Quantitative Understanding in Biology
Introduction to R

Written by Luce Skrabanek, Jason Banfelder

Q1 2015

I Prologue

i. What is R?

R is a free software environment for statistical computing and graphics (`www.r-project.org`). It can effectively analyze large-scale datasets, such as those resulting from high-throughput sequencing experiments. It promotes automated and reproducible analyses of scientific data, creates a wide spectrum of publication quality figures, and has an extensive library of add-on packages to facilitate many complex statistical analyses. Because it is free and ubiquitously available (it runs on Windows, Mac, and Linux computers), your investment in learning R will pay dividends for years to come.

ii. What is RStudio?

While R is very powerful, it is essentially a command line program and is thus not the friendliest thing to use. Especially when learning R, a friendlier environment is helpful, and RStudio provides this, giving you things you expect in a modern interface like integrated file editing, syntax highlighting, code completion, smart indentation, tools to manage plots, browse files and directories, visualize object structures, etc.

From your computer, choose the RStudio application. This will start R under the hood for you.

II Introduction

i. R is a calculator

1. The Console panel (lower left panel) is where you type commands to be run immediately. When R is waiting for a new command, you will see a prompt character, >.
2. To add two numbers, type after the prompt:

   ```
   1 + 2
   ```

   When you hit return, you should see ...

   ```
   [1] 3
   ```

3. The answer is of course 3, but what is the `[1]` before it? It turns out that all numbers in R are vectors, and the `[1]` is a hint about how the vector is indexed. To see a long vector of random numbers, type:

   ```
   rnorm(100)
   ```

   For now we can ignore the vector indexing; we will learn more about vectors and indexing shortly.

4. R understands basic math. Try typing:

   ```
   3 - 4
   5 * 6
   7 / 8
   ```

5. The order of operations is kept, but you can always force an order with parentheses. Note the difference between ...

   ```
   1 + 2 * 3
   ```

   and ...

   ```
   (1 + 2) * 3
   ```

6. You can force R to do integer division using the `%/%` operator (division symbol inside two percent signs):

   ```
   17 %/% 4
   ```

   and to get the remainder:

   ```
   17 %% 4
   ```

7. You can also compute powers:

   ```
   2 ^ 4
   ```

   even with fractional exponents.

   ```
   2 ^ 4.3
   ```

8. Spacing doesn't matter, but the order of operations does, so be careful with using negative numbers.

   ```
   -2.1 ^ -4.5
   ```

   is not the same as

   ```
   (-2.1) ^ -4.5
   ```

9. R comes with an extensive library of built-in functions.


```
log(4)       # natural log
log10(4)     # log in base 10
log(4,10)    # same as above
sqrt(9)      # square root
abs(3-4)     # absolute value
exp(2)       # exponential
besselI(4,5) # whatever this is!!
```

10. Note in the examples above, we have used comments (preceded by the `#` character). You can type them if you want but they do not add anything to the work that R does. Comments are not usually used when interactively typing commands into the Console, but are essential when writing scripts - stay tuned!

ii. R has variables

1. It can be really useful to assign values to variables, so they can be referred to later. This is done using the assignment operator (`<-`).

   ```
   us.population <- 3.19e8 # From Wolfram|Alpha
   us.area <- 3719000      # From Wolfram|Alpha
   us.pop.density <- us.population / us.area
   us.pop.density
   ( us.pop.density <- us.population / us.area )
   ```

   Some notes:

   (a) Once a variable is defined, you will see it show up in the environment panel in RStudio.

   (b) R will not automatically print out the value of an assigned variable. Type the name of the variable by itself to see it. Alternatively, wrapping the assignment in parentheses executes the assignment and prints the result.

   (c) Case matters: `US.area` is not the same as `us.area`.

   (d) Word separation in R is traditionally done with periods (other programming languages tend to use underscores, but the R community will look at you funny).

2. Often, "quick and dirty" variable names that you will be using often in the Console are named with single letter variables, whereas variables in a script are long enough to be self-explanatory.

   Tip: Note that in RStudio, the `Tab` key will attempt to autocomplete the variable or function name that your cursor is currently on.

3. Use the `rm()` function to get rid of a variable from your environment.

   ```
   rm(us.pop.density) # gets rid of the us.pop.density variable
   ```

   Note that removing variables from your environment can help reduce clutter and is essential when dealing with large objects.

iii. Working with environments and history

1. You can save your environment (the set of variables you have defined). To do so, click the Save icon in the Environment tab (top right). Once you have saved your environment, the actual R command

that was run pops up in your Console. Note that RStudio automatically adds the traditional file extension of `.RData`.

2. To clear your current environment, click the broom icon on the Environment panel. You can also achieve this by typing:

   ```
   rm(list = ls(all = TRUE))
   ```

   Note that all the variables you just defined have disappeared!

3. To load an environment, click the Load icon and select the `.RData` file that you saved earlier. Again, you'll see the corresponding R command in the Console panel. Note that loading an environment does not empty your existing environment, but it will overwrite any existing variables.

4. It is good practice to have a separate directory for each project or analysis that you are working on. If you tell R about this directory, it will, by default, load and save files from it. We call this the working directory. You can browse files and directories from the Files tab of the lower right panel. Set the working directory using the gear icon (the More button). Alternatively, you can use the `ctrl-shift-K` shortcut. Note that the R command to set the working directory is also shown in the Console tab.

5. When you quit R, you will be asked if you want to save your workspace image (meaning your environment) in your working directory in a file called `.RData`.

6. RStudio always saves your history in your working directory. This can be a problem when restarting RStudio just by clicking on the Dock or the Start menu as your working directory will be your home directory and you will not see the history saved from your last session. On Macs, an easy way to specify your working directory when starting R is to drag the folder you want onto the RStudio icon. (Note that for Windows, the icon in your Toolbar does not work; you will have to use an alias on your Desktop). This will also load any saved `.Rdata` files from the directory.

7. Note that you can easily copy a line from your history to the Console by double-clicking it, or using the To Console icon.

iv. Getting help

1. Much work has gone into making R self-documenting. There are extensive built-in help pages for all R commands, which are accessible with the `help()` function. Thus, to see how `sqrt()` works, type:

   ```
   help(sqrt)
   ```

   The help page will show up in the Help section of the RStudio window. In case typing `help` is too long, there is a shortcut.

   ```
   ?sqrt
   ```

2. It should be noted that some special characters or reserved words have to be quoted when using either of the above help functions.

   ```
   ?"+"
   ```

   will show the help page for the arithmetic operators. Note that since the `+` function is just one of a group of similar operators, the help page explains all of them in a single page, rather than having separate pages for `+`, `-`, `*`, `/` etc. The help pages quite often will group similar functions together this way (e.g., the related functions `log()` and `exp()` are found on the same page).

3. Another very useful command is the `example()` function. Almost all R commands will include on their help pages (accessible using the `help()` or `?` functions) a series of examples. You can run these examples directly from your console by using the `example()` function. To see the examples for the `sqrt()` function, type:

   ```
   example(sqrt)
   ```

   This runs the set of examples that are listed at the bottom of the help page, exactly as if you had typed them out yourself.

4. The R help is not always as transparent as one would like and StackOverflow (`stackoverflow.com`) may be a better bet for answering your questions.

v. Data types

1. So far, we have only been dealing with numerical data, but in the real world, data takes many forms. R has several basic data types that you can use.

   ```
   has.diabetes    <- TRUE         # logical  (note case!)
   patient.name    <- "Jane Doe"   # character
   moms.age        <- NA           # used to represent an unknown ("missing") value
   NY.socialite.iq <- NULL         # used to represent something that does not exist
   ```

2. When working with truth values, you can use the logical operators:

   ```
   AND (&)
   OR (|)
   NOT (!)
   ```

   ```
   is.enrolled  <- FALSE
   is.candidate <- has.diabetes & ! is.enrolled
   ```

3. R uses tri-state logic when working with truth values.

   ```
   has.diabetes <- NA
   is.a.minor   <- TRUE
   has.diabetes & ! is.enrolled                  # returns NA
   has.diabetes & ! is.enrolled & ! is.a.minor # returns FALSE
   ```

4. R can convert among datatypes using a series of `as.()` methods.

   ```
   as.numeric(is.a.minor)
   as.numeric(is.enrolled)
   as.character(us.population)
   as.character(moms.age) # still NA - we still don't know!
   ```


III Data Structures

i. Overview

R has several different types of data structures and knowing what each is used for and when they are appropriate is fundamental to the efficient use of R. The ones that we are going to examine in detail here are: vectors, matrices, lists and data frames.

