Workshop 1: Introduction to R

The goal of this first workshop is to get you started working in R, and to introduce the most commonly-used data types, operations, and functions.

Try out the command line

The command line in the R console is where you interact with R. The command prompt is a > symbol.

Calculator

At its most basic, the command line is a calculator. The basic operations are

```
+
-
*
/
```

for addition, subtraction, multiplication, division. Familiar calculator functions also work on the command line as built in functions within R. For example, to take the natural log of 2, enter

```
log(2)
```

Note that R interprets the end of a line of input as the end of a command. This is in contrast to some other langues, such as C, which require a symbol such a semicolon to indicate the end of a command.

1. Try the calculator out to get a feel for this basic application and the style of the output. Try `log` and a few other functions (find some online).

2. In R you can store or assign numbers and character strings to named variables called vectors, which are a type of "object" in R. For example, to assign the number 3 to a variable `x`, use

   ```
   x <- 3
   ```

   Note that the `=` symbol also works for assignment, however, in other contexts, the two symbols are different and so it is more standard convention to use `<-`. Try assigning a single number to a named variable.

3. In R you can also assign character strings (enter using single or double quotes) to named variables. Try entering

```
z <- 'Hello' # single or double quotes needed
```

4. At any time, enter `ls()` to see the names of all the objects in the working R environment. You can save the environment for later use by entering `save.image()` or by saving when you exit R. Assign a single number to the variable `x` and another number to the variable `y`. Then watch what happens when you type an operation, such as

   ```
   x * y
   ```

   Finally, you can also store the result in a third variable.

   ```
   z <- x * y
   ```

   To print the contents of z, just enter the name on the command line, or enter print(z).

5. The calculator will also give a TRUE or FALSE response to a logical operation. Try one or more variations of the following examples on the command line to see the outcome.

   ```
   2 + 2 == 4      # Note that == for logical 'is equal to'
   3 <= 2          # less than or equal to
   'A' > 'a'       # greater than
   'Hi' != 'hi'    # not equal to (i.e.,\R{} is case sensitive)
   ```

Vectors

Vectors in R are used to represent variables. R can assign sets of numbers or character strings to named variables using the `c()` command, for concatenate. R treats a single number or character string as a vector containing just one element.

```
x <- c(1,2,333,65,45,-88)
```

1. Assign a set of 10 numbers to a variable `x`. Make sure it includes some positive and some negative numbers. To see the contents afterward, enter `x` on the command line. Is it really a vector? Enter `is.vector(x)` to confirm.

2. Use integers in square brackets to access specific elements of vector `x`.

   ```
   x[5] # fifth element
   ```

   Try this out. See also what happens when you enter vectors of indices,

   ```
   x[1:3] # 1:3 is a shortcut for c(1,2,3)
   x[c(2,4,9)]
   x[c(-1,-3)]
   ```

   Print the 3rd and 6th elements of `x` with a single command.

3. Some functions of vectors yield integer results and so can be used as indices too. For example, enter the function

   ```
   length(x)
   ```

   Since the result is an integer, it is ok to use as follows,

   ```
   x[length(x)]
   ```

4. Logical operations can also be used to generate indicators. First, enter the following command and compare with the contents of `x`,

```
x > 0
```

Now enter

```
x[x > 0]
```

Try this yourself: print all elements of `x` that are non-negative.
The `which` command will identify the elements corresponding to `TRUE`. For example, try the following and compare with your vector `x`.

```
which(x > 0)
```

5. Indicators can be used to change individual elements of the vector `x`. For example, to change the fifth element of `x` to `0`,

```
x[5] <- 0
```

Try this yourself. Change the last value of your `x` vector to a different number. Change the 2nd, 6th, and 10th values of `x` all to 3 new numbers with a single command.

