Moving from IDK to IDE

Getting Started with R for Laboratory Medicine
Sunday Aug 04, 2019
AACC 2019, Anaheim CA

Daniel T. Holmes, MD (dtholmes@mail.ubc.ca)
Department of Pathology and Laboratory Medicine, University of British Columbia
Stephen R Master, MD PhD (masters@email.chop.edu)
Department of Pathology and Laboratory Medicine, Children's Hospital of Philadelphia

Contents

0.1 Preparation for the Course

This Short Course requires you to bring your own laptop. Plugs will be provided on the tables so that you do not have to rely on your battery life. You must install the necessary software before you arrive. It can take up to 30 mins and you do not want to be stuck doing this in the session because you get behind.

You need to install both the R programming Language and the R-Studio interface to R. While one can use R without R-Studio, we will all use it to make things uniform. The R programming language is available for download from many different places. Here are three places in Canada, http://cran.stat.sfu.ca/; http://cran.utstat.utoronto.ca/; https://mirror.its.dal.ca/cran/. Choose the download that is appropriate for your laptop (whether Linux, Mac or Windows). Rstudio can be downloaded and installed free for personal use from: https://www.rstudio.com/products/RStudio/. You want the Open Source Edition.

Throughout this handout, we'll be showing you R code in a shaded box like this

```
R code goes here
```

and output will come be in an unshaded area after two "#" signs

```
## [1] "Here is some R output"
```

1 Lesson 1: Basics and Data Types

1.1 What is R?

The purpose of this course is to show you that many of the data management and data analysis task you *wish* you could do in Excel are, in fact, very easy if you move to the statistical programming language R.

R is what is known as a "scripting language". This means that R does not build portable, stand–alone programs like commercial software we usually purchase. In contrast, R requires the R interpreter to be pre-installed on the computer before we can use any of its utilities. The same is true of other popular scripting languages like Python, PHP, Ruby and JavaScript. In principle this means that if you are using R on a company laptop and you don't have administrative privileges, you are going to need IT to help you install it.

R has been built for Windows, Linux and Mac, which means that R code you write can be run by other people on other systems *without any alterations to the code*, provided that they have installed the R interpreter. There are occasional exceptions to this rule but any code alterations are–most often–trivial.

The programs you write in R are saved as text or "ASCII" files that you save like any old document. You can use any text editor to write your program. This might include programs like Notepad or Notepad++ in the Windows environment

or Gedit, eMacs, vi, Nano or Sublime Text in the Linux/Mac environments. In this course we will use a text editor in an environment specifically built for R called RStudio. This program has the benefit that it can both allow you to edit your text and run it without leaving the RStudio program. RStudio has many other convenient features that you will discover should you choose to continue this jou–R–ney. We could spend a lot of time on the background, but we do not have a great deal of time together so we should just jump right in.

1.2 Algebra

R can act as a calculator. It follows these basic rules or algebra.

Operation	Expression	R Code
Addition	$x + y$	`x + y`
Subtraction	$x - y$	`x - y`
Multiplication	$x \times y$	`x * y`
Division	$x \div y$	`x / y`
Exponents	x^y	`x^y` or `x**y`
Logarithm	$\log(x)$	`log(x)`
Exponential	e^x	`exp(x)`
Trig Functions	$\sin(x)$	`sin(x)`

There are some other useful built-in functions in R. These are:

- `abs(x)`
- `sqrt(x)`
- `floor(x)`
- `round(x, digits = n)`
- `signif(x, digits = n)`

1.2.1 Exercise

1. Experimentation is a great way to find out what a function does. Determine the square root of 64 with the `sqrt()` function and using the fact that the square root of 64 is really $64^{1/2}$
2. Try the following:
   - `ceiling(1.2)`
   - `ceiling(1.49)`
   - `ceiling(1)` What does ceiling do?
3. Try the following:
   - `round(1.4503,1)`
   - `round(exp(1),4)`
   - `round(pi,4)`

- Now try typing `round(12314,-1)` and `round(12314,-2)`. What's happening here?

