Assignment 10

Instructions (the same as for Assignment 1): Make an R Notebook and in it answer the question below. When you are done, hand in on Quercus the *output* from Previewing (or Knitting) your Notebook. Do *not* hand in the Notebook itself. You want to show that you can (i) write code that will answer the questions, (ii) run that code and get some sensible output, (iii) write some words that show you know what is going on and that reflect your conclusions about the data. Your goal is to convince the grader that you *understand* what you are doing: not only doing the right thing, but making it clear that you know *why* it's the right thing.

Do *not* expect to get help on this assignment. The purpose of the assignments is for you to see how much *you* have understood. You will find that you also learn something from grappling with the assignments. The time to get help is after you watch the lectures and work through the problems from PASIAS, via tutorial and the discussion board. The only reason to contact the instructor while working on the assignments is to report something missing like a data file that cannot be read.

You have 5 hours to complete this assignment after you start it.

My solutions to this assignment, with extra discussion, will be available after everyone has handed in their assignment. **Note**: this assignment is playing the role of a final exam, so I am not allowed to discuss the marks on it, or your course grade. If you wish to contest either of those, you will have to go through the Registrar's office.

1. Suppose we have this vector of values:

```
z <- c(10, 14, 11)
z
```

```
## [1] 10 14 11
```

We want to scale these so that the smallest value is 0 and the largest is 1. We are going to be doing this a lot, so we are going to write a function that will work for any input.

(a) Using a copy of my `z`, work out `min(z)` and `max(z)`. What do they do? Explain (very) briefly.

Solution:

Simply this – define `z` first:

```
z <- c(10, 14, 11)
min(z)
```

```
## [1] 10
```

```
max(z)
```

```
## [1] 14
```

They are respectively (and as you would guess) the smallest and largest of the values in `z`. (A nice gentle warmup, but I wanted to get you on the right track for what is coming up.)

(b) What do these lines of code do, using the same `z` that I had? Run them and see, and describe briefly what `s` contains.

```r
lo <- min(z)
hi <- max(z)
s <- (z - lo) / (hi - lo)
s
```

Solution:

Here we go:

```r
lo <- min(z)
hi <- max(z)
s <- (z - lo) / (hi - lo)
s
```

```
## [1] 0.00 1.00 0.25
```

The lowest value became 0, the highest 1, and the other one to something in between. Saying this shows the greatest insight.

Extra: the reason for doing it in three steps rather than one (see below) is (i) it makes it clearer what is going on (and thus makes it less likely that you make a mistake), and (ii) it is more efficient, since my way only finds the minimum once instead of twice. Compare this approach with mine above:

```r
(z - min(z)) / (max(z) - min(z))
```

```
## [1] 0.00 1.00 0.25
```

More complicated with all the brackets, and two `min(z)`. Admittedly the difference here will be thousandths of a second, but why call a function twice when you don't have to?

(c) Write a function called `rescale` that implements the calculation above, for any input vector called `x`. (Note that I changed the name.)

Solution:

Write a function skeleton:

```r
rescale <- function(x) {

}
```

and inside the curly brackets put the code from above, replacing `z` with `x` everywhere:

```r
rescale <- function(x) {
  lo <- min(x)
  hi <- max(x)
  (x - lo) / (hi - lo)
}
```

You'll need to make sure your function returns something to the outside world. Either don't save the last line in `s` (as I did here), or save it in `s` and then return `s`:

```r
rescale <- function(x) {
  lo <- min(x)
  hi <- max(x)
  s <- (x - lo) / (hi - lo)
  s
}
```

or use `return()` if you must, but be aware that this is bad style in R (unlike Python, where you need it). The approved way of using `return` in R is when you are returning something *earlier* than the last line of a function, for example, you are testing a simple case first and returning the value that goes with that, before getting down to the serious computation.

Extra: in the spirit of what's coming up below, you might check first whether the maximum and minimum are the same and return something else if that's the case:

```r
rescale0 <- function(x) {
  lo <- min(x)
  hi <- max(x)
  if (lo == hi) return(0)
  s <- (x - lo) / (hi - lo)
  s
}
```

This is good style; in this case, if `lo` and `hi` are the same, we want to return something else (zero) to the outside world, and *then* we do the calculation, knowing that `lo` and `hi` are different, so that we are sure we are not dividing by zero (but I get ahead of myself).

Doing it this way, there is something to be careful of: a function ought to return a predictable *type* of thing: numbers, in this case. If you have your function return *text* on error, like this:

```r
rescale0 <- function(x) {
  lo <- min(x)
  hi <- max(x)
  if (lo == hi) return("High and low need to be different")
  s <- (x - lo) / (hi - lo)
  s
}
```

then you can get into trouble:

```r
rescale0(z)
```

```
## [1] 0.00 1.00 0.25
```

```r
rescale0(c(3,3,3))
```

```
## [1] "High and low need to be different"
```

The first one is numbers and the second one is text.

(d) Test your function on my `z`, and on another vector of your choosing. Explain briefly why the answer you get from your vector makes sense.

Solution:

On my `z`:

```r
rescale(z)
```

```
## [1] 0.00 1.00 0.25
```

The same values as I got before, so that works.

