# Week 2: an interactive session in R

Dan Rabosky

November 1, 2011

Here we will cover some basic aspects of data input and output in R . The purpose of these exercises is to give you a hands-on feeling for working in the R environment.

## 1 intro to working with data

R comes with a number of built-in datasets. We will make limited use of these. In your editor, load the MASS library (which you should have). We'll work with the `Animals` dataset, which contains the brain and body sizes for a number of animal species.

```
> library(MASS);
> data(Animals);
```

I mentioned earlier that - initially - we will be concerned with three fundamental types of data: vectors, matrices, and data frames. `Animals` is a `data.frame` with 2 columns. Now, a `data.frame` is a very special type of data *object* , where each column is a different measured variable or factor.

Each of the rows has a name, and both columns have names. When working with data frames, we should always check basic attributes. Was our data "read in" correctly? You can check this by looking at a few sample rows, or by looking at the dimensions of the data frame, or the "class" of the data frame. For example:

```
> class(Animals);

[1] "data.frame"

> dim(Animals);

[1] 28  2

> nrow(Animals);

[1] 28
```

```
> ncol(Animals);

[1] 2
```

To look at the first five rows of a data frame, you do:

```
> head(Animals);

                   body brain
Mountain beaver     1.35   8.1
Cow               465.00 423.0
Grey wolf          36.33 119.5
Goat               27.66 115.0
Guinea pig          1.04   5.5
Dipliodocus     11700.00  50.0
```

And the function `tail` should look at the last 5 rows.

To access elements of a data frame, we can either reference the name of the column, or the "index value" of the column (e.g., column 1 or column 3). The `data.frame` is like a matrix: it has rows and columns. The syntax for accessing rows and columns is **super, super important**. The syntax is always something like: `MyDataFrame[ whatRow, whatColumn ]` where `whatRow` is the row you want, and `whatColumn` is the column. If you leave either of these blank, you will get all the rows and/or all the columns.
Let's access row 1, but take *all* the columns for that row:

```
> Animals[ 1, ]

               body brain
Mountain beaver 1.35   8.1
```

Now, even though this `data.frame` looks like is has 3 columns, it doesn't. The first column is the "row names". And we can access each row by its name, just as if the name were an index:

```
> Animals[ "Mountain beaver" ,  ]

               body brain
Mountain beaver 1.35   8.1
```

We can view the first column as:

```
> Animals[ , 1]

 [1]     1.350   465.000    36.330    27.660     1.040 11700.000  2547.000
 [8]   187.100   521.000    10.000     3.300   529.000   207.000    62.000
[15]  6654.000  9400.000     6.800    35.000     0.120     0.023     2.500
[22]    55.500   100.000    52.160     0.280 87000.000     0.122   192.000
```

This should just have given us a vector of the values, without the corresponding row names. Most usefully, we can access columns of data frames by the name of the column. We will do this two ways. My preference is to use the "dollar sign" operator, which will give you something like this:

```
> Animals$brain;

 [1]    8.1  423.0  119.5  115.0    5.5   50.0 4603.0  419.0  655.0  115.0
[11]   25.6  680.0  406.0 1320.0 5712.0   70.0  179.0   56.0    1.0    0.4
[21]   12.1  175.0  157.0  440.0    1.9  154.5    3.0  180.0
```

Alternatively, and I rarely do this, you can access the same column like this:

```
> Animals[ , 'brain' ];

 [1]    8.1  423.0  119.5  115.0    5.5   50.0 4603.0  419.0  655.0  115.0
[11]   25.6  680.0  406.0 1320.0 5712.0   70.0  179.0   56.0    1.0    0.4
[21]   12.1  175.0  157.0  440.0    1.9  154.5    3.0  180.0
```

**Exercise 1**: compute the mean and standard deviation for the `log`-transformed body and brain sizes, using the functions `mean` and `sd`. Use the $ operator to access body and brain from `Animals`.

