Roith

R Tutorial: Writing R expressions

1. **Arithmetic Expressions:**
   Earlier we discussed the R expression

   ```
   >3*(11.5 + 2.3)
   ```

   This is an example of a simple arithmetic expression using the *operators* * and + and the pair of parentheses enclosing a sub-expression. The value of an expression can be *assigned* to a name to create a new R *object* that may be used in other R expressions. Names of existing objects and constants are combined with *operators* and *functions* to form more complex *expressions.*

   ```
   > g <- 3*(11.5 + 2.3)
   > g
   [1] 41.4
   > g1 <- 4.7 / sqrt(g + 2) -6.4
   > g1
   [1] -5.686567
   ```

   As in languages like C, rules for the application of various operators when an R expression is *evaluated* have been established. These rules, applied for the evaluation of arithmetic expressions like above, are known as *precedence rules*. In general, they are used for the evaluation of other types of R expressions, as well.

   In an arithmetic expression, sub-expressions enclosed in parentheses (as well as expressions within a function call) are evaluated first. In each sub-expression, operators with higher precedence are applied first. If two operators in an expression have the same precedence, the one on the left is applied first. In the expression 3*(11.5 + 2.3) the sum of 11.5 and 2.3 is first calculated and the result is multiplied by 3. In 4.7 / sqrt(g + 2) - 6.4, square root of the value of g + 2 is first computed by applying the function sqrt(), 4.7 is divided by the result, and then 6.4 is subtracted. Note that the division operator has higher precedence than the subtraction operator. Below, more examples are provided to illustrate the evaluation of expressions further:

   ```
   > -8 * 7^2/3 + 2
   [1] -128.6667
   > -8 * 7^(2/3) + 2
   [1] -27.27445
   > -8 * 7^2 / (3 + 2)
   [1] -78.4
   > n <- 10
   > 1:n+1
   ```

```
 [1]  2  3  4  5  6  7  8  9 10 11
> 1:(n+1)
 [1]  1  2  3  4  5  6  7  8  9 10 11
```

In -8 * 7^2/3 + 2, the exponentiation operator ^, has the highest precedence, the unary minus operator – the next highest, the * and / operators have identical precedence but at a lower level, and the + operator has the lowest precedence. Thus 7 is first squared and multiplied by -8, the result is divided by 3 and 2 is added to get the final result. In -8 * 7^(2/3) + 2 the order of evaluation is changed by the use of parentheses. Thus 2/3 is evaluated first, and 7 is raised to that value, then multiplied by -8, and 2 added to get the final result. In -8 * 7^2 / (3 + 2), 3 and 2 are added first; thus the expression evaluates to a different result. In the expression 1:n+1, the sequence operator : has higher precedence than the + operator.

2. **Vectorized Computations:**
Some operators and functions are designed to work in an intuitive way when used with objects other than just scalars and constants. For example, when used on R objects such as vectors and matrices, the operator or function is applied to each individual element of the object. R is thus said to be capable of *vectorized* computation meaning that operations such as squaring or computing square roots can be performed on entire vectors rather than performing them one-at-a-time on individual elements of a vector. In the following example, exponentiation and division operators, and the log() and sqrt() functions, are applied to vector object h:

```
> h <- c(15.1, 11.3, 7, 9)
> h^2
[1] 228.01 127.69  49.00  81.00
> 1/h
[1] 0.06622517 0.08849558 0.14285714 0.11111111
> log(h)
[1] 2.714695 2.424803 1.945910 2.197225
> sqrt(h)
[1] 3.885872 3.361547 2.645751 3.000000
> sum(h^2)
[1] 485.7
```

Note that the sum() function is a built-in R function that works on various R objects to compute the sum of the elements of the object. It is computationally efficient to make sure user-written R functions are *vectorized.* More experienced R users take advantage of vectorized computation to avoid unnecessary loops, thus making their code more efficient.

3. **Coercion:**
All R objects have two *attributes:* their *mode* and *length.* For example:

```
> mode(h)
[1] "numeric"
```

```
> mode(actors)
[1] "character"
> length(h)
[1] 4
```

When objects with different attributes are entered in an expression, R converts the attributes of some of the objects to different ones so that the computation can be done in the best way possible. This is called coercion. The following example shows how R coerces a numeric value to a character value so that the resulting vector is of a common mode:

```
> c(1, 2)
[1] 1 2
> c(1, "A")
[1] "1" "A"
```

