# Exploratory data analysis and graphics: lab 2



*30 October 2017*

## 1. Reading data

This lab will cover many if not all of the details you actually need to know about R to read in data and produce the figures shown in Chapter 2, and more. The exercises, which will be considerably more difficult than those in Lab 1, will typically involve variations on the figures shown in the text. You will work through reading in the different data sets and constructing the figures shown, or variants of them. It would be even better to work through reading in and making exploratory plots of your own data.

Find the file called 'seedpred.dat': it's in the right format (plain text, long format), so you can just read it in with

```
data = read.table("seedpred.dat", header = TRUE)
```

Add the variable available to the data frame by combining taken and remaining (using the $ symbol):

```
data$available = data$taken + data$remaining
```

**Pitfall #1: finding your file** If `R` responds to your `read.table()` or `read.csv()` command with an error like

```
Error in file(file, "r") : unable to open connection In addition:
Warning message: cannot open file 'myfile.csv'
```

it means it can't find your file, probably because it isn't looking in the right place. By default, R's working directory is the directory in which the R program starts up, which is (again by default) something like `C:/Program Files/R/rw2010/bin`. (`R` uses / as the [operating-system-independent] separator between directories in a file path.) The simplest way to change this for the duration of your `R` session is to go to `File/Change dir ...`, click on the Browse button, and move to your Desktop (or wherever your file is located). If you are using Rstudio, go to 'Session' and click 'Set working directory'. You can also use the `setwd()` command to set the working directory (`getwd()` tells you what the current working directory is). While you could just throw everything on your desktop, it's good to get in the habit of setting up a separate working directory for different projects, so that

your data files, metadata files, R script files, and so forth, are all in the same place. Depending on how you have gotten your data files onto your system (e.g. by downloading them from the web), Windows will sometimes hide or otherwise screw up the extension of your file (e.g. adding .txt to a file called `mydata.dat`). R needs to know the full name of the file, including the extension.

For example to set a working directory::

```
setwd("D:/BobDouma/Education/CSA-34306 Ecological Models and Data in R/tutorials/")
```

*Pitfall #2: checking number of fields* The next potential problem is that R needs every line of your data file to have the same number of fields (variables). You may get an error like:

```
Error in read.table(file = file, header = header, sep = sep, quote
= quote, : more columns than column names
```

or

```
Error in scan(file = file, what = what, sep = sep, quote = quote,
dec = dec, : line 1 did not have 5 elements
```

If you need to check on the number of fields that R thinks you have on each line, use

```
count.fields("myfile.dat",sep=",")
```

(you can omit the `sep=","` argument if you have whitespace- rather than comma delimited data). If you are checking a long data file you can try

```
cf = count.fields("myfile.dat",sep=",")
which(cf!=cf[1])
```

to get the line numbers with numbers of fields different from the first line. By default R will try to fill in what it sees as missing fields with `NA` ("not available") values; this can be useful but can also hide errors. You can try

```
mydata <- read.csv("myfile.dat", fill = FALSE)
```

to turn off this behavior; if you don't have any missing fields at the end of lines in your data this should work.

If your file is a comma separated file, you can also use `read.csv`. `read.csv` has set some arguments to default, e.g. the seperator is a comma in this case (`sep=","`)

## 1.1. Checking data

Here's the quickest way to check that all your variables have been classified correctly:

```
sapply(data, class)
```

```
##   species      tcum      tint remaining     taken available
##  "factor" "integer" "integer" "integer" "integer" "integer"
```

(this applies the `class()` command, which identifies the type of a variable, to each column in your data). Non-numeric missing-variable strings (such as a star, *) will also make `R` misclassify. Use `na.strings` in your `read.table()` command:

```
mydata <- read.table("mydata.dat", na.strings = "*")
```

(you can specify more than one value with (e.g.) `na.strings=c("*","***","bad","-9999")`).

**Exercise 1.1**: Try out `head()`, `summary()` and `str()` on data; make sure you understand the results.

## 1.2 Reshaping data

It's hard to give an example of reshaping the seed predation data set because we have different numbers of observations for each species - thus, the data won't fit nicely into a rectangular format with (say) all observations from each species on the same line. However, as in the chapter text I can just make up a data frame and reshape it. Here are the commands to generate the data frame I used as an example in the text (I use `LETTERS`, a built-in vector of the capitalized letters of the alphabet, and `runif()`, which picks a specified number of random numbers from a uniform distribution between 0 and 1. The command `round(x,3)` rounds `x` to 3 digits after the decimal place.):

```
loc = factor(rep(LETTERS[1:3],2))
day = factor(rep(1:2,each=3))
val = round(runif(6),3)
d = data.frame(loc,day,val)
```

This data set is in long format. To go to wide format, we first need to (install and) load the library `reshape2`. In Lab 1 you learned how to install packages. You can load a package by `library()` or `require()`. Thus to use an additional package it must be (i) installed on your machine (with `install.packages()`) or through the menu system and (ii) loaded in your current `R` session (with `library()`):

```
library(reshape2)
d2 = dcast(d,loc~day )
```

```
## Using val as value column: use value.var to override.
```

```
d2
```

```
##   loc     1     2
## 1   A 0.696 0.550
## 2   B 0.758 0.845
## 3   C 0.712 0.230
```

loc~day specifies that `loc` will be used as rows and `day` will be put into columns.

To go back to long format, we simply write:

```
melt(d2, variable_name="day")
```

By specifying the `variable_name` to `day`, we put the columns (`1` and `2`) into a new column called `day`.

**Exercise 1.2**: Add a new column `month` (consisting of two levels) to `d` in which `loc` and `day` are nested in and reshape to wide format and back to long format. The long format is what we commonly use in statistics.

```
loc = factor(rep(LETTERS[1:3],4))
month = factor(rep(1:2,each=6))
day = factor(rep(1:2,each=3))
val = round(runif(6),3)
d1 = data.frame(loc,month,day,val)

d3 = dcast(d1,loc~month+day)
```

```
## Using val as value column: use value.var to override.
```

```
melt(d3,variable_name="month_day")
```

```
## Using loc as id variables
```

```
##    loc variable value
## 1    A      1_1 0.742
## 2    B      1_1 0.645
## 3    C      1_1 0.247
## 4    A      1_2 0.771
## 5    B      1_2 0.998
## 6    C      1_2 0.366
```

```
## 7    A      2_1 0.742
## 8    B      2_1 0.645
## 9    C      2_1 0.247
## 10   A      2_2 0.771
## 11   B      2_2 0.998
## 12   C      2_2 0.366
```

### 1.3 Advanced data types (Time permitting)

While you can usually get by coding data in not quite the right way - for example, coding dates as numeric values or categorical variables as strings - R tries to "do the right thing" with your data, and it is more likely to do the right thing the more it knows about how your data are structured.

**Strings instead of factors** Sometimes R's default of assigning factors is not what you want: if your strings are unique identifiers (e.g. if you have a code for observations that combines the date and location of sampling, and each location combination is only sampled once on a given date) then R's strategy of coding unique levels as integers and then associating a label with integers will waste space and add confusion. If all of your non-numeric variables should be treated as character strings rather than factors, you can just specify `as.is=TRUE`; if you want specific columns to be left "as is" you can specify them by number or column name. For example, these two commands have the same result:

```
data2 = read.table("seedpred.dat", header = TRUE, as.is = "Species")
```

```
## Warning in read.table("seedpred.dat", header = TRUE, as.is = "Species"):
## not all columns named in 'as.is' exist
```

```
data2 = read.table("seedpred.dat", header = TRUE, as.is = 1)
sapply(data2, class)
```

```
##     species        tcum        tint   remaining       taken
## "character"   "integer"   "integer"   "integer"   "integer"
```

(use `c()` - e.g. `c("name1","name2")` or `c(1,3)` - to specify more than one column). You can also use the `colClasses="character"` argument to `read.table()` to specify that a particular column should be converted to type character -

```
data2 = read.table("seedpred.dat", header = TRUE, colClasses = c("character",
+ rep("numeric", 4)))
```

again has the same results as the commands above. To convert factors back to strings *after* you have read them into R, use `as.character()`.

```
data2 = read.table("seedpred.dat", header = TRUE)
sapply(data2, class)
```

```
data2$Species = as.character(data2$Species)
sapply(data2, class)
```