What if you want a particular row and column entry, e.g., a single number? Let's say you want the 5th row, 2nd column entry:

```
> Animals[ 5, 2];

[1] 5.5
```

**Exercise 2**: What if you try to take the 2nd row, 5th column, from `Animals` ?

Now let's try some data exploration using plot. You can spend a lot of time learning options for plotting. For now, let's just look at brain size as a function of body size. We'll do this on a log scale. In fact, let's make new log-transformed variables for `Animals$brain` and `Animals$body`.

```
> log.body <- log(Animals$body);
> log.brain <- log(Animals$brain);
```

Now we'll try plotting them:

```
> plot(x= log.body, y=log.brain);
```

Other ways to do this include:

```
> plot(log.brain ~ log.body);
```

which you can read as "log.brain as a function of log.body". I often use this notation with a tilda. You use the tilda in specifying the relationship between dependent and independent variables in linear models, so it is important to recognize this.

You should get familiar with some of the simple `plot` options. For example, you can tweak this as follows:

```
> plot(log.brain ~ log.body, pch=19, cex=1.5, col='blue');
```

If you look at the help on `plot` and `par` (the latter controls graphical options) you will find explanations of these and many more. `cex` controls the size of points, `pch` controls the type of points (here, 'filled circles'), and `col` is point color. There are far too many options to cover here in plot parameters. Also, you can do exactly what we just did while working directly with the `Animals` data frame, like this:

```
> plot( log(Animals$brain) ~ log(Animals$body), pch=19, cex=1.5, col='blue');
```

And also, you can use the `data` argument to the `plot` function directly, to make the syntax a little easier:

```
> plot( log(brain) ~ log(body), data=Animals, pch=19, cex=1.7, col='gray50');
```

We can also use the histogram function to visualize the distribution of log body mass:

```
> hist(log(Animals$body));
```

So - let's actually try doing a phylogenetically inappropriate analysis of the relationship between log-brain size and log-body size. We can do this with the simple function `lm`, which is Rś general purpose function for linear models.

```
> fit1 <- lm( log(brain) ~ log(body), data=Animals );
> fit1

Call:
lm(formula = log(brain) ~ log(body), data = Animals)

Coefficients:
(Intercept)    log(body)
      2.555        0.496
```

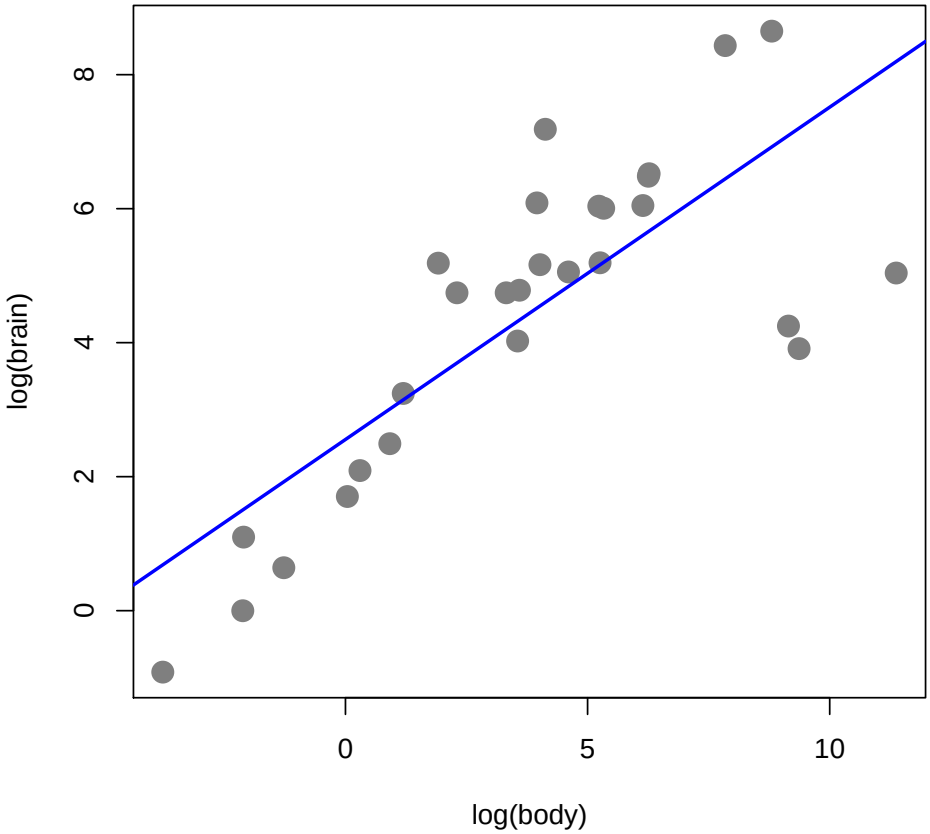
And we can view a more detailed summary of our analysis with:

```
> summary(fit1);
```

```
Call:
lm(formula = log(brain) ~ log(body), data = Animals)

Residuals:
    Min      1Q  Median      3Q     Max
-3.2890 -0.6763  0.3316  0.8646  2.5835

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.55490    0.41314   6.184 1.53e-06 ***
log(body)    0.49599    0.07817   6.345 1.02e-06 ***
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 1.532 on 26 degrees of freedom
Multiple R-squared: 0.6076,        Adjusted R-squared: 0.5925
F-statistic: 40.26 on 1 and 26 DF,  p-value: 1.017e-06
```