For your vector, use whatever you like. I think it makes sense to have the values already in order, to make it easier to check. Here's one possibility:

```r
w <- 2:6
w
```

```
## [1] 2 3 4 5 6
```

and then

```r
rescale(w)
```

```
## [1] 0.00 0.25 0.50 0.75 1.00
```

The smallest value is 2, which goes to zero; the largest is 6, which goes to 1, and the others are equally spaced between in both the input and the output.

Another possibility is to use a vector with values whose largest and smallest you can clearly see:

```r
w <- c(10, 11, 100, 0, 20)
rescale(w)
```

```
## [1] 0.10 0.11 1.00 0.00 0.20
```

Clearly the smallest value is 0 and the largest is 100. These become 0 and 1, and these particular values make it easy to see what happened: each of the other values got divided by 100.

Some discussion is needed here, in that you need to say something convincing about why your answer is right.

Extra: This is why I had you use a name *other than* `z` for the input to your function. The function can be used on any input, not just the `z` that we tested it on. There's another R-specific reason, which is that you need to be careful about using the named inputs *only*. Consider this function:

```r
ff <- function(x) {
  x + z
}

ff(10)
```

```
## [1] 20 24 21
```

Where did `z` come from? R *used the* `z` *we had before*, which is rather dangerous: what if we had a `z` lying around from some completely different work? Much better to have a function work with only inputs in the top line:

```r
ff <- function(x, z) {
  x + z
}
ff(10, 3)
```

```
## [1] 13
```

```
ff(10)

## Error in ff(10): argument "z" is missing, with no default
```

The first time, the two inputs are added together, but the second time it tells you it was expecting a value to use for `z` and didn't see one. Much safer.

(e) What happens if your input to `rescale` is a vector of numbers all the same? Give an example. Rewrite your function to intercept this case and give a helpful error message.

Solution:

First, try it and see. Any collection of values all the same will do:

```
rescale(c(3,3,3))

## [1] NaN NaN NaN
```

`NaN` stands for "not a number". The way we got it is that the minimum and maximum were the same, so our function ended up dividing by zero (in fact, working out zero divided by zero). This is, in R terms, not even an error, but the answer is certainly not helpful.

The easiest way to check inputs is to use `stopifnot` to express what should be *true* if the function is to proceed. Here, we want the maximum and minimum to be different, so:

```
rescale <- function(x) {
  lo <- min(x)
  hi <- max(x)
  stopifnot(hi != lo)
  (x - lo) / (hi - lo)
}
rescale(c(3,3,3))

## Error in rescale(c(3, 3, 3)): hi != lo is not TRUE
```

This is much clearer: I only have to recall what my `hi` and `lo` are to see what the problem is.

Extra 1: by calculating and saving the min and max up front, I still only need to calculate them once. If you do it this way:

```
rescale <- function(x) {
  stopifnot(max(x) != min(x))
  (x - min(x)) / (max(x) - min(x))
}
rescale(c(3,3,3))

## Error in rescale(c(3, 3, 3)): max(x) != min(x) is not TRUE
```

you get a slightly more informative error message, but you have calculated the max *twice* and the min *three times* for no reason.

Extra 2: `stopifnot` is shorthand for this:

```
rescale <- function(x) {
  lo <- min(x)
  hi <- max(x)
  if (hi == lo) stop("min and max are the same!")
  (x - lo) / (hi - lo)
}
rescale(c(3,3,3))
```

```
## Error in rescale(c(3, 3, 3)): min and max are the same!
```

I didn't show you this, so if you use `stop`, you must tell me where you found out about it. This is better than returning some text (see `rescale0` above) or printing a message: it's an error, so you want to make it look like an error. I am very sympathetic to being persuaded that this is *better* than `stopifnot`, because you can customize the message (and, also, you don't have to go through the double-negative contortions of `stopifnot`). Another way to use `stopifnot` and get a customized message is this one (that I only learned about right when you were writing this Assignment):

```
rescale <- function(x) {
  lo <- min(x)
  hi <- max(x)
  stopifnot("high and low must be different" = (hi != lo))
  (x - lo) / (hi - lo)
}
rescale(c(3,3,3))
```

```
## Error in rescale(c(3, 3, 3)): high and low must be different
```

This is called a "named argument", and the name, if given, is used as an error message.

Extra 3: returning to my `rescale0` from above:

```
rescale0
```

```
## function(x) {
##   lo <- min(x)
##   hi <- max(x)
##   if (lo == hi) return("High and low need to be different")
##   s <- (x - lo) / (hi - lo)
##   s
## }
## <bytecode: 0x55e10eeb7638>
```

this can get you into trouble if you use it in a dataframe. This is a bit complicated, since it has to use list-columns. Here we go:

```
tibble(x = list(z, c(3,3,3)))
```

```
## # A tibble: 2 x 1
##   x
##   <list>
## 1 <dbl [3]>
## 2 <dbl [3]>
```

Just to check that this does contain what you think it does:

```
tibble(x = list(z, c(3,3,3))) %>% unnest(x)
```

```
## # A tibble: 6 x 1
##       x
##   <dbl>
## 1    10
## 2    14
## 3    11
## 4     3
## 5     3
## 6     3
```