**Factors instead of numeric values** In contrast, sometimes you have numeric labels for data that are really categorical values - for example if your sites or species have integer codes (often data sets will have redundant information in them, e.g. both a species name and a species code number). It's best to specify appropriate data types, so use colClasses to force R to treat the data as a factor. For example, if we wanted to make tcum a factor instead of a numeric variable:

```
data2 = read.table("seedpred.dat", header = TRUE, colClasses = c(rep("factor",
+ 2), rep("numeric", 3)))
sapply(data2, class)
```

**n.b.**: by default, `R` sets the order of the factor levels alphabetically. You can find out the levels and their order in a factor `f` with `levels(f)`. If you want your levels ordered in some other way (e.g. site names in order along some transect), you need to specify this explicitly. Most confusingly, `R` will sort strings in alphabetic order too, even if they represent numbers. This is OK:

```
f = factor(1:10)
levels(f)
```

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

but this is not, since we explicitly tell `R` to treat the numbers as characters (this can happen by accident in some contexts):

```
f = factor(as.character(1:10))
levels(f)
```

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

In a list of numbers from 1 to 10, "10" comes after "1" but before "2" ... You can fix the levels by using the levels argument in `factor()` to tell `R` explicitly what you want it to do, e.g.:

```
f = factor(as.character(1:10), levels = 1:10)
x = c("north", "middle", "south")
f = factor(x, levels = c("far_north", "north", "middle", "south"))
```

so that the levels come out ordered geographically rather than alphabetically. Sometimes your data contain a subset of integer values in a range, but you want

to make sure the levels of the factor you construct include all of the values in the range, not just the ones in your data. Use levels again:

```
f = factor(c(3, 3, 5, 6, 7, 8, 10), levels = 3:10)
```

Finally, you may want to get rid of levels that were included in a previous factor but are no longer relevant:

```
f = factor(c("a", "b", "c", "d"))
f2 = f[1:2]
levels(f2)
```

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

```
f2 = factor(as.character(f2))
levels(f2)
```

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

For more complicated operations with `factor()`, use the `recode()` function in the `car` package. **Exercise 1.3** : Illustrate the effects of the levels command by plotting the `factor f=factor(c(3,3,5,6,7,8,10))` as created with and without intermediate levels. For an extra challenge, draw them as two side-by-side subplots. (Use `par(mfrow=c(1,1))` to restore a full plot window.)

**Dates** Dates and times can be tricky in `R`, but you can (and should) handle your dates as type `Date` within 'R rather than messing around with Julian days (i.e., days since the beginning of the year) or maintaining separate variables for day/month/year.

You can use `colClasses="Date"` within `read.table()` to read in dates directly from a file, but only if your dates are in four-digit-year/month/day (e.g. 2005/08/16 or 2005-08-16) format; otherwise `R` will either butcher your dates or complain

```
Error in fromchar(x) : character string is not in a standard unambiguous
format
```

If your dates are in another format in a single column, read them in as character strings (`colClasses="character"` or using `as.is`) and then use `as.Date()`, which uses a very flexible format argument to convert character formats to dates:

```
as.Date(c("1jan1960", "2jan1960", "31mar1960", "30jul1960"),
        format = "%d%b%Y")
```

```
## [1] "1960-01-01" "1960-01-02" "1960-03-31" "1960-07-30"
```

```
as.Date(c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92"),
        format = "%m/%d/%y")
```

```
## [1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28" "1992-02-01"
```

The most useful format codes are `%m` for month number, `%d` for day of month, `%j%` for Julian date (day of year), `%y%` for two-digit year (dangerous for dates before 1970!) and `%Y%` for four-digit year; see `?strftime` for many more details. If you have your dates as separate (`numeric`) day, month, and year columns, you actually have to squash them together into a character format (with `paste()`, using `sep="/"` to specify that the values should be separated by a slash) and then convert them to dates:

```
year = c(2004,2004,2004,2005)
month = c(10,11,12,1)
day = c(20,18,28,17)
datestr = paste(year,month,day,sep="/")
date = as.Date(datestr)
date
```

```
## [1] "2004-10-20" "2004-11-18" "2004-12-28" "2005-01-17"
```

When you want to split a date to month, year and day, you can use 'strsplit':

```
date.c = as.character(date)
date.char = strsplit(date.c, "-" )
```

Which you subsequently can turn in to multiple colums through `matrix`:

```
dat.mat = matrix(unlist(date.char), ncol=3, byrow=TRUE)
```

Although `R` prints the dates in `date` out so they look like a vector of character strings, they are really dates: `class(date)` will give you the answer `"Date"`. Note that when using the dat.mat these are characters.

**Potential trap:** quotation marks in character variables: if you have character strings in your data set with apostrophes or quotation marks embedded in them, you have to get `R` to ignore them. I used a data set recently that contained lines like this: `Western Canyon|valley|Santa Cruz|313120N|1103145WO'Donnell Canyon`

I used

```
data = read.table("datafile", sep = "|", quote = "")
```

to tell `R` that | was the separator between fields and that it should ignore all apostrophes/single quotations/double quotations in the data set and just read them as part of a string.

### 1.4 Accessing data

**Data** To access individual variables within your data set use `mydata$varname` or `mydata[,n]` or `mydata[,"varname"]` where `n` is the column number and `varname` is the variable name you want. You can also use `attach(mydata)` to set things up so that you can refer to the variable names alone (e.g. varname rather than mydata$varname). However, **beware**: if you then modify a variable, you can end up with two copies of it: one (modified) is a local variable called `varname`, the other (original) is a column in the data frame called `varname`: **it's probably better not to attach a data set, or only until after you've finished cleaning and modifying it**. Furthermore, if you have already created a variable called varname, `R` will find it before it finds the version of `varname` that is part of your data set. Attaching multiple copies of a data set is a good way to get confused: try to remember to `detach(mydata)` when you're done.

I'll start by attaching the data set (so we can refer to `Species` instead of `data$Species` and so on).

```
attach(data)
```

```
species
data[,"species"]
data[,1]
data$Species # recommended! You explictly define the dataframe and name of the column
```

To access data that are built in to R or included in an R package (which you probably won't need to do often), say

```
data(dataset)
```

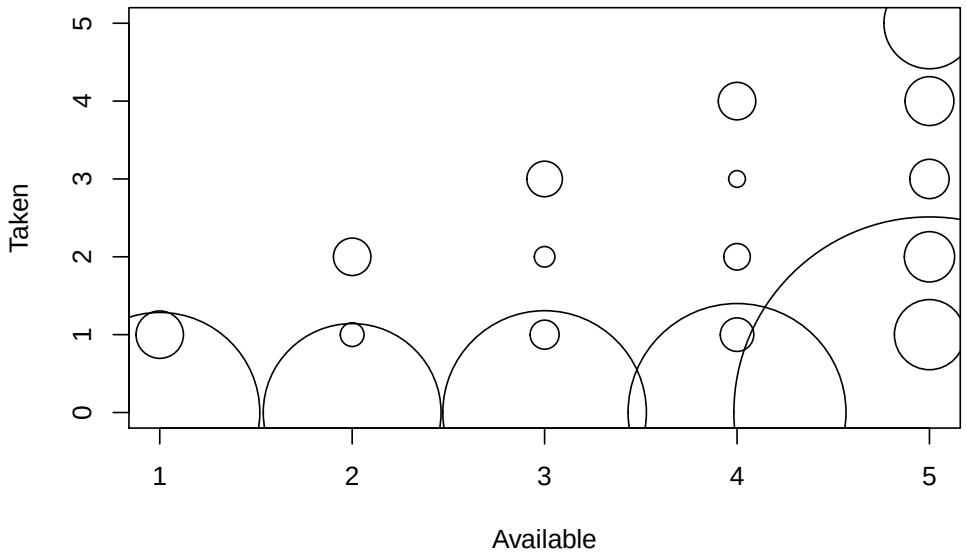
(`data()` by itself will list all available data sets.)

## 2. Exploratory graphics

Below we will show you a diversity of possible (specialized) graphs that you can produce in R. Those graphs come from a number of packages (including `plotrix` and the `base` package). In addition, we will show you the use of the `ggplot2` package. This package is very popular as it creates nice graphs without much effort. Note that the 'grammar' to produce a graph is different from how the other graphs in `R` are constructed. Graphics are really important to explore the data that you collected and want to analyse. A good data exploration improves the efficiency of your statistical analysis as you will have expectations how relations between varialbes look like.

### 2.1 Bubble plot

```
library(plotrix)
sizeplot(data$available, data$taken, xlab = "Available", ylab = "Taken")
```




will give you approximately the same basic graph shown in the chapter, although I also played around with the x- and y-limits (using xlim and ylim) and the axes. (The basic procedure for showing custom axes in `R` is to turn off the default axes by specifying `axes=FALSE` and then to specify the axes one at a time with the `axis()` command.)

