Chapter 3

The Building Blocks

Up to this point we have hopefully learned how to use R as a basic calculator and you know how to do some basic arithmetic, use basic functions, and access R's help functionality. To move beyond using R as a calculator this chapter will introduce the main building blocks of R – *objects* and their *modes*. The discussion that follows is not technically correct (in fact it is a gross mischaracterization) but it should help make sense of things and why things in R happen the way they do.

From here on out, you are no longer allowed type directly into the R Console!

3.1 Objects

R is an object oriented language. As I mentioned in passing above everything in R is an *object*. When R does anything, it creates and manipulates *objects*. R's *objects* come in different types and flavors. The most basic ones are:

→ **Vectors:** These are one-dimensional sequences of elements of the same *mode*. (More on *modes* later: see section 3.2.) For example, this could be vector of length 26 (i.e. one containing 26 elements) where each element is a letter in the alphabet.

→ **Matrices & Arrays:** These are two dimensional rectangular objects (matrices) and higher-dimensional rectangular objects (arrays). All elements of matrices or arrays have to be of the same mode.

→ **Lists:** Lists are like vectors but they do not have to contain elements of the same mode. The first element of a list could be a vector of the 26 letters of the alphabet. The second element could contain a vector of all the prime numbers below 1000. A third could be a 2 by 7 matrix.

→ **Data Frames:** Data frames are best understood as special matrices (technically they are a type of list). For most applications involving datasets you will use data frames. They are two dimensional containers with rows corresponding to 'observations' and columns corresponding to 'variables.'

→ **Factors:** Factors are vectors to classify categorical data. They behave differently than vectors containing numerical, integer, or character elements.

→ **Functions:** Functions are *objects* that take other *objects* as inputs and return some new *object*. We will deal with functions separately in a later chapter.

3.2 Modes

All objects have a certain *mode*. Some objects can only deal with one *mode* at a time, others can store elements of multiple *modes*. R distinguishes the following modes:

1. **integer**: integers (e.g. 1, 2 or -69)
2. **numeric**: real numbers (e.g 2.336, -0.35)
3. **complex**: complex or imaginary numbers
4. **character**: elements made up of text-strings (e.g. "text", "Hello World!", or "123")
5. **logical**: data containing logical constants (i.e. `TRUE` and `FALSE`)

3.3 Assignment and Reference

Knowing the types of objects R can work with is not terribly useful without knowing how to store these objects and without knowing how to recall or reference them when needed. You often will compute some statistic or manipulate some matrix. Instead of recalculating everything over and over again we can give things names and recall them later. Below we will cover how to create, assign to and refer to various *objects*. We shall use vectors as examples.

3.3.1 Playing with trivial Vectors

Recall our basic arithmetic examples from above. We implicitly relied on and then manipulated objects and R implicitly printed these objects to the screen.

```
> 1 + 2
[1] 3
```

Let's assign and recall names instead. We can do that by using the assignment operator "`<-`". Think of this as the M+ button on your calculator.

```
> Answer <- 1 + 2
>
```

We can use just about any name we like so long as it is not a number or does not start with a number (e.g. `3 <- 1 + 2` will not work, neither will `3Answer <- 1 + 2`). It is very useful to use descriptive names such as `NumberOfStudents <- 17` instead of `n <- 17`. Don't confuse yourself.

As you can see in the example above. **R** no longer gives you the answer to our problem. It just returns the prompt. Luckily you are familiar with **R**'s `print()` function and you can recall or print the results to the screen.

```
> print(Answer)
[1] 3
```

If you give **R** the name of some object it knows you don't even have to use the `print()` function. Just type in the name and **R** will do it's thing.

```
> Answer
[1] 3
```

Whether you know it or not you have now already created an *object* of the vector type (of length 1). We can verify this with the `is()` function. When supplied with the name of an *object*, this function will tell you what type of *object* we have as well as its *mode*.

```
> is(Answer)
[1] "numeric" "vector"
```

Recall: 1, 2, 3, or 16 are internal objects. Try it!

```
> is(3)
[1] "numeric" "vector"
```

Named *objects* behave just like the ones **R** already knows. This is pretty useful:

```
> Answer * 2
[1] 6
```

Or ...

```
> Answer2 <- Answer * sqrt(Answer)
> Answer2
[1] 5.196152
```

Keeping Track of Objects

To see what objects you have created (the ones **R** stored in active memory) you can use the `ls()` function.

```
> ls()
[1] "Answer"  "Answer2"
```

If you want to remove an *object* from memory use the `rm()` function. Be very careful. This will delete thing permanently. Don't delete things you need.

```
> rm(Answer2)
> ls()
[1] "Answer"
```

If you want to remove all *objects* from active memory this will do the trick:

```
> rm(list = ls())
>
```

