Lab 1: R Basics

STAT218

This lab is intended to introduce you to the nuts and bolts of R: arithmetic operations, data types, vectors, and other data structures. While you won't be expected to do much data manipulation or manual calculation for this class, it will be helpful for you to have some understanding of this material going forward, as most R objects are arranged as data structures comprising one or more data types.

This lab covers a lot of ground; consider it a reference you can return to later as needed.

Arithmetic operations

While R does a lot more than function as a calculator, it can be used to perform basic arithmetic. All of the usual operations — addition, subtraction, multiplication, division, exponentiation — can be performed as shown below.

```
# addition
2 + 1

# subtraction
2 - 1

# multiplication
2*3

# division
2/3

# exponentiation
3^2

# parentheses for order of operations: compare
(2 + 1)/4
```

```
2 + 1/4
```

Basic functions can also be evaluated. To name a few:

```
# square root
sqrt(9)

# exponential function e^x
exp(1)

# logarithm: compare, default base is e (i.e., natural log)
log(2, base = exp(1))
log(2)
```

The result of each of these calculations is a *value*. Values are the most elementary objects in the R environment. Values can be assigned names using the assignment operator `<-`:

```
# store a value
x <- log(2)

# print value in console
x
```

Names can consist of any combination of letters, numbers, and the delimiters `_` and `.`; names cannot start with numbers (try it and see what happens).

If a name is used multiple times, the most recently assigned value will be stored. However, 'most recent' in this context means 'most recently executed': R will not pay attention to the order of lines in your script, but the order in which you run them. To see this, try redefining `x` to be a new value. if you then go back and run the previous line that displays the value, you'll notice you get the new value, not the old one, even though the line appears before the new assignment.

```
# assign a new value the name 'x': this overwrites the first value
x <- 4
```

Your turn

Write one line of code each to perform the following calculations:

1. Calculate the sum of the ages of each member of your group, in years.
2. How about in months? Multiply by 12.

3. Calculate the average number of siblings among the members of your group.

```
# sum of ages of group members

# convert to months: multiply by 12

# average number of siblings among group members
```

Data types

Values may be of different types. The function `class(...)` will return the "class" of object it is given; for values, the object class is simply the data type.

There are four main data types in R.

```
# numeric
class(12)

# character
class('text')

# logical
class(TRUE)

# integer
class(12L)
```

Arithmetic only works with logical, integer, and numeric values:

```
# valid operations
12*12
12L*12
12L*12L
12*TRUE
12*FALSE
12L*TRUE
12L*FALSE
```

It will not work with character values. Notice the text of the error message:

```
# invalid operation
'12'*'12'
```

```
Error in "12" * "12": non-numeric argument to binary operator
```

This is a very common error message — when you see it, you'll know that most likely, somewhere R attempted to perform arithmetic with character values, so it's probably a data type issue.

> **Your turn**
>
> 1. Calculate the product of TRUE and FALSE. Which data type results from this operation?
> 2. What do you think will happen if you *add* a logical and numeric value? Discuss first, then verify.
> 3. What kind of value does each of the above operations return? Discuss first, then verify.
>
> ```
> # product of true and false
>
> # add a logical and numeric value
>
> # check data type of the above
> ```

Vectors

In practice, data usually consists of multiple values, not just one; in R, the most basic type of collection of values is called a *vector*. Technically, a vector is a "concatenation" of values of the same data type. Vectors can be formed using the concatenation function `c(...)`:

```
# concatenate values with c(...) to form a vector
c(1, 2, 3)

# store it
x <- c(1, 2, 3)

# data type
class(x)
```

If values of different types are concatenated, they are coerced to the same data type; vectors cannot contain values of mixed type. The lines below demonstrate this behavior:

```
# integer and logical -> integer
class(c(2L, TRUE))

# integer and numeric -> numeric
class(c(2L, 12))

# numeric and logical -> numeric
class(c(12, TRUE))

# character and anything -> character
class(c('text', TRUE))
class(c('text', 2L))
class(c('text', 12))
```

Vectorization

Arithmetic between vectors is carried out elementwise; arithmetic operations between vectors and values are 'vectorized', meaning operations are applied over each element. This makes certain calculations quite efficient. The following examples illustrate vectorized arithmetic operations in two verbose ways: first as an elementwise operation between two vectors; then as a concatenation of the results of each elementwise operation.

```
# equivalent
2*c(1, 2, 3) # vectorized
c(2, 2, 2)*c(1, 2, 3) # verbose
c(2*1, 2*2, 2*3) # verbose

# equivalent
c(1, 2, 3) + 1 # vectorized
c(1, 2, 3) + c(1, 1, 1) # verbose
c(1 + 1, 2 + 1, 3 + 1) # verbose

# equivalent
c(1, 2, 3)/3 # vectorized
c(1, 2, 3)/c(3, 3, 3) # verbose
c(1/3, 2/3, 3/3) # verbose

# equivalent
```

```r
c(1, 2, 3)^2 # vectorized
c(1, 2, 3)^c(2, 2, 2) # verbose
c(1^2, 2^2, 3^2) # verbose
```

