Assignment 9

Due Thursday November 24 at 11:59pm on Blackboard

1. Male tree crickets produce "mating songs" by rubbing their wings together to produce a chirping sound. It is hypothesized that female tree crickets identify males of the correct species by how fast (in chirps per second) the male's mating song is. This is called the "pulse rate". Some data for two species of crickets are in `http://www.utsc.utoronto.ca/~butler/c32/crickets.txt`. The columns, which are unlabelled, are temperature and pulse rate (respectively) for *Oecanthus exclamationis* (first two columns) and *Oecanthus niveus* (third and fourth columns). The columns are separated by tabs. There are some missing values in the first two columns because fewer *exclamationis* crickets than *niveus* crickets were measured.

   The research question is whether males of the different species have different average pulse rates. It is also of interest to see whether temperature has an effect, and if so, what.

   (a) Read in the data, allowing for the fact that you have no column names. You'll see that the columns have names `X1` through `X4`. This is OK.

   Solution: Tab-separated, so `read_tsv`; no column names, so `col_names=F`:

```
my_url="http://www.utsc.utoronto.ca/~butler/c32/crickets.txt"
crickets=read_tsv(my_url,col_names=F)
```

```
## Error in read_tsv(my_url, col_names = F): could not find function "read_tsv"
```

```
crickets
```

```
## Error in eval(expr, envir, enclos):  object 'crickets' not found
```

   As promised.

   If you didn't catch the tab-separated part, this probably happened to you:

```
d=read_delim(my_url," ",col_names=F)
```

```
## Error in read_delim(my_url, " ", col_names = F): could not find function "read_delim"
```

   This doesn't look good:

```
problems(d)
```

```
## Error in problems(d):  could not find function "problems"
```

   The "expected columns" being 1 should bother you, since we know there are supposed to be 4 columns. At this point, we take a look at what got read in:

```
d
```

```
## Error in eval(expr, envir, enclos):  object 'd' not found
```

and there you see the `\t` or "tab" characters separating the values, instead of spaces. (This is what I tried first, and once I looked at this, I realized that `read_tsv` was what I needed.)

(b) These data are rather far from being tidy. There need to be three variables, temperature, pulse rate and species, and there are $14 + 17 = 31$ observations altogether. This one is tricky in that there are temperature and pulse rate for each of two levels of a factor, so I'll suggest combining the temperature and chirp rate together into one thing for each species, then gathering them, then splitting them again. Create new columns, named for each species, that contain the temperature and pulse rate for that species in that order, `unite`d together.

For the rest of this question, start from the data frame you read in, and build a pipe, one or two steps at a time, to save creating a lot of temporary data frames.

Solution: Breathe, and then begin. `unite` creates new columns by joining together old ones:[1]

```
crickets %>%
  unite(exclamationis,X1:X2) %>%
  unite(niveus,X3:X4)
```

```
## Error in crickets %>% unite(exclamationis, X1:X2) %>% unite(niveus, X3:X4):  could
not find function "%>%"
```

Note that the original columns `X1:X4` are *gone*, which is fine, because the information we needed from them is contained in the two new columns. `unite` by default uses an underscore to separate the joined-together values, which is generally safe since you won't often find those in data.

Digression: `unite`-ing with a space could cause problems if the data values have spaces in them already. Consider this list of names:

```
names=c("Cameron McDonald","Durwin Yang","Ole Gunnar Solskjaer","Mahmudullah")
```

Two very former students of mine, a Norwegian soccer player, and a Bangladeshi cricketer. Only one of these has played for Manchester United:

```
manu=c(F,F,T,F)
```

and let's make a data frame:

```
d=tibble(name=names,manu=manu)
```

```
## Error in tibble(name = names, manu = manu):  could not find function "tibble"
```

```
d
```

```
## Error in eval(expr, envir, enclos):  object 'd' not found
```

Now, what happens if we `unite` those columns, separating them by a space?

```
d %>% unite(joined,name:manu,sep=" ")
```

```
## Error in d %>% unite(joined, name:manu, sep = " "):  could not find function
"%>%"
```

If we then try to separate them again, what happens?

```
d %>% unite(joined,name:manu,sep=" ") %>%
  separate(joined,c("one","two")," ")
```

```
## Error in d %>% unite(joined, name:manu, sep = " ") %>% separate(joined, :  could
not find function "%>%"
```

Things have gotten lost: most of the original values of `manu` and some of the names. If we use a different separator character, either choosing one deliberately or going with the default underscore, everything works swimmingly:

```
d %>% unite(joined,name:manu,sep=":") %>%
  separate(joined,c("one","two"),":")
```

```
## Error in d %>% unite(joined, name:manu, sep = ":") %>% separate(joined, :  could
not find function "%>%"
```

and we are back to where we started.

If you run just the `unite` line (move the pipe symbol to the next line so that the `unite` line is complete as it stands), you'll see what happened.

(c) The two columns `exclamationis` and `niveus` that you just created are both temperature-pulse rate combos, but for different species. `gather` them together into one column, labelled by species. (This is a straight `tidyr gather`, even though they contain something odd-looking.)

Solution: Thus, this, naming the new column `temp_pulse` since it contains both of those things. Add to the end of the pipe you started building in the previous part:

```
crickets %>%
  unite(exclamationis,X1:X2) %>%
  unite(niveus,X3:X4) %>%
  gather(species,temp_pulse,exclamationis:niveus)
```

```
## Error in crickets %>% unite(exclamationis, X1:X2) %>% unite(niveus, X3:X4) %>%
:  could not find function "%>%"
```

Yep. If you scroll down with Next, you'll see the other species of crickets, and you'll see some missing values at the bottom, labelled, at the moment, `NA_NA`.

This is going to get rather long, but don't fret: we debugged the two `unite` lines before, so if you get any errors, they must have come from the `gather`. So that would be the place to check.

(d) Now split up the temperature-pulse combos at the underscore, into two separate columns. This is `separate`. When specifying what to separate by, you can use a number ("split after this many characters") or a piece of text, in quotes ("when you see this text, split at it").

Solution: The text to split by is an underscore (in quotes), since `unite` by default puts an underscore in between the values it pastes together. Glue the `separate` onto the end. We are creating two new variables `temperature` and `pulse_rate`:

```
crickets %>%
  unite(exclamationis,X1:X2) %>%
  unite(niveus,X3:X4) %>%
  gather(species,temp_pulse,exclamationis:niveus) %>%
  separate(temp_pulse,c("temperature","pulse_rate"),"_")
```

```
## Error in crickets %>% unite(exclamationis, X1:X2) %>% unite(niveus, X3:X4) %>%
:  could not find function "%>%"
```

You'll note that `unite` and `separate` are opposites ("inverses") of each other, but we haven't just done something and then undone it, because we have a `gather` in between; in fact, arranging it this way has done precisely the tidying we wanted.

(e) Almost there. Temperature and pulse rate are still text (because `unite` turned them into text), but they should be numbers. Create new variables that are numerical versions of temperature and pulse rate (using `as.numeric`). Check that you have no extraneous variables (and, if necessary, get rid of the ones you don't want). (Species is also text and really ought to be a factor, but having it as text doesn't seem to cause any problems.)

You can, if you like, use `parse_number` instead of `as.numeric`. They should both work. The distinction I prefer to make is that `parse_number` is good for text with a number in it (that we want to pull the number out of), while `as.numeric` is for turning something that looks like a number but isn't one into a genuine number.[2]

Solution: `mutate`-ing into a column that already exists overwrites the variable that's already there (which saves us some effort here).

```
crickets.1 = crickets %>%
  unite(exclamationis,X1:X2) %>%
  unite(niveus,X3:X4) %>%
  gather(species,temp_pulse,exclamationis:niveus) %>%
  separate(temp_pulse,c("temperature","pulse_rate"),"_") %>%
  mutate(temperature=as.numeric(temperature)) %>%
  mutate(pulse_rate=as.numeric(pulse_rate))
```

```
## Error in crickets %>% unite(exclamationis, X1:X2) %>% unite(niveus, X3:X4) %>%
:  could not find function "%>%"
```

```
crickets.1
```

```
## Error in eval(expr, envir, enclos):  object 'crickets.1' not found
```

I saved the data frame this time, since this is the one we will use for our analysis.

The warning message tells us that we got genuine missing-value NAs back, which is probably what we want. Specifically, they got turned from missing *text* to missing *numbers*![3] The R word "coercion" means values being changed from one type of thing to another type of thing. (We'll ignore the missings and see if they cause us any trouble. The same warning messages will show up on graphs later.) So I have 34 rows (including three rows of missings) instead of the 31 rows I would have liked. Otherwise, success.

There is (inevitably) another way to do this. We are doing the `as.numeric` twice, exactly the same on two different columns, and when you are doing the same thing on a number of columns, here a `mutate` with the same function, you have the option of using `mutate_if` or `mutate_at`.

These are like `summarize_if` and `summarize_at` that we used way back to compute numerical summaries of a bunch of columns: the `if` variant works on columns that share a property, like being numeric, and the `at` variant works on columns whose names have something in common or that we can list, which is what we want here:

```
crickets %>%
  unite(exclamationis,X1:X2) %>%
  unite(niveus,X3:X4) %>%
  gather(species,temp_pulse,exclamationis:niveus) %>%
  separate(temp_pulse,c("temperature","pulse_rate"),"_") %>%
  mutate_at(vars(temperature:pulse_rate),funs(as.numeric))
```

```
## Error in crickets %>% unite(exclamationis, X1:X2) %>% unite(niveus, X3:X4) %>%
:  could not find function "%>%"
```

Can't I just say that these are columns 2 and 3?

```
crickets %>%
  unite(exclamationis,X1:X2) %>%
  unite(niveus,X3:X4) %>%
  gather(species,temp_pulse,exclamationis:niveus) %>%
  separate(temp_pulse,c("temperature","pulse_rate"),"_") %>%
  mutate_at(vars(2:3),funs(as.numeric))
```

```
## Error in crickets %>% unite(exclamationis, X1:X2) %>% unite(niveus, X3:X4) %>%
:  could not find function "%>%"
```

Yes. Equally good. What goes into the `vars` is the same as can go into a `select`: column numbers, names, or any of those "select helpers" like `starts_with`.

You might think of `mutate_if` here, but if you scroll back, you'll find that all the columns are text, before you convert temperature and pulse rate to numbers, and so there's no way to pick out just the two columns you want that way.

Check that the temperature and pulse rate columns are now labelled `dbl`, which means they actually *are* decimal numbers (and don't just look like decimal numbers).

Either way, using `unite` and then `separate` means that all the columns we created we want to keep (or, all the ones we would have wanted to get rid of have already been gotten rid of).

Now we can actually do some statistics.

(f) Do a two-sample t-test to see whether the mean pulse rates differ between species. What do you conclude?

Solution: Drag your mind way back to this:

```
t.test(pulse_rate~species,data=crickets.1)
```

```
## Error in eval(m$data, parent.frame()):  object 'crickets.1' not found
```

There is strong evidence of a difference in means (a P-value around 0.00001), and the confidence interval says that the mean chirp rate is higher for *exclamationis*. That is, not just for the crickets that were observed here, but for *all* crickets of these two species.

(g) The analysis in the last part did not use temperature, however. Is it possible that temperature also

has an effect? To assess this, draw a scatterplot of pulse rate against temperature, with the points distinguished, somehow, by the species they are from.[4]

Solution: One of the wonderful things about `ggplot` is that doing the obvious thing works:

```
ggplot(crickets.1,aes(x=temperature,y=pulse_rate,colour=species))+
  geom_point()
```

```
## Error in ggplot(crickets.1, aes(x = temperature, y = pulse_rate, colour = species)):
could not find function "ggplot"
```

(h) What does the plot tell you that the t-test doesn't? How would you describe differences in pulse rates between species now?

Solution: The plot tells you that (for both species) as temperature goes up, pulse rate goes up as well. *Allowing for that*, the difference in pulse rates between the two species is even clearer than it was before. To see an example, pick a temperature, and note that the mean pulse rate at that temperature seems to be at least 10 higher for *exclamationis*, with a high degree of consistency.

The t-test mixed up all the pulse rates at all the different temperatures. Even though the conclusion was clear enough, it could be clearer if we incorporated temperature into the analysis.

There was also a potential source of unfairness in that the *exclamationis* crickets tended to be observed at higher temperatures than *niveus* crickets; since pulse rates increase with temperature, the apparent difference in pulse rates between the species might have been explainable by one species being observed mainly in higher temperatures. This was *utterly invisible* to us when we did the t-test, but it shows the importance of accounting for all the relevant variables when you do your analysis.[5] If the species had been observed at opposite temperatures, we might have concluded[6] that *niveus* have the higher pulse rates on average. I come back to this later when I discuss the confidence interval for species difference that comes out of the regression model with temperature.

(i) Fit a regression predicting pulse rate from species and temperature. Compare the P-value for species in this regression to the one from the t-test. What does that tell you?

Solution: This is actually a so-called "analysis of covariance model", which properly belongs in D29, but it's really just a regression:

```
pulse.1=lm(pulse_rate~species+temperature,data=crickets.1)
```

```
## Error in is.data.frame(data):  object 'crickets.1' not found
```

```
summary(pulse.1)
```

```
## Error in summary(pulse.1):  object 'pulse.1' not found
```

The P-value for species is now 6.27×10^{-14} or 0.00000000000006, which is even less than the P-value of 0.00001 that came out of the t-test. That is to say, when you know temperature, you can be even more sure of your conclusion that there is a difference between the species.

The R-squared for this regression is almost 99%, which says that if you know both temperature and species, you can predict the pulse rate almost exactly.

In the regression output, the slope for species is about -10. It is labelled `speciesniveus`. Since

species is categorical, `lm` uses the first category, *exclamationis*, as the baseline and expresses each other species relative to that. Since the slope is about -10, it says that at any given temperature, the mean pulse rate for *niveus* is about 10 less than for *exclamationis*. This is pretty much what the scatterplot told us.

