Lecture 4. Programming for data processing and simulation

One of the reasons R is a powerful platform for statistics is because it combines a large number of built-in and add-on statistical functions, and at the same time it is an extensive programming language. Taking advantage of the programming part will make it easier for you to manipulate and process your data, in a reproducible way. It will also let you be comfortable simulating what data should look like under different assumptions. When I introduced the central limit theorem, I used simulation to show how the process works (and when it doesn't work). This is one example of how simulation can be used to better understand statistical concepts and methods. Simulation is also useful for designing and interpreting research, and for doing inference on a statistical model (e.g. bootstrapping).

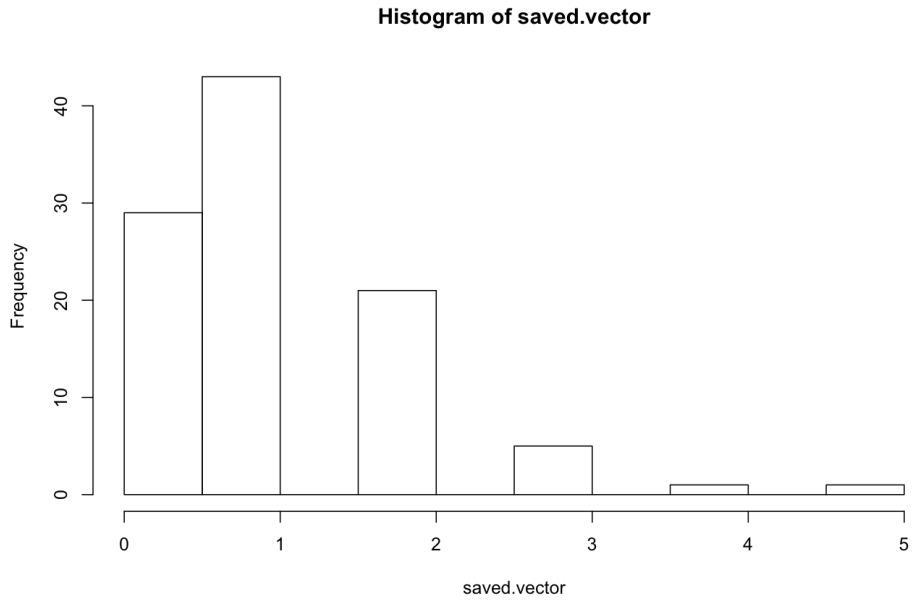
Here we'll go through some basics of programming, how it can help with data processing, and how to simulate a statistical model. I will focus on 'base R', because it is the foundation of R and shares features with other popular programming languages. If you plan to do considerable data wrangling you may want to look into the 'tidyverse' (https://www.tidyverse.org).

Basic programming in R

There are a few bits of the R programming language that will get you very far. Some of these are common to most programming languages, and some features are specific to R.

Loops with *for*: loops let you iterate actions. Maybe you have data from a number of sites, and you want to do some plotting or analysis for each site, and show/save the results. Here is a fairly impractical example to show how the syntax works:

```
saved.vector = vector(length = 100)

for (i in 1:100) {
  saved.vector[i] = rpois(1, lambda = 1)
  }

hist(saved.vector)
```