If an operation is carried out between vectors of different lengths, a warning is printed indicating as much.

```r
# can you figure out what calculation was performed?
c(1, 2, 3)*c(4, 5)
```

```
Warning in c(1, 2, 3) * c(4, 5): longer object length is not a multiple of
shorter object length
```

```
[1]  4 10 12
```

Many functions, including those mentioned at the outset, are also vectorized (calculations are performed elementwise).

```r
sqrt(c(1, 4, 9))
log(c(1, 10, 100), base = 10)
exp(c(0, 1, 2))
```

Indexing

Elements of vectors are *indexed* by consecutive integers; elements can be retrieved by specifying the indices in square brackets after the object name.

```r
# define a vector
x <- c(10, 20, 30, 40)

# second element
x[2]

# first and third elements
x[c(1, 3)]

# second through fourth elements
x[2:4]
```

Missing values

Many datasets have missing values that occur for various reasons: equipment failure, participant dropout, survey nonresponse, and so on. These can be represented in R as well. This is useful as a means of retaining the information that there is an observation that was somehow lost. Missing values are displayed as the special character `NA` in R.

```
# vector with a missing third element
c(1, 2, NA, 4)

# note: still numeric
class(c(1, 2, NA, 4))
```

Your turn

1. Create a vector with the ages of each person in your group (in years). Call it `ages`.
2. Convert to months using vectorized arithmetic.
3. Now pretend one of you is absent; repeat (1) but replace their age by a missing value. call the vector `ages_incomplete`.
4. Convert to months again using the same operation as in (2); how is the missing value handled?

```
# vector of ages in years

# convert to months with vectorized arithmetic

# pretend one person is absent: input a missing value

# repeat conversion to months: how is the NA handled?
```

Matrices, arrays, lists, and data frames

Matrices can be constructed from vectors by specifying row and column dimensions, and how to arrange the values – by row or by column.

```
x <- c(1, 2, 3, 4)
matrix(data = x, nrow = 2, ncol = 2, byrow = TRUE)
```

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

```
[1,]    1    2
[2,]    3    4
```

We won't really see arrays much, but since they are mentioned in the reading, here's an example of how to construct one, and how it is displayed.

```
x <- c(1, 2, 3, 4, 5, 6, 7, 8)
array(data = x, dim = c(2, 2, 2))
```

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

A list is a highly general data structure; it's just an indexed amalgamation of objects of any type.

```
# make a list
list('anchor', c(22, 5), log)
```

```
[[1]]
[1] "anchor"

[[2]]
[1] 22  5

[[3]]
function (x, base = exp(1))  .Primitive("log")
```

```
# the list elements can be named, which makes for easy retrieval
my_list <- list(words = 'anchor', numbers = c(22, 5), functions = log)
my_list$words
```

```
[1] "anchor"
```

```
# even so, indexing can still be used
my_list[[1]]
```

```
[1] "anchor"
```

Data frames are like lists, except each element is a vector of the same length; they appear much like matrices, but behave differently in important ways.

```
# make a data frame
data.frame(col1 = c(1, 2, 3, 4),
           col2 = c(T, F, T, T),
           col3 = c('red', 'blue', 'green', 'yellow'))
```

```
  col1  col2   col3
1    1  TRUE    red
2    2 FALSE   blue
3    3  TRUE  green
4    4  TRUE yellow
```

```
# unlike matrices, however, columns can be retrieved by name
my_df <- data.frame(number = c(1, 2, 3, 4),
                    truth = c(T, F, T, T),
                    color = c('red', 'blue', 'green', 'yellow'))

# the result is a vector containing the values of that column
my_df$color
```

```
[1] "red"    "blue"   "green"  "yellow"
```

Data frames are the fundamental data structure on which a large portion of R software is built. When data files are read into R, as you saw very briefly earlier, they are read in by default as data frames. Many R functions used in performing data analyses require data to be supplied as a data frame.

Your turn

Create a data frame in which each row corresponds to one member of your group and columns are the variables age, number of siblings, and favorite singer.

```
# data frame with columns age, no. of siblings, and favorite singer; each row correspond
```