So now, for each of those two input vectors, what happens when we run `rescale0` on them? This is `map`:

```
tibble(x = list(z, c(3,3,3))) %>%
  mutate(ans = map(x, ~rescale0(.)))
```

```
## # A tibble: 2 x 2
##   x         ans
##   <list>    <list>
## 1 <dbl [3]> <dbl [3]>
## 2 <dbl [3]> <chr [1]>
```

The first `ans` is a vector of 3 numbers, and the second one is one piece of text (the "error message"). I was actually surprised it got this far. So what happens when we `unnest` the second column?

```
tibble(x = list(z, c(3,3,3))) %>%
  mutate(ans = map(x, ~rescale0(.))) %>%
  unnest(ans)
```

```
## Error: Can't combine `..1$ans` <double> and `..2$ans` <character>.
```

Now we get a confusing error: it's *here* that combining some numbers and some text in one column of a dataframe doesn't work. To forestall this, we need to go back and rewrite `rescale0` to not mix things up. Having it return an error, as the latest version of `rescale` does, gives an error here too, but at least we know what it means:

```
tibble(x = list(z, c(3,3,3))) %>%
  mutate(ans = map(x, ~rescale(.))) %>%
  unnest(ans)
```

```
## Error: Problem with `mutate()` input `ans`.
## x high and low must be different
## i Input `ans` is `map(x, ~rescale(.))`.
```

because this is the error we anticipated: it says "somewhere within the list-column `x` is a vector where everything is the same".

Extra 3a (or wherever we are now): these also work with `rowwise`, if you find that easier to think about:

```
tibble(x = list(z, c(3,3,3))) %>%
  rowwise() %>%
  mutate(ans = list(rescale0(x))) %>%
  unnest(ans)
```

```
## Error: Can't combine `..1$ans` <double> and `..2$ans` <character>.
```

```
tibble(x = list(z, c(3,3,3))) %>%
  rowwise() %>%
  mutate(ans = list(rescale(x))) %>%
  unnest(ans)
```

```
## Error: Problem with `mutate()` input `ans`.
## x high and low must be different
## i Input `ans` is `list(rescale(x))`.
## i The error occurred in row 2.
```

The `rescales` (might) return a vector of several numbers, so you have to wrap it in `list` inside the `mutate` here. Or, do what I did: try it first, and when it doesn't work, then put the `list` around it.

(f) Make a dataframe (containing any numeric values), and in it create a new column containing the rescaled version of one of its columns, using your function. Show your result.

Solution:

This is less difficult than you might be expecting: make a dataframe with at least one numeric column, and use `mutate`:

```
d <- tibble(y=2:6)
d
```

```
## # A tibble: 5 x 1
##       y
##   <int>
## 1     2
## 2     3
## 3     4
## 4     5
## 5     6
```

and then

```
d %>% mutate(s=rescale(y))
```

```
## # A tibble: 5 x 2
##       y     s
##   <int> <dbl>
## 1     2  0
## 2     3  0.25
## 3     4  0.5
## 4     5  0.75
## 5     6  1
```

You can supply the values for what I called `y`, or use random numbers. It's easier for you to check that it has worked if your column playing the role of my `y` has not too many values in it.

Extra: this is actually already in the `tidyverse` under the name `percent_rank` ("percentile ranks"):

```
d %>% mutate(s = percent_rank(y))
```

```
## # A tibble: 5 x 2
##       y     s
##   <int> <dbl>
## 1     2  0
## 2     3  0.25
## 3     4  0.5
## 4     5  0.75
## 5     6  1
```

The value 5, for example, is at the 75th percentile.

(g) We might want to rescale the input not to be between 0 and 1, but between two values `a` and `b` that we specify as input. If `a` and/or `b` are not given, we want to use the values 0 for `a` and 1 for `b`. Rewrite your function to rescale the input to be between `a` and `b` instead of 0 and 1. Hint: allow your function to produce values between 0 and 1 as before, and then note that if all the values in a vector `s` are between 0 and 1, then all the values in `a+(b-a)*s` are between *a* and *b*.

Solution:

I'm showing you my thought process in this one. The answer I want from you is the one at the end.

So, start by copying and pasting what you had before:

```
rescale <- function(x) {
  lo <- min(x)
  hi <- max(x)
  stopifnot(hi != lo)
  (x - lo) / (hi - lo)
}
```

On the top line, add the extra inputs and their default values. I also changed the name of my function, for reasons you'll see later:

```
rescale2 <- function(x, a=0, b=1) {
  lo <- min(x)
  hi <- max(x)
  stopifnot(hi != lo)
  (x - lo) / (hi - lo)
}
```

Save the last line, since we have to do something else with it:

```r
rescale2 <- function(x, a=0, b=1) {
  lo <- min(x)
  hi <- max(x)
  stopifnot(hi != lo)
  s <- (x - lo) / (hi - lo)
}
```

and finally add the calculation in the hint, which we don't need to save because we are returning it:

```r
rescale2 <- function(x, a=0, b=1) {
  lo <- min(x)
  hi <- max(x)
  stopifnot(hi != lo)
  s <- (x - lo) / (hi - lo)
  a + (b-a) * s
}
```

This complete function is what I want to see from you. (You should keep the `stopifnot`, because this function will have the exact same problem as the previous one if all the values in `x` are the same.)