Finally, let's plot the data as well as our fitted linear model object:

```
> plot( log(brain) ~ log(body), data=Animals, pch=19, cex=1.7, col='gray50');
> abline(fit1, lwd=2, col='blue');
```




Here's our fitted relationship! Note: abline is a very simplistic way of adding a fitted regression line to a plot. We will do much better in a few weeks.

# 2 Vectors

Vectors are one of the most important data types in RȦ vector is just a one-dimensional array of numbers. We create a vector of numbers when we extract a single column from the `Animals` dataset, e.g. `Animals$brain`. To explicitly construct a vector, we can use the `c()` function, as follows:

```
> myvec <- c(7, 3, 2, 9, 10);
> myvec

[1]  7  3  2  9 10
```

This vector has 5 "numeric" elements, and you can check by assessing the "mode" of the vector:

```
> mode(myvec)

[1] "numeric"
```

Do you see the little `[1]` at the start of your vectors? Don't worry about this: it isn't actually part of your vector. This is just a reference that lets you know the start of your vector (e.g., index = 1). It will make more sense when we look at longer vectors. You can also make vectors of 'character strings'. Character strings are data that are treated literally as text. You can't add them together, multiply them, or anything else. Each character string (an individual data element) is contained in quotes. Here we will make a vector of character strings:

```
> myvec <- c('charlie', 'olga', 'this is not a pizza', '#$%#!$^!');
> mode(myvec);

[1] "character"

> myvec

[1] "charlie"             "olga"                "this is not a pizza"
[4] "#$%#!$^!"
```

How many character strings are in the vector above? As we did earlier, you can check this using the `length` function. Finally, vectors can be 'logical':

```
> myvec <- c(TRUE, TRUE, FALSE, TRUE);
> myvec

[1]  TRUE  TRUE FALSE  TRUE

> mode(myvec);
```

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

Let's focus on numeric vectors for a bit. We can do calculations on vectors. First, let's use a new operator ":" to generate a vector of integers between 5 and 15. The colon operator, used as below, generates an integer sequence of numbers.

```
> newvec <- 5:15;
> newvec

 [1]  5  6  7  8  9 10 11 12 13 14 15
```

Let's try taking the log of the vector:

```
> log(newvec);

 [1] 1.609438 1.791759 1.945910 2.079442 2.197225 2.302585 2.397895 2.484907
 [9] 2.564949 2.639057 2.708050
```

You can see that R sequentially applied the log function to all elements of the vector, generating a new vector of log-transformed values. If we do:

```
> vec1 <- 15:30;
> vec2 <- log(vec1);
```

We created a vector `vec2` using the log of `vec1`. Has this operation modified any of the original values of `vec1` ? Also, consider this operations:

```
> vec1 <- 15:30;
> vec1 - 5
> vec1 / 3
> vec1 * .25
```

What happened? Other handy functions for working with vectors include:

```
> max(vec1);
> min(vec1);
> sum(vec1);
```

Without even giving a vector a "name", you can do calculations on it:

```
> sum(1:50)

[1] 1275
```

