Intro-to-R Workshop Exercise

Dealing with Data

ER Deyle

Fall 2023

Introduction

Purpose

The primary objective of this workshop is to get every BU Marine Semester student comfortable with using R to do simple tasks that come up in many courses as you collect, analyze, and visualize data. To this end, we're going to spend a bit of the time discussing data structures. R doesn't make this nearly as complicated as some other coding environments, but ultimately all computational science has to deal with translating numbers and information we can interpret as scientists into the sequence of 1's and 0's that computers ultimate must work with. To get everyone introduced to the basics in R, we're going to go from:

- a variable (object) with a single value
- vector objects with multiple values of the same type
- list objects with multiple values of different types
- data frame objects that treat a list of vectors as columns of a table

While data observations you make (like today the length, width, and height of gastropod shells) can be stored in the simpler forms, the functions you may likely use to analyze data (say, perform a linear regression or analysis of variance) will generally return lists or data frames.

Learning Outcomes

By the end of this lesson you should:

- Know your way around the R-studio "IDE"
- Know how to use the R Console to do arithmetic
- How to store values in variables, vectors, lists, and data frames
- Use these data objects to do basic computations

The first half will involve reading and typing commands into the R console, as well as doing a few activities. In the second half you will do a short exercise to put these basic pieces together to do a bit of marine science and examining the relationship between the length and weight of gastropod shells that BU Ph.D. student Daniel Wuitchik is studying for his dissertation.

Variables and Vectors

Creating objects in R

You can get output from R simply by typing math in the console. In R studio the `Console` tab should by default be available in the lower left panel. Type a few simple arithmetic expressions at the `>` and hit enter/return.

```
3 + 5
```

```
## [1] 8
```

```
12 / 7
```

```
## [1] 1.714286
```

Great! Now you can use R to add decimal numbers instead of googling it. However, to do useful and interesting things, we need to assign *values* to *objects*. To create an object, we need to give it a name followed by the assignment operator `<-`, and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. For historical reasons, you can also use `=` for assignments, but not in every context. Because of the slight differences in syntax, it is good practice to always use `<-` for assignments.

In RStudio, typing Alt + - (push Alt at the same time as the - key) will write `<-` in a single keystroke in a PC, while typing Option + - (push Option at the same time as the - key) does the same in a Mac.

Objects can be given almost any name such as `x`, `current_temperature`, or `subject_id`. Here are some further guidelines on naming objects:

- You want your object names to be explicit and not too long.
- They cannot start with a number (`2x` is not valid, but `x2` is).
- R is case sensitive, so for example, `weight_kg` is different from `Weight_kg`.
- There are some names that cannot be used because they are the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). If in doubt, check the help to see if the name is already in use.
- It's best to avoid dots (`.`) within names. Many function names in R itself have them and dots also have a special meaning (methods) in R and other programming languages. To avoid confusion, don't include dots in names.
- It is recommended to use nouns for object names and verbs for function names.
- Be consistent in the styling of your code, such as where you put spaces, how you name objects, etc. Styles can include "lower_snake", "UPPER_SNAKE", "lowerCamelCase", "UpperCamelCase", etc. Using a consistent coding style makes your code clearer to read for your future self and your collaborators. In R, three popular style guides come from Google, Jean Fan and the tidyverse. The tidyverse style is very comprehensive and may seem overwhelming at first. You can install the **`lintr`** package to automatically check for issues in the styling of your code.

Objects vs. variables

What are known as `objects` in `R` are known as `variables` in many other programming languages. Depending on the context, `object` and `variable` can have drastically different meanings. However, in this lesson, the two words are used synonymously. For more information see: https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects

When assigning a value to an object, R does not print anything. You can force R to print the value by using parentheses or by typing the object name:

```
weight_kg <- 55    # doesn't print anything
(weight_kg <- 55)  # but putting parenthesis around the call prints the value of `weight_kg`
```

```
## [1] 55
```

```
weight_kg          # and so does typing the name of the object
```

```
## [1] 55
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

```
## [1] 121
```

We can also change an object's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