A better way is to observe that you can call functions inside functions. The function above is now a bit messy since it has several steps. Something that corresponds better to my hint is to call the original `rescale` first, and then modify its result:

```r
rescale3 <- function(x, a=0, b=1) {
  s <- rescale(x)
  a + (b-a) * s
}
```

The logic to this is rather clearly "rescale the input to be between 0 and 1, then rescale *that* to be between a and b." My `rescale2` does exactly the same thing, but it's much less clear that it does so, unless you happen to have in your head how `rescale` works. (I think you are more likely to remember, sometime in the future, what `rescale` *does*, compared to precisely how it *works*.)

That is why `rescale3` is better than `rescale2`. Remember that you can, and generally should, use functions that have already been written (by you or someone else) as part of functions that do more complex things. See also my second point below.

Extra: there are two important principles about why functions are important:

1. they allow you to re-do a calculation on many different inputs (the point I've been making up to now)
2. by abstracting a calculation into a thing with a name, it makes it easier to understand that calculation's role in something bigger. The only thing we had to remember in `rescale3` is what the last line did, because the name of the function called on the first line tells us what happens there. This is much easier than remembering what the first four lines of `rescale2` do.

The second principle here is what psychologists call "chunking": you view a thing like my function `rescale` as a single item, rather than as four separate lines of code, and then that single item can be part of something larger (like my `rescale3`), and you have a smaller number of things to keep track of.

(h) Test your new function two or more times, on input where you know or can guess what the output is going to be. In each case, explain briefly why your output makes sense.

Solution:

I'll start by using the default values for `a` and `b` (so I don't have to specify them):

```
rescale2(2:6)
```

```
## [1] 0.00 0.25 0.50 0.75 1.00
```

```
rescale3(2:6)
```

```
## [1] 0.00 0.25 0.50 0.75 1.00
```

I did both of the variants of my function; of course, you'll only have one variant.

We got the same answer as before for the same input, so the default values $a = 0, b = 1$ look as if they have been used.

Let's try a different one:

```
v <- c(7, 11, 12)
rescale2(v, 10, 30)
```

```
## [1] 10 26 30
```

```
rescale3(v, 10, 30)
```

```
## [1] 10 26 30
```

The lowest value in `v` has become 10, and the highest has become 30. (Also, the in-between value 11 was closer to 12 than to 7, and it has become something closer to 30 than to 10.)

Extra: you can also name any of your inputs:

```
rescale3(x=v, a=10, b=30)
```

```
## [1] 10 26 30
```

and if you name them, you can shuffle the order too:

```
rescale3(a=10, b=30, x=v)
```

```
## [1] 10 26 30
```

The point of doing more than one test is to check that different aspects of your function all work. Therefore, the best testing here checks that the defaults work, and that the answer is sensible for some different `a` and `b` (to check that this works as well).

When you write your version of `rescale` with the optional inputs, it's best if you do it so that the things you have to supply (the vector of numbers) is *first*. If you put `a` and `b` first, when you want to omit them, you'll have to call the input vector by name, like this:

```
rescale3(x=v)
```

```
## [1] 0.0 0.8 1.0
```

because otherwise the input vector will be taken to be `a`, not what you want.

2. A company keeps track of the contracts it makes with its suppliers. On November 15 of each year, it looks at all the current contracts and decides what the status of each contract is. Seven of the current contracts and their original negotiation dates are as shown in http://ritsokiguess.site/STAD29/contracts.csv. The

company has asked you to work out the status of each of these contracts, according to the following scheme:

- anything up to and including November 15, 2013: “expired”
- after that and up to and including Nov 15, 2014: “renegotiate”
- after that and up to and including Nov 15, 2016: “arbitration”
- after that, “current”, with month name and year

The company says that if you can get this one to work, they will give you the entire file, containing several thousand contracts, to work out the statuses of!

(a) Read in and display the data.

Solution:

The exact usual thing:

```
my_url <- "http://ritsokiguess.site/STAD29/contracts.csv"
contracts <- read_csv(my_url)
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   contract = col_double(),
##   date = col_character()
## )
```

```
contracts
```

```
## # A tibble: 7 x 2
##   contract date
##      <dbl> <chr>
## 1  5829014 November 6, 2013
## 2  2301911 January 23, 2014
## 3  1540956 December 1, 2014
## 4  6051271 April 11, 2015
## 5  9330471 September 21, 2015
## 6  6894300 August 21, 2016
## 7  7465502 November 18, 2016
```

Extra: the original data file came with statuses as well (computed by someone using SAS). I copied and pasted it, and wanted to get it into a data frame. I did this by first pasting it into a piece of text:

```
data_txt <- "
contract status        date
5829014  EXPIRED       November 6, 2013
2301911  RENEGOTIATION January 23, 2014
1540956  ARBITRATION   December 1, 2014
6051271  ARBITRATION   April 11, 2015
9330471  ARBITRATION   September 21, 2015
6894300  ARBITRATION   August 21, 2016
7465502  NOV2016       November 18, 2016
"
```