I used

```
t1 = table(data$available, data$taken)
```

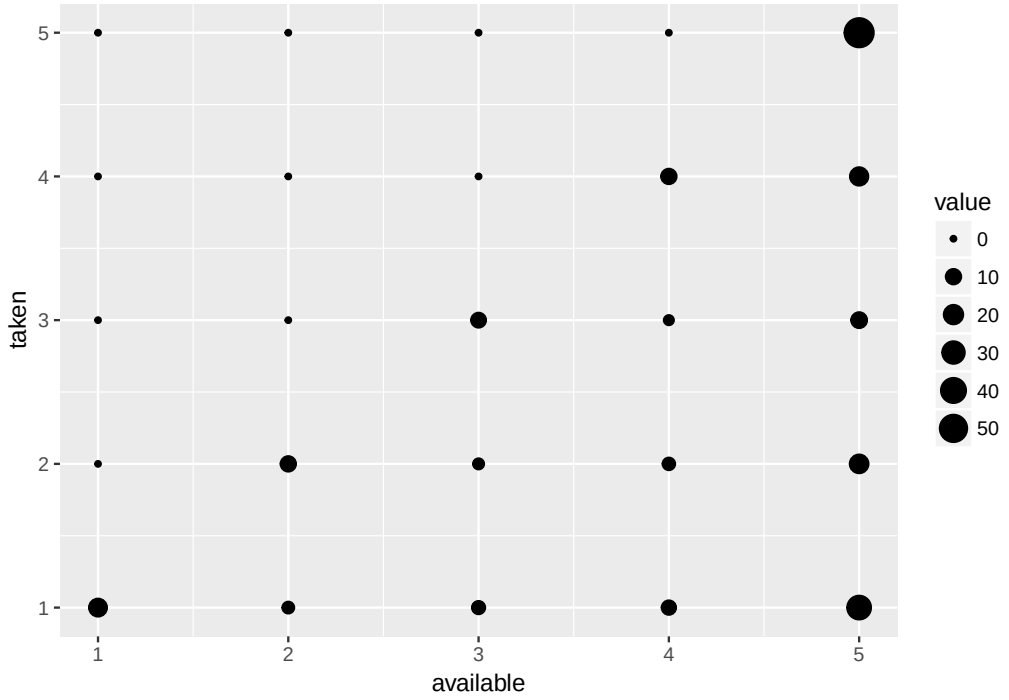
to cross-tabulate the data, and then used the `text()` command to add the numbers to the plot. There's a little bit more trickery involved in putting the numbers in the right place on the plot. `row(x)` gives a matrix with the row numbers corresponding to the elements of x; `col(x)` does the same for column numbers. Subtracting 1 `(col(x)-1)` accounts for the fact that columns 1 through 6 of our table refer to 0 through 5 seeds actually taken. When `R` plots, it simply matches up each of the `x` values, each of the `y` values, and each of the text values (which in this case are the numbers in the table) and plots them, even though the numbers are arranged

in matrices rather than vectors. I also limit the plotting to positive values (using [t1>0]), although this is just cosmetic.

```
r = row(t1)
c = col(t1) - 1
text(r[t1 > 0], c[t1 > 0], t1[t1 > 0])
```

is the final version of the commands.

```
library(ggplot2)
    t2 = melt(t1) # make long format
    colnames(t2) = c("available","taken","value")
    t2 = t2[t2$taken >0,] # remove the zeros from your data
    ggplot(data=t2)+
        geom_point(aes(x = available, y = taken, size = value))+
      ylab("taken")+
      xlab("available")
```

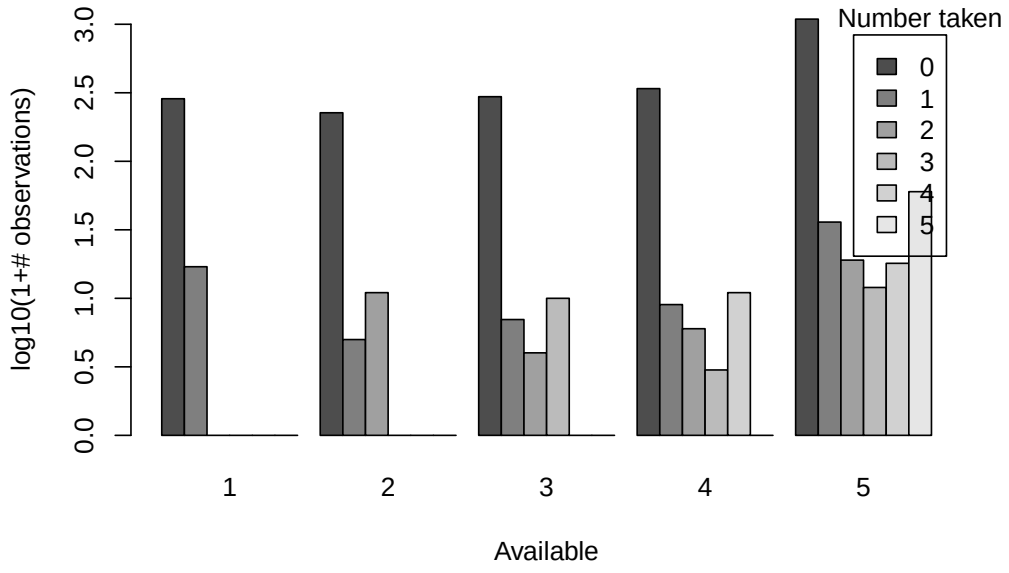



In `ggplot2` you need to specify the dataset the variables come from. You do this throug `data=...`, a specification of type of plot, a point plot can be specified through `geom_point`, a specifcation of what is plotted on the axes of the point plot, e.g `aes(x...,y...)`, and optionally the size of the points through `size`. If data is specified in the `ggplot` statement, it means that all plotting commands

below `ggplot(...)+` use that dataframe as reference. If the size command is put inside the `aes` then size is dependent on some variable, if put outside the `aes` it requires a single value.

The command to produce the barplot (Figure 3) was:

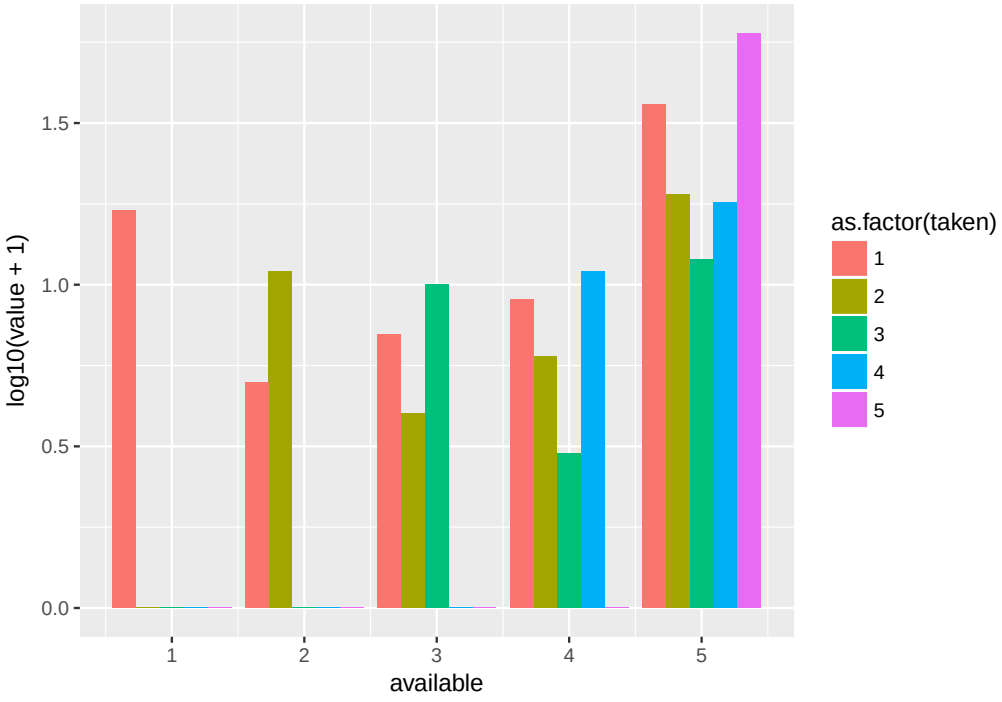
```
barplot(t(log10(t1 + 1)), beside = TRUE, legend = TRUE, xlab = "Available",
 ylab = "log10(1+# observations)")
op = par(xpd = TRUE)
text(34.5, 3.05, "Number taken")
```




```
par(op)
```

