DOING ARITHMETIC IN R

A Caveat.

Before we begin, let me issue a caveat. These tutorials are not a complete reference manual for the R language--far from it! R can do many more things than you have or will see outlined in these brief tutorials. My dilemma is to balance the desire for completeness with the need for brevity. I want to include things that you MAY find handy someday, if for no other reason than to illustrate what R is capable of. On the other hand, too much detail all at once can be overwhelming. So use your best judgment as to what you think you need to know. If something doesn't look useful to you right now, skip it! You can always come back if you find eventually that you need to know it. On the other hand, don't get carried away with skipping stuff either. Most of this material you do need to know in order to use R effectively. Okay, on with it!

More On R Functions.

We will begin this tutorial by looking at two functions that create vectors of numerical values that are in a regular sequence. We've already seen that this can be done as follows...

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 10:1                                  # And backwards too.
 [1] 10  9  8  7  6  5  4  3  2  1
```

Recall, in this context, that the colon means "to", as in "1 to 10". Those values are printed to the console (screen), and that's the end of it. We have not saved them by assigning them to an object, and therefore, we cannot use them in any further calculations. This is not terribly useful, but we shall get to usefulness later.

There is another way to produce this same sequence, which is by using the `seq( )` function...

```
> seq(from=1, to=10, by=1)
 [1]  1  2  3  4  5  6  7  8  9 10
```

The syntax should be self-explanatory. The function has created a regular sequence of integers "from" 1 "to" 10 "by" adding 1 at each step. The `seq( )` function is more flexible than the colon operator because the function can be made to step by any amount you want, whereas the colon operator can only step by one...

```
> seq(from=1, to=10, by=2)
[1] 1 3 5 7 9
```

It can even step by fractional values...

```
> seq(from=1, to=10, by=2.5)
[1] 1.0 3.5 6.0 8.5
```

So here is the point I want to illustrate about R functions. In this case, every argument inside the `seq( )` function is named: from, to, and by. It's unnecessary to use these names...

```
> seq(1,10,2)
[1] 1 3 5 7 9
```

R will understand what the arguments are, AS LONG AS they are given in the correct order: first the "from" value, then the "to" value, and finally the "by" value. On the other hand, if you don't remember what order the arguments are supposed to be in, and you don't want to take the time to look it up, then they can be given IN ANY ORDER, as long as they are labelled...

```
> seq(to=10, by=2, from=1)
[1] 1 3 5 7 9
```

This is a general behavior for R functions. Arguments can be given without labels, as long as they are in the correct order, or they can be given in any order with labels. The correct order, as well as the correct labels, can be found by going to the help page for the function.

Of course, just having something printed to the screen is not often useful. If we are creating this sequence to be used in subsequent calculations, we would want it stored in an object...

```
> seq(to=10, by=2, from=1) -> my.seq
> my.seq / 3
[1] 0.3333333 1.0000000 1.6666667 2.3333333 3.0000000
> class(my.seq)
[1] "numeric"
> is.vector(my.seq)
[1] TRUE
```

Once the sequence is stored, it can be manipulated mathematically, in this case, divided by 3. The `class( )` function tells you this object you've created is numeric (i.e., has numbers in it). The `is.vector( )` function is a question: Is this a vector? TRUE, "my.seq" is a vector.

One final point should be made here. The values in "my.seq" have not been changed by the arithmetic we did on them...

```
> my.seq
[1] 1 3 5 7 9
> my.seq / 3 -> my.seq
> my.seq
[1] 0.3333333 1.0000000 1.6666667 2.3333333 3.0000000
```

That doesn't happen unless the results of the division are stored back into the "my.seq" object. As a general rule, when R prints something to the screen, nothing in the workspace or in the computer's memory has been altered. That is, whatever values you have stored in objects are still the same. On the other hand, when an assignment is made, R generally does not print anything to the screen until you ask it to. This is worth remembering!

Another function that creates regular sequences is the `rep( )` function, which stands for "repeat"...