This is aligned columns (think `read_table`) and in fact you can read from a piece of text just as easily as from a file (which is why I wanted to show you this):

```
contracts0 <- read_table(data_txt)
contracts0
```

```
## # A tibble: 7 x 3
##   contract status       date
##      <dbl> <chr>        <chr>
## 1  5829014 EXPIRED      November 6, 2013
## 2  2301911 RENEGOTIATION January 23, 2014
## 3  1540956 ARBITRATION  December 1, 2014
## 4  6051271 ARBITRATION  April 11, 2015
## 5  9330471 ARBITRATION  September 21, 2015
## 6  6894300 ARBITRATION  August 21, 2016
## 7  7465502 NOV2016      November 18, 2016
```

There is also a package called `datapasta` (see here) that is nice for creating dataframes and other things out of copy-pasted text.

(b) Explain briefly why you will not be able to use your dataframe as it currently stands to work out the status of each contract.

Solution:

The column `date` in our current dataframe is text rather than an actual date, and we need actual dates in order to make the comparisons required to work out the statuses. You need to make the observation that we are *comparing* dates, and we cannot do that if the dates are text.

Extra: if the dates were something like 2020-08-12, you could compare them as text and get the right answer, but these are not. That's another reason to format your dates consistently as year-month-day.

(c) Create and display a column of dates that you will be able to work with in computing the status of each contract.

Solution:

For this, you will need `lubridate`, installed and loaded. Give the column of actual dates whatever name you like. I realized afterwards that a shorter name would have been better, since I was going to be typing it a lot:

```
contracts %>% mutate(date_date = mdy(date))
```

```
## # A tibble: 7 x 3
##   contract date               date_date
##      <dbl> <chr>              <date>
## 1  5829014 November 6, 2013   2013-11-06
## 2  2301911 January 23, 2014   2014-01-23
## 3  1540956 December 1, 2014   2014-12-01
## 4  6051271 April 11, 2015     2015-04-11
## 5  9330471 September 21, 2015 2015-09-21
## 6  6894300 August 21, 2016    2016-08-21
## 7  7465502 November 18, 2016  2016-11-18
```

The header of my `date_date` column shows that it is indeed a `date`. (It may look as if the text has just been reformatted, but it is indeed a date.)

(d) Use `case_when` to create a column in your dataframe containing the status of each contract. Display your results. (Recalling exactly how `case_when` works may make your coding easier.)

Solution:

There are two things to remember about `case_when` that will help you here:

- the *first* of the conditions that is true will provide the answer, so that if you get to the second one, it must mean that the first one was *false*. This is like if-elif-elif-else in Python.
- the catch-all condition `TRUE` means, in English, "otherwise".

Strategy-wise, therefore, if you start from one end of the range of dates, then you don't need to worry about "between". The best answer, therefore, looks something like this:

```
contracts %>% mutate(date_date = mdy(date)) %>%
  mutate(status = case_when(
    date_date <= ymd("2013-11-15") ~ "expired",
    date_date <= ymd("2014-11-15") ~ "renegotiate",
    date_date <= ymd("2016-11-15") ~ "arbitration",
    TRUE                           ~ str_c("current ",
                                         month(date_date, label = TRUE),
                                         year(date_date))
  )) -> contracts
contracts
```

```
## # A tibble: 7 x 4
##   contract date               date_date  status
##      <dbl> <chr>              <date>     <chr>
## 1  5829014 November 6, 2013   2013-11-06 expired
## 2  2301911 January 23, 2014   2014-01-23 renegotiate
## 3  1540956 December 1, 2014   2014-12-01 arbitration
## 4  6051271 April 11, 2015     2015-04-11 arbitration
## 5  9330471 September 21, 2015 2015-09-21 arbitration
## 6  6894300 August 21, 2016    2016-08-21 arbitration
## 7  7465502 November 18, 2016  2016-11-18 current Nov2016
```

By the time your code gets to the second line, it already knows that the date is later than Nov. 15, 2013 so you don't have to test that again. If you do, it reveals that you don't really know how `case_when` works. If you are in that situation, the best way to proceed is to use `between`, which you will have to find out about (and, as ever, cite your source) in order to get the job done, which is better than not getting it done. Going this way will make your code a *lot* messier, though.

The other thing to remember is that you have to compare a date *with another date*, not with a piece of text that looks like a date, so that you have to make those Nov 15s into dates. You can write them in any format and convert them with `ymd` or `mdy` or whatever is appropriate, or you can write them in ISO format as I did and use `as.Date` (a base R function) instead of `ymd`.