A quick summary of the four main data structures:

Vectors are ordered collections of elements, where each of the objects must be of the same data type or mode, but can be any mode.

A **matrix** is rectangular array, having some number of columns and some number of rows. Matrices can only comprise one data type (if you want multiple data types in a single structure, use a data frame).

Lists are like vectors, but whereas elements in vectors must all be of the same type, a single list can include elements from any data type. One of the nice things about lists is that elements can be named. A common use of lists is to combine multiple values into a single variable that can then be passed to, or returned by, a function.

Data frames are similar to matrices, in that they can have multiple rows and multiple columns, but in a data frame, each of the columns can be of a different data type, although within a column, all elements must be of the same data type. You can think of a data frame as being like a list, but instead of each element in the list being just one value, each element corresponds to a complete vector.

ii. Vectors

1. We've already seen a vector when we ran the `rnorm()` command. Let's run that again, but this time assigning the result to a variable.

   ```
   x <- rnorm(100)
   ```

2. Many commands in R take a vector as input.

   ```
   sum(x)
   max(x)
   summary(x)
   plot(x)
   hist(x)
   ```

   Don't get too excited about the plotting yet; we will be making prettier plots soon!

3. There are many ways of creating vectors. The most common way is using the `c()` function, where c stands for concatenation. Here we assign a vector of characters (character strings must be quoted).

   ```
   colors <- c("red", "orange", "yellow", "green", "blue", "indigo", "violet")
   ```

4. The `c()` function combines vectors. By assigning the result back to the `colors` variable, we are updating its value. The net effect is to both prepend and append new colors to the original `colors` vector.

   ```
   colors <- c("infrared", colors, "ultraviolet")
   # remember that "infrared" is a one-element vector
   ```

5. You can get the length of a vector using the `length()` function


```
length(colors)
```

6. You can access an individual element of a vector by its position (or "index"). In R, the first element has an index of 1.

   ```
   colors[7]
   ```

7. You can also change the elements of a vector using the same notation as you use to access them.

   ```
   colors[7] <- "purple"
   ```

   Tip: Appending an element is a slow operation because it actually creates a new vector. If you do this a limited number of times, this is fine, but if you are doing this 1000s of times, it is more efficient to create an empty vector of a pre-determined size, and then change the elements. You can create a blank vector using the `vector()` function.

   ```
   a.numeric.vector <- vector(mode="numeric", length=1000)
   a.numeric.vector[50] <- 5
   a.numeric.vector[750] <- 10
   plot(a.numeric.vector)
   ```

8. You can access multiple elements of a vector by specifying a vector of element indices.

9. R has many built-in datasets for us to play with. You can view these datasets using the `data()` function. Two examples of vector datasets are `state.name` and `state.area`.

10. We can get the last ten states (alphabetically) by using R's convenient way of making a vector of sequential numbers, with the ":" operator

    ```
    indexes <- 41:50
    indexes[1]
    indexes[2]
    length(indexes)
    state.name[indexes]
    ```

    Exercise:

    a. How would you list the first 10 and last 10 states (alphabetically)? Can you change this so that it works for any arbitrary length vector?

11. We can test all the elements of a vector at once using logical expressions. Let's use this to get a list of small states. First, how do we determine what a small state is?

    ```
    summary(state.area)
    ```

    Next, figure out which states are in the bottom quartile.

    ```
    cutoff <- 37320
    state.area < cutoff
    ```

    Note that this returns a vector of logical elements. We have seen that we can access vector elements using their indices, but we can also access them using logical vectors.

    ```
    state.name[state.area < cutoff]
    ```

12. We can test for membership in a vector using the `%in%` operator. To see if a state is among the smallest:

    ```
    "New York" %in% state.name[state.area < cutoff]
    "Rhode Island" %in% state.name[state.area < cutoff]
    ```


13. You can also get the positions of elements that meet your criteria using the `which()` function.

    ```
    which(state.area > cutoff)
    state.name[which(state.area > cutoff)]
    ```

    Techniques like this can be useful for removing outliers from your data.

14. Let's get the area of Wyoming:

    ```
    state.area[state.name == "Wyoming"]
    ```

    Notes:

    (a) The `==` is a test for equality. This is different from assignment.

    (b) The indexing vector here is a logical vector.

15. While this works, it can be a little long-winded. Luckily, R lets us name every element of a vector using the `names()` function.

    ```
    names(state.area) <- state.name
    ```

16. And now we can access Wyoming directly:

    ```
    state.area["Wyoming"]
    ```

17. Remember that we are using indexing vectors to access elements, and the indexing vectors can be characters.

    ```
    state.area[c("Wyoming", "Alaska")]
    ```

18. Now we can see all the small states and their areas in one shot:

    ```
    state.area[state.area < cutoff]
    ```

19. Sadly, not all functions that fetch an element from a vector keep the associated name.

    ```
    min(state.area)
    ```

    But you can find the index at which the minimum occurs, and use that.

    ```
    state.area[which.min(state.area)]
    ```

20. In addition to using the `:` notation to create vectors of sequention numbers, there are a handful of useful functions for generating vectors with systematically created elements.

    ```
    seq(1, 10)     # same as 1:10
    seq(1, 4, 0.5) # shows all numbers from 1 to 4, incrementing by 0.5 each time
    ```

    Let's look carefully at the help page for the `seq()` function.

    ```
    ?seq
    ```

21. The `seq()` function can take five different arguments, but not all of them make sense at the same time. In particular, it would not make sense to give the `from`, `to`, `by`, and `length` arguments, since you can figure out the length given `from`, `to`, and `by`. You can pass argments by name rather than position; this is helpful for skipping arguments.

    ```
    seq(0, 1, length.out = 10)
    ```

    Tip: In scripts, it is often good form to used named arguments, even when not necessary, as it makes your intent clearer.


```
seq(from = 1, to = 4, by = 0.5)
seq(from = 0, to = 1, length.out = 10)
```

22. Take a look at the help again: note that all of the arguments have default values, which will be used if you don't specify them.

    ```
    seq(to = 99)
    ```

23. Another commonly used function for making regular vectors is `rep()`. This repeats the values in the argument vector as many times as specified. This can be used with character and logical vectors as well as numeric.

    ```
    rep(colors, 2)
    rep(colors, times = 2) # same as above
    rep(colors, each = 2)
    rep(colors, each = 2, times = 2)
    ```

24. When using the `length.out` argument, you may not get a full cycle of repetition.

    ```
    rep(colors, length.out = 10)
    ```

25. In many cases, R will implicitly "recycle" vector elements as needed to get operations on vectors to make sense. When vector operations align, results are as you would expect:

    ```
    x <- 0:9
    y <- seq(from = 0, to = 90, by = 10)
    x + y
    ```

    Here, the first element of `x` has the first element of `y` added to it, the second element of `x` has the second element of `y` added to it, etc. What happens, though, when the vectors are not the same length?

    ```
    (1:5) + y
    ```

26. Here the elements of the first vector were recycled (your linear algebra professor would be horrified). When one vector is shorter than the other, the elements of that entire vector get recycled, starting from the first element and getting repeated as often as necessary. Note that if this mechanism does not use a complete cycle, you'll get a warning.

    ```
    (1:4) + y
    ```

27. Finally, note that using a single value (i.e., a scalar) is just a special case of recycling the same value over and over.

    ```
    y * 2
    ```

    Exercise:

    a. `0:10 / 10` yields the same result as `seq(from = 0, to = 1, by = 0.1)`. Can you understand why? Which do you think is more efficient?

    b. Can you predict what this command does?

    ```
    10 ^ (0:5)
    ```

28. R supports sorting, using the `sort()` and `order()` functions.

    ```
    sort(state.area)              # sorts the areas of the states from smallest to largest
    order(state.area)             # returns a vector of the positions of the sorted elements
    state.name[order(state.area)] # sort the state names by state size
    ```



```
state.name[order(state.area, decreasing = TRUE)]
                                # sort the state names by state size
```

29. We can also randomly sample elements from a vector, using `sample()`.

```
sample(state.name, 4)              # randomly picks four states
sample(state.name)                 # randomly permute the entire vector of state names
sample(state.name, replace = TRUE) # selection with replacement
```

    This is frequently used in bootstrapping techniques.

30. Other miscellaneous useful commands on vectors include

```
rev(x)    # reverses the vector
sum(x)    # sums all the elements in a numeric or logical vector
cumsum(x) # returns a vector of cumulative sums (or a running total)
diff(x)   # returns a vector of differences between adjacent elements
max(x)    # returns the largest element
min(x)    # returns the smallest element
range(x)  # returns a vector of the smallest and largest elements
mean(x)   # returns the arithmetic mean
```

Summary: Vector elements are accessed using indexing vectors, which can be numeric, character or logical vectors.

