Assignment 8.7

Nothing to hand in

The usual beginning:

```
library(tidyverse)
```

1. Let's write some simple R functions to convert temperatures, and later to play with text.

   (a) A temperature in Celsius is converted to one in Kelvin by adding 273.15. (-273.15 Celsius, 0 Kelvin, is the "absolute zero" temperature that nothing can be colder than.) Write a function called `c_to_k` that converts an input Celsius temperature to one in Kelvin, and test that it works.

Solution: This is mostly an exercise in structuring your function correctly. Let's call the input `C` (uppercase C, since lowercase c has a special meaning to R):

```
c_to_k=function(C) {
    C+273.15
}
c_to_k(0)
```

```
## [1] 273.15
```

```
c_to_k(20)
```

```
## [1] 293.15
```

This is the simplest way to do it: the last line of the function, if calculated but not saved, is the value that gets returned to the outside world. The checks suggest that it worked.

If you're used to Python or similar, you might prefer to calculate the value to be returned and then return it. You can do that in R too:

```
c_to_k=function(C) {
    K=C+273.15
    return(K)
}
c_to_k(0)
```

```
## [1] 273.15
```

```
c_to_k(20)
```

```
## [1] 293.15
```

That works just as well, and for the rest of this question, you can go either way.

   (b) Write a function to convert a Fahrenheit temperature to Celsius. The way you do that is to subtract 32 and then multiply by 5/9.

Solution: On the model of the previous one, we should call this `f_to_c`. I'm going to return the last line, but you can save the calculated value and return that instead:

```
f_to_c=function(F) {
    (F-32)*5/9
}
f_to_c(32)

## [1] 0

f_to_c(50)

## [1] 10

f_to_c(68)

## [1] 20
```

Americans are very good at saying things like "temperatures in the 50s", which don't mean much to me, so I like to have benchmarks to work with: these are the Fahrenheit versions of 0, 10, and 20 Celsius.

Thus "in the 50s" means "between about 10 and 15 Celsius".

(c) *Using the functions you already wrote*, write a function to convert an input Fahrenheit temperature to Kelvin.

Solution: This implies that you can piggy-back on the functions you just wrote, which goes as below. First you convert the Fahrenheit to Celsius, and then you convert *that* to Kelvin. (This is less error-prone than trying to use algebra to get a formula for this conversion and then implementing that):

```
f_to_k=function(F) {
    C=f_to_c(F)
    K=c_to_k(C)
    return(K)
}
f_to_k(32)

## [1] 273.15

f_to_k(68)

## [1] 293.15
```

These check because in Celsius they are 0 and 20 and we found the Kelvin equivalents of those to be these values earlier.

I wrote this one with a **`return`** because I thought it made the structure clearer: run one function, save the result, run another function, save the result, then return what you've got.

(d) Rewrite your Fahrenheit-to-Celsius convertor to take a suitable default value and check that it works as a default.

Solution: You can choose any default you like. I'll take a default of 68 (what I would call "a nice day"):

```
f_to_c=function(F=68) {
    (F-32)*5/9
}
f_to_c(68)
```

```
## [1] 20
```

```
f_to_c()
```

```
## [1] 20
```

The change is in the top line of the function. You see the result: if we run it without an input, we get the same answer as if the input had been 68.

(e) What happens if you feed your Fahrenheit-to-Celsius convertor a *vector* of Fahrenheit temperatures? What if you use it in a `mutate`?

Solution: Try it and see:

```
temps=seq(30,80,10)
temps
```

```
## [1] 30 40 50 60 70 80
```

```
f_to_c(temps)
```

```
## [1] -1.111111  4.444444 10.000000 15.555556 21.111111 26.666667
```

Each of the Fahrenheit temperatures gets converted into a Celsius one. This is perhaps more useful in a data frame, thus:

```
tibble(temps=seq(30,80,10)) %>%
  mutate(celsius=f_to_c(temps))
```

```
## # A tibble: 6 x 2
##   temps   celsius
##   <dbl>     <dbl>
## 1    30 -1.111111
## 2    40  4.444444
## 3    50 10.000000
## 4    60 15.555556
## 5    70 21.111111
## 6    80 26.666667
```

All the temperatures are side-by-side with their equivalents.

(f) Write another function called `wrap` that takes two arguments: a piece of text called `text`, which defaults to `hello`, and another piece of text called `outside`, which defaults to `*`. The function returns `text` with the text `outside` placed before and after, so that calling the function with the defaults should return `*hello*`. To do this, you can use `str_c` from `stringr` (loaded with the `tidyverse`) which places its text arguments side by side and glues them together into one piece of text. Test your function briefly.

Solution: This:

```
wrap=function(text="hello",outside="*") {
    str_c(outside,text,outside)
}
```

I can run this with the defaults:

```
wrap()
```

```
## [1] "*hello*"
```

or with text of my choosing:

```
wrap("goodbye","_")
```

```
## [1] "_goodbye_"
```

I think that's what I meant by "test briefly".

(g) What happens if you want to change the default `outside` but use the default for `text`? How do you make sure that happens? Explore.

Solution: The obvious thing is this, which doesn't work:

```
wrap("!")
```

```
## [1] "*!*"
```

This takes *`text`* to be `!`, and `outside` to be the default. How do we get `outside` to be `!` instead? The key is to specify the input by name:

```
wrap(outside="!")
```

```
## [1] "!hello!"
```

This correctly uses the default for `text`.

If you specify inputs without names, they are taken to be in the order that they appear in the function definition. As soon as they get out of order, which typically happens by using the default for something early in the list, as we did here for `text`, you have to specify names for anything that comes after that. These are the names you put on the function's top line.

You can always use names:

```
wrap(text="thing",outside="**")
```

```
## [1] "**thing**"
```

and if you use names, they don't even have to be in order:

```
wrap(outside="!?",text="fred")
```

```
## [1] "!?fred!?"
```

(h) What happens if you feed your function `wrap` a vector for either of its arguments? What about if you use it in a `mutate`?

Solution: Let's try:

```
mytext=c("a","b","c")
wrap(text=mytext)
```

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

```
myout=c("*","!")
wrap(outside=myout)
```

```
## [1] "*hello*" "!hello!"
```

If one of the inputs is a vector, the other one gets "recycled" as many times as the vector is long. What if they're both vectors?

```
mytext2=c("a","b","c","d")
wrap(mytext2,myout)
```

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

This uses the two inputs in parallel, repeating the short one as needed. But this, though it works, gives a warning:

```
wrap(mytext,myout)
```

```
## Warning in stri_c(..., sep = sep, collapse = collapse, ignore_null = TRUE): longer
object length is not a multiple of shorter object length
```

```
## [1] "*a*" "!b!" "*c*"
```

This is because the shorter vector (of length 2 here) doesn't go evenly into the longer one (length 3). It gives a warning because this is probably not what you wanted.

The `mutate` thing is easier, because all the columns in a data frame have to be the same length. `LETTERS` is a vector with the uppercase letters in it:

```
tibble(mytext=LETTERS[1:6],myout=c("*","**","!","!!","_","__")) %>%
  mutate(newthing=wrap(mytext,myout))
```

```
## # A tibble: 6 x 3
##   mytext myout newthing
##    <chr> <chr>    <chr>
## 1      A     *      *A*
## 2      B    **    **B**
## 3      C     !      !C!
## 4      D    !!    !!D!!
## 5      E     _      _E_
## 6      F    __    __F__
```

2. In 2010, a group of students planted some Mizuna lettuce seeds, and recorded how they grew. The data were saved in an Excel spreadsheet, which is at `http://www.utsc.utoronto.ca/~butler/c32/mizuna.xlsx`. The columns are: the date, the height (in cm) of (I presume) the tallest plant, the amount of water added since the previous date (ml), the temperature in the container where the seedlings were growing, and any additional notes that the students made (edited for length by me). The top line of the data file is variable names.

   (a) Read the spreadsheet directly into SAS using `proc import`. That will mean (i) saving the spreadsheet somewhere on your computer (or finding it in your Downloads folder), (ii) uploading it to SAS Studio, (iii) getting the `proc import` right. *I do not want you doing any copying and pasting*

here.

Display the whole of your data set (it is not very big).

(When I did this, the year came out wrong. If that happens to you, ignore it.)

Solution: Once the spreadsheet is in the right place, you'll need code like this:

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

There is only one sheet in the workbook, so you don't have to name it.

Did it work?

```
proc print;
```

Obs	date	height	water	temperature
1	40225	0	400	21
2	40227	0	0	22.5
3	40228	0	200	20.9
4	40231	3.2	100	20.8
5	40232	4.5	100	22.9
6	40234	6	100	21.8
7	40235	6.5	200	21.2
8	40238	9.5	200	21.8
9	40240	11.1	200	21.7
10	40242	13	250	21.9
11	40245	14.5	500	22.5
12	40247	16	200	21.2
13	40254	18.5	800	20.8

Obs	notes
1	planted seeds; water soaked up rapidly
2	2 of 6 seeds not appeared; soil still moist
3	4 of 6 plants broken surface
4	Last seed hasnt broken surface
5	Plants growing well.
6	Last seed sprouted; plants looking green and healthy
7	
8	
9	Plants needing more water
10	
11	No water left, leaves droopy. Added water, came back to life
12	Plants green and healthy
13	Harvest. Tips of plants turning brown.

Well, almost everything worked: except for the dates, which came out as random-looking integers. Usually, you can go looking in the Log tab to see what formats `proc import` read and displayed the data as, but that seems not to be working for me with Excel spreadsheets. So let's re-do our `proc print` with the dates formatted in some friendly way like this:

```
proc print;
  format date yymmdd10.;
```

Obs	date	height	water	temperature
1	2070-02-17	0	400	21
2	2070-02-19	0	0	22.5
3	2070-02-20	0	200	20.9
4	2070-02-23	3.2	100	20.8
5	2070-02-24	4.5	100	22.9
6	2070-02-26	6	100	21.8
7	2070-02-27	6.5	200	21.2
8	2070-03-02	9.5	200	21.8
9	2070-03-04	11.1	200	21.7
10	2070-03-06	13	250	21.9
11	2070-03-09	14.5	500	22.5
12	2070-03-11	16	200	21.2
13	2070-03-18	18.5	800	20.8

Obs	notes
1	planted seeds; water soaked up rapidly
2	2 of 6 seeds not appeared; soil still moist
3	4 of 6 plants broken surface
4	Last seed hasnt broken surface
5	Plants growing well.
6	Last seed sprouted; plants looking green and healthy
7	
8	
9	Plants needing more water
10	
11	No water left, leaves droopy. Added water, came back to life
12	Plants green and healthy
13	Harvest. Tips of plants turning brown.

OK, now we have dates, but they are *the wrong year!* They should be 2010, not 2070.

(If it worked out right for you, let me know; this might be one of those operating-system things.) I'm going to ignore the fact that the year is wrong, since the month and day is correct.

