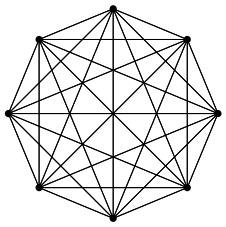
Chapter 4

Vectors



The estimated populations of Florida during the years 1900 through 1990, in thousands, from the United States decennial census counts, are given below.

1900	1910	1920	1930	1940	1950	1960	1970	1980	1990
529	753	968	1,468	1,897	2,771	4,952	6,789	9,746	12,938

The spectacular growth of the population was partially fueled by the invention of air conditioning during the twentieth century. One could easily create ten simple objects in R, perhaps named `p00`, `p10`, `p20`, ... , `p90`, to hold the ten population values. For a longer list of data values, however, this approach would be cumbersome. There is a more efficient mechanism in R for storing the ten values. Since the values are all of the same nature (that is, they are all populations), a single object, known as a *vector*, can hold all ten values in a single object. A vector is an ordered sequence of elements of a particular type that can be accessed by an index. The three topics introduced in this chapter are (a) creating a vector, (b) extracting elements of a vector by using subscripts, and (c) performing arithmetic operations on a vector.

4.1 Creating a vector

A vector is a linear arrangement of elements. Four different mechanisms for creating a vector in R are:

- the `:` operator,
- the `c` (combine) function,
- the `seq` (sequence) function, and
- the `rep` (repeat) function.

These four mechanisms will be presented in the order given above. Each mechanism for creating a vector is useful in various settings. Vectors are an instance of what computer scientists refer to as a *data structure*. In subsequent chapters, other data structures, such as matrices, arrays, lists, and data frames, will be defined. The vector is the primary data structure in R, so mastering it is important before moving on to more complex data structures.

We begin with the `:` operator for creating a vector. The number that appears to the left of the colon tells R the value in the first element of the vector; the number that appears to the right of the

colon tells R the value in the last element of the vector. (A minor exception to this rule is illustrated subsequently.) The absolute difference between successive elements in a vector created with the `:` operator is one. As a simple example, the R command to create a vector whose elements are the first 10 positive integers is

```
> 1:10                    # a vector of the first ten positive integers
 [1]  1  2  3  4  5  6  7  8  9 10
```

The ten elements, which reside in positions 1 to 10, are displayed following the creation of the vector. The elements of a vector can be negative, as illustrated in the creation of a nine-element vector.

```
> -3:5                    # integers from -3 to 5
[1] -3 -2 -1  0  1  2  3  4  5
```

The colon operator increments each successive element by one. If the number to the left of the colon is larger than the number to the right of the colon, R decrements each successive element by one.

```
> 5:-3                    # integers from 5 to -3
[1]  5  4  3  2  1  0 -1 -2 -3
```

There is no limit, other than machine memory, to the length of a vector. The R command to create a vector whose elements are the first 100 positive integers, for example, is

```
> 1:100                   # integers from 1 to 100
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
 [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
 [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
 [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
 [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
 [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

The 100 elements of the vector are displayed on one line after another. The reason for the `[1]` on the display provided by R seen in the previous two chapters now becomes clear. The index of the first element displayed on each line is given in brackets. R displays the vector using a default width of 80 character positions. The format for the display is known as *monospace*, which means that each character displayed assumes the same width on the screen. This is handy for aligning values. The display width can be altered with the `options` function, which will be described in Chapter 9. The R command above simply echoes the value of the vector. It can also be useful to store the vector in an object. So the R command

```
> x = 1:100               # store the integers from 1 to 100 in the vector x
```

stores the first 100 positive integers in the vector named `x`. The numbers on either side of the colon need not necessarily be integers.

```
> 4.5:25                  # the elements won't make it to 25
 [1]  4.5  5.5  6.5  7.5  8.5  9.5 10.5 11.5 12.5 13.5 14.5 15.5 16.5 17.5
[15] 18.5 19.5 20.5 21.5 22.5 23.5 24.5
```

The first element is 4.5, and each subsequent element is one more than the previous element. The 15th element of this vector, for example, contains 18.5.

The colon operator is useful for creating vectors whose elements differ by one. But not all vectors have this property. A more general way of creating vectors is with the `c` (combine) function, which concatenates its arguments together to form a vector. The syntax for the `c` function is

```
c(element1, element2, ...)
```

The numeric values that are placed inside the parentheses are known generically as *arguments* to the function. So to create a ten-element vector that contains the populations (in thousands) of Florida every decade from 1900 to 1990, use the R command

```
> c(529, 753, 968, 1468, 1897, 2771, 4952, 6789, 9746, 12938)
 [1]   529   753   968  1468  1897  2771  4952  6789  9746 12938
```

In order to set this vector to the object `FloridaPop`, use the assignment operator as in the previous chapter.