Summary: List of logical expression functions:
`<` `>` `<=` `>=` `!=` `==` `%in%`

Summary: Methods of generating regular vectors:
1. Numeric vector, from scratch, shortcut:
   `from:to`
2. Numeric vector, from scratch:
   `seq(from, to, by, length.out, along.with)`
3. Any type of vector, derived from an existing one (x):
   `rep(x, times, length.out, each)`

iii. Factors

Factors are similar to vectors, but they have another tier of information. A factor keeps track of all the distinct values in that vector, and notes the positions in the vector where each distinct value can be found. Factors are R's preferred way of storing categorical data.

The set of distinct values are called levels. To see (and set) the levels of a factor, you can use the `levels()` function, which will return the levels as a vector.

1. R has an example factor built in:

```
state.division
levels(state.division)
```

2. To get a hint about how R stores factors (or any other object), we can use the `str()` function to view the structure of that object. You can also use the `class()` function to learn the class of an object, without having to see all the details.

```
str(state.division)
class(state.division)
```


Note the list of integers corresponds to the level at each position. While factors may behave like character vectors in many ways, they are much more efficient because they are internally represented as integers and computers are good at working with integers.

3. You can convert a vector to a factor using the `factor()` function. Let's wish for some ponies.

```
pony.colors <- sample(colors, size = 500, replace = TRUE)
str(pony.colors)
```

Note that we are storing each color as a character string. This is not ideal. Let's convert this vector to a factor.

```
pony.colors.f <- factor(pony.colors)
str(pony.colors.f)
```

4. You can plot a factor to see how frequently each level appears.

```
plot(pony.colors.f)
```

The levels are plotted in the order they are returned by `levels()`. But you can control the order of the levels when you create the factor.

```
pony.colors.f <- factor(pony.colors, levels=colors)
str(pony.colors.f)
plot(pony.colors.f)
```

5. You can make a factor from a factor, reordering its levels as you go.

```
plot(state.division)
state.division <- factor(state.division, levels = sort(levels(state.division)))
plot(state.division)
```

6. You can rename the levels in a factor by assignment to its `levels()`. This only changes the labels, not the underlying integer representation. In this case, the labels we have are quite long; let's abbreviate them.

```
levels(state.division)
levels(state.division) <- c("ENC", "ESC", "MA", "MT", "NE", "PAC", "SA", "WNC", "WSC")
plot(state.division)
```

7. In most cases, you can treat a factor as a character vector, and R will do the appropriate conversions. Here we list the states in the North East, and then compare the sizes of various groups of states.

```
state.name[state.division == "NE"]
mean(state.area[state.division == "NE"]) / mean(state.area[state.division == "WSC"])
t.test(state.area[state.division == "SA"], state.area[state.division == "MT"])
```

iv. Matrices

The multi-dimensional structures in R, where all the elements are of the same data type, are called arrays. Arrays have three dimensions: number of rows, number of columns and number of layers. Matrices are a special type of array using only two of those dimensions (rows and columns) and can be thought of as tables.

1. Let's use one of R's built-in datasets, the `USPersonalExpenditure` dataset, which describes how much Americans spent in five categories from 1940-1960.


```
?USPersonalExpenditure
USPersonalExpenditure
```

2. Notice that the rows and columns of the matrix are named, using the categories and years as row names and column names, respectively. You can access (and set) the row names and column names using the `rownames()` and `colnames()` functions. There is a third function, `dim()`, which tells you the number of rows and columns making up your matrix.

   ```
   rownames(USPersonalExpenditure)
   colnames(USPersonalExpenditure)
   dim(USPersonalExpenditure)
   ```

   These functions return character vectors, which can be accessed with all usual access methods.

3. Accessing (and assigning) elements of a matrix is analogous to accessing (and assigning) elements of a vector, but unlike vectors which only have one index position, matrices have two, one for rows and one for columns. As with vectors, each specification is an indexing vector.

4. Let's say we wanted to see the Food and Tobacco expenditure for 1950:

   ```
   USPersonalExpenditure[1, 3]
   USPersonalExpenditure["Food and Tobacco", "1950"]
   USPersonalExpenditure[1, "1950"]
   ```

5. To get this expenditure for several years, we pass a vector for the column index.

   ```
   USPersonalExpenditure[1, c(5, 3, 1)]
   USPersonalExpenditure["Food and Tobacco", c("1960", "1950", "1940")]
   ```

6. Omitting a row or column index implies that all of the elements are wanted along that dimension.

   ```
   USPersonalExpenditure[1, ]
   USPersonalExpenditure["Food and Tobacco", ]
   USPersonalExpenditure["Food and Tobacco", , drop = FALSE]
   USPersonalExpenditure[ , c("1940","1950")]
   USPersonalExpenditure[1:3 , c("1940","1950")] # the result is a non-square matrix
   ```

7. Observe that if your result is one-dimensional, it is by default returned as a vector. All the vector operations you learned can be used on this output. If you don't want this behavior, you can use the `drop=FALSE` option, as was shown in the third example above.

   ```
   sum(USPersonalExpenditure[ , "1940"])
   ```

8. If you access an element of a matrix using only one index, R will treat the elements of the matrix as a vector of stacked columns.

   ```
   USPersonalExpenditure[1]
   USPersonalExpenditure[2]
   USPersonalExpenditure[7]
   ```

9. This is usually not the clearest way to access elements, but can be useful when you want to work with all of the elements.

   ```
   length(USPersonalExpenditure)
   sum(USPersonalExpenditure)
   ```

10. There are a few ways to make a new matrix. The `matrix()` function takes as arguments a vector of all the elements, and then some information about how many rows and columns there are. By default,

the matrix is filled in column order, but you can change this behaviour with the `by.rows=TRUE` option.

```
game1 <- matrix(c("X","","O","","X","O","","",""), ncol = 3)
game2 <- matrix(c("X","","O","","X","O","","",""), ncol = 3, byrow = TRUE)
```

Notes:

a. The elements of this matrix are characters. As with vectors, all elememts in a matrix must be of the same datatype.

b. In this matrix, the rows and columns are not named, and can only be accessed using numeric indices.

11. Assignment of values is just like for vectors, except using two dimensions.

```
game1[3,3] <- "X" # I win!!!!!
```

Example: Build a chess board

```
m <- matrix(data = "", ncol = 8, nrow = 8)
rownames(m) <- 8:1
colnames(m) <- letters[1:8]
m[c(2, 7), ] <- "pawn"
pieces <- c("rook", "knight", "bishop")
m[c(1, 8), ] = rep(c(pieces, rep("", 2), rev(pieces)), each = 2)
```

Notes:

a. When constructing this board, we set the size in advance. This is much more efficient than extending the matrix each time a new row or column is added.

b. `letters` is a very handy built-in vector of all 26 lowercase letters. See the help for `constants` for a few more handy built-ins.

c. Note how the pawn is recycled.

d. The last line is mouthful, but you've see all the components. Take your time, and understand how this works.

e. Can you finish the board?

12. Another very common way of assembling a matrix is by combining vectors or matrices. They can be stacked by row or column, using the rbind() or cbind() functions, respectively.

```
pieces <- c("rook", "knight", "bishop", "queen", "king", "bishop", "knight", "rook")
pawns <- rep("pawn", 8)
board <- rbind(rev(pieces), pawns, matrix("", nrow = 4, ncol = 8), pawns, pieces)
rownames(board) <- 8:1
colnames(board) <- letters[1:8]
```

Which method of building a chess board do think is clearer? You don't always have to be so clever.

13. One of the most powerful, but difficult to learn capabilities of R is its ability to systematically apply functions over data structures.

The `apply()` function works on matrices, and performs a specified function either on every row, or every column. The general form of the `apply()` function is:

```
apply(matrix, dimension, function, function_arguments)
```


The dimension argument can either be a 1, to represent rows, or 2, to represent columns. The function then works on each complete row, or each complete column. The first argument to the function is the row or column from the matrix. Additional arguments to the function can be specified, and they will be the same for every invocation of the function.

14. To compute the total expenditure for every year in the `USPersonalExpenditure` dataset:

    ```
    apply(USPersonalExpenditure, 2, sum)
    ```

    Note that in the example above, `sum()` returns a scalar, so the result of `apply()` is a vector. When the function that you apply returns a vector, the result is a matrix. The matrix is always built by columns, so you probably want to transpose the result (`t()`) when you are applying a function over rows. Here we compute five-year changes in expenditures in each category.