What actually happened, I think, is that the date as-a-number got read in wrong:

```
proc print;
  var date height water;
```

Obs	date	height	water
1	40225	0	400
2	40227	0	0
3	40228	0	200
4	40231	3.2	100
5	40232	4.5	100
6	40234	6	100
7	40235	6.5	200
8	40238	9.5	200
9	40240	11.1	200
10	40242	13	250
11	40245	14.5	500
12	40247	16	200
13	40254	18.5	800

How many years after 1960[1] are those? The numbers are days, so divide by days in a year:

```
40235/365.25
```

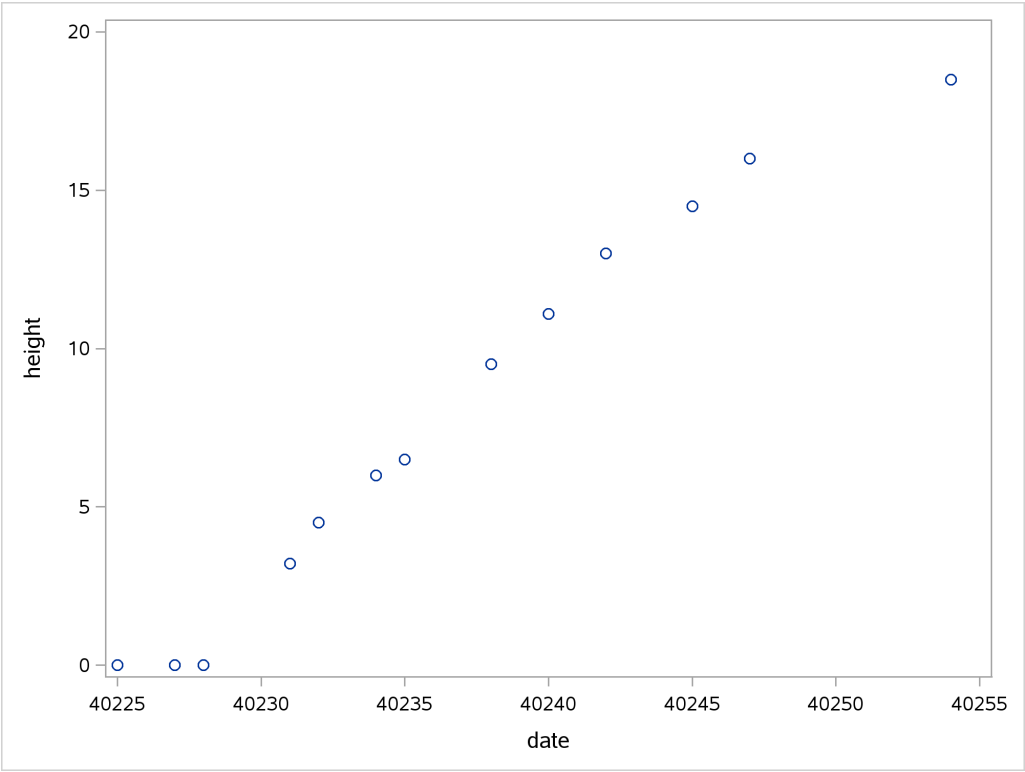
```
## [1] 110.1574
```

110 years after 1960, so 2070 is right. Something went wrong between my entering the numbers in my spreadsheet and them being read into SAS. SAS did the right conversion to go from days-since-1960 to dates, but got the wrong days-since-1960.

(b) Make a suitable plot that shows how the lettuce seeds grow over time.

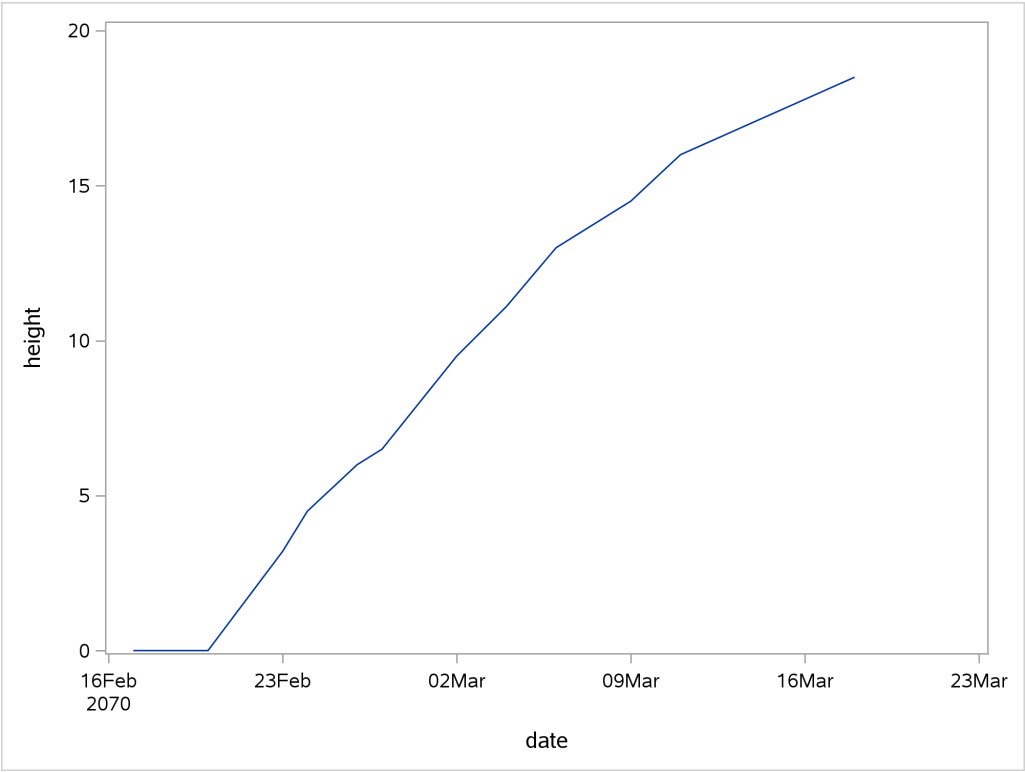
Solution: My first thought is a scatterplot of height against date:

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


20
15
10
5
0
height
40225
40230
40235
40240
40245
40250
40255
date

There are a couple of things I would fix here. One, the date is not shown properly (it is days-since-1960). Two, the plants grow continuously over time, so I would join the points by lines as `series` does. This leads to:

```
proc sgplot;
  series x=date y=height;
  format date yymmdd10.;
```

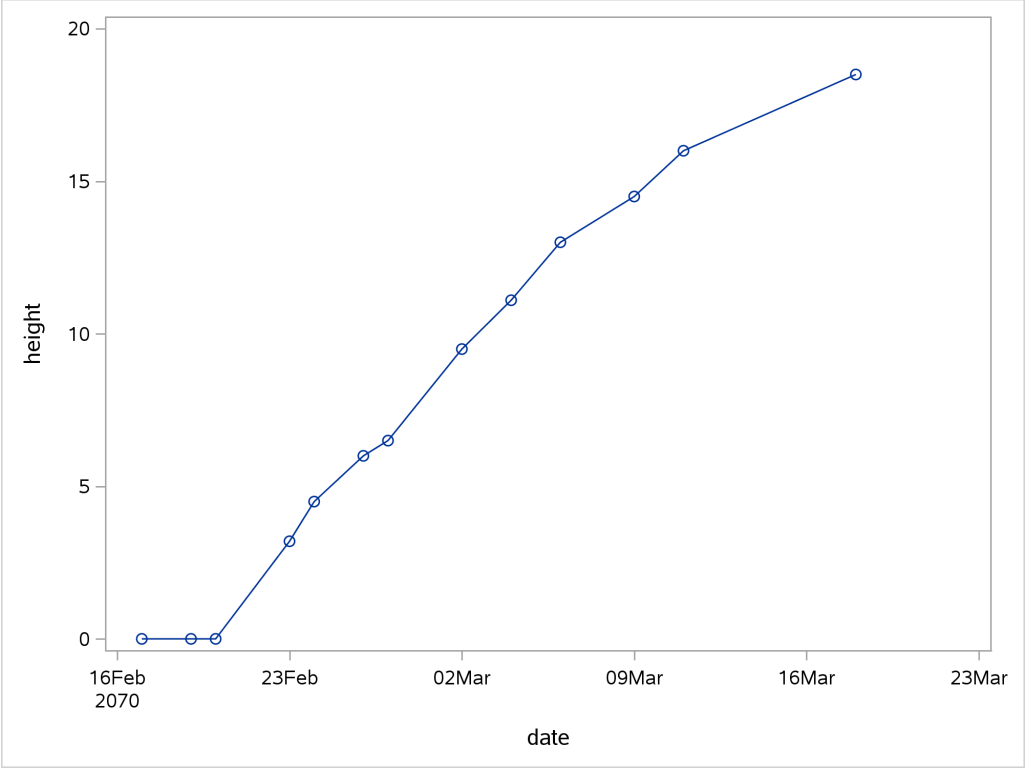

20
15
10
5
0
height
16Feb
2070
23Feb
02Mar
09Mar
16Mar
23Mar
date

Apart from the fact that the year is wrong, everything looks good, and you see that the growth of the plants slows down slightly towards the end (and that there is a delay of a few days at the beginning before any of the seedlings pop above the soil and there is any height to measure).

Note that `proc sgplot` can take a `format` just as `proc print` can.

Here's how to show the data points as well as the lines joining them:

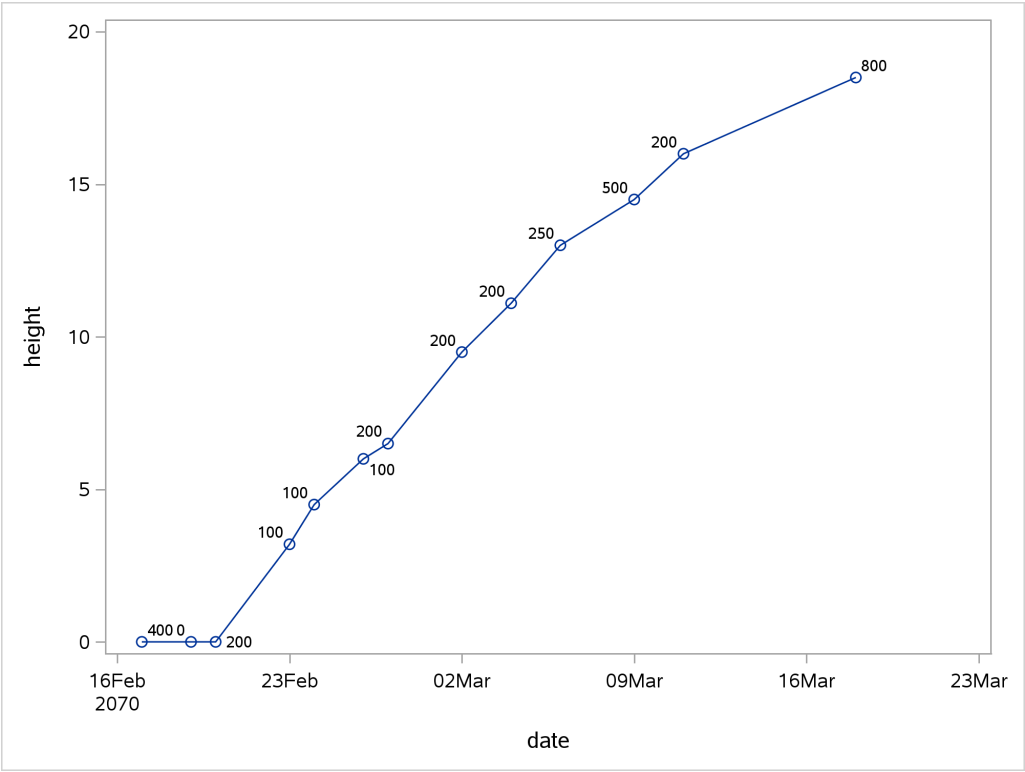
```
proc sgplot;
  series x=date y=height / markers;
  format date yymmdd10.;
```



(c) Label each point with the amount of water given to the plants between the previous time point and this one.

