```r
# This is a comment; it is ignored by R.

# The greater than sign, >, in the console is the command prompt;
# this is where we enter a command we want R to execute. In a .R
# script file we do not include the command prompt; however, when
# we click the Run button, the command is sent to the R console
# at the open greater than symbol and executed.

1 + 3

# Returning the answer of 4 in the console. The number that
# appears in brackets before the result of the calculation is the
# index of the first value returned on that line, in this case 1.
# We will see soon how to make use of this bracket and index. It
# is useful to assign the result of a calculation to a named
# object so that we can access it later. As we see here, names
# are case-sensitive.

answer = 1 + 3

Answer = 2 + 4

# Note that when we assign the result of a calculation to an
# object, the result of the calculation is not returned to the
# console. Storing data in objects allows us to use the data in
# calculations by using the object's name.

answer + Answer

# There are a variety of objects that we can use to store data,
# of which we will consider four: vectors, data frames,
# matrices, and lists. A vector is an ordered collection of
# elements all of the same type. Here are three vectors: a
# numeric vector, a logical vector, and a character vector; these
# vectors use the concatenate function c() to create the vector.

vector01 = c(1, 2, 3)

vector02 = c(TRUE, TRUE, FALSE)

vector03 = c("alpha", "bravo", "charley")

# Oops! the third element in vector03, which is the first three
# characters in NATO's phonetic alphabet, is misspelled; we can
# correct this value by using the bracket symbol and the value's
# index to identify the element we wish to change.

vector03[3] = "charlie"

# We can create a new vector by combining existing vectors using
```

```
# the concatenate function; note that the elements of a vector
# must be of the same type, so here R converts all elements to
# characters.

vector04 = c(vector01, vector02, vector03)

# The sequence function, seq(), makes it easy to create numeric
# vectors with patterned values. The sequence function has three
# arguments: from, which is the first value in the sequence; to,
# which is the last value in the sequence; and by, which is the
# difference between each successive value. Note that if we do
# not use the names of the arguments to the function, then R
# assumes that the order is from, to, and by. Note, as well, that
# x:y is the same as seq(x, y, 1).

vector05 = seq(from = 0, to = 20, by = 4)

vector06 = seq(0, 10, 2)

vector07 = 1:10

# The repeat function, rep(), creates a vector by repeating a set
# of values a specified number of time. Note the difference
# between times and each.

vector08 = rep(x = 1:4, times = 2)

vector09 = rep(1:4, each = 2)

# We can identify which elements in a vector matches a specified
# requirement. Here we use the exactly equals operator, ==, to
# determine which elements of vector08 have a value of 2 and
# which do not; the which() function to determine which elements
# of vector08 have a value less than 3; and the not operator, !,
# to determine which elements of vector08 have values that are
# not less than 3. Note that the == and the ! operators return
# logical values for all of the vector's elements, but that the
# which() function returns just the index values for the elements
# of vector08 that are less than 3.

vector08 == 2

which(vector08 < 3)

!vector08 < 3

# A data frame is a collection of vectors of equal length that
# need not be of a single type of element, it is created using
# the data.frame() function. Look carefully at how we entered the
# third vector: when we subtract one vector from another vector,
```

```
# the subtraction is carried out element-by-element. Note that a
# data frame is like a spreadsheet where each column is a vector.

dataframe01 = data.frame(vector08, vector09,
                         vector10 = vector09 - vector08)

View(dataframe01)

# We can extract elements from a data frame by identify the cells
# of interest using the bracket notation, [rows, columns], to
# identify the desired row(s) and column(s). These three examples
# return all values in the first row, all values in the second
# and the third columns, and the single element in the fourth row
# and the third column.

dataframe01[1, ]

dataframe01[ , 2:3]

dataframe01[4, 3]

# A matrix is similar to a data frame, but all elements in a
# matrix must have the same type. Here are two examples of how to
# make a matrix with 10 values, the first a 5x2 matrix (five rows
# and two columns) and the second a 2x5 (two rows and five
# columns) matrix.

matrix01 = matrix(1:10, nrow = 5)

matrix02 = matrix(1:10, ncol = 5)

View(matrix01)

View(matrix02)

# We can complete calculations using matrices, although the
# details of how this is done are not always obvious; we will
# explore this in more detail in the last third of the course;
# for now, here is an example of matrix multiplication.

matrix01 %*% matrix02

# A list is similar to a vector, but its elements are other
# objects, which can be of very different types.

list01 = list(vector01, dataframe01, matrix01)

# The bracket notation for extracting elements from a vector or a
# data frame works here as well, but we use double brackets,
# [[]], to identify the one of the list's objects and we use
```

```r
# single brackets, [], to identify element's with that object.

list01[[2]]

list01[[3]][4, 2]

# Although we can create complex data sets using the commands
# highlighted above, it usually is easier to read in data from
# .csv file created using Excel, or by reading in data saved
# during an earlier session as a .RData file. To do this we need
# to know the pathname to the file. The simplest approach is to
# do our work in the same directory where the files we need are
# stored by either (a) choosing Session: Set Working Directory
# from RStudio's main menu and navigating to the desired folder,
# or by (b) selecting More: Set as Working Directory from the
# Files tab from the Files, Plots, Packages... panel. Both
# approaches are equivalent to entering the following command.

setwd("~/Box Sync/p-harvey/Teaching/Chem 351/Class Units/
01.Course_Introduction")

# Assuming that we are in the working directory, we can use the
# load() function to read in a .RDaa file and the read.csv()
# function to read in a .csv file. For the former, the objects
# are added directly to the our workspace; for the latter, we
# have to assign the data to an object so that it is available to
# us.

load(file = "BeefLiver.RData")

elements = read.csv(file = "ElementData.csv")

# We also can load files without worrying about identifying the
# working directory by using the file.choose() function within
# load() or read.csv()

elements.new = read.csv(file.choose())

# The source() function reads in a script file and runs it. The
# script in the file sampleScript.R, for example, creates four
# plots of the variables x1, x2, y1, and y2 where x1 and x2 each
# contain 1000 values from a random uniform distribution between
# 0 and 1, and y1 and y2 each contain 1000 values from a random
# normal distribution with a mean of 0 and a standard deviation
# of 1.

source(file = "sampleScript.R")

# Saving your work is important; you can save your data to an
# .RData file or to a .csv file using the save() or the
```

```
# write.csv() functions.

save(PBconc, elements, file = "saved.RData")

write.csv(elements, file = "saved.cvs")

# Finally, you can find help on how to use a function by passing
# the function's name to the help() function.

help(rnorm)
```