We can go a little further here:

```
confint(pulse.1)
```

```
## Error in confint(pulse.1):  object 'pulse.1' not found
```

The second line says that the pulse rate for *niveus* is between about 8.5 and 11.5 less than for *exclamationis*, at any given temperature (comparing the two species at the same temperature as each other, but that temperature could be anything). This is a lot shorter than the CI that came out of the t-test, that went from 14 to 32. This is because we are now accounting for temperature, which also makes a difference. (In the t-test, the temperatures were all mixed up). What we also see is that the t-interval is shifted up compared to the one from the regression. This is because the t-interval conflates[7] two things: the *exclamationis* crickets do have a higher pulse rate, but they were also observed at higher temperatures, which makes it look as if their pulse rates are more higher[8] than they really are, when you account for temperature.

This particular model constrains the slope with temperature to be the same for both species (just the intercepts differ). If you want to allow the slopes to differ between species, you add an interaction between temperature and species:

```
pulse.2=lm(pulse_rate~species*temperature,data=crickets.1)
```

```
## Error in is.data.frame(data):  object 'crickets.1' not found
```

```
summary(pulse.2)
```

```
## Error in summary(pulse.2):  object 'pulse.2' not found
```

To see whether adding the interaction term added anything to the prediction,[9] compare the model with and without using `anova`:

```
anova(pulse.1,pulse.2)
```

```
## Error in anova(pulse.1, pulse.2):  object 'pulse.1' not found
```

There's no significant improvement by adding the interaction, so there's no evidence that having different slopes for each species is necessary. Note that `anova` gave the same P-value as did the t-test for the slope coefficient for the interaction in `summary`, 0.254 in both cases. This is because there were only two species and therefore only one slope coefficient was required to distinguish them. If there had been three species, we would have had to look at the `anova` output to hunt for a difference among species, since there would have been two slope coefficients, each with its own P-value.[10]

The upshot is that we do not need different slopes; the model `pulse.1` with the same slope for each species describes what is going on.

`ggplot` makes it almost laughably easy to add regression lines for each species to our plot, thus:

```
ggplot(crickets.1,aes(x=temperature,y=pulse_rate,colour=species))+
  geom_point()+geom_smooth(method="lm",se=F)
```

```
## Error in ggplot(crickets.1, aes(x = temperature, y = pulse_rate, colour = species)):
could not find function "ggplot"
```

The lines are almost exactly parallel, so having the same slope for each species makes perfect sense.

2. In Denali National Park, Alaska, the size of the wolf population depends on the size of the caribou population (since wolves hunt and kill caribou). This is a large national park, so caribou are found in very large herds, so big, in fact, that the well-being of the entire herd is not threatened by wolf attacks.[11] Can the size of the caribou population be used to predict the size of the wolf population?

   The data can be found at `http://www.utsc.utoronto.ca/~butler/c32/caribou.txt`. The columns are: the date of the survey,[12] the name of the park employee in charge of the survey, the caribou population (in hundreds) and the wolf population (actual count).[13]

   We are going to use SAS for this question, but this format of data file is one we don't know how to read into SAS, so we are going to use R to help us first.

   (a) Take a look at the data file. How would you describe its format? Read it into R, and check that you got something sensible.

   Solution: This looks at first sight as if it's separated by spaces, but most of the data values are separated by *more than one* space. If you look further, you'll see that the values are *lined up in columns*, with the variable names aligned at the top. This is exactly the kind of thing that `read_table` will read. We start with the usual `library(tidyverse)`:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.0.0     v purrr   0.2.5
## v tibble  1.4.2     v dplyr   0.7.6
## v tidyr   0.8.1     v stringr 1.3.1
## v readr   1.1.1     v forcats 0.3.0
```

```
## -- Conflicts ------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
my_url="http://www.utsc.utoronto.ca/~butler/c32/caribou.txt"
denali=read_table(my_url)
```

```
## Parsed with column specification:
## cols(
##   date = col_character(),
##   name = col_character(),
##   caribou = col_integer(),
##   wolf = col_integer()
## )
```

```
denali
```

```
## # A tibble: 7 x 4
##   date       name          caribou  wolf
##   <chr>      <chr>           <int> <int>
## 1 09/01/1995 David S.           30    66
## 2 09/24/1996 Youngjin K.        34    79
## 3 10/03/1997 Srinivasan M.      27    70
## 4 09/15/1998 Lee Anne J.        25    60
## 5 09/08/1999 Stephanie T.       17    48
## 6 09/03/2000 Angus Mc D.        23    55
## 7 10/06/2001 David S.           20    60
```

That worked: four columns with the right names, and the counts of caribou and wolf are numbers. The only (small) weirdness is that the dates are text rather than having been converted into dates. This is because they are not year-month-day, which is the only format that gets automatically converted into dates when read in. (You could use `mdy` from `lubridate` to make them dates.)

(b) Save your R data frame as a `.csv` file. This goes using `write_csv`, which is the exact opposite of `read_csv`. It takes two things: a data frame to save as a `.csv`, and the name of a file to save it in.

Solution: You probably haven't seen this before, so I hope I gave you some clues:

```
write_csv(denali,"denali.csv")
```

This saves the data frame as a `.csv` into your project folder on `rstudio.cloud`. You can go to the Files pane and click on it to open it. Select View File: this will open the actual file in a new tab, so you can see what it looks like, namely this:

```
date,name,caribou,wolf
```

```
09/01/1995,David S.,30,66
09/24/1996,Youngjin K.,34,79
10/03/1997,Srinivasan M.,27,70
09/15/1998,Lee Anne J.,25,60
09/08/1999,Stephanie T.,17,48
09/03/2000,Angus Mc D.,23,55
10/06/2001,David S.,20,60
```

The columns don't line up any more, but there are no extra spaces, so that reading this into SAS as a `.csv` should go smoothly.

Download the file from `rstudio.cloud` to your computer. To do this, go back to the Files pane, and click the checkbox to the left of `denali.csv` (or whatever you called it). Then click on More (above the Files pane) and Export, then click Download. It will go to your Downloads folder, or wherever downloaded things go.

If you are running R Studio on your computer, it will be in the folder associated with the project you're in now, and you can find it there.

Then upload the `.csv` file to SAS Studio.

(c) Read your `.csv` file into SAS, and list the values.

Solution: This is the usual `proc import`. First, though, make sure you have uploaded the `.csv` from wherever it is now to your account on SAS Studio, and then, with your username rather than mine:

```
proc import
  datafile='/home/ken/denali.csv'
  out=denali
  dbms=csv
  replace;
  getnames=yes;
```

and then

```
proc print;
```

Obs	date	name	caribou	wolf
1	09/01/1995	David S.	30	66
2	09/24/1996	Youngjin K.	34	79
3	10/03/1997	Srinivasan M.	27	70
4	09/15/1998	Lee Anne J.	25	60
5	09/08/1999	Stephanie T.	17	48
6	09/03/2000	Angus Mc D.	23	55
7	10/06/2001	David S.	20	60

Go searching in the log tab for `proc import`, and below it you'll see some lines with `format` on them. This tells you how the variables were read. Mine is:

```
1596            informat date mmddyy10. ;
1597            informat name $13. ;
1598            informat caribou best32. ;
1599            informat wolf best32. ;
1600            format date mmddyy10. ;
1601            format name $13. ;
1602            format caribou best12. ;
1603            format wolf best12. ;
```

The dates were correctly deduced to be month-day-year, the names as text, and the caribou and wolf counts as numbers (that's what the `best` followed by a number is).

These appear to be duplicated because the `informat` lines are how the values are read in, and the `format` lines are how they are listed out (by default). These are not necessarily the same.

(d) Display the data set with the dates in Canadian/British format, day before month.

Solution: Add a `format` to the `proc print`. The right one is `ddmmyy10.`, to get the day before the month; a width of 10 characters gives room for the slashes and 4-digit years.

```
proc print;
format date ddmmyy10.;
```

Obs	date	name	caribou	wolf
1	01/09/1995	David S.	30	66
2	24/09/1996	Youngjin K.	34	79
3	03/10/1997	Srinivasan M.	27	70
4	15/09/1998	Lee Anne J.	25	60
5	08/09/1999	Stephanie T.	17	48
6	03/09/2000	Angus Mc D.	23	55
7	06/10/2001	David S.	20	60

Compare this one:

```
proc print;
  format date ddmmyy8.;
```

Obs	date	name	caribou	wolf
1	01/09/95	David S.	30	66
2	24/09/96	Youngjin K.	34	79
3	03/10/97	Srinivasan M.	27	70
4	15/09/98	Lee Anne J.	25	60
5	08/09/99	Stephanie T.	17	48
6	03/09/00	Angus Mc D.	23	55
7	06/10/01	David S.	20	60

This one has only two-digit years, leaving us prone to the "Y2K problem",[14] where it is not clear which century each year belongs to.

(e) Display the data set in such a way that you see the days of the week and the month names for the dates.

Solution: I left this open to you to make the precise choice of format, but one possibility is this one:

```
proc print;
  format date weekdate20.;
```

Obs	date	name	caribou	wolf
1	Fri, Sep 1, 1995	David S.	30	66
2	Tue, Sep 24, 1996	Youngjin K.	34	79
3	Fri, Oct 3, 1997	Srinivasan M.	27	70
4	Tue, Sep 15, 1998	Lee Anne J.	25	60
5	Wed, Sep 8, 1999	Stephanie T.	17	48
6	Sun, Sep 3, 2000	Angus Mc D.	23	55
7	Sat, Oct 6, 2001	David S.	20	60

The number on the end of `weekdate` is how many characters SAS uses to display the date. It makes the best of the space you give it:

```
proc print;
  format date weekdate5.;
```

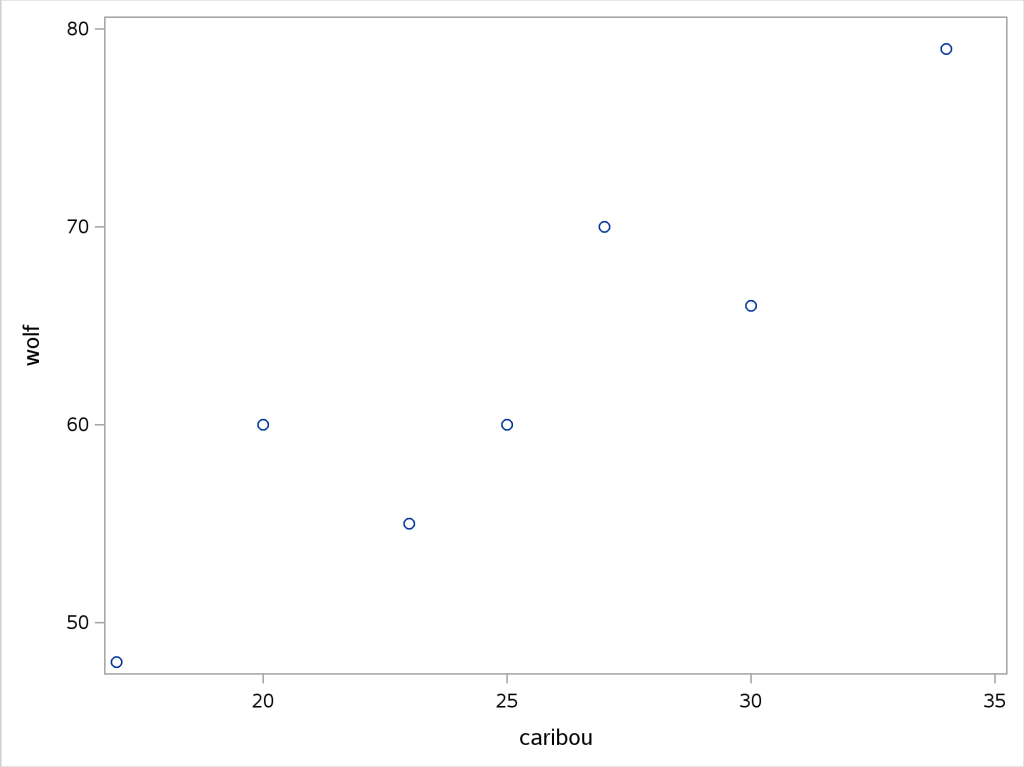
Obs	date	name	caribou	wolf
1	Fri	David S.	30	66
2	Tue	Youngjin K.	34	79
3	Fri	Srinivasan M.	27	70
4	Tue	Lee Anne J.	25	60
5	Wed	Stephanie T.	17	48
6	Sun	Angus Mc D.	23	55
7	Sat	David S.	20	60

The best it can do is to show you the day of the week.

(f) Enough playing around with dates. Make a scatterplot of caribou population (explanatory) against wolf population (response). Do you see any relationship?

Solution: The usual with `proc sgplot`:

```
proc sgplot;
  scatter x=caribou y=wolf;
```



That looks like an upward trend: when the caribou population is large, the wolf population is large too.[15]

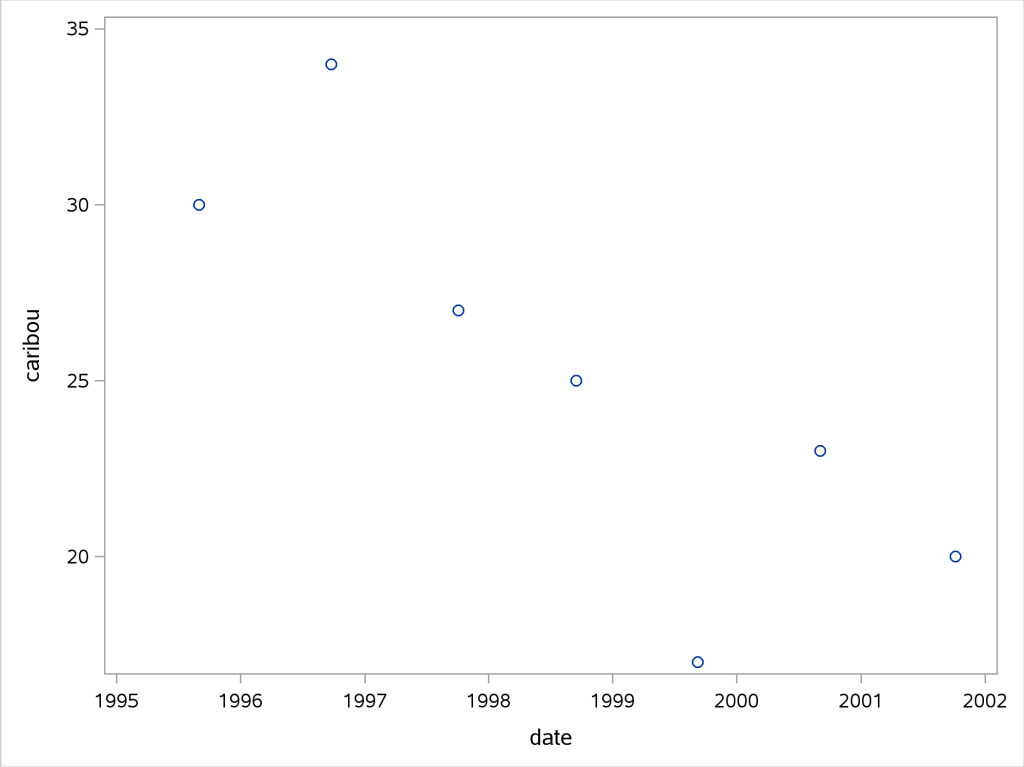
I should point out that the wolf and caribou surveys were taken at different times of the year. The dates in the data file were actually of the caribou surveys (in the fall, as I said). The wolf surveys were taken in the "late winter" (of the following year). This makes sense if you think of the wolf population as varying as a response to the caribou population; you need to allow some time for this "response" to happen. It might even be that the response happens over a longer time frame than this, if you think of the time required for wolves to have and raise pups,[16] which might be a period of years. In the grand scheme of things, there might be a multi-year cyclic variation in caribou and wolf populations; they go up and down together, but there might also be a time lag.

According to `http://www.pbs.org/wnet/nature/river-of-no-return-gray-wolf-fact-sheet/7659/`, wolves in the wild typically live 6–8 years, but many die earlier, often of starvation (so the size of the population of the wolves' prey animals matters a lot).

(g) Make a plot of caribou population against time (this is done the obvious way). What seems to be happening to the caribou population over time?

Solution: Make a scatterplot, with the survey date as explanatory variable, and caribou population as response (since time always goes on the x-axis):