Solution: I had to try to find the words that would lead you towards using `water` rather than making it more complicated than it is. The magic word is `datalabel`. I've left the `markers` in, because it's easier to judge what the labels refer to when you can actually *see* the points. (Try it without the `markers`: is it clear or confusing?)

```
proc sgplot;
  series x=date y=height / markers datalabel=water;
  format date yymmdd10.;
```


20
15
10
5
0
height
400 0
200
100
100
200
100
200
200
250
500
200
800
16Feb
2070
23Feb
02Mar
09Mar
16Mar
23Mar
date

SAS chooses the orientation of the labels so that you can best see them (unlike R, which needs to be told where to put the labels, unless you use something like `ggrepel`).

It occurs to me that the amount of water is over a differing number of days, and that what really matters is how much water was supplied *per day* over the time period in question. That means counting the number of days between each date and the previous one, and working out the water per day. This is what I came up with (making a new data set, since we are making a new variable or two):

```
data miz2;
  set mizuna;
  drop notes;
  datediff=dif(date);
  waterperday=water/datediff;


proc print;
  format date date9.;
proc sgplot;
  series x=date y=height / markers datalabel=waterperday;
  format date yymmdd10.;
```

The new data set is below. I got rid of `notes`; `dif` calculates the difference between each value and the previous one. In this case, you can see that it figured out the number of days between the date on that line and the previous one. (Since the dates are stored as days internally, subtracting them will give a number of days, which is what we want.)

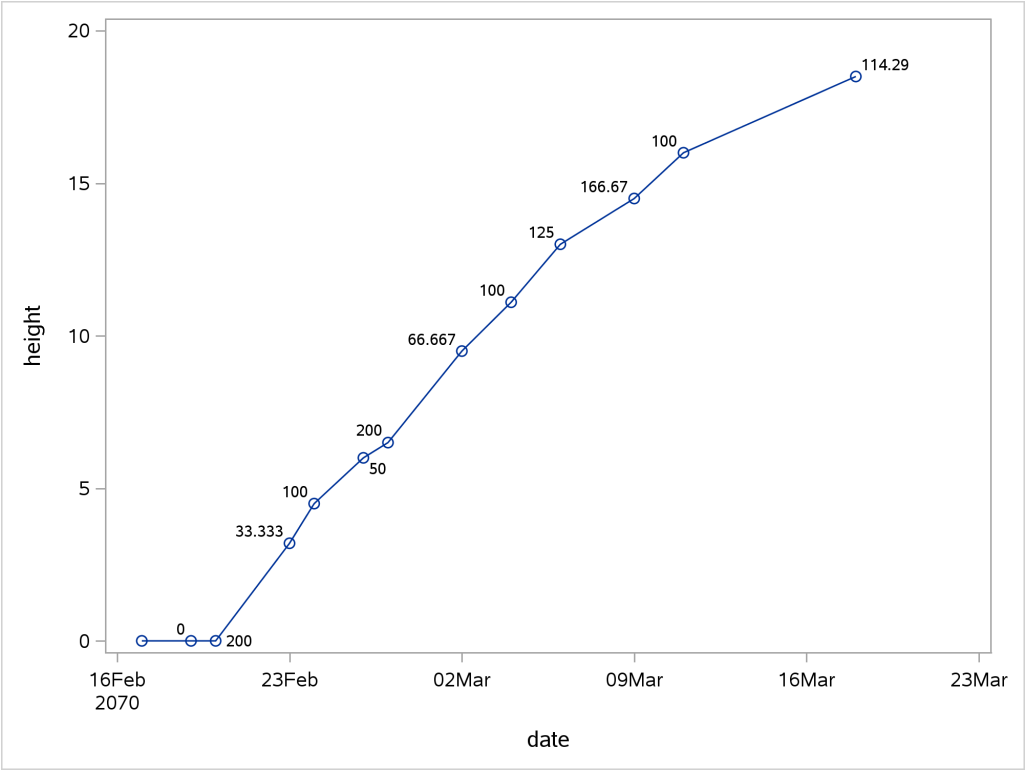
I used a different `format` for the dates, just for fun.

The first date doesn't have a number of days since the previous one (because there *isn't* a previous one), so the first `datediff`, and hence the first `waterperday`, are both missing.

Obs	date	height	water	temperature	datediff	waterperday
1	17FEB2070	0	400	21	.	.
2	19FEB2070	0	0	22.5	2	0.000
3	20FEB2070	0	200	20.9	1	200.000
4	23FEB2070	3.2	100	20.8	3	33.333
5	24FEB2070	4.5	100	22.9	1	100.000
6	26FEB2070	6	100	21.8	2	50.000
7	27FEB2070	6.5	200	21.2	1	200.000
8	02MAR2070	9.5	200	21.8	3	66.667
9	04MAR2070	11.1	200	21.7	2	100.000
10	06MAR2070	13	250	21.9	2	125.000
11	09MAR2070	14.5	500	22.5	3	166.667
12	11MAR2070	16	200	21.2	2	100.000
13	18MAR2070	18.5	800	20.8	7	114.286

You can check that the other `waterperday` values make sense.

The new plot looks like this, with the points labelled by water per day (since the last time the plants were measured):



I don't really see any patterns here. I think they watered the plants when the soil seemed to be getting dry, or perhaps on a Friday before they left for the weekend. Note that SAS has rounded off the water-per-day values.

(d) Now, we repeat the above parts using R. First, read the spreadsheet data using R.

Solution: This is `read_excel` from package `readxl`. I'm not sure what will happen to the dates yet. Note that this needs a "local" copy of the spreadsheet (that is, you have to download it and save it on your computer), possibly using `file.choose` to help R find it. I put my local copy in the same folder as I was working, so I just need the file name:

```
library(readxl)
mizuna=read_excel("mizuna.xlsx")
mizuna
```

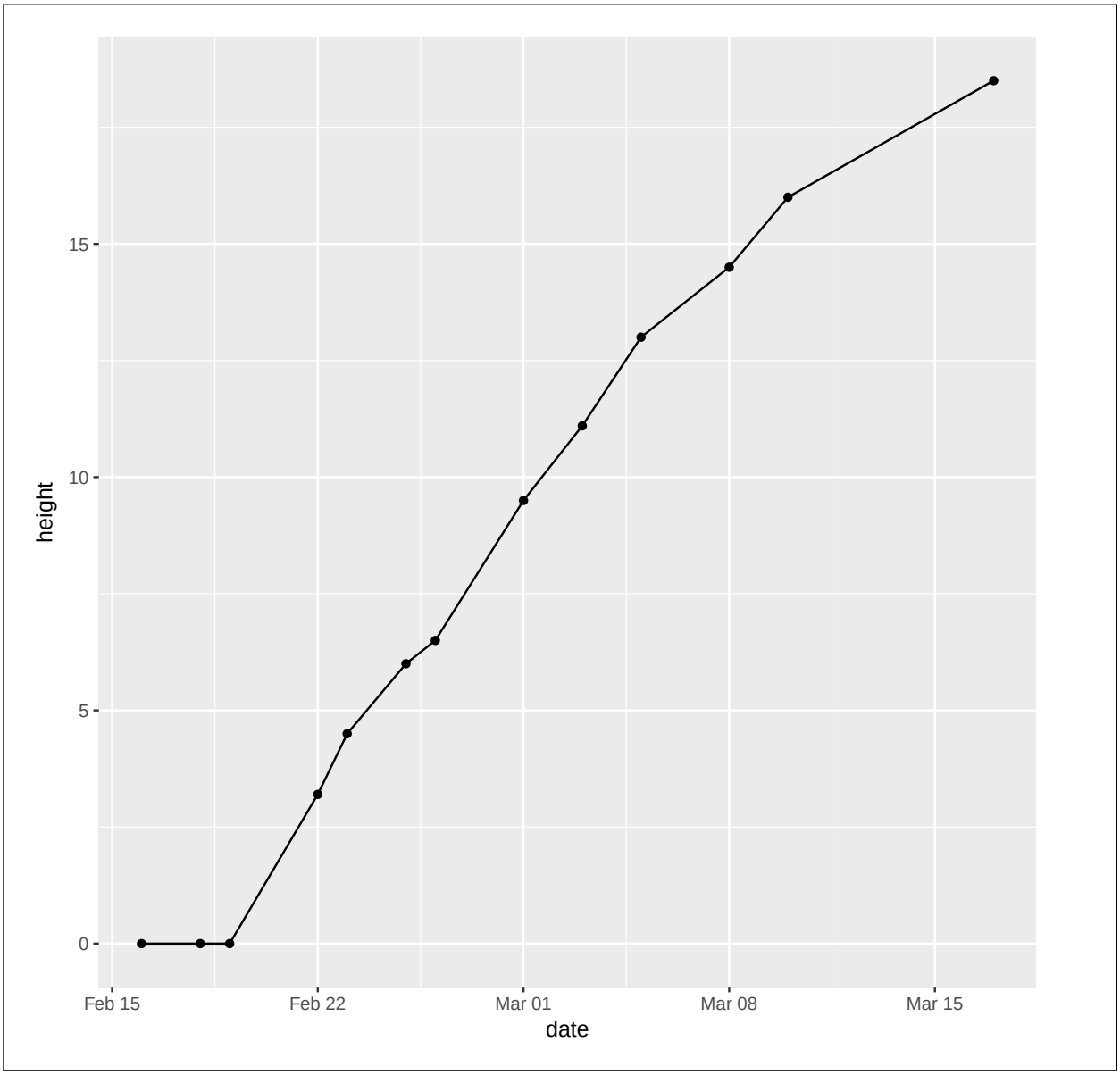
```
## # A tibble: 13 x 5
##          date height water temperature
##        <dttm>  <dbl> <dbl>       <dbl>
##  1 2010-02-16    0.0   400        21.0
##  2 2010-02-18    0.0     0        22.5
##  3 2010-02-19    0.0   200        20.9
##  4 2010-02-22    3.2   100        20.8
##  5 2010-02-23    4.5   100        22.9
##  6 2010-02-25    6.0   100        21.8
##  7 2010-02-26    6.5   200        21.2
##  8 2010-03-01    9.5   200        21.8
##  9 2010-03-03   11.1   200        21.7
## 10 2010-03-05   13.0   250        21.9
## 11 2010-03-08   14.5   500        22.5
## 12 2010-03-10   16.0   200        21.2
## 13 2010-03-17   18.5   800        20.8
## # ... with 1 more variables: notes <chr>
```

The dates *did* get read properly. `dttm` is "date-time", so I guess it's allowing for the possibility that my dates had times attached as well. Unlike with SAS, the years came out right.

(e) Make a plot of `height` against your dates, with the points joined by lines.

