BioEE 758
Week 9

Brief review of lists:

Lists are extremely useful for data storage. They contain *components* that can be of mixed datatype (e.g., a single list may contain character data, numeric data, and logical data), and the components do not have to be of the same length. The **dataframe** is a special type of list where all the components have the same length; think of a **dataframe** as a list where all the components are bound together as columns.

1) Which of the following types of objects gives you the most flexibility in data storage: list, dataframe, or matrix? The least? Why?

Consider the following list:

```
dan.list <- list();
dan.list$friends <- c('Larry', 'Chuck', 'Bud');
dan.list$numbers <- c(7, 11, 13);
dan.list$statement <- c("Better", "than", "Craigs List!");
```

This list 'dan.list' has four components. You can access elements within each component the traditional way, by using the $ symbol (e.g., `dan.list$friends[2]`. Alternatively, you can use the *general method*, which works when your list components are not named. List components are numbered in the order that they are assigned, meaning that component 1 is "friends', component 2 is 'numbers', etc. To access a list component, you need to use the **double square bracket**:

```
dan.list[[1]]
```

And you can check the following:

```
dan.list[[3]] == dan.list$statement
```

What is this expression doing? What is its value and what does it tell you?

Finally, to access individual elements belonging to list components, you

simply do the following:

```
dan.list[[2]][1]
```

Which asks R to go to the second component (numbers) and take the first element (7). What is the output resulting from the following expressions?

```
cat('dans list is ', dan.list[[3]][1:3], '\n', sep=' ');

cat('everyone wants to be on', dan.list[[1]][1], dan.list[[3]][3], sep=' ');
```

The 'double square bracket' bit will be a source of endless confusion when working with strings if you don't understand it.

This week we cover two topics:

1) vectorization (briefly)

2) strings and things

Vectorization

One of the great strengths of R is the ability to perform operations over entire vectors of numbers. In fact, the R language has been optimized for these sorts of operations; in general, you find such vectorization to be far, far faster than non-vectorized calculations. We have already seen some of this in operation:

Consider this example, similar to an in-class exercise we did last week.

```
fx1 <- function(n)
{
    res <- rep(NA, n);
    for (i in 1:n)
        res[i] <- rnorm(1);
    return(res);
}
```

This is a non-vectorized function to generate n random numbers from a standard normal distribution. Rather than vectorization, this approach takes a vector and fills it element by element using a **for** loop.

Contrast the time required for fx1 to generate n = 1000000 random numbers to the time required by `rnorm` to perform the same operation. Why this difference in time? The simple answer is that `rnorm(1000000)` is a vectorized operation; `fx1(1000000)` is not.

You have already used vectorization extensively, probably without realizing it:

```
z<- 1:1000
```

is vectorized.

```
for (i in 1:1000)
    z <- c(z, i)
```

is not.

Perform the following vectorized operations on a vector of integers 1:10:

```
y <- 1:10;
#power
y ^ 2;
#subtract:
y - 3;
#multiply
y * 2;
#modulo
y %% 3;
```

2) Consider the following vectors:

```
x<-1:7
y <- seq(1, 14, by=2);
```

Give the simplest expressions possible to multiply each element of x by the corresponding element in y using using (i) non-vectorized methods, i.e., with a loop, and (ii) vectorized methods, no loops allowed:

3) Suppose you have a vector **A** of integers; you wish to determine whether all of the numbers in **A** share any factors between 1 and 300. Write a function `findFactors` that takes as an argument a vector of integers and tests for common factors on 1:300. By *common factor*, I mean that every number in **A** must be divisible by the factor with a remainder of 0; else it is not a common factor.

If I had a vector `z <- c(31, 62, 93)`, `findFactors(z)` would return 1 and 31. You will need to combine a vectorized operation with a for loop to do this correctly.

Remember Brownie? Here's a vectorization trick that simulates the two-dimensional random walk (no wall this time...) using the function `cumsum()`.

```
x <- cumsum(rnorm(1000));
y <- cumsum(rnorm(1000));
plot(y~x, type='l');
```

4) What does the function `cumsum` do? And why does this work?

Strings, strings, strings.

Strings are things that come between quotation marks, as you already know. A string is a *string of characters* and is interpreted differently than numerical data. Strings are also considered different from expressions to be executed. Normally, when you type an expression in R, we say that R **parses** the expression and stores the expression as a logical sequence of operations, then *executes* or *evaluates* the sequence of operations.