```
## [1] 126.5
```

This means that assigning a value to one object does not change the values of other objects. For example, let's store the animal's weight in pounds in a new object, `weight_lb`:

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220?

ACTIVITY:

> What are the values after each statement in the following?
>
> ```
> mass <- 47.5            # mass?
> age  <- 122             # age?
> mass <- mass * 2.0      # mass?
> age  <- age - 20        # age?
> mass_index <- mass/age  # mass_index?
> ```

Functions of variables

Functions are "canned scripts" that automate more complicated sets of commands including operations, assignments, etc. Many functions are predefined, or can be made available by importing R *packages* (more on that later). A function usually takes one or more inputs called *arguments*. Functions often (but not always) return a *value*. A typical example would be the function `sqrt()`. The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called *calling* the function. An example of a function call is:

```
weight_kg <- sqrt(10)
```

Here, the value of 10 is given to the `sqrt()` function, the `sqrt()` function calculates the square root, and returns the value which is then assigned to the object `weight_kg`. This function takes one argument, other functions might take several.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a dataset. We'll see that when we read data files into R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below). Some functions take arguments which may either be specified by the user, or, if left out, take on a *default* value: these are called *options*. Options are typically used to alter the way the function operates, such as whether it ignores 'bad values', or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let's try a function that can take multiple arguments: `round()`.

```
round(3.14159)
```

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

Here, we've called `round()` with just one argument, `3.14159`, and it has returned the value `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the `round` function. We can use `args(round)` to find what arguments it takes, or look at the help for this function using `?round`.

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

```
?round
```

We see that if we want a different number of digits, we can type `digits = 2` or however many we want.

```
round(3.14159, digits = 2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

And if you do name the arguments, you can switch their order:

```
round(digits = 2, x = 3.14159)
```

```
## [1] 3.14
```

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to then specify the names of all optional arguments. If you don't, someone reading your code might have to look up the definition of a function with unfamiliar arguments to understand what you're doing.

Vectors and data types

A vector is the most common and basic data type in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of animal weights and assign it to a new object `weight_g`:

```
weight_g <- c(50, 60, 65, 82)
weight_g
```

```
## [1] 50 60 65 82
```

A vector can also contain characters, say if we have categorical information to store:

```
stages <- c("juvenile", "juvenile", "juvenile","adult")
stages
```

```
## [1] "juvenile" "juvenile" "juvenile" "adult"
```

The quotes around "juvenile", "adult", etc. are essential here. Without the quotes R will assume objects have been created called `juvenile` and `adult`. As these objects don't exist in R's memory, there will be an error message. [Try it!].

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(weight_g)
```

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

```
length(stages)
```

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

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates what kind of object you are working with:

```
class(weight_g)
```

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

```
class(stages)
```

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

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(weight_g)
```

```
##  num [1:4] 50 60 65 82
```

```
str(stages)
```

```
##  chr [1:4] "juvenile" "juvenile" "juvenile" "adult"
```

You can use the `c()` function to add other elements to your vector:

```
weight_g <- c(weight_g, 90) # add to the end of the vector
weight_g <- c(30, weight_g) # add to the beginning of the vector
weight_g
```

```
## [1] 30 50 60 65 82 90
```

In the first line, we take the original vector `weight_g`, add the value `90` to the end of it, and save the result back into `weight_g`. Then we add the value `30` to the beginning, again saving the result back into `weight_g`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

An **atomic vector** is the simplest R **data type** and is a linear vector of a single type. Above, we saw 2 of the 6 main **atomic vector** types that R uses: `"character"` and `"numeric"` (or `"double"`). These are the basic building blocks that all R objects are built from. The other 4 **atomic vector** types are:

- `"logical"` for `TRUE` and `FALSE` (the boolean data type)
- `"integer"` for integer numbers (e.g., `2L`, the `L` indicates to R that it's an integer)
- `"complex"` to represent complex numbers with real and imaginary parts (e.g., `1 + 4i`) and that's all we're going to say about them
- `"raw"` for bitstreams that we won't discuss further

You can check the type of your vector using the `typeof()` function and inputting your vector as the argument.

Vectors are one of the many **data structures** that R uses. Other important ones are lists (`list`), matrices (`matrix`), data frames (`data.frame`), factors (`factor`) and arrays (`array`).