Solution:

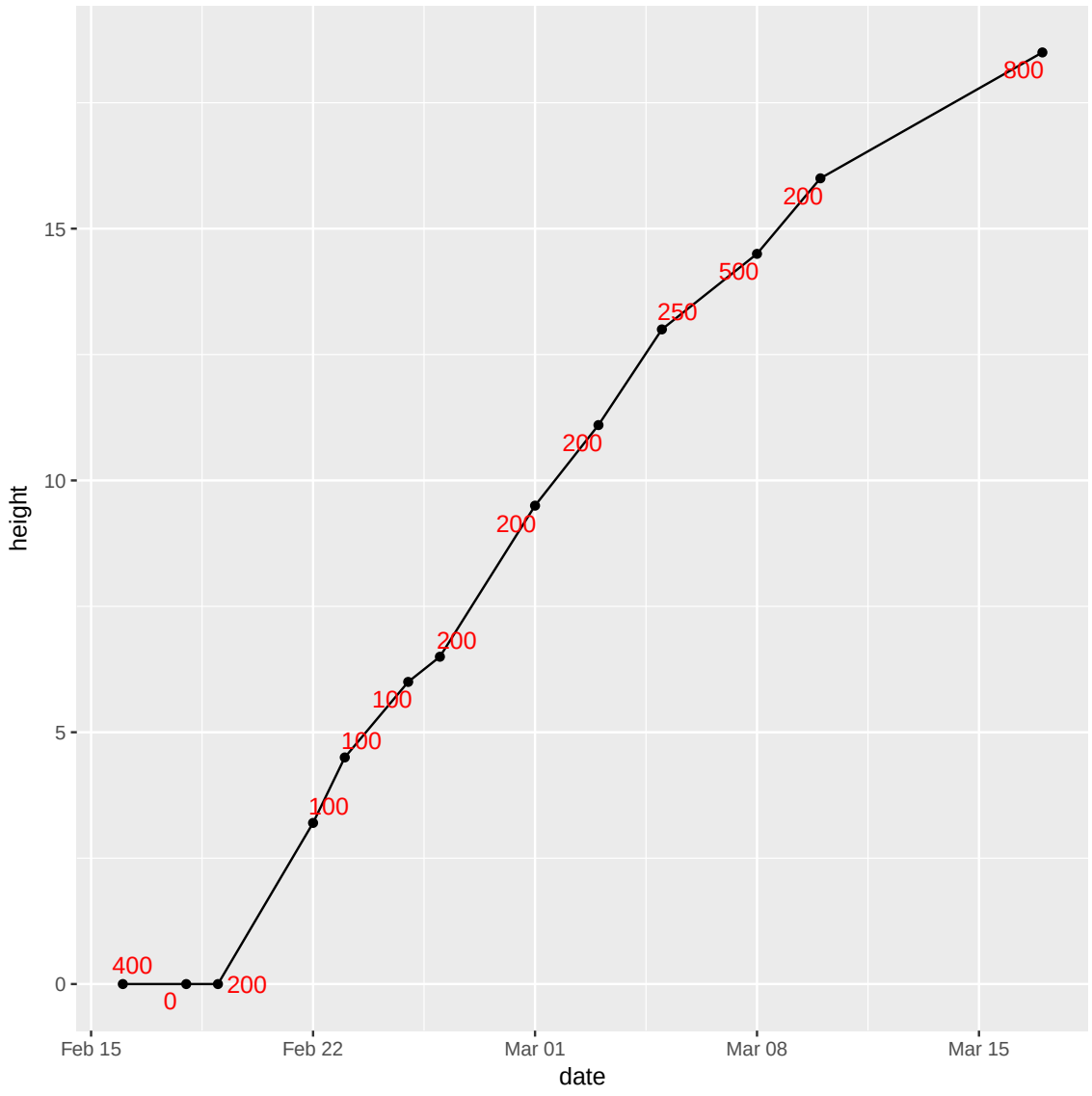
```
ggplot(mizuna,aes(x=date,y=height))+geom_point()+geom_line()
```



(f) Label each point on the plot with the amount of water added up to that point.

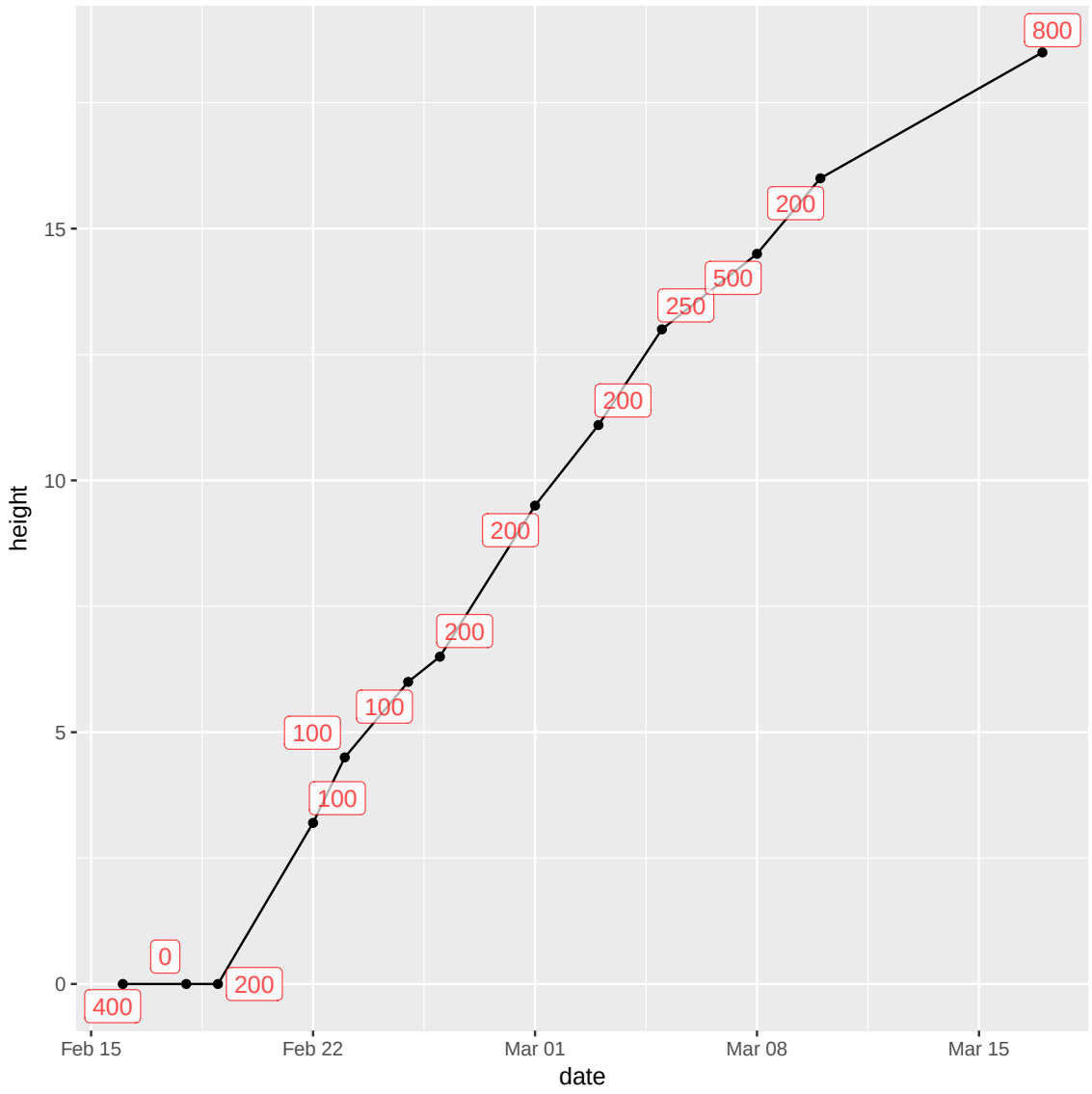
Solution: This is `water` again. The way to do this is to load `ggrepel`, then add `geom_text_repel` to the plot, by adding `label=water` to the *original* `aes`:

```
library(ggrepel)
ggplot(mizuna,aes(x=date,y=height,label=water))+
  geom_point()+geom_line()+geom_text_repel(colour="red")
```



I made the text red, so that you can see it more easily. It "repels" away from the points, but not from the lines joining them. Which makes me wonder whether this would work better (I explain `alpha` afterwards):

```
library(ggrepel)
ggplot(mizuna,aes(x=date,y=height,label=water))+
  geom_point()+geom_line()+geom_label_repel(colour="red",alpha=0.7)
```



The difference between `text` and `label` is that `text` just uses the text of the variable to mark the point, while `label` puts that text in a box.

I think it works better. You can see where the line goes (under the boxes with the labels in them), but you can see the labels clearly.

What that `alpha` does is to make the thing it's attached to (the labels) partly *transparent*. If you leave it out (try it), the black line disappears completely under the label boxes and you can't see where it goes at all. The value you give for `alpha` says how transparent the thing is, from 1 (not transparent at all) down to 0 (invisible). I first tried 0.3, and you could hardly see the boxes; then I tried 0.7 so that the boxes were a bit more prominent but the lines underneath were still slightly visible, and I decided that this is what I liked. I think making the labels a different colour was a good idea, since that helps to distinguish the number on the label from the line underneath.

You can apply `alpha` to pretty much any `ggplot` thing that might be on top of something else, to make it possible to see what's underneath it. The commonest use for it is if you have a scatterplot with a lot

of points; normally, you only see some of the points, because the plot is then a sea of black. But if you make the points partly transparent, you can see more of what's nearby that would otherwise have been hidden.

At some point, I also have to show you folks `jitter`, which plots in slightly different places points that would otherwise overprint each other exactly, and you wouldn't know how many of them there were, like the outliers on the boxplots of German children near the new airport.

3. Childbirths can be of two types: a "vaginal" birth in which the child is born through the mother's vagina in the normal fashion, and a "cesarean section" where a surgeon cuts through the wall of the mother's abdomen, and the baby is delivered through the incision. Cesarean births are used when there are difficulties in pregnancy or during childbirth that would make a vaginal birth too risky.

   A hospital kept track of the number of vaginal and Cesarean births for the twelve months of 2012. Of interest is whether the Cesarean rate (the ratio of Cesarean births to all births) was increasing, decreasing or remaining stable over that time.

   The data may be found at `http://www.utsc.utoronto.ca/~butler/c32/birthtypes.txt`. The columns are the names of the months (in 2012), the number of cesarean births and the number of vaginal births. (The data are not real, but are typical of the kind of thing you would observe.)

   (a) Read the data into R and display your data frame.

Solution: This is a space-delimited text file, which means:

```
my_url="http://www.utsc.utoronto.ca/~butler/c32/birthtypes.txt"
births=read_delim(my_url, " ")
```

```
## Parsed with column specification:
## cols(
##  month = col_character(),
##  cesarean = col_integer(),
##  vaginal = col_integer()
## )
```

```
births
```

```
## # A tibble: 12 x 3
##    month cesarean vaginal
##    <chr>    <int>   <int>
##  1   Jan       11      68
##  2   Feb        9      63
##  3   Mar       10      72
##  4   Apr       18     105
##  5   May       10      90
##  6   Jun       10      92
##  7   Jul       11      78
##  8   Aug        9      83
##  9   Sep        9      90
## 10   Oct       15     101
## 11   Nov       12     130
## 12   Dec        8     101
```

Some text and two numbers for each month. Check.

(b) Create a column of actual dates and also a column of cesarean rates, as defined above. Store your new data frame in a variable and display it. For the dates, assume that each date is of the 1st of the month that it belongs to.

Solution: The easiest way is to use `str_c` or `paste` to create a text date with year, month and day in some order, and then to use the appropriate function from `lubridate` to turn that into an actual `date`. If you use `str_c`, you (probably) need the `sep` thing to make sure the values get a space between them; `paste` does this automatically. (The next question is whether `ymd` or whatever can cope without spaces, but I'm not exploring that.)