4. What does `trunc(5.99)` give you? What does `trunc(-3.43)` give you? How is `trunc()` different than `floor()`?

1.3 Comments

Note that anything we type on an R line that comes after the `#` sign is ignored by R. This is very useful for including comments in your code and helps to remind you (and others) what you were thinking. Because anything after the `#` is ignored, we can either put a comment on its own line:

```
# here is a comment
```

or after some code that is going to be executed

```
2 + 2  # we had better get the answer "4"
```

```
## [1] 4
```

1.4 Variables

Variables allow you to store your data so that it can be easily retrieved at a later time. For example, suppose you calculated the standard deviation of a data set and you had to use this result over and over again (and you did not want to type it our each time!). The best way to reuse this value is to store it in a variable.

```
my.sd <- 0.352
my.sd
```

```
## [1] 0.352
```

```
1.96 * my.sd
```

```
## [1] 0.68992
```

Notice that assigning the variable is performed with an "arrow" <- . The arrow can actually go the other way too but we don't do that all too much.

```
a <- 5 -> b
a
```

```
## [1] 5
```

```
b
```

```
## [1] 5
```

Getting rid of a variable is sometimes convenient, and the way to do this is with the `rm()` function.

```
rm(a)
a
```

```
## Error in eval(expr, envir, enclos): object 'a' not found
```

To list all active variables type `ls()`. To remove all active variables type `rm(list = ls)`. This can also be achieved by clicking the broom icon in the **Environment** tab of the top right pane of Rstudio.

1.5 Data Types

We will encounter a variety of different data types during this course, and each has specific applications.

```
var.1 <- TRUE # a logical variable
class(var.1)
```

```
## [1] "logical"
```

```
var.2 <- 32.5 # a numeric variable
class(var.2)
```

```
## [1] "numeric"
```

```
var.3 <- "Michael" # a character variable
class(var.3)
```

```
## [1] "character"
```

Also, we can make integer variables and factor variables. R guesses which you want, and if it guesses wrong you may need to force it to assume the correct data type. You will see this later on. Don't assume that R has correctly read your mind on the data type you wanted.

```
var.4 <- 5
class(var.4)
```

```
## [1] "numeric"
```

```
var.4 <- as.integer(var.4)  # coerces the value to the class of integer
class(var.4)
```

```
## [1] "integer"
```

```
var.5 <- "4"
class(var.5)
```

```
## [1] "character"
```

```
7 * var.5                   # what happened?
```

```
## Error in 7 * var.5: non-numeric argument to binary operator
```

```r
var.5 <- as.numeric(var.5)  # coerces the character into a numeric
class(var.5)
```

```
## [1] "numeric"
```

```r
7 * var.5                   # ahhh no error now.
```

```
## [1] 28
```

1.6 Vectors

Vectors are a way to store multiple data points in a single variable. They are like a column from an Excel file and we will use them a lot. To define a vector you have to use the combine `c()` function to group the data.

```r
x <- c(5,3,6,4,7,2,6) # defines the variable x
class(x)              # what class will this be?
```

```
## [1] "numeric"
```

Importantly, every member of the vector must be of the same type and if they are not, R will choose a data type that will be compatible with all the elements.

```r
y <- c(5,3,6,4,7,2,"Hi There")
class(y)              # What happened?
```

```
## [1] "character"
```

```r
y
```

```
## [1] "5"        "3"        "6"        "4"        "7"        "2"
## [7] "Hi There"
```

Let's explore some algebra:

```r
x + 2   # What does it do?
```

```
## [1] 7 5 8 6 9 4 8
```

```r
x + x       # How is this different from x+2
```

```
## [1] 10  6 12  8 14  4 12
```

```r
x / 2       # What does this do?
```

```
## [1] 2.5 1.5 3.0 2.0 3.5 1.0 3.0
```

```r
x * x       # What does this do?
```

```
## [1] 25  9 36 16 49  4 36
```

```r
x / x       # What does this tell you about dividing vectors?
```

```
## [1] 1 1 1 1 1 1 1
```

```
sum(x)  # What does this calculate?
```

```
## [1] 33
```

```
mean(x) # And this?
```

```
## [1] 4.714286
```

```
sd(x)       # And this?
```

```
## [1] 1.799471
```

```
length(x)   # And this? This is really useful.
```

```
## [1] 7
```

```
c(x, x)
```

```
##  [1] 5 3 6 4 7 2 6 5 3 6 4 7 2 6
```

```
rep(x,3)
```

```
##  [1] 5 3 6 4 7 2 6 5 3 6 4 7 2 6 5 3 6 4 7 2 6
```

What if we wanted to find out an individual value from x?

```
x[2]  # Note the SQUARE brackets.
```

```
## [1] 3
```

```
# OK, makes sense
```

What if we wanted to know which value of x was 6, if any?

```
which(x  ==  6)
```

```
## [1] 3 7
```