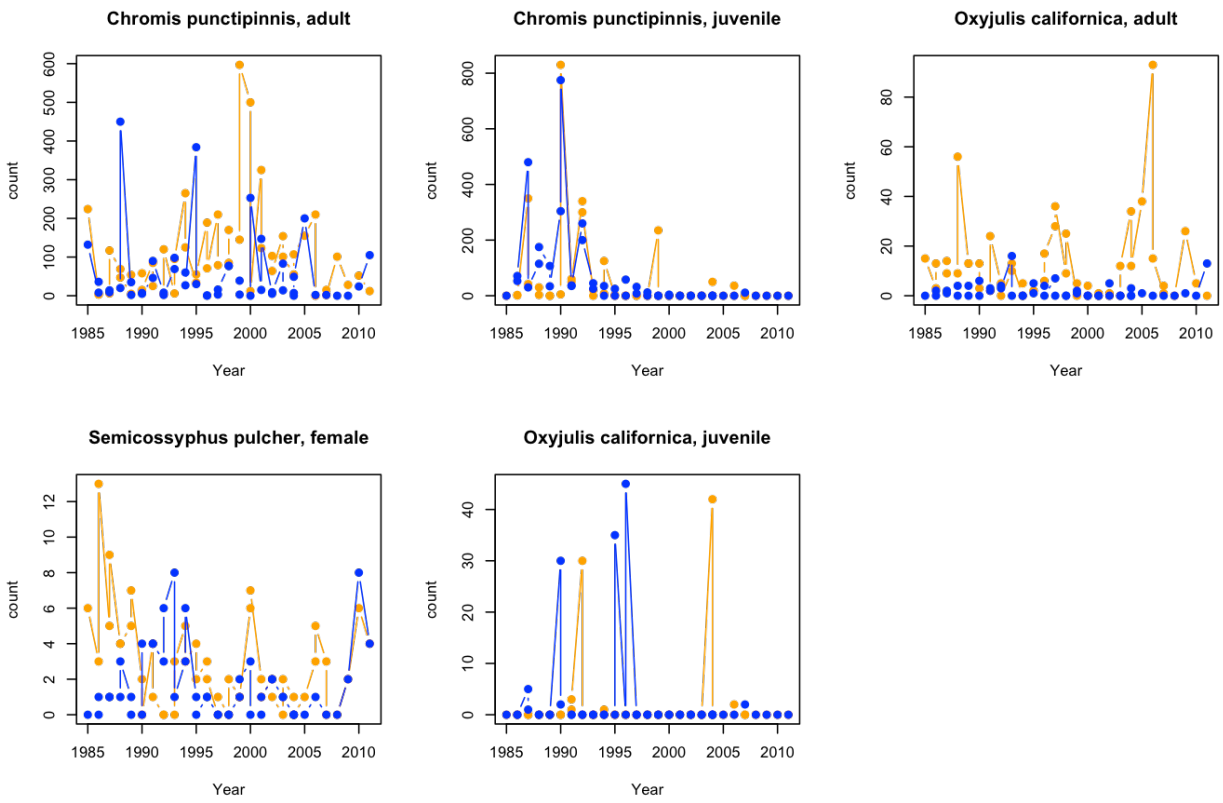


This example takes a sample from a Poisson distribution, looping through 100 times. The syntax defines a dummy variable *i*, and uses that to index the iterations. Before I started the for loop, I defined an empty vector *saved.results* that has a length of 100. In each iteration, a random draw from a Poisson distribution with lambda = 1 is taken and saved at position *i* in saved.results. At the end I plotted a histogram of the random draws.

Here's a more realistic example using benthic fish that are sampled every year at a number of sites in the Channel Islands in California. There are a number of processing steps needed to get this data ready, and I've appended the full code to the end of these notes. For now, let's just look at the loop for plotting:

```
head(fish1)
##     Site Year    Date Species count                     namefix
## 460   11 1985 8/30/85   14001   224 Chromis punctipinnis, adult
## 461   11 1986 8/27/86   14001     4 Chromis punctipinnis, adult
## 462   11 1986 9/12/86   14001     3 Chromis punctipinnis, adult
## 463   11 1987 10/1/87   14001     6 Chromis punctipinnis, adult
## 464   11 1987 9/15/87   14001   117 Chromis punctipinnis, adult
## 465   11 1988 10/3/88   14001    69 Chromis punctipinnis, adult

species.to.use
## [1] "Chromis punctipinnis, adult"    "Chromis punctipinnis,
juvenile"
```

```
## [3] "Oxyjulis californica, adult"    "Semicossyphus pulcher, female"
## [5] "Oxyjulis californica, juvenile"

#Loop through 5 iterations, to plot the 5 most abundant species at two
sites
par(mfrow = c(2,3))
for (i in 1:5) {

  #subset for species i from fish1, the dataframe for site 2
  species.sub1 = subset(fish1, namefix == species.to.use[i])
  #subset for species i from fish2, the dataframe for site 2
  species.sub2 = subset(fish2, namefix == species.to.use[i])

  #to make the right ylim on the plot, see what the max count is at the
two sites
  ymax = max(c(species.sub1$count, species.sub2$count))

  #plot it
  with(species.sub1, plot(count ~ Year, type = 'b', pch = 19, ylim =
c(0,ymax), col = 'orange', main = species.to.use[i]))
  with(species.sub2, points(count ~ Year, type = 'b', pch = 19, col =
'blue'))

}
```



Here I used two dataframes, fish1 and fish2, that included yearly samples of the benthic fish community at two site in the Channel Islands (a different dataframe for each site). I wanted to look at the abundance of 5 common species over time, so I defined those species in advance and then iterated some plotting code 5 times. In each iteration, I took the subsets of fish1 and fish2 where `namefix == species.to.use[i]`, and then plotted the time series for that species at both sites.

Vectorization. Loops are often necessary, but R has many functions that let you avoid loops by performing operations on a whole vector at a time. The R lingo for working on with whole vectors at once is to *vectorize* (yes, this is kind of an awful word). For example, the first example I gave with taking random draws from a Poisson distribution can be highly simplified by using the rpois() function to just return 100 values at once: rpois(100, lambda = 1). The various population packages developed by Hadly Wickham (reshape, plyr, ggplot) seem to be predicated on the fact that loops should be avoided at all costs.