6. Missing values in R are indicated by `NA`. Try changing the 2nd value of `x` to a missing value. Print `x` to see the result. You can use the `is.na(x)` command to identify which values are `NA`. See what the following gives you

```
x[!is.na(x)]
```

7. `R` can be used as a calculator for arrays of numbers too. To see this, create a second numerical vector `y` of the same length as `x`. Now try out a few ordinary mathematical operations on the whole vectors of numbers,

```
z <- x * y
z <- y - 2 * x
```

Examine the results to see how R behaves. It executes the operation on the first elements of `x` and `y`, then on the corresponding second elements, and so on. Each result is stored in the corresponding element of `z`. Logical operations are the same,

```
z <- x >= y               # greater than or equal to
z <- x[abs(x) < abs(y)]   # absolute values
```

What does R do if the two vectors are not the same length? The answer is that the elements in the shorter vector are "recycled", starting from the beginning. This is basically what R does when you multiply a vector by a single number. The single number is recycled, and so is applied to each of the elements of `x` in turn.

```
z <- 2 * x
```

Functions

One of the beautiful features of R is how easy it is to write your own functions. This is a great way to stream-line your code and minimize repetition (which is also a good way to prevent copy-paste errors).

1. Syntax for function construction is as follows

```
myfunction1 <- function(x) {
  out <- 2*x^2 + 3
  return(out)
}
```

   Here the `return` command tells R what value to return from the function call. The following will call this function with `x=5`, and assign the output to a value `z`

```
z <- myfunction1(5)
z # to look at the output
```

   Try this function with some different values of `x`.

2. Now try typing the variable name `'out'` into the R console. This should cause an error:

```
Error: object 'out' not found
```

   The variable `out` is only defined *locally* within `myfunction1`, so if you try to access it from "outside" the function, it won't exist.

3. Construct your own function with two arguments (call them `x` and `y`). Name this function `myfunction2`. Try calling this function with some different combinations of `x` and `y`.

4. You can use various `apply` statements (there are many - `lapply`, `mapply`, `sapply`, `tapply`, `rapply`) to apply functions to vectors, lists, matrices, etc. For example, we could apply `myfunction1` to the vector we created earlier using `sapply`:

```
x <- c(1,2,333,65,45,-88)
sapply(x, myfunction1)
```

Analyze vector of data: flying snakes

Paradise tree snakes (*Chrysopelea paradisi*) leap into the air from trees, and by generating lift they glide downward and away rather than plummet. An airborn snake flattens its body everywhere except for the heart region. It forms a horizontal "S" shape and undulates from side to side. By orienting the head and anterior part of the body, a snake can change direction, reach a preferred landing site, and even chase aerial prey. To better understand lift and stability of glides, Socha (2002, Nature 418: 603–604) videotaped eight snakes leaping from a 10m tower. One measurement taken was the rate of side-to-side undulation. Undulation rates of the eight snakes, measured in Hertz (cycles per second), were as follows:

```
0.9 1.4 1.2 1.2 1.3 2.0 1.4 1.6
```

We'll store these data in a vector (variable) and try out some useful vector functions in R.

1. Put the glide undulation data above into a named vector. Afterward, check the number of observations stored in the vector.

2. Apply the `hist` command to the vector and observe the result (a histogram). Examine the histogram and you will see that it counts two observations between 1.0 and 1.2. Are there any measurements in the data between these two numbers? What is going on? The default in R is to use right-closed, left-open intervals. To change to left-closed right-open, modify an option in the hist command as follows,

   ```
   hist(myvector, right = FALSE)
   ```

   We'll be doing more on graphs next week.

3. Hertz units measure undulations in cycles per second. The standard international unit of angular velocity, however, is radians per second. 1 Hertz is 2π radians per second. Transform the snake data so that it is in units of radians per second (note: `pi` is a programmed constant in R).

4. Using the transformed data hereafter, **write a function** to calculate the sample mean undulation rate *without* using a call to `mean()` (hint: `sum` and `length` may be useful).

5. Ok, try the function `mean()` and compare your answer.

6. Calculate the sample standard deviation in undulation rate by writing your own function and *without* using calls to `var()` or `sd()`. Then calculate using `sd()` to compare your answer**.

7. Sort the observations using the `sort()` command.

8. Calculate the median undulation rate. When there are an even number of observations (as in the present case), the population median is most simply estimated as the average of the two middle measurements in the sample.