    ```
    apply(USPersonalExpenditure, 1, diff)
    t(apply(USPersonalExpenditure, 1, diff))
    ```

    To compute 10-year changes, we need to call `diff()` with the lag argument set to 2.

    ```
    t(apply(USPersonalExpenditure, 1, diff, lag = 2))
    ```

    Tip: While R supports for loops and other iterative programming constructs, the `apply()` function (and friends) are much more efficient and should be used whenever possible.

15. Remember all of the elements in a matrix must be of the same data type. If you assign one element of a different data type, R will convert (or coerce) elements as necessary.

    ```
    USPersonalExpenditure["Personal Care", "1955"] <- "Unknown"
    USPersonalExpenditure
    ```

    All the numbers have been converted to characters. How should we have done this to avoid this coercion?

16. R coerces elements up the data type chain, only far enough to satisfy the rule that all elements remain the same type. If you add an integer to a matrix of logicals, you'll get a matrix of integers, not characters, as in the example above.

    ```
    NULL < raw < logical < integer < double < complex < character < list < expression
    ```

    The same coercion strategy applies to vectors.

    The next section introduces a data structure that allows mixed types.

v. Lists

Lists are similar to vectors, but with some differences. Lists can hold data structures of different types, and of different sizes. Each component in a list can be (optionally, but commonly) separately named (a list in R is analogous to a hash or associative array in most other programming languages). In fact, one list can be a member of another list, allowing for deeply nested and arbitrarily complex data structures to be modeled.

1. The results of many high-level analyses in R are packaged as lists. Let's use R to fit a linear model to some data.

    ```
    (edu.spend <- unname(USPersonalExpenditure["Private Education", ]))
    (edu.yr <- seq(from = 0, to = 20, by = 5))
    plot(edu.yr, edu.spend)
    ```

```
my.model <- lm(edu.spend ~ edu.yr)
my.model
summary(my.model)
abline(my.model)
plot(my.model)
```

The `lm()` function fits data to a linear model (i.e., performs a linear regression), and packages up all of the results into an object we have named `my.model`. The `my.model` object is of class `lm` (linear model), which is a special kind of `list`.

2. Remember that we can look into any object using the `str()` function. Let's do that now with our linear model. Brace yourself!

   ```
   str(my.model)
   ```

   Take a moment to get a sense of what is packaged in the linear model object (but don't stress over the details).

   Now look carefully at the very first line of the output and note that this object is a "List of 12". This object has 12 components, and many of those have sub-components (an element of a list can be another list).

   We'll need to learn about lists so we can access the subcomponents of the model object; for example, you'd probably want to extract the slope and intercept of the best-fit line from this object. Let's start with the basics...

3. Lists can be created using the `list()` function. When using the `list()` function, you can optionally give names to the components (component names are called tags).

   ```
   pete <- list("Peter", "O'Toole", 1932, FALSE)
   pete <- list(first.name = "Peter", last.name = "O'Toole",
                yob = 1932, oscar.winner = FALSE)
   ```

   Note that we have different data types in the same list (character, numeric and logical). Note also the difference between the first and second attempt to model a great actor. In the first case, you need to know what the position of each component is; in the second, each component is named (tagged) with the tag we provided at creation.

   Note that we did not record Peter's age, but rather his year of birth. This way, the data is always correct, and you can always compute an age.

4. Let's add some of Peter's filmography...

   ```
   m1 <- list(title = "Lawrence of Arabia", year = 1962)
   m2 <- list(title = "Stardust", year = 2007)
   m3 <- list(title = "Troy", year = 2004)
   pete$movies <- list(m1, m2, m3)
   pete[["roles"]] <- c("T.E. Lawrence", "King", "Priam")
   str(pete)
   ```

   Note that we have used two different notations to refer to components of a list.

5. There are several ways to refer to the components of a list; the differences can be subtle, but important.

   The most straightforward way to refer to a single component is using the `[[` notation. You can always provide an integer to access the component by position, or you can provide a character string if the component is tagged.

For tagged components where the tag is a literal character string, you can use the `$` notation. This is less flexible, but looks clearer. Use this when you can.

6. If you want to access multiple components, you can use the `[` notation, which takes the usual indexing vector. Note that this will always return a list (this has to be so, because the components returned could be of different datatypes).

```
pete$yob
pete[[ paste("first", "name", sep = ".") ]] # new function! paste()
filmography <- pete[c("movies", "roles")]
pete$roles[3]
```

Exercise:

a. What does this return?

```
pete$movies[[2]]$title
```

b. Why does this NOT work?

```
pete$movies[2]$title
```

Tip: Use tags! Which of these makes for more readable code: `pete[[5]][[2]][[1]]` or `pete$movies[[2]]$title`

7. As with vectors, the `names()` function works on lists, for both retrieval and assignment of component tags.

```
names(pete)
names(pete)[5] <- "films"
```

8. Also as with vectors, the `length()` function reports the number of components in the list.

```
length(pete)
length(pete$films)
```

9. To remove a component, assign it the value `NULL`.

```
pete$oscar.winner <- NULL
```

Note that this changes the index positions of all subsequent components (yet another reason to use tags!)

10. The `lapply()` function (list apply) is the variant of the `apply()` function specifically for list data structures.

Remember that the general form of an `apply()` function is:

```
apply(data_structure, dimension, function, function_arguments)
```

For `lapply()`, we still have to specify the data structure we are analyzing, and the function we want performed on each component, but the dimension is fixed and is therefore left out.

```
lapply(list, function, function_arguments)
```

It works in much the same way as the `apply()` function, but instead of working on either all the rows or all the columns, it works on all the components of a list. By default, the result of `lapply()` is a list, which may not always be the most useful format for your result.

```
lapply(pete, length)
```


11. There is another function called `sapply()` (simplified list apply) which will try to simplify the result into either a vector or a matrix. This is only attempted if all the return values are of the same (positive) length.

    ```
    sapply(pete, length)
    ```

12. The `apply()` functions become very powerful when you use them to execute your own functions repeatedly. In this example, we'll compute how old Peter O'Toole was when he acted in each film.

    ```
    lapply(pete$films, function(x) x$year - pete$yob)
    sapply(pete$films, function(x) x$year - pete$yob)
    ```

    We'll learn a lot more about writing our own functions later; this is just to whet your appetite.

    Tip: `sapply()` could be a disaster waiting to happen, since the data type of the result you get back is dependent on the data you give it, so use with caution. You may want to use it only when you are guaranteed to get a data type you expect.

    Exercise:

    a. What does this do?

    ```
    subjects <- c("M", "F", "F", "I", "M", "M", "F")
    lapply(list(male = "M", female = "F", infant = "I"),
      function(sex) which(subjects == sex))
    ```

    Note that if you pass a vector as the first argument to `lapply()`, it will be coerced into a list (if the vector has names, they will become the tags, otherwise the list will be untagged).

    ```
    lapply(c("M", "F", "I"), function(sex) which(subjects == sex))
    ```

13. Remember our linear model?

    Exercise:

    a. Can you extract the slope and intercept of the best-fit line?

    b. Can you extract the value of R^2 reported by the `summary()` function we used? *Hint: save the result of this function, and look into its structure.*

vi. Data frames

Data frames are two-dimensional data structures like matrices, but, unlike matrices, they can contain multiple different data types. You can think of a data frame as a list of vectors, where all the vector lengths are the same. Data frames are commonly used to represent tabular data.

1. When we were learning about vectors, we were used several parallel vectors, each with length 50 to represent information about US states. The collection of vectors really belongs together, and a data frame is the tool for doing this.

    ```
    state.db <- data.frame(state.name, state.abb, state.area, state.center,
                           stringsAsFactors = FALSE)
    state.db
    ```

    The `data.frame()` function combines the four data sets into a single data frame. Note that the first three data sets are vectors (two character, one numeric), but the last data set is a list with two components.

Note the `stringsAsFactors = FALSE` argument. Some of the vectors that we are using are character vectors, but will be automatically converted to factors if this option is not set. Since we will want to work with our character data as vectors, not as factors, we want to set this argument to `FALSE`.

2. Data frames have a split personality. They behave both like a tagged list of vectors, and like a matrix! This gives you many options for accessing elements. Fortunately, you know them all already!

   When accessing a single column, the list notation is preferred.

   ```
   state.db$state.abb
   state.db[[ "state.abb" ]]
   state.db[[ 2 ]]
   ```

   When accessing multiple columns or a subset of rows, the matrix notation is used (rows and columns are given indexing vectors).