All of the mathematical operators in R are vectorized, which is very convenient for dealing with output from statistical analyses and simulations. That means if you have a vector *test* that is length 100, and a vector *test2* that is length 100, when you add *test* + *test2* then R will create a new vector length 100 that is equal to c(test[1]+test2[1], test[2]+test2[2], …). In addition, if you add a single number to a vector, that number will be added to each element of the vector. So *test*+5 will add 5 to each element of *test*.

These features make it easy to do math on vectors of data. E.g. maybe you want to use a vector of temperatures as a predictor, but instead of using the raw data you want to standardize temperature by it's mean and standard deviation. You can just write

```
temperature.std = (temperature - mean(temperature))/sd(temperature)
```

Now you have a variable where a value of 1 means the temperature is 1 standard deviation greater than the mean.

Apply

A common occurrence is that you want to take some object (a vector, a data frame), break it into pieces, and apply some function to each piece. For example, in homework 1 the first question asked you to calculate the mean and standard error of a variable (isopod Length), broken down by Sex and by Region. This is the kind of thing the function tapply() is designed for. The first argument is a vector you want to process (Length), the second argument is a factor that breaks Length into groups (e.g. Sex), and the third argument is a function that you want to apply to these groups (e.g. mean()). So to calculate the means by Sex, you can say

```
length.by.sex = tapply(Length, Sex, mean)
```

R includes a whole family of apply functions: apply(), tapply(), lapply(), etc. You can look up how each one works, I primarily use tapply() and apply(). The apply() function operates on arrays, as well as data frames. Maybe I have a data frame called "traits" that where each column is a different trait (e.g. length, width, mass) measured on a bunch of individuals, where each row is a different individual. You could calculate the mean for each column like this:

```
mean.by.column = apply(traits, 2, mean)
```

Here the second argument, '2', is telling the function calculate the mean by column; if I wanted the mean by row, I would put '1' instead.

The apply functions are useful, though they can be a bit cumbersome if you want to do some complex processing of a data frame. You can check out the "plyr" package for some functions that make complex apply operations a little easier.

Defining a function

It is easy to define your own function in R, which is something you might do if you want to do a calculation many times (e.g. in a loop). The syntax looks like this:

```
function.name = function(input.variable) {
    my.calculations
    return(my.calculations)
    }
```

For example, maybe you want to calculate the standard error of the mean for different groups of data. For some reason R does not have a built-in standard error function (who knows why), though various packages do. Nonetheless, you can define your own function like this:

```
standard.error = function(x) {
    se = sd(x)/sqrt(length(x))
    return(se)
    }
```

Anything you want the function to do goes in the curly brackets. The return() statement specifies what you want the function to return as output. For complicated functions, it is helpful to use this whole syntax. However, there are some shortcuts that you will see commonly used. You don't really need to say return(), because by default your function will return whatever is calculated by the last statement in the function. So you could just write this:

```
standard.error = function(x) {
     sd(x)/sqrt(length(x))
     }
```

Furthermore, because this function only has one statement, you can skip the curly brackets as well, and just but the one statement directly after function() :

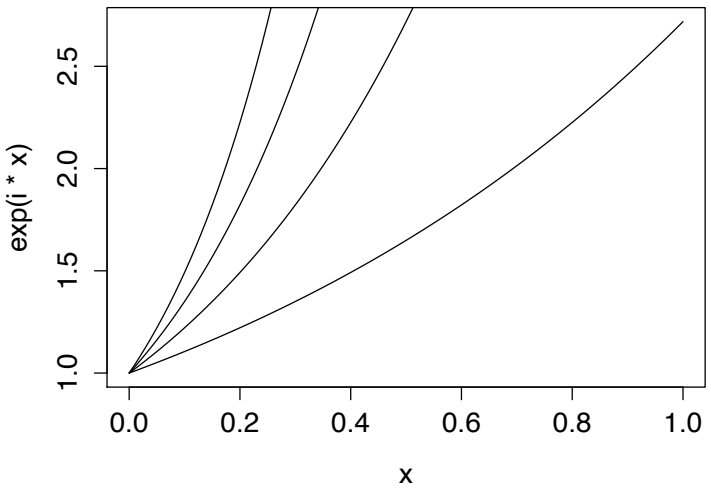
```
standard.error = function(x) sd(x)/sqrt(length(x)
```

Now you can calculate the standard error by Sex with this statement:

```
se.by.sex = tapply(Length, Sex, standard.error)
```