Alternatively through `ggplot`

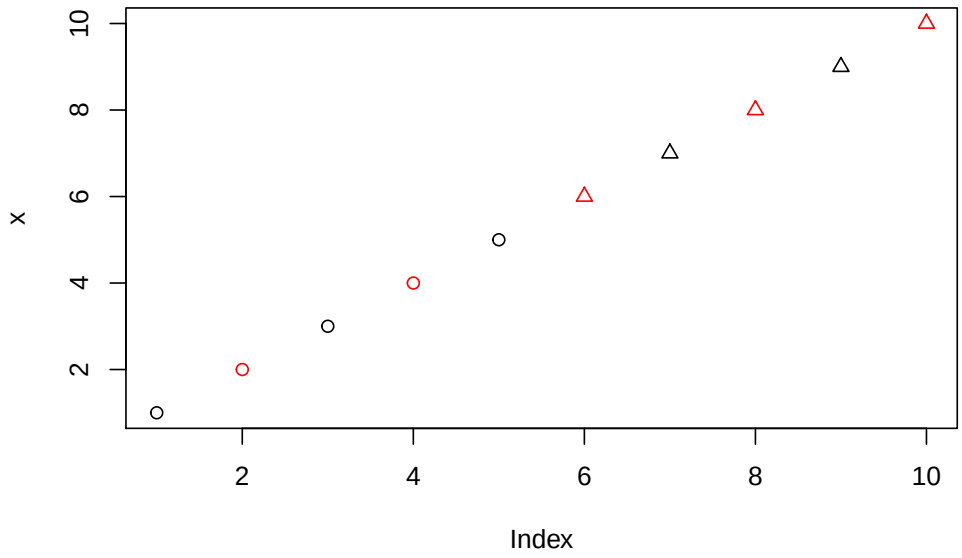
```
ggplot(data=t2)+
  geom_bar(aes(x=available,y=log10(value+1),fill=as.factor(taken)),stat="identity",posi
```




Again through `aes` we specify what is on the `x` and `y`. Through `fill` we subdivide the bars by the values in `taken`. `stat=identity` expresses that the values assigned to `y` will be used (compare `stat="count"`). Through specifying `position_dodge()` bars are printed side by side instead of stacked bars (`position_fill()`).

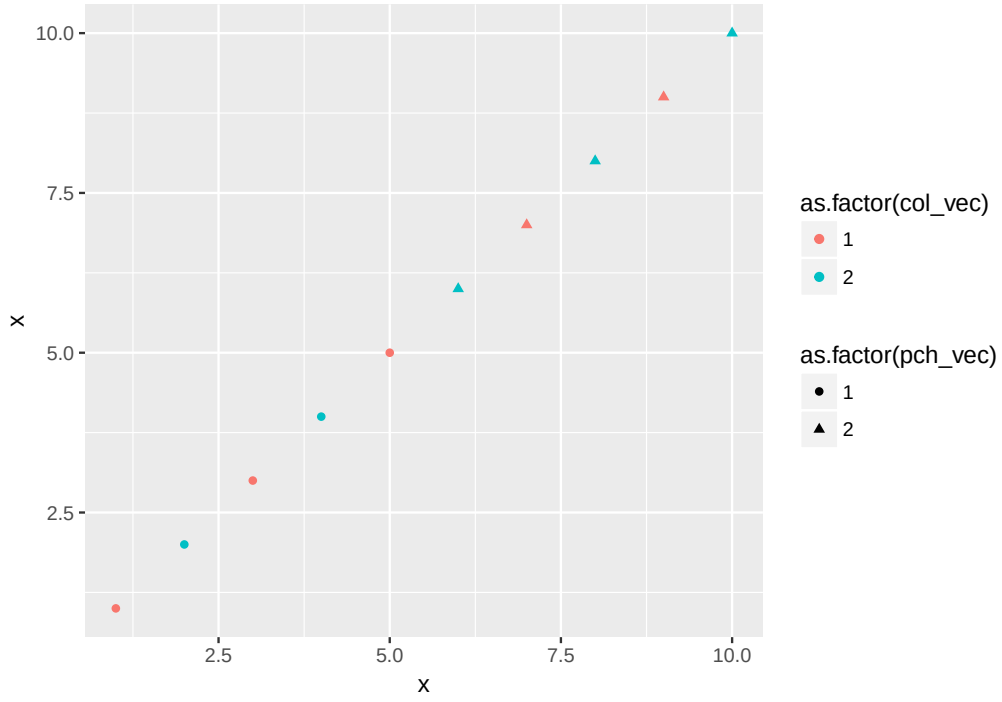
**Exercise 2.1***: In general, you can specify plotting characters and colors in parallel with your data, so that different points get plotted with different plotting characters and colors. For example:

```
x = 1:10
col_vec = rep(1:2, length = 10)
pch_vec = rep(1:2, each = 5)
plot(x, col = col_vec, pch = pch_vec)
```




To use `col_vec` and `pch_vec` in `ggplot2` we need to make them factors

```
ggplot()+
  geom_point(aes(x=x,y=x,shape=as.factor(pch_vec),colour=as.factor(col_vec)))
```



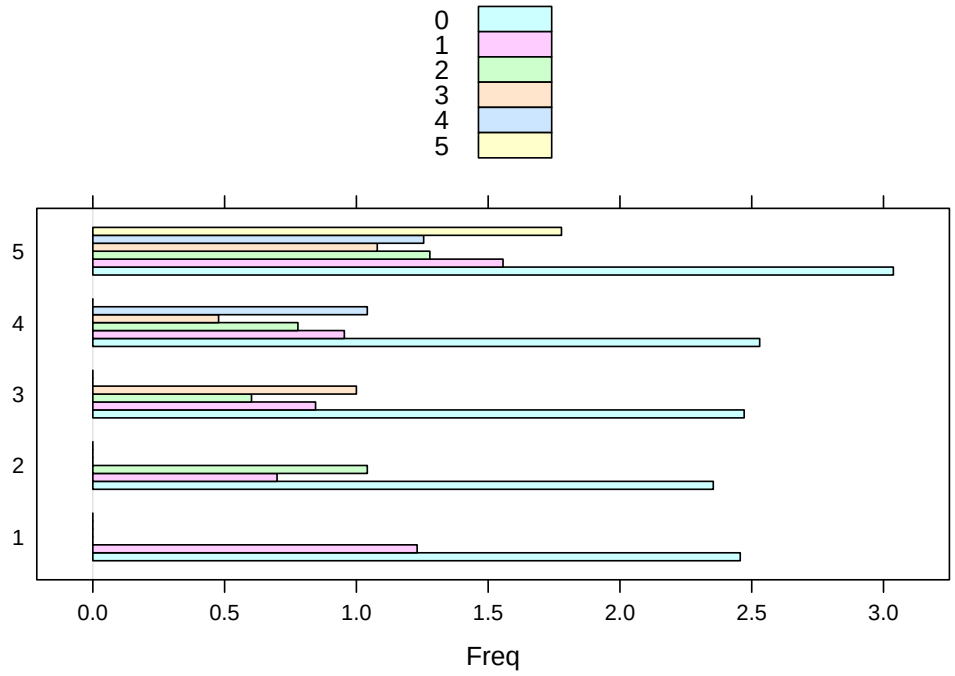

Take the old tabular data (`t1`), `log(1+x)`-transform them, and use `as.numeric()` to drop all the information in tabular form and convert them to a numeric vector. Plot them (plotting the data numeric vector will generate a scatterplot of values on the y-axis vs. observation number on the x-axis), color-coded according to the number available (rows) and point-type-coded according the number taken (columns: note, there is no color 0, so don't subtract 1). `order(x)` is a function that gives a vector of integers that will put `x` in increasing order. For example, if I set `x=c(3,1,2)` then `order(z)` is 2 3 1: putting the second element first, the third element second, and the first element last will put the vector in increasing order. In contrast, `rank(x)` just gives the ranks.

`y[order(x)]` sorts `y` by the elements of `x`.

Redo the plot with the data sorted in increasing order; make sure the colors and point types match the data properly. Does this way of plotting the data show anything the bubbleplot didn't? Can you think of other ways of plotting these data?