```
proc sgplot;
  scatter x=date y=caribou;
```



A coding note here: I didn't need a `format` on my `proc sgplot`, because the dates are already formatted (from `proc import`). If they had been dates that we constructed ourselves, eg. from year, month and day as numbers, they would not have come with a format, and we would have had to supply one when we printed or plotted them.

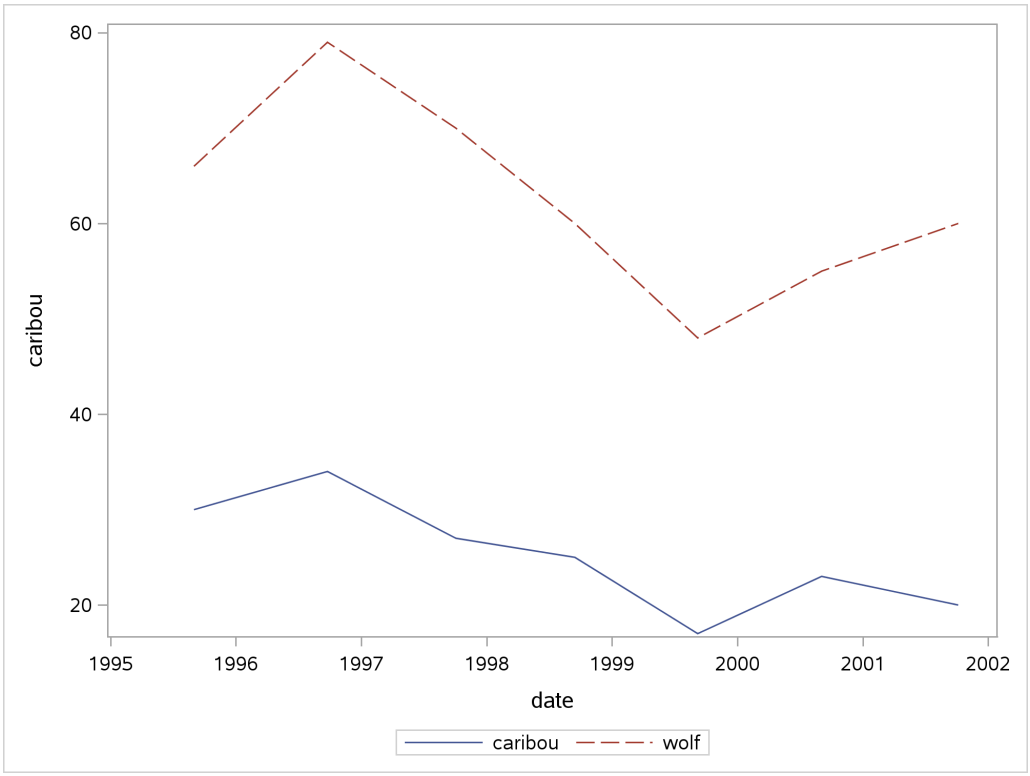
The caribou population is declining over time. We only have seven years' data, though, so it's not clear whether this is to do with climate change or some multi-year cycle in which wolf and caribou populations go up and down together, and we just happen to have hit the "down" part of the cycle.

Where the tick marks are on the x-axis mark the *start* of the year in question, so that the surveys come correctly about $\frac{3}{4}$ of the way through the year. SAS displayed just the years on the x-axis, since that was the scale of the data. If our dates happened to be all one year or all one month, you would have seen more of the format.

(h) The caribou and wolf populations over time are really "time series", so they can be plotted against time by `series` instead of `scatter`. Make a plot of *both* the caribou and wolf populations against time. (That is, use one `sgplot` with two `series` lines, where the `series` lines look just as `scatter` lines would.) How is `series` different from `scatter`?

Solution: I tried to give you enough hints:

```
proc sgplot;
  series x=date y=caribou;
  series x=date y=wolf;
```


80
60
40
20
caribou
1995
1996
1997
1998
1999
2000
2001
2002
date
caribou
wolf

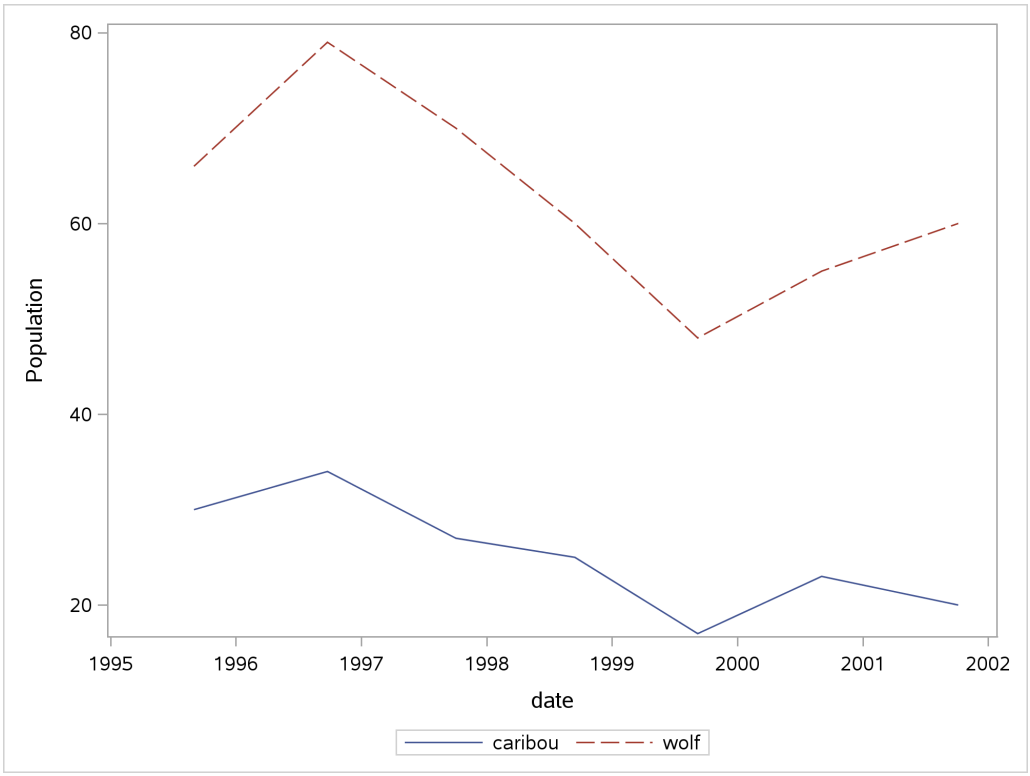
The big difference is that the points are joined by lines, each one to the next one in time order, so that the time nature of the data is more apparent. (Without the lines, it could be much less obvious which data value belongs to which series.)

Because we plotted two series, we also get a legend, and the two series are distinguished by colour and line type.

You should probably recall the scales: the caribou population was measured in hundreds, so the caribou numbers are a lot bigger than the wolf numbers. Evidently they measured caribou population in hundreds to get comparable numbers with the wolf population. In fact, I expect that park officials produced a graph very like this one. Probably in Excel, though.[17]

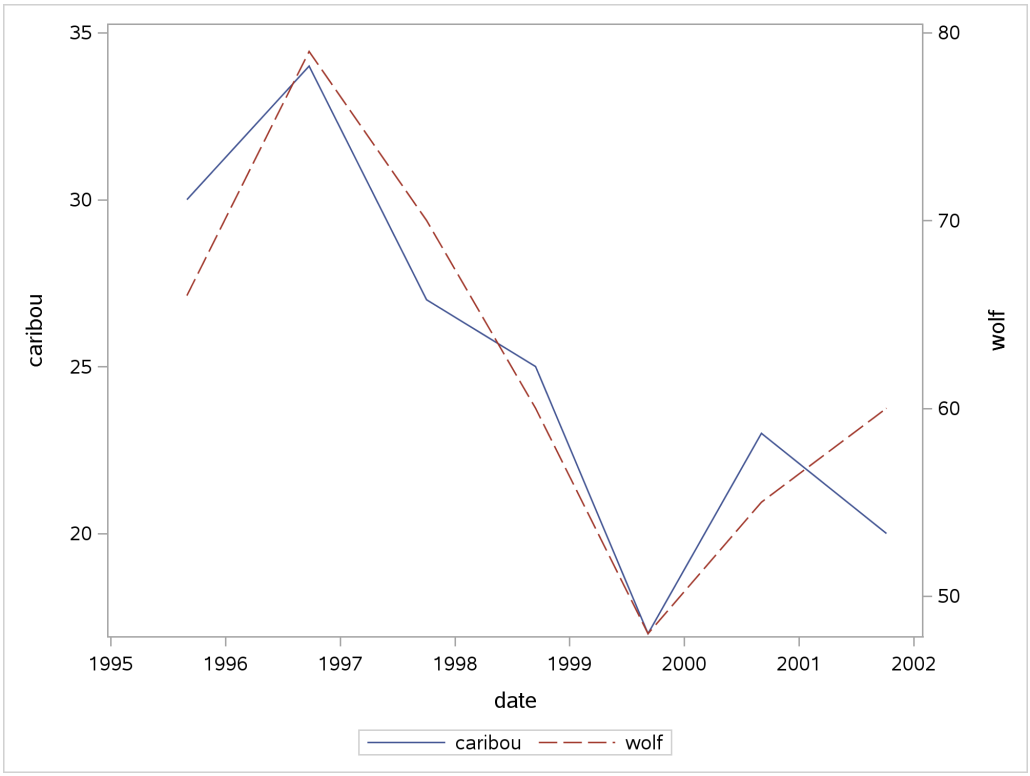
I should have labelled the y-axis "population". That can be done:

```
proc sgplot;
  series x=date y=caribou;
  series x=date y=wolf;
  yaxis label='Population';
```


80
60
40
20
Population
1995
1996
1997
1998
1999
2000
2001
2002
date
caribou
wolf

This is, in fact, one of those rare cases where we can justify having a second y-axis: left one for caribou, right one for wolf. Be aware, though, that you can scale the second y-axis how you like, which means that you can obtain a variety of apparent relationships between the two variables. This is the right way to do it (letting SAS choose the scale):

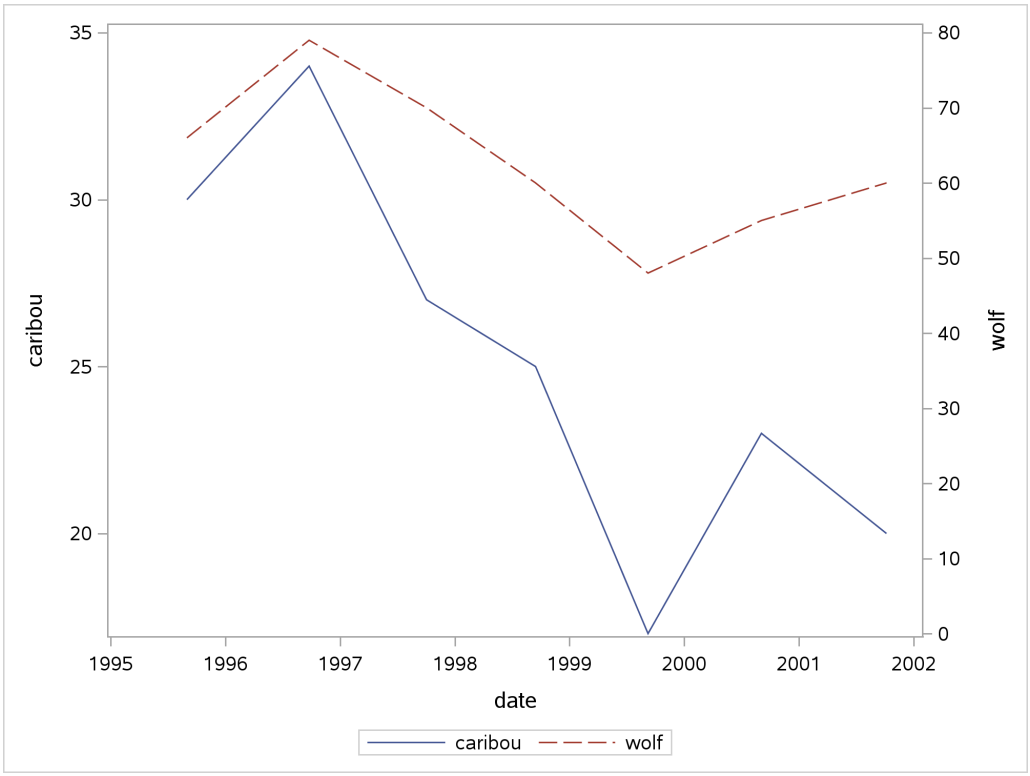
```
proc sgplot;
  series x=date y=caribou;
  series x=date y=wolf / y2axis;
```


35
30
25
20
caribou
80
70
60
50
wolf
1995
1996
1997
1998
1999
2000
2001
2002
date
caribou
wolf

This makes it even clearer that caribou and wolf populations rise and fall together.

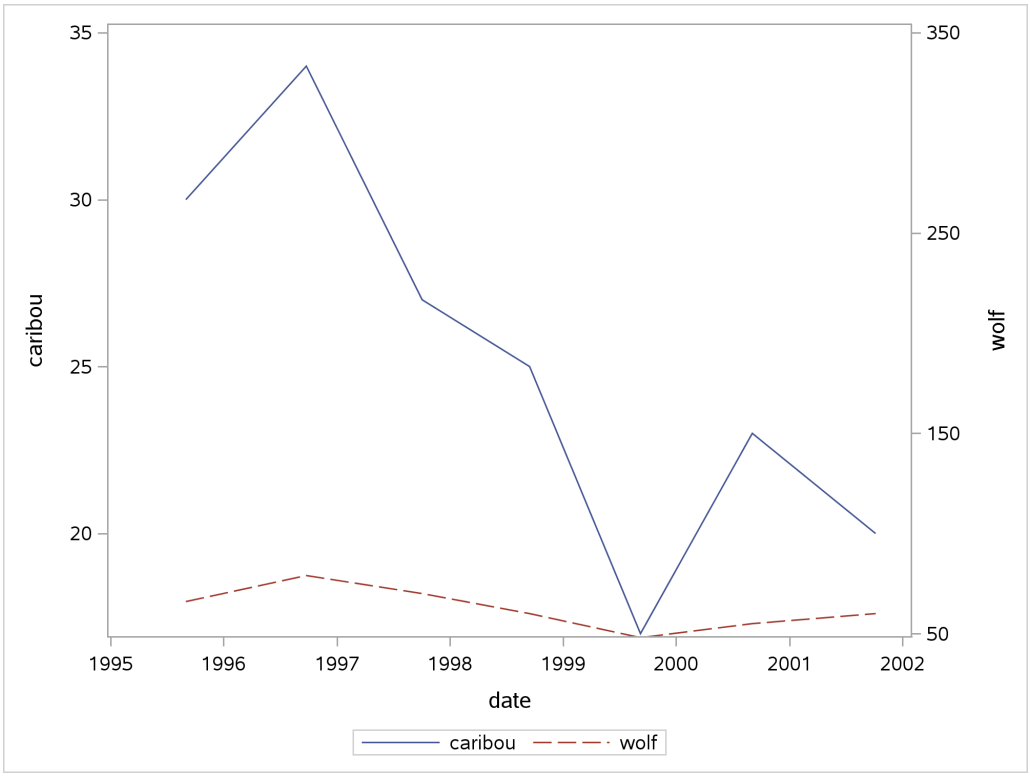
This is about the only double y-axis setup that I like, because you can choose the scale of the second y-axis however you like, to make it look as if the wolf population is very big:

