Unit 3: Acquiring Flexibility with your Data

Today we will delve into more complicated data management issues, as well as how to get basic summary statistics from you data.

For today's fun we will use a data set from the Ecological Archives (http://esapubs.org/archive/ecol/E090/119/metadata.htm) on nesting ecology and offspring recruitment of a population of painted turtles.

Background Info and Data Description from metadata information accompanying data:

> We have been monitoring a population of painted turtles (*Chrysemys picta*) in Illinois, USA, for 18 years in an effort to better understand the ecology, evolution and demographic consequences of maternal nesting behavior (i.e., timing of nesting and nest-site selection) and temperature-dependent sex determination on offspring phenotype. The population of painted turtles used for this ongoing long-term project was studied at one major nesting beach (41°57' N, 90°07' W) along the backwaters of the Mississippi River near Thomson, Illinois, USA. Painted turtles are an aquatic species that nests in large numbers on nesting beaches at our study site. They dig shallow nests that contain 3–21 eggs per clutch (mean = 10.5, mode = 10), and females may deposit up to three separate nests during a single nesting season in this area. Hatchling painted turtles remain in the nest for winter hibernation and emerge the following spring (Weisrock and Janzen 1999).
>
> Data were collected to examine the role of nesting phenology, nest-site selection, depredation rates, and clutch success on hatchling recruitment. The site was monitored daily for the duration of the nesting season between 1989 and 2006. We excavated nests in the fall of each year to determine nest survivorship. These efforts provided data for each nest describing date of laying, vegetation cover, depredation, and hatchling survival. For a subset of the nests, clutch size was also known.

B. Variable definitions

Year: Year of data collection.

Nest: Unique nest identification number.

Nest_Date: Date nest was laid if known. Window of dates in which nest must have been laid, or date before (“<”) or after (“>”) which nest must have been laid if the nest was discovered later (Julian days).

S+W_Vegetation: South + West vegetation cover (%) over a nest.

Clutch_Size: The number of eggs in the nest, if recorded.

Nest_Predation: Indicates whether a nest was depredated before the fall excavation period (third weekend in September); 1 = depredated, 0 = not depredated.

Nest_Survival: Indicates whether a nest produced any live hatchlings at the fall excavation period (third weekend in September); 1 = the nest produced at least one live hatchling; 0 = the nest produced no live hatchlings (due to depredation or clutch mortality from unknown causes).

Live_Hatchlings: The number of live hatchlings excavated from the nest in the fall sampling period.

To begin, download the data file to your laptop and read it into R. Let's call the data `turtles`, so we are all dealing with the same name.

After reading in a data file you should always check it using the `str()` function.

Immediately clear are two difficulties with the data that must be addressed. The missing data are coded as -999.9; we need to let R know this means missing data. Also nest_date has '<' and '>' used in many observations. Note that because of this, R has assumed this variable is a factor.

We can correct both of these easily, by setting better options in our `read.csv()` function.

```
turtles <- read.csv("turtle_data.csv", stringsAsFactors = FALSE, na.strings
  = c("-999.9", "-1000", ""))
```

To check if this worked as expected, we can run `str()` again.

The function `summary()` is also a helpful way to get an overview of the data.

I also immediately decide to make the variable names easier for typing using the following:

```
names(turtles) <- c("year","nest","date", "sw.veg", "clutch.size",
"predation", "survival", "hatchlings")
```

Missing Values

R codes missing values as NA. This has special meaning in R – it is not numeric or character and most importantly it is not comparable. This means you cannot look for missing values by a logical test such as turtles$Nest_Survival == NA (this will always be false). Instead you must use functions in R for working with missing values. This include `is.na()` and `na.omit()`. Similarly, R uses the symbol NaN (not a number) if a returned value is not defined (e.g. 0/0). This is summarized in the table below taken from *R in Action*, 2nd ed. (2015).

Table 18.1 Examples of return values for the `is.na()`, `is.nan()`, and `is.infinite()` functions

x	is.na(x)	is.nan(x)	is.infinite(x)
x <- NA	TRUE	FALSE	FALSE
x <- 0 / 0	TRUE	TRUE	FALSE
x <- 1 / 0	FALSE	FALSE	TRUE

