Data Wrangling III: Working with strings and dates

EDH7916 | Spring 2020

Benjamin Skinner

The data we've used so far has been almost entirely numerical. Even when the field represented an expected level of education, for example, we didn't see "complete a Bachelor's degree." Instead, we saw **6**. In the few cases in which we've seen strings or dates, the values have been very regular.

Much education-related data, however, are not this uniform. Particularly when using administrative data, you are likely to read in columns that contain *unstructured strings*: names, addresses, dates, *etc.* Why are they unstructured? Almost always, the answer is that person who initially inputted the data neither had a dropdown menu of options to choose from nor separate fields for each part of the data element (*e.g.*, `first name`, `last name`). Instead, they have a blank field in which they type:

Enter name: `Isaiah Berlin`

Why is this a problem? With an open field, the variations are (often) unlimited:

- `I. Berlin`
- `isaiah berlin`
- `Berlin, Isaiah`

Similarly, the same date can be written any number of ways:

Enter date: `February 11, 2020`

- `11 February 2020`
- `11 Feb 2020`
- `Feb. 11, 2020`
- `2/11/2020` (American)
- `11/2/2020` (most everyone else)
- `2/11/20`

Imagine having 1,000 or 1 million rows of such data and needing to pull out only last names or the month. When data are irregular, that can be an impossible task to do by hand.

To be clear, this is not to impugn those who enter the data. Rather, it's an acknowledgment that the original uses of the data we analyze may differ from our own: compliance with an administrative task vs. data input for statistical analysis, for example.

You won't always need to work with strings and dates, but when you do, having a few specialty tools in your toolbox will be greatly beneficial. Sometimes they can mean the difference between being able vs. unable to answer your question.

Setup

As before, we'll continue working within the {tidyverse}. We'll focus, however, on using two specific libraries:

- {stringr} for strings
- {lubridate} for dates

You may have noticed already that when we load the {tidyverse} library with `library(tidyverse)`, the {stringr} library is already loaded. The {lubridate} library, though part of the {tidyverse}, is not. We need to load it separately.

```
## ---------------------------
## libraries
## ---------------------------

## NB: The {stringr} library is loaded with {tidyverse}, but
## {lubridate} is not, so we need to load it separately

library(tidyverse)
```

```
── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──

✔ ggplot2 3.2.1     ✔ purrr   0.3.3
✔ tibble  2.1.3     ✔ dplyr   0.8.4
✔ tidyr   1.0.2     ✔ stringr 1.4.0
✔ readr   1.3.1     ✔ forcats 0.4.0

── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
```

```
library(lubridate)
```

```
Attaching package: 'lubridate'

The following object is masked from 'package:base':

    date
```

NB: As we have done in the past few lessons, we'll run this script assuming that our working directory is set to the `scripts` directory.

```
## ---------------------------
## directory paths
## ---------------------------

## assume we're running this script from the ./scripts subdirectory
dat_dir <- file.path("..", "data")
```

Part 1: Working with strings

To practice working with strings, we'll use data from Integrated Postsecondary Education Data System (IPEDS):

> The National Center for Education Statistics (NCES) administers the Integrated Postsecondary Education Data System (IPEDS), which is a large-scale survey that collects institution-level data from postsecondary institutions in the United States (50 states and the District of Columbia) and other U.S. jurisdictions. IPEDS defines a postsecondary institution as an organization that is open to the public and has the provision of postsecondary education or training beyond the high school level as one of its primary missions. This definition includes institutions that offer academic, vocational and continuing professional education programs and excludes institutions that offer only avocational (leisure) and adult basic education programs. Definitions for other terms used in this report may be found in the IPEDS online glossary.

> NCES annually releases national-level statistics on postsecondary institutions based on the IPEDS data. National statistics include tuition and fees, number and types of degrees and certificates conferred, number of students applying and enrolled, number of employees, financial statistics, graduation rates, student outcomes, student financial aid, and academic libraries.

You can find more information about IPEDS here. As higher education scholars, IPEDS data are a valuable resource that you may often turn to (I do).