```
> rep(x=1, times=10)                    # With the arguments labelled.
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(1,10)                             # Without the arguments labelled.
 [1] 1 1 1 1 1 1 1 1 1 1
```

What the heck good could that possibly be? A few examples should illustrate...

```
> my.seq = 1:5                          # Create a fresh sequence.
> rep(my.seq, times=3)                  # Repeat this vector 3 times.
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> rep(my.seq, times=c(3,2,4,0,1))       # See the explanation below.
 [1] 1 1 1 2 2 3 3 3 3 5
```

If the first argument is a vector, it will be repeated "times" times. If both arguments are vectors, then things start to get really useful. The "times" vector specifies how often each element of the first vector should be repeated. In the example above, 1 is repeated 3 times, 2 is repeated 2 times, 3 is repeated 4 times, 4 is not repeated at all, and 5 is repeated 1 time. And this is useful how?

Suppose you have a vector of measurements, the first five of which are from men and the second five of which are from women...

```
> height = c(70, 72, 67, 66, 75, 64, 66, 68, 63, 65)
```

You can create a vector of gender labels for the measurements as follows...

```
> gender = rep(c("male","female"),c(5,5))
> gender
 [1] "male"   "male"   "male"   "male"   "male"   "female" "female" "female"
 [9] "female" "female"
```

Now you can use the gender vector as a grouping variable (or indexing vector) for the height vector to get means by gender, for example...

```
> by(FUN=mean, IND=gender, data=height)
gender: female
[1] 65.2
------------------------------------------------------------
gender: male
[1] 70
>                                       # Just press Enter to put a space here.
> t.test(height ~ gender)

        Welch Two Sample t-test

data:  height by gender
t = -2.588, df = 6.039, p-value = 0.04108
...                                     # Some output omitted here.
```

Or suppose you have the following frequency distribution, and you want to put all the data into a single vector...

X	freq
12	3

11	2
10	0
9	3
8	3
7	5
6	3
5	0
4	1

```
> X = 12:4
> freq=c(3,2,0,3,3,5,3,0,1)
> rep(X, freq) -> all.in.one
> all.in.one
 [1] 12 12 12 11 11  9  9  9  8  8  8  7  7  7  7  7  6  6  6  4
> mean(all.in.one)
[1] 8.3
```

Or, following the instructions in your elementary statistics book, you can also get the mean of X as follows...

```
> sum(X * freq) / sum(freq)
[1] 8.3
```

This, in my opinion, makes R an unparalleled teaching tool for elementary stat courses. Let's see you do that in SPSS!

Enough of this for now. We will use both of these functions in due time. You should clean up your workspace now.

R As a Simple Calulator.

R will do simple arithmetic from the command line. Several examples should suffice to illustrate...

```
> 18 + 12                                 # addition
[1] 30
> 18 - 12                                 # subtraction
[1] 6
> 18 * 12                                 # multiplication
[1] 216
> 18 / 12                                 # division
[1] 1.5
> 18 %/% 12                               # just the integer part of the quotient
[1] 1
> 18 %% 12                                # just the remainder part (modulo)
[1] 6
> 18 ^ 12                                 # exponentiation (raising to a power)
[1] 1.156831e+15
```

In the last case, the answer was such a large number that R printed it in scientific notation. Don't ignore the exponent part! This answer is not 1.1568, as many of my students often try to claim. It is 1.156831 TIMES 10 raised to the 15th power. In other words, it is:

```
1.156831 x 1,000,000,000,000,000 = 1,156,831,000,000,000
```

R prints all very large and very small numbers in scientific notation. You will need to know how it works. If you don't (including cases where the exponent of 10 is negative), here is a link to a Wikipedia page that explains it...

Scientific notation at Wikipedia

Of course, R obeys the usual rules for order of operations, and it uses parentheses for grouping operations...