3.3.2 *Real* Vectors

So far we have only created trivial vectors of length 1. Let's assign some longer ones. To do this you will use the `c()` function. The "c" stands for concatenate, and you can string a bunch of elements together, separated by commas.

```
> Vector1 <- c(1,2,3,4,5,6,7,8,9,10)
> Vector1
 [1]  1  2  3  4  5  6  7  8  9 10
```

How about a character vector?

```
> Vector2 <- c("a", "b", "c", "d")
> Vector2
[1] "a" "b" "c" "d"
```

Or ...

```
> Vector3 <- c("1", "2", "3", "4")
> Vector3
[1] "1" "2" "3" "4"
```

You can also string multiple vectors together with the `c()` function.

```
> Vector4 <- c(Vector2, Vector3, Vector2, Vector2, Vector2)
> Vector4
 [1] "a" "b" "c" "d" "1" "2" "3" "4" "a" "b" "c" "d" "a"
[14] "b" "c" "d" "a" "b" "c" "d"
```

Vector Operations

Most standard mathematical functions work with vectors.

```
> Vector1 + Vector1
 [1]  2  4  6  8 10 12 14 16 18 20
```

```
> Vector1 / Vector1
[1] 1 1 1 1 1 1 1 1 1 1
```

```
> log(Vector1)
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
[6] 1.7917595 1.9459101 2.0794415 2.1972246 2.3025851
```

Here we are nesting the `log()` function inside the `round()` function.

```
> round(log(Vector1))
[1] 0 1 1 1 2 2 2 2 2 2
```

The `round()` function takes an argument (`digit`) to specify how many decimals to display. It defaults to 0. Let's see a few more digits.

```
> round(log(Vector1), digit = 3)
 [1] 0.000 0.693 1.099 1.386 1.609 1.792 1.946 2.079 2.197
[10] 2.303
```

To do other useful things to vectors consider these functions:

Function	Description
`sum()`	sums of the elements of the vector
`prod()`	product of the elements of the vector
`min()`	minimum of the elements of the vector
`max()`	maximum of the elements of the vector
`mean()`	mean of the elements
`median()`	median of the elements
`range()`	the range of the vector
`sd()`	the standard deviation
`var()`	the variance (on `n-1`)
`cov()`	the covariance (takes two inputs `cov(x,y)`)
`cor()`	the correlation coefficient (takes two imputs cor(x,y))
`sort()`	sorts the vector (argument: `decreasing = FALSE`)
`length()`	returns the length of the vector
`summary()`	returns summary statistics
`which()`	returns the index after evaluating a logical statement
`unique()`	returns a vector of all the unique elements of the input

```
> sum(Vector1)
[1] 55

> prod(Vector1)
[1] 3628800

> median(Vector1)
[1] 5.5

> sd(Vector1)
[1] 3.02765

> sort(Vector1, decreasing = TRUE)
[1] 10  9  8  7  6  5  4  3  2  1

> length(Vector1)
[1] 10

> summary(Vector1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00

>which(Vector1 >= 5) # note this returns the index not the
   elements (try it with Vector2)
[1]  5  6  7  8  9 10
```

Simplifying Vector Creation

Most of the time using the `c()` function will be tedious as you don't want to manually type all elements of a vector. Luckily the good folks responsible for R have thought of you.

You can use the colon to tell R to create an integer vector.

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

Or the `seq()` function, which is more general and has some neat features.

```
> seq(from = 0, to = 10) # you can drop the argment names
[1]  0  1  2  3  4  5  6  7  8  9 10

> seq(0, 10)
[1]  0  1  2  3  4  5  6  7  8  9 10

> seq(0, 10, by = 2) # the 'by' argument let's you set the
                     # increments
[1]  0  2  4  6  8 10

> seq(0, 10, length.out = 25) # the 'length.out' argument
                             # specifies the length of the
                             # vector and R figures out the
                             # increments itself

 [1]  0.0000000  0.4166667  0.8333333  1.2500000  1.6666667
 [6]  2.0833333  2.5000000  2.9166667  3.3333333  3.7500000
[11]  4.1666667  4.5833333  5.0000000  5.4166667  5.8333333
[16]  6.2500000  6.6666667  7.0833333  7.5000000  7.9166667
[21]  8.3333333  8.7500000  9.1666667  9.5833333 10.0000000
```

Another useful function is `rep()` which allows you to repeat things.

```
> rep(0, time = 10)
[1] 0 0 0 0 0 0 0 0 0 0

> rep("Hello", 3) # as always you can drop the argument name
[1] "Hello" "Hello" "Hello"

> rep(Vector1, 2) #  repeating Vector 1 twice
 [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9
[20] 10

> rep(Vector2, each = 2) # we can repeat each element as well

[1] "a" "a" "b" "b" "c" "c" "d" "d"
```