ACTIVITY:

Challenge

- We've seen that atomic vectors can be of type character, numeric (or double), integer, and logical. But what happens if we try to mix these types in a single vector?
- What will happen in each of these examples? (hint: use `class()` to check the data type of your objects):

  ```
  num_char <- c(1, 2, 3, "a")
  num_logical <- c(1, 2, 3, TRUE)
  char_logical <- c("a", "b", "c", TRUE)
  tricky <- c(1, 2, 3, "4")
  ```

- Why do you think it happens?
- How many values in `combined_logical` are `"TRUE"` (as a character) in the following example (reusing the 2 `..._logical`s from above):

  ```
  combined_logical <- c(num_logical, char_logical)
  ```

- You've probably noticed that objects of different types get converted into a single, shared type within a vector. In R, we call converting objects from one class into another class *coercion*. These conversions happen according to a hierarchy, whereby some types get preferentially coerced into other types. Can you draw a diagram that represents the hierarchy of how these data types are coerced?

Subsetting vectors

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
fish <- c("hake", "herring", "sprat", "cod")
fish[2]
```

```
## [1] "herring"
```

```
fish[c(3, 2)]
```

```
## [1] "sprat"   "herring"
```

We can also repeat the indices to create an object with more elements than the original one:

```
more_fish <- fish[c(1, 2, 3, 2, 1, 4)]
more_fish
```

```
## [1] "hake"    "herring" "sprat"   "herring" "hake"    "cod"
```

R indices start at 1. Programming languages like Fortran, MATLAB, Julia, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

List objects

Lists are a more general and flexible data object that are otherwise very similar to vectors. Why are they more flexible? Well, above we showed how a vector couldn't contain values of different classes. In a list that restriction is gone. Let's have a look.

First, you can create a list object much the same way as a vector, but instead of wrapping the list of values within `c()` instead you use `list()`.

```
num_logical <- list(1, 2, 3, TRUE)
```

This new list has the same length it would as a vector.

```
length(num_logical)
```

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

But now when we ask about the class, what do we get?

```
class(num_logical)
```

```
## [1] "list"
```

Interesting.

`[ ]` selects a list of the elements while `[[ ]]` selects the single element of the list.

Finally, interacting with list data can be especially easy when you give names.

```
specimen_1 <- list(common_name="bay scallop",width_in=4.25,height_in=0.5,length_in=2.625,sex="female")
```

We can refer to list element by name using the `$` between the list object name and the name of the element we want inside.

```
specimen_1[[3]]
```

```
## [1] 0.5
```

```
specimen_1$height_in
```

```
## [1] 0.5
```

Have we done anything yet that we couldn't do with vectors? Not really. Here's the kicker, though, not only can the list elements be a mix of numbers, logicals, and characters, but the list elements can also be vectors... or even more lists!

```
specimen_data <- list(weight_g = c(50, 60, 65, 82),
                      stages=c("juvenile","juvenile","juvenile","adult"))
```

Why do you need to know how to use lists? A primary concern for this workshop is that if you run most statistical tests in R, be it an ANOVA or a linear regression, those functions tend to return list objects since the analysis results contain multiple kinds of information. For example, if we just grab one of the included R datasets, `trees` (type `?trees` in console if you want a bit of context), we can run a simple correlation test with `cor.test()` (type `?cor.test` while you're at it!).

```
cor_out <- cor.test(trees$Volume, trees$Girth)
```

We save the output of the `cor.test` function call. If you look in the **Environment** tab in the upper right of R-studio you'll see `List of 9`. We can check for ourselves:

```
length(cor_out)
```

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

```
names(cor_out)
```

```
## [1] "statistic"   "parameter"   "p.value"     "estimate"    "null.value"
## [6] "alternative" "method"      "data.name"   "conf.int"
```

Lots of different information here, including the linear correlation statistic, significance levels, and p.value. Some of these elements are just single numeric values.

```
cor_out$statistic
```

```
##        t
## 20.47829
```

While the `conf.int` element of the list happens to be a numeric named vector:

```
str(cor_out$conf.int)
```

```
##  num [1:2] 0.932 0.984
##  - attr(*, "conf.level")= num 0.95
```

Data frame objects

Now we come to data frames or `data.frames`. Data frames are really just special kinds of lists that make dealing with multivariate data much easier and breazier by assuming some constraints. Basically, a data frame is a list of vectors that all have the same length and so can be treated as the columns of a table. Let's back up to that R object `trees` we just used to do a quick linear regression. You can inspect data.frame objects a number of ways, including just asking the console to print it.