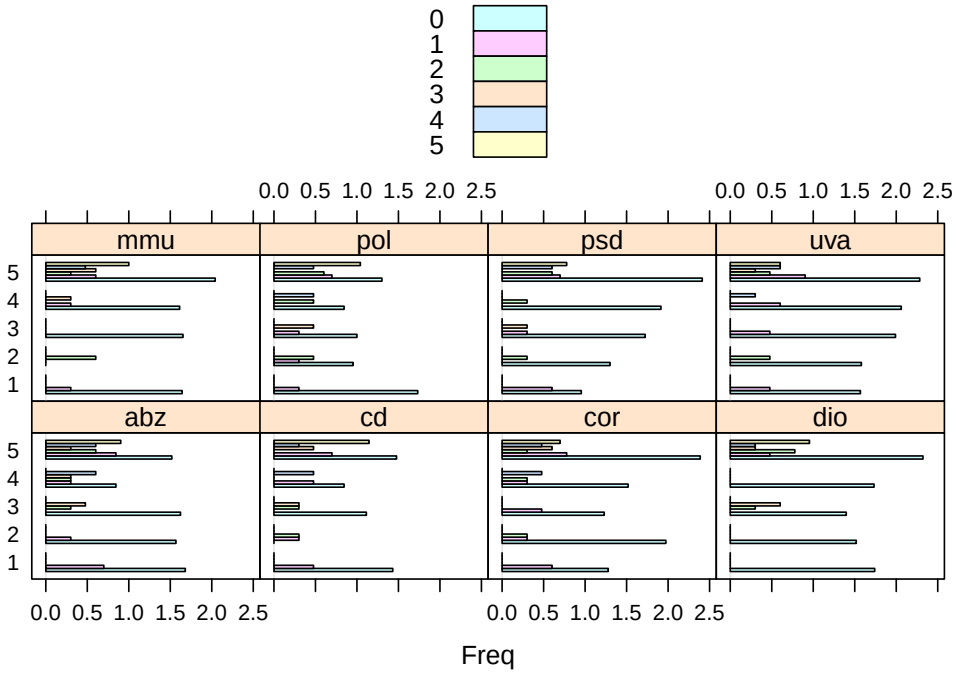
You can use `barchart()` in the `lattice` package to produce these graphics, although it seems impossible to turn the graph so the bars are vertical. Try the following (`stack=FALSE` is equivalent to `beside=TRUE` for `barplot()`):

```
library(lattice)
barchart(log10(1 + table(data$available, data$taken)), stack = FALSE, auto.key = TRUE)
```




More impressively, the lattice package can automatically plot a barplot of a three-way cross-tabulation, in small multiples (I had to experiment a bit to get the factors in the right order in the `table()` command): try

```
barchart(log10(1 + table(data$available, data$species, data$taken)), stack = FALSE, aut
```




**Exercise 2.2***: Restricting your analysis to only the observations with 5 seeds available, create a barplot showing the distribution of number of seeds taken broken down by species. Hints: you can create a new data set that includes only the appropriate rows by using row indexing, then attach() it. Choose whether you do this with geom_bar ggplot2 or through another function.

## 2.3 Barplots with error bars

Computing the fraction taken:

```
data$frac_taken = data$taken/data$available
```

Computing the mean fraction taken for each number of seeds available, using the `tapply()` function: `tapply()` ("table apply", pronounced "t apply"), is an extension of the `table()` function; it splits a specified vector into groups according to the factors provided, then applies a function (e.g. `mean()` or `sd()`) to each group. This idea of applying a function to a set of objects is a very general, very powerful idea in data manipulation with `R`; in due course we'll learn about `apply()` (apply a function to rows and columns of matrices), `lapply()` (apply a function to lists), `sapply()` (apply a function to lists and simplify), and `mapply()` (apply a function to multiple lists). For the present, though,

```
mean_frac_by_avail = tapply(data$frac_taken, data$available, mean)
```

computes the mean of `frac_taken` for each group defined by a different value of available (R automatically converts available into a factor temporarily for this purpose). If you want to compute the mean by group for more than one variable in a data set, use `aggregate()`. We can also calculate the standard errors, $\frac{\sigma}{\sqrt{(n)}}$:

```
n_by_avail = table(data$available)
se_by_avail = tapply(data$frac_taken, data$available, sd)/
              sqrt(n_by_avail)
```

I'll actually use a variant of `barplot()`, `barplot2()` (from the `gplots` package, which you may need to install, along with the the `gtools` and `gdata` packages) to plot these values with standard errors. (I am mildly embarrassed that R does not supply error-bar plotting as a built-in function, but you can use the `barplot2()` in the `gplots` package or the `plotCI()` function (the `gplots` and `plotrix` packages have slightly different versions).

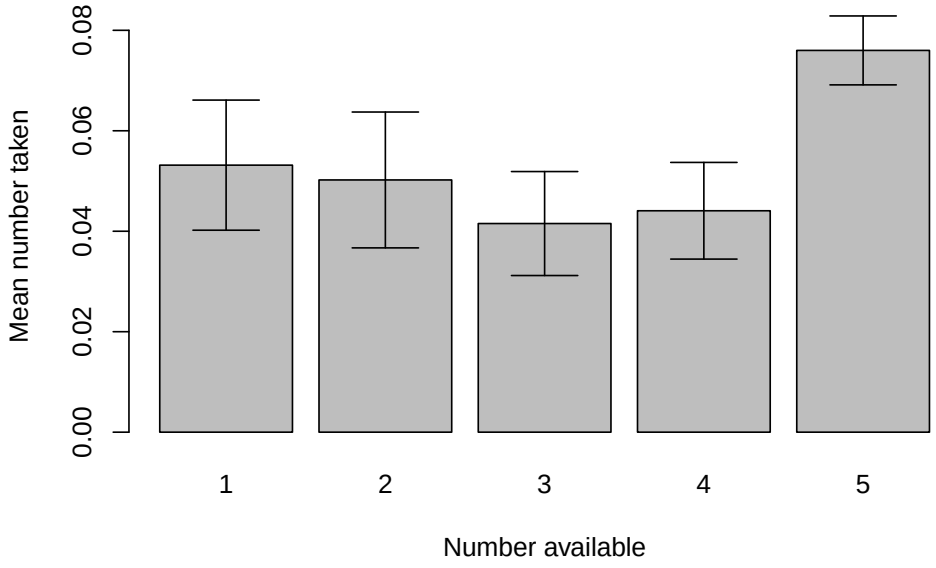
```
library(gplots)
```

```
##
## Attaching package: 'gplots'
```

```
## The following object is masked from 'package:plotrix':
##
##     plotCI
```

```
## The following object is masked from 'package:stats':
##
##     lowess
```

```
lower_lim = mean_frac_by_avail - se_by_avail
upper_lim = mean_frac_by_avail + se_by_avail
b = barplot2(mean_frac_by_avail, plot.ci = TRUE, ci.l = lower_lim,
ci.u = upper_lim, xlab = "Number available", ylab = "Mean number taken")
```



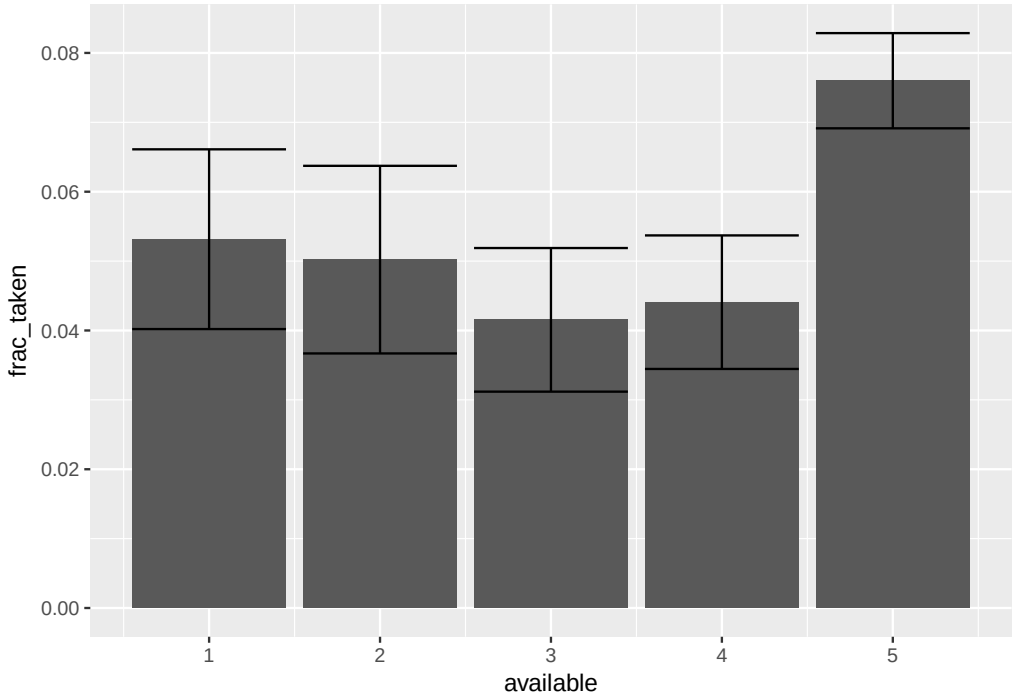

I specified that I wanted error bars plotted (plot.ci=TRUE) and the lower (ci.l) and upper (ci.u) limits.