The following examples are character strings:

```
x<- "c(1, 2, 3)"
```

```
x <- "1 2 3"
x <- "1:5"
```

Consider

```
x<- "for (i in 1:10) {print(i);} "
```

Because the expression is contained within quotation marks, R doesn't bother tryi ng to evaluate the expression – it just treats it literally as a string of characters, without *parsing* and *evaluating*.

You can coerce R to treat a character string as an expression to be evaluated, using the functions parse (to parse the string) and eval (to evaluate the parsed string). For example:

```
x<- "for (i in 1:10) {print(i);} "
#thus, x is
x
#but parsed, evaluated x is
eval(parse(text = x))
```

This is actually quite useful to know, but might be a bit abstract at present.

Here is what we need to learn about strings:

i) The difference between a character string and a vector of character strings (you should already know this).

If you cannot immediately and easily answer `length(lps)`, where `lps` is

```
c('jerboa', 'nutria', 'herpestes', 'gulo');
```

you really, really must go back and review information on character strings in vectors, or this homework will prove very difficult. If you don't understand this, I would not read further until you do.

ii) Reading character strings from file

iii) Accessing elements within a string

iv) Decomposing a string into component pieces and putting them back together

v) pattern matching with strings

We will start with (iii). Enter the following as a character string:

```
x <- 'orange horse are happy over apples'
```

One of the most important string manipulation functions is `strsplit()`, which takes a character string and splits it into a character vector, with the elements of the vector determined by the split. If you wanted to split a sentence into a vector of words, you might specify `split = " "` , or a simple space character (since we generally separate words by spaces). Given x above, notice:

```
strsplit(x, ' ')
```
Gives us

```
[[1]]
[1] "orange" "horses" "are"    "happy"  "over"   "apples"
```

Which is simply a list containing a character vector. So, if `z <- strsplit(x, split= ' ')`, then `z[[1]][3]` is 'are', because `z[[1]]` is a vector, where each element is the result of splitting x by the specified value. The double brackets indicate that the strsplit returns a *list* with components. Here, the first component of the list is the vector of character strings created by splitting x. Don't worry about why there is only one component of the list just yet. Just assume that if you send a single character string to strsplit, it will return a list with a single component.

The split can be any value you wish; it is also important to note what happens to the split characters themselves. Try

```
strsplit(x, split = 'a')
```

What happens to the character 'a'?

5) Write a simple function that takes a length-1 character string as an

argument and returns the number of words in the character string. You must use `strsplit`.

If you want to split the character string into individual characters, you can simply do (you should *try* this):

```
strsplit(x, split = '')
```

which says that you want to split the string into all possible characters (literally, 'split the string into all character strings separated by nothing').

6) Modify your function above to a new function, `countChar`, which takes a string as an argument and returns the number of characters present in the string.

Why does `strsplit` return a list with a component that we then have to access using the stupid double –square bracket expression? This seems confusing and unnecessary. The advantage of this approach is that it will let you perform operations on character vectors, such that each component of the list returned by strsplit corresponds to an element of the original vector:

```
x <- c('string cheese', 'string beans', 'guitar string', 'g-string');
```

Consider `strsplit(x, split = 'ing')`.

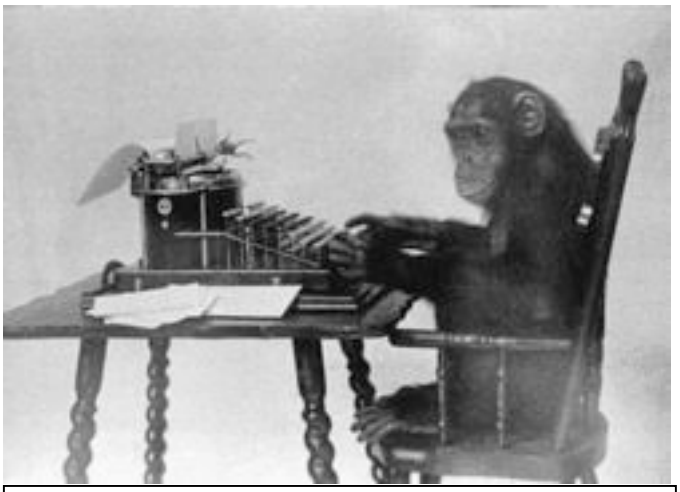
Understand what is happening? The list returned by strsplit has a component for each character string in the original vector (e.g., 'string cheese').