Recode and Clean Up Data

For the date variable (which is actually Julian day), we still have the issue that the data is not in a format that we can actually use in the analysis. For example, we might want to create categories based on Julian days, but this is difficult to do when there are 80 different character codings! So we will start by fixing our date variable.

```
turtles$date <- as.character(turtles$date)
```
make vector of mode character so we can use grep; if necessary – not in our case because we changed read.csv function to not make character data a factor

```
turtles$date <- sub("<", "0",turtles$date)   # sub() is grep function that allows us
        to replace characters in values; here we replace the '<' sign with a zero to make this a
        number

turtles$date <- sub(">", "0", turtles$date) # ditto for '>' sign

turtles$date <- sub("-1..", " ", turtles$date) # note the use of a period in the
        expression to indicate it can be any character in that place

turtles$date <- as.numeric(turtles$date) # we'll make vector mode numeric now
        so that we can base levels on numbers
```

If we only have 2 levels, we can use `ifelse()` function

```
turtles$date2 <- factor(ifelse(turtles$date <= 165, "early", "late"))
```

If we have >2 levels we can use within()

```
turtles <- within(turtles, {
                    season <- NA
                    season[date < 160] <- "early"
                    season[date >= 160 & date <= 170] <- "middle"
                    season[date > 170] <- "late"
                    }
                   )

turtles$season <- factor(turtles$season, ordered=TRUE, levels=c("early",
     "middle", "late"))
str(turtles)
```

For our purposes here, let's also remove all the observations with missing data.

```
rturtles <- na.omit(turtles)
```

Summarizing Data

`summary()` – the base package of R has this function that will provide the following for each numerical variable: minimum value, 1st quartile, median, mean, 3rd quartile, maximum value; for factors the number of observations for the 6 most common levels will be given.

Aggregating Data

Now that we have a reasonably clean data set, we can start to explore it with some basic summaries. The function aggregate() provides a useful way to do this with data frames that have factors.

Let's see if we can get some information about the %cover of the nests.

```
aggregate(rturtles$sw.veg, by=list(rturtles$year), mean)
aggregate(rturtles$clutch.size, by=list(rturtles$year), mean)
```

```
aggregate(rturtles$clutch.size, by=list(rturtles$year, rturtles$season),
mean)
```

Note that `aggregate()` is meant specifically for working with data frames (the 'by' option requires a factor). If you have a matrix or array, then the function `apply()` aggregates data by row or column.

```
apply(matrix_name, margin (=1 for rows; =2 for columns), function)
```

e.g., for a matrix annual number of murders per year and province (each row is a year and each column is a province) we would use `apply(murders, 1, sum)` to get the total number of murders in all provinces for a year and `apply(murders, 2, mean)` to get the average number of murders per year for each province.

For a list, use `sapply(list_name, function)` to apply a function to each list component. e.g. the same morbid data as above might instead be in a list object with each province a separate vector in the list composed of murders per year for that province.

Calculate means and SE for hatchlings

One thing that many are thrown by when graphing in R is the lack of a built-in function to add standard error bars to a plot. However, it is not necessary because it is trivial to add in this information using the existing language. Let's consider that here, as it is something we often do when first organizing and exploring data. Imagine we want to look at data on the number of hatchlings produced each year.

First we'll create a new data.frame of mean number of hatchlings per year.

```
hatch.means <- aggregate(turtles$hatchlings, by=list("year" =
  turtles$year), mean)
```

Now we create a function to calculate SE. The vectorized feature of R functions makes this easy (we talk about this more on the section about writing functions below).

```
se <- function(x) {sqrt(var(x)/length(x))} # create function called "se"
```

Use `aggregate()` to apply the function for hatchlings by year. You can actually write the function into the aggregate() function option, as below, or, if you have defined se(), you can put that into the aggregate function call.

```
hatch.se <- aggregate(turtles$hatchlings, by=list("year"=turtles$year),
  function(x){sqrt(var(x)/length(x))})
```

Use the merge() function to put the two data.frames together:

```
hatch.plot <- merge(hatch.means, hatch.se, by = “year”)
names(hatch.plot)[2:3] <- c("mean", "se")
```

Now we can use the new SE data to add error bars to our graph. We add these types of lines to our graph using the function `arrows()`.

```
plot(hatch.plot$mean ~ as.numeric(hatch.plot$year))
arrows(x0 = as.numeric(hatch.plot$year), y0 = hatch.plot$mean -
  hatch.plot$se, y1 = hatch.plot$mean + hatch.plot$se, length = 0.05, angle
  = 90, code = 3)
```

Tables

The functions `table()` and `xtabs()` in R provide ways to easily create cross-tabulation tables. These are useful for checking numbers of observations in specific factor level combinations of data sets. These also allow contingency table analysis.

```
table(turtles$year, turtles$season)

xtabs(~year + season, data=turtles)

xtabs(cbind(clutch.size, hatchlings) ~ year + season, data=turtles)

ftable(xtabs(cbind(clutch.size,hatchlings) ~ year + season, data =
  turtles))
```

We can modify our table to be in proportions using `prop.table()` and use `addmargins()` to create row and/or column summaries (by default, the sum).