The cesarean rate is `cesarean` divided by `cesarean` plus `vaginal`:

```
library(lubridate)
```

```
## Loading required package:  methods
```

```
##
## Attaching package:  'lubridate'
```

```
## The following object is masked from 'package:base':
##
##    date
```

```
b2 = births %>% mutate(datestr=str_c("2012",month,"1",sep=" ")) %>%
  mutate(thedate=ymd(datestr)) %>%
  mutate(cesarean_rate=cesarean/(cesarean+vaginal))
b2
```

```
## # A tibble: 12 x 6
##    month cesarean vaginal    datestr    thedate cesarean_rate
##    <chr>    <int>   <int>      <chr>     <date>         <dbl>
##  1   Jan       11      68 2012 Jan 1 2012-01-01    0.13924051
##  2   Feb        9      63 2012 Feb 1 2012-02-01    0.12500000
##  3   Mar       10      72 2012 Mar 1 2012-03-01    0.12195122
##  4   Apr       18     105 2012 Apr 1 2012-04-01    0.14634146
##  5   May       10      90 2012 May 1 2012-05-01    0.10000000
##  6   Jun       10      92 2012 Jun 1 2012-06-01    0.09803922
##  7   Jul       11      78 2012 Jul 1 2012-07-01    0.12359551
##  8   Aug        9      83 2012 Aug 1 2012-08-01    0.09782609
##  9   Sep        9      90 2012 Sep 1 2012-09-01    0.09090909
## 10   Oct       15     101 2012 Oct 1 2012-10-01    0.12931034
## 11   Nov       12     130 2012 Nov 1 2012-11-01    0.08450704
## 12   Dec        8     101 2012 Dec 1 2012-12-01    0.07339450
```

If you don't like that, create columns that contain 2012 and 1 all the way down. If you set a column name equal to a single value, that single value gets repeated the right number of times:[2]

```
births %>% mutate(year=2012,day=1)
```

```
## # A tibble: 12 x 5
##    month cesarean vaginal  year   day
##    <chr>    <int>   <int> <dbl> <dbl>
##  1   Jan       11      68  2012     1
##  2   Feb        9      63  2012     1
##  3   Mar       10      72  2012     1
##  4   Apr       18     105  2012     1
##  5   May       10      90  2012     1
##  6   Jun       10      92  2012     1
##  7   Jul       11      78  2012     1
##  8   Aug        9      83  2012     1
##  9   Sep        9      90  2012     1
## 10   Oct       15     101  2012     1
## 11   Nov       12     130  2012     1
## 12   Dec        8     101  2012     1
```

and then use `unite` as in class. The distinction is that `unite` *only* works on columns. It also "swallows up" the columns that it is made out of; in this case, the original year, month and day disappear:

```
b3 = births %>% mutate(year=2012, day=1) %>%
  unite(datestr,year,month,day) %>%
  mutate(thedate=ymd(datestr)) %>%
  mutate(cesarean_rate=cesarean/(cesarean+vaginal))
b3
```

```
## # A tibble: 12 x 5
##       datestr cesarean vaginal    thedate cesarean_rate
##         <chr>    <int>   <int>     <date>         <dbl>
##  1 2012_Jan_1       11      68 2012-01-01    0.13924051
##  2 2012_Feb_1        9      63 2012-02-01    0.12500000
##  3 2012_Mar_1       10      72 2012-03-01    0.12195122
##  4 2012_Apr_1       18     105 2012-04-01    0.14634146
##  5 2012_May_1       10      90 2012-05-01    0.10000000
##  6 2012_Jun_1       10      92 2012-06-01    0.09803922
##  7 2012_Jul_1       11      78 2012-07-01    0.12359551
##  8 2012_Aug_1        9      83 2012-08-01    0.09782609
##  9 2012_Sep_1        9      90 2012-09-01    0.09090909
## 10 2012_Oct_1       15     101 2012-10-01    0.12931034
## 11 2012_Nov_1       12     130 2012-11-01    0.08450704
## 12 2012_Dec_1        8     101 2012-12-01    0.07339450
```

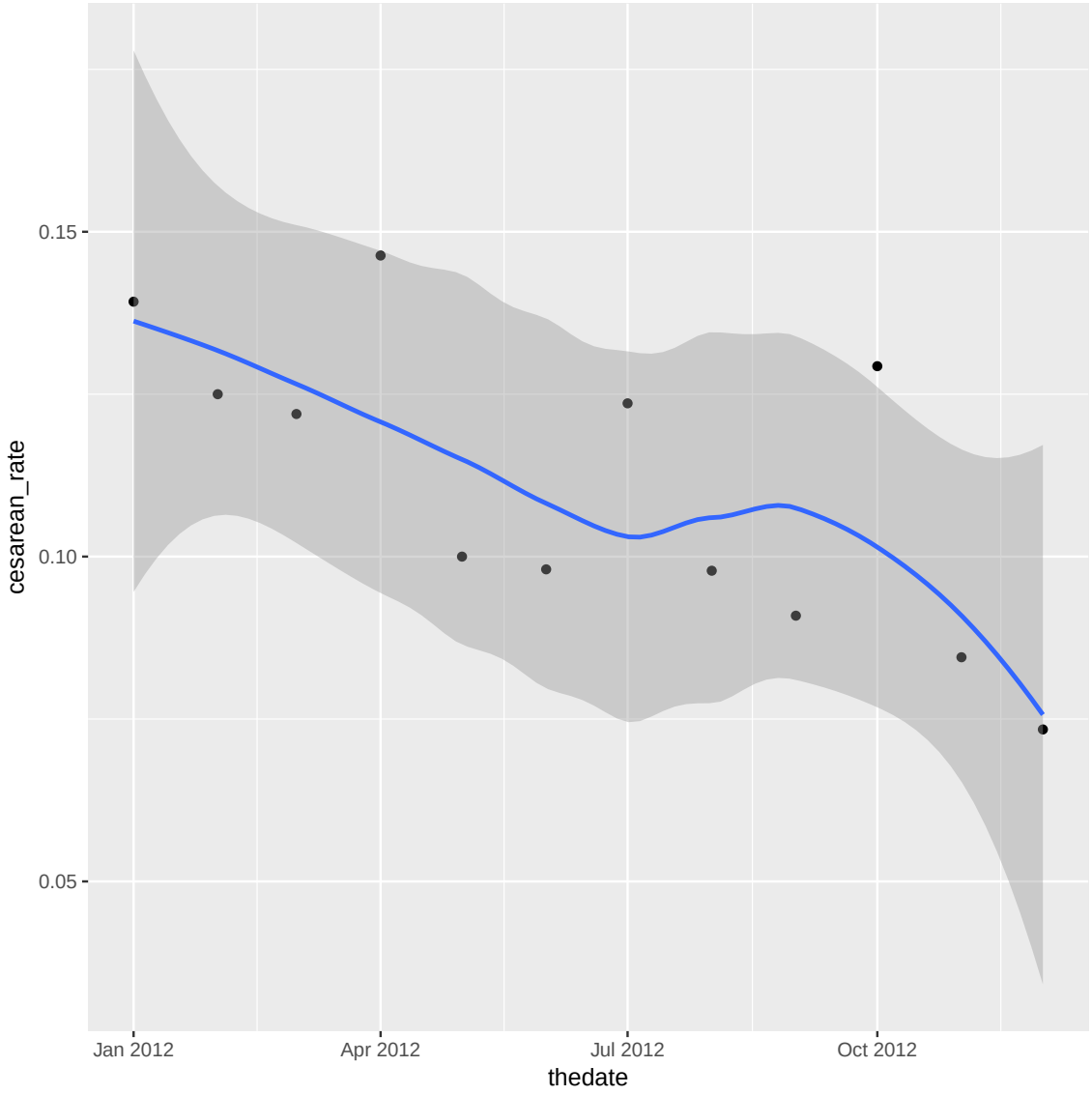
I don't mind which order you glue your year, month and day together, as long as you construct the dates with the consistent `lubridate` function.

(c) Plot the cesarean rate against time, with a smooth trend. Do you see an upward trend, a downward trend, no trend, or something else?

Solution: This is a scatterplot with time on the x axis:

```
ggplot(b2,aes(x=thedate,y=cesarean_rate))+geom_point()+geom_smooth()
```

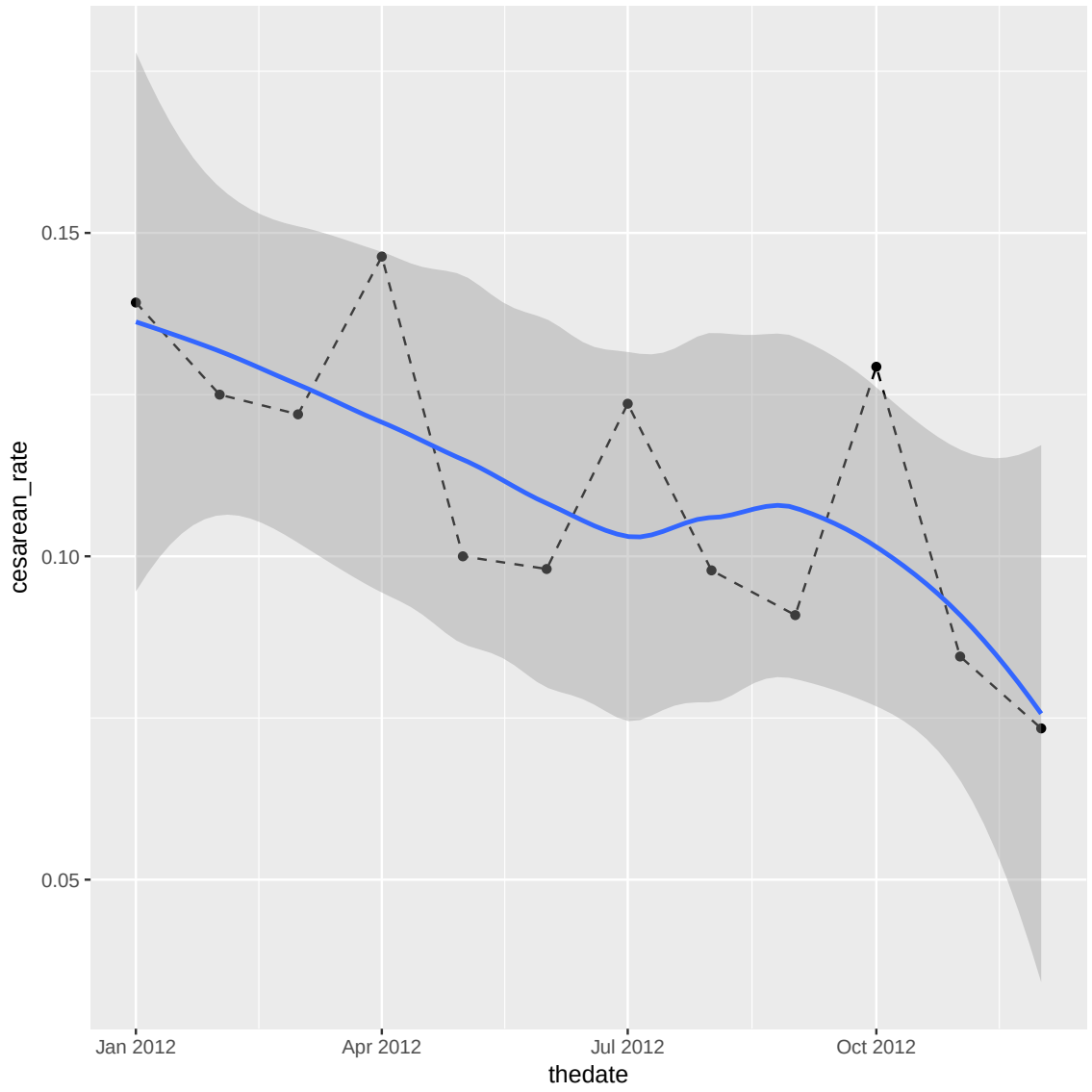
```
## `geom_smooth()` using method = 'loess'
```



I like this better than joining the points by lines, since we already have a trend on the plot, but you can do that in some contrasting way:

```
ggplot(b2,aes(x=thedate,y=cesarean_rate))+geom_point()+
  geom_line(linetype="dashed")+geom_smooth()
```

```
## `geom_smooth()` using method = 'loess'
```



I see a downward trend. ("A downward trend with a wiggle" if you like.) I didn't ask for any explanation. There is a certain unevenness in the trend of the actual data, but the overall picture appears to be downhill.