With `ggplot2` this is possible through the following syntax. Note that there is a convenient function, called `summarySE` in the `Rmisc` package to compute the means and se:

```
library(Rmisc)
```

```
## Loading required package: plyr
```

```
sum.data = summarySE(data,measurevar= "frac_taken",groupvars=c("available"))
pd = position_dodge(0)
ggplot(aes(x=available,y=frac_taken),data=sum.data)+
  geom_bar(,stat="identity")+
  geom_errorbar(aes(ymin=frac_taken-se, ymax=frac_taken+se),
                stat="identity", position=pd)
```

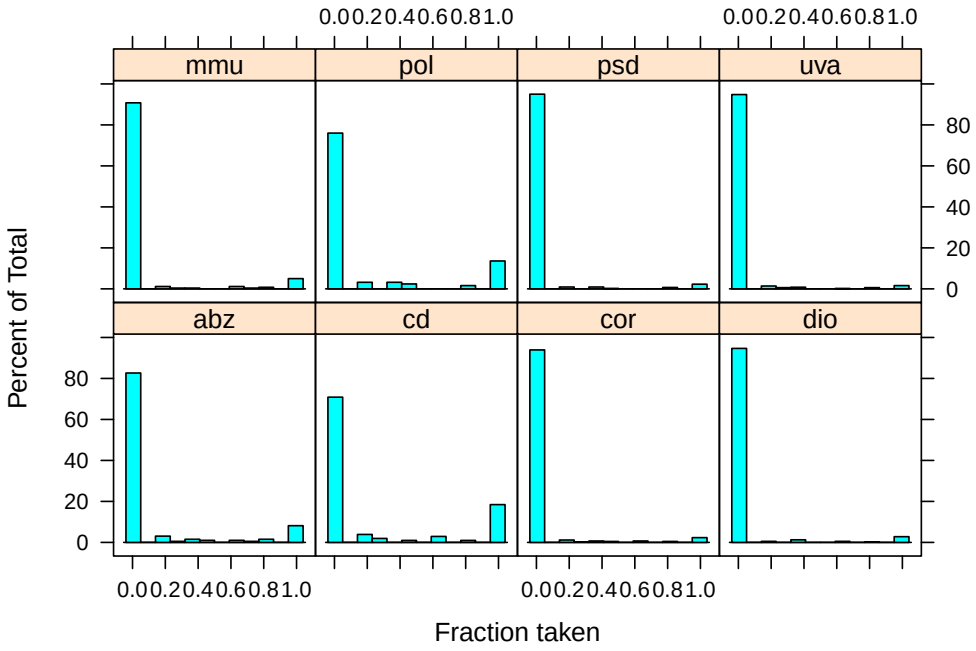



Make sure that the data reference is in the `ggplot()` function so all other functions such as `geom_bar` and `geom_errobar` make use of the same `data.frame`.

## 2.4 Histogram by species

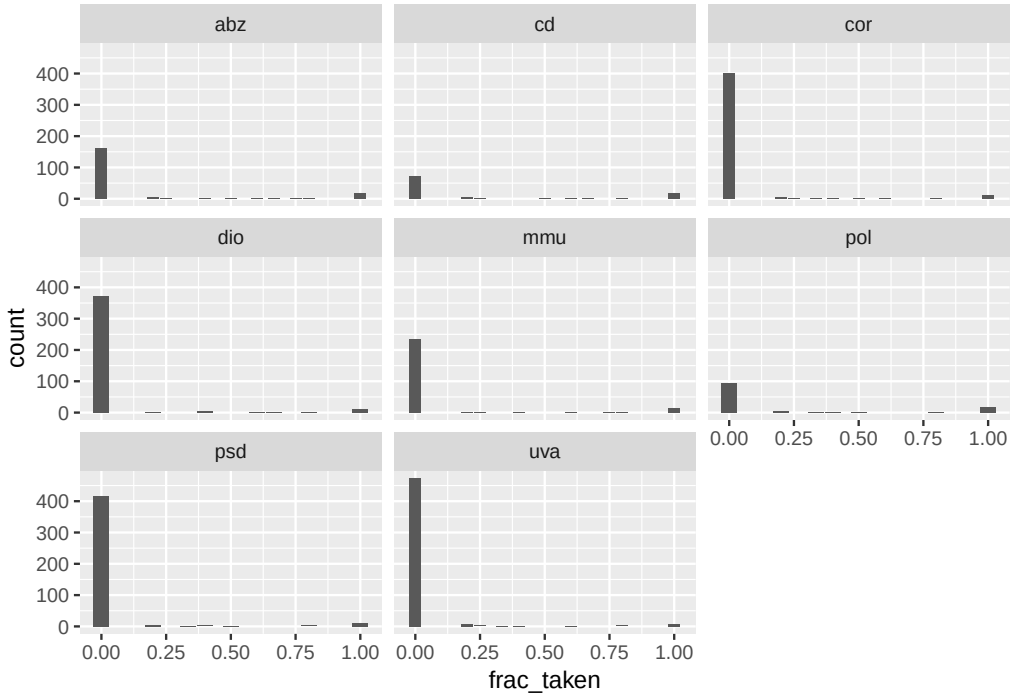
All I had to do to get the lattice package to plot the histogram by species was:

```
histogram(~frac_taken | species, xlab = "Fraction taken",data=data)
```



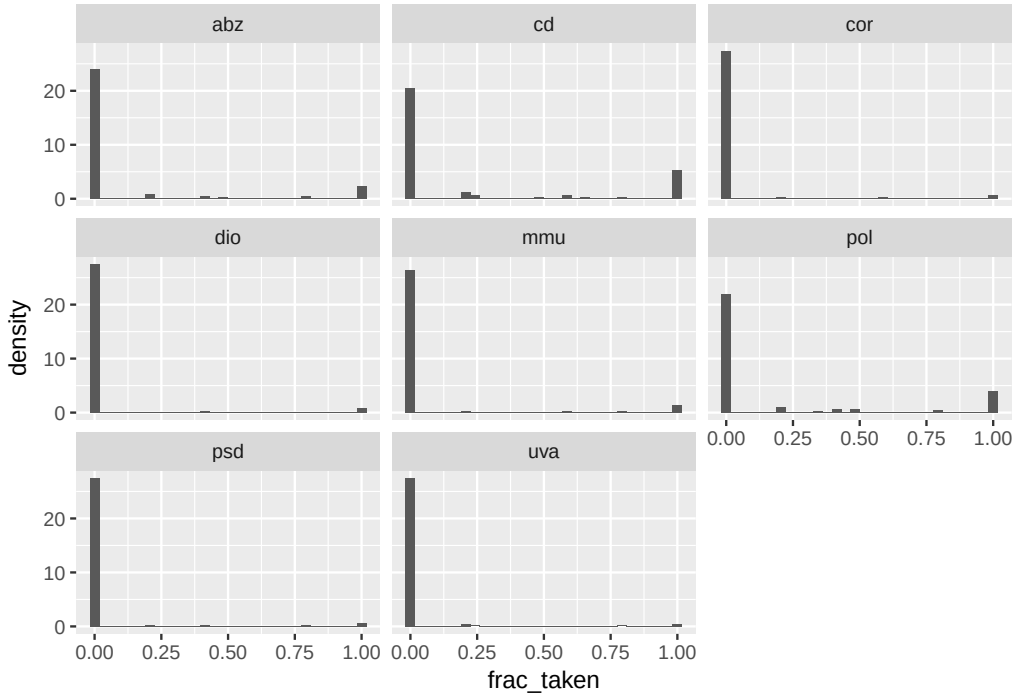

With `ggplot2` you can get the frequencies less easily, so will be plot the counts

```
ggplot(data=data)+
geom_bar(aes(x=frac_taken),stat="count")+
  facet_wrap(~ species)
```




```
ggplot(data=data,aes(x=frac_taken))+
  geom_histogram(aes(y = ..density..))+
  facet_wrap(~species)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




**Time permitting** It's possible to do this with base graphics, too, but you have to rearrange your data yourself: essentially, you have to split the data up by species, tell `R` to break the plotting area up into subplots, and then tell `R` to draw a histogram in each subplot.

- To reorganize the data appropriately and draw the plot, I first use split(), which cuts a vector into a list according to the levels of a factor - in this case giving us a list of the fraction-taken data separated by species:

```
splitdat = split(data$frac_taken, data$species)
```

- Next I use the `par()` command

```
op = par(mfrow = c(3, 3), mar = c(2, 2, 1, 1))
```

to specify a $3 \times 3$ array of mini-plots (`mfrow=c(3,3)`) and to reduce the margin spacing to 2 lines on the bottom and left sides and 1 line on the top and right (`mar=c(2,2,1,1)`).

- Finally, I combine `lapply()`, which applies a command to each of the elements in a list, with the `hist()` (histogram) command. You can specify extra arguments in `lapply()` that will be passed along to the `hist()` function - in this case they're designed to strip out unnecessary detail and make the subplots bigger.