When we are dealing with single character strings (and not vectors with length > 1), it is convenient to coerce our character list into a single vector such that we don't have to use double brackets. This is done using the function `unlist`.

```
x <- "dont enlist. theyll just delist you"
z <- unlist(strsplit(x, ''))
```

You can now treat z as a character vector. You should confirm that it is in fact no longer a list.

7) Why is it dangerous to apply unlist when splitting a vector of character strings?



A participant in a recent study of the feasibility of Darwinian Evolution attempts to reproduce King Lear, Hamlet, and Othello at his keyboard. The results of this study strongly rejected the hypothesis that Blind Chance can explain biological complexity, the origin of time and space[1]*, and Britney's decision to shave her head.*

[1] *From Kent Hovind's $250,000 "You Can't Prove Evolution!" challenge*

8) Modify `countChar` to `countChar_noSpace`, which takes a character string as an argument and counts the number of non-space characters. You know how to do this, but you might have to rephrase the question. `unlist` will make this easier.

Now, having learned how to break strings, we shall consider putting them back together. The reconstruction of strings is performed using the function `paste`. For example, let's consider a scenario where something is broken into pieces:

```
z <- unlist(strsplit('foreign policy', split = ''));
z; #in pieces, broken

paste(z, collapse=''); #fixes it.
```

How easy! The argument collapse in `paste()` will collapse a character vector to a single string, separating elements by whatever value you specify. Example:

```
x<-c("am","n","pl","n","c","n","lp","n","ma")
```

What is z, where

```
z <- paste(x, collapse='a')
```

9) Write a function which returns the inverse of any character string argument. For example, if the function is `invert`, then

`invert('dog')` gives the string `'god'`.

10) Write a function 'scramble' that takes a character string and returns a randomly reshuffled version of the same. Sending the string 'scramble' might give you 'crasblem'.

You can also get frequency data out of character strings. For example, I want to know the frequency distribution of characters in the following:

```
x<- 'April is the cruelest month, breeding Lilacs out of the dead land'

z <- unlist(strsplit(x, ''));
freq<-table(z)
freq;
```

You can even plot a frequency histogram of the character counts; the function `plot()` by default will recognize a table (`freq`, in this case) and plot it accordingly:

11) Molecular evolution people are always interested in the frequency of the bases A, C, T, and G. Write a function `getBaseFreq` that takes a character string corresponding to a filename containing DNA sequence(s) and outputs a vector with *frequencies* of the four bases (remember the trick you learned last week?). You can find a DNA sequence in the file 'skinkDNA.txt'. Before you even think about dealing with the DNA sequence bit, make sure you can create a function that can take a filename as an argument and *open the file* and *read the file content*. You haven't done this file manipulation stuff, but it is *super, super* important. Once you have this function, all you have to do is something like `getBaseFreq('skinkDNA.txt')` to get the required output.

To read character data, you need to use the function `scan`. The help on this function is a bit dense; for your purposes, the most important arguments are `what` and `sep` (figure out what they do and how you can use them to read character data).

Pattern Matching with strings

The simplest pattern-matching tool is `grep`. `grep` takes a vector of character strings and returns the index of each element containing the pattern. For example,

```
x <- c('cat', 'hat', 'at', 'fat', 'bat');
```

Then

```
grep('fat', x)
```

returns the integer 4, since only the 4th element of x contains the string 'fat'. Likewise, if `string` is a character string, `grep(string, x)` will return the indices of x that contain `string` – even if `string` is only a part of the element of x. The letter 'b' occurs only in the 5th element, but there is more than simply 'b' in x[5]. Note the results of

```
grep('b', x).
```

grep will return multiple matches as well:

```
grep('at', x)
```

If you aren't clear on this, make some random strings and/or vectors of character strings, and continue playing with `grep` until you understand it.

12) The file 'beer.txt' contains a list of (surprise!) beers. Each species of beer occurs on a new line. Scan the contents of beer to a character vector, where each element contains the name of a different beer.

Do the following, provide your code and the answer, etc.

a) How many beers contain the phrase "ale"? There is an `ignore.case` argument to scan that might be useful for this.

b) Extract a vector with the names of all beers containing the word "Belgium" (case insensitive)

c) What is the most common word in the entire set of beers? Before you do this, you should convert the entire set of 'beer' to upper case or lower case; you can do this with the functions `toupper` and `tolower`. What is the probability that a randomly chosen word from 'beer' will be the most common word (i.e., what is its frequency?)? This will require several lines of code.