(d) Read the same data into SAS and display all 12 rows.

Solution: The usual stuff first:

```
filename myurl url 'http://www.utsc.utoronto.ca/~butler/c32/birthtypes.txt';

proc import
  datafile=myurl
  out=births
  dbms=dlm
```

```
  replace;
  getnames=yes;
  delimiter=' ';


proc print;
```

Obs	month	cesarean	vaginal
1	Jan	11	68
2	Feb	9	63
3	Mar	10	72
4	Apr	18	105
5	May	10	90
6	Jun	10	92
7	Jul	11	78
8	Aug	9	83
9	Sep	9	90
10	Oct	15	101
11	Nov	12	130
12	Dec	8	101

Next, we'll fix this up to make actual dates.

(e) Those "months" are really just the names of the months as text. Construct a new data set that makes real dates out of the month names (and a supplied year and day-of-month), and display it in such a way as to show that you now have dates. You might need to use `cat` and `input` in your data step (which in turn would mean finding out what they do). You might it convenient to note that a date in the form 01Jan1972 is what SAS calls `date9.`, with a dot on the end.

Solution: There are two steps: one, to make a piece of text that looks like a `date9.` date, which is what `cat` does, and two, to persuade SAS to treat this as a date, which is what `input` does.

`cat` works like `paste` in R (or, more precisely, `str_c` or `paste0`, since it glues the text together without any intervening spaces). It takes any number of inputs, which can be literal text or variables. In our case, it is the text `01` followed by the month name followed by the text `2012`, to make a date in `date9.` format.

`input` takes a variable and a format (precisely, an "informat") and expresses the value of the variable as input in the appropriate format. In this case, it converts the text-that-looks-like-a-date (in `thedate` below) into a real date (in `realdate`).

Then, to display the real dates so that they look like dates, you run `proc print` with a `format`. You can use any date format you like; I chose one that looks different from the dates-as-text in `thedate`. If you forget the `format`, the real dates will be output as days since Jan 1, 1960, which might convince you that they are real dates, but not *which* real dates they are.

After all that preamble, the code, followed by the output that it produces:

```
data births2;
  set births;
  thedate=cat('01',month,'2012');
  realdate=input(thedate,date9.);

proc print;
  format realdate yymmdd10.;
```

Obs	month	cesarean	vaginal	thedate	realdate
1	Jan	11	68	01Jan2012	2012-01-01
2	Feb	9	63	01Feb2012	2012-02-01
3	Mar	10	72	01Mar2012	2012-03-01
4	Apr	18	105	01Apr2012	2012-04-01
5	May	10	90	01May2012	2012-05-01
6	Jun	10	92	01Jun2012	2012-06-01
7	Jul	11	78	01Jul2012	2012-07-01
8	Aug	9	83	01Aug2012	2012-08-01
9	Sep	9	90	01Sep2012	2012-09-01
10	Oct	15	101	01Oct2012	2012-10-01
11	Nov	12	130	01Nov2012	2012-11-01
12	Dec	8	101	01Dec2012	2012-12-01

The dates in `realdate` are the same dates as in `thedate`, but written a different way. The fact that SAS re-formats the dates this way *and gets them right* is evidence that it is handling the dates properly.

The `mdy` idea from the lecture notes doesn't work, because our months are *names* and they need to be *numbers* for `mdy` to work. It seems to be necessary to construct a piece of text that looks like a date and then use `input` to convert it into an actual date.

If you get stuck, go back and edit (a copy of) the data file to include the years and month days and then read that in. This will enable you to do the remaining parts, but don't expect to get more than one point for this part if that's what you do.

(f) With yet another new data set, create a variable containing the cesarian rates, as you did earlier.

Solution: This is just a matter of keeping your head after the craziness of the previous part. There is no craziness here:

```
data births3;
  set births2;
  ces_rate=cesarean/(cesarean+vaginal);

proc print;
  format realdate ddmmyy8.;
```

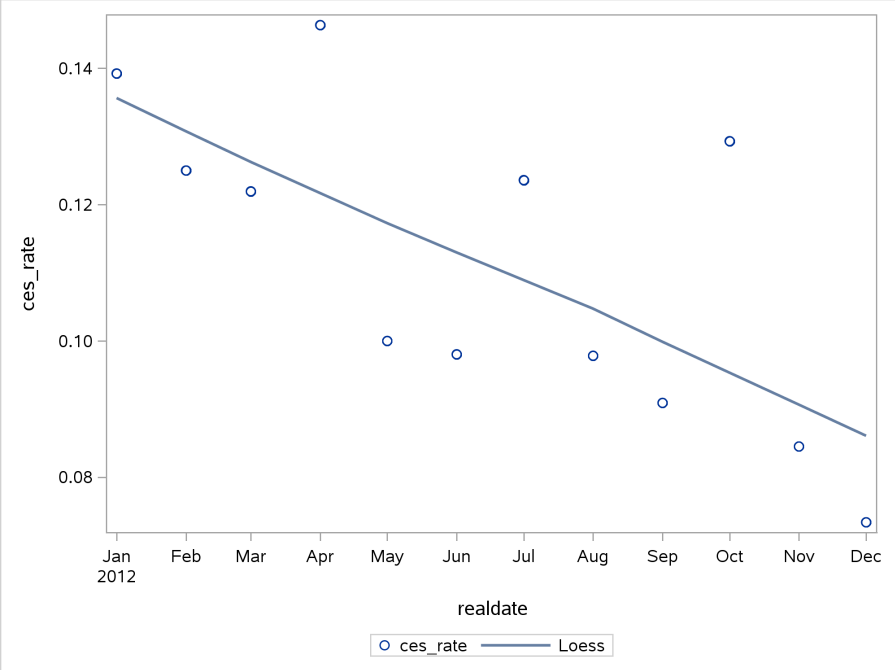
Obs	month	cesarean	vaginal	thedate	realdate	ces_rate
1	Jan	11	68	01Jan2012	01/01/12	0.13924
2	Feb	9	63	01Feb2012	01/02/12	0.12500
3	Mar	10	72	01Mar2012	01/03/12	0.12195
4	Apr	18	105	01Apr2012	01/04/12	0.14634
5	May	10	90	01May2012	01/05/12	0.10000
6	Jun	10	92	01Jun2012	01/06/12	0.09804
7	Jul	11	78	01Jul2012	01/07/12	0.12360
8	Aug	9	83	01Aug2012	01/08/12	0.09783
9	Sep	9	90	01Sep2012	01/09/12	0.09091
10	Oct	15	101	01Oct2012	01/10/12	0.12931
11	Nov	12	130	01Nov2012	01/11/12	0.08451
12	Dec	8	101	01Dec2012	01/12/12	0.07339

Well, apart from the craziness I introduced with the display of the dates this time, at any rate. Again, you can use any date format you like for these. The point of this part is to get the cesarean rates right.

(g) Make a scatterplot of cesarean rates against time. Add a smooth trend.

Solution: This should look about the same as the R one. You need to add a `format` to the x-axis, or else it will come out as seconds since Jan 1, 1960. Having one `format` seems to be enough, though there's no harm in having a format after `scatter` and one after `loess` as well. (The `format` actually belongs to `proc sgplot`, so one format applies to the whole plot, no matter how many points, lines or curves are on it.)

```
proc sgplot;
  scatter x=realdate y=ces_rate;
  loess x=realdate y=ces_rate;
  format realdate date9.;
```

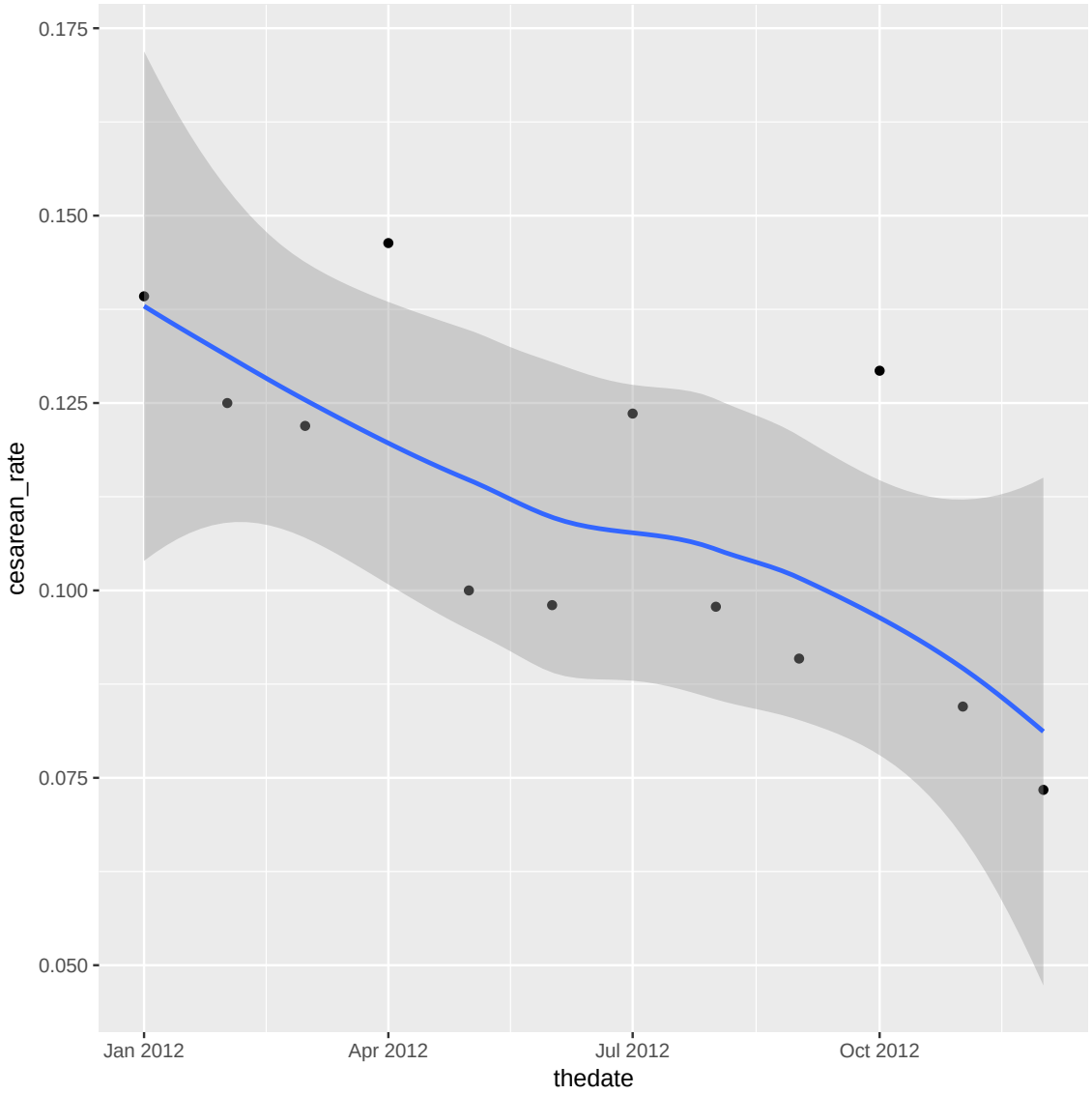


Any date format will do. (If you use a format like `yymmdd10.` that displays the month as a number, you might get numbers on the x-axis. I'm OK with that.)