   ```
   state.db[ , 1:2]
   state.db[41:50, 1:2]
   state.db[c(50, 1), c("state.abb", "x", "y")]
   state.db[order(state.db$state.area)[1:5], ]
   state.db[order(state.db$state.area), ][1:5, ]
   ```

   The last two examples produce the same output; which is more efficient?

3. As with matrices, the rows can be given names as well. This makes picking out specific rows less error-prone. Column names are accessed with the `names()` or `colnames()` functions.

   ```
   rownames(state.db) <- state.abb
   state.db[c("NY", "NJ", "CT", "RI"), c("x", "y")]
   names(state.db) <- c("name", "abb", "area", "long", "lat")
   ```

   Note that if you only fetch data from one column, you'll get a vector back. If you want a one-column data frame, use the `drop = FALSE` option.

4. You can add a new column the same way you would add one to a list.

   ```
   state.db$division <- state.division # Remember, this is a factor

   state.db$z.size <- (state.db$area - mean(state.db$area))/sd(state.db$area)
   state.db[ , "z.size", drop = FALSE]
   ```

5. Remember that you can pass logical vectors as indices.

   ```
   state.db[state.db$area < median(state.db$area), "name"]
   state.db[state.db$area < median(state.db$area), "name", drop = FALSE]
   flyover <- state.db[ ! state.db$division %in%
     c("New England", "Middle Atlantic", "South Atlantic", "Pacific"), ]
   ```

6. Another way to select data from a data frame is using the `subset()` function. You can choose rows based on a logical expression and can choose columns with the `select` option. With this function, we only have to type the dataset name once.

   ```
   subset(state.db, area < median(area), select = name)
   flyover <- subset(state.db, ! division %in%
     c("New England", "Middle Atlantic", "South Atlantic", "Pacific") )
   ```

   The logical condition is optional, and you can specify columns to omit instead of columns to include.


```
subset(state.db, select = c(name, abb))
subset(state.db, select = -c(long, lat))
```

7. Many tools in R work naturally with data frames. For example, visualizing the size distribution of state within each division could not be easier once the data is in a well-designed data frame.

   ```
   plot(area ~ division, data = state.db)
   plot(log(area) ~ division, data = state.db)
   plot(lat ~ long, data = state.db)
   ```

   Here we are using R's function notation. Read the first argument in the first `plot()` example as "area as a function of division".

8. In addition to the `str()` function, you can glean useful information about a data frame (and other data structures) using the `summary()` and `head()` functions.

   ```
   summary(state.db)
   head(state.db)
   ```

9. In most cases in the lab, you won't be typing the data in by hand but rather importing it. R provides tools for importing a variety of text file formats. If you receive data in Excel format, you'll want to save it as tab-delimited or CSV (comma separated values) text. The `read.delim()` or `read.csv()` functions can then be used to import the data into a data frame. Of course, you should also tell your colleagues that R is the preferred tool for data wrangling!

   We have prepared an Excel file for you to import.

   a. Make a new folder on your Windows machine for this project.

   b. Go to `http://chagall.med.cornell.edu/Rcourse/` and download the Ablation.xlsx file into the folder you just made.

   c. Open the file using Microsoft Excel and save it in CSV format. You can quit Excel now!

   d. Return to RStudio and set your working directory to your newly created project folder.

   e. Use the Files tab (bottom right) to take a look at the .csv file.

   f. Import the data into a data frame

      ```
      ablation <- read.csv("Ablation.csv", header = TRUE, stringsAsFactors = TRUE)
      ```

      Note that we explicitly asked that strings be converted to factors. In cases where you are importing string data, you would want to set this to `FALSE`, and after import, convert appropriate columns to factors.

   g. Rename the `SCORE` column to be consistent with the other column names.

      ```
      names(ablation)[names(ablation) == "SCORE"] <- "Score"
      ```

      Note that we did not use the command `names(ablation)[6] <- "Score"`. Why do you think that is?

   There is another more general import function, `read.table()`, that gives you exquisite control over how to import your data. One of the defaults of this function is `header = FALSE`. For this reason, we suggest that you always explicitly use the `header` option (you don't want to accidentally miss your first data point).

10. If your uninitiated colleagues insist on an Excel-compatible format, you can also export a data frame using `write.csv()`.

IV CRAN and Libraries

One of the major advantages of using R for data analysis is the rich and active community that surrounds it. There is a rich ecosystem of extensions (also known as libraries or packages) to the base R system. Some of these provide general functionality while others address very specific tasks.

The main hub for this ecosystem is known as CRAN (Comprehensive R Archive Network). CRAN can be accessed from `http://cran.r-project.org/`. This is also where you go to download the R software.

Follow the `Packages` link to browse the 5000+ packages currently available.

Because `R` is a not a very specific search term, often when doing a web search, the term `CRAN` is used.

In the next section, we will learn how to use the ggplot2 package for preparing publication-quality figures. Here we will download and install this package. This couldn't be easier, because R knows all about CRAN.

```
install.packages(c("ggplot2")) # Library name is given as a string
```

If this is the first time a package is being installed on your computer, R may ask you to select a CRAN mirror. Pick something geographically close by. Note that you only have to install a package once.

Depending on how your computer (and R installation) is set up, you may receive a message indicating that the central location for packages is not writable; in this case R will ask if you want to use a personalized collection of packages stored in your home directory.

Installing a package does not make it ready for use in your current R session. To do this, use the `library()` function.

```
library(ggplot2)               # Library name is an object (not a string)
```

You need to do this in every session or script that will use functions from this library.

V Plotting

Although R has some basic plotting functionality which we have seen hints of, the **ggplot2** package is more comprehensive and consistent. We'll use ggplot2 for plotting for the rest of this workshop.

ggplot2 is written by Hadley Wickham (`http://had.co.nz/`). He maintains a number of other libraries; they are of excellent quality, and are very well documented. However, they are updated frequently, so make sure that you are reading the current documentation. For ggplot2, this can be found at...

`http://docs.ggplot2.org/current/`

In this workshop, we will also be using his **reshape2** and **plyr** packages.

ggplot2 relies entirely on data frames for input.

1. Let's make our first ggplot with the `ablation` data that we imported earlier.

   ```
   ggplot(ablation, aes(x = Time, y = Score)) + geom_point()
   ```

   At a minimum, the two things that you need to give ggplot are:

   a. The dataset (which must be a data frame), and the variable(s) you want to plot

   b. The type of plot you want to make.

2. ggplot gives you exquisite control over plotting parameters. Here, we'll change the color and size of the points.

   ```
   ggplot(ablation, aes(x = Time, y = Score)) + geom_point(color = "red", size = 4)
   ```

   Aesthetics are used to bind plotting parameters to your data.

   ```
   ggplot(ablation, aes(x = Time, y = Score)) +
     geom_point(aes(color = Experiment), size = 4)
   ggplot(ablation, aes(x = Time, y = Score)) +
     geom_point(aes(color = Experiment, shape = CellType), size = 4)
   ```

3. When using ggplot, layers are added to a ggplot object. You can add as many layers as you like.

   ```
   ggplot(ablation, aes(x = Time, y = Score)) +
     geom_point(aes(color = Experiment), size = 4) +
     geom_text(aes(label = CellType), hjust = 0, size = 3)
   ```

   It is sometimes useful to save off the base ggplot object and add layers in separate commands. The plot is only rendered when R "prints" the object. This is useful for several reasons:

   a. We don't need to create one big huge command to create a plot, we can create it piecemeal.

   b. The plot will not get rendered until it has received all of its information, and therefore allows ggplot2 to be more intelligent than R's built-in plotting commands when deciding how large a plot should be, what the best scale is, etc.

   ```
   p <- ggplot(ablation, aes(x = Time, y = Score))
   p <- p + geom_point(aes(color = Experiment, shape = Measurement), size = 4)
   p <- p + geom_line(aes(group = interaction(Experiment, Measurement, CellType),
                          color = Experiment,
                          linetype = CellType))
   print(p) # plot gets rendered now
   ```


Sourcing a file will not automatically generate output, so here we have to explicitly ask for the plot to be printed.

Here we've added a layer that plots lines. We want a separate line for each unique combination of Experiment, Measurement, and CellType. The `interaction()` function takes a set of factors, and computes a composite factor. Try running...

```
interaction(ablation$Experiment, ablation$Measurement, ablation$CellType)
```

...to see what this does. This composite factor is passed to the group aesthetic of `geom_line()` to inform ggplot which data values go together.