9. Calculate the standard error of the mean undulation rate. Remember, the standard error of the mean is calculated as the standard deviation of the data divided by the square root of sample size.

*8.63938, **2.035985

Data frames

Data frames are objects where each column within a data frame is a vector. We can look at the structure of the data frame using the function `str()`.

1. Make a data frame called `mydata` from the two vectors, `x` and `y`. A great tool to use while using R is to search in google what we do not know what to do. Try searching how to make a data frame. Print `mydata` on the screen to view the result. If all looks good, remove the vectors `x` and `y` from the R environment using the `rm` command. They are now stored only in the data frame. Type `names(mydata)` to see the names of the stored variables. If you are using RStudio you should see that `x` and `y` no longer appear in the environment tab (upper right hand side).

2. Vector functions applied to data frames may give unexpected results - data frames are not vectors. For example, `length(mydata)` won't give you the same answer as `length(x)` or `length(y)`. But you can still access each of the original vectors using `mydata$x` and `mydata$y` (or `mydata[,'x']` and `mydata[,'y']`). Notice the "," before the x and y. When we are dealing with two dimensional objects (like a data frame with rows and columns) we subset it by stating mydata[rows, columns]. Try printing one of them. All the usual vector functions and operations can be used on the variables in the data frame. We'll do more with data frames below.

Anolis lizards in a data frame

Here we will read data on several variables from a comma-delimited (.csv) text file into a data frame, which is the usual way to bring data into R. The data are all the known species of *Anolis* lizards on Caribbean islands, the named clades to which they belong, and the islands on which they occur. A subset of the species is also classified into "ecomorphs" clusters according to their morphology and perching habitat. Each ecomorph is a phylogenetically heterogeneous group of species having high ecological and morphological similarity. The list was compiled by Jonathan Losos from various sources and are provided in the Afterword of his book (Losos 2009. Lizards in an evolutionary tree. University of California Press).

1. Download the file anolis.csv (click file name to initiate download) and save in a convenient place. We suggest that you allocate a separate directory for this class.

2. Open a new script file to write and submit your commands (or cut and paste to the command window) for the remainder of this section. To open a new script file in RStudio you can use file > new file >[your R script]. R scripts are a very useful way to interact with R. By writing your code in scripts you can easily re-run commands and this will make your work more reproducible.

3. In order to read data into R, we need to know the address where the data is stored and set that place as our working directory. Alternatively, you set your working directory to the location where the data is stored using the `setwd()` command. For example:

```
setwd('~/Desktop/BISC869')
```

   In RStudio, look in the lower right panel (you should see the tabs "Files", "Plots", "Packages" etc.). With the "Files" tab open, look to the very right, directly underneath refresh, for the "..." symbol and click it. This will open a new window showing all the places where you can store a file in the computer. Navigate to the folder where your data is held. This could be in "Documents" or "Downloads" or "Desktop". Click open and everything within this folder should appear in the white space of the "Files" panel. The final step is to click the blue gear to the right of the refresh symbol and choose "Set as Working Directory". In RStudio, you can press tab after the quotation marks to auto-complete. This will help you to reduce spelling mistakes.

4. Read the data from the file into a data frame (e.g., call it `mydata`) using the `read.csv` command. For this first attempt, include none of the recommended options for the

read.csv command, so we can explore R's behavior. Remember to put quotation marks around the name of the file. By default, read.csv will convert all columns with character data to factors. A factor is like a character variable except that its unique values represent "levels" that have names but also have a numerical interpretation.

5. Use the str command to obtain a compact summary of the contents of the data frame. Every variable shown should be a factor (because they are all character data). Another way to check the type of a specific variable in the data frame is to use the is.factor command, e.g.,

```
is.factor(mydata$Island) # returns TRUE or FALSE $
```

   Another useful command is class, which will tell you what data type your object is. Try it out on both mydata$Island and mydata.

6. Use the head command to inspect the variable names and the first few lines of the data frame (or tail to inspect the last few lines). Every variable in this data set contains character strings.