```
> FloridaPop = c(529, 753, 968, 1468, 1897, 2771, 4952, 6789, 9746, 12938)
> FloridaPop                        # display Florida populations
 [1]   529   753   968  1468  1897  2771  4952  6789  9746 12938
```

If a data set contains missing observations, R allows you to set these elements of a vector to `NA`, which stands for "Not Available." This is for leaving a position in a vector for a data value that should be present, but, for some reason, is missing. The 2000, 2010, 2020, 2030, and 2040 Florida populations can be placed in the vector `FloridaPop2` in the same fashion.

```
> FloridaPop2 = c(15982, 18801, 21538, NA, NA)
```

The `c` function can be used to concatenate several vectors. The R command to concatenate all of the population counts since Florida became a state in 1845 is

```
> FloridaPopAll = c(c(87, 140, 188, 269, 391), FloridaPop, FloridaPop2)
> FloridaPopAll                     # display Florida populations
 [1]    87   140   188   269   391   529   753   968  1468  1897  2771  4952
[13]  6789  9746 12938 15982 18801 21538    NA    NA
```

The resulting vector `FloridaPopAll` has $5 + 10 + 5 = 20$ elements.

So far we have used two methods, namely the `:` operator and the `c` function, to create vectors. The `seq` (sequence) function, which is a generalization of the `:` operator, can also be used to create a vector of numeric values. The syntax for the `seq` function is

```
seq(from = 1, to = 1, ...)
```

The first argument to the `seq` function is named `from`, and it has a default value of `1`; the second argument is named `to`, and it also has a default value of `1`. There are several optional arguments that follow the first two arguments, and this is indicated by the three dots that follow the `from` and `to` arguments in the syntax. The `from` and `to` arguments can be given without including their names by placing numerical values in the first two positions in a call to the `seq` function. For example, to create a vector associated with the years in the Florida population values with first element 1850, last element 2040, and increment `by = 10`, use

```
> seq(1850, 2040, by = 10)          # a vector from 1850 to 2040 by 10
 [1] 1850 1860 1870 1880 1890 1900 1910 1920 1930 1940 1950 1960 1970 1980
[15] 1990 2000 2010 2020 2030 2040
```

The `by` argument allows increments other than the $+1$ and -1 increments that are used by the `:` operator. The next example creates a vector identical to the one in the previous call to the `seq` function, but this time uses the argument `length.out = 20` to force the vector to be of length 20.

```
> seq(1850, 2040, length.out = 20) # a vector from 1850 to 2040 of length 20
 [1] 1850 1860 1870 1880 1890 1900 1910 1920 1930 1940 1950 1960 1970 1980
[15] 1990 2000 2010 2020 2030 2040
```

The choice between using the `by` or `length.out` argument to control the increment and the length of the vector created by the `seq` function depends on the context of the problem being addressed. Depending on the setting, the `by` argument or the `length.out` argument might be more convenient for creating a particular vector. A final example of the `seq` function illustrates a negative increment between elements.

```
> seq(2040, 1845, by = -10)        # a vector from 2040 to 1845 by -10
 [1] 2040 2030 2020 2010 2000 1990 1980 1970 1960 1950 1940 1930 1920 1910
[15] 1900 1890 1880 1870 1860 1850
```

A vector of twenty elements is created with the years running from 2040 to 1850. The last element of the vector is 1850 because the increment of -10 between the elements takes precedence over the vector ending with the year 1845, which was the year of Florida statehood.

The fourth and final method for creating a vector presented here is the `rep` (repeat) function. The syntax for the `rep` function is

```
rep(x, times, ...)
```

The first argument `x` is an object to be repeated; the second argument `times` is an object that indicates how many times the first object should be repeated. The first example of `rep` repeats 5 four times.

```
> rep(5, 4)                        # repeat 5 four times
[1] 5 5 5 5
```

The next example repeats the vector consisting of 1, 2, and 3 twice.

```
> rep(1:3, 2)                      # repeat 1:3 two times
[1] 1 2 3 1 2 3
```

The next example repeats the vector consisting of 1, 3, and 5 twice.

```
> rep(seq(1, 5, by = 2), 2)        # repeat (1, 3, 5) two times
[1] 1 3 5 1 3 5
```

The next example repeats two copies of 1, followed by four copies of 5.

```
> rep(c(1, 5), c(2, 4))            # repeat 1 twice and 5 four times
[1] 1 1 5 5 5 5
```

The next example creates a vector that repeats one copy of 1, followed by three copies of 3, followed by five copies of 5.

```
> rep(seq(1, 5, 2), seq(1, 5, 2))  # one 1, then three 3s, then five 5s
[1] 1 3 3 3 5 5 5 5 5
```

Finally, using the `each` argument, each element in the vector in the first argument is repeated four times.

```
> rep(1:3, each = 4)               # four 1s, then four 2s, then four 3s
 [1] 1 1 1 1 2 2 2 2 3 3 3 3
```