```
proc sgplot;
  y2axis values=(0 to 80 by 10);
  series x=date y=caribou;
  series x=date y=wolf / y2axis;
```


35
30
25
20
caribou
80
70
60
50
40
30
20
10
0
wolf
1995
1996
1997
1998
1999
2000
2001
2002
date
caribou
wolf

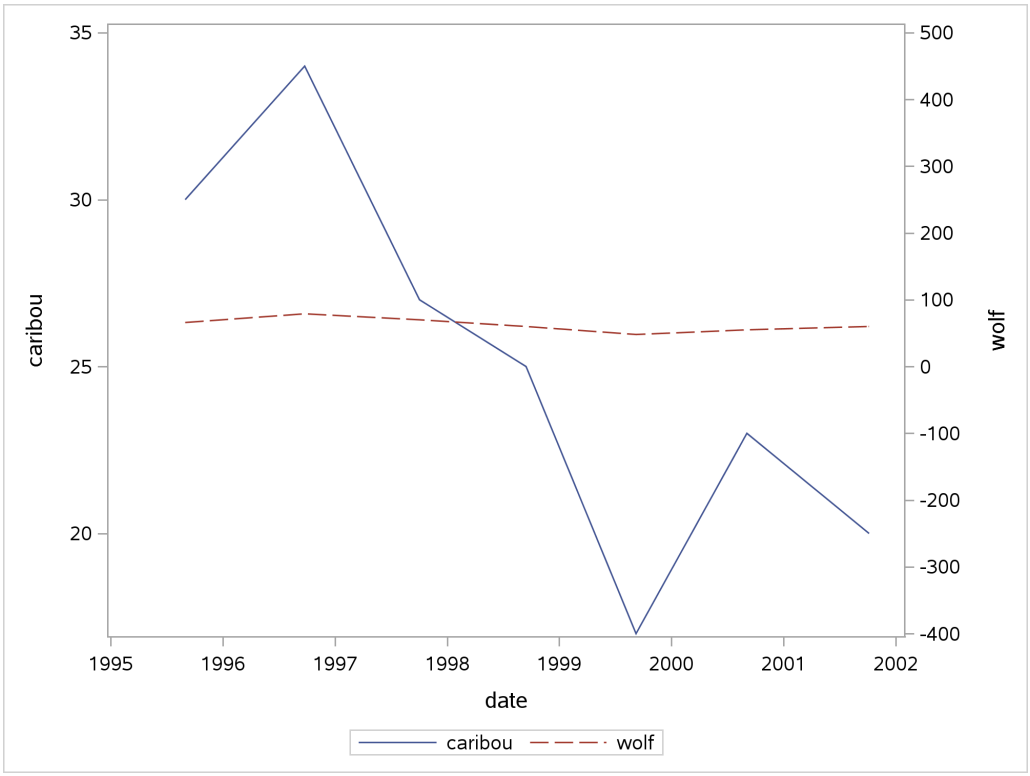
or very small:

```
proc sgplot;
  y2axis values=(50 to 400 by 100);
  series x=date y=caribou;
  series x=date y=wolf / y2axis;
```


35
30
25
20
caribou
350
250
150
50
wolf
1995
1996
1997
1998
1999
2000
2001
2002
date
caribou
wolf

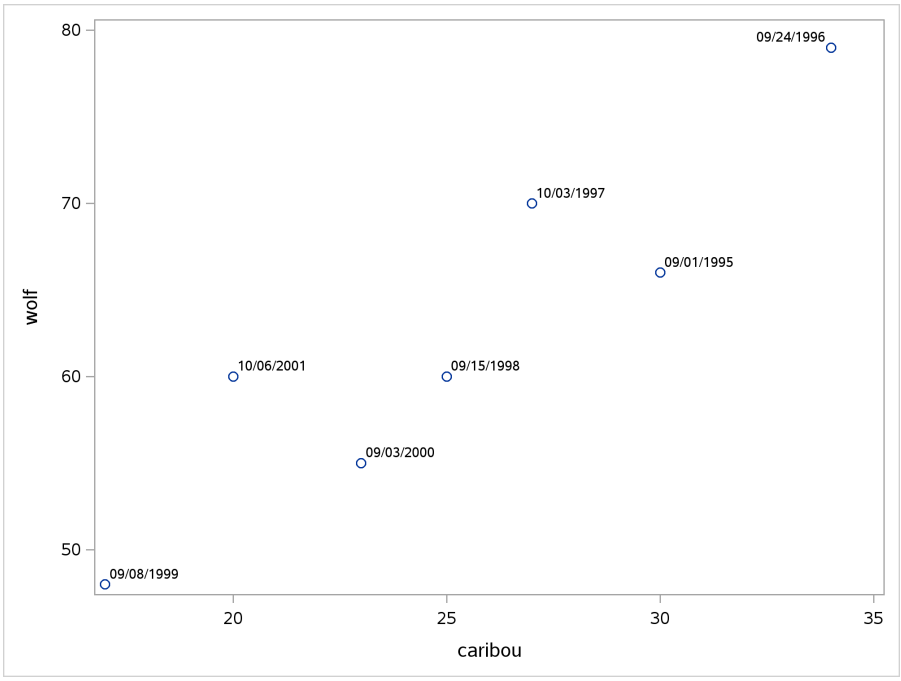
or hardly varies at all:

```
proc sgplot;
  y2axis values=(-400 to 500 by 100);
  series x=date y=caribou;
  series x=date y=wolf / y2axis;
```


35
30
25
20
caribou
500
400
300
200
100
0
-100
-200
-300
-400
wolf
1995
1996
1997
1998
1999
2000
2001
2002
date
caribou
wolf

In my opinion, too many people just plot series against time, possibly with a second y-axis. Variables that vary together, like the wolf and caribou populations here, ought to be plotted *against each other* on a scatterplot, possibly with the time points labelled:

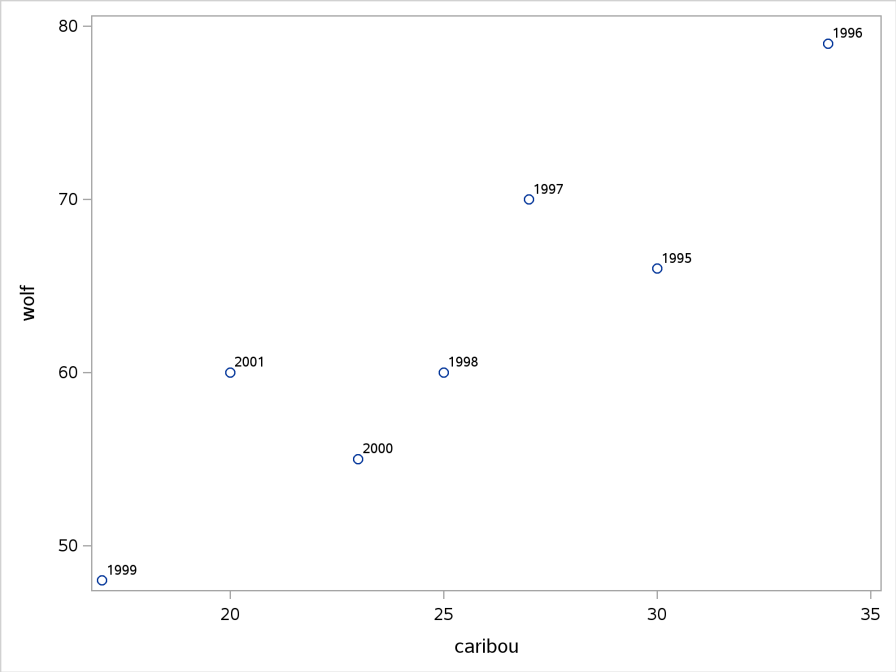
```
proc sgplot;
  scatter x=caribou y=wolf / datalabel=date;
```


80
70
60
50
wolf
09/24/1996
10/03/1997
09/01/1995
10/06/2001
09/15/1998
09/03/2000
09/08/1999
20
25
30
35
caribou

or, better, with just the years, which we obtain first:

```
data denali2;
  set denali;
  year=year(date);

proc sgplot;
  scatter x=caribou y=wolf / datalabel=year;
```



1996 was the high year for the populations, followed by a precipitous decline, and by 2000 or 2001 we would guess that the populations were beginning to increase again.

The ambitious among you may like to join the points by arrows, in time order. The further ambitious may like to compare the graphs here with other predator-prey relationships.

(i) Back in part (f), you drew a scatterplot of wolf population against caribou population. How does any trend there show up in the time plot you just drew?

Solution: Back in (f), we found an upward trend with the two populations: they tended to be large or small together. That should show up here as this: in a year where caribou population is large, wolf population is also large. In the fall 1996 survey, both populations were at their biggest, and in the fall 1999 survey, both populations were smallest. Also, the shapes of the time trends are very similar. So the plot of (f) and this one are telling the same story, with this plot giving the additional information that both populations (over this time span) are decreasing over time.

3. My cars data file can be found at `http://www.utsc.utoronto.ca/~butler/c32/cars.csv`. The values in the data file are separated by commas; the car names are up to 29 characters long. Display your results for each part after (a). In R, displaying a `tibble` normally shows its first ten lines, which is all

you need here; there's no need to display all the lines.

(a) Read the data into R and list the values.

Solution: `read_csv` will do it:

```
my_url="http://www.utsc.utoronto.ca/~butler/c32/cars.csv"
cars=read_csv(my_url)
```

```
## Parsed with column specification:
## cols(
##   car = col_character(),
##   MPG = col_double(),
##   weight = col_double(),
##   cylinders = col_integer(),
##   hp = col_integer(),
##   country = col_character()
## )
```

```
cars
```

```
## # A tibble: 38 x 6
##    car                 MPG weight cylinders    hp country
##    <chr>             <dbl>  <dbl>     <int> <int> <chr>
##  1 Buick Skylark      28.4   2.67         4    90 U.S.
##  2 Dodge Omni         30.9   2.23         4    75 U.S.
##  3 Mercury Zephyr     20.8   3.07         6    85 U.S.
##  4 Fiat Strada        37.3   2.13         4    69 Italy
##  5 Peugeot 694 SL     16.2   3.41         6   133 France
##  6 VW Rabbit          31.9   1.92         4    71 Germany
##  7 Plymouth Horizon   34.2   2.2          4    70 U.S.
##  8 Mazda GLC          34.1   1.98         4    65 Japan
##  9 Buick Estate Wagon 16.9   4.36         8   155 U.S.
## 10 Audi 5000          20.3   2.83         5   103 Germany
## # ... with 28 more rows
```

(b) Display only the car names and the countries they come from.

Solution:

```
cars %>% select(car,country)
```

```
## # A tibble: 38 x 2
##    car                country
##    <chr>              <chr>
##  1 Buick Skylark      U.S.
##  2 Dodge Omni         U.S.
##  3 Mercury Zephyr     U.S.
##  4 Fiat Strada        Italy
##  5 Peugeot 694 SL     France
##  6 VW Rabbit          Germany
##  7 Plymouth Horizon   U.S.
##  8 Mazda GLC          Japan
##  9 Buick Estate Wagon U.S.
## 10 Audi 5000          Germany
## # ... with 28 more rows
```

This *almost* works, but not quite:

```
cars %>% select(starts_with("c"))
```

```
## # A tibble: 38 x 3
##    car                cylinders country
##    <chr>                  <int> <chr>
##  1 Buick Skylark              4 U.S.
##  2 Dodge Omni                 4 U.S.
##  3 Mercury Zephyr             6 U.S.
##  4 Fiat Strada                4 Italy
##  5 Peugeot 694 SL             6 France
##  6 VW Rabbit                  4 Germany
##  7 Plymouth Horizon           4 U.S.
##  8 Mazda GLC                  4 Japan
##  9 Buick Estate Wagon         8 U.S.
## 10 Audi 5000                  5 Germany
## # ... with 28 more rows
```

It gets *all* the columns that start with `c`, which includes `cylinders` as well.

(c) Display everything *except* horsepower:

Solution: Naming what you *don't* want is sometimes easier:

```
cars %>% select(-hp)
```

```
## # A tibble: 38 x 5
##    car                  MPG weight cylinders country
##    <chr>              <dbl>  <dbl>     <int> <chr>
##  1 Buick Skylark       28.4   2.67         4 U.S.
##  2 Dodge Omni          30.9   2.23         4 U.S.
##  3 Mercury Zephyr      20.8   3.07         6 U.S.
##  4 Fiat Strada         37.3   2.13         4 Italy
##  5 Peugeot 694 SL      16.2   3.41         6 France
##  6 VW Rabbit           31.9   1.92         4 Germany
##  7 Plymouth Horizon    34.2   2.2          4 U.S.
##  8 Mazda GLC           34.1   1.98         4 Japan
##  9 Buick Estate Wagon  16.9   4.36         8 U.S.
## 10 Audi 5000           20.3   2.83         5 Germany
## # ... with 28 more rows
```

(d) Display only the cars that have 8-cylinder engines (but display all the variables for those cars).

Solution: This:

```
cars %>% filter(cylinders==8)
```

```
## # A tibble: 8 x 6
##   car                         MPG weight cylinders    hp country
##   <chr>                     <dbl>  <dbl>     <int> <int> <chr>
## 1 Buick Estate Wagon         16.9   4.36         8   155 U.S.
## 2 Chevy Malibu Wagon         19.2   3.60         8   125 U.S.
## 3 Chrysler LeBaron Wagon     18.5   3.94         8   150 U.S.
## 4 Ford LTD                   17.6   3.72         8   129 U.S.
## 5 Dodge St Regis             18.2   3.83         8   135 U.S.
## 6 Ford Country Squire Wagon  15.5   4.05         8   142 U.S.
## 7 Mercury Grand Marquis      16.5   3.96         8   138 U.S.
## 8 Chevy Caprice Classic      17     3.84         8   130 U.S.
```

8 of them, all from the US.

(e) Display the cylinders and horsepower for the cars that have horsepower 70 or less.

Solution: This one is selecting some observations and some variables:

```
cars %>% filter(hp<=70) %>% select(cylinders:hp)
```

```
## # A tibble: 6 x 2
##   cylinders    hp
##       <int> <int>
## 1         4    69
## 2         4    70
## 3         4    65
## 4         4    65
## 5         4    68
## 6         4    68
```

Cylinders and horsepower are consecutive columns, so we can select them either using the colon `:` or by `c(cylinders,hp)`.

You can also do the `filter` and the `select` the other way around. This one works because the *rows* you want to choose are determined by a column you're going to keep. If you wanted to display the cylinders and horsepower of the cars with `mpg` over 30, you would have to choose the rows first, because after you've chosen the columns, there is no `mpg` any more.

(f) Find the mean and SD of gas mileage of the cars with 4 cylinders.

Solution:

```
cars %>% filter(cylinders==4) %>% summarize(m=mean(MPG),s=sd(MPG))
```

```
## # A tibble: 1 x 2
##       m     s
##   <dbl> <dbl>
## 1  30.0  4.18
```

Or you can get the mean and SD of gas mileage for all numbers of cylinders, and pick out the one you want:

```
cars %>% group_by(cylinders) %>% summarize(m=mean(MPG),s=sd(MPG))
```

```
## # A tibble: 4 x 3
##   cylinders     m     s
##       <int> <dbl> <dbl>
## 1         4  30.0  4.18
## 2         5  20.3 NA
## 3         6  21.1  4.08
## 4         8  17.4  1.19
```

Top row is the same as before. And since the output is a data frame, you can do any of these things with *that*, for example:

```
cars %>% group_by(cylinders) %>%
  summarize(m=mean(MPG),s=sd(MPG)) %>%
  filter(cylinders==4)
```

```
## # A tibble: 1 x 3
##   cylinders     m     s
##       <int> <dbl> <dbl>
## 1         4  30.0  4.18
```

to pick out just the right row.

This is a very easy kind of question to set on an exam. Just so you know.

4. The City of Toronto collects all kinds of data on aspects of life in the city. See `http://www1.toronto.ca/wps/portal/contentonly?vgnextoid=1a66e03bb8d1e310VgnVCM10000071d60f89RCRD`. One collection of data is records of the number of cyclists on certain downtown streets. The data in `http://www.utsc.utoronto.ca/~butler/c32/bikes.csv` are a record of the cyclists on College Street on the block west from Huron to Spadina on September 24, 2010. In the spreadsheet, each row relates to one cyclist. The first column is the time the cyclist was observed (to the nearest 15 minutes). After that, there are four pairs of columns. The observer filled in (exactly) one X in each pair of columns, according to whether (i) the cyclist was male or female, (ii) was or was not wearing a helmet, (iii) was or was not carrying a

passenger on the bike, (iv) was or was not riding on the sidewalk. We want to create a tidy data frame that has the time in each row, and has columns containing appropriate values, often `TRUE` or `FALSE`, for each of the four variables measured.

I will lead you through the process, which will involve developing a (long) pipe, one step at a time.

(a) Take a look at the spreadsheet (using Excel or similar: this may open when you click the link). Are there any obvious header rows? Is there any extra material before the data start? Explain briefly.

Solution: This is what I see (you should see something that looks like this):

	A	B	C	D	E	F	G	H	I	J
1	Ontario Traffic Inc - Bicycle Counts									
2										
3	Location:	College St from Spadina Ave to Huron St				Date:	09/24/2010			
4										
5	Direction:	WB				Weather:	Clear			
6										
7	Start Time	Gender		Wearing Helmet		Passenger On The Bike		Riding On The Sidewalk		
8	(every 15 min)	Male	Female	Yes	No	Yes	No	Yes	No	
9	7:00	X			X		X		X	
10		X			X		X		X	
11		X			X		X		X	
12		X			X		X		X	
13		X			X		X		X	
14	7:15	X		X			X		X	
15			X	X			X		X	
16		X		X			X		X	
17			X	X			X	X		

There are really *two* rows of headers (the rows highlighted in yellow). The actual information that says what the column pair is about is in the first of those two rows, and the second row indicates which category of the information above this column refers to.

This is not the usual way that the column headers encode what the columns are about: we are used to having *one* row which here would contain things like `gender.female` or `helmet.yes`.

There are also six lines above the highlighted ones that contain background information about this study. (This is where I got the information about the date of the study and which block of which street it is about.)

I am looking for two things: the apparent header line is actually two lines (the ones in yellow), and there are extra lines above that which are not data.

(b) Read the data into an R data frame. Read *without* headers, and instruct R how many lines to skip over using `skip=` and a suitable number.

When this is working, display the first few lines of your data frame. Note that your columns have names `X1` through `X9`.

Solution: The actual data start on line 9, so we need to skip 8 lines. `col_names=F` is the way to say that we have no column names (not ones that we want to use, anyway). Just typing the name of the data frame will display "a few" (that is, 10) lines of it, so that you can check it for plausibleness:

```
bikes=read_csv("bikes.csv",skip=8,col_names=F)
```

```
## Parsed with column specification:
## cols(
##   X1 = col_time(format = ""),
##   X2 = col_character(),
##   X3 = col_character(),
##   X4 = col_character(),
##   X5 = col_character(),
##   X6 = col_character(),
##   X7 = col_character(),
##   X8 = col_character(),
##   X9 = col_character()
## )
```

```
bikes
```

```
## # A tibble: 1,958 x 9
##    X1     X2    X3    X4    X5    X6    X7    X8    X9
##    <time> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
##  1 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  2    NA  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  3    NA  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  4    NA  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  5    NA  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  6 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X
##  7    NA  <NA>  X     X     <NA>  <NA>  X     <NA>  X
##  8    NA  X     <NA>  X     <NA>  <NA>  X     <NA>  X
##  9    NA  <NA>  X     X     <NA>  <NA>  X     X     <NA>
## 10    NA  X     <NA>  X     <NA>  <NA>  X     <NA>  X
## # ... with 1,948 more rows
```

This seems to have worked: a column with times in it, and four pairs of columns, with exactly one of each pair having an X in it. The variable names `X1` through `X9` were generated by `read_csv`, as it does when you read in data with `col_names=F`. The times are correctly `times`, and the other columns are all text. The blank cells in the spreadsheet have appeared in our data frame as "missing" (`NA`).

The first line in our data frame contains the first 7:00 (am) cyclist, so it looks as if we skipped the right number of lines.

(c) What do you notice about the times in your first column? What do you think those "missing" times should be? (Have a think about this before you move on.)

`fill` from `tidyr` fills in the missing times with the previous non-missing value. Display the first few lines of your "filled" data frame. (This will mean finding the help for `fill` in R Studio or online.)

Start a pipe from the data frame you read in, that updates the appropriate column with the filled-in times.

Solution: It looks as if those `NA` values in `X1` are the same time as the previous non-missing value. For example, the first five rows are cyclists observed at 7:00 am (or, at least, between 7:00 and 7:15). So they should be recorded as 7:00, and the ones in rows 7–10 should be recorded

as 7:15, and so on.

If you look in the help for `fill` via `?fill` (or if you Google `tidyr::fill`, which is the full name for "the `fill` that lives in `tidyr`"), you'll see that it requires up to two things (not including the data frame): a column to fill, and a direction to fill it (the default of "down" is exactly what we want). Thus:

```
bikes %>% fill(X1)
```

```
## # A tibble: 1,958 x 9
##    X1     X2    X3    X4    X5    X6    X7    X8    X9
##    <time> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
##  1 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  2 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  3 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  4 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  5 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  6 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X
##  7 07:15  <NA>  X     X     <NA>  <NA>  X     <NA>  X
##  8 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X
##  9 07:15  <NA>  X     X     <NA>  <NA>  X     X     <NA>
## 10 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X
## # ... with 1,948 more rows
```

Success!

We will probably want to rename `X1` to something like `time`, so let's do that now before we forget. There is a `rename` that does about what you'd expect:

```
bikes %>% fill(X1) %>% rename(Time=X1)
```

```
## # A tibble: 1,958 x 9
##    Time   X2    X3    X4    X5    X6    X7    X8    X9
##    <time> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
##  1 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  2 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  3 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  4 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  5 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X
##  6 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X
##  7 07:15  <NA>  X     X     <NA>  <NA>  X     <NA>  X
##  8 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X
##  9 07:15  <NA>  X     X     <NA>  <NA>  X     X     <NA>
## 10 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X
## # ... with 1,948 more rows
```

I gave it a capital T so as not to confuse it with other things in R called `time`.

(d) R's `ifelse` function works like `=IF` in Excel. You use it to create values for a new variable, for example in a `mutate`. The first input to it is a logical condition (something that is either true or false); the second is the value your new variable should take if the condition is true, and the third is the value of your new variable if the condition is false. Create a new column `gender` in your data frame that is "male" or "female" depending on the value of your `X2` column, using `mutate`. (You can assume that exactly one of the second and third columns has an `X` in it.) Add your code to the end of your pipe and display (the first 10 rows of) the result.

Solution: Under the assumption we are making, we only have to look at column `X2` and we ignore `X3` totally:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=ifelse(X2=="X","male","female"))
```

```
## # A tibble: 1,958 x 10
##    Time   X2    X3    X4    X5    X6    X7    X8    X9    gender
##    <time> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
##  1 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  2 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  3 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  4 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  5 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  6 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male
##  7 07:15  <NA>  X     X     <NA>  <NA>  X     <NA>  X     <NA>
##  8 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male
##  9 07:15  <NA>  X     X     <NA>  <NA>  X     X     <NA>  <NA>
## 10 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male
## # ... with 1,948 more rows
```

Oh, that didn't work. The gender column is either `male` or missing; the two missing ones here should say `female`. What happened? Let's just look at our logical condition this time:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(isX=(X2=="X"))
```

```
## # A tibble: 1,958 x 10
##    Time   X2    X3    X4    X5    X6    X7    X8    X9    isX
##    <time> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <lgl>
##  1 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     TRUE
##  2 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     TRUE
##  3 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     TRUE
##  4 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     TRUE
##  5 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     TRUE
##  6 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     TRUE
##  7 07:15  <NA>  X     X     <NA>  <NA>  X     <NA>  X     NA
##  8 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     TRUE
##  9 07:15  <NA>  X     X     <NA>  <NA>  X     X     <NA>  NA
## 10 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     TRUE
## # ... with 1,948 more rows
```

This is not true and false, it is true and missing. The idea is that if `X2` is missing, we don't (in general) know what its value is: it might even be `X`! So if `X2` is missing, any comparison of it with another value ought to be missing as well.

That's in general. Here, we know where those missing values came from: they were blank cells in the spreadsheet, so we actually have more information.

Perhaps a better way to go is to test whether `X2` is missing (in which case, it's a female cyclist). R has a function `is.na` which is `TRUE` if the thing inside it is missing and `FALSE` if the thing inside it has some non-missing value. In our case, it goes like this:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=ifelse(is.na(X2),"female","male"))
```

```
## # A tibble: 1,958 x 10
##    Time   X2    X3    X4    X5    X6    X7    X8    X9    gender
##    <time> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
##  1 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  2 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  3 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  4 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  5 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male
##  6 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male
##  7 07:15  <NA>  X     X     <NA>  <NA>  X     <NA>  X     female
##  8 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male
##  9 07:15  <NA>  X     X     <NA>  <NA>  X     X     <NA>  female
## 10 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male
## # ... with 1,948 more rows
```

Or you can test `X3` for missingness: if missing, it's male, otherwise it's female. That also works.