We'll use one file (which can be found here), that covers institutional characteristics for one year:

- Directory information, 2007 (`hd2007.csv`)

```
## ---------------------------
## input
## ---------------------------

## read in data and lower all names using rename_all(tolower)
df <- read_csv(file.path(dat_dir, "hd2007.csv")) %>%
    rename_all(tolower)
```

```
Parsed with column specification:
cols(
  .default = col_double(),
  INSTNM = col_character(),
  ADDR = col_character(),
  CITY = col_character(),
  STABBR = col_character(),
  ZIP = col_character(),
  CHFNM = col_character(),
  CHFTITLE = col_character(),
  EIN = col_character(),
  OPEID = col_character(),
  WEBADDR = col_character(),
  ADMINURL = col_character(),
  FAIDURL = col_character(),
  APPLURL = col_character(),
  ACT = col_character(),
  CLOSEDAT = col_character(),
  IALIAS = col_character()
)

See spec(...) for full column specifications.
```

Finding: `str_detect()`

So far, we've filtered data using {dplyr}'s `filter()` verb. When matching a string, we have used `==` (or `!=` for negative match). For example, if we wanted to limit our data to only those institutions in Florida, we could filter using the `stabbr` column:

```
## filter using state abbreviation (not saving, just viewing)
df %>%
    filter(stabbr == "FL")
```

```
# A tibble: 316 x 59
   unitid instnm addr  city  stabbr zip    fips obereg chfnm chftitle gentele
    <dbl> <chr>  <chr> <chr> <chr>  <chr> <dbl>  <dbl> <chr> <chr>      <dbl>
 1 132268 Wyote… 470 … Ormo… FL     32174    12      5 Stev… Preside… 3.86e12
```

```
 2 132338 The A… 1799… Fort… FL     3331…    12      5 Char… Preside… 9.54e13
 3 132374 Atlan… 4700… Coco… FL     3306…    12      5 Robe… Director 7.54e 9
 4 132408 The B… 5400… Grac… FL     32440    12      5 Thom… Preside… 8.50e 9
 5 132471 Barry… 1130… Miami FL     3316…    12      5 Sist… Preside… 8.01e 9
 6 132523 Goodi… 615 … Pana… FL     32401    12      5 Dr. … CRNA Ph… 8.51e 9
 7 132602 Bethu… 640 … Dayt… FL     3211…    12      5 Dr T… Preside… 3.86e 9
 8 132657 Lynn … 3601… Boca… FL     3343…    12      5 Kevi… Preside… 5.61e 9
 9 132666 Brade… 5505… Brad… FL     34209    12      5 A. P… CEO      9.42e 9
10 132675 Bradf… 609 … Star… FL     32091    12      5 Rand… Director 9.05e 9
# … with 306 more rows, and 48 more variables: ein <chr>, opeid <chr>,
#   opeflag <dbl>, webaddr <chr>, adminurl <chr>, faidurl <chr>, applurl <chr>,
#   sector <dbl>, iclevel <dbl>, control <dbl>, hloffer <dbl>, ugoffer <dbl>,
#   groffer <dbl>, fpoffer <dbl>, hdegoffr <dbl>, deggrant <dbl>, hbcu <dbl>,
#   hospital <dbl>, medical <dbl>, tribal <dbl>, locale <dbl>, openpubl <dbl>,
#   act <chr>, newid <dbl>, deathyr <dbl>, closedat <chr>, cyactive <dbl>,
#   postsec <dbl>, pseflag <dbl>, pset4flg <dbl>, rptmth <dbl>, ialias <chr>,
#   instcat <dbl>, ccbasic <dbl>, ccipug <dbl>, ccipgrad <dbl>, ccugprof <dbl>,
#   ccenrprf <dbl>, ccsizset <dbl>, carnegie <dbl>, tenursys <dbl>,
#   landgrnt <dbl>, instsize <dbl>, cbsa <dbl>, cbsatype <dbl>, csa <dbl>,
#   necta <dbl>, dfrcgid <dbl>
```

This works well because the `stabbr` column, even though it uses strings, is regular. But what happens when the strings aren't so regular? For example, let's look the different titles chief college administrators take.

```
## see first few rows of distinct chief titles
df %>%
    distinct(chftitle)
```

```
# A tibble: 556 x 1
   chftitle
   <chr>
 1 Commandant
 2 President
 3 Chancellor
 4 Interim President
 5 CEO
 6 Acting President
 7 Director
 8 President/CEO
 9 Interim Chancellor
10 President/COO
# … with 546 more rows
```

We find over 500 unique titles. Just looking at the first 10 rows, we see that some titles are pretty similar — *President* vs. *CEO* vs. *President/CEO* — but not exactly the same. Let's look again, but this time get counts of each distinct title and arrange from most common to least.

```
## return the most common titles
df %>%
    ## get counts of each type
    count(chftitle) %>%
    ## arrange in descending order so we see most popular at top
    arrange(desc(n))
```

```
# A tibble: 556 x 2
   chftitle               n
   <chr>              <int>
```

```
 1 President           3840
 2 Director             560
 3 Chancellor           265
 4 Executive Director   209
 5 Owner                164
 6 Campus President     116
 7 Superintendent       105
 8 CEO                   90
 9 <NA>                  85
10 Interim President     75
# … with 546 more rows
```

> **Quick exercise** What do you notice about the data frames returned by `distinct()` and `count()`? What's the same? What does `count()` do that `distinct()` does not?