```
h = lapply(splitdat, hist, xlab = "", ylab = "", main = "", col = "gray")
```

Assigning the answer to a variable stops `R` from printing the results, which I don't really want to see in this case.

- `par(op)` will restore the previous graphics parameters.

It's a bit harder to get the species names plotted on the graphs: it is technically possible to use `mapply()` to do this, but then we've reinvented most of the wheels used in the lattice version . . . Plots in this section: `scatterplot` (`plot()` or `xyplot()`) bubble plot (`sizeplot()`), barplot (`barplot()` or `barchart()` or `barplot2()`), histogram (`hist()` or `histogram()`). Data manipulation: `dcast()`,`melt`,`table()`, `split()`, `lapply()`,`sapply()`

## 3. Measles Data

I'm going to clear the workspace (`rm(list=ls())`) lists all the objects in the workspace with `ls()` and then uses `rm()` to remove them. It is recommend to have this code at the top of your script, so every time you start working on the script you are sure the memory is cleared. You can also Clear workspace from the menu (in Rstudio the little brush in the topright panel)) and read in the measles data, which are space separated and have a header:

```
detach(data)
rm(list = ls())
data = read.table("ewcitmeas.dat", header = TRUE, na.strings = "*")
attach(data)
```

`year`, `mon`, and `day` were read in as integers: I'll create a date variable as described above. For convenience, I'm also defining a variable with the city names.

```
date = as.Date(paste(year + 1900, mon, day, sep = "/"))
city_names = colnames(data)[4:10]
```

Later on it will be useful to have the data in long format. It's easiest to do use `melt` for this purpose. Note that only need to select the appropriate columns to melt (i.e. 4-11).

```
library(reshape2)
data= cbind(data,date)
data_long = melt(data[,4:11],id.vars=8,variable.name = "city")
```

### 3.1 Multiple-line plots

If we use the long format and the lattice package we can plot different numeric variables on a common vertical axis.

```
xyplot(incidence ~ date, groups = city, data = data_long, type = "l",
       auto.key = TRUE)
```

To plot each city in its own subplot, use the formula `incidence~date|city` and omit the groups argument. You can also draw any of these plots with different kinds of symbols ("l" for lines, "p" for points (default): see `?plot` for other options). With ggplot2 we specify

```
ggplot() +
  geom_line(aes(x=date,y=incidence, colour=city),data=data_long)
```

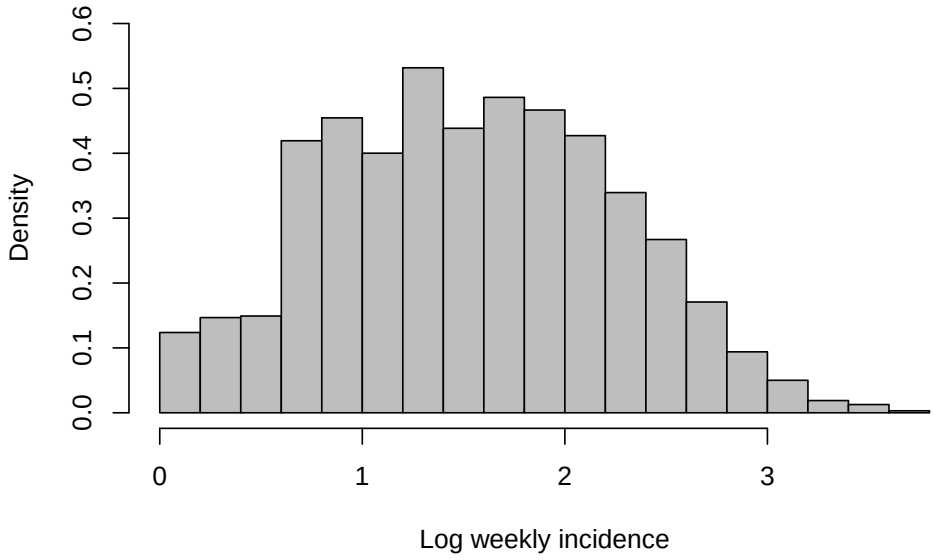
## 3.2 Histogram and density plots

I'll start by just collapsing all the incidence data into a single, logged, non-NA vector (in this case I have to use `c(as.matrix(x))` to collapse the data and remove all of the data frame information):

```
allvals = na.omit(c(as.matrix(data[, 4:10])))
logvals = log10(1 + allvals)
```

The histogram (`hist()` command is fairly easy: the only tricks are to leave room for the other lines that will go on the plot by setting the y limits with ylim, and to specify that we want the data plotted as relative frequencies, not numbers of counts (`freq=FALSE` or `prob=TRUE`). This option tells `R` to divide by total number of counts and then by the bin width, so that the area covered by all the bars adds up to 1; this scaling makes the vertical scale of the histogram compatible with a density plot, or among different histograms with different number of counts or bin widths (?? include in chapter ??).

```
hist(logvals, col = "gray", main = "", xlab = "Log weekly incidence",
 ylab = "Density", freq = FALSE, ylim = c(0, 0.6))
```



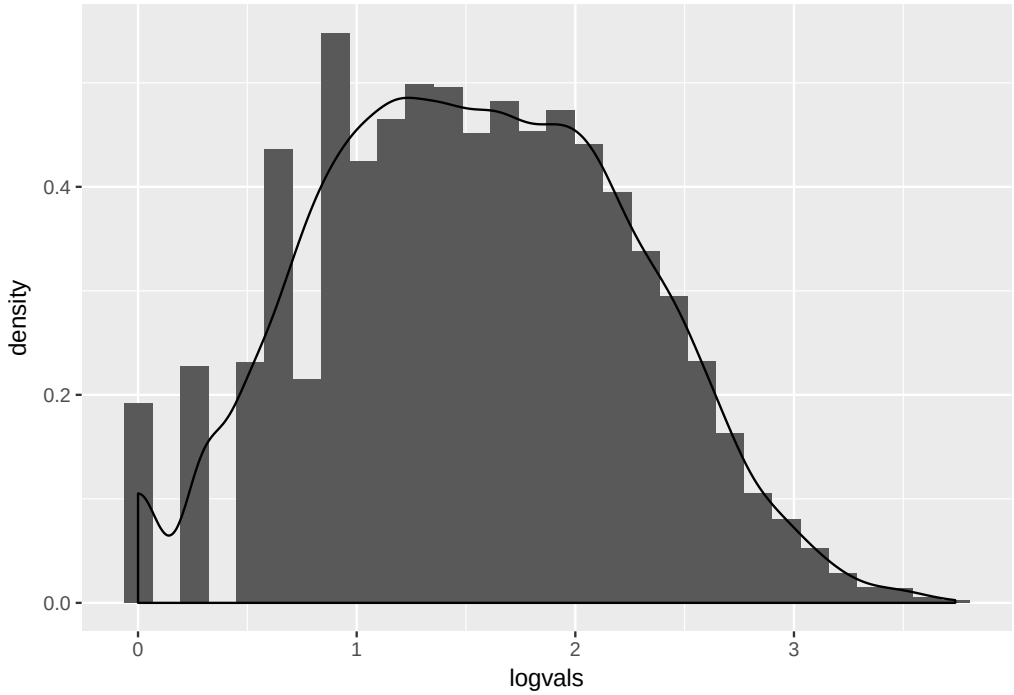

Adding lines for the density is straightforward, since R knows what to do with a density object - in general, the lines command just adds lines to a plot.