Conditional statements. Often when you are iterating some calculation with a loop, you want the outcome to depend on some characteristic of the data. For example, if you want to plot a bunch of curves on one plot, you may want add=FALSE for the first curve, and add=TRUE for the subsequent curves, so you can draw them all together. You could do this:

```
n = 4
for (i in 1:n) {
     if (i == 1) curve(exp(i*x), add = FALSE)
     if (i > 1) curve(exp(i*x), add = TRUE)
     }
```



So the way *if* works is that you put a logical statement in parentheses. If the logical statement is true, then the part after the parentheses will be evaluated (here, the

curve() function). If the logical statement is false, then that part will not be evaluated. This is called a conditional statement. You can put a whole series of commands after an *if* statement, in which case you use curly brackets. So the above example could be written like this:

```
n = 4
for (i in 1:n) {
     if (i == 1) {
          curve(exp(i*x), add = FALSE)
          }
     if (i > 1) {
          curve(exp(i*x), add = TRUE)
          }
     }
```

It sometimes happen that you have a vector of count data, but when the researcher did not observe any organisms they didn't put 0, instead they just left the entry blank, which will get turned into NA when you load the data into R. You could use a loop and an *if* statement to turn those NA's into zeros:

```
for (i in 1:length(the.data)) {
     if (is.na(the.data[i])) the.data[i] = 0
     }
```

The function is.na() returns TRUE or FALSE, depending on whether the value is NA or not. However this is an example where it would be convenient to use logical indexing instead.

Logical indexing. A convenient way to manipulate data in R is with logical indexing. Hopefully you are familiar with the standard way to index a vector:

```
the.data[1:5]
##  [1] 2 1 NA 4 NA
```

This pulls out the first 5 values of the.data.

You can used a logical statement to see which elements of the.data are NA:

```
data.short = the.data[1:5]
is.na(data.short)
##  [1] FALSE  FALSE  TRUE  FALSE TRUE
```

This statement returns a vector of true/false values. This is a *logical* vector, and you can use it to index another vector:

```
data.short[is.na(data.short)]
```

```
## [1] NA NA

data.short[!is.na(data.short)]
## [1] 2 1 4
```

These statements pull out the NA's, and the values that are not NA. Finally, you could set the NA's to 0 in one step:

```
data.short[is.na(data.short)] = o
data.short
##  [1] 2 1 0 4 0
```

R objects and classes

Understanding a bit about how R stores data and other things is useful, especially when something goes wrong and you have to interpret a mysterious error message.

One thing to know is that everything in R is an *object*, and that includes everything from vectors to data frames to functions to fitted models. Objects are treated differently based on their *class*. This lets functions in R have a kind of rudimentary intelligence, because when you say summary(x) what goes on behind the scenes is that R first says "what class is x?", and then it selects a summary() function that is appropriate for that class. So for example, if you call summary() for a linear model, and call summary() for a data frame, the output is actually generated by two entirely different functions that are both defined as summary() functions.

Vector types

For data storage, the fundamental units are vectors, and vectors come in different atomic types:

numeric – c(1.1, 2.5, 0.4)

integer – c(1, 3, 5)

character – c("oh", "hai")

logical – c(TRUE, FALSE, FALSE)

The difference between *numeric* and *integer* is that numeric numbers can be decimals but integers can only be integers. The distinction will rarely come up in daily use, but you should be aware that they both exist. A *character* vector is made

up of a series of strings, which is programming lingo for bits of text. They will always be in quotes, and you need to put text in quotes to make it a string. A *logical* vector is simply TRUE's and FALSE's. If you're not sure what R thinks about your vector, you can use class():

```
my.vector = c("what", "fun")
class(my.vector)

## [1] "character"
```

Matrices and arrays

More complex data structures in R are created by modifying or combining these atomic types. An *array* is a multidimensional vector. You can make one with as many dimensions as you want, e.g. a 3 by 3 by 3 cube of data, but most often you'll use 2-dimensional arrays, which are also called *matrices*. You create a matrix from a vector like this:

```
a.vector = 1:10
a.matrix = matrix(a.vector, nrow = 5)
a.matrix

##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

I created a.matrix from a.vector, which has a length of 10. Because I told the matrix() function to use 5 rows, it created a 5x2 matrix. It filled up the matrix by columns, which is the default setting. If you wanted to process some data and store it in a matrix (e.g., maybe store some means in column 1 and some corresponding standard errors in column 2), you could first create an empty matrix:

```
empty.matrix = matrix(nrow = 5, ncol = 2)
empty.matrix

##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
## [3,]   NA   NA
## [4,]   NA   NA
## [5,]   NA   NA
```

Then you could fill in the entries of your matrix using a loop to calculate the means and SE's, or using tapply.

A matrix (or an array) is really just a special kind of vector, one where you've said

"treat this vector as something with X rows and Y columns". So you can have a matrix of any vector type (numeric, character, etc.).

Lists

A matrix lets you give a vector some multidimensional structure, but a matrix can only be one atomic type. Objects that store multiple data types together are called *lists*. Lists have a very flexible structure, in the sense that you can put any stuff together into a list. E.g. you could combine some numeric, character, and logical vectors of different length together:

```
some.numbers = 1:5
some.letters = letters[1:10]
some.logicals = c(TRUE, TRUE, FALSE)
my.list = list(some.numbers, some.letters, some.logicals)
my.list
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
##
## [[3]]
## [1]  TRUE  TRUE FALSE
```

This list has three entries, each of which is a vector I defined. It's probably rare that you'll want to make a list like this, but data frames are a special kind of list.

Data frames

Data frames are a kind of list where each entry is some vector, and all the vectors are the same length. That means the data frame has the shape of a matrix, but can have different data types in the different columns. Let's take a look at the fish data frame I was working with in the plotting example above:

```
head(fish)
```

```
##      Site Year    Date Species count                        namefix
## 497    11 2004 10/1/04   14001    55    Chromis punctipinnis, adult
## 1373   11 2004 10/1/04   14002    50 Chromis punctipinnis, juvenile
## 2249   11 2004 10/1/04   14003    34    Oxyjulis californica, adult
## 3125   11 2004 10/1/04   14004    42 Oxyjulis californica, juvenile
## 4001   11 2004 10/1/04   14005     0       Sebastes mystinus, adult
## 4877   11 2004 10/1/04   14006     0    Sebastes mystinus, juvenile
```

```
sapply(fish, class)
```

```
##      Site      Year      Date   Species     count   namefix 
## "integer" "integer"  "factor" "numeric" "integer"  "factor" 
```

The function head() shows the first few rows of the data frame. Looks like we have counts that are coded by site, year, date, speciesID, and then a column namefix where I previously translated the IDs into informative names. I was curious what type of data was in each column, so I used sapply() to perform the function class() on each column. sapply() is appropriate here because it is an apply function for lists, and dataframes are a kind of list where each column is an entry in the list. It looks like the columns of this data frame are either numbers (numeric or integer), or factors. We'll explain factors shortly, but first a few more things about data frames.

Data frames have a bit of an identity crisis, which is intentional, though a little confusing at first. As I said above, data frames are a kind of list because the columns of a dataframe can be different data types. That means you can pull out the columns of a data frame like this:

```
fish.sub[1]
```

```
##      Site
## 497    11
## 1373   11
## 2249   11
## 3125   11
## 4001   11
## 4877   11
## 5753   11
## 6629   11
## 7505   11
## 8381   11
```

```
fish.sub[2]
```

```
##      Year
## 497  2004
## 1373 2004
## 2249 2004
## 3125 2004
## 4001 2004
## 4877 2004
## 5753 2004
## 6629 2004
## 7505 2004
## 8381 2004
```

because each column is one of the entries in the list. (Note that fish.sub is just a subset of fish where I took only the first ten rows to make the output simple). You can also use the dollar sign to index the columns by name, which is a feature of

lists:

```
fish.sub$Species
```

```
##  [1] 14001 14002 14003 14004 14005 14006 14007 14008 14009 14010
```

R also lets you index a data frame like a matrix, which is convenient:

```
fish.sub[2,4]
```

```
## [1] 14002
```

This pulls out the 2nd row entry from the 4th column.

Factors

Factors are used to code data into different groups. Let's take a look at the first 10 entries of fish$namefix:

```
fish$namefix[1:10]
```

```
##  [1] Chromis punctipinnis, adult    Chromis punctipinnis, juvenile
##  [3] Oxyjulis californica, adult    Oxyjulis californica, juvenile
##  [5] Sebastes mystinus, adult       Sebastes mystinus, juvenile
##  [7] Sebastes serranoides, adult    Sebastes serranoides, juvenile
##  [9] Sebastes atrovirens, adult     Sebastes atrovirens, juvenile
## 27 Levels: Chromis punctipinnis, adult ... Semicossyphus pulcher, ma
le
```

Looks like we have 5 species that have been separately entered for adults and juveniles. The output also says the factor has 27 levels, and it shows us a very abbreviated list of those levels. Although the entries of this factor are a bunch of text, it is important to note that *a factor is different from a character vector, and the elements of a factor are not strings*. If this was a character vector, every entry of the output would be shown with quote marks around it. What's the difference between a character vector and a factor? When you make a character vector, you're just storing a bunch of strings, and R doesn't care if some of the strings are the same or not. In contrast, the point of a factor is that it is a grouping variable that is coding which items belong to the same group. So the way R actually stores a factor is with a vector of integers, with labels (levels) that correspond to those integers. We can see this using the function unclass():