We have also added a new binding to `geom_point()`. The shape of each point is determined by the corresponding Measurement. Note that ggplot prefers six or fewer distinct shapes (i.e., there are no more than six levels in the corresponding factor). You can, however, use more using a command like...

```
scale_shape_manual(values = 1:11)
```

4. In the above example, we specified that the color of both points and lines should be determined by the Experiment. It is therefore tidier to specify this binding once in the ggplot aesthetic. Any bindings defined there are inherited by all layers (but can be overridden by any individual layer's aesthetic).

   Now is a good time to introduce the Source panel in RStudio. We've already seen that you can view the contents of a data file here. It is also a best practice to store R commands in files. Choose File ⇒ New File ⇒ R Script. An empty tab should appear. Enter the following, and then save it in your project directory by clicking the disk icon. Now click the Source button to run these commands.

   ```
   p <- ggplot(ablation, aes(x = Time, y = Score, color = Experiment))
   p <- p + geom_point(aes(shape = Measurement), size = 4)
   p <- p + geom_line(aes(group = interaction(Experiment, Measurement, CellType),
                          linetype = CellType))
   print(p)
   ```

5. Some layers don't plot data, but affect the plot in other ways. For example, there are layers that control plot labeling and plot theme. The `labs()` function can also modify legend labels.

   ```
   ( p <- p + labs(title = "Ablation", x = "Time (minutes)", y = "% Saturation") )
   ( p <- p + theme_bw() )
   ```

6. ggplot gives you control over the scales of your plot. There is one scale for each binding. In the plot we just made, there are five scales that we can manipulate: the x and y axes and the three legends.

   Let's change our x-axis to include the 5 minute timepoint. This is achieved with yet another layer.

   ```
   p + scale_x_discrete(breaks = c(0, 5, 10, 20, 30))
   p + scale_x_discrete(breaks = unique(ablation$Time))
   ```

   Tip: In the second example above, we have computed the breaks from the data, rather than listing them individually. This makes the code we are writing usable even when the data changes. This is an essential strategy for reproducibly analyzing data at scale.

7. We can also manipulate legends with scale layers.

   ```
   p <- p + scale_shape_discrete(labels = c("LDLR", "TfR")) +
            scale_linetype_discrete(name = "Cell type") +
            scale_color_brewer(palette = "Set1")
   ```


Here we provide the labels for the Measurement scale (remember that we used an aesthetic to bind shape to Measurement). Note that ggplot will always order the labels according to the levels of the underlying factor, so the labels should be provided in that order. If you want to change the order in which the legend elements are displayed, change the underlying factor.

We have also changed the title of the CellType legend (the linetype binding) to be two words and used a different color palette (for the binding to Experiment).

Note the general form of the scale layer functions:

`scale_aestype_colortype`

where the aestype is the bound aesthetic, and the colortype is the type of color associated with that binding.

Common values for the aestype and colortype include:

Aestype	
colour	Color of lines and points
fill	Color of area fills (e.g. bar graph)
linetype	Solid/dashed/dotted lines
shape	Shape of points
size	Size of points
alpha	Opacity
x,y	x and y axes

Colortype	
hue	Equally-spaced colors from the color wheel
manual	Manually-specified values (e.g., colors, point shapes, line types)
gradient	Color gradients
grey	Shades of grey
discrete	Discrete values (e.g., colors, point shapes, line types, point sizes)
continuous	Continuous values (e.g., alpha, colors, point sizes)

Table 1: Scale layer function components.

8. This plot is probably showing too much data at once. One approach to resolve this would be to make separate plots for the LDLR and TfR measurements. You can make multiple plots at once using facets. Here are a few options.

```
p + facet_grid(Measurement ~ .)
p + facet_grid(. ~ Measurement)
p + facet_grid(Experiment ~ Measurement)
p + facet_grid(Measurement ~ Experiment)
```

In these plots, you can remove the color and shape legends entirely (an option that can be specified in each of the respective legend layers)...

```
p + scale_colour_discrete(guide = "none") + scale_shape_discrete(guide = "none")
```

...or you may no longer want to bind the Measurement and Experiment variables to shape and color at all.

Tip: The `facet_wrap()` function in ggplot can be used to wrap a 1D ribbon of plots into a 2D layout. You can also use the **gridExtra** package to place independently generated plots on the same page.

9. When plotting many points, controlling opacity can be useful. Let's model an ant-infested park. (To create a reproducible data set, you can set the seed for the random number generator with the `set.seed()` function).

```
trees <- data.frame(x = rnorm(100), y = rnorm(100),
  size = rnorm(100, mean = 5, sd = 2))
ants <- data.frame(a = rnorm(10000, sd = 0.4, mean = 1),
  b = rnorm(10000, sd = 0.2, mean = -1))
p1 <- ggplot() +
  geom_point(data = ants, aes(x = a, y = b),
    color = "brown", size = 2, alpha = 0.01) +
  geom_point(data = trees, aes(x = x, y = y, size = size),
    color = "green", shape = 8)
print(p1)
```

Note that here we are plotting points from two different data frames, so we don't provide a default dataset or default bindings to x or y in the `ggplot()` function. These can always be set (or overridden) in individual layers.

Exercise:

a. Compare the result without specifying alpha.

10. ggplot can do statistical analyses on the fly. We won't cover the details here, but here's an example to whet your appetite:

```
p2 <- ggplot() +
  stat_density2d(data = ants, aes(x = a, y = b, fill= ..level.. ),
                 alpha = 0.5, geom = "polygon") +
  geom_density2d(data = trees, aes(x = x, y = y)) +
  geom_point(data = trees, aes(x = x, y = y, size = size),
             color =" green", shape = 8) +
  scale_fill_gradient(low = "white", high = "brown")
print(p2)
```

Note the use of the binding to the `..level..` variable. This binds to a statistic (in this case the 2D density of points) computed by the `stat_density2d()` layer (computed variables are identified by the appended and prepended "`..`").

11. You can ask R to produce your plots as PDF files rather than display them on the screen.

```
pdf(file = "figures.pdf", paper = "letter")
print(p)
print(p1)
print(p2)
dev.off()
```

Whenever a PDF device is open, all plotting (with ggplot or otherwise) will be to the PDF file, not to the RStudio plot tab. So don't forget that `dev.off()` command.

Note that you can also use RStudio to export individual plots as PDFs or PNGs from the Plot tab.

VI Data wrangling

In addition to making plots, R has tools to help you prepare tabular views of your data. This task can be viewed as a two-part process: first, selecting (aka subsetting) the data points that you want to present, and then arranging that data into the rows and columns that you need.

i. Casting data

1. We've already seen how to use the `subset()` function to filter data, so you're halfway there. Let's assume that we want to show the LDLR data from the E1909 experiment.

   ```
   subset(ablation, Measurement == "LDLR-ABLATION" & Experiment == "E1909")
   ```

   To reshape this list of data points into a dataframe where there is one row per time point and one column per cell type, we can use the `dcast()` function from the **reshape2** package.

   ```
   library(reshape2)       # This is installed with ggplot2,
                           # but it must be loaded to use the functions.
   dcast(subset(ablation, Measurement == "LDLR-ABLATION" & Experiment == "E1909"),
         Time ~ CellType,
         value.var = "Score")
   ```

   Note that all of the experimentally measured values in this table come from the original `Score` column; this is indicated by the `value.var` option in the above command.

   The values in the `Time` column, and the column names, are referred to as identifying variables. In other words, if you are looking up a specific experimentally measured value, you would need to specify the identifying variables (Time and CellType) in order to find it.

2. When using `dcast()`, you can specify more than one identifying variable on the right side of the tilde. For example, if you want to show all of the data from the E1909 experiment, you could use this command:

   ```
   dcast(subset(ablation, Experiment == "E1909"),
         Measurement + Time ~ CellType,
         value.var = "Score")
   ```

   Now there are two columns of identifiers (Measurement and Time). The first one you specify will be the "slowest" to change. You can swap their order if you like:

   ```
   dcast(subset(ablation, Experiment == "E1909"),
         Time + Measurement ~ CellType,
         value.var = "Score")
   ```

   You can also use more than one identifier in the column names.

   ```
   dcast(ablation,
         Measurement + Time ~ Experiment + CellType,
         value.var = "Score")
   ```

   Note that column names are amalgamations of the originating identifiers. Sometimes this is OK, but it can be annoying.

3. The **reshape2** package includes an analogous `acast()` function which returns a matrix or multidimensional array. We won't cover that here.

ii. Melting data

You may have noticed that the format of the `ablation` data frame is a bit peculiar. The Excel sheet you imported for the plotting exercise is probably not what you are used to getting from your colleagues, or working with yourself. It is, however, in the canonical format for storing and manipulating data that you should be using.