With that in mind, you might want to pre-compute those key dates (and give them short names), something like this:

```
exp_date <- ymd("2013-11-15")
reneg_date <- ymd("2014-11-15")
arb_date <- ymd("2016-11-15")
contracts %>% mutate(date_date = mdy(date)) %>%
  mutate(status = case_when(
    date_date <= exp_date   ~ "expired",
    date_date <= reneg_date ~ "renegotiate",
    date_date <= arb_date   ~ "arbitration",
    TRUE                    ~ str_c("current ",
                                   month(date_date, label = TRUE),
                                   year(date_date))
  )) -> contracts2
contracts2
```

```
## # A tibble: 7 x 4
##   contract date               date_date  status
##      <dbl> <chr>              <date>     <chr>
## 1  5829014 November 6, 2013   2013-11-06 expired
## 2  2301911 January 23, 2014   2014-01-23 renegotiate
## 3  1540956 December 1, 2014   2014-12-01 arbitration
## 4  6051271 April 11, 2015     2015-04-11 arbitration
## 5  9330471 September 21, 2015 2015-09-21 arbitration
## 6  6894300 August 21, 2016    2016-08-21 arbitration
## 7  7465502 November 18, 2016  2016-11-18 current Nov2016
```

This would also make it easier to edit your code if any of the key dates changed later. For this one, though, the key dates only appear once, and it doesn't make the `case_when` much longer if they are in it rather than pre-computed, so I don't have any strong feelings either way.

For the contracts that are current, you need to get hold of three things:

- the text `current`
- the *name* of the month, which is `month` with `label=TRUE`
- the year, which is a straight `year`.

Then you have to glue[1] these together into one piece of text, for which I prefer `str_c` from `stringr` (in the `tidyverse`), but you can also use `paste` or `paste0` if you can find out how they work.

Also, you could start from the latest date and work the other way:

```
contracts %>% mutate(date_date = mdy(date)) %>%
  mutate(status = case_when(
    date_date > mdy("Nov 15, 2016") ~ str_c("current ",
                                           month(date_date, label = TRUE),
                                           year(date_date)),
    date_date > mdy("Nov 15, 2014") ~ "arbitration",
    date_date > mdy("Nov 15, 2013") ~ "renegotiate",
    TRUE                            ~ "expired"
  ))
```

```
## # A tibble: 7 x 4
##   contract date               date_date  status
##      <dbl> <chr>              <date>     <chr>
## 1  5829014 November 6, 2013   2013-11-06 expired
## 2  2301911 January 23, 2014   2014-01-23 renegotiate
## 3  1540956 December 1, 2014   2014-12-01 arbitration
## 4  6051271 April 11, 2015     2015-04-11 arbitration
## 5  9330471 September 21, 2015 2015-09-21 arbitration
## 6  6894300 August 21, 2016    2016-08-21 arbitration
## 7  7465502 November 18, 2016  2016-11-18 current Nov2016
```

Getting this answer with a strategy like one of these is best.

It is good coding practice, in your `case_when`, to line up your `~`s, so that it is easy to see what you are testing and what the result is if that thing is true. It is also smart to split up long lines, like the ones with `str_c` in them, and also to line up the three inputs to `str_c` or `paste`. This is rather directly for the benefit of your grader,[2] but also in the real world is for the benefit of your colleagues and (something that is easy to forget) *you* in the future, when you come back to it in six months or six years to modify something.

Extra 1: an alternative to using `str_c` or `paste` is called `glue`. It is a package that you will need to install first, containing a function of the same name. Here's how it works:

```
library(glue)
exp_date <- ymd("2013-11-15")
reneg_date <- ymd("2014-11-15")
arb_date <- ymd("2016-11-15")
contracts %>% mutate(date_date = mdy(date)) %>%
  mutate(if_current = as.character(glue("current {month(date_date, label = TRUE)} {year(date_d
  mutate(status = case_when(
    date_date <= exp_date   ~ "expired",
    date_date <= reneg_date ~ "renegotiate",
    date_date <= arb_date   ~ "arbitration",
    TRUE                    ~ if_current
  )) -> contracts2
contracts2
```

```
## # A tibble: 7 x 5
##   contract date               date_date  status           if_current
##      <dbl> <chr>              <date>     <chr>            <chr>
## 1  5829014 November 6, 2013   2013-11-06 expired          current Nov 2013
## 2  2301911 January 23, 2014   2014-01-23 renegotiate      current Jan 2014
## 3  1540956 December 1, 2014   2014-12-01 arbitration      current Dec 2014
## 4  6051271 April 11, 2015     2015-04-11 arbitration      current Apr 2015
## 5  9330471 September 21, 2015 2015-09-21 arbitration      current Sep 2015
## 6  6894300 August 21, 2016    2016-08-21 arbitration      current Aug 2016
## 7  7465502 November 18, 2016  2016-11-18 current Nov 2016 current Nov 2016
```

This turned out to be more fiddly than I would have liked, but there are some lessons that are worth learning:

- `glue` itself takes a single piece of text (unlike the others such as `str_c`, which take a mixture of text and variable values). Any literal text (like `current`) is used as is. Anything in curly brackets, like `year(date_date)`, is evaluated as if it were in a code chunk, and the value is inserted into the answer. This one, therefore, makes a piece of text (but see the next point) out of the literal text `current`, the name of the month, and the year as a number. Is this clearer than `str_c`? I dunno. Up to you.