7. Let's focus on the variable "Ecomorph", since it has a manageable number of categories. Since "Ecomorph" is a factor, it will have "levels" representing the different groups. Use the levels command to list them. Notice anything unexpected? One of the categories is an empty character string. A couple of the groups appear to be listed twice. But look more closely - are they really duplicates?

8. Use the table function on the "Ecomorph" vector to see the frequency (number of entries) belonging to each named group. See, for example, that one species belongs to the "Trunk-Crown " (trailing space) group rather than to the "Trunk-Crown" (no spaces). Use the which() command to identify the row with the typo.

9. Using assignment (<-), fix the single typo. Use the table function afterward to check the effect of your change.

10. Weirdly, the now-eliminated category of "Trunk-Crown " (trailing space) is still present in the frequency table. This is because, even though no species belong to this category, the category remains a factor level! Confirm this using the levels function. This confusing behavior is one reason why I recommend you avoid reading character variables in as factors. The presence of factor levels with no members can wreak havoc when fitting models to data. One way to delete unused levels of a factor variables is with the droplevels command.

```
mydata$Ecomorph <- droplevels(mydata$Ecomorph)
```

    Check that this solved the problem.

11. Re-read the data from the file into R. This time, use the read.csv function with options to 1) keep character data as-is; 2) strip leading and trailing spaces from character string entries, minimizing typos; and 3) treat empty fields as missing rather than as words with no letters.

How will you even know how to change these settings?

With more general questions about how a particular function or package works in R, we would recommend consulting the relevant R help files. Let?s say we wanted to pull up the help file for the read.csv() function. We can do so using two different lines of code:

`help(read.csv)` OR `?read.csv`

At the very top of the file, you will see the function itself and the package it is in (in this case, it is base). Next is a description of what the function does. You?ll find that the most helpful sections on this page are "Usage", "Arguments" and "Examples". "Usage" give you an idea of how you would use the function when coding–what the syntax would be and how the function itself is structured. "Arguments" tells you the different parts that can be added to the function to make it more simple or more complicated. Often the "Usage" and "Arguments" sections don?t provide you with step by step instructions. Instead, they provide users with a general understanding as to what the function could do and parts that could be added. At the end of the day, the user must interpret the help file and figure out how best to use the functions and which parts are most important to include for their particular task. The "Examples" section is often the most useful part of the help file as it shows how a function could be used with real data. It provides a skeleton code that the users can work off of.

Note: Not all help files for functions are created equally and some can actually cause more confusion than help. In cases like this, it can sometimes be helpful to examine the help file for the original package that the function comes from. For example, the help file for mutate() from the dplyr package is quiet sparse on information. Yet, if we look at the dplyr package vignette, all the information we could possible need about the mutate() function is laid out for us.

12. Use `table` once more to tally up the numbers of species in each Ecomorph category. Is there an improvement from the previous attempts? Which is the commonest Ecomorph and which is the rarest?

13. What happened to the missing values? Use `table` but using the `useNA = 'ifany'` option to include them in the table. In this data set, `NA` refers to lizard species that do not belong to a standard ecomorph category, so it is worthwhile to include them. Perhaps they should be given their own named group ("none"), which is less ambiguous than `NA`.

14. How many *Anolis* species inhabit Jamaica exclusively?*

15. What is the total number of Anolis species on Cuba?** This is not the same as the number occurring exclusively on Cuba — a few species live there and also on other islands. Figure out an elegant way in R to count the number of species that occur on Cuba. Bonus points for the briefest command! [Hint: check the vector functions for character data on the R tips Vector page.]

16. What is the tally of species belonging to each ecomorph on the four largest Caribbean islands: Jamaica, Hispaniola, Puerto Rico and Cuba?*** Try to figure out a solution before looking at one below. (The solution below works but is not very economical. Try to come up with a more elegant solution yourself.)