Chi-square Test of Independence: this is readily done in R using `chisq.test()`. Let's save this result and use it to see how to interpret the objects returned from functions.

```
turtle_table <- xtabs(~turtles$year + turtles $season, data = turtles)

result <- chisq.test(turtle.table)
```

The chisq.test function produces much more information than what you get if you print ‘result’. To see this, use `str(result)`.

We can explore this more after a short side trip to better understand indexing of lists.

Tables for Printing

Unfortunately getting a nicely formatted table out of R is difficult unless you use LaTeX. The package xtable has a function of the same name that allows you to save a LaTeX or HTML-formatted table. The package SWeave together with LaTeX allows a great deal of flexibility and the possibility of producing publishable reports from within R. The downside of course, is you have to learn LaTeX first.

If you save your table as a text or csv file it should import into Excel readily and from here you can usually get it into a table in Word and format it there.

Lists and Indexing

Remembering that data frames are actually a type of list, and given that most analysis output that you will be dealing with is saved in lists, it is well worth spending a bit more time at this stage to become more comfortable with using the index operators ("[]" and "[[]]") with lists.

Create the following list to explore this topic:

```
part1 <- c(letters[1:4])
part2 <- c(seq(1:6))
part3 <- c(month.abb[1:12])
part4 <- pi
(list <- list(c(part1,part2,part3,part4))) # use parentheses around whole
command to have result output to screen
(l1 <- list("part1"=part1, "part2"=part2, "part3" = part3, "part4"=part4))
(l2 <- list(part1,part2,part3,part4))
```

Note the distinction between the three lists: "list" is a one-vector list with the elements of the parts concatenated together into a vector of length 23; "l1" is a list containing four named components of different lengths; "l2" is also a list containing four components, but they are not named. (In this case, each component is a vector, but a component can also be another list).

`object[integer]` pulls out a component in a list (as a list):

```
> l1[2]

$part2
[1] 1 2 3 4 5 6 # class(l1[2]) is list
```

`object[[integer]]` pulls out the elements in a component (as a matrix or vector):

```
> l1[[2]]
[1] 1 2 3 4 5 6  # class(l1[[2]]) is int
```

This distinction is very important at times! Note also that `object$name` is equivalent to `object[[integer]]`.

If we want to pull out one element of a component, we can append another [] notation. e.g.,

```
> l1[[1]][3]
[1] "c"
```

We can extend this as necessary when our list contains lists.

```
l3 <- list(l1, "p2"=part2, "p3"=part3)
> l3[[1]][[2]]
[1] 1 2 3 4 5 6
> l3[[1]][[2]][3]
[1] 3
```

Okay, now lets go back to `str(result)` with the data from our chi-square test and see if we can interpret it better.

The Reshape2 Package

You will need to install this package first and then load it using `library("reshape2")`.

Crucial to using the reshape2 package is understanding the wide versus long format of data.

Data are in **long form** if there is **only one data point per row**. When there is more than one measured variable (data point) in a row, data are said to be in **wide form**.

Consider a subset of the hypothetical data set below. The data are in the file "small_table.csv". Place these data into the object "examp".

Notice that the data need a clean-up in the column "Subject.ID" both the species and replicate number are combined. Since we are interested in species differences we need to separate out species as a factor. *(This is a contrived example to give you more information on how to manipulate data – I cannot imagine any of you would code your data so foolishly!)* To do this, we'll use a handy function called `strsplit()` (string-split)

```
examp <- read.csv("small_table.csv", header=TRUE, sep=",")
```

```
names(examp) <- c("id", "treatment", "svl", "mass")  #  I change the column names
        to make it easier for me
```

```
spprep <- strsplit(as.character(examp$id), ".", fixed=TRUE)  # it is the column 'id'
      that I wish to split; it must be a character vector and I want to split it either side of the
      period; fixed=TRUE indicates that the character supplied is the actual character and not a
      regular expression such as those we used with grep()
```

```
examp$spp <- sapply(spprep, "[", 1) # strsplit returns a list; the first value in each list is
      what was to the left of '.', so we extract that part to a new column called 'spp'
```

```
examp$rep <- sapply(spprep, "[", 2)   # now we extract the info to the right of the '.' into
      a column 'rep'
```

Now I'll clean things up a bit, so the data print logically:

```
examp[,1] <- examp[,5] # copy column 5 to column 1
examp <- examp[,-5] # delete column 5
names(examp)[1] <- "spp" # remember that names are an attribute of the data frame, not
  an element of the data frame
```

The table below shows these data in wide form. (This is also how data currently print.) The number 3.850 is only uniquely identified if we also know is the variable SVL, as well as that it is species is Rcat and the treatment A. In the reshape package, we refer to species and treatment as **identifier variables** and svl and mass as **measurement variables**.