- The output from `glue` actually has a special class `glue/character` which is not the same as `character`. `case_when` is very picky: all the things on the right side of the squiggles have to be of *exactly* the same class. The others are ordinary text, so I have to make this one into ordinary text as well, which is the reason for the `as.character` around the outside. You might be thinking "how did he figure that out ahead of time?", and the answer is that I didn't: the first time I had it without the `as.character` and it didn't work:

```
contracts %>% mutate(date_date = mdy(date)) %>%
  mutate(if_current = glue("current {month(date_date, label = TRUE)} {year(date_date)}")) %>%
  mutate(status = case_when(
    date_date <= exp_date    ~ "expired",
    date_date <= reneg_date ~ "renegotiate",
    date_date <= arb_date    ~ "arbitration",
    TRUE                     ~ if_current
  )) -> contracts2
```

```
## Error: Problem with `mutate()` input `status`.
## x must have class `character`, not class `glue/character`.
## i Input `status` is `case_when(...)`.
```

This gave me the hint that something was up with the output from `glue`. I remembered that `case_when`, like the other things in the `tidyverse`, is picky about types, having recently run into trouble with "types of `NA`" (yes, there is such a thing). So I turned it into actual text and then it worked. As ever, try it, and if it fails, figure out why and then fix it. Nobody will know or care how many errors you made if you fix them all before you hand your work in.

- the `glue` call is rather long, and not easy to split up, and I thought it was too long to go into the `case_when`. So what I did was to pre-compute it: I created a column that was the months and years for each contract, *in case* I needed it. This might strike you as wasteful, since I might not need it for certain contracts. Make a call for yourself about whether it's better to compute it for all the contracts and just grab it in the `case_when` if you happen to need it, or whether you think it's important to only compute it when it's needed, and then have rather a long and hard-to-read last line on the `case_when`. It's a trade-off, and there are no right answers.

Extra 2: if you are unable to get `case_when` working, you can do this with nested `ifelse`s, though I think you'll need to format this carefully to get the logic to make sense. `ifelse` takes three things: something that is true or false, the value if the thing is true, and the value if it is false, like this:

```
x <- 3
ifelse(x==3, "three", "not three")
```

```
## [1] "three"
```

```
x <- 4
ifelse(x==3, "three", "not three")
```

```
## [1] "not three"
```

In Python, or indeed in `case_when`, you can have more than two choices, by using `elif` or by adding another line to your `case_when`. What you do with `ifelse` is to replace the last input by *another* `ifelse`:

```
x <- 3
ifelse(x==3, "three", ifelse(x==4, "four", "something else"))
```

```
## [1] "three"
```

```
x <- 4
ifelse(x==3, "three", ifelse(x==4, "four", "something else"))
```

```
## [1] "four"
```

or perhaps better written as

```
x <- 5
ifelse(
  x==3, "three",
  ifelse(x==4, "four", "something else"))
```

```
## [1] "something else"
```

For our actual example, that might go like this:

```
exp_date <- ymd("2013-11-15")
reneg_date <- ymd("2014-11-15")
arb_date <- ymd("2016-11-15")
contracts %>% mutate(date_date = mdy(date)) %>%
  mutate(if_current = as.character(glue("current {month(date_date, label = TRUE)} {year(date_d
  mutate(status = ifelse(
    date_date <= exp_date,           "expired",
    ifelse(date_date <= reneg_date , "renegotiate",
    ifelse(date_date <= arb_date   , "arbitration", if_current
  )))) -> contracts2
contracts2
```

```
## # A tibble: 7 x 5
##   contract date               date_date  status           if_current
##      <dbl> <chr>              <date>     <chr>            <chr>
## 1  5829014 November 6, 2013   2013-11-06 expired          current Nov 2013
## 2  2301911 January 23, 2014   2014-01-23 renegotiate      current Jan 2014
## 3  1540956 December 1, 2014   2014-12-01 arbitration      current Dec 2014
## 4  6051271 April 11, 2015     2015-04-11 arbitration      current Apr 2015
## 5  9330471 September 21, 2015 2015-09-21 arbitration      current Sep 2015
## 6  6894300 August 21, 2016    2016-08-21 arbitration      current Aug 2016
## 7  7465502 November 18, 2016  2016-11-18 current Nov 2016 current Nov 2016
```

On the `mutate` with the `ifelse` in it, you'll need to make sure you close the right number of brackets at the end, since each `ifelse` opens a new bracket. This also works with `str_c` or `paste` rather than `glue`.

(e) For the future, the company wants to define the status of a contract based on the number of years ago it was negotiated. Create and display a new column in your dataframe that contains the number of years between the negotiation date and today's date. This could be completed years or decimal years; either one is good.

Solution:

There are a couple of ways to get today's date. One way is to look at your phone, and make an R date from the date you see there. When I did this originally, it was August 6, 2020, thus:

```
d <- ymd("2020-08-06")
d
```

```
## [1] "2020-08-06"
```

```
class(d)
```

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

which doesn't look as if it did much, but the last line verifies that it is indeed a Date.

A better way is to recognize that `lubridate` will probably have something that gives you today's date more easily, and it does:

```
today()
```

```
## [1] "2020-12-17"
```

or there is the no-`lubridate`-required base-R way:

```
Sys.Date()
```

```
## [1] "2020-12-17"
```

If you found out about these somewhere, tell me where.