The hallmark of this canonical (melted) format is that there is only one (set of) independently observed value(s) in each row. All of the other columns are identifying values. They explain what exactly was measured. This is also known as metadata in some circles.

When your data is in this format, it is straightfoward to subset, transform, and aggregate it by any combination of factors of the identifying variables. Not so for data that has been cast. That is why, for example, the **ggplot** package essentially requires that your data is in melted format.

1. If you are given data in non-canonical format, you can use the `melt()` function to fix it. Melting will convert a data frame with several measurement columns (i.e., "fat") into a "skinny" data frame which has one row for every observed (measured) value.

   Let's start with a "fat" data frame about states.

   ```
   state.stats <- data.frame(Name = rownames(state.x77), state.x77)
   str(state.stats)
   ```

   Here we have a data frame with eight observations per row. This is begging to be melted; we shall oblige...

   ```
   state.m <- melt(state.stats, id.vars = "Name")
   str(state.m)
   ```

   By default, the name of the measurement variables will be put into a column called "variable", and their associated values will be put into a column called "value", but these can be changed by using the arguments `variable.name` and `value.name`.

   You need to tell `melt()` which of your variables are id variables, and which are measured variables. If you only supply one of `id.vars` and `measure.vars`, `melt()` will assume the remainder of the variables in the data set belong to the other. If you supply neither, `melt()` will assume factor and character variables are id variables, and all others are measured.

   Although it may seem counter-intuitive at first, this is how you want to store your data, especially when you have more than two identifying variables.

2. You can melt data structures other than data frames. For example, we can melt the USPersonalExpenditure matrix.

   ```
   us.pe.m <- melt(USPersonalExpenditure,
                   varnames = c("Category", "Year"),
                   value.name = "Amount")
   ```

   Once melted, the data can readily be plotted with ggplot. Here we'll use stacked bar charts, showing the expenditure per year, colored by Category.

   ```
   ggplot(us.pe.m, aes(x = Year, y = Amount)) +
     geom_bar(stat = "identity", aes(fill = Category))
   ```

   ```
   ggplot(us.pe.m, aes(x = Year, y = Amount)) +
     geom_bar(stat = "identity", aes(fill = Category, order = -as.numeric(Category))) +
     theme(legend.justification = c(0,1), legend.position = c(0,1))
   ```


By default, `geom_bar()` is set up to plot frequencies of categorical observations. Since we are plotting numerical values, we need to use the `stat = "identity"` option.

Also note that in the second example, we have reordered the bars to match the legend, and relocated the legend.

iii. Merging data

1. It is usually a good idea to keep all of your data from a particular study or project in a very small number of canonical "skinny" data frames. Consider the ablation data we've been using; when new experiments are performed, you can add new rows to the `ablation` data frame with the `rbind` function. If the data for the new experiment is given to you in "fat" format (say via a new Excel workbook), you may need to `melt()` the new data first, and then `rbind()` it.

2. Sometimes, you will want to add new columns before doing this. For example, if all of the data thus far was collected by you, you probably did not bother to store that metadata. However, if, after some early success, your PI assigns another post-doc to the project, you may want to create a new data frame with this information and store it in the project environment.

```
experiment.log <- data.frame(Experiment = c("E1909", "E1915", "E1921"),
                             Tech = c("Jason", "Luce", "Jason"),
                             stringsAsFactors = TRUE)
str(experiment.log)
experiment.log
```

When looking for technician-specific bias, you will need to merge the technician data with your main data using the `merge()` function.

```
merge(ablation, experiment.log)
```

The `merge()` function merges two data frames based on common column values. By default, it looks for common column names, but these can be specified explicitly. By default, only rows which have elements common to both data frames will be kept (it is similar to a database table inner join). You can force a left or right (or full) database-style join by setting the `all.x` or `all.y` arguments to `TRUE`, respectively.

The merged data frame contains redundant information; i.e., if you know the Experiment, you know the Tech (we say the data is "denormalized"). While the merged data frame may be convenient when investigating "tech effects", you probably don't want to store this data frame permanently. This becomes more important at scale, when the cost of storing the redundant information becomes a limiting factor (usually in terms of the memory needed by R).

Tip: Use `merge()`! Don't depend on vectors being aligned unless you are absolutely, positively sure they are, and `merge()` is not an option. Such assumptions are a very, very common source of errors in data analysis (not just in R – think about what you do when you paste a new column into an Excel sheet).

3. To save your work, use R's `save()` function. This will save it in an Rdata format, which can later be reloaded with the `load()` function. In the example below, we save a single object to a file; you can also pass a list of objects as the first argument to save the collection.

```
save(ablation, file = "ablation.Rdata")
load("ablation.Rdata")
```


iv. Summarizing data by groups

1. When we were using `dcast()` earlier we were only rearranging (and optionally subsetting) the raw data. In other words, every value in the cast data frame could be found in the original data frame. The `dcast()` function can also be used to summarize your data (this is also known as data aggregation). For example, we may want to compute the mean Score for each combination of Measurement, Time, and CellType (i.e., averaging across experiments).

```
# Show all raw data in a table
dcast(ablation, Experiment + Time + Measurement ~ CellType, value.var = "Score")
# Averages across experiments
dcast(ablation,
      Time + Measurement ~ CellType,
      value.var = "Score",
      fun.aggregate = mean)
```

2. You can pass any function that takes a vector and returns a single value as an aggregation function; `dcast()` will call this function for each unique combination of levels of the factors that you specify in the formula. Here we compute the number of observations for each unique combination of Measurement, CellType and Experiment.

```
# Number of observations for each case
dcast(ablation,
      Measurement + CellType ~ Experiment,
      value.var = "Score",
      fun.aggregate = length)
```

   When grouping by only two factors, you are essentially producing a simple contingency table; the R base package includes a `table()` function which will achieve the same results, and which you are likely to see. We prefer the use of `dcast()` as it is more general.

```
# Number of observations for each case
dcast(ablation,
      CellType ~ Measurement,
      value.var = "Score",
      fun.aggregate = length)
table(ablation$CellType, ablation$Measurement)
```

   Note that if your data has a column of counts, then you will make a contingency table using `sum()` as the aggregation function given to `dcast()`.

3. If you don't specify an aggregation function and there is more than one row of data that repeats a unique combination of levels for the factors specified in your formula (i.e., if there is more than one value, `dcast()` will default to aggregating data using the `length()` function (and emit a warning indicating it is doing so).

```
# Bad form; better to specify aggregation function.
dcast(ablation, Measurement + CellType ~ Experiment, value.var = "Score")
```

   Tip: Especially when scripting, do not rely on default behavior; explicitly specify the desired aggregation function when using `dcast()`.

4. You can also use `max()` or `min()` as the aggregation function. If you try this, you'll see that R issues an odd warning about non-missing arguments to `max()`. Under the hood, `dcast()` will always call your aggregation function once with an empty vector (to figure out what to put in the result table if

there are no cases for a particular combination of levels). Calling `max()` with an empty vector issues this warning (try it!). You can explicitly state what value should be used for missing cases.

```
# Maximum measurement of each type for each experiment
dcast(ablation,
      Measurement + CellType ~ Experiment,
      value.var = "Score",
      fun.aggregate = max)
dcast(ablation,
      Measurement + CellType ~ Experiment,
      value.var = "Score",
      fun.aggregate = max,
      fill = -Inf)
```

If you want to compute the range (max and min) for each combination of levels, you can't use `dcast()` (remember, the aggregation function must return a single value). The next section will introduce a different package that will allow you do this, and much, much more!

v. Plyr and the "split, apply, combine" approach

When you use `dcast()`, what you are actually doing is splitting your data into groups (by sets of factors), applying some function on each group and then combining the results into a set with one entry per group.

As we saw above, `dcast()` requires that the apply phase of this approach only return a scalar. Many analyses require something a little more flexible. Luckily, Hadley Wickham has another package that allows us to do just that: **plyr**.

The plyr package contains a family of functions that implement the split, apply, combine approach in a flexible but consistent manner. The plyr functions can work with any data structure as input, and can output any data structure. The general form of the plyr functions are xyply(), where x and y are one-letter abbreviations of the different data structures (a: array, matrix or vector; l: list; d: data frame).

1. Let's see how we can find the range of Scores for each combination of Measurement, CellType and Time, using the `ddply()` function.

   ```
   ddply(ablation,
         ~ Measurement + CellType + Time,
         summarize,
         the.max = max(Score), the.min = min(Score))
   ```

   We use the ddply function because our input is the ablation data frame, and we want the result to be a data frame. The second argument specifies the factors that we will **split** by. The third argument gives the function to be **applied** to each group thus split. The fourth and additional arguments are passed to this applied function.