CAREFUL "==" is used to compare two values to see if they are equal; don't confuse this with and "=" or "<-", which are for assignment.

```
x[which(x  ==  6)]    # should give us back a number of 6's of course.
```

```
## [1] 6 6
```

```
x  ==  6    #this gives us a logical vector answering the question "Is the value 6?"
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

```
y <- x  ==  6
y
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

```
x[y]    #this should just give us back the 6's
```

```
## [1] 6 6
```

```
x[x  ==  6]
```

```
## [1] 6 6
```

1.6.1 Exercise

1. Define a variable, `days`, which contains all the days of the week.
2. Now imagine that you move to a planet where there are an 8th and 9th day of the week, called "Chillday" and "Sleepday". Can you use `c()` to add these days to your days variable so that you do not have to retype everything?

1.6.2 Exercise

1. Type `1:10` and see what happens.
2. Now type `x <- 1:10`. What did this do? Find out by asking R what `x` is.
3. Now type `x <- 5:10`. What did this do?
4. Type ?seq in the console to find out what the `seq()` function does. Can you replicate the results of `x <- 5:10` with `seq()`? Why is `seq()` more flexible?

1.6.3 Exercise

1. Write an expression in R that will always return the last value in a vector named `z`.
2. Test your expression's success. We'll start by generating a sequence of letters of the alphabet that terminates randomly.

```
z <- letters[1:round(runif(1,0,26),0)]  # don't worry about why this works right now.
z
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r"
```

```
z[length(z)]                            # check what the last value is
```

```
## [1] "r"
```

Now apply your expression to identify the last letter in the sequence you generated.

2 Lesson 2: Matrices, Dataframes and Lists

2.1 Matrices

A matrix is a 2D analogue of the vector. Matrices may not seem all that important but there are certain R statistical functions requiring their use and so you are certainly going to encounter them.

```
days <- 1:28  # Generates a sequence of 28 integers
days          # confirm that you have done what you thought
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28
```

```
# Note: another way to do this is with days <- seq(from = 1, to = 28, by = 1)
```

Now, let's convert this to something that looks more like the calendar layout of February.

```
matrix(days, nrow = 4, ncol = 7)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    5    9   13   17   21   25
## [2,]    2    6   10   14   18   22   26
## [3,]    3    7   11   15   19   23   27
## [4,]    4    8   12   16   20   24   28
```

```
# Huh? What happened?
#
# Let's try again
matrix(days, nrow = 4, ncol = 7, byrow = TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    2    3    4    5    6    7
## [2,]    8    9   10   11   12   13   14
## [3,]   15   16   17   18   19   20   21
## [4,]   22   23   24   25   26   27   28
```

```
# Ahhh
```

2.1.1 Exercise

- February 2016 started on a Monday but was a leap-year. Make a matrix for February 2016 and fill days that are not in February with NA. We'll talk later about different places that NA pops up, but for now just know that it's R's way of denoting data that is "**N**ot **A**vailable". Hint for this exercise: start by using the `c()` command to prepend the appropriate

number of NA's to the beginning and the end of variable days. Assign the name feb.2016 to your matrix.

To get a specific value of your matrix, you simply refer to the row and column.

```
feb.2016[3, 4] #Gives entry from row 3 column 4.  Should return 17 as the result.
```

```
## [1] 17
```

```
# To get all the values in the column, you simply omit the row number.
feb.2016[ ,6] # Gives us what we need: all the Saturdays.
```

```
## [1]  5 12 19 26 NA
```

```
# ...and the same trick works for columns
feb.2016[3, ] # Gives us week 3.
```

```
## [1] 14 15 16 17 18 19 20
```

2.1.2 Exercise

The raw speed data from your latest bike ride can be exported from your Garmin and brought into R. The ride happens to be exactly 38 mins duration and data is sampled every second. This means that there will be 2280 measurements in total. Execute this code to import the data from a file.

```
speed.data <- read.csv("Data_Files/speed.csv")
speed.data <- speed.data[,1]
```

- Now, convert this data into a matrix of 38 columns where each column is the speed data of one minute.

2.2 Data Frames

Data frames are the closest thing you are going to get to Excel-like storage of your data. When you read your data into R from a file (if you have saved it from Excel in a standard format), it will become a data frame.

Let's convert our February 2016 matrix to a data frame, and then we will look at how to import Excel-like data to a data frame. We will spend some time working with data frames because they are going to be our bread and butter.