```
> 18 - 12 / 3
[1] 14
> (18 - 12) / 3
[1] 2
```

And R recognizes certain goofs, like trying to divide by zero, and points them out...

```
> 18 / 0
[1] Inf
> 0 / 0
[1] NaN
> "eighteen" / 12
Error in "eighteen"/12 : non-numeric argument to binary operator
```

Technically, eighteen divided by zero is undefined, but most computer software will tell you it is infinity ("Inf" in R-speak). Zero

divided by zero is not a number ("NaN" in R-speak). Trying to divide a word by a number is just silliness, of course. There are also more advance operators, such as those that manipulate matrices, but I'll leave those to be investigated by the few readers who may be interested in such things. These operators will also work with complex numbers, but once again, that's beyond the scope of these tutorials. See `help("+")` for more details.

Mathematical Functions.

The following examples illustrate SOME of the mathematical functions available in R...

```
> log(10)                             # natural log (base e)
[1] 2.302585
> exp(2.302585)                       # antilog, e raised to a power
[1] 10
> log10(100)                          # base 10 logs; log(100, base=10) is the same
[1] 2
> sqrt(88)                            # square root
[1] 9.380832
> factorial(8)                        # factorial
[1] 40320
> choose(12,8)                        # combinations (binomial coefficients)
[1] 495
> round(log(10), digits=3)            # rounding; round(log(10),3) also works
[1] 2.303
> signif(log(10), digits=3)           # significant digits (wrong answer in this case!)
[1] 2.3
> runif(5)                            # five uniform random numbers between 0 and 1
[1] 0.3088106 0.6893187 0.5312068 0.2848143 0.4390779
> rnorm(5)                            # random numbers from a normal distribution; N(0,1)
[1]  0.39655158 -0.90683680  0.70820865 -0.06417678  0.25064385
> abs(18 / -12)                       # absolute value
[1] 1.5
```

There are others, such as the standard trig functions (cos(x), sin(x), and tan(x), along with their inverses: acos(x), asin(x), atan(x)), and the hyperbolic trig functions, as well as many more advanced math functions. Try `help(gamma)` for an example.

R Can Get You a Date (Kind Of).

Here are a couple R functions for working with dates...

```
> date()
[1] "Wed Jul 28 12:48:18 2010"
> difftime("2008-07-05","1992-08-15")
Time difference of 5803 days
```

In fact, there is an entire add-on package that does nothing but deal with dates written in different formats. More on add-on packages in a future tutorial.

Vectorized Arithmetic.

Now for some good stuff! In R arithmetic, one or more of the arguments can be a vector. Here is an example of how useful this can be...

```
> height.inches = c(68,65,70,71,69)
> height.cm = height.inches * 2.54
> height.cm
[1] 172.72 165.10 177.80 180.34 175.26
```

When a vector is operated on by a single value, the single value operates on each value of the vector in turn. Math functions work the same way...

```
> log(height.inches)
[1] 4.219508 4.174387 4.248495 4.262680 4.234107
```

When a vector operates on a vector, the operation is done term by term, which is to say, the first term of one operates on the first term of the other, the second term on the second term, and so on...

```
> correction = c(1,0,0,-1,-2)
> height.inches + correction
[1] 69 65 70 70 67
```

There are special functions designed to work specifically on vectors. Here are a few...

```
> max(height.inches)                  # maximum value
[1] 71
> min(height.inches)                  # minimum value
[1] 65
> sum(height.inches)                  # sum
[1] 343
> mean(height.inches)                 # arithmetic mean
[1] 68.6
> median(height.inches)               # median
[1] 69
> range(height.inches)                # range (actually min and max in one)
[1] 65 71
> var(height.inches)                  # sample variance
[1] 5.3
> sd(height.inches)                   # sample standard deviation
[1] 2.302173
> length(height.inches)               # number of values in the vector
[1] 5
```

These functions can be very useful for teaching purposes. They take the tedium out of calculating the sum of square, for example, for which there is no R function...

