `v' is a matrix
```

1.2.3 Data Frames

A data frame (or data set) is a special type of list in which each variable is constrained to have the same number of observations. A data frame may contain variables of different types (numeric, integer, logical, character, and factor), so long as each variable has the same number of observations.

Thus, a data frame can use both matrix commands and list commands to manipulate variables and observations.

```
> dat[1:10,]      # Extracts observations 1-10 and all associated variables
> dat[dat$grp == 1,] # Extracts all observations that belong to group 1
> group <- dat$grp  # Saves the variable `grp' as a vector `group' in
                    # the workspace, not in the data frame
> var4 <- dat[[4]] # Saves the 4th variable as a `var4' in the workspace
```

For a comprehensive introduction to data frames and recoding data, see Section ??.

1.2.4 Identifying Objects and Data Structures

Each data structure has several *attributes* which describe it. Although these attributes are normally invisible to users (e.g., not printed to the screen when one types the name of the object), there are several helpful functions that display particular attributes:

- For arrays, `dims()` returns the size of each dimension.
- For arrays, `is()` returns the scalar value type and specific type of array (vector, matrix, array). For more complex data structures, `is()` returns the default methods (classes) for that object.
- For lists and data frames, `names()` returns the variable names, and `str()` returns the variable names and a short description of each element.

For almost all data types, you may use `summary()` to get summary statistics.