First, we can convert a matrix to a data frame:

```
feb.2016 <- as.data.frame(feb.2016)
feb.2016
```

```
##   V1 V2 V3 V4 V5 V6 V7
## 1 NA  1  2  3  4  5  6
## 2  7  8  9 10 11 12 13
```

```
## 3 14 15 16 17 18 19 20
## 4 21 22 23 24 25 26 27
## 5 28 29 NA NA NA NA NA
```

```
# this has uninformative column names, but we can name the columns as we can in Excel
names(feb.2016) <- c("Sun","Mon","Tue","Wed","Thu","Fri","Sat")
feb.2016
```

```
##   Sun Mon Tue Wed Thu Fri Sat
## 1  NA   1   2   3   4   5   6
## 2   7   8   9  10  11  12  13
## 3  14  15  16  17  18  19  20
## 4  21  22  23  24  25  26  27
## 5  28  29  NA  NA  NA  NA  NA
```

```
# much better
```

Second, we can build a data frame from vectors as follows.

```
odd.nums <- c(1,3,5,7)
even.nums <- c(2,4,6,8)
numbers <- data.frame(odds = odd.nums, evens = even.nums)
numbers
```

```
##   odds evens
## 1    1     2
## 2    3     4
## 3    5     6
## 4    7     8
```

```
# alternatively, numbers <- as.data.frame(rbind(odds,evens))
```

So, the columns of a data frame could be the results of different data fields in your study, name, health number, sex, age, date of last visit, blood pressure, medications, creatinine, hemoglobin etc. Whatever you could store in an Excel sheet could be in a dataframe. We often want to pull out specific columns from a data frame–usually to perform statistical tests.

Pulling out a column is easy. We can do it by the column name or we can do it by the column number.

```
# by column name
feb.2016$Tue # gives all the Tuesdays in Feb 2016
```

```
## [1]  2  9 16 23 NA
```

```
# by column number
feb.2016[,3] # gives all rows of the third column, which is the same
```

```
## [1]  2  9 16 23 NA
```

```
# You can pull out data from an individual cell.
feb.2016[2,3]
```

```
## [1] 9
```

```
# Or you can do the same with the $ approach because feb.2016$Tue is a vector
feb.2016$Tue[2]
```

```
## [1] 9
```

If you need to pull more than one column out, you can do so by numbers or the names:

```
# by number
feb.2016[,3:4]
```

```
##   Tue Wed
## 1   2   3
## 2   9  10
## 3  16  17
## 4  23  24
## 5  NA  NA
```

```
#by name - this is particularly convenient when dealing with large dataframes
feb.2016[,c("Tue","Wed")]
```

```
##   Tue Wed
## 1   2   3
## 2   9  10
## 3  16  17
## 4  23  24
## 5  NA  NA
```

Let's go back to that bike speed data we have above. We will start by turning it into a data frame.

```
speed.frame <- as.data.frame(speed.mat)
names(speed.frame ) <- paste0("min_",1:38)
#head(speed.frame)
```

2.2.1 Exercise

- Using the speed.frame dataframe, find the average speed of the 20th minute of your ride.

Note that if you try to find the average by row instead of a column, you will run into a problem that illustrates something about data frames, namely that if you extract a row, you can't just simply do algebra on it because there is generally no guarantee that different columns of a data frame will all be numbers.

```
min.start.avg <- mean(speed.frame[1,]) #does not work
min.start.avg <- mean(speed.mat[1,]) #does work
```

Here are some things you can do with data in a data frame. You can ask R to tell you things about your data points. For example, you could ask R for some descriptive statistics of the last minute of your ride:

```
mean(speed.frame$min_38)
```

```
## [1] 25.15944
```

```
median(speed.frame$min_38)
```

```
## [1] 23.103
```

```
sd(speed.frame$min_38)
```

```
## [1] 4.482852
```

```
summary(speed.frame$min_38) # a statistical summary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   21.50   22.52   23.10   25.16   26.81   38.38
```

```
quantile(speed.frame$min_38 ,probs = c(0.25,0.50, 0.75)) # specific quantiles
```

```
##     25%     50%     75%
## 22.5216 23.1030 26.8137
```

But this does not work when you try to take the grand mean:

```
mean(speed.frame)
```

```
## [1] NA
```

But this approach does work for matrices

```
mean(speed.mat)
```

```
## [1] 30.9961
```

Remember, we'll say more later about "NA", but for now notice that R gives us a nasty warning message. The reason that the `mean` function does not operate on data frames in this case because the data in a data frame is often not numeric.