Getting our counts and arranging, we can see that *President* is by far the most common title. That said, we also see *Campus President* and *Interim President* (and before we saw *Acting President* as well).

If your research question asked, *how many chief administrators use the title of "President"?* regardless the various iterations, you can't really use a simple == filter any more. In theory, you could inspect your data, find the unique versions, get counts of each of those using ==, and then sum them up — but that's a lot of work!

Instead, we can use the stringr function `str_detect()`, which looks for a *pattern* — in our case "President" — anywhere in the title.

```
## how many use some form of the title president?
df %>%
    ## still starting with our count
    count(chftitle) %>%
    ## ...but keeping only those titles that contain "President"
    filter(str_detect(chftitle, "President")) %>%
    ## arranging as before
    arrange(desc(n))
```

```
# A tibble: 173 x 2
   chftitle              n
   <chr>             <int>
 1 President          3840
 2 Campus President    116
 3 Interim President    75
 4 President/COO        47
 5 President/CEO        46
 6 School President     31
 7 Vice President       29
 8 President and CEO    17
 9 College President    15
10 President & CEO      14
# … with 163 more rows
```

Now we're seeing many more versions. We can even more clearly see a few titles that are almost certainly the same title, but were just inputted differently — *President/CEO* vs. *President and CEO* vs. *President & CEO*.

Quick exercise Ignoring the sub-counts of the various versions, how many chief administrators have the word "President" in their title?

Seeing the different versions of basically the same title should have us stopping to think: since it seems that this data column contains free form input (*e.g.* **Input chief administrator title:**), maybe we should allow for typos? The easiest: Is there any reason to assume that "President" will be capitalized?

Quick exercise What happens if we search for "president" with a lowercase "p"?

Ah! We find a few stragglers. How can we restructure our filter so that we get these, too? There are at least two solutions.

1. Use regular expressions

Regular expressions (aka *regex*) are special strings that use a particular syntax to create patterns that can be used to match other strings. They are very useful when you need to match strings that have some general form, but may differ in specifics.

We already used this technique in the last lesson when we matched columns in the `all_schools_wide.csv` with `contains("19")` so that we could `pivot_longer()`. Instead of naming all the columns specifically, we recognized that each column took the form of `<test>_19<YY>`. This is a type of regular expression.

In the {tidyverse} some of the {stringr} and {tidyselect} helper functions abstract-away some of the nitty-gritty behind regular expressions. Knowing a little about regular expression syntax, particularly how it is used in R, can go a long way.

In our first case, we can match strings that have a capital **P** *President* or lowercase **p** *president* using square brackets (`[]`). If we want either "P" or "p", then we can use the regex, `[Pp]`, in place of the first character: `"[Pp]resident"`. This will match either "President" or "president".

```
## solution 1: look for either P or p
df %>%
    count(chftitle) %>%
    filter(str_detect(chftitle, "[Pp]resident")) %>%
    arrange(desc(n))
```

```
# A tibble: 175 x 2
   chftitle              n
   <chr>             <int>
 1 President          3840
 2 Campus President    116
 3 Interim President    75
 4 President/COO        47
 5 President/CEO        46
 6 School President     31
 7 Vice President       29
 8 President and CEO    17
 9 College President    15
10 President & CEO      14
# … with 165 more rows
```

2. Put everything in the same case and match with that case

Another solution, which is probably much easier in this particular case, is to set all potential values in `chftitle` to the same case and then match using that case. In many situations, this is preferable since you don't need to guess cases up front.