```
unclass(fish$namefix[1:10])
```

```
##  [1]  1  2 13 14 21 22 23 24 19 20
## attr(,"levels")
```

```
##  [1] "Chromis punctipinnis, adult"     "Chromis punctipinnis, juvenile"
##  [3] "Embiotoca jacksoni, adult"       "Embiotoca jacksoni, juvenile"
##  [5] "Embiotoca lateralis, adult"      "Embiotoca lateralis, juvenile"
##  [7] "Girella nigricans, adult"        "Girella nigricans, juvenile"
##  [9] "Halichoeres semicinctus, female" "Halichoeres semicinctus, male"
## [11] "Hypsypops rubicundus, adult"     "Hypsypops rubicundus, juvenile"
## [13] "Oxyjulis californica, adult"     "Oxyjulis californica, juvenile"
## [15] "Paralabrax clathratus, adult"    "Paralabrax clathratus, juvenile"
## [17] "Rhacochilus vacca, adult"        "Rhacochilus vacca, juvenile"
## [19] "Sebastes atrovirens, adult"      "Sebastes atrovirens, juvenile"
## [21] "Sebastes mystinus, adult"        "Sebastes mystinus, juvenile"
## [23] "Sebastes serranoides, adult"     "Sebastes serranoides, juvenile"
## [25] "Semicossyphus pulcher, female"   "Semicossyphus pulcher, juvenile"
## [27] "Semicossyphus pulcher, male"
```

Now it's revealed that each fish species is coded as a different integer (from 1 to 27), and then the function tells us the names of the levels that correspond to the integers. You could also look at just the level names with levels(fish$namefix). Note that the level names are stored as a character vector; in other word a factor is an integer vector with an associated character vector of level names. Why does R store factors like this? If you recall when we discussed linear models, in order to fit an Anova the factor has to be converted to a numeric coding scheme, and this way of storing factors facilitates that. Also this makes it easy to change the names of the factor levels, for example if I want to change `"Chromis punctipinnis, adult"` to something shorter, I could do this:

```
levels(fish$namefix)[1] = "C. punct ad"
```

Converting between data types

A vector can be converted from one type (numeric, character, logical) to another, and conversion between more complex classes is possible as well. Why would you want to do this? Sometimes you need to do it to clean up your data. For example, you may read in a file and expect a certain column of your data frame to be numeric, but instead you get this:

```
example.vector

## [1] "2.1"      "3.0"      "0.2"      "surprise"
```

Someone made a mistake in data entry, or had a weird coding scheme, and so there is some text mixed in with the numbers. And when you read the file into R it treated this vector as a character vector (or maybe as a factor) by default, due to the text. Now you want to turn the text entries into NAs and keep the numbers:

```
example.vector

## [1] "2.1"      "3.0"      "0.2"      "surprise"
```

```
example.vector[example.vector == "surprise"] = NA
example.vector = as.numeric(example.vector)
example.vector
```

```
## [1] 2.1 3.0 0.2  NA
```

First I used logical indexing to make the NA changes, then I converted the vector from character to numeric using as.numeric(). Note that a numeric vector is allowed to have NA's. NA is not a string, it is a special kind of built-in value.

Converting from numeric (or character or factor) to logical often comes up when exploring some data. For example, in the fish dataset, maybe I want to know how many zero counts there are:

```
fish.sub$count
```

```
##  [1] 55 50 34 42  0  0  0  0  0  0
```

I can just count them by eye here, but with the full dataset I would want to calculate it. One way to do this is to 1) create a logical vector for whether an entry equals zero or not, 2) convert that logical vector to numbers, 3) add the numbers:

```
fish.sub$count == 0
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
as.numeric(fish.sub$count == 0)
```

```
##  [1] 0 0 0 0 1 1 1 1 1 1
```

```
sum(fish.sub$count == 0)
```

```
## [1] 6
```

This seems like a lot of steps, but as you see from the last statement, you can actually do it with very little code. When you ask R to sum() the logical vector returned by `fish.sub$count == 0`, it automatically converts the TRUE's to 1's and the FALSE's to 0's.

In this case I could also use table, which is easy:

```
table(fish.sub$count)
```

```
##
##  0 34 42 50 55
##  6  1  1  1  1
```

But in other cases, such as asking 'how many values are less than zero', table won't work but the logical conversion will.

A last example of type conversion is making factors. If you want to make a new factor, you can start with a character vector and use as.factor():

```
my.factor = rep(c("a", "b"), each = 4)
my.factor
```

```
## [1] "a" "a" "a" "a" "b" "b" "b" "b"
```

```
my.factor = as.factor(my.factor)
my.factor
```

```
## [1] a a a a b b b b
## Levels: a b
```

Note how the character version has each element in quotes (because they are strings), while the factor version does not (but has associated levels).

Object attributes and structure

Earlier I said that everything in R is an object, and that objects are distinguished by their class. That means you can take anything and get some info about it with class(). For example, what class is the linear model function lm()?