**Exercise 3**: Compute the sum of all log-transformed integers between 2 and 123456.

# 3 Vector indexing

To access an element of a vector, you just give the index of the element you want. Suppose we have a vector:

```
> myvec <- c('alpha', 'beta', 'gamma', 'nu', 'lambda');
> myvec;

[1] "alpha"  "beta"   "gamma"  "nu"     "lambda"

> myvec[2];

[1] "beta"

> myvec[4];

[1] "nu"

> myvec[1:2];

[1] "alpha" "beta"

> myvec[3:5];

[1] "gamma"  "nu"     "lambda"
```

Do you understand what is happening in the last example? We indexed elements 3, 4, and 5 of the vector by putting a vector of integers in the square brackets. In fact, we can always take elements of vectors by using another vector of the indices that we want. We can even use this to generate new vectors with repeated elements:

```
> myvec[ c(2,2,2,2,2,2, 3,3,3,3,3,3) ];

 [1] "beta"  "beta"  "beta"  "beta"  "beta"  "beta"  "gamma" "gamma" "gamma"
[10] "gamma" "gamma" "gamma"
```

Vectors can also have names, and we can use these names like we did earlier with data frames. Here, we'll (i) make a vector, (ii) assign a name to each element of the vector, and (iii) use those names to access elements of the vector:

```
> myvec <- c( 50, 75, 25, 35 );
> names(myvec) <- c('Sceloporus', 'Varanus', 'Ctenotus', 'Draco');
> myvec[ 'Draco' ];
> myvec[ 'Varanus' ];
> myvec[ 'Anolis' ];
```

And you can also use vector indexing to change the value of elements of a vector:

```
> myvec
> myvec[3] <- 0.2571;
> myvec
```

There are a number of ways to make vectors from scratch. Many functions in R automatically generate vectors. For example, the random number generator `rnorm` does this:

```
> rnorm(25);
```

In fact, many of the functions you will use will return a vector of numbers. What if you wanted to make a vector of 10 "ones"? You could do:

```
> myvec <- c(1,1,1,1,1,1,1,1,1,1);
```

but it is much easier to use the "replicate" function, and do:

```
> myvec <- rep(1, 10);
```

Obviously, this makes it easy to make a vector of arbitrary length.

# 4 Logical comparisons and logical vectors

Here we will learn the very special technique of 'logical comparisons'. R has a set of operators that perform logical comparisons. Here we will go back to the `Animals` data frame. Here, we are going to make a reduced version of the 'body' data.

```
> body <- Animals$body;
> names(body) <- rownames(Animals);
> body;
```

```
 Mountain beaver             Cow        Grey wolf             Goat
           1.350         465.000           36.330           27.660
       Guinea pig      Dipliodocus   Asian elephant           Donkey
           1.040        11700.000         2547.000          187.100
          Horse     Potar monkey              Cat          Giraffe
        521.000           10.000            3.300          529.000
        Gorilla           Human African elephant      Triceratops
        207.000           62.000         6654.000         9400.000
  Rhesus monkey         Kangaroo   Golden hamster            Mouse
          6.800           35.000             0.120            0.023
         Rabbit            Sheep           Jaguar       Chimpanzee
          2.500           55.500          100.000           52.160
            Rat    Brachiosaurus             Mole              Pig
          0.280        87000.000             0.122          192.000
```

Now we've take a column from a dataframe and made a new vector. This vector has names. Now, what if we want to know which elements of our vector have a body size greater than 1000 Kg? We just do:

```
> body > 1000
```

```
 Mountain beaver              Cow        Grey wolf             Goat
           FALSE            FALSE            FALSE            FALSE
      Guinea pig      Dipliodocus   Asian elephant           Donkey
           FALSE             TRUE             TRUE            FALSE
           Horse     Potar monkey              Cat          Giraffe
           FALSE            FALSE            FALSE            FALSE
         Gorilla            Human African elephant      Triceratops
           FALSE            FALSE             TRUE             TRUE
   Rhesus monkey         Kangaroo   Golden hamster            Mouse
           FALSE            FALSE            FALSE            FALSE
          Rabbit            Sheep           Jaguar       Chimpanzee
           FALSE            FALSE            FALSE            FALSE
             Rat    Brachiosaurus             Mole              Pig
           FALSE             TRUE            FALSE            FALSE
```