We won't change the values in `chftitle` permanently — only while filtering. To compare apples to apples (rather than Apples to apples), we'll wrap our column name with the function `str_to_lower()`, which will make character lowercase, and match using lowercase `"president"`.

```
## solution 2: make everything lowercase so that case doesn't matter
df %>%
    count(chftitle) %>%
    filter(str_detect(str_to_lower(chftitle), "president")) %>%
    arrange(desc(n))
```

```
# A tibble: 177 x 2
   chftitle              n
   <chr>             <int>
 1 President          3840
 2 Campus President    116
 3 Interim President    75
 4 President/COO        47
 5 President/CEO        46
 6 School President     31
 7 Vice President       29
 8 President and CEO    17
 9 College President    15
10 President & CEO      14
# … with 167 more rows
```

We recover an additional two titles when using this second solution. Clearly, our first solution didn't account for other cases (perhaps *PRESIDENT*?).

In general, I find it's a good idea to try a solution like the second one before a more complicated one like the first. But because every problem is different, so too are the solutions. You may find yourself using a combination of the two.

> **Not-so-quick exercise** Another chief title that was high on the list was "Owner." How many institutions have an "Owner" as their chief administrator? Of these, how many are private, for-profit institutions (`control == 3`)? How many have the word "Beauty" in their name?

Replace using string position: `str_sub()`

In addition to filtering data, we sometimes need to create new variables from pieces of exiting variables. For example, let's look at the zip code values that are included in the file.

```
## show first few zip code values
df %>%
    select(unitid, zip)
```

```
# A tibble: 7,052 x 2
   unitid zip
    <dbl> <chr>
 1 100636 36112-6613
 2 100654 35762
 3 100663 35294-0110
 4 100690 36117-3553
 5 100706 35899
 6 100724 36101-0271
 7 100733 35401
```

```
 8 100751 35487-0166
 9 100760 35010
10 100812 35611
# … with 7,042 more rows
```

We can see that we have both regular 5 digit zip codes as well as those that include the extra 4 digits (ZIP+4). Let's say we don't need those last four digits for our analysis (particularly because not every school uses them anyway). Our task is to create a new column that pulls out only the main part of the zip code. It is has to work both for `zip` values that include the additional hyphen and 4 digits as well as those that only have the primary 5 digits to begin with.

One solution in this case is to take advantage of the fact that zip codes — minus the sometimes extra 4 digits — should be regular: 5 digits. If want the sub-part of a string and that sub-part is always in the same spot, we can use the function, `str_sub()`, which takes a string or column name first, and has arguments for the starting and ending character that mark the sub-string of interest.

In our case, we want the first 5 digits so we should `start == 1` and `end == 5`:

```
## pull out first 5 digits of zip code
df <- df %>%
    mutate(zip5 = str_sub(zip, start = 1, end = 5))

## show (use select() to subset so we can set new columns)
df %>%
    select(unitid, zip, zip5)
```

```
# A tibble: 7,052 x 3
   unitid zip        zip5
    <dbl> <chr>      <chr>
 1 100636 36112-6613 36112
 2 100654 35762      35762
 3 100663 35294-0110 35294
 4 100690 36117-3553 36117
 5 100706 35899      35899
 6 100724 36101-0271 36101
 7 100733 35401      35401
 8 100751 35487-0166 35487
 9 100760 35010      35010
10 100812 35611      35611
# … with 7,042 more rows
```

A quick visual inspection of the first few rows shows that our `str_sub()` function performed as expected (for a real analysis, you'll want to do more formal checks).

Replace using regular expressions: `str_replace()`

We can also use a more sophisticated regex pattern with the function `str_replace()`. The pieces of our regex pattern, `"([0-9]+)(-[0-9]+)?"`, are translated as this:

- `[0-9]` := any digit, 0 1 2 3 4 5 6 7 8 9
- `+` := match the preceding one or more times
- `?` := match the preceding 0 or more times
- `()` := subexpression

Put together, we have:

- `([0-9]+)` := first, look for 1 or more digits
- `(-[0-9]+)?` := second, look for a hyphen and one or more digits, but you may not find any of that

Because we used parentheses, `()`, to separate our subexpressions, we can call them using their numbers (in order) in the last argument of `str_replace()`:

- `"\\1"` := return the first subexpression

So what's happening? If given a zip code that is `"32605"`, the regex pattern will collect each digit — `"3"` `"2"` `"6"` `"0"` `"5"` — into the first subexpression because it never sees a hyphen. That first subexpression, `"\\1"`, is returned: `"32605"`. That's what we want.

If given `"32605-1234"`, it will collect the first 5 digits in the first subexpression, but will stop adding characters there when it sees the hyphen. From then on out, it adds everything it sees the second subexpression: `"-"` `"1"` `"2"` `"3"` `"4"`. But because `str_replace()` only returns the first subexpression, we still get the same answer: `"32605"`. This is what we want.