17. What is the most frequent ecomorph for species that do not occur on the four largest islands?****

*6

**63

*** Note: the below works because no single species appears on multiple big islands

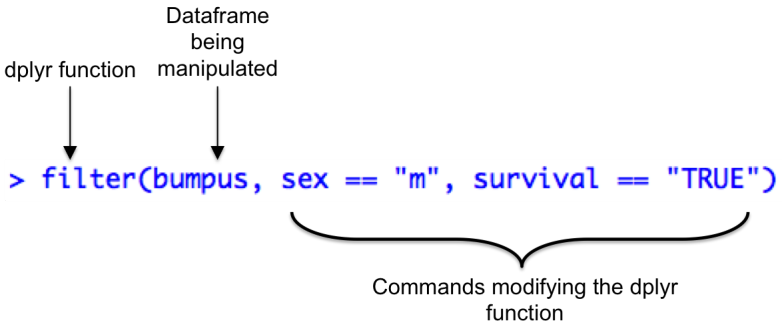
```
big.4 <- c('Cuba', 'Hispaniola', 'Jamaica', 'Puerto Rico')
t(sapply(big.4, function(s)
table(anolis[grep(s, anolis[,'Island']), 'Ecomorph'])))

Crown-Giant Grass-Bush Trunk Trunk-Crown Trunk-Ground Twig
Cuba                   6         15     1            7           14
      5
Hispaniola             3          7     6            4            9
      4
Jamaica                1          0     0            2            1
      1
Puerto Rico            2          3     0            2            3
      1
```

****Trunk-Crown

dplyr

dplyr is a helpful package employed when rearranging, modifying and manipulating datasets. You will be using this package frequently during labs. Please install and load this package into R using the `install.packages()` and `library()` functions respectively. We are going to talk about three specific functions in the dplyr package: `filter()`, `mutate()` and `select()`. However, before we enter into the nitty-gritty of how these functions are applied, we should quickly talk about the structure of dplyr functions. Conveniently, dplyr functions all follow the same basic structure: The first part of this line of code is the dplyr function itself, applied

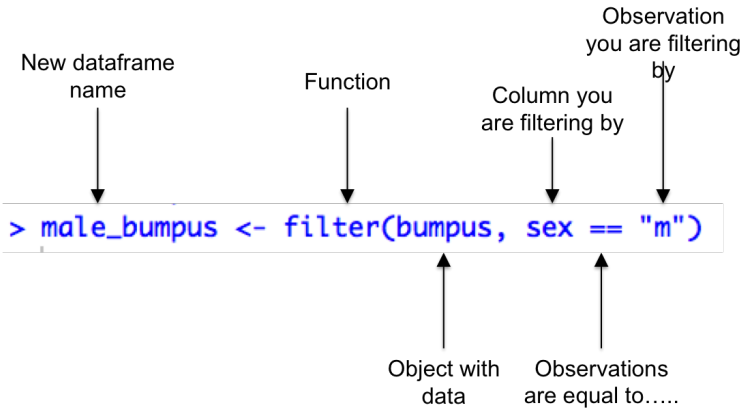


as any other function in R might be. The second part is the name of the dataframe you will be manipulating placed at the beginning of the function bracket. In this case, our dataframe is called "bumpus" which contains morphometric data from a population of sparrows. You will encounter this data frame further in the following examples and in the questions found at the end of this exercise. The third part of this line of code is a series of commands used to specify exactly what the dplyr function is doing. There can be as many as one to thousands of commands applied to a single dplyr function to make it more general of specified.
In preparation for the next few examples, please download and load the bumpus.csv dataset into R under the object name "bumpus". The data are all measurements of sparrows caught in a wind storm in the 1880s.

dplyr: filter()