Consider the following example where vectors of different lengths are used in an arithmetic expression:

```
> hh <- c(h, 0, 0, 0, 0, h);hh
 [1] 15.1 11.3  7.0  9.0  0.0  0.0  0.0  0.0 15.1 11.3  7.0  9.0
> hhh <- 2*h + hh + 1
> hhh
 [1] 46.3 34.9 22.0 28.0 31.2 23.6 15.0 19.0 46.3 34.9 22.0 28.0
```

In the expression 2*h + hh + 1, the operation 2*h produces a vector of length 4. Then it is to be added to hh, a vector of length 12. To perform this operation, the vector resulting from the operation 2*h is replicated three times to match the length of hh. The result is a vector of length 12 that is then to be added to the constant 1 i.e, a vector of length 1. This is performed by replicating one 12 times to make a vector of length 12. Thus, this expression can be executed and a result is returned. This is possible only whe the longer vector is a multiple of the shorter vector in an operation involving two vectors. Thus, for example, the following does not work:

```
> hh <- c(h, 0, 0, 0, h);hh
 [1] 15.1 11.3  7.0  9.0  0.0  0.0  0.0 15.1 11.3  7.0  9.0
> hhh <- 2*h + hh + 1
Warning message:
In 2 * h + hh :
  longer object length is not a multiple of shorter object length
```

4. **Logical Expressions:**

Results of a logical expression are represented as the values TRUE or FALSE. Thus:

```
> h
[1] 15.1 11.3  7.0  9.0
```

```
> h < 9
[1] FALSE FALSE  TRUE FALSE
> sum(h < 9)
[1] 1
```

As seen in the above example, the comparison operator < is vectorized. It compares each of the vector h to the constant 9 and produces a TRUE or a FALSE value, thus the result is a vector. To enable arithmetic operations to be performed on logical vectors, values of 1 and 0 are assigned to T and F, respectively. Thus, the expression sum(h < 9) counts the number of values in the numeric vector h that are smaller than nine. Simple comparisons can be combined using logical operators such as && (AND) or || (OR). For example:

```
> x <- 3.5
> y<- -5.7
> x < 5 && y > -3
[1] FALSE
```

The && operator is typically used in if-else type conditional evaluations. For combining results of vector comparisons, the operator & must be used instead of &&. For example:

```
> c(1, 1) < 1:2
[1] FALSE  TRUE
> c(1, 1) < 1:2 && c(2, 2) > 2:3
[1] FALSE
> c(1, 1) < 1:2 & c(2, 2) > 2:3
[1] FALSE FALSE
```

Obviously, comparisons and arithmetic operators can appear in the same expression. Thus, they must obey to the precedence rules.

```
> x + y < -1
[1] TRUE
```

Because logical comparisons have lower precedence than arithmetic operators, the addition takes place first and the result, -2.2, is compared with -1.

5. **Built-in R Functions:**

Most computations involve *invoking* functions already built-in to R. R contains a large number of built-in functions. Invocation of an R function usually results in an R object, but others result in what are known as *side effects*, such as plotting or printing. Functions can be called directly in R expressions to obtain quantities needed for the computation, taking advantage of properties such as vectorization and coercion to avoid unnecessary intermediate computations. This is illustrated by the computation of the sample variance using the formula

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$$

In a lower level language using a loop is unavoidable; in R the function sum() can be made use of as long as the deviations can be calculated as a vector. This can be accomplished easily as the subtraction and exponentiation operations are both vectorized:

```
> attach(chickwts)
> weight
 [1] 179 160 136 227 217 168 108 124 143 140 309 229 181 141 260 203 148 169
[19] 213 257 244 271 243 230 248 327 329 250 193 271 316 267 199 171 158 248
[37] 423 340 392 339 341 226 320 295 334 322 297 318 325 257 303 315 380 153
[55] 263 242 206 344 258 368 390 379 260 404 318 352 359 216 222 283 332

> var(weight)
[1] 6095.503
> sum((weight - mean(weight))^2)/(length(weight)-1)
[1] 6095.503
```

Function calls have arguments some of which are *positional* implying that that function recognizes the argument by the sequential position in the *argument list* supplied to the function. Other arguments are of the form `name=value` where the name is called a *tag*. The first two arguments to the function seq(from, to, by=, length=) are positional. It also has two tagged arguments by= and length=.

```
> seq(1, 100, by=10)
 [1]  1 11 21 31 41 51 61 71 81 91
```

This function provides an example where some tagged arguments cannot be used together in the same call:

```
> seq(1, 100, by=10, length=5)
Error in seq.default(1, 100, by = 10, length = 5) : too many arguments

> seq(1, 100, length=5)
[1]   1.00  25.75  50.50  75.25 100.00
> seq(to=100, from=1, 10)
 [1]  1 11 21 31 41 51 61 71 81 91
```

In the previous example, we see that positional arguments may also be used as tagged arguments. Also note that positional arguments can be specified in any sequence in a function if they are used in the named form.

```
> seq(length=5, 1, 100)
[1]   1.00  25.75  50.50  75.25 100.00
```
Finally, in the last example we see that a named argument does not count as positional argument if it is used in the named form; i.e., in seq(length=5, 1, 100), 1 is still considered to be in the first position.

