

# Some Fundamental R Programming Concepts

Lesson 01

Psychology 310

## 1 Why Are We Doing This?

It's a fair enough question to ask. You signed up to study *Psychology*, and here you are in a *Statistics* course! And no sooner does the statistics course start, than you're learning some *Computer Programming*!

To some of you, this may seem like you signed up for a sea cruise to the Mediterranean, and now you've found yourself on a train, it's freezing outside, and the destination is Siberia. What's going on?

OK, we admit it. We tricked you . . . .

Well, not really. If you are going to study Psychology, and get a Ph.D., you will need to be able to do some statistics. Actually, you will need to do a *lot* of statistics. You may think there is a way around this. You'll get someone else to do the statistics. Your advisor will tell you what buttons to press. Sorry, that is unlikely to be enough. Not only that, but if you want to be an outstanding Ph.D. with a chance at a really good job at a top 20 institution, you're going to need to *really know what you are doing* when you do your statistics. Your goal should not be to learn how to follow orders from your advisor. Your goal should be to end up knowing more than your advisor.

How are we going to reach that goal together?

### 1.1 Leverageable Knowledge

Many of you have studied statistics in a single undergraduate course. The goal was to teach you a few basic techniques, and perhaps a few key ideas. We want to take you deeper, and further.

The key is to teach ideas that are *leverageable*, that is, *ideas that can be used easily to generate new ideas and new knowledge*.

Statistical ideas can be difficult to study. At times they can seem bewilderingly abstract. We need a tool that we can use to quickly put our ideas into action, so that we can make them more concrete. That way we can play with them at the same time we study them.

R is just such a tool. Just a few minutes after using R for the first time, you can use it to answer some challenging questions while playing with some important ideas. R can bring ideas to life. All you need to know is how to tell it what to do.

So although it may *seem* esoteric and arcane, and at times it *will* prove clumsy and annoying, R is worth understanding.

R is a statistical program, but it is also a statistical programming environment. By learning how to program R, you can create your own statistics functions to do *exactly what you want to do, the way you want to do it*. You can also use R to play *what-if games*, scenarios in which you examine *what* would happen in your data analyses *if* certain conditions were true. To some extent, this process can help make a game out of studying statistics. It can also provide you accurate answers to complex questions in situations in which you might feel like your statistical knowledge level is overmatched.

Moreover, you will never pay for R. It is free, open source software. There are no annual license fees, few if any problems if you shift from a PC to a Mac, or to Linux. R runs in all those environments. SAS and SPSS programs are excellent software, and they have worked well for many people in many places. Your own research lab may use this software extensively, and it may not cost you anything to use it. Consequently, You may *think* SAS or SPSS is a great deal with no downside. However, if you ever leave academics, you'll spend a lot of money on licensing fees to run the software.

That is why a lot of people and departments are moving away from SAS and SPSS, and starting to rely more on R.

## 1.2 Programming — It's Not Just for R

There are lots of great commercial statistical software packages out there besides SAS and SPSS. For example, Stata and Statistica are both terrific packages. All of these programs have a programming capability built into them. While you're learning some basic concepts of R programming, you're also going to learn about programming in general. What you learn can be generalized easily to commercial statistical software packages. It can also be used to program *Microsoft Excel*, or, further on down the road, your own web page should you choose to create one.

The truth is, someone with absolutely no computer programming experience has a serious knowledge deficit in the modern scientific community. Trust me, this is not a deficit you want to carry with you on your road to

the Ph.D..

## 1.3 Summing it Up

So why are we here? Because to gain the full power of statistical software, you need to be able to write programs. To be able to write programs, you need to understand some basic programming concepts. Once you learn these concepts, you'll be able to make R do what you want, when you want it. You can use R to have fun while learning statistics. You'll have a highly leverageable skill that will serve you well in other venues, as well. In what follows, we'll be concentrating on *general* programming concepts, rather than their specific application to statistics.