```
trees
```

```
##    Girth Height Volume
## 1    8.3     70   10.3
## 2    8.6     65   10.3
## 3    8.8     63   10.2
## 4   10.5     72   16.4
## 5   10.7     81   18.8
## 6   10.8     83   19.7
## 7   11.0     66   15.6
## 8   11.0     75   18.2
## 9   11.1     80   22.6
## 10  11.2     75   19.9
## 11  11.3     79   24.2
## 12  11.4     76   21.0
## 13  11.4     76   21.4
## 14  11.7     69   21.3
## 15  12.0     75   19.1
## 16  12.9     74   22.2
## 17  12.9     85   33.8
## 18  13.3     86   27.4
## 19  13.7     71   25.7
## 20  13.8     64   24.9
## 21  14.0     78   34.5
## 22  14.2     80   31.7
## 23  14.5     74   36.3
## 24  16.0     72   38.3
## 25  16.3     77   42.6
## 26  17.3     81   55.4
## 27  17.5     82   55.7
## 28  17.9     80   58.3
## 29  18.0     80   51.5
## 30  18.0     80   51.0
## 31  20.6     87   77.0
```

We live in the era of big data, though! Even 31 rows can fill up your screen quickly. Often, just a glance is enough to check out a data.frame. Enter the `head()` command which lets you peak at the first few rows.

```
head(trees)
```

```
##   Girth Height Volume
## 1   8.3     70   10.3
## 2   8.6     65   10.3
## 3   8.8     63   10.2
## 4  10.5     72   16.4
## 5  10.7     81   18.8
## 6  10.8     83   19.7
```

Aside: can you guess what `tail()` does?

Either way, it is easy to see that the data.frame `trees` has 4 columns, named Diameter, Height, and Volume. Alternatively, or additionally you can use many of the same interfacing modes for lists that we just discussed, beginning with `str()`.

```
str(trees)
```

```
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

Then if you want to extract the girth of the 5th tree for example, you do it just as we did above for a list:

```
trees$Girth[5]
```

```
## [1] 10.7
```

The data frame can also act like an array, which we've skipped over a bit, but is the generalization of a vector that has rows and columns of data. Since Girth is the first variable, we can pull the same value by requesting the 5th row and the 1st column:

```
trees[5,1]
```

```
## [1] 10.7
```

Finally, it can be useful to add on new columns of data as you do calculations. For example, if you read the help entry on the `trees` data set, you might have seen the note that the variable `Girth` is an erroneous labeling of diameter data. Girth really is more suggestive of circumference than diameter. But we could do that conversion! Assuming the trunk has a roughly circular cross-section, then `circumference = pi * diameter`. And guess what, R knows the value of pi! (Well a lot of the digits at least).

First, let us rename `Girth`. We can do this via the `names()` function.

```
names(trees)
```

```
## [1] "Girth"  "Height" "Volume"
```

```
names(trees)[1] <- "Diameter"
names(trees)
```

```
## [1] "Diameter" "Height"   "Volume"
```

Now what happens if we ask for the "Girth"?

```
trees$Girth
```

```
## NULL
```

It's empty! But, R is more than happy to let us assign something there.

```
trees$Girth <- pi * trees$Diameter
trees$Girth
```

```
##  [1] 26.07522 27.01770 27.64602 32.98672 33.61504 33.92920 34.55752 34.55752
##  [9] 34.87168 35.18584 35.50000 35.81416 35.81416 36.75663 37.69911 40.52655
## [17] 40.52655 41.78318 43.03982 43.35398 43.98230 44.61062 45.55309 50.26548
## [25] 51.20796 54.34955 54.97787 56.23451 56.54867 56.54867 64.71681
```

Great! Past this, there's lots of tools you can pick up along the way to work with data.frame objects, but now you have the basics, including renaming variables and construct additional columns. Why do you need to know how to use data.frames? As we've already seen, they are a way to supply data to functions that perform statistical tests. They are also the way data will often be loaded into R, e.g. if you are reading in table from an excel or google spreadsheet. And… that is how we will proceed today!

Putting it all together

At this point you should have the basic idea for how to create variables in R and perform basic arithmetic. Now let's put it all together to do some marine science in the form of a paired-down lab exercise. You will collect data on the length, width, and height of slipper shells, use a few mathematical formula (dare we call it a model!) to make predictions about the weight of those shells, then weigh the shells to validate the model predictions.