This made an assumption that the person recording the X's actually *did* mark an X in exactly one of the columns. For example, the columns could *both* be missing, or *both* have an X in them. This gives us more things to check, at least three. `ifelse` is good for something with only two alternatives, but when you have more, `case_when` is much better.[18] Here's how that goes. Our strategy is to check for three things: (i) `X2` has an `X` and `X3` is missing; (ii) `X2` is missing and `X3` has an `X`; (iii) anything else, which is an error:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=case_when(
    X2=="X" & is.na(X3) ~ "Male",
    is.na(X2) & X3=="X" ~ "Female",
    TRUE                ~ "Error!"))
```

```
## # A tibble: 1,958 x 10
##    Time   X2    X3    X4    X5    X6    X7    X8    X9    gender
##    <time> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
##  1 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     Male
##  2 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     Male
##  3 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     Male
##  4 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     Male
##  5 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     Male
##  6 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     Male
##  7 07:15  <NA>  X     X     <NA>  <NA>  X     <NA>  X     Female
##  8 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     Male
##  9 07:15  <NA>  X     X     <NA>  <NA>  X     X     <NA>  Female
## 10 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     Male
## # ... with 1,948 more rows
```

It seems nicest to format it like that, with the squiggles lining up, so you can see what possible values `gender` might take.

The structure of the `case_when` is that the thing you're checking for goes on the left of the squiggle, and the value you want your new variable to take goes on the right. What it does is to go down the list of conditions that you are checking for, and as soon as it finds one that is

true, it grabs the value on the right of the squiggle and moves on to the next row. The usual way to write these is to have a catch-all condition at the end that is always true, serving to make sure that your new variable always gets *some* value. `TRUE` is, um, always true. If you want an English word for the last condition of your `case_when`, "otherwise" is a nice one.

I wanted to check that the observer did check exactly one of `V2` and `V3` as I asserted, which can be done by gluing this onto the end:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=case_when(
    X2=="X" & is.na(X3) ~ "Male",
    is.na(X2) & X3=="X" ~ "Female",
    TRUE                ~ "Error!")) %>%
  count(gender)
```

```
## # A tibble: 2 x 2
##   gender     n
##   <chr>  <int>
## 1 Female   861
## 2 Male    1097
```

There are only Males and Females, so the observer really did mark exactly one X. (As a bonus, you see that there were slightly more male cyclists than female ones.)

(e) Create variables `helmet`, `passenger` and `sidewalk` in your data frame that are `TRUE` if the "Yes" column contains `X` and `FALSE` otherwise. This will use `mutate` again, but you don't need `ifelse`: just set the variable equal to the appropriate logical condition. As before, the best way to create these variables is to test the appropriate things for missingness. Note that you can create as many new variables as you like in one `mutate`. Show the first few lines of your new data frame. (Add your code onto the end of the pipe you made above.)

Solution:

On the face of it, the way to do this is to go looking for `X`'s:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=ifelse(is.na(X2),"female","male")) %>%
  mutate(helmet=(X4=="X"),
         passenger=(X6=="X"),
         sidewalk=(X8=="X"))
```

```
## # A tibble: 1,958 x 13
##    Time  X2    X3    X4    X5    X6    X7    X8    X9    gender helmet
##    <tim> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>  <lgl>
##  1 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   NA
##  2 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   NA
##  3 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   NA
##  4 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   NA
##  5 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   NA
##  6 07:15 X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
##  7 07:15 <NA>  X     X     <NA>  <NA>  X     <NA>  X     female TRUE
##  8 07:15 X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
##  9 07:15 <NA>  X     X     <NA>  <NA>  X     X     <NA>  female TRUE
## 10 07:15 X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
## # ... with 1,948 more rows, and 2 more variables: passenger <lgl>,
## #   sidewalk <lgl>
```

But, we run into the same problem that we did with `gender`: the new variables are either `TRUE` or missing, never `FALSE`.

The solution is the same: look for the things that are *missing* if the cyclist is wearing a helmet, carrying a passenger or riding on the sidewalk. These are `X5, X7, X9` respectively:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=ifelse(is.na(X2),"female","male")) %>%
  mutate(helmet=is.na(X5),
         passenger=is.na(X7),
         sidewalk=is.na(X9))
```

```
## # A tibble: 1,958 x 13
##    Time  X2    X3    X4    X5    X6    X7    X8    X9    gender helmet
##    <tim> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>  <lgl>
##  1 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  2 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  3 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  4 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  5 07:00 X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  6 07:15 X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
##  7 07:15 <NA>  X     X     <NA>  <NA>  X     <NA>  X     female TRUE
##  8 07:15 X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
##  9 07:15 <NA>  X     X     <NA>  <NA>  X     X     <NA>  female TRUE
## 10 07:15 X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
## # ... with 1,948 more rows, and 2 more variables: passenger <lgl>,
## #   sidewalk <lgl>
```

Again, you can do the `mutate` all on one line if you want to, or all four variable assignments in one `mutate`, but I used newlines and indentation to make the structure clear.

It is less elegant, though equally good for the purposes of the assignment, to use `ifelse` for

these as well, which would go like this, for example:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=ifelse(X2=="X","male","female")) %>%
  mutate(helmet=ifelse(is.na(X5),T,F))
```

```
## # A tibble: 1,958 x 11
##    Time   X2    X3    X4    X5    X6    X7    X8    X9    gender helmet
##    <time> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>  <lgl>
##  1 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  2 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  3 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  4 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  5 07:00  X     <NA>  <NA>  X     <NA>  X     <NA>  X     male   FALSE
##  6 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
##  7 07:15  <NA>  X     X     <NA>  <NA>  X     <NA>  X     <NA>   TRUE
##  8 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
##  9 07:15  <NA>  X     X     <NA>  <NA>  X     X     <NA>  <NA>   TRUE
## 10 07:15  X     <NA>  X     <NA>  <NA>  X     <NA>  X     male   TRUE
## # ... with 1,948 more rows
```

and the same for `passenger` and `sidewalk`. The warning is, whenever you see a `T` and an `F` in an `ifelse`, that you could probably get rid of the `ifelse` and use the logical condition directly.[19] For `gender`, though, you need the `ifelse` (or a `case_when`) because the values you want it to take are `male` and `female`, something other than `TRUE` and `FALSE`.

I like to put brackets around logical conditions when I am assigning them to a variable. If I don't, I get something like

```
helmet=V4=="X"
```

which actually works, but is very hard to read. Well, I *think* it works. Let's check:

```
exes=c("X","","X","","X")
y=exes=="X"
y
```

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

Yes it does. But I would never recommend writing it this way, because unless you are paying attention, you won't notice which `=` is saving in a variable, and which one is "logically equal".

It works because of a thing called "operator precedence": the logical-equals is evaluated first, and the result of that is saved in the variable. But unless you or your readers remember that, it's better to write

```
y=(exes=="X")
```

to draw attention to the order of calculation. This is the same reason that

```
4+5*6
```

```
## [1] 34
```

evaluates this way rather than doing the addition first and getting 54. BODMAS and all that.

(f) Finally (for the data manipulation), get rid of all the original columns, keeping only the new ones that you created. Save the results in a data frame and display its first few rows.

Solution: This is a breath of fresh air after all the thinking needed above: this is just `select`, added to the end:

```
mybikes = bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=ifelse(is.na(X2),"female","male")) %>%
  mutate(helmet=is.na(X5),
         passenger=is.na(X7),
         sidewalk=is.na(X9)) %>%
  select(-(X2:X9))
mybikes
```

```
## # A tibble: 1,958 x 5
##    Time   gender helmet passenger sidewalk
##    <time> <chr>  <lgl>  <lgl>     <lgl>
##  1 07:00  male   FALSE  FALSE     FALSE
##  2 07:00  male   FALSE  FALSE     FALSE
##  3 07:00  male   FALSE  FALSE     FALSE
##  4 07:00  male   FALSE  FALSE     FALSE
##  5 07:00  male   FALSE  FALSE     FALSE
##  6 07:15  male   TRUE   FALSE     FALSE
##  7 07:15  female TRUE   FALSE     FALSE
##  8 07:15  male   TRUE   FALSE     FALSE
##  9 07:15  female TRUE   FALSE     TRUE
## 10 07:15  male   TRUE   FALSE     FALSE
## # ... with 1,948 more rows
```

You might not have renamed your `X1`, in which case, you still have it, but need to keep it (because it holds the times).

Another way to do this is to use a "select-helper", thus:

```
bikes %>% fill(X1) %>% rename(Time=X1) %>%
  mutate(gender=ifelse(is.na(X2),"female","male")) %>%
  mutate(helmet=is.na(X5),
         passenger=is.na(X7),
         sidewalk=is.na(X9)) %>%
  select(-num_range("X",2:9))
```

```
## # A tibble: 1,958 x 5
##    Time   gender helmet passenger sidewalk
##    <time> <chr>  <lgl>  <lgl>     <lgl>
##  1 07:00  male   FALSE  FALSE     FALSE
##  2 07:00  male   FALSE  FALSE     FALSE
##  3 07:00  male   FALSE  FALSE     FALSE
##  4 07:00  male   FALSE  FALSE     FALSE
##  5 07:00  male   FALSE  FALSE     FALSE
##  6 07:15  male   TRUE   FALSE     FALSE
##  7 07:15  female TRUE   FALSE     FALSE
##  8 07:15  male   TRUE   FALSE     FALSE
##  9 07:15  female TRUE   FALSE     TRUE
## 10 07:15  male   TRUE   FALSE     FALSE
## # ... with 1,948 more rows
```

This means "get rid of all the columns whose names are `X` followed by a number 2 through 9".

The pipe looks long and forbidding, but you built it (and tested it) a little at a time. Which is how you do it.

(g) The next few parts are a quick-fire analysis of the data set. They can all be solved using `count`.

How many male and how many female cyclists were observed in total?

Solution:

I already got this one when I was checking for observer-notation errors earlier:

```
mybikes %>% count(gender)
```

```
## # A tibble: 2 x 2
##   gender     n
##   <chr>  <int>
## 1 female   861
## 2 male    1097
```

861 females and 1097 males.

(h) How many male and female cyclists were not wearing helmets?

Solution:

You can count two variables at once, in which case you get counts of all combinations of them:

```
mybikes %>% count(gender,helmet)
```

```
## # A tibble: 4 x 3
##   gender helmet     n
##   <chr>  <lgl>  <int>
## 1 female FALSE    403
## 2 female TRUE     458
## 3 male   FALSE    604
## 4 male   TRUE     493
```

403 females and 604 males were not wearing helmets, picking out what we need.

The real question of interest here is "what *proportion* of male and female cyclists were not wearing helmets?"[20] This has a rather elegant solution that I will have to explain. First, let's go back to the `group_by` and `summarize` version of the `count` here:

```
mybikes %>% group_by(gender,helmet) %>%
  summarize(count=n())
```

```
## # A tibble: 4 x 3
## # Groups:   gender [?]
##   gender helmet count
##   <chr>  <lgl>  <int>
## 1 female FALSE    403
## 2 female TRUE     458
## 3 male   FALSE    604
## 4 male   TRUE     493
```

That's the same table we got just now. Now, let's calculate a proportion and see what happens:

```
mybikes %>% group_by(gender,helmet) %>%
  summarize(count=n()) %>%
  mutate(prop=count/sum(count))
```

```
## # A tibble: 4 x 4
## # Groups:   gender [2]
##   gender helmet count  prop
##   <chr>  <lgl>  <int> <dbl>
## 1 female FALSE    403 0.468
## 2 female TRUE     458 0.532
## 3 male   FALSE    604 0.551
## 4 male   TRUE     493 0.449
```

We seem to have the proportions of males and females who were and were not wearing a helmet, and you can check that this is indeed the case, for example:

```
403/(403+458)
```

```
## [1] 0.4680604
```

47% of females were not wearing helmets, while 55% of males were helmetless. (You can tell from the original frequencies that a small majority of females wore helmets and a small majority of males did not.)

Now, we have to ask ourselves: how on earth did that work?

When you calculate a summary (like our `sum(count)` above), it figures that you can't want the sum by gender-helmet combination, since you already have those in `count`. You must want the sum *over* something. What? What happens is that it goes back to the `group_by` and "peels off" the last thing there, which in this case is `helmet`, leaving only `gender`. It then sums the counts for each gender, giving us what we wanted.

It just blows my mind that someone (ie., Hadley Wickham) could (i) think that this would be a nice syntax to have (instead of just being an error), (ii) find a way to implement it and (iii) find a nice logical explanation ("peeling off") to explain how it worked.

What happens if we switch the order of the things in the `group_by`?