# 2 Vector Objects—and Simple Operations on Them

## 2.1 Introduction

R is an *object-oriented* language. In R, there are several *types* of objects. You can discover the type of something with the `typeof` function. R is also an *interpreted* language. When you enter a syntactically complete statement and press the <Enter> key, R evaluates it immediately. If you press the <Enter> key and rather than getting a response, you see a + sign, it means you have not actually entered a complete response. Often this will happen when you accidentally leave out a parenthesis in a complicated expression.

For example, in your R console window, if you type `4 + 2` and press the <Enter> key, you get 6.

```
> 4 + 2
[1] 6
```

But what happens if you type just `4 +` and press the <Enter> key? Try it.

You see a '+' character. This is the *continuation character*

When you see this character at the beginning of the line, it is an invitation to “keep going.” In this case, just enter the 2 to complete the statement.

Now you will see the result

```
> 4 +  
+ 2  
[1] 6
```

## 2.2 Vector Objects

Those of you who have taken a course in linear algebra learned that a single number is called a *scalar* and a list of numbers is called a *vector*.

The distinction between scalars and vectors is not maintained in R. That is, a scalar is a vector of length 1.

Think of a vector of length  $n$  as a stack of  $n$  cells that might be filled with things.

## 2.3 Variables, Assignment Statements, and Vectorized Operations

For example, a vector  $\mathbf{x}$  might have the numbers 1 through 4. R has a very elegant way of entering a contiguous sequence of numbers. The sequence of integers from 1 to 4 is entered as `1:4`.

To assign these numbers to a vector  $\mathbf{x}$ , simply do the following.

```
> x <- 1:4
```

If we then type the letter  $\mathbf{x}$  and press <Enter>, R evaluates the expression by printing the current value of the vector object  $\mathbf{x}$ .

```
> x  
[1] 1 2 3 4
```

Although you seldom see them, left-to-right assignment operations are also allowed, such as

```
> 5:7 -> w  
> w  
[1] 5 6 7
```

Many computer languages define their assignment statements in terms of an equals sign, such as

```
> y = 10:15
> y
[1] 10 11 12 13 14 15
```

This works in R too, but it is better to use the `<-` operator.

A right-to-left assignment statement assigns the result of the expression on the right to the object on the left of the assignment operator.

So if we start with the vector `x` defined as

```
> x
[1] 1 2 3 4
```

and then write

```
> x <- x + 1
```

we end up with

```
> x
[1] 2 3 4 5
```

Look carefully at what happened. R took the vector `x`, added 1 to it, and assigned the result to — the vector `x`. This type of statement allows us to update a variable easily.

The operation was vectorized, i.e., what statisticians call a *listwise* operation was performed. That is, the number 1 was added to all elements of the vector `x`.

Now, *actually*, what happened internally in R may be thought of in terms that are slightly more complex than what I described. R has a rule for arithmetic operations on vectors. Remember that everything is a vector. An individual number is a vector of length 1, and in the above example, `x` is a vector of length 4.

The general rule for vector operations is that all operations are *element-wise*. For example, consider two vectors `x` and `y`. If they are the same size  $n$ , then the product `x * y` is a vector of length  $n$ , and each element in that vector is the product of the corresponding elements in `x` and `y`.

For example, suppose `x` and `y` are

```
> x
[1] 1 2 3 4
> y
[1] 0 1 2 3
```

Then their product is? See for yourself.

```
> x * y
[1] 0 2 6 12
```

But what if  $y$  is not the same size as  $x$ ? For example,

```
> x
[1] 1 2 3 4
> y
[1] 1 2
```

Then what?

```
> x * y
[1] 1 4 3 8
```

Here is what R did. It *recycled* the elements from the vectors that are too short, so that they equalized the lengths of the vectors. In the example immediately preceding, the vector  $y$  has only 2 elements, while  $x$  has 4. So the two elements were recycled. Thus, for the multiplication with  $x$ , the vector  $y$  was expanded to

```
[1] 1 2 1 2
```

Now we can see what is really happening when we write  $x+1$ . The vector  $x$  has length 4, while the vector  $1$  (*remember, in R single numbers are really vectors!*) is only of length 1. So when you give R an expression that tells it to evaluate  $x + 1$ , it recycles the  $1$  four times and really adds the vector  $x$  to a vector with four 1's in it.