In many cases, not all arguments to a function need to be specified in the function call. This is because many functions have been designed to work with reasonable default values already assigned to several named arguments. The function specification for rnorm(n, mean=0, sd=1), for example, implies that, by default, a random sample of size n from the *standard normal distribution* will be generated:
```
> rnorm(10)
 [1] -0.6450606167 -0.7233606137  0.0006909495  0.6603013039  0.0833284466
 [6]  0.3599082968  2.1421453532  1.7201697519  1.0719041827  0.5791592045
> rnorm(10, 0, 1)
 [1]  0.1098768 -1.6877923  0.1202216 -1.4155483 -0.6503909  0.9144681
 [7]  0.5399951  1.1127428 -1.1599403 -0.4488810
> rnorm(10, mean=5)
 [1] 4.385616 6.861654 5.110625 4.239153 3.066983 3.280312 4.460029 3.705528
 [9] 3.282116 4.118984
> rnorm(10, 5, 2)
 [1] 8.775934 5.082372 2.674691 7.660589 3.826040 5.999559 4.269809 2.945798
 [9] 3.802781 2.402747
```

The argument `na.rm=` is an important argument in many statistical functions. The character NA is used in R to indicate missing values, i.e., values that are unavailable or unobserved. NA is actually a logical value like TRUE and FALSE, and calculation that involves NA returns an NA. Thus, many built-in functions provide users with the option of requesting the function to remove all NA's in the data prior to performing calculations. This is done by the use of the logical argument `na.rm=`. The default value is set to FALSE, meaning that NA's will not be removed. The statistical function median(x, na.rm=FALSE) serves as an example:

```
> scores <- c(35, 48, 31, NA, 27, 42, 50, 44, 36, 49, 33, 45, 28, 46)
> scores
 [1] 35 48 31 NA 27 42 50 44 36 49 33 45 28 46
> median(scores)
[1] NA
> median(scores, na.rm=T)
[1] 42
> quantile(scores, c(.1, .25, .5, .75, .9), na.rm=T)
 10%  25%  50%  75%  90%
28.6 33.0 42.0 46.0 48.8
> summary(scores)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
```

```
  27.00   33.00   42.00   39.54   46.00   50.00       1
```

Observe that summary() automatically omits the NA's. The function summary() is an example of a *generic function* that operates on objects to produce summaries of the results of various functions. It invokes a particular *method* which depend on the *class* of the first argument.

Functions such as floor() and ceiling() are provided in R for performing truncating numerical values to nearest whole numbers (integers), the functions round() and signif() are important for controlling the accuracy of printed output resulting from calculations rather than relying on R to determine the accuracy for you. Note that the function options() can be used to set the number of digits printed globally. These functions are illustrated through some examples:

```
> 1000/7
[1] 142.8571
> options(digits=10)
> 1000/7
[1] 142.8571429
> round(1000/7, digits=5)
[1] 142.85714
> signif(1000/7, digits=5)
[1] 142.86
```

While most mathematical, statistical, and matrix manipulation functions are commonly recognizable, functions such as those above, although extremely useful, are not familiar enough to many novice R programmers. Thus, some may attempt to rewrite code for tasks for which built-in functions are already available or can be adapted. Some examples of useful functions are:

```
> x <- 1:10
> y <- c(95, 100, 101, 115, 124)
> append(x, y)
 [1]   1   2   3   4   5   6   7   8   9  10  95 100 101 115 124
> append(x, y, after=5)
 [1]   1   2   3   4   5  95 100 101 115 124   6   7   8   9  10
> x <- replace(x, c(3:6), 0)
> x
 [1]  1  2  0  0  0  0  7  8  9 10
> xx <- c(6.1, 3.5, 4.0, 7.4, 9.2, 1.7)
> rank(xx)
[1] 4 2 3 5 6 1
> sort(xx)
[1] 1.7 3.5 4.0 6.1 7.4 9.2
> i <- order(xx);i
[1] 6 2 3 1 4 5
```

```
> xx[i]
[1] 1.7 3.5 4.0 6.1 7.4 9.2
```