Indexing

Sometimes you do not want to print or manipulate an entire vector. This is where indexing comes in. Indexing vectors is done with [ ]. Check it out.

```
> Vector6 <- c("The", "Starlab", "Fellow", "is", "a Fool.")
> Vector6
[1] "The" "Starlab" "Fellow" "is" "a Fool"

> length(Vector6) # how long is Vector6
[1] 5

> Vector6[3] # with the bracket we reference the third
   element
[1] "Fellow"

> Vector6[2:4] # we can reference a sequence of elements
[1] "Starlab" "Fellow"  "is"

> Vector6[c(1,3,4)] #  or any elements we like
[1] "The" "Fellow" "is"

> Vector6[-2] #  all except the 2nd element
[1] "The" "Fellow" "is" "a Fool."

> Vector6[5] <- "great." # and we can change elements
> Vector6
[1] "The" "Starlab" "Fellow" "is" "great."
```

Logical operators come in handy when indexing:

```
> Vector7 <- c(1, 1, 2, 3, 4, 4.5, 6, 6, 10)
> Vector7
[1]  1.0  1.0  2.0  3.0  4.0  4.5  6.0  6.0 10.0

> Vector7[Vector7 == 1]
[1] 1 1

> Vector7[Vector7 >= 4]
[1]  4.0  4.5  6.0  6.0 10.0

> Vector7[Vector7 != sqrt(16) & Vector7 > 2]
[1]  3.0  4.5  6.0  6.0 10.0
```

More Functions

Consider the following three functions: `na.omit()`, `subset()`, and `sample()`. This will become very useful later when dealing with real data. Let's make a new vector called `foo`:

```
> foo <- c(2, 3, 4, 3, NA, NA, 6, 6, 10, 11, 2, NA, 4, 3)
> foo
[1]  2  3  4  3 NA NA  6  6 10 11  2 NA  4  3

> max(foo) # this won't work because many function can't deal with NAs
[1] NA

> summary(foo) # this works
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  2.000   3.000   4.000   4.909   6.000  11.000       3
```

This is where the `na.omit()` function comes in. This function returns the vector supressing the `NA`s and adds an attribute to it called `na.action`.

```
> na.omit(foo)
[1]  2  3  4  3  6  6 10 11  2  4  3
attr(,"na.action")
[1]  5  6 12
attr(,"class")
[1] "omit"
```

This is helpful because now we can compute all those functions that break when they encounter NAs. Instead of supplying the object foo we can supply the object returned by na.omit().

```
> max(na.omit(foo))
[1] 11
```

The summary() function is useful check whether NAs are present in your object. The is.na() function is more powerful. Combined with the subset() function we can remove the NAs manually. This requires you to write a logical statement. The first argument you need to supply is the object you want to subset. The second should be the logical statement **R** should evaluate.

```
>is.na(foo)
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
[9] FALSE FALSE FALSE  TRUE FALSE FALSE

> foo.noNA <- subset(foo, is.na(foo)==FALSE)
> foo.noNA
[1]  2  3  4  3  6  6 10 11  2  4  3
```

Of course the subset() function can be used for more than NA removal. Let's use it to find numbers divisible by 7.

```
> X <- 1:500 # creating a vector from 1 to 500
> Multiple7 <- subset(X, X%%7==0) # recall the modulo
   operator
> Multiple7
 [1]   7  14  21  28  35  42  49  56  63  70  77  84
[13]  91  98 105 112 119 126 133 140 147 154 161 168
[25] 175 182 189 196 203 210 217 224 231 238 245 252
[37] 259 266 273 280 287 294 301 308 315 322 329 336
[49] 343 350 357 364 371 378 385 392 399 406 413 420
[61] 427 434 441 448 455 462 469 476 483 490 497
```

The sample() function will also come in handy later. It takes the following arguments: size for the sample size, and replace = TRUE for whether you want to sample with or without replacement. Let's sample from our vector, Multiple7. Obviously, your output may/will look different than what I got here.

```
> sample(Multiple7, size = 10, replace = FALSE)
[1] 497 238 322  63  77 245 455 126 490 392
```

The `print()`, `cat()`, and `paste()` Functions

We already know that the `print()` function prints an object to the screen by explicitly creating an object in the computers active memory. The `paste()` function is a bit more useful as you can paste multiple objects together and print them to the screen (by creating an implicit object - a character vector). The `cat()` function does the same thing but it does not create an object in the computer's active memory.

```
> print(0.2)
[1] 0.2

> X <- 0.2
> print(X)
[1] 0.2

> paste(X, "is equal to", X)
[1] "0.2 is equal to 0.2"

> cat(X, "is equal to", X) # notice the missing [1] below
0.2 is equal to 0.2 >
```