Most of the time, you are not aware of the fact that this is going on behind the scenes. But it turns out to be very convenient, because it means that you can write a statement like  $w <- 2*x + 5$  and R will multiply every element of  $x$  by 2, add 5 to every element, and assign the result to  $w$ .

Take a look at this example, using the current  $x$ .

```
> x
[1] 1 2 3 4
> 2*x + 5
[1] 7 9 11 13
```

## 2.4 The Concatenation Function

A key function in R is the *concatenation function* `c`. It takes a group of objects and stacks them in a vector.

Here are some examples.

```
> x <- c(1,2,3,4,6)
> x
[1] 1 2 3 4 6
> c(x,5)
[1] 1 2 3 4 6 5
> c(0,x,7)
[1] 0 1 2 3 4 6 7
> c(x,x)
[1] 1 2 3 4 6 1 2 3 4 6
```

## 2.5 The rep Function

In the last example, we stacked two copies of `x`. Another way of doing that is to use the replicate function

```
> rep(x,2)
[1] 1 2 3 4 6 1 2 3 4 6
> rep(1,10)
[1] 1 1 1 1 1 1 1 1 1 1
```

## 2.6 Arithmetic Operator Precedence

Sometimes, an operation will produce a result that is completely surprising. In quite a few instances, that will be because you didn't understand the rules of *operator precedence*. When evaluating an expression, R will apply operators according to precedence rules. In general, expressions are evaluated from left to right, but with certain operators taking precedence over others where there is an ambiguity. We won't describe the precedence rules in full detail, but for the basic arithmetic and logical operations, the order of precedence, from highest to lowest, is

Operator	Function
+ -	(unary use of the plus or minus sign)
:	(sequence creation)
^	(exponentiation)
* /	(multiplication, division)
+ -	(addition, subtraction)
== != < > <= >=	(comparison)
!	(logical "not")
& &&	(logical "and", front-element "and")
	(logical "or", front-element "or")
->	(left-to-right assignment)
<-	(right-to-left assignment)

You can always use parentheses to group operations and help avoid errors. Such errors occur due sometimes because of a lack of understanding of operator precedence, but they can also result from the sheer visual confusion induced by complex expressions. Just remember that multiplication is always explicit in R. Unlike some other languages where the expression  $4(4+2)$  would evaluate to 24 this expression generates an error message in R. You need to enter

```
> 4*(4+2)
[1] 24
```

Here are some examples that illustrate how operator precedence and expression evaluation work:

```
> 4 + 2 * 7
```

```
[1] 18
```

Note that, because the multiplication operation takes precedence over addition, the answer is  $4 + 14 = 18$ , not  $6 * 7 = 42$ .

```
> 4 * 2^3 + 7
```

```
[1] 39
```

In this case, the exponentiation operation was performed first, yielding  $2^3 = 8$ . Next came the multiplication, yielding  $4 * 8 = 32$ . Last came the addition, yielding  $32 + 7 = 39$ .

```
> J <- 5
```

```
> 0:J-1
```

```
[1] -1 0 1 2 3 4
```

Note that since the sequence operation takes precedence over the subtraction operation, the second line in the example *first* generates the sequence  $0:5$ , then subtracts 1 from it, thus yielding the result shown.

```
> -1:5
```

```
[1] -1 0 1 2 3 4 5
```

Note that the unary  $-$  is evaluated before the sequence operation! Study carefully the difference between this next expression and the preceding one.

```
> -(1:5)
```

```
[1] -1 -2 -3 -4 -5
```

```
> 5 - 1 / 2
```

```
[1] 4.5
```

```
> (5-1)/2
```

```
[1] 2
```

## 2.7 Logical Expressions and Comparison Operators

A substantial part of computer programming involves calculating whether certain conditions are true. Often, decisions are made by comparing quantities.

Consider the following expression and its result:

```
> 4 > 3
[1] TRUE
```

The `>` operator evaluates whether the expression to its left is greater than the expression to its right. So the expression `4 > 3` evaluates whether 4 is greater than 3. Since it is, the logical value `TRUE` is returned.