Additional things to know about strings:

Substituting text: the function `gsub` is useful and operates on character vectors as follows:

```
gsub(pattern, replacement, x)
```

where pattern is the pattern to be located, replacement is the string to swap in, and x is the character string (or vector of character strings) to be searched.

Look at:

```
x <- 'April is the cruelest month, breeding Lilacs out of the dead land'
gsub('cruelest', 'warmest', x)
```

And notice that it also pertains to vectors of character strings:

```
x<- c('root', 'coot', 'boot', 'toot', 'flute');
sub_this <- 'a'
gsub('oo', sub_this, x);
```

13) Write a function that takes a string of DNA sequence data as an argument and replaces each of the bases with the following:

```
x <- a
y <- c
z <- g
u <- t
```

The function should use a loop and `gsub` and should return the altered string.

Another important pattern matching function is `gregexpr,` which returns the actual position within a character string containing a particular pattern (in computer jargon, this pattern-to-be-matched is known as a *regular expression*).

Consider:

```
x<- 'the cat in the hat';
gregexpr('cat', x)
gregexpr('at', x)


x<- 'xxgoldxxeeskerjgold';
gregexpr('gold', x)
```

What does `gregexpr` return if the pattern is not found?

Do at least ONE of the following, either the super-string challenge (14), or the DNA sequence translation program (15). Feel free to do both (you may have to the second next week anyway!).

14) Super-string challenge: The file `string_challenge1.txt` contains a character string...a *long* character string. I have hidden a message within this character string, which you will never find without the help of your stellar string manipulation skills. After considering the following points, I want you to write code that extracts the message and prints it to your screen as a new character string (length = 1), containing exactly and only the characters that are supposed to be there.

a) the *target string* contains real sentences/phrases.

b) the *target string* is embedded in a matrix of random letters. I have taken the *target string*, split it into individual letters, and inserted *n* random letters between each character of the target string. So, for example, if the target string was:

```
G: I am the g-string
```

and if I separated each character by two random letters, you would have the following:

```
ajGhb:mn xzIqr agayjmtg iltvfhmkeiu tggyh-rfsedtwarqsiwsnecg
```

Notice that *n* = 2 random letters, and I put *n*=2 random letters before the first real character (G). I did this in the *long string* as well.

c) The random component contains only letters[1:26]. If you observe any other characters, you can assume that they are part of the *target string*. Thus, a good way to start this problem is to find the index values of some character(s) that you are sure must belong to the *target string*. You know how to do this.

d) Then you can use a function you've already written for this homework to find a common factor for all of those numbers between 1 and 300. Since all the characters in the *target string* are equally spaced, their corresponding index values in the long string will be divisible by a common number. In the example above, look at the index values for all characters in `G: I am the g-string` in the long string: write them out if you have to in order to see the pattern.

e) Now: find the *target string*!

****Note:** Make sure you read the long string correctly from file. You want scan() to read a single item (this is critical). Since there are no newline characters in your file, this suggests a possibly convenient way to do this...

15) Write a function that takes a protein-coding DNA sequence and returns a character string of amino acids corresponding to each codon in the sequence. Essentially, you are creating a virtual R-ribosome for sequence translation. If the DNA sequence string is:

```
ATGGTTTCGTAA
```

you would first have to partition it into codons

```
ATG, GTT, TCG, TAA
```

and perform the translation using the appropriate genetic code. In this case, we would end up with

```
Methionine, valine, serine, stop (the termination
codon)
```

Or, using the appropriate 3-letter amino acid code and pasting it back together:

```
MetValSerTer
```

This last line is the sort of output that should be returned by your function (a single character string with amino acid abbreviations).

The `skinkDNA.txt` sequence is a skink cytochrome B DNA sequence (mitochondrial) that you need to translate. I have also included a data file "`genetic_code.txt`" , which is a dataframe of codons along with the respective one and three letter amino acid abbreviations.

This can certainly be done in fewer than 15 lines of code, including any initializations you might need to do.

Summary of functions / ideas from this week

(i) Accessing components of lists; ‘double square brackets’

(ii) vectorization

(iii) strings and things:

functions to know:

```
scan
strsplit
unlist
grep
paste
gregexpr
gsub
```