```
class(lm)
```

```
## [1] "function"
```

Yep it's a function. That's a fairly useless example, but it shows that you can stick anything into class() because everything is an object.

Let's look at something more practical. Different classes of data structure typically have different *attributes*. As the name implies, an attribute is not the data itself, but it is something that helps you interact with the data. You can see what the attributes are with the attributes() function. Let's see the attributes for our fish data frame:

```
attributes(fish.sub)
```

```
## $names
## [1] "Site"    "Year"    "Date"    "Species" "count"   "namefix"
##
## $row.names
##  [1]  497 1373 2249 3125 4001 4877 5753 6629 7505 8381
##
## $class
## [1] "data.frame"
```

This data frame (and all other data frames) has three attributes: names, row.names, and class. The names are the column names, and the row.names are the row names. I don't use row names that often, but you can use them to index your data by row:

```
fish.sub["497",]
```

```
##     Site Year    Date Species count                      namefix
## 497   11 2004 10/1/04   14001    55 Chromis punctipinnis, adult
```

So attributes() tells you about what accessory parts are there to help you interact with some object. If you want to go further and get a lot of info about some object, you can use str()

```
str(fish.sub)
```

```
## 'data.frame':    10 obs. of  6 variables:
##  $ Site   : int  11 11 11 11 11 11 11 11 11 11
##  $ Year   : int  2004 2004 2004 2004 2004 2004 2004 2004 2004 2004
##  $ Date   : Factor w/ 734 levels "10/1/04","10/1/08",..: 1 1 1 1 1 1
 1 1 1 1
##  $ Species: num  14001 14002 14003 14004 14005 ...
##  $ count  : int  55 50 34 42 0 0 0 0 0 0
##  $ namefix: Factor w/ 27 levels "Chromis punctipinnis, adult",..: 1
2 13 14 21 22 23 24 19 20
```

str() displays the internal structure of an R object, i.e. it shows you all the real gory innards. Here we don't learn much we didn't already know, but it shows you at a glance what the dimensions of the data frame are, what classes the different columns are, etc.

Attributes can be useful for figuring out exactly what kind of info is created by a complex function. Maybe I fit a model to the fish counts, where I test whether the mean count is different between the different species:

```
fish.model = lm(count ~ namefix, data = fish)
summary(fish.model)
```

```
##
## Call:
## lm(formula = count ~ namefix, data = fish)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -34.3   -0.9   -0.3   -0.1 1350.6
##
## Coefficients:
##                                       Estimate Std. Error t value
## (Intercept)                             34.277      0.788    43.5
## namefixChromis punctipinnis, juvenile  -14.878      1.115   -13.3
## namefixEmbiotoca jacksoni, adult       -32.725      1.115   -29.4
## namefixEmbiotoca jacksoni, juvenile    -34.033      1.115   -30.5
## namefixEmbiotoca lateralis, adult      -33.779      1.115   -30.3
## namefixEmbiotoca lateralis, juvenile   -34.161      1.115   -30.6
## namefixGirella nigricans, adult        -33.370      1.115   -29.9
## namefixGirella nigricans, juvenile     -34.273      1.115   -30.8
## namefixHalichoeres semicinctus, female -33.694      1.214   -27.8
## namefixHalichoeres semicinctus, male   -33.955      1.214   -28.0
```