You can also ask R to tell you which values have certain properties.

```
# did you dip below 25 kph in the last minute of your ride?
speed.frame$min_38 < 25
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
## [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
## [34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [56]  TRUE  TRUE  TRUE  TRUE  TRUE
```

But the code authors acknowledge that it is a pain to have to write out those `$` signs all the time, so try this:

```
attach(speed.frame)
min_1 # now min_1 is a local variable
```

```
##  [1] 14.1984 17.2188 18.8964 20.0448 21.3732 22.4172 22.9752 23.2992
##  [9] 23.3424 23.1228 22.6908 22.2876 21.9456 21.6648 21.3768 21.1140
## [17] 20.9412 20.7648 20.6244 20.5056 20.1960 19.9080 19.3608 18.9612
## [25] 18.8460 18.8424 18.9324 19.1232 19.2600 19.5696 20.3868 20.0412
## [33] 19.2276 18.0828 16.9488 16.2684 15.5304 15.0048 14.7204 14.3928
## [41] 14.3172 14.5800 15.1416 15.1884 12.4380 12.3840 12.8736 13.3128
## [49] 13.6764 14.0724 14.4720 14.8104 15.1596 15.4800 15.6384 15.8400
## [57] 16.0596 16.2900 16.4952 16.6860
```

```
min_2 # ...and so is min_2!
```

```
##  [1] 17.0064 17.4456 17.8488 18.1260 18.3996 18.6264 18.9576 19.4436
##  [9] 18.9756 19.3752 19.8648 20.4372 21.0564 21.6972 22.6152 23.4756
## [17] 24.1200 24.5772 24.8796 25.1820 25.5852 25.9020 26.1864 26.4996
## [25] 26.8560 27.2232 27.6732 28.2204 28.7856 29.2536 29.6244 29.9376
## [33] 30.2544 30.5064 30.6576 30.8412 31.0320 31.2156 31.3128 31.3488
## [41] 31.3164 31.3200 31.3452 31.4280 31.4712 31.3632 31.3020 31.2048
## [49] 31.0608 30.9024 30.7404 30.5892 30.3804 30.0780 29.5488 28.7388
## [57] 28.0620 28.0980 28.4004 28.5876
```

Note that if you alter the attached variables, they *do not* alter the original data frame. To remove these variables we type:

```
detach(speed.frame)
```

There are lots of reasons not to use `attach`, but it is good for quick and dirty things.

2.2.2 Exercise

- We are going to read in a little more data from your bike ride. This time the data will contain time, cadence, heart rate, distance, speed and power.

```
ride.data <- read.csv("Data_Files/ride_file.csv")
head(ride.data)
```

```
##   secs cad hr      km     kph watts
## 1    1   1 80 0.00347 14.1984   360
## 2    2  75 81 0.00788 17.2188   360
```

```
## 3    3  77 83 0.01290 18.8964   255
## 4    4  91 84 0.01817 20.0448   245
## 5    5  99 86 0.02400 21.3732   334
## 6    6 101 87 0.03020 22.4172   249
```

Isolate the data from your ride for which your speed was over 65 kph.

What you have just done is called subsetting your data frame, and it is a very frequent task that we are going to be doing more of in the next hour. Because it is such a common task (e.g. pulling out all the subjects less than 40 years, pulling out all the male subjects, excluding an outlier), there is a specific command for it:

```
subset(ride.data, kph > 65)
```

```
##      secs cad  hr      km     kph watts
## 1387 1387  96 129 10.7909 65.7180    88
## 1388 1388  96 129 10.8098 66.7512    88
## 1389 1389  96 129 10.8284 67.1940    95
## 1390 1390  96 129 10.8471 66.3840    96
## 1450 1450 102 134 11.7990 66.8952   131
## 1451 1451 102 135 11.8178 67.6728   131
## 1452 1452 103 134 11.8365 67.0788   131
## 1453 1453 103 134 11.8553 65.0520   123
```