```
> sum(height.inches^2) - sum(height.inches)^2 / length(height.inches)
[1] 21.2
```

A student who can do this will not soon forget how the SS is calculated!

Sorting, Ranking, and Ordering Vectors.

If you've already erased height.inches, recreate it. Then use the following functions to sort it in increasing then in decreasing order...

```
> sort(height.inches)                       # Increasing order is the default.
[1] 65 68 69 70 71
> sort(height.inches, decreasing=TRUE)      # Or just sort(height.inches, T).
[1] 71 70 69 68 65
```

To find the ranks that correspond to values in a vector, use the `rank( )` function...

```
> height.inches
[1] 68 65 70 71 69
> rank(height.inches)                     # Rank 1 is the minimum value.
[1] 2 1 4 5 3
```

The `order( )` function is tricky but very useful. Let's see it at work, and then I will explain what it is doing...

```
> height.inches
[1] 68 65 70 71 69
> sort(height.inches)
[1] 65 68 69 70 71
> order(height.inches)
[1] 2 1 5 3 4
> order(height.inches) -> ord
> height.inches[ord]
[1] 65 68 69 70 71
```

In the example above, the first command simply printed out the "height.inches" vector so you could get another look at it. The `sort( )` function was then used to rearrange it into ascending order. A useful thing to know, sometimes, is how the vector was rearranged to achieve that sorting. That's what the `order( )` function tells you. The output of that function says essentially, "Put the second item first, the first item second, the fifth item third, the third item fourth, and the fourth item last." The output of the `order( )` function can also be used to sort a vector, as is shown by the last two commands. This is useful when you want to sort two vectors in the same order, i.e., keeping the values in the two vectors properly paired up. If you don't see it now, don't worry, but we will also be using this function to sort a data frame by one of its variables, so you will see it eventually.

Relational and Logical Operations.

Values can be compared using the following operations:

- > means greater than
- >= means greater than or equal to
- < means less than
- <= means less than or equal to
- == means equal to (two equal signs)
- != means not equal to (exclamation mark means "not")

Here are some examples of how these might be used. Once again, I will use the "height.inches" vector to illustrate, so recreate it if you've erased it.

```
> height.inches
[1] 68 65 70 71 69
> height.inches >= 70
[1] FALSE FALSE  TRUE  TRUE FALSE
> height.inches == 70
[1] FALSE FALSE  TRUE FALSE FALSE
> height.inches != 70
[1]  TRUE  TRUE FALSE  TRUE  TRUE
> which(height.inches >= 70)
[1] 3 4
> all(height.inches <= 72)
[1] TRUE
> any(height.inches <= 65)
[1] TRUE
> which(height.inches <= 65)
[1] 2
```

First, the "height.inches" vector is printed out so you have a copy to look at. The second command compares each value of the vector to 70 and returns a logical result: FALSE if the value is not greater than or equal to 70, TRUE if it is. The third command returns TRUE only if the value is exactly equal to 70, and the fourth command returns TRUE only if the value is not equal to 70. The fifth command asks a question: which values in the vector are greater than or equal to 70? The answer is items 3 and 4. The sixth command asks if all values in the vector and less than or equal to 72. The answer is TRUE, meaning yes. The seventh command asks if any of the values are equal to or less than 65, and once again the answer is TRUE, meaning yes. The last command asks which ones? The answer is item 2 in the vector is less than or equal to 65. These functions are useful for subsetting the data, for example, in situations where you might want to look at cases where the subjects were six feet tall or more. Play with these commands a bit, and you will get used to them.

One final note: In R, logical values can be added. TRUEs add as ones, and FALSEs add as zeros. Here are a couple examples...

```
> sum(height.inches >= 70)
[1] 2
> sum(height.inches <= 69)
[1] 3
```

So there are two cases in the vector in which the height is equal to or greater than 70, and three cases in which the height is less than or equal to 69.

Finally, don't forget to clean up!

---

revised 2010 July 28

Return to the Table of Contents