Then take the difference between today and the dates in your dataframe:

```
contracts %>%
  mutate(since_date = today()-date_date)
```

```
## # A tibble: 7 x 5
##   contract date               date_date  status           since_date
##      <dbl> <chr>              <date>     <chr>            <drtn>
## 1  5829014 November 6, 2013   2013-11-06 expired          2598 days
## 2  2301911 January 23, 2014   2014-01-23 renegotiate      2520 days
## 3  1540956 December 1, 2014   2014-12-01 arbitration      2208 days
## 4  6051271 April 11, 2015     2015-04-11 arbitration      2077 days
## 5  9330471 September 21, 2015 2015-09-21 arbitration      1914 days
## 6  6894300 August 21, 2016    2016-08-21 arbitration      1579 days
## 7  7465502 November 18, 2016  2016-11-18 current Nov2016 1490 days
```

(it chose the units), and convert that into fractional years by dividing by the number of days in a year:

```
contracts %>%
  mutate(since_date = today()-date_date) %>%
  mutate(years = since_date  / dyears(1))
```

```
## # A tibble: 7 x 6
##   contract date               date_date  status           since_date years
##      <dbl> <chr>              <date>     <chr>            <drtn>     <dbl>
## 1  5829014 November 6, 2013   2013-11-06 expired          2598 days   7.11
## 2  2301911 January 23, 2014   2014-01-23 renegotiate      2520 days   6.90
## 3  1540956 December 1, 2014   2014-12-01 arbitration     2208 days   6.05
## 4  6051271 April 11, 2015     2015-04-11 arbitration     2077 days   5.69
## 5  9330471 September 21, 2015 2015-09-21 arbitration     1914 days   5.24
## 6  6894300 August 21, 2016    2016-08-21 arbitration     1579 days   4.32
## 7  7465502 November 18, 2016  2016-11-18 current Nov2016 1490 days   4.08
```

Extra: another way, also good, is to get the number of completed years since the contract was negotiated. This is also of value to the company since they can decide what to do on the basis of this being greater than or equal to something; the fractional part is not necessarily needed.

The way to solve this is to make what `lubridate` calls a "period", in which the start and end are also stored, as per lecture 23a. Here, that goes like this:

```
contracts %>%
  mutate(since_date = as.period(date_date %--% today())) %>%
  mutate(years = year(since_date))
```

```
## # A tibble: 7 x 6
##   contract date               date_date  status          since_date          years
##      <dbl> <chr>              <date>     <chr>           <Period>            <int>
## 1  5829014 November 6, 2013   2013-11-06 expired         7y 1m 11d 0H 0M 0S      7
## 2  2301911 January 23, 2014   2014-01-23 renegotiate     6y 10m 24d 0H 0M 0S     6
## 3  1540956 December 1, 2014   2014-12-01 arbitration     6y 0m 16d 0H 0M 0S      6
## 4  6051271 April 11, 2015     2015-04-11 arbitration     5y 8m 6d 0H 0M 0S       5
## 5  9330471 September 21, 20~  2015-09-21 arbitration     5y 2m 26d 0H 0M 0S      5
## 6  6894300 August 21, 2016    2016-08-21 arbitration     4y 3m 26d 0H 0M 0S      4
## 7  7465502 November 18, 2016  2016-11-18 current Nov20~  4y 0m 29d 0H 0M 0S      4
```

The values of `since_date` now are a number of years, months, and days, and these are now exact because the start and end dates are known, so that `lubridate` knows whether the periods in question include any February 29 dates and can account for them.

As far as I am concerned, either way of figuring out the years is equally good.

Extra extra: this works because the times-since-negotiation have the years as the biggest time unit (there aren't any decades or anything like that). Note that these are completed years, not rounded-off years. If you wanted completed *months*,[3] you'd have to treat years as 12 months, thus:

```
contracts %>%
  mutate(since_date = as.period(date_date %--% today())) %>%
  mutate(months = month(since_date) + 12*year(since_date))
```

```
## # A tibble: 7 x 6
##   contract date               date_date  status         since_date          months
##      <dbl> <chr>              <date>     <chr>          <Period>             <dbl>
## 1  5829014 November 6, 2013   2013-11-06 expired        7y 1m 11d 0H 0M 0S      85
## 2  2301911 January 23, 2014   2014-01-23 renegotiate    6y 10m 24d 0H 0M 0S     82
## 3  1540956 December 1, 2014   2014-12-01 arbitration    6y 0m 16d 0H 0M 0S      72
## 4  6051271 April 11, 2015     2015-04-11 arbitration    5y 8m 6d 0H 0M 0S       68
## 5  9330471 September 21, 20~  2015-09-21 arbitration    5y 2m 26d 0H 0M 0S      62
## 6  6894300 August 21, 2016    2016-08-21 arbitration    4y 3m 26d 0H 0M 0S      51
## 7  7465502 November 18, 2016  2016-11-18 current Nov2~  4y 0m 29d 0H 0M 0S      48
```

Eyeballing the years and months, and doing some quick mental calculations, suggests that those numbers in the `months` column are correct.

Notes

1. There is also a package called “glue” that does exactly this. I show you this later.

2. If your code is messy, the grader, being human, is likely to check it carefully for errors. If you don’t show attention to detail on the layout of your code, it will make people wonder what else you don’t show attention to detail on.

3. Which you might do if the company policy was something like renegotiating contracts after 66 months (5 and a half years).