Spp	Treatment	SVL	Mass	Rep
Rcat	A	3.850	5.752	1
Rcat	A	3.587	4.608	2
Rcat	B	4.983	6.710	3
Rcat	B	4.631	6.499	4
Rs	A	2.012	1.828	1
Rs	A	2.652	2.908	2
Rs	B	5.027	5.549	3
Rs	B	4.592	5.190	4

In long format, the data are re-arranged to have only one measured value per row. To do this, we need to create a new variable that will indicate whether the measured variable is SVL or Mass. The resulting table is seen below.

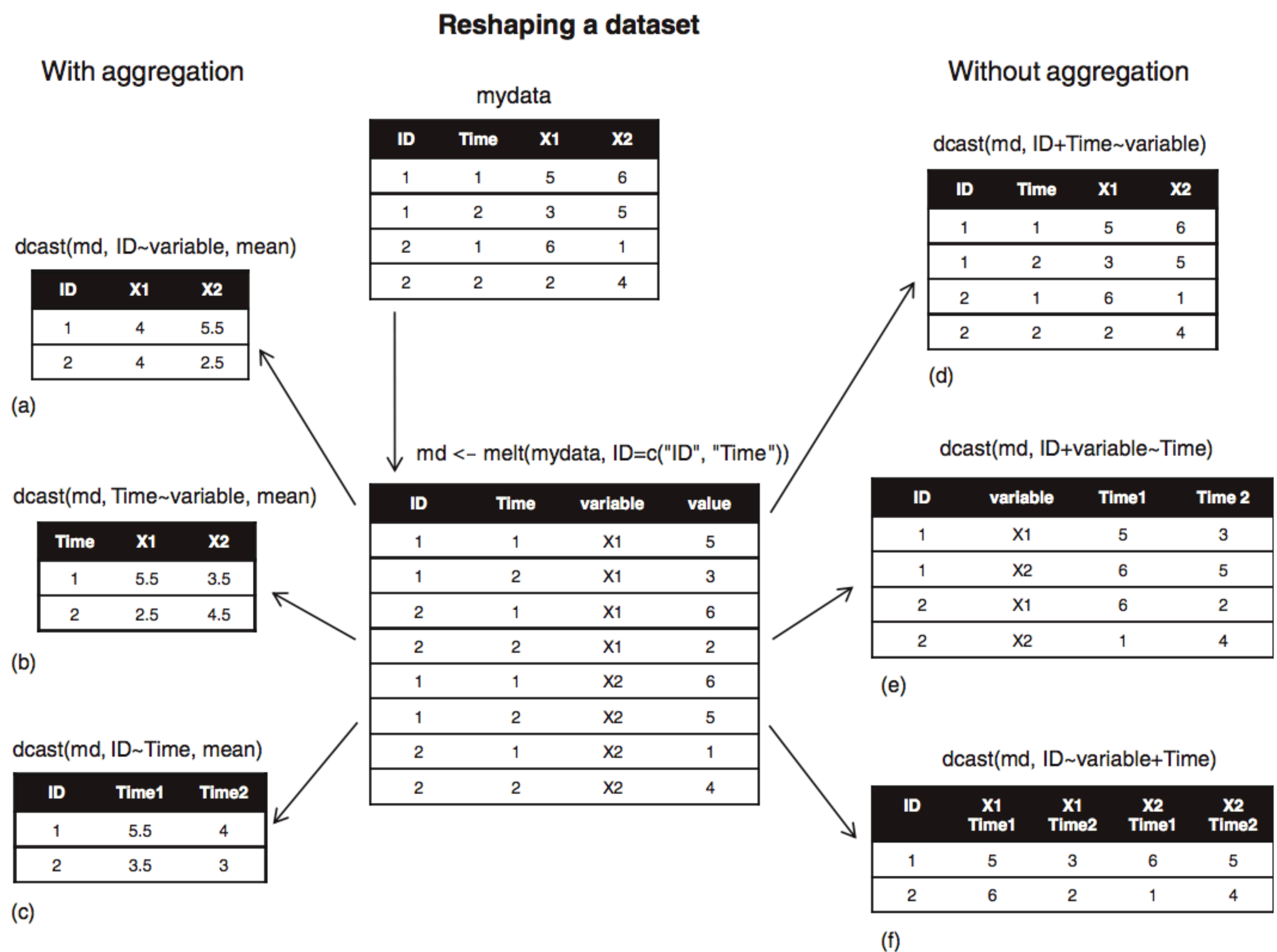
Spp	Treatment	Rep.	trait	value
Rcat	A	1	svl	3.850
Rcat	A	2	svl	3.587
Rcat	B	3	svl	4.983
Rcat	B	4	svl	4.631
Rs	A	1	svl	2.012
Rs	A	2	svl	2.652
Rs	B	3	svl	5.027
Rs	B	4	svl	4.592
Rcat	A	1	mass	5.752
Rcat	A	2	mass	4.608
Rcat	B	3	mass	6.71
Rcat	B	4	mass	6.499
Rs	A	1	mass	1.828
Rs	A	2	mass	2.908
Rs	B	3	mass	5.549
Rs	B	4	mass	5.19