Let's try it on the data.

```
## drop last four digits of extended zip code if they exist
df <- df %>%
    mutate(zip5_v2 = str_replace(zip, "([0-9]+)(-[0-9]+)?", "\\1"))

## show (use select() to subset so we can set new columns)
df %>%
    select(unitid, zip, zip5, zip5_v2)
```

```
# A tibble: 7,052 x 4
   unitid zip        zip5  zip5_v2
    <dbl> <chr>      <chr> <chr>
 1 100636 36112-6613 36112 36112
 2 100654 35762      35762 35762
 3 100663 35294-0110 35294 35294
 4 100690 36117-3553 36117 36117
 5 100706 35899      35899 35899
 6 100724 36101-0271 36101 36101
 7 100733 35401      35401 35401
 8 100751 35487-0166 35487 35487
 9 100760 35010      35010 35010
10 100812 35611      35611 35611
# … with 7,042 more rows
```

> **Quick exercise** What if you wanted to the get the last 4 digits (after the hyphen)? What bit of two bits of code above would you change so that you can store the last 4 digits without including the hyphen? Make a new variable called `zip_plus4` and store these values. **HINT** Look at the help file for `str_replace()`.

Let's compare our two versions: do we get the same results?

```
## check if both versions of new zip column are equal
identical(df %>% select(zip5), df %>% select(zip5_v2))
```

```
[1] FALSE
```

No! Let's see where they are different:

```
## filter to rows where zip5 != zip5_v2 (not storing...just looking)
df %>%
    filter(zip5 != zip5_v2) %>%
    select(unitid, zip, zip5, zip5_v2)
```

```
# A tibble: 4 x 4
  unitid zip        zip5  zip5_v2
   <dbl> <chr>      <chr> <chr>
1 108199 90015--350 90015 90015--350
2 113953 92113--191 92113 92113--191
3 431707 06360--709 06360 06360--709
4 435240 551012595  55101 551012595
```

> **Quick exercise** What happened? In this scenario, which string subsetting technique worked better?

Depending on the task, regular expressions can either feel like a blessing or a curse. To be honest, I've spent more time cursing than thanking them. That said, regular expressions are often *the only* way to perform a data wrangling task on unstructured string data. They are also a cornerstone of natural language processing techniques, which are increasingly of interest to education researchers.

We've only scratched the surface of what regular expressions can do. If you face string data in the future, taking a little time to craft a regular expression can be well worth it.

Part 2: Working with dates

In opening section, we've seen that dates often come in many different formats. While you can format and clean them using regular expressions, you may also want to format them such that R knows they are dates.

Why?

When dealing with something straightforward like years, it's easy enough to store the years a regular numbers and then subtract the recent year from the past year to get a duration: `2020 - 2002` equals `18` years.

But what if you have daily data for the school year and you want to know how many days a student had between a first and second test? What if the differences were more than a month of days and every student took the first and second tests on different days? What if you had a panel data set, with students across years, some of which were leap years? You can see how calculating the exact number days between tests for each student could quickly become difficult if trying to do it using regular numerical values.

R makes this easier by having special time-based data types that will keep track of these issues for us and allow us to work with dates almost as we do with regular numbers.

In our IPEDS data set, we can see that few institutions closed in 2007 and 2008. We'll limit our next analyses to these institutions.

```
## subset to schools who closed during this period
df <- df %>%
    filter(closedat != -2)

## show first few rows
df %>% select(unitid, instnm, closedat)
```

```
# A tibble: 83 x 3
   unitid instnm                                          closedat
    <dbl> <chr>                                          <chr>
 1 103440 Sheldon Jackson College                        6/29/07
 2 104522 DeVoe College of Beauty                        3/29/08
 3 105242 Mundus Institute                               Sep-07
 4 105880 Long Technical College-East Valley             3/31/07
 5 119711 New College of California                      Jan-08
```

```
 6 136996 Ross Medical Education Center                              7/31/07
 7 137625 Suncoast II the Tampa Bay School of Massage Therapy LLC 5/31/08
 8 141583 Hawaii Business College                                    Sep-07
 9 150127 Ball Memorial Hospital School of Radiologic Technology  May-07
10 160144 Pat Goins Shreveport Beauty School                         3/1/08
# … with 73 more rows
```

We can see that `closedat` is stored as a string. Based on our domain knowledge and context clues, we know that the dates are generally in a `MM/DD/YYYY` (American) format.