You just generated a new vector, a logical vector of `TRUE` and `FALSE` elements created by doing some logical test on each element of `body`. You can actually assign these elements to a new vector like this:

```
> compare_vec <-  body > 1000;
> compare_vec
```

```
 Mountain beaver              Cow        Grey wolf             Goat
           FALSE            FALSE            FALSE            FALSE
      Guinea pig      Dipliodocus   Asian elephant           Donkey
           FALSE             TRUE             TRUE            FALSE
           Horse     Potar monkey              Cat          Giraffe
           FALSE            FALSE            FALSE            FALSE
         Gorilla            Human African elephant      Triceratops
           FALSE            FALSE             TRUE             TRUE
   Rhesus monkey         Kangaroo   Golden hamster            Mouse
           FALSE            FALSE            FALSE            FALSE
          Rabbit            Sheep           Jaguar       Chimpanzee
           FALSE            FALSE            FALSE            FALSE
             Rat    Brachiosaurus             Mole              Pig
           FALSE             TRUE            FALSE            FALSE
```

And this should not have changed any of the values of the `body` vector. (verify this). The critical logical operations you need to be aware of are:

- "Greater than", or `>`
- "Greater than or equal to", or `>=`
- "Less than", or `<`
- "Less than or equal to", or `<=`
- "Exactly equal to", or `==`. Note that this is **DOUBLE** equals signs!
- "Not equal to", or `!=`.
- "AND", or `&`. This operator lets you combine 2 or more statements together. If you wanted to write "x is greater than 50 and x is less than 100", you would do: `x > 50 & x < 100`.
- "OR". This is done with the `|` character. e.g., if you wanted maple bacon or cheesecake ice cream, you would write `x == 'maple bacon' | x == 'cheesecake'` .

You'll need to be fluent in these "operators" to work effectively in R . Logical operators let us find members of dataframes, vectors, or matrices that meet some particular criteria. For example, we can find the animal with a body mass equal to 87000 kg:

```
> body == 87000
```

```
 Mountain beaver             Cow        Grey wolf             Goat
           FALSE           FALSE            FALSE            FALSE
     Guinea pig      Dipliodocus   Asian elephant           Donkey
           FALSE           FALSE            FALSE            FALSE
           Horse     Potar monkey              Cat          Giraffe
           FALSE           FALSE            FALSE            FALSE
        Gorilla           Human African elephant      Triceratops
           FALSE           FALSE            FALSE            FALSE
  Rhesus monkey         Kangaroo   Golden hamster            Mouse
           FALSE           FALSE            FALSE            FALSE
         Rabbit           Sheep           Jaguar       Chimpanzee
           FALSE           FALSE            FALSE            FALSE
             Rat    Brachiosaurus             Mole              Pig
           FALSE            TRUE            FALSE            FALSE
```

What if you want to know how many animals have body mass greater than 50 Kg? You can get the indices of the elements meeting this criterion using the `which` function:

```
> which(body > 50);
```

```
            Cow      Dipliodocus   Asian elephant           Donkey
              2                6               7                8
          Horse          Giraffe         Gorilla           Human
              9               12              13               14
African elephant      Triceratops           Sheep          Jaguar
             15               16              22               23
      Chimpanzee    Brachiosaurus             Pig
             24               26              28
```

and the number of species meeting this criterion is just:

```
> bigbodies <- which(body > 50);
> length(bigbodies);

[1] 15
```

If you actually want the values - not just the index values - you can also use `which` directly, as follows:

```
> body[ which(body > 50) ]
```

```
            Cow      Dipliodocus   Asian elephant           Donkey
         465.00         11700.00         2547.00           187.10
          Horse          Giraffe         Gorilla           Human
         521.00           529.00          207.00           62.00
African elephant      Triceratops           Sheep          Jaguar
        6654.00          9400.00           55.50          100.00
      Chimpanzee    Brachiosaurus             Pig
         52.16         87000.00          192.00
```

**Exercise 4**: In a single line of code for each: how many animals have body sizes less than 20 Kg? How many have body sizes less than 100 but greater than 30?