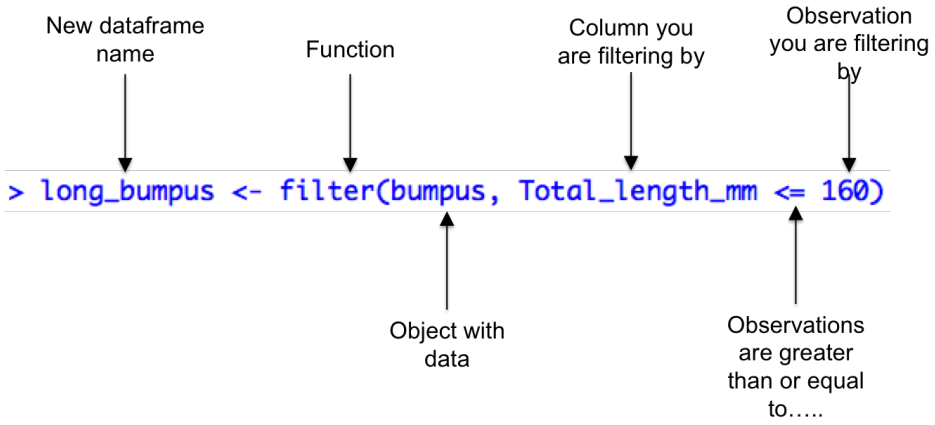
One of the most useful functions of the dplyr is `filter()`. With this function, it is possible to filter out specific observations based on their entries in one or more columns.
Let's look at the bumpus dataset in a bit more detail. We can open this dataset by using the `view()` function or clicking on "bumpus" in the "data" panel. Here we can see that the column "Sex" is separated into two groups, "m" and "f". What if we wished to create a data set that consisted of only the "male" observations? This is when we would employ the `filter` function in the following line of code:

```
male_bumpus <- filter(bumpus, sex == "m")
```



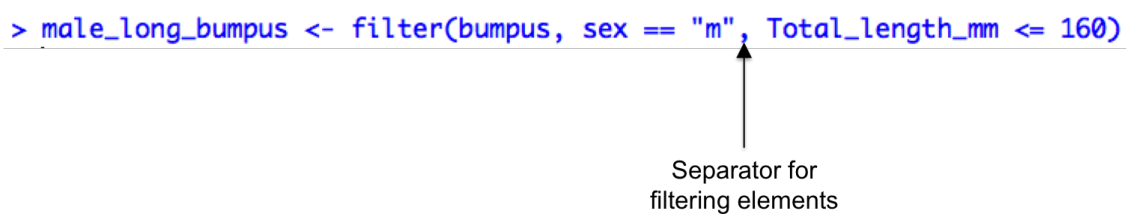
It is also possible to filter observations numerically. For example, to obtain a dataset that contains all sparrow individuals whose length is 160 mm or longer, we would use the following line of code. Note when filtering by numbers, you do not put quotes around the number.

```
long_bumpus <- filter(bumpus, Total_length_mm <= 160)
```



Finally, you can filter a dataset by as many different observations as you wish contained all within the same line of code. For example, if you wished to create a dataframe containing only males that had a total length of 160 mm or greater you would use this line of code:

```
male_long_bumpus <- filter(bumpus, sex == "m", Total_length_mm <=
    160)
```

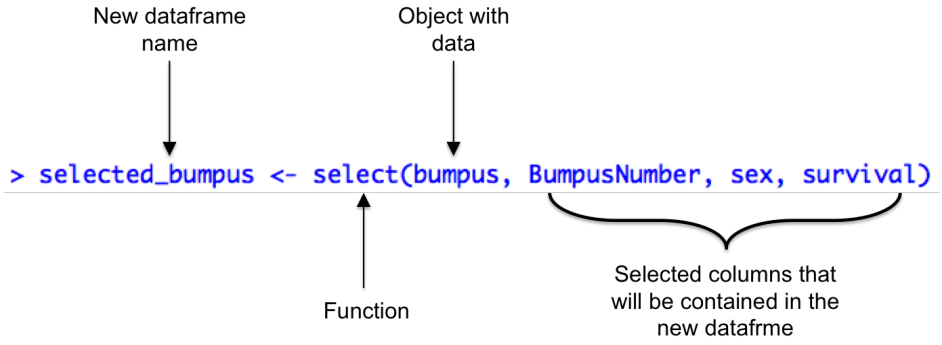


NOTE: The `filter()` function will filter by the order that the elements are imputed into the function. So for the line of code above, R will filter by sex and THEN by length.