Do some housekeeping

Very quickly when using R or other coding languages to do data science you'll find your work is involving several different pieces. Before you get any further, it would be good to make a new folder somewhere on your computer for this workshop. Our recommendation is to either do this on your Desktop or in your Documents. You can name it something like "BUMP-R-workshop" (or whatever else makes sense to you).

Open a new R script

Just like in other labs, it's good to do your work in some sort of notebook. Here, we will keep it simple and make a blank R-script in Rstudio. While we can do all the calculations entering line-by-line into the console, writing the whole sequence of steps down in order minimizes error and maximizes the ease of reproducing. In some Marine Semester classes, you may also be introduced to a slightly fancier notebook format with Rmarkdown.

There are a few ways to get a new script open in RStudio. There's a small green "+" sign button in the upper left you can use or you can navigate to the file menu and select `New File -> R Script`. Your new, blank file should appear in the upper left pane of RStudio. Go ahead and save it in the "BUMP-R-workshop" folder you just made and use a name like "R-intro-workshop_exercise.R".

Collect and record data

Courtesy of Dan Wuitchik, there are several sets of slipper shells available for you to measure. The samples are organized by collection site (Cape May, NJ to the tip of Maine) and need to be kept that way. Work with other students to measure the height, width, and length in mm. Make sure you all agree on which dimensions you'll be calling "height", "width", and "length"!

As you measure, you need to record the measurements. As always will be the case, there's more than one option. Since everyone should be able to quickly access Google Sheets, let's do it that way. Many of you will have collected measurements in a lab into a data sheet; ask for help if this part isn't obvious. Set the document up like so:

Length	Width	Height
#	#	#
#	#	#

TASK:

> Record physical measurements of slipper shells in a Google Sheet.

When you are done, we need to export the data so that it's easy to turn around and load into R.

TASK:

> Download the data in your Google Sheet as a "csv" file. Once you've downloaded, drag-and-drop (move) the file from your downloads to the "BUMP-R-workshop" folder using the normal file system on your computer.

Import the data into R

There are many functions to help you read data saved in a file into an object in R from various formats (including .xls and .xlsx). We're going with the classic, `read.table()`. Have at least a quick look at the help function by typing:

```
?read.table
```

This is a very flexible function, so it may be overwhelming. It has 25 formal arguments! Luckily, there are sensible default values available for all but one, `file`. So you'll have to give it the file name. The other important arguments here are `header` and `sep`. The first of those specifies if the data coming in has headers (it should) and what character is uses to separate the values in a row. We've made a "csv", so the separator is `","`.

```
data_shell_sizes <- read.table(file="Slipper Shell Data - Sheet1.csv",header=TRUE,sep=",")
```

THIS PROBABLY DIDN'T WORK. R likely just told you "No such file or directory". That's because R doesn't know where to look yet. Remember how you made a new folder for this workshop? That's going to pay off now! We can tell R that we're working in a particular file location on the computer; R calls this a "working directory". If you navigate at the top of the window to `Session -> Set Working Directory -> Choose Directory ...`, R will open a window and let you click through your file system to show R where to go.

TASK:

> Set the working directory of your R `Session` to the folder on your computer where you've put the data file. Use `read.table()` to import your recorded data.

You can check if this worked using the `head()` command.

```
head(data_shell_sizes)
```

```
##   Length Width Height
## 1   17.5  11.2    3.7
## 2   16.2  10.0    2.5
## 3   25.2  21.5    8.8
```

Do additional computations from these data

Alas! We have measured in millimeters but I'm going to ask you next to work in centimeters. The unit conversion for mm to cm is 10 mm = 1 cm.

TASK:

> Convert the measurements from milimeters into centimeters, storing these as additional columns in the data.frame.