**Exercise 5**: How many animals have log-transformed body sizes greater than 4? What is the mean of the log-transformed values for this subset of the data?

# 5 Accessing vector and dataframe member elements with logical statements

The mechanics of logical comparisons and subsetting are critical to understand. Consider:

```
> body <- Animals$body;
> brain <- Animals$brain;
```

Now we have not added a 'names' attribute to our body or brain vectors. Here we'll make a new vector - a logical vector - of `TRUE` and `FALSE` for all elements where body size is greater than 100:

```
> bigbodieslogical <- body > 100;
```

We can use this logical vector directly to make a smaller vector that ONLY contains the body sizes that meet the criterion. In effect, by using a logical vector for indexing, we tell R to make a new vector by taking only the `TRUE` values and discarding all of the `FALSE` values:

```
> bigbodies <- body[ bigbodieslogical ];
> bigbodies
```

Do you understand what is going on here? This is one of the most important things we will learn! You can also use this logical vector to make a vector of all of the brain sizes for which the BODY SIZE is greater than 100 Kg:

```
> newbrains <- brain[ bigbodieslogical ];
> newbrains
```

What if we want the brain size corresponding to the animal with the largest body size? First, we'll make a logical vector that assigns a `TRUE` to the position where the body size is the maximum, and `FALSE` everywhere else. Then we use this to access the corresponding brain size:

```
> x <- body == max(body)
> brain[x];
```

Does this look right? Check against the `Animals` dataframe. We can use these tricks with data frames as well. We can make a new data frame for which all of the species have (say) a body size less than 10 Kg.

```
> x <- body < 10;
> newdf <- Animals[ x,  ];
> newdf;
```

```
                 body brain
Mountain beaver 1.350   8.1
Guinea pig      1.040   5.5
Cat             3.300  25.6
Rhesus monkey   6.800 179.0
Golden hamster  0.120   1.0
Mouse           0.023   0.4
Rabbit          2.500  12.1
Rat             0.280   1.9
Mole            0.122   3.0
```

We told R : Take all of the rows of the `Animals` data frame where the body size is less than 10. We can actually work directly with the data frame as follows. Try to understand exactly what is going on in each of the following examples:

```
> Animals[Animals$body > 10, ];
> Animals[Animals$brain == max(Animals$Brain), ];
> Animals[Animals$brain < 10 | Animals$body > 1000, ];
```

And you can use the function `which` to directly get the indices of the data frame rows that meet some logical criterion:

```
> which(Animals$brain < 10 | Animals$body > 1000);
> which(Animals$brain > 50);
```

You can (and will) be using logical statements such as these to make subsets of data frames and vectors. For example, here we will partition the `Animals` dataframe into 2 subsets: one with all species less than the median body size, and the other with body sizes greater than the median:

```
> AnimalsSmall <- Animals[ Animals$body < median(Animals$body),  ];
> AnimalsBig <- Animals[Animals$body > median(Animals$body), ];
```

**Exercise 6**: Repeat the linear model fitting exercise we did earlier for log-brain size as a function of log-body size, for the BIG and SMALL subsets of the animals dataframe. Use log-transformed values. Do the slopes differ?

# 6 Homework

1. In a single line of code, generate a vector of all integers that are multiples of 10, from 10 and 500. There are several ways to do this, only one of which you've learned yet!

2. What happens if you divide a vector by another vector of equal length? Try this with some integer vectors of length 5. What if you multiply two vectors together? Note: these results will only make sense if your vectors are of equal length!

3. Create a vector, in order, of the letters: R, T, E, C, V, O. Create a second (numeric) vector of index positions and use it to create a third vector that spells a word that appears twice in this sentence.

4. **A handy trick:** In logical statements, `TRUE` has a value of 1 and `FALSE` has a value of 0. You can exploit this property in concert with the `sum` function to quickly count the number of elements of vectors that meet some particular condition. Create a logical vector corresponding to `Animals` brain sizes greater than or equal to 10, and count the number of `TRUE` values using `sum`.

5. Read in the data table agamids_morphometrics.csv. You will use the `read.table` function, and you will have to tell the function how to process the input. Is there a header row with variable names? What separates the data? Here, the data are 'comma separated values', hence the .csv extension on the filename. Often, you will want to look at the data file in a text editor (textwrangler, notepad++, etc, but NEVER notepad) to check it out before trying to read it in R . Write single lines of code that check the dimensions of the data frame.