Such evaluations may be vectorized. Consider the following example.

```
> x <- -2 : 5
> x
[1] -2 -1  0  1  2  3  4  5
> x > 0
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

The `<` operator evaluates whether the expression to its left is less than the expression to its right.

```
> 5 < 4
[1] FALSE
> x
[1] -2 -1  0  1  2  3  4  5
> x < 4
[1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

In a similar vein, the `>=` and `<=` operators evaluate “greater than or equal to” and “less than or equal to,” respectively.

```
> 5 >= 5
[1] TRUE
```

```

> x <- -2 : 5
> x
[1] -2 -1  0  1  2  3  4  5
> x >= 2
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
> x <= 3
[1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE

```

Equality is evaluated with the == operator. *Note the double equal sign!* Many programming errors in R result from using a single equals sign when two should be used. Study the following example:

```

> x <- -2 : 5
> x
[1] -2 -1  0  1  2  3  4  5
> y <- x == 2
> y
[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE

```

The “not equals” operator is !=.

```

> 4 != 2
[1] TRUE
> x <- -2 : 5
> x
[1] -2 -1  0  1  2  3  4  5
> x != 3
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

```

There are several logical operators for evaluation of more complex conditions.

The logical “and” operator & is used to evaluate whether two expressions are simultaneously TRUE.

Here is an example.

```
> x <- 4
> x < 6
[1] TRUE
> x > 3
[1] TRUE
> x < 6 & x > 3
[1] TRUE
> x < 3
[1] FALSE
> x > 1
[1] TRUE
> x < 3 & x > 1
[1] FALSE
```

The logical “or” operator is applied to two expressions to determine whether at least one of the two expressions is true.

```
> x <- 4
> x < 6
[1] TRUE
> x > 3
[1] TRUE
> x < 6 | x > 3
[1] TRUE
```

There are longer forms of the “and” and “or” operators that work in the following way: The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined.

```
> x <- 1:5
> y <- 3:7
> x > 2
```

```

[1] FALSE FALSE TRUE TRUE TRUE
> y > 2
[1] TRUE TRUE TRUE TRUE TRUE
> x > 2 & y > 2
[1] FALSE FALSE TRUE TRUE TRUE
> x > 2 && y > 2
[1] FALSE
> x > 2 | y > 2
[1] TRUE TRUE TRUE TRUE TRUE
> x > 2 || y > 2
[1] TRUE

```

### 3 Character Expressions, Lists, and Indexing

So far, we've been working only with vectors, and the vectors have contained either numbers or logical results. Another form of data we frequently process with R is character data. We signify character data by placing the character string within quotes.

```

> last.names <- c("Romney", "Obama", "Bush", "Clinton")
> last.names
[1] "Romney" "Obama" "Bush" "Clinton"

```

We now see that vectors (at least) can contain numbers, logical results, or character strings.

However, vectors have a serious limitation. They cannot contain a mixture of things. Try entering the following:

```

> presidents <- c("Romney",2012)
> presidents
[1] "Romney" "2012"

```

Notice how the 2012 which I tried to enter as a number got converted to "2012". R realized that the two elements were of different types. It is much easier to convert 2012 to a character string than to convert "Romney" to a number. Consequently, R made the internal decision to convert the 2012 to a character string.

I can access the second element of `presidents` by using a subscript, i.e.,

```
> presidents
[1] "Romney" "2012"
> presidents[2]
[1] "2012"
```

However, if I try to divide that number by 4, I run into a problem because R converted the number to text.

```
> presidents[2]/4
```

```
Error in presidents[2]/4 : non-numeric argument to binary operator
```

*Lists* in R have many of the virtues of vectors, but allow you to include items of different types.

```
> presidents <- list(c("Obama","Bush","Bush"),c(2008,2004,2000))
> presidents
[[1]]
[1] "Obama" "Bush"  "Bush"

[[2]]
[1] 2008 2004 2000
> presidents[[2]]/4
[1] 502 501 500
```