```
mybikes %>% group_by(helmet,gender) %>%
  summarize(count=n()) %>%
  mutate(prop=count/sum(count))
```

```
## # A tibble: 4 x 4
## # Groups:   helmet [2]
##   helmet gender count  prop
##   <lgl>  <chr>  <int> <dbl>
## 1 FALSE  female   403 0.400
## 2 FALSE  male     604 0.600
## 3 TRUE   female   458 0.482
## 4 TRUE   male     493 0.518
```

Now we get the proportion of helmeted riders of each gender, which is not the same as what we had before. Before, we had "out of males" and "out of females"; now we have "out of helmeted riders" and "out of helmetless riders". (The riders with helmets are almost 50–50 males and females, but the riders without helmets are about 60% male.)

This is row and column proportions in a contingency table, B22 style.

Now, I have to see whether the `count` variant of this works:

```
mybikes %>% count(gender,helmet) %>%
  mutate(prop=count/sum(count))
```

```
## Error in mutate_impl(.data, dots): Evaluation error: invalid 'type' (closure)
of argument.
```

It doesn't. So I think you have to do this the `group_by` and `summarize` way.

(i) How many cyclists were riding on the sidewalk *and* carrying a passenger?

Solution: Not too many, I'd hope. Again:

```
mybikes %>% count(passenger, sidewalk)
```

```
## # A tibble: 3 x 3
##   passenger sidewalk     n
##   <lgl>     <lgl>    <int>
## 1 FALSE     FALSE     1880
## 2 FALSE     TRUE        73
## 3 TRUE      FALSE        5
```

We're looking for the "true", "true" entry of that table, which seems to have vanished. That means the count is *zero*: none at all. (There were only 5 passenger-carrying riders, and they were all on the road.)

(j) What was the busiest 15-minute period of the day, and how many cyclists were there then?

Solution: The obvious way is to list every 15-minute period and eyeball the largest frequency. There are quite a few 15-minute periods, so we need to use `print` to show them all:

```
mybikes %>% count(Time) %>% print(n=Inf)
```

```
## # A tibble: 48 x 2
##    Time       n
##    <time> <int>
##  1 07:00      5
##  2 07:15      8
##  3 07:30      9
##  4 07:45      5
##  5 08:00     18
##  6 08:15     14
##  7 08:30     12
##  8 08:45     22
##  9 09:00     17
## 10 09:15     15
## 11 09:30     16
## 12 09:45     12
## 13 10:00      8
## 14 10:15     21
## 15 10:30     22
## 16 10:45     19
## 17 11:00     30
## 18 11:15     14
## 19 11:30     27
## 20 11:45     28
## 21 12:00     22
## 22 12:15     35
## 23 12:30     40
## 24 12:45     33
## 25 13:00     44
## 26 13:15     46
## 27 13:30     29
## 28 13:45     41
## 29 14:00     36
## 30 14:15     45
## 31 14:30     33
## 32 14:45     50
## 33 15:00     49
## 34 15:15     51
## 35 15:30     61
## 36 15:45     79
## 37 16:00    101
## 38 16:15     56
## 39 16:30     51
## 40 16:45     75
## 41 17:00    104
## 42 17:15    128
## 43 17:30     80
## 44 17:45     73
## 45 18:00     75
## 46 18:15     78
## 47 18:30     63
## 48 18:45     58
```

17:15, or 5:15 pm, with 128 cyclists.

But, computers are meant to save us that kind of effort. How? Note that the output from `count` is itself a data frame, so anything you can do to a data frame, you can do to *it*: for example, display only the rows where the frequency equals the maximum frequency:

```
mybikes %>% count(Time) %>%
  filter(n==max(n))
```

```
## # A tibble: 1 x 2
##   Time       n
##   <time> <int>
## 1 17:15    128
```

That will actually display *all* the times where the cyclist count equals the maximum, of which there might be more than one.

5. In summer, the city of Toronto issues Heat Alerts for "high heat or humidity that is expected to last two or more days". The precise definitions are shown at `http://www1.toronto.ca/wps/portal/contentonly?vgnextoid=923b5ce6dfb31410VgnVCM10000071d60f89RCRD`. During a heat alert, the city opens Cooling Centres and may extend the hours of operation of city swimming pools, among other things. All the heat alert days from 2001 to 2016 are listed at `http://www.utsc.utoronto.ca/~butler/c32/heat.csv`.

   The word "warning" is sometimes used in place of "alert" in these data. They mean the same thing.[21]

   (a) Read the data into a SAS data set. There are four columns, as follows:

   - a numerical `id` (numbered upwards from the first Heat Alert in 2001; some of the numbers are missing)
   - the `date` of the heat alert, in year-month-day format with 4-digit years.
   - a text `code` for the type of heat alert
   - `text` describing the kind of heat alert. Read the first 60 characters of this text.

   There are 200 heat-alert days in total. Show the code you used to read in the data. (Listing out the data is in the next part.)

Solution: Here we go:

```
filename myurl url "http://www.utsc.utoronto.ca/~butler/c32/heat.csv";

proc import
  datafile=myurl
  dbms=csv
  out=heat
  replace;
  getnames=yes;
```

   (b) Display the first 10 lines of your data set, *with the dates formatted in day, month, year order*. (Hint: until you have sorted out how to do (a), display all the lines of data, to make sure you have all 200. When that is correct, format and display just the first ten lines. Since the text of the type of heat alert is long, the output may come in two blocks.)

Solution: This is `proc print` with the right `format`, `ddmmyy10.`, for the date (or else it will display as whatever SAS read it in as). Displaying the first 10 rows is the same thing you used a couple of assignments ago:

```
proc print data=heat(obs=10);
```

```
format date ddmmyy10.;
```

```
Obs           id          date    code

  1          232    08/09/2016    HAU
  2          231    07/09/2016    HAE
  3          230    06/09/2016    HA
  4          228    13/08/2016    EHAE
  5          227    12/08/2016    EHAE
  6          226    11/08/2016    HAU
  7          225    10/08/2016    HAE
  8          224    09/08/2016    HA
  9          222    05/08/2016    HAE
 10          221    04/08/2016    HA

Obs text

  1 Toronto's Medical Officer of Health has upgraded the Heat Warning to an Exte
  2 Toronto's Medical Officer of Health has continued the Heat Warning for today
  3 Toronto's Medical Officer of Health has issued a Heat Warning
  4 Toronto's Medical Officer of Health has continued the Extended Heat Warning
  5 Toronto's Medical Officer of Health has continued the Extended Heat Warning
  6 Toronto's Medical Officer of Health has upgraded the Heat Warning to an Exte
  7 Toronto's Medical Officer of Health has continued the Heat Warning for today
  8 Toronto's Medical Officer of Health has issued a Heat Warning
  9 Toronto's Medical Officer of Health has continued the Heat Warning for today
 10 Toronto's Medical Officer of Health has issued a Heat Warning
```

The dates are in reverse order, with the most recent at the top (at the time I collected the data; there have evidently been more heat alerts since then).

I previously checked, and I did indeed have all 200 rows. The text has been cut off (after some number of characters). Some of the text is longer than that. This is the `guessingrows` thing I mentioned eleswhere: it looks at the first (by default) 20 lines, and finds the longest piece of text out of those, and then any longer texts it finds later in the file get cut off after that length.

The nice thing about `proc import` is that this happens behind the scenes and we don't have to worry about it too much. Knowing what is actually going on, however, is a plus, since then we can understand things like why text is getting cut off.

(c) SAS has a procedure called `proc freq` that counts up how many observations are in each category of a categorical variable. After the `proc freq` line, you have lines that say `tables` and the name of a (categorical) variable.[22] How many different codes are there in this data set, and what codes are they?

Solution: Following the hints:

```
proc freq;
  tables code;
```

with output

```
                          The FREQ Procedure

                                                 Cumulative    Cumulative
code     Frequency     Percent     Frequency      Percent
-----------------------------------------------------------
EHA             59       29.50            59        29.50
EHAD             1        0.50            60        30.00
EHAE            18        9.00            78        39.00
HA              93       46.50           171        85.50
HAE             16        8.00           187        93.50
HAU             13        6.50           200       100.00
```

Six codes, EHA, EHAD, EHAE, HA, HAE, HAU.

(d) Use the `text` in your dataset (or look back at the original data file) to describe briefly in your own words what the various codes represent.

Solution: You can check that each time a certain code appears, the text next to it is identical.

The six codes and my brief descriptions are:

EHA (Start of) Extended Heat Alert

EHAD Extreme Heat Alert downgraded to Heat Alert

EHAE Extended Heat Alert continues

HA (Start of) Heat Alert

HAE Heat Alert continues

HAU Heat Alert upgraded to Extended Heat Alert

I thought there was such a thing as an Extreme Heat Alert, but here the word is (usually) Extended, meaning a heat alert that extends over several days, long in duration rather than extremely hot. The only place Extreme occurs is in EHAD, which only occurs once.

I want your answer to say or suggest something about whether a code applies *only* to continuing heat alerts (ie., that EHAD, EHAE, HAE and HAU are different from the others).

(e) How many (regular and extended) heat alert events are there altogether? A heat alert event is a stretch of consecutive days, on all of which there is a heat alert or extended heat alert. Hints: (i) you can answer this from output you already have; (ii) how can you tell when a heat alert event *starts*?

Solution: This turned out to be more messed-up than I thought. There is a detailed discussion below.

The codes EHAD, EHAE, HAE, HAU all indicate that there was a heat alert on the day before. Only the codes HA and EHA can indicate the start of a heat alert (event). The problem is that HA and EHA sometimes indicate the start of a heat alert event and sometimes one that is continuing. You can check by looking at the data that HA and EHA days can (though they don't always: see below) have a non-heat-alert day before (below) them in the data file: for example, August 4, 2012 is an HA day, but August 3 of that year was not part of any kind of heat alert.

I had intended the answer to be this:

> So we get the total number of heat alert events by totalling up the number of HA and EHA days: $59 + 93 = 152$.

This is not right because there are some consecutive EHA days, eg. 5–8 July 2010, so that EHA sometimes indicates the continuation of an extended heat alert and sometimes the start of one. I was expecting EHA to be used only for the start, and one of the other codes to indicate a continuation. The same is (sometimes) true of HA.

So reasonable answers to the question as set include:

- 93, the number of HAs

- 59, the number of EHAs
- 152, the number of HAs and EHAs combined
- "152 or less", "between 93 and 152", "between 59 and 152" to reflect that not all of these mark the start of a heat alert event.

Any of these, or something similar *with an explanation of how you got your answer*, are acceptable. In your career as a data scientist, you will often run into this kind of thing, and it will be your job to do something with the data and *explain* what you did so that somebody else can decide whether they believe you or not. A good explanation, even if it is not correct, will help you get at the truth because it will inspire someone to say "in fact, it goes *this* way", and then the two of you can jointly figure out what's actually going on.

Detailed discussion follows. If you have *any* ambitions of working with data, you should try to follow the paragraphs below, because they indicate how you would get an *actual* answer to the question. I used R and `dplyr`, because that seemed to be the easiest route.[23]

I think the key is the number of days between one heat alert day and the next one. `dplyr` has a function `diff` that works out exactly this. Building a pipe, just because:

```
my_url="http://www.utsc.utoronto.ca/~butler/c32/heat.csv"
read_csv(my_url) %>%
select(-text) %>%
mutate(daycount=as.numeric(date)) %>%
mutate(daydiff=abs(c(diff(daycount),0)))
```

```
## Parsed with column specification:
## cols(
##   id = col_integer(),
##   date = col_date(format = ""),
##   code = col_character(),
##   text = col_character()
## )
```

```
## # A tibble: 200 x 5
##       id date       code  daycount daydiff
##    <int> <date>     <chr>    <dbl>   <dbl>
##  1   232 2016-09-08 HAU      17052       1
##  2   231 2016-09-07 HAE      17051       1
##  3   230 2016-09-06 HA       17050      24
##  4   228 2016-08-13 EHAE     17026       1
##  5   227 2016-08-12 EHAE     17025       1
##  6   226 2016-08-11 HAU      17024       1
##  7   225 2016-08-10 HAE      17023       1
##  8   224 2016-08-09 HA       17022       4
##  9   222 2016-08-05 HAE      17018       1
## 10   221 2016-08-04 HA       17017      11
## # ... with 190 more rows
```

Oof. I have some things to keep track of here:

- Get rid of the `text`, since it serves no purpose here.
- The `date` column is a proper `Date` (I checked).

- Then I want the date as number of days; since it is a number of days internally, I just make it a number with `as.numeric`.
- Then I use `diff` to get the difference between each date and the previous one, remembering to glue a 0 onto the end so that I have the right number of differences.
- Since the dates are most recent first, I take the absolute value so that the `daydiff` values are positive (except for the one that is 0 on the end).

Still with me? All right. You can check that the `daydiff` values are the number of days between the date on that line and the line below it. For example, there were 24 days between August 13 and September 6.

Now, when `daydiff` is 1, there was also a heat alert on the previous day (the line below in the file), but when `daydiff` is *not* 1, that day must have been the *start* of a heat alert event. So if I count the non-1's, that will count the number of heat alert events there were. (That includes the difference of 0 on the first day, the one at the end of the file.)

Thus my pipe continues like this:

```
my_url="http://www.utsc.utoronto.ca/~butler/c32/heat.csv"
read_csv(my_url) %>%
  select(-text) %>%
  mutate(daycount=as.numeric(date)) %>%
  mutate(daydiff=abs(c(diff(daycount),0))) %>%
  count(daydiff!=1)
```

```
## Parsed with column specification:
## cols(
##   id = col_integer(),
##   date = col_date(format = ""),
##   code = col_character(),
##   text = col_character()
## )
```

```
## # A tibble: 2 x 2
##   `daydiff != 1`     n
##   <lgl>          <int>
## 1 FALSE            121
## 2 TRUE              79
```

And that's how many actual heat alert events there were: 79, less even than the number of HAs. So that tells me that a lot of my HAs and EHAs were actually continuations of heat alert events rather than the start of them. I think I need to have a word with the City of Toronto about their data collection processes.

`count` will count anything that is, or can be made into, a categorical variable. It doesn't have to be one of the columns of your data frame; here it is something that is either `TRUE` or `FALSE` about every row of the data frame.

One step further: what *is* the connection between the codes and the start of heat alert events? We can figure that out now:

```
my_url="http://www.utsc.utoronto.ca/~butler/c32/heat.csv"
read_csv(my_url) %>%
  select(-text) %>%
  mutate(daycount=as.numeric(date)) %>%
  mutate(daydiff=abs(c(diff(daycount),0))) %>%
  mutate(start=(daydiff!=1)) %>%
  count(code,start)
```

```
## Parsed with column specification:
## cols(
##   id = col_integer(),
##   date = col_date(format = ""),
##   code = col_character(),
##   text = col_character()
## )
```

```
## # A tibble: 8 x 3
##   code  start     n
##   <chr> <lgl> <int>
## 1 EHA   FALSE    53
## 2 EHA   TRUE      6
## 3 EHAD  FALSE     1
## 4 EHAE  FALSE    18
## 5 HA    FALSE    20
## 6 HA    TRUE     73
## 7 HAE   FALSE    16
## 8 HAU   FALSE    13
```

I made a column `start` that is `TRUE` at the start of a heat alert event and `FALSE` otherwise, by comparing the days from the previous heat alert day with 1. Then I can make a `table`, or, as here, the `dplyr` equivalent with `count`[24] (or `group_by` and `summarize`). What this shows is that EHAD, EHAE, HAE and HAU *never* go with the start of a heat alert event (as they shouldn't). But look at the HAs and EHAs. For the HAs, 73 of them go with the start of an event, but 20 do not. For the EHAs, just 6 of them go with the start, and 53 do not. (Thus, counting just the HAs was very much a reasonable thing to do.)

The 79 heat alert events that we found above had 73 of them starting with an HA, and just 6 starting with an EHA.

I wasn't quite sure how this would come out, but I knew it had something to do with the number of days between one heat alert day and the next, so I calculated those first and then figured out what to do with them.

(f) We are going to investigate how many heat alert days there were in each year. To do that, we have to extract the year from each of our dates. SAS has a function `year()` that can be used in a `data` step to extract the year from a date. Create a new data set containing all the variables from the previous one plus a variable `y` that contains the year of each date.

Solution: The hints suggest this:

```
data heat2;
  set heat;
  y=year(date);
```

Did that work? Let's check by listing a few values from `heat2`, skipping `text`. I think we can avoid `format`ting the date, since `proc import` read it in in some format, that you can check by looking in the log tab.