Now that we have centimeters, we can calculate volume in cubic centimeters. To do this, you'll need to make some approximations about the shape of the "specimens". Here are a few formula and pick the one(s) you think are most appropriate. Real shells are of course neither of these idealized shapes, but let's see how close we can get. After all, *all models are wrong, some are useful.*

Shape	Formula
Elliptical Disc	pi × L × W × H
Half-Ellipsoid	2/3 × pi × L/2 × W/2 × H/2

Recall: R knows the approximate value of pi:

```
pi
```

```
## [1] 3.141593
```

While you do it, go ahead and make use of R data structures and descriptive variable names! If you come back a week later and just see

```
x1 <- 4.3
x2 <- 1.2
x3 <- 2.9
y <- x1*x2*x3
```

You stand a good chance at not actually remembering what that code was even about.

TASK:

> Use arithmetic to estimate the volume of each specimen you measured in cubic centimeters. Store the estimated volume as a new column in the data frame with your imported observations of length, width, and height.

Aragonite has a density of 2.93 g/cc. However, mollusk shells aren't composed of pure aragonite and are also porous, so they are less dense. Let's assume the shells have a density of 2.7 g/cc. Use this to estimate the mass of each specimen.

TASK:

> Use this approximate density and the estimated volume of each specimen to estimate their mass. Again, store this estimated mass as a new column in the same data frame.

TASK:

> Compare your results to direct measurements of mass for each specimen. Collect this new data in the same spreadsheet, adding another column, then display the data in a graph (see below). How close are the estimates? Does there appear to be a systematic bias?

HINT 1: You can create a basic x-y scatter plot in R using the command `plot(x,y)` where `x` and `y` are vectors of the same length. If you want to dress it up a bit you can add a "1-1" line afterwards using `abline(a=0,b=1)`. Type `?abline` to see the syntax. You can pass additional arguments to fancy it up, like a dotted line type with `lty=2` and a red color with `color='red'`.

If you have time

Option 1: plotting with a more advanced package

Alternatively, you can create a basic x-y scatter plot using the vaunted `ggplot2` package. `ggplot2` uses a grammer of graphics (hence `gg`) to specify how a plot is built of sequential commands and layers "added" together.

First, you need to make sure ggplot2 is installed.

```
install.packages("ggplot2")
```

Once you have installed a package, you also need to tell R you want to use it in the current session.

```
library("ggplot2")
```

Now, for a basic `ggplot2` command you'll need to first specify which variables you're working with through a `mapping` arguement, then the style of plot you want made from them. Try `ggplot(mydata,mapping=aes(x,y)) + geom_point()` where `x` and `y` are now the names of the data columns in `mydata`. If you want to dress it up a bit you can add a "1-1" line afterwards using `+ geom_abline(aes(intercept=0,slope=1),lty=2,color="red")`.

Option 2: dig deeper into geometry and scaling

In fisheries, it is often much easier to measure length of individuals than many other related variables of interest like age or weight. Allometric scaling relationships are a way to parameterize relationships between variables to estimate unmeasured variables from length (or in other cases, mass). Let's look at the relationship between length and mass for the shells.

TASK:

> Create graph that shows observed mass as a function of length for the specimens you measured, possibly including measurement data from other students if you like. Does mass look to be predictable from just length? Do you expect this relationship to be linear? Does it appear linear?

Reflection Questions

- Was there a systematic bias to your estimates of mass? List the different assumptions made along the way to get that estimate. Which of these might have been sources of bias? How might you change these assumptions to get better estimates? [There is possibly a lesson here that correctly written code doesn't necessarily produce correct answers!].
- What questions do you still have about the basics of variables and vectors in R?

Appendix: Resources

Portions of this material were adapted from:

> François Michonneau, Tracy Teal, Auriel Fournier, Brian Seok, Adam Obeng, Aleksandra Natalia Pawlik, … Ye Li. (2019, July 1). datacarpentry/R-ecology-lesson: Data Carpentry: Data Analysis and Visualization in R for Ecologists, June 2019 (Version v2019.06.1). Zenodo. http://doi.org/10.5281/zenodo.3264888

More on Crepidula fornicata:

Henry, J. Q., & Lyons, D. C. (2016). Molluscan models: Crepidula fornicata. Current Opinion in Genetics & Development, 39, 138-148. https://doi.org/10.1016/j.gde.2016.05.021

Global change dimensions of mollusk shell properties:

Gizzi, F., Caccia, M., Simoncini, G. et al. Shell properties of commercial clam Chamelea gallina are influenced by temperature and solar radiation along a wide latitudinal gradient. Sci Rep 6, 36420 (2016). https://doi.org/10.1038/srep36420