dplyr: select()

The `select()` function is similar to the function of the "$" symbol mention earlier in that they both allow you to zoom in on a specific part of your dataset. It is particularly helpful when working with extremely large datasets. More specifically the `select()` function allows you to separate one or more columns from your dataset and transfer them into their own data frame. For example, imagine that you wished to create a dataframe containing only the columns "BumpusNumber", "sex" and "survival" from the original bumpus data set. You would complete this task with the following line of code:

```
selected_bumpus <- select(bumpus, BumpusNumber, sex, survival)
```

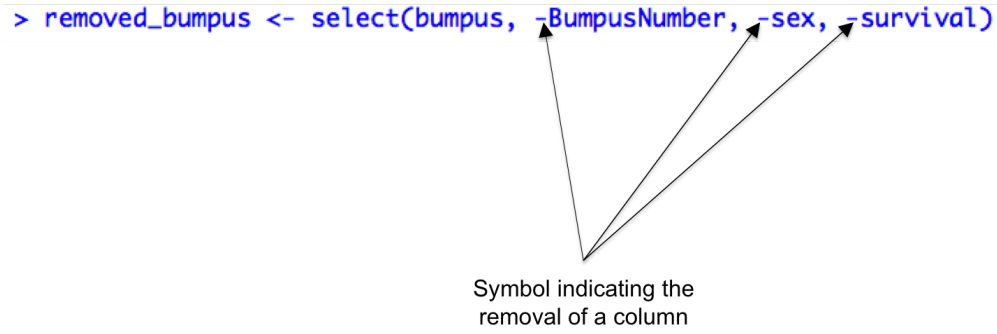


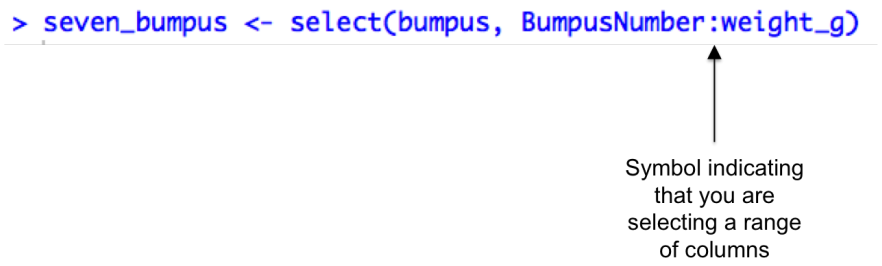
Note: The different columns that you are selected are separated by commas and the names of the columns must match what is seen in the original data set COMPLETELY (this includes capitalizations, spaces, hyphens, underscores, periods, etc).
If you were planning to create a dataframe that contained the majority of the columns seen in the original dataframe, it can be very tedious to write out all the column names by hand. Fortunately, the `select()` function also allows you to remove columns using the "-" symbol. Say we wanted to create a dataframe that contained all the columns in the original "bumpus" dataset EXCEPT "BumpusNumber", "sex" and "survival". We would use the following line of code:

```
removed_bumpus <- select(bumpus, -BumpusNumber, -sex, -survival)
```

Finally, imagine you wished to create a dataframe that contained a set of columns that were all grouped together in the original dataframe. In this case you can use the ":" symbol to save you the labours have typing out all the required column names. For example, look at the bumpus dataframe and imagine you wished to create a dataframe containing the first seven columns of the "bumpus" data set from "BumpusNumber" to "weight_g" To complete this task, you would use the following line of code:

```
seven_bumpus <- select(bumpus, BumpusNumber:weight_g)
```





dplyr: mutate()

The function `mutate()` is used to add columns to an existing dataset where the new column is usually a function of one of more of the the existing columns. For example imagine you wanted to add a column to the bumpus dataset that was equal to the log of "weight_g". This action can be down in two separate ways.

First, you can create a new dataframe that includes all the columns found in the original "bumpus" data set, plus your new addition. This is similar to the earlier examples with `filter()` and `select()`:

```
mutate.bumpus <- mutate(bumpus, log.weight = log(weight_g))
```