   Here we have introduced the `summarize()` (or `summarise()`) function. The general form of `summarize()` is:

   ```
   summarize(df, new.col.name = some.function)
   ```

   The first argument to `summarize()` is a data frame, and subsequent arguments are functions that will be called in the context of the data frame. The result of `summarize()` is a new data frame where each column is the result of one of the functions you supplied (the name of the column will be the argument name you gave to the function).


```
summarize(ablation, the.max = max(Score), the.min = min(Score))
```

The `ddply()` function then **combines** each of the resultant data frames into one master result.

> **Tip**: This example uses a common recipe for summarizing data that is in a melted data frame. The general form is:
> `ddply(df, grouping.factors, summarize, new.col.name=some.function, ...)`

2. There are a few ways to list grouping factors in the plyr functions. You can use the notation above, where the factors are part of an R formula, or you can pass their names as a character vector, or use plyr's unique syntax. Use your favorite.

```
ddply(ablation,
      ~ Measurement + CellType + Time,
      summarize,
      the.max = max(Score), the.min = min(Score))
ddply(ablation,
      c("Measurement", "CellType", "Time"),
      summarize,
      the.max = max(Score), the.min = min(Score))
ddply(ablation,
      .(Measurement, CellType, Time),
      summarize,
      the.max = max(Score), the.min = min(Score))
```

3. A common use for this pattern is to compute the mean and standard deviation within each grouping. This can be the basis for including error bars on your plot.

```
ablation.mean.sd <- ddply(ablation,
                          .(Measurement, CellType, Time),
                          summarize,
                          mean = mean(Score), sd = sd(Score))

ggplot(ablation.mean.sd, aes(x = Time, y = mean)) +
  geom_point(size = 4) +
  geom_errorbar(aes(ymin = mean-sd, ymax = mean+sd), width = 0.4) +
  facet_grid( Measurement ~ CellType) +
  geom_line() +
  geom_point(data = ablation, aes(y = Score), color = "blue", shape = 1) +
  labs(title = "+/- 1 SD")
```

In the above plot, we used the `geom_errorbar()` function which requires a unique aesthetic that binds `ymax` and `ymin` to the upper and lower bounds of the error bars.

4. You can pass any function to `ddply()`, not just `summarize()`. In many cases, you may write your own function. Here, we'll prepare a similar plot to the one above, but using confidence intervals computed from a t-test as the limits of the error bars.

```
ablation.score.limits <-
  ddply(ablation,
        .(Measurement, CellType, Time),
        function(d) { c(mean = mean(d$Score), limits = t.test(d$Score)$conf.int) }
       )

ggplot(ablation.score.limits, aes(x = Time, y = mean)) +
```



```
  geom_point(size = 4) +
  geom_errorbar(aes(ymax = limits2, ymin = limits1), width = 0.4) +
  facet_grid(Measurement ~ CellType) +
  geom_line() +
  geom_point(data = ablation, aes(y = Score), color = "blue", shape = 1) +
  labs(title = "95% CIs")
```

Note here that our function returns a three-element named vector; `ddply()` adds a column to the data frame for each vector element.

5. There is another function called `transform()` which is similar to `summarize()`, but instead of putting its result into a new data frame, the output gets appended to the input data frame. This can be especially useful when you want to compute a value for each line in your data frame. Like `summarize()`, `transform()` can output multiple new columns.

   ```
   transform(ablation, rate=ifelse (Time > 0, Score/Time, NA ))
   ```

   Note the use of the `ifelse()` function. This is a vector operation that tests the expression given as the first argument for every element in a vector. If the expression evaluates to TRUE, the second argument is the result, otherwise the third one is. The `transform()` function adds a column to the ablation data frame with the computed result.

6. When coupled with `ddply()`, `transform()` can compute a value for every line that is a function of some grouping. In the following example, we use `transform()` with `ddply()` to determine whether a Score is an outlier within a group of experiments (here we defined an outlier as being outside of ± 1 SD of the mean.

   ```
   ggplot(ablation.mean.sd, aes(x = Time, y = mean)) +
     geom_point(size = 2) +
     geom_errorbar(aes(ymin = mean - sd, ymax = mean + sd), width = 0.4) +
     facet_grid( Measurement ~ CellType) +
     geom_line() +
     geom_point(data = ddply(ablation,
                             .(Measurement, CellType, Time),
                             transform,
                             outlier = abs((Score - mean(Score)) / sd(Score)) > 1),
                aes(y = Score, color = outlier),
                size = 4,
                shape = 1) +
     labs(title = "+/- 1 SD") +
     scale_colour_discrete(name = "Outlier Status",
                             labels = c("Within 1 SD", "Outside 1 SD"))
   ```

   Here, the data argument to the second `geom_point()` function is an inline call to `ddply()`. This is not recommended in practice, but is shown to give you an idea of what is possible. Also, note that a more appropriate cutoff for outliers is ± 3 SD.

7. Plyr provides the `d_ply()` function which is a sister to `ddply()`. As the name suggests, this function takes a data frame as input and has no output (i.e., the **combine** step in the normal approach is skipped). This function is useful when you are interested in the "side effects" of the function, for example, printing a plot.

   In this example, we define a function to generate a plot, and then apply it repeatedly to subsets of a data frame.


```
plot.ci <- function(d) {

  cell.types <- paste(unique(d$CellType), collapse = "_")
  measurements <- paste(unique(d$Measurement), collapse = "_")
  title <- paste(cell.types, measurements, sep = " / ")

  ggplot(d, aes(x = Time, y = mean)) +
    geom_point(size = 4) +
    geom_errorbar(aes(ymax = limits2, ymin = limits1), width = 0.4) +
    geom_line() +
    labs(title = title)
}

d_ply(ablation.score.limits, .(CellType, Measurement), plot.ci, .print = TRUE)
```

Note that this function has multiple statements. We will be seeing longer functions in the next section.

VII Reproducible analysis

To facilitate reproducible analysis, it is a best practice to write a script that loads your raw data, runs your entire analysis, and produces appropriate plots and output without any intervention. Keeping the script open in the Source panel in RStudio and checking the Source on Save option can be helpful as you develop your script.

An example based on the material we have covered in this workshop is shown below.

```
library(ggplot2)
library(plyr)

plot.ablation <- function(d) {

  title <- paste(unique(d$Measurement), collapse = "_")

  ggplot(d, aes(x = Time, y = mean)) +
    geom_point() +
    geom_line(aes(color = CellType)) +
    labs(title = title)
}

analyze.all <- function(save.plots = TRUE) {

  # Load data
  ablation <- read.csv(file = "Ablation.csv")
  names(ablation)[names(ablation) == "SCORE"] <- "Score"
  print(ablation)

  ablation.means <- ddply(ablation,
                          .(CellType, Measurement, Time),
                          summarize,
                          mean = mean(Score))
  print(ablation.means)

  # Set up plotting
  if (save.plots) {
    pdf(file = "plot.pdf")
  }

  # Plot all data
  g <- ggplot(ablation, aes(x = Time, y = Score)) +
    geom_point() +
    geom_line(aes(color = Experiment)) +
    facet_grid(Measurement ~ CellType) +
    theme_bw()
  print(g)

  # Plot averages over experiments
  g <- ggplot(ablation.means, aes(x = Time, y = mean)) +
    geom_point() +
```



```
    geom_line(aes(color = CellType)) +
    facet_wrap(~ Measurement) +
    theme_bw()
  print(g)

  # Separate plots of averages over experiments
  for (measurement in levels(ablation.means$Measurement)) {
    g <- ggplot(subset(ablation.means, Measurement == measurement), aes(x = Time, y = mean)) +
      geom_point() +
      geom_line(aes(color = CellType)) +
      labs(title = measurement) +
      theme_bw()
    print(g)
  }

  # Same as above, but using d_ply (note no theme_bw)
  d_ply(ablation.means, .(Measurement), plot.ablation, .print = TRUE)

  # Close plotting device
  if (save.plots) {
    dev.off()
  }
}

analyze.all(FALSE)
```

Note the use of `for` loops and `if` blocks. Control structures such as these are often needed to ensure your script can run autonomously. Here we use an `if` block to control whether plots are saved to a PDF or viewed in RStudio, a technique that can be handy when developing your script.

Also note that the script works when invoked with the appropriate working directory and with an empty environment. It is important that you test this to ensure that you are not dependent on some objects left in your workspace from interactive sessions.

We close by noting that some journals, such as PLoS One, are now requiring scripts such as these to address the problem of imprecise or incomplete descriptions of analysis methods.