```
proc print data=heat2(obs=40);
  var id date code y;
```

Obs	id	date	code	y
1	232	2016-09-08	HAU	2016
2	231	2016-09-07	HAE	2016
3	230	2016-09-06	HA	2016
4	228	2016-08-13	EHAE	2016
5	227	2016-08-12	EHAE	2016
6	226	2016-08-11	HAU	2016
7	225	2016-08-10	HAE	2016
8	224	2016-08-09	HA	2016
9	222	2016-08-05	HAE	2016
10	221	2016-08-04	HA	2016
11	219	2016-07-24	EHAE	2016
12	218	2016-07-23	HAU	2016
13	217	2016-07-22	HAE	2016
14	216	2016-07-21	HA	2016
15	214	2016-07-14	HAU	2016
16	213	2016-07-13	HAE	2016
17	212	2016-07-12	HA	2016
18	210	2016-07-07	EHA	2016
19	209	2016-07-06	HAE	2016
20	208	2016-07-05	HA	2016
21	206	2016-06-20	HAE	2016
22	205	2016-06-19	HA	2016
23	203	2015-09-08	EHAE	2015
24	202	2015-09-07	HAU	2015
25	201	2015-09-06	HAE	2015
26	200	2015-09-05	HA	2015
27	198	2015-08-17	HAE	2015
28	197	2015-08-16	HA	2015
29	195	2015-07-30	EHAE	2015
30	194	2015-07-29	HAU	2015
31	193	2015-07-28	HAE	2015
32	192	2015-07-27	HA	2015
33	190	2015-07-19	HAE	2015
34	189	2015-07-18	HA	2015
35	187	2014-09-05	HA	2014
36	185	2013-09-11	HAE	2013
37	184	2013-09-10	HA	2013
38	182	2013-07-19	EHAE	2013
39	181	2013-07-18	EHAE	2013
40	180	2013-07-17	EHAE	2013

That seems to have worked. I listed a lot of rows to get back to previous years, since the 2016 values might have been gotten right by chance and I wanted some others to check as well.

(g) Count the number of heat alert days for each year, by tabulating the year variable. Looking at this

table, would you say that there have been more heat alert days in recent years? Explain (very) briefly.

Solution:

```
proc freq;
  tables y;
```

The FREQ Procedure

y	Frequency	Percent	Cumulative Frequency	Cumulative Percent
2001	9	4.50	9	4.50
2002	16	8.00	25	12.50
2003	6	3.00	31	15.50
2004	2	1.00	33	16.50
2005	26	13.00	59	29.50
2006	17	8.50	76	38.00
2007	15	7.50	91	45.50
2008	9	4.50	100	50.00
2009	3	1.50	103	51.50
2010	16	8.00	119	59.50
2011	12	6.00	131	65.50
2012	21	10.50	152	76.00
2013	13	6.50	165	82.50
2014	1	0.50	166	83.00
2015	12	6.00	178	89.00
2016	22	11.00	200	100.00

There are various things you could say, most of which are likely to be good. My immediate reaction is that most of the years with a lot of heat-alert days are in the last few years, and most of the years with not many are near the start, so there is something of an upward trend. Having said that, 2014 is unusually low (that was a cool summer, if you recall), and 2005 was unusually high. (Was that the summer of the big power outage? I forget.[25])

You could also reasonably say that there isn't much pattern: the number of heat-alert days goes up and down. In fact, anything that's not obviously nonsense will do.

I was thinking about making a graph of these frequencies against year, and sticking some kind of smooth trend on it. The SAS way to pull data off output is to create an output data set, which I discovered was possible, and not that hard:

```
proc freq;
  tables y / out=byyear;
```

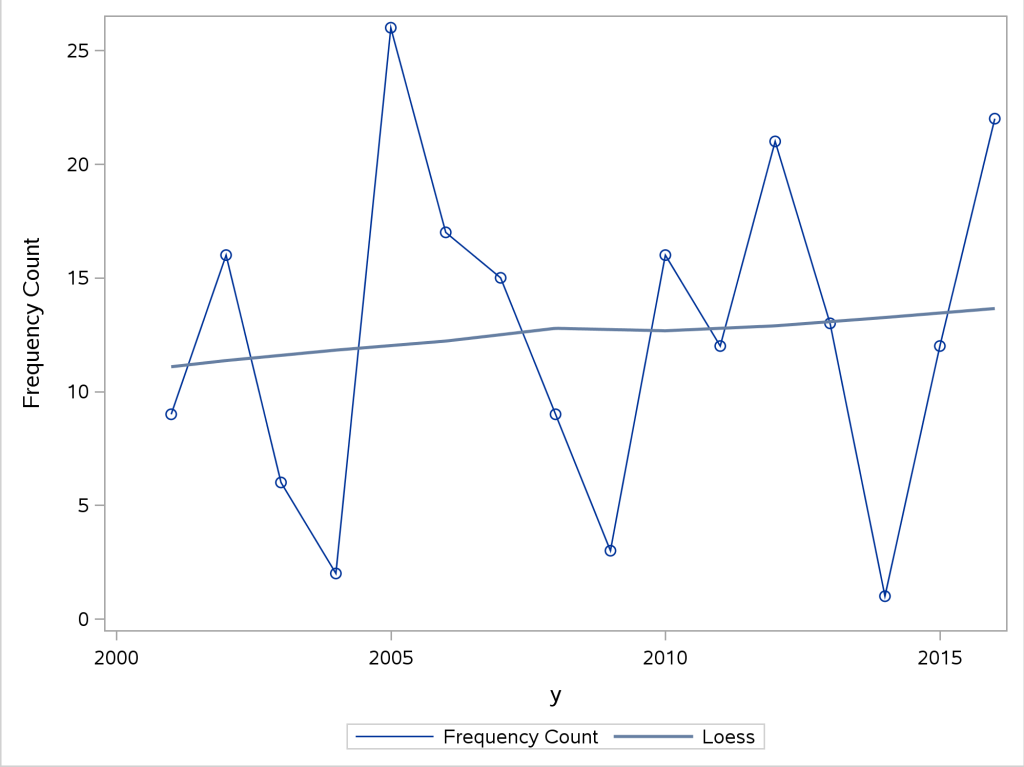
I don't need to look at the output from this, since I only ran `proc freq` to get the output data set, which I look at now. Note that this is the most recently created data set, so `proc print` will display it:

```
proc print;
```

Obs	y	COUNT	PERCENT
1	2001	9	4.5
2	2002	16	8.0
3	2003	6	3.0
4	2004	2	1.0
5	2005	26	13.0
6	2006	17	8.5
7	2007	15	7.5
8	2008	9	4.5
9	2009	3	1.5
10	2010	16	8.0
11	2011	12	6.0
12	2012	21	10.5
13	2013	13	6.5
14	2014	1	0.5
15	2015	12	6.0
16	2016	22	11.0

We have variables `y`, the year, and `count`, the frequency (case insensitive, this being SAS). So we can make a plot like this. I'm using `series` since this is data over time:

```
proc sgplot;
  series x=y y=count;
  loess x=y y=count;
```



I realized that I should not have called my year variable `y` since it is actually the x-coordinate of the plot! The `loess` thing does a smooth trend like `geom_smooth` does in `ggplot`.

The pattern is very scattered, as is commonly the case with environmental-science-type data, but there is a very small upward trend. So it seems that either answer is justified, either "there is no trend" or "there is something of an upward trend".

The other thing I notice on this plot is that if there are a lot of heat-alert days one year, there will probably also be a lot in the next year (and correspondingly if the number of heat-alert days is below average: it tends to be below average again in the next year). This pattern is known to time-series people as "autocorrelation" and indicates that the number of heat-alert days in one year and the next is not independent: if you know one year, you can predict the next year. (Assessment of trend and autocorrelation are hard to untangle properly.)

I learn from Environmental Science grad students (of whom we have a number at UTSC) that the approved measure of association is called the Mann-Kendall correlation, which is the Kendall correlation of the data values with time. In the same way that we use the sign test when we doubt normality, and it uses the data more crudely but safely, the regular (so-called Pearson) correlation assumes normality (of the errors in the regression of one variable on the other), and when you doubt that (as you typically do with this kind of data) you compute a different kind of correlation with time. What the Kendall correlation does is to take each pair of observations and ask whether the trend with time is uphill or downhill. For example, there were 3 heat-alert days in 2009, 16 in 2010 and 12 in 2011. Between 2009 and 2010, the trend is uphill (increasing with time), and also between 2009 and 2011 (there were more heat-alert days in the later year), but between 2010 and 2011 the trend is downhill. The idea of the Kendall correlation is you take *all* the pairs of points, of which there are typically rather a lot, count up how many pairs are uphill and how many downhill, and apply a formula to get a correlation between -1 and 1. (If there are about an equal number of uphills and downhills, the correlation comes out near 0; if they are mostly uphill, the correlation is near 1, and if they are mostly downhill, the correlation is near -1.) It doesn't matter *how* uphill or downhill the trends are, only the number of each, in the same way that the sign test only counts the *number* of values above or below the hypothesized median, not how far above or below they are.

SAS will compute the Kendall correlation, via `proc corr`:

```
proc corr kendall;
  var y count;
```

```
                          The CORR Procedure

                    2  Variables:    y        COUNT

                          Simple Statistics

Variable         N        Mean     Std Dev      Median     Minimum     Maximum

y               16        2009     4.76095        2009        2001        2016
COUNT           16    12.50000     7.28469    12.50000     1.00000    26.00000

                          Simple Statistics

                          Variable  Label

                          y
                          COUNT     Frequency Count
```

```
Kendall Tau b Correlation Coefficients, N = 16
         Prob > |tau| under H0: Tau=0

                                    y          COUNT

  y                         1.00000        0.05908
                                           0.7519

  COUNT                     0.05908        1.00000
  Frequency Count            0.7519
```

The Mann-Kendall correlation is a thoroughly unremarkable 0.06, and with only 16 data points, a null hypothesis that the correlation is zero is far from being rejected, P-value 0.7519 as shown. So this is no evidence of a time trend at all.

I'd like to say a word about how I got these data. They came from `http://app.toronto.ca/opendata/heat_alerts/heat_alerts_list.json`. If you take a look there, there are no obvious rows and columns. This format is called JSON. Look a bit more carefully and you'll see stuff like this, repeated:

```
{"id":"232","date":"2016-09-08","code":"HAU",
"text":"Toronto's Medical Officer of Health has upgraded the Heat Warning to an Extended Heat
```

one for each heat alert day. These are "keys" (on the left side of the `:`) and "values" (on the right side).[26] The keys are column headers (if the data were in a data frame) and the values are the data values that would be in that column. In JSON generally, there's no need for the keys to be the same in every row, but if they are, as they are here, the data can be arranged in a data frame. How? Read on.

I did this in R, using a package called `jsonlite`, with this code:

```
library(jsonlite)
url="http://app.toronto.ca/opendata/heat_alerts/heat_alerts_list.json"
heat=fromJSON(url,simplifyDataFrame = T)
head(heat)
write_csv(heat,"heat.csv")
```

After loading the package, I create a variable `url` that contains the URL for the JSON file. The `fromJSON` line takes something that is JSON (which could be text, a file or a URL) and converts it to and saves it in a data frame. Finally, I save the data frame in a `.csv` file. That's the `.csv` file you used in SAS. If you run that code, you'll get a `.csv` file of heat alerts right up to the present, and you can update my analysis.

Why `.csv`? If I had used `write_delim`, the values would have been separated by spaces. *But*, the `text` is a sentence of several words, which are themselves separated by spaces. I could have had you read everything else into SAS and not the text, and then separated-by-spaces would have been fine, but I wanted you to see the text so that you could understand the `code` values, and (to be honest) I wanted to give you practice at reading in some long strings of characters. So `.csv` is what it was.

Notes

[1] As str_c or paste do, *actually*, *but the advantage of* unite *is that it gets rid of the other columns*, *which you probably no longer need*.

[2]You could just as well make the point that the text 20.8 contains the number 20.8 and nothing else, so that `parse_number` will pull out 20.8 as a number. If that logic works for you, go with it.

[3]You might think that missing is just missing, but R distinguishes between types of missing.

[4]This was the actual reason I gave you this question: I wanted you to do this. It sort of morphed into all the other stuff as well.

[5]And it shows the value of looking at relevant plots.

[6]Mistakenly.

[7]Mixes up.

[8]This is actually grammatically correct.

[9]Though it's hard to imagine being able to improve on an R-squared of 99%.

[10]This wouldn't have told us about the overall effect of `species`.

[11]In fact, it is believed that wolves help keep caribou herds strong by preventing over-population: that is, the weakest caribou are the ones taken by wolves.

[12]The survey is always taken in the fall, but the date varies.

[13]Counting animals in a region, especially rare, hard-to-find animals, is a whole science in itself. These numbers are probably estimates (with some uncertainty).

[14]You are probably too young to remember that. Everyone thought the world was going to end when all the computers went to year `00` and we thought they would interpret it as 1900.

[15]Because the wolves have more to eat.

[16]Baby wolves.

[17]Ken makes a rude noise.

[18]In some languages it is called `switch`. Python appears not to have it.

[19]If I was helping you, and you were struggling with `ifelse` but finally mastered it, it seemed easier to suggest that you used it again for the others.

[20]But I didn't want to complicate this question any farther.

[21]Unlike "thunderstorm watch" and "thunderstorm warning", which mean different things.

[22]If you have two or more variables that you want cross-tabulations of, you separate them with `*` and you get a two-way or multi-way table.

[23]I think it is possible to follow the same kind of procedure in SAS, now that I've worked out my approach, but I found that it was much easier to figure out what to do in R.

[24]I didn't know until just now that you could put two variables in a `count` and you get counts of all the combinations of them. Just goes to show the value of "try it and see".

[25]I looked it up. It was 2003, my first summer in Ontario.

[26]This is the same kind of thing as a "dictionary" in Python.