6. Make a new data frame with the first 3 rows of the agamid lizard dataframe. Make yet another dataframe with rows 20 to 25. Bind these rows together into another dataframe. You can bind rows of dataframes together using the `rbind` function, but make sure both of your subsetted dataframes are correct (check them by looking at them!) before doing this!. Write a statement that makes a vector of the species included in the new dataframe.

7. **Another trick:** Look at the row names for the 58 agamid species in the data frame. These are the species names. We are going to make a reduced data frame that contains ONLY the species in the genus *Ctenophorus* . We will do this using the function `grep`. This function will be explored in much more detail later, but for now, all you need to know is that it is a 'pattern matching' function. If you give it a vector of character strings and a "target string", it will return the index values of all elements that contain the target. For example, suppose you want the row with data from the frilled dragon, *Chlamydosaurus kingii*. You could just do:

   ```
   > agamids['CHLAMYDOSAURUS_KINGII', ];
   ```

   But this only works if you remember how to spell the species name exactly as it is. What if all you remember is 'kingii'? Then you can exploit the `grep` function as follows:

   ```
   > myindex <- grep('KINGII', rownames(agamids));
   > myindex;
   > agamids[ myindex,  ];
   ```

   See how this works? the `grep` function (look at help) finds a **pattern** within a particular target vector (in this case, the row name vector from the agamid data), and returns the index values of all the elements where the target is found.

   Use the `grep` function to make a new data frame with only species from the genus Ctenophorus. How many species are in the dataset from this genus?

8. Make a second dataframe for the genus *Diporiphora*, just as you did for *Ctenophorus*. Write an expression that counts the number of *Diporiphora* species in the dataset.

9. Morphometrics. We will do a "non-phylogenetic" analysis of the relationship between lizard snout vent length (svl) and femur length (variable named 'sha', for shank). Fit a simple linear model to sha as a function of svl for both the Ctenophorus and Diporiphora data.

10. You can set up a 2 panel plot using the graphical parameter `par(mfrow = c(2,1)`. Do this. Plot sha as a function of svl for Ctenophorus, then add the fitted regression line. Then, without closing your plot window, add a second plot for the Diporiphora data and also add the fitted line. When you have figured out how to do this, make sure that both plots have exactly the same x and y axes, so they can be compared visually. You can do this with the `xlim` and `ylim` graphical parameters, which are arguments that go to the `plot` function.

11. How many agamid species have an svl greater than 4.5? How many have an svl greater than 4.5 and a tail length greater than 5.25? How many have an svl greater than 4.5 OR a tail length greater than 5.25?

12. Compute the mean, median, and standard deviation (functions `mean`, `median`, `sd`) for the first 2 columns of the Ctenophorus data frame. What variables are these (extract the names of the variables using R code!) ?

13. Compute the mean, median, and standard deviation (functions `mean`, `median`, `sd`) for columns 3, 6, and 7 of the Diporiphora data frame. What variables are these?

14. Write a line of code that pulls out the ONLY the name of the species with the largest svl. Do it again for the species with the smallest svl.

15. First, use what you know about vector operations to make a new vector that contains the UNLOGGED svl's for all agamid species (to reverse a `log` transformation, you use the `exp` function). Call this new vector 'svl_unlogged'. Now, make a new dataframe by "binding" this vector to the agamids dataframe. This is a column vector, and you can make a new data frame with the function `cbind`. Look at the first 5 rows of your new data frame. How does it look? Did it work? Finally, in one line of code, create a new data frame of *all rows where the unlogged svl is within 120 mm of the MAX value* . How many species are in this?

16. Read the dataframe 'biogeography.csv'. This contains 2 columns, including species names and an integer code corresponding to the geographic regions in which they are located. Use the `sum` trick on a logical vector to count the number of species in region 0. Then count the number that are not found in region 1. And another line of code to count the number of species NOT found in EITHER region 2 or region 3!

17. On your own: make sure you have updated to R 2.14. Install the packages nlme, diversitree,