```
lines(density(logvals), lwd = 2)
lines(density(logvals, adjust = 0.5), lwd = 2, lty = 2)
```

With ggplot2 we specify:

```
ggplot()+
  geom_histogram(aes(x=logvals,y=..density..))+
  geom_density(aes(x=logvals,y=..density..))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




#3.3 Scaling data Scaling the incidence in each city by the population size, or by the mean or maximum incidence in that city, begins to get us into some non-trivial data manipulation. This process may actually be easier in the wide format. Several useful commands: * `rowMeans()`, `rowSums()`, `colMeans()`, and `colSums()` will compute the means or sums of columns efficiently. In this case we would do something like `colMeans(data[,4:10])` to get the mean incidence for each city. * `apply()` is the more general command for running some command on each of a set of rows or columns. When you look at the help for apply() you'll see an argument called MARGIN, which specifies whether you want to operate on rows (1) or columns (2). For example, apply(data[,4:10],1,mean) is the equivalent of rowMeans(data[,4:10]), but we can also easily say (e.g.) apply(data[,4:10],1,max) to get the maxima instead. Later, when you've gotten practice defining your own functions, you can apply any function - not just R's built-in functions. * `scale()` is a function for subtracting and dividing specified amounts out of the columns of a matrix. It is fairly flexible: scale(x,center=TRUE,scale=TRUE) will center by subtracting the means and then scale by dividing by the standard errors of the columns. Fairly obviously, setting either to FALSE will turn off that part of the operation. You can also specify a vector for either center or scale, in which case scale() will subtract or divide the columns by those vectors instead.

**Exercise 3.1***: figure out how to use apply() and scale() to scale all columns so they have a minimum of 0 and a maximum of 1 (hint: subtract the minimum and*

*divide by (max-min)).* `sweep()` is more general than scale; it will operate on either rows or columns (depending on the MARGIN argument), and it will use any operator (typically "-", "/", etc. - arithmetic symbols must be in quotes) rather than just subtracting or dividing. For example, `sweep(x,1,rowSums(x),"/")` will divide the rows (1) of x by their sums. **Note Bob: I never use this command**

**Exercise 3.2***: figure out how to use a call to sweep() to do the same thing as scale(x,center=TRUE,scale=FALSE). **Note Bob: I never use this command**

So, if you are using the short format, and I want to divide each city's incidence by its mean (allowing for adding 1) and take logs:

```
logscaledat = as.data.frame(
    log10(
      scale(1 + data[, 4:10], center = FALSE, scale = colMeans(1 + data[, 4:10], na.rm
```

The easier approach is when you are using the long format, but you have to think about it differently. Use `tapply()` to compute the mean incidence in each city, ignoring NA values, and adding 1 to all values:

```
city_means <- tapply((1 + data_long$incidence), data_long$city, mean, na.rm = TRUE)
```

Now you can use vector indexing to scale each incidence value by the appropriate mean value - `city_means[data_long$city]` does the trick. (Why?)

```
city_means[data_long$city]

scdat <- (1 + data_long$incidence)/city_means[data_long$city]
```

You can add this column to the long datamframe by

```
data_long$scdat = scdat
```

**Exercise 3.3** *: figure out how to scale the long-format data to minima of zero and maxima of 1.

## 3.4 Box-and-whisker and violin plots

By this time, box-and-whisker and violin plots will (I hope) seem easy: Since the labels get a little crowded (`R` is not really sophisticated about dealing with axis labels-crowded labels just disappear-although you can try the `stagger.labs()` command from the `plotrix` package), I'll use the `substr()` (substring) command to abbreviate each city's name to its first three letters.

```
city_abbr = substr(city_names, 1, 3)
```

The `boxplot()` command uses a formula - the variable before the ~ is the data and the variable after it is the factor to use to split the data up.

```
boxplot(log10(1 + incidence) ~ city, data = data_long, ylab = "Log(incidence+1)",
 names = city_abbr)
```

Or through ggplot

```
ggplot(data=data_long)+
  geom_boxplot((aes(x=city,y=log10(incidence+1))))
```

If I want to make a violin plot, you can specify:

```
ggplot(data=data_long)+
  geom_violin((aes(x=city,y=log10(incidence+1))))
```

Plots in this section: multiple-groups plot (`matplot()` or `xyplot(...,groups)`, box-and-whisker plot (`boxplot()` or `bwplot()`), density plot (`plot(density())` or `lines(density())` or `densityplot()`), `geom_boxplot()`, `geom_violin()` Data manipulation: `rowMeans()`/`colMeans()`, `rowSums()`/`colSums()`, `sweep()`, `scale()`, `apply()`, `mapply()`

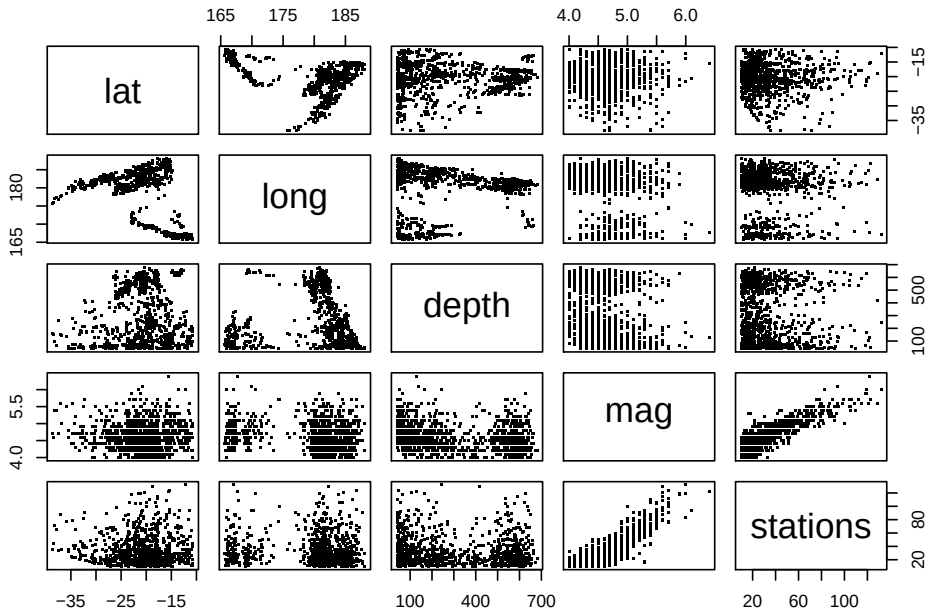
## 4. Continuous data

First let's make sure the earthquake data are accessible:

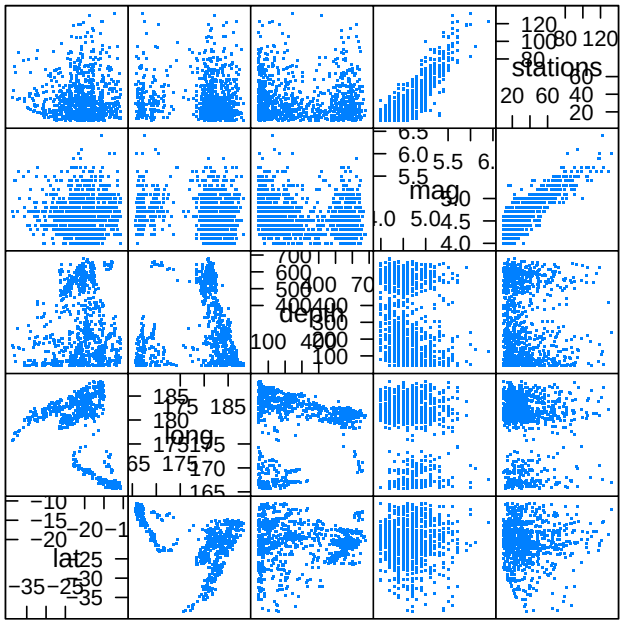
```
data(quakes)
```

Luckily, most of the plots I drew in this section are fairly automatic. To draw a scatterplot matrix, just use `pairs()` (base) or `splom()` (lattice):

```
pairs(quakes, pch = ".")
```




```
splom(quakes, pch = ".")
```




Scatter Plot Matrix

(`pch="."` marks the data with a single-pixel point, which is handy if you are fortunate enough to have a really big data set).

**Exercise 4.1***: generate three new plots based on one of the data sets in this lab, or on your own data.