In the reshape2 package, going from the wide to the long format is known as 'melting' the data. The function to do this is `melt()`. You can specify id.vars and/or measured.vars; if you specify neither then the function will assume all factor and integer variables are ID and the rest are measured. If you specify either one of these, then the rest of the variables present are assumed to belong to the other class. By default the new variable that must be created will have the name "value", but you can choose to give it a more descriptive name as we do below.

```
library(reshape2)

examp.m <- melt(examp, id.vars=c("spp", "treatment"), variable.name="trait")
```

The reason to do this is that once you have the data in this format you can use the function `dcast()` to re-format the data in any way you please. You can also choose to aggregate the data as you rearrange it. Let's run through some examples.



mydata

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

md <- melt(mydata, ID=c("ID", "Time"))

ID	Time	variable	value
1	1	X1	5
1	2	X1	3
2	1	X1	6
2	2	X1	2
1	1	X2	6
1	2	X2	5
2	1	X2	1
2	2	X2	4

(a) dcast(md, ID~variable, mean)

ID	X1	X2
1	4	5.5
2	4	2.5

(b) dcast(md, Time~variable, mean)

Time	X1	X2
1	5.5	3.5
2	2.5	4.5

(c) dcast(md, ID~Time, mean)

ID	Time1	Time2
1	5.5	4
2	3.5	3

(d) dcast(md, ID+Time~variable)

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

(e) dcast(md, ID+variable~Time)

ID	variable	Time1	Time 2
1	X1	5	3
1	X2	6	5
2	X1	6	2
2	X2	1	4

(f) dcast(md, ID~variable+Time)

ID	X1 Time1	X1 Time2	X2 Time1	X2 Time2
1	5	3	6	5
2	6	2	1	4

Figure 5.1 from Kabacoff, R. I. (2015) *R in Action,* 2nd ed., Manning, Shelter Island, NY p. 113.

Reshaping without Aggregating

To get back the original data format, we use:

```
dcast(examp.m, spp + treatment + rep ~ trait) # regenerate original table
dcast(examp.m, spp + treatment + rep ~ ...) # same as above "..." = rest of variables
dcast(examp.m, ... ~ trait) # also the same as above
```

We can replicate our melted data form by having all variables as columns and none as rows:

```
dcast(examp.m, ... ~ .) # same as examp.m - all variables in columns, no variables in rows
```

More alternatives:

```
dcast(examp.m, spp + rep ~ trait + treatment) #this reveals our inaccurate rep. coding
examp.m[,3] <- rep(1:2) # let's fix replicate coding to be accurate
dcast(examp.m, spp + rep ~ trait + treatment)
dcast(examp.m, spp + trait + treatment ~ rep)
```

Aggregating Data with Reshape2

```
dcast(examp.m, spp + trait + treatment ~.) # get number of replicates for each treatment combination
dcast(examp.m, spp + trait + treatment ~., mean) # get mean for each treatment combination
dcast(examp.m, spp + trait + treatment ~., sd) # now get standard deviation
dcast(examp.m, treatment ~ trait, mean)
examp.3d <- acast(examp.m, treatment ~ trait ~ spp, mean) # produce 3-D arrays
acast(examp.m, treatment ~ spp + rep ~ trait, mean)
```

Data Transformation

Data values are easily transformed in R, simply assign a new column name in your data frame for the transformed data. e.g.,

```
examp.m$t_value <- log(examp.m$value) # default for log() is base = e, others can be specified using base option
```

R contains all the typical mathematical functions, including:

```
abs(), sqrt(), trun(), round(), signif(),
sin(), cos(), tan(),
asin(), acos(), atan()
```

R also provides the useful function `scale()` to standardize your variable to mean = 0 and standard deviation = 1.

```
examp.m$std_value <- scale(examp.m$value)
```