This trend is a rather more evidently downward one than R's, probably because SAS does a bit more smoothing than R does. The amount of smoothing done by `geom_smooth` is controlled by something called `span`,[3] which has a default value of 0.75, so making `span` a bit bigger smooths out the smooth trend:

```
ggplot(b2,aes(x=thedate,y=cesarean_rate))+geom_point()+
  geom_smooth(span=1)
```

```
## `geom_smooth()` using method = 'loess'
```



This still has a bit more of a wiggle than SAS's picture, but it tells the same story. Making `span` smaller would make the trend more wiggly, and in my opinion makes it over-react to where data values happen to be.

(h) Use `proc corr` to find the Kendall correlation with time (the "Mann-Kendall correlation").[4] This will involve digging into the help for `proc corr`. You can find this by entering `proc corr` into your favourite search engine, and looking at the results that start with `support.sas.com`. The output should also give you a P-value.

Solution: A small amount of digging reveals two things:

1. to get the Kendall correlation (officially called "Kendall's tau-b"), you add `kendall` to the `proc corr` line.
2. to specify the variables to calculate correlations between, put them on a `var` line, as you would for `proc means`. (If you don't, you get correlations for all the pairs of variables.)

Thus:

```
proc corr kendall;
  var realdate ces_rate;
```

```
                          The CORR Procedure

                    2  Variables:    realdate ces_rate
                          Simple Statistics

Variable         N        Mean     Std Dev      Median     Minimum     Maximum

realdate        12       19160   109.93249       19160       18993       19328
ces_rate        12     0.11084     0.02304     0.11098     0.07339     0.14634
               Kendall Tau b Correlation Coefficients, N = 12
                        Prob > |tau| under H0: Tau=0

                                    realdate      ces_rate

                    realdate         1.00000      -0.60606
                                                    0.0061

                    ces_rate        -0.60606       1.00000
                                      0.0061
```

The correlation with time is -0.606 (with a P-value of 0.0061 that we use below).

Extra stuff: if you try to calculate the Kendall correlation with time in R, it doesn't work:

```
with(b2,cor(thedate,cesarean_rate),method="kendall")
```

```
## Error in cor(thedate, cesarean_rate):  'x' must be numeric
```

Why did it work in SAS but not in R? The error message says that `x`, the first input to `cor`, was not numeric:

```
class(b2$thedate)
```

```
## [1] "Date"
```

and R won't even pretend it's a number. SAS, on the other hand, stores dates as numbers (days since Jan 1, 1960), and the number is rather more visible: you have to do the `format` thing to even make them display as dates, otherwise you get that number. Thus, in SAS, when you calculate a correlation with a date (or a time), SAS will use the underlying number, and will get you a correlation with time. (A higher number means a later date or time, so it is doing what you want.)

It is a matter of opinion whether you think of dates as being "ordinal" or "interval" — for example, do you think it makes sense to calculate a *mean* date? — but the Kendall correlation only uses the order of the dates, not how far apart they are, so it is good if you think dates are only ordinal, while the regular Pearson correlation is not.

With R, you have to literally make the date into a number:

```
nd=as.numeric(b2$thedate)
nd
```

```
##  [1] 15340 15371 15400 15431 15461 15492 15522 15553 15584 15614 15645
## [12] 15675
```

These are days since January 1, *1970*. This is the Unix "epoch date". Unix was developed in the early 1970s; the way time was measured on early Unix systems meant that you had to have a zero date as something in the recent past, and so it was first 1971 and then 1970.[5] R was first developed on Unix systems, so it inherited date-and-time handling from there, as well as other things like `ls` for "list my objects" and `rm` for "delete".

SAS began life in the late 1960s, before Unix even existed, and so its "zero" date was chosen independently to be Jan 1 1960.

Anyway, by expressing our variable `thedate` in R as a literal number, we can now calculate the Kendall correlation with time:

```
cor(nd,b2$cesarean_rate,method="kendall")
```

```
## [1] -0.6060606
```

which gives the same answer as SAS, and we can do the test this way:

```
cor.test(nd,b2$cesarean_rate,method="kendall",exact=F)
```

```
##
##  Kendall's rank correlation tau
##
## data:  nd and b2$cesarean_rate
## z = -2.7429, p-value = 0.00609
## alternative hypothesis: true tau is not equal to 0
## sample estimates:
##        tau
## -0.6060606
```

The reason for the `exact=F` is to get the same P-value as SAS. The Kendall correlation has an approximate normal distribution (an approximation that improves as the sample size increases: here we only have 12 pairs of observations, which is not exactly large). SAS's default is to use the normal approximation and R's default is to calculate the P-value exactly for small samples, but they are both tweakable. (The exact P-value, from R, is about 0.0054, so it doesn't make much difference here.)

(i) What do you conclude from the P-value you obtained in the last part? Explain briefly.

Solution: You can try to reason out what is being tested, or you can look in the documentation. The first example, at `http://support.sas.com/documentation/cdl/en/procstat/66703/HTML/default/viewer.htm#procstat_corr_examples01.htm`, is a good place to look. From there, the P-value is for a test of "is there a time trend", with a small P-value meaning that there is a trend. If you prefer to reason things out, this is just like testing for a regression slope: the null hypothesis is that the slope is zero, and therefore that there is no relationship, and the two-sided alternative is that the slope is not zero, ie. that there is either an upward or downward trend.

Thus, the P-value of 0.0061 here is small, and therefore we *reject* the null hypothesis (that says there is no trend) in favour of the alternative, and therefore conclude that there *is* a trend, or that the trend observed over time on the plot is "real" or "reproducible" or some word like that.

In practice, you would typically have a much longer time series of measurements than this, such as monthly measurements for several years. In looking at only one year, like we did here, we could get trapped by seasonal effects: for example, cesarean rates might always go down through the year and then jump up again in January. Looking at several years would enable us to disentangle seasonal effects that happen every year from long-term trends. (As an example of this, think of Toronto snowfall: there is almost always snow in the winter and there is never snow in the summer, a seasonal effect, but in assessing climate change, you want to think about long-term trends in snowfall, after allowing for which month you're looking at.)

4. Previously, we did some row and column selection with my cars data set, `http://www.utsc.utoronto.ca/~butler/c32/cars.csv`. This time, we're going to create a permanent data set out of these data, and then demonstrate that we can use it in a `proc` without reading it in again.

   (a) Read the data in and create a permanent data set. To do this, modify your `proc import` from before along the lines of the lecture notes, replacing the `ken` that is there with your username.

Solution: This is the `proc import` from before:

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

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

You need to do two things to this: first, you create a `libname` up above the `proc import` that says where you want this permanent data set created, and second, you put the `libname` before the data set name on the `out` line. If you do that, you'll get something like this:

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

libname mylib V9 '/home/ken';

proc import
  datafile=myurl
  dbms=csv
  out=mylib.mycars
  replace;
  getnames=yes;
```

In place of the `mylib` you can use any name you like (the same in both places), and the `ken` at the end should be replaced by your username, but otherwise use that line as is.

I think the `V9` on the `libname` line has to be there because the format of the permanent dataset file changed with version 9 of SAS, and SAS didn't want to break any code that created permanent data sets from earlier versions of SAS. Backward compatibility, and all that.

To check whether it worked, look in your SAS file storage, on the left side of SAS Studio, under Files (Home). You should see a file with the improbable name of `cars.sas7bdat`. That's the permanent data set. Fortunately, we never have to use that crazy extension.

I actually have SAS running on my computer rather than SAS Studio (in any of its flavours), so that I can check for the existence of a file (in Linux)[6] like this:

```
ls -l /home/ken/mycars.*
```

```
## -rw-rw-r-- 1 ken ken 131072 Nov 27 23:00 /home/ken/mycars.sas7bdat
```

There it is, and just created (as I write this).

(b) Close down SAS, and start it up again. Find the mean gas mileage for cars with each number of cylinders, *without* reading the data in again (that is to say, without using `proc import` or trickery involving `data` steps).

Solution: *Start* with the `proc means`, and to that append a `data=` that tells SAS to use your permanent data set. This is done by putting the data set name in quotes and starting it with a `/home/username`, as if it were a file, but with *no extension*. For me, that goes like this:

```
proc means data='/home/ken/mycars';
  var MPG;
  class cylinders;
```

For you, replace `ken` with your username. Here's what I got; you should get the same thing:

```
                          The MEANS Procedure

                        Analysis Variable : MPG

                 N
  cylinders  Obs   N          Mean       Std Dev       Minimum       Maximum
-------------------------------------------------------------------------------
          4   19  19    30.0210526     4.1824473    21.5000000    37.3000000

          5    1   1    20.3000000             .    20.3000000    20.3000000

          6   10  10    21.0800000     4.0775265    16.2000000    28.8000000

          8    8   8    17.4250000     1.1925363    15.5000000    19.2000000
-------------------------------------------------------------------------------
```

And that works, using my permanent data set.

Another way to do it is via the `libname` thing, same as you did when you saved the permanent data set:

```
libname mylib V9 '/home/ken';

proc means data=mylib.mycars;
  var MPG;
  class cylinders;
```

```
                          The MEANS Procedure

                        Analysis Variable : MPG

                 N
  cylinders  Obs   N          Mean       Std Dev       Minimum       Maximum
-------------------------------------------------------------------------------
          4   19  19    30.0210526     4.1824473    21.5000000    37.3000000

          5    1   1    20.3000000             .    20.3000000    20.3000000

          6   10  10    21.0800000     4.0775265    16.2000000    28.8000000

          8    8   8    17.4250000     1.1925363    15.5000000    19.2000000
-------------------------------------------------------------------------------
```

Success. Either way is good.

If you're in a company that uses SAS, permanent data sets are a good way to share data, since the person receiving the data doesn't have to do anything to open it: they can just start running `proc`s right away.

5. A poll on the Discovery Channel asked people to nominate the best roller-coasters in the United States. We will examine the 10 roller-coasters that received the most votes. Two features of a roller-coaster that are of interest are the distance it drops from start to finish, measured here in feet[7] and the duration of the ride, measured in seconds. Is it true that roller-coasters with a bigger drop also tend to have a longer ride? The data are at `http://www.utsc.utoronto.ca/~butler/c32/coasters.csv`.[8]

   (a) Read the data into R and verify that you have a sensible number of rows and columns.

Solution: A `.csv`, so the usual for that:

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

```
## Parsed with column specification:
## cols(
##   coaster_name = col_character(),
##   state = col_character(),
##   drop = col_integer(),
##   duration = col_integer()
## )
```