```
## namefixHypsypops rubicundus, adult      -32.703      1.115   -29.3
## namefixHypsypops rubicundus, juvenile   -34.201      1.115   -30.7
## namefixOxyjulis californica, adult      -27.733      1.115   -24.9
## namefixOxyjulis californica, juvenile   -30.998      1.115   -27.8
## namefixParalabrax clathratus, adult     -32.652      1.115   -29.3
## namefixParalabrax clathratus, juvenile  -34.050      1.115   -30.6
## namefixRhacochilus vacca, adult         -33.645      1.115   -30.2
## namefixRhacochilus vacca, juvenile      -34.193      1.115   -30.7
## namefixSebastes atrovirens, adult       -33.668      1.115   -30.2
## namefixSebastes atrovirens, juvenile    -34.095      1.115   -30.6
## namefixSebastes mystinus, adult         -33.736      1.115   -30.3
## namefixSebastes mystinus, juvenile      -31.798      1.115   -28.5
## namefixSebastes serranoides, adult      -34.018      1.115   -30.5
## namefixSebastes serranoides, juvenile   -33.833      1.115   -30.4
## namefixSemicossyphus pulcher, female    -32.557      1.115   -29.2
## namefixSemicossyphus pulcher, juvenile  -34.079      1.273   -26.8
## namefixSemicossyphus pulcher, male      -34.100      1.115   -30.6
##                                        Pr(>|t|)
## (Intercept)                             <2e-16 ***
## namefixChromis punctipinnis, juvenile   <2e-16 ***
## namefixEmbiotoca jacksoni, adult        <2e-16 ***
## namefixEmbiotoca jacksoni, juvenile     <2e-16 ***
## namefixEmbiotoca lateralis, adult       <2e-16 ***
## namefixEmbiotoca lateralis, juvenile    <2e-16 ***
## namefixGirella nigricans, adult         <2e-16 ***
## namefixGirella nigricans, juvenile      <2e-16 ***
## namefixHalichoeres semicinctus, female  <2e-16 ***
## namefixHalichoeres semicinctus, male    <2e-16 ***
## namefixHypsypops rubicundus, adult      <2e-16 ***
## namefixHypsypops rubicundus, juvenile   <2e-16 ***
## namefixOxyjulis californica, adult      <2e-16 ***
## namefixOxyjulis californica, juvenile   <2e-16 ***
## namefixParalabrax clathratus, adult     <2e-16 ***
## namefixParalabrax clathratus, juvenile  <2e-16 ***
## namefixRhacochilus vacca, adult         <2e-16 ***
## namefixRhacochilus vacca, juvenile      <2e-16 ***
## namefixSebastes atrovirens, adult       <2e-16 ***
## namefixSebastes atrovirens, juvenile    <2e-16 ***
## namefixSebastes mystinus, adult         <2e-16 ***
## namefixSebastes mystinus, juvenile      <2e-16 ***
## namefixSebastes serranoides, adult      <2e-16 ***
## namefixSebastes serranoides, juvenile   <2e-16 ***
## namefixSemicossyphus pulcher, female    <2e-16 ***
## namefixSemicossyphus pulcher, juvenile  <2e-16 ***
## namefixSemicossyphus pulcher, male      <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 23.3 on 22818 degrees of freedom
## Multiple R-squared:  0.0893, Adjusted R-squared:  0.0882
## F-statistic:   86 on 26 and 22818 DF,  p-value: <2e-16
```

I've done two things here that aren't that smart: I've used a normal error distribution for count data, and also by using a standard linear model (instead of a

mixed model) I've fit a lot of parameters to estimate all these species differences. But we will get to these issues later in the course.

Now maybe I want to understand exactly what lm() is doing. What kind of object is the fitted model?

```
class(fish.model)
```

```
## [1] "lm"
```

OK it's a class called "lm", that makes sense. What are the attributes of the fitted model?

```
attributes(fish.model)
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
##
## $class
## [1] "lm"
```

OK it looks like the model contains all kinds of stuff, listed under "names". That means I can access this stuff with the $ operator:

```
fish.model$coefficients
```

```
##                                (Intercept)
##                                      34.28
##  namefixChromis punctipinnis, juvenile
##                                     -14.88
##       namefixEmbiotoca jacksoni, adult
##                                     -32.72
##    namefixEmbiotoca jacksoni, juvenile
##                                     -34.03
##      namefixEmbiotoca lateralis, adult
##                                     -33.78
```

…(abbreviated output)

So this returns all the fitted coefficients for the model, which could be useful for plotting and stuff. Indeed it's useful enough that there's also a built-in function to extract the coefficients:

```
coef(fish.model)
```

```
##                                (Intercept)
##                                      34.28
##  namefixChromis punctipinnis, juvenile
```

```
##                                    -14.88
##       namefixEmbiotoca jacksoni, adult
##                                    -32.72
##    namefixEmbiotoca jacksoni, juvenile
##                                    -34.03
```

…

Now to continue our line of thought here, everything in R is an object that (usually) has some useful attributes, so what does coef() return?

```
class(coef(fish.model))
```

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

OK this is just a numeric vector. Does it have any attributes?

```
attributes(coef(fish.model))
```

```
## $names
##  [1] "(Intercept)"
##  [2] "namefixChromis punctipinnis, juvenile"
##  [3] "namefixEmbiotoca jacksoni, adult"
##  [4] "namefixEmbiotoca jacksoni, juvenile"
```

…

Ah, the names of the coefficients are an attribute. That's why when I use coef() I get numbers with names associated with them. So this is an example of a vector with names; not all vectors have names, but you can give any vector names, and it can be useful e.g. for extracting coefficients:

```
coef(fish.model)["namefixGirella nigricans, adult"]
```

```
## namefixGirella nigricans, adult
##                          -33.37
```