You can build other logical constraints to isolate data of interest in vectors, matrices and data frames by using `&` (AND), `|` (OR), and `!` (NOT). For example:

```
x <- c(3,4,2,7,5,8,9,5,-2,34,15,7)
# values of x greater than 2 and less than 7. Use the & for AND.
x[x > 2 & x < 7]
```

```
## [1] 3 4 5 5
```

```
# values of x not less than 15
x[!(x < 15)]
```

```
## [1] 34 15
```

```
# which is the same as
x[x >=  15]
```

```
## [1] 34 15
```

```
# values of x less than 3 or greater than 5. Use the | for OR.
x[x < 3 | x > 5]
```

```
## [1]  2  7  8  9 -2 34 15  7
```

```
# all values except the first three: use a minus sign
x[-(1:3)]
```

```
## [1]  7  5  8  9  5 -2 34 15  7
```

and with the data frame we might ask for rows your speed was under 20 km/h but your power was over 350 watts.

```
subset(ride.data, kph < 15 & watts > 350)
```

```
##      secs cad  hr       km     kph watts
## 1       1   1  80  0.00347 14.1984   360
## 1282 1282  98 162 10.08370  6.8832   386
## 1291 1291  98 163 10.10570  9.4284   359
## 1292 1292  98 163 10.10830  9.6624   356
## 1293 1293  98 163 10.11100  9.8424   354
## 1294 1294  98 164 10.11380 10.0656   362
## 1295 1295  98 164 10.11670 10.3860   360
## 1296 1296 100 164 10.11960 10.7208   360
## 1297 1297 100 164 10.12260 11.0880   353
## 1298 1298 102 165 10.12580 11.4624   353
## 1302 1302 108 166 10.13940 13.5828   353
```

(Note that – just like algebra – we can use parenthesis to make absolutely clear what order we are using to evaluate the expressions)

2.3 Lists

Lists are another way of storing data in a conveniently accessible way. Usually they are used for data types that are different in structure (or store different types of data) but are related because they are part of the same analysis. For example, R frequently provides output of statistical analysis in the form of a list. Suppose you had a vector, a data frame and a matrix:

```
a <- c("Doe","a","deer")
b <- data.frame(lyrics1 = c("a","female","deer"), lyrics2 = c("Ray","a","drop"))
d <- matrix(seq(from = 0, to = 100, length.out = 25), nrow = 5, ncol = 5)
```

But they were all related to some specific problem and you wanted to bundle them together in another variable. This is where you would use a list.

```
my.list <- list(a,b,d)
my.list
```

```
## [[1]]
## [1] "Doe"  "a"    "deer"
##
## [[2]]
##   lyrics1 lyrics2
## 1       a     Ray
## 2  female       a
```

```
## 3    deer    drop
##
## [[3]]
##           [,1]     [,2]     [,3]     [,4]      [,5]
## [1,]  0.000000 20.83333 41.66667 62.50000  83.33333
## [2,]  4.166667 25.00000 45.83333 66.66667  87.50000
## [3,]  8.333333 29.16667 50.00000 70.83333  91.66667
## [4,] 12.500000 33.33333 54.16667 75.00000  95.83333
## [5,] 16.666667 37.50000 58.33333 79.16667 100.00000
```

Components of the list are addressed using a double bracket notation. For example:

```
my.list[[2]] # gives the data frame.
```

```
##   lyrics1 lyrics2
## 1       a     Ray
## 2  female       a
## 3    deer    drop
```

If you happen to give the components of the list names, you can address with the $ notation:

```
my.list <- list(one = a,two = b,three = d)
my.list
```

```
## $one
## [1] "Doe"  "a"    "deer"
##
## $two
##   lyrics1 lyrics2
## 1       a     Ray
## 2  female       a
## 3    deer    drop
##
## $three
##           [,1]     [,2]     [,3]     [,4]      [,5]
## [1,]  0.000000 20.83333 41.66667 62.50000  83.33333
## [2,]  4.166667 25.00000 45.83333 66.66667  87.50000
## [3,]  8.333333 29.16667 50.00000 70.83333  91.66667
## [4,] 12.500000 33.33333 54.16667 75.00000  95.83333
## [5,] 16.666667 37.50000 58.33333 79.16667 100.00000
```

```
my.list$two # gives the same dataframe data
```

```
##   lyrics1 lyrics2
## 1       a     Ray
## 2  female       a
## 3    deer    drop
```

R very often uses lists for statistical reports. For example, the `lm()` function is used to generate a regression report by R. You can assign the output of `lm()` to a variable and this variable contains a very useful list.

```
x <- 1:10
y <- 2 * x + rnorm(10,0,1)  # generates some fake data
reg <- lm(y ~ x)
str(reg)                    # shows what variables that reg stores.

# ...but they generate a lot of output, so try it on your own

reg$residuals
reg$fitted.values
```

```
# OK, fine--we'll show one...
reg$coefficients
```

```
## (Intercept)           x
##  -0.7340475   2.0885229
```

As a final trivia point, data frames are really just lists of columns, each of which is an equal-length vector.