```
coasters
```

```
## # A tibble: 10 x 4
##        coaster_name        state  drop duration
##               <chr>        <chr> <int>    <int>
##  1  Incredible Hulk      Florida   105      135
##  2 Millennium Force         Ohio   300      105
##  3          Goliath   California   255      180
##  4            Nitro   New Jersey   215      240
##  5   Magnum XL-2000         Ohio   195      120
##  6        The Beast         Ohio   141       65
##  7     Son of Beast         Ohio   214      140
##  8      Thunderbolt Pennsylvania    95       90
##  9      Ghost Rider   California   108      160
## 10            Raven      Indiana    86       90
```

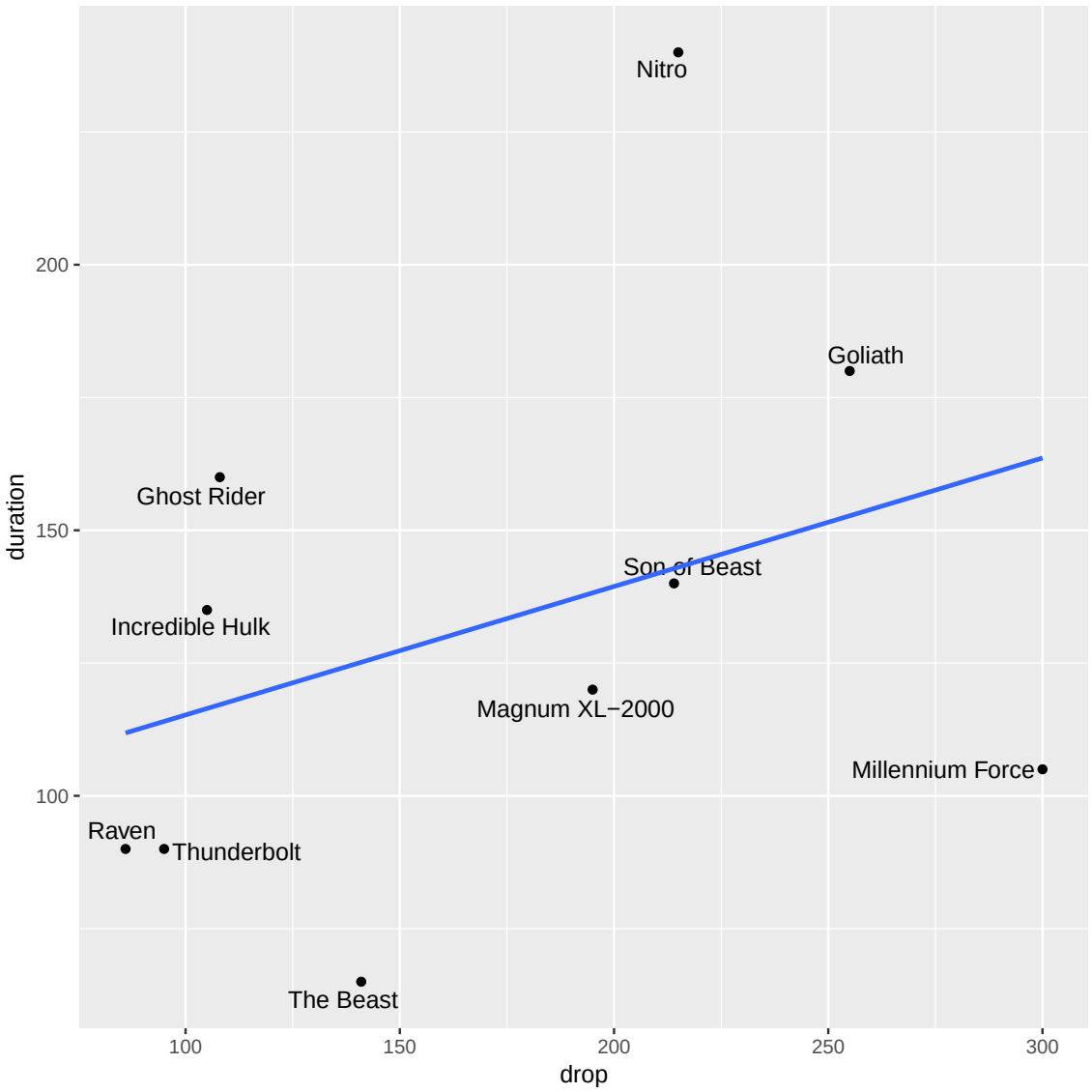
The number of marks for this kind of thing has been decreasing through the course, since by now you ought to have figured out how to do it without looking it up.

There are 10 rows for the promised 10 roller-coasters, and there are several columns: the drop for each roller-coaster and the duration of its ride, as promised, as well as the name of each roller-coaster and the state that it is in. (A lot of them seem to be in Ohio, for some reason that I don't know.) So this all looks good.

(b) Make a scatterplot of duration (response) against drop (explanatory), labelling each roller-coaster with its name in such a way that the labels do not overlap. Add a regression line to your plot.

Solution: The last part, about the labels not overlapping, is an invitation to use `ggrepel`, which is the way I'd recommend doing this. (If not, you have to do potentially lots of work organizing where the labels sit relative to the points, which is time you probably don't want to spend.) Thus:

```
library(ggrepel)
ggplot(coasters,aes(x=drop,y=duration,label=coaster_name))+
  geom_point()+geom_text_repel()+geom_smooth(method="lm",se=F)
```



The `se=F` at the end is optional; if you omit it, you get that "envelope" around the line, which is fine here.

Note that with the labelling done this way, you can easily identify which roller-coaster is which.

(c) Would you say that roller-coasters with a larger drop tend to have a longer ride? Explain briefly.

Solution: I think there are two good answers here: "yes" and "kind of".

Supporting "yes" is the fact that the regression line does go uphill, so that overall, or on average, roller-coasters with a larger drop do tend to have a longer duration of ride as well.

Supporting "kind of" is the fact that, though the regression line goes uphill, there are a lot of roller-coasters that are some way off the trend, far from the regression line.

I am happy to go with either of those. I could also go with "not really" and the same discussion that I attached to "kind of".

(d) Find a roller-coaster that is unusual compared to the others. What about its combination of `drop` and `duration` is unusual?

Solution: This is an invitation to find a point that is a long way off the line. I think the obvious choice is my first one below, but I would take either of the others as well:

- "Nitro" is a long way above the line. That means it has a long duration, relative to its drop. There are two other roller-coasters that have a larger drop but not as long a duration. In other words, this roller-coaster drops slowly, presumably by doing a lot of twisting, loop-the-loop and so on.
- "The Beast" is a long way below the line, so it has a short duration relative to its drop. It is actually the shortest ride of all, but is only a bit below average in terms of drop. This suggests that The Beast is one of those rides that drops a long way quickly.
- "Millennium Force" has the biggest drop of all, but a shorter-than-average duration. This looks like another ride with a big drop in it.

A roller-coaster that is "unusual" will have a residual that is large in size (either positive, like Nitro, or negative, like the other two). I didn't ask you to find the residuals, but if you want to, `augment` from `broom` is the smoothest way to go:

```
library(broom)
duration.1=lm(duration~drop,data=coasters)
augment(duration.1,coasters) %>%
  select(coaster_name,duration,drop,.resid) %>%
  arrange(desc(abs(.resid)))
```

```
## Warning:  Deprecated:  please use 'purrr::possibly()' instead

## Warning:  Deprecated:  please use 'purrr::possibly()' instead

## Warning:  Deprecated:  please use 'purrr::possibly()' instead

## Warning:  Deprecated:  please use 'purrr::possibly()' instead

## Warning:  Deprecated:  please use 'purrr::possibly()' instead
```

```
##        coaster_name duration drop    .resid
## 1             Nitro      240  215  96.95170
## 2         The Beast       65  141 -60.14522
## 3  Millennium Force      105  300 -58.61266
## 4       Ghost Rider      160  108  42.83859
## 5           Goliath      180  255  27.27435
## 6       Thunderbolt       90   95 -24.01628
## 7             Raven       90   86 -21.83887
## 8   Incredible Hulk      135  105  18.56439
## 9    Magnum XL-2000      120  195 -18.20963
## 10     Son of Beast      140  214  -2.80637
```

`augment` produces a data frame (of the original data frame with some new columns that come from the regression), so I can feed it into a pipe to do things with it, like only displaying the columns I want, and arranging them in order by absolute value of residual, so that the roller-coasters further from the line

come out first. This identifies the three that we found above. The fourth one, "Ghost Rider", is like Nitro in that it takes a (relatively) long time to fall not very far.

You may safely ignore the warning. It's supposed to be in the process of being fixed.

You can also put `augment` in the *middle* of a pipe, but then you seem to lose things that were not in the regression:

```
coasters %>%
  lm(duration~drop, data=.) %>%
  augment() %>%
  arrange(desc(abs(.resid)))
```

```
## Warning:  Deprecated:  please use 'purrr::possibly()' instead
## Warning:  Deprecated:  please use 'purrr::possibly()' instead
## Warning:  Deprecated:  please use 'purrr::possibly()' instead
## Warning:  Deprecated:  please use 'purrr::possibly()' instead
## Warning:  Deprecated:  please use 'purrr::possibly()' instead
```

```
##    duration drop  .fitted  .se.fit    .resid      .hat   .sigma
## 1       240  215 143.0483 18.91474  96.95170 0.1378057 37.54500
## 2        65  141 125.1452 17.53092 -60.14522 0.1183794 48.79432
## 3       105  300 163.6127 33.36918 -58.61266 0.4289016 45.90966
## 4       160  108 117.1614 21.61376  42.83859 0.1799397 51.45256
## 5       180  255 152.7256 24.90924  27.27435 0.2389942 53.17339
## 6        90   95 114.0163 23.68523 -24.01628 0.2160836 53.49714
## 7        90   86 111.8389 25.22254 -21.83887 0.2450440 53.63586
## 8       135  105 116.4356 22.07396  18.56439 0.1876840 53.91145
## 9       120  195 138.2096 16.98158 -18.20963 0.1110766 53.97930
## 10      140  214 142.8064 18.79672  -2.80637 0.1360914 54.45872
##          .cooksd  .std.resid
## 1  0.3355862058  2.04920817
## 2  0.1061087925 -1.25716851
## 3  0.8700715796 -1.52219103
## 4  0.0945675641  0.92842216
## 5  0.0591230416  0.61361184
## 6  0.0390597997 -0.53235830
## 7  0.0394909652 -0.49329040
## 8  0.0188788219  0.40425125
## 9  0.0089770351 -0.37905502
## 10 0.0002765802 -0.05925762
```

I wanted to hang on to the roller-coaster names, so I went the other way. The advantage of this way is less typing. You have to specify the `data=.` thing in the regression because the data frame is *not* the first input to `lm` (the model is), but you have a much simpler `augment` because its first input is the model that came out of the previous step. If the second input to `augment` is missing, as it is here, it "attempts to reconstruct the data from the model"[9] which I think means that things like the coaster names that were not part of the regression won't be part of `augment`'s output either. That's my understanding. My first way explicitly supplies the original data frame to `augment` so there is no question about what it is using.

Notes

[1]Jan 1, 1960 is SAS's "zero" date.

[2]This is an example of R's so-called "recycling rules".

[3]If `span` is less than 1, only that nearest fraction of points to where you are fitting is used, with nearer points having a higher weight. If `span` is 1 or bigger, all the points are used, but nearby points still have a higher weight.

[4]Environmental scientists like the Mann-Kendall correlation, rather than the standard "Pearson" correlation, because it is not assuming a linear trend, just a "monotone" one that keeps going up or keeps going down, and it is not affected by outliers, which natural data tends to have rather a lot of. The Kendall correlation also treats the variables as "ordinal" rather than "interval": like the sign test, it only thinks about whether the points are indicating an uphill or downhill trend, not how big of a trend it is. For *that*, they use a thing called the Theil-Sen slope, which is the median of the pairwise slopes between the points, again not affected by outliers.

[5]For those who know about "32-bit signed integers", starting in 1970 means that dates in Unix and Linux can be represented by one of these up until January 19, 2038, at which date it will wrap around to December 13, 1901: as many days before Jan 1 1970 as Jan 19 2038 is after. Many of us on Unix-like systems (including Macs) are now using 64-bit signed integers; Wikipedia tells me:

> Using a signed 64-bit value introduces a new wraparound date that is over twenty times greater than the estimated age of the universe: approximately 292 billion years from now, at 15:30:08 UTC on Sunday, 4 December 292,277,026,596.

I guess this problem is now solved.

[6]If you have a Mac, you can do something very similar by firing up a terminal window. I think Windows has `bash` now, which is what this is.

[7]Roller-coasters work by gravity, so there must be some drop.

[8]These are not to be confused with what your mom insists that you place between your coffee mug and the table.

[9]A quote from the package vignette.