We can use the {lubridate} command `mdy()` to make a new variable that contains the same information, but in a format that R recognizes as a date.

```
## create a new close date column
df <- df %>%
    mutate(closedat_dt = mdy(closedat))
```

```
Warning: 35 failed to parse.
```

```
## show
df %>% select(starts_with("close"))
```

```
# A tibble: 83 x 2
   closedat closedat_dt
   <chr>    <date>
 1 6/29/07  2007-06-29
 2 3/29/08  2008-03-29
 3 Sep-07   NA
 4 3/31/07  2007-03-31
 5 Jan-08   NA
 6 7/31/07  2007-07-31
 7 5/31/08  2008-05-31
 8 Sep-07   NA
 9 May-07   NA
10 3/1/08   2008-03-01
# … with 73 more rows
```

Well, we are part of the way there. It seems that `mdy()` didn't really work with dates like `Sep-2007`. What can we do?

One solution is to add in a fake day for the ones that didn't parse and then convert using `mdy()`. We'll use regular expressions with an `str_replace()`.

```
## convert MON-YYYY to MON-01-YYYY
df <- df %>%
    mutate(closedat_fix = str_replace(closedat, "-", "-01-"),
           closedat_fix_dt = mdy(closedat_fix))
```

```
Warning: 7 failed to parse.
```

```
## show
df %>% select(starts_with("close"))
```

```
# A tibble: 83 x 4
   closedat closedat_dt closedat_fix closedat_fix_dt
   <chr>    <date>      <chr>        <date>
 1 6/29/07  2007-06-29  6/29/07      2007-06-29
 2 3/29/08  2008-03-29  3/29/08      2008-03-29
 3 Sep-07   NA          Sep-01-07    2007-09-01
```

```
 4 3/31/07  2007-03-31  3/31/07      2007-03-31
 5 Jan-08   NA          Jan-01-08    2008-01-01
 6 7/31/07  2007-07-31  7/31/07      2007-07-31
 7 5/31/08  2008-05-31  5/31/08      2008-05-31
 8 Sep-07   NA          Sep-01-07    2007-09-01
 9 May-07   NA          May-01-07    2007-05-01
10 3/1/08   2008-03-01  3/1/08       2008-03-01
# … with 73 more rows
```

> **Quick exercise** We had 7 parsing errors. Can you figure out which rows failed to parse and guess why? **HINT** if `mdy()` failed to parse `closedat`, then the subsequent new columns are likely missing values.

Now that we've successfully converted the string date into a proper date type, it's easy to pull out the pieces of that date, including:

- **year** with `year()`
- **month** with `month()`
- **day** with `day()`
- **day of week** with `wday()`

```
## add columns for
## - year
## - month
## - day
## - day of week (dow)
df <- df %>%
    mutate(close_year = year(closedat_fix_dt),
           close_month = month(closedat_fix_dt),
           close_day = day(closedat_fix_dt),
           close_dow = wday(closedat_fix_dt, label = TRUE))
## show
df %>%
    select(closedat_fix_dt, close_year, close_month, close_day, close_dow)
```

```
# A tibble: 83 x 5
   closedat_fix_dt close_year close_month close_day close_dow
   <date>               <dbl>       <dbl>     <int> <ord>
 1 2007-06-29            2007           6        29 Fri
 2 2008-03-29            2008           3        29 Sat
 3 2007-09-01            2007           9         1 Sat
 4 2007-03-31            2007           3        31 Sat
 5 2008-01-01            2008           1         1 Tue
 6 2007-07-31            2007           7        31 Tue
 7 2008-05-31            2008           5        31 Sat
 8 2007-09-01            2007           9         1 Sat
 9 2007-05-01            2007           5         1 Tue
10 2008-03-01            2008           3         1 Sat
# … with 73 more rows
```

> **Quick exercise** Can we trust our `close_dow` variable? Why?

```r
## how long since the institution closed
## - as of 1 January 2020
## - as of today
df <- df %>%
    mutate(time_since_close_jan = ymd("2020-01-01") - closedat_fix_dt,
           time_since_close_now = today() - closedat_fix_dt)

## show
df %>% select(starts_with("time_since_close"))
```

```
# A tibble: 83 x 2
   time_since_close_jan time_since_close_now
   <drtn>               <drtn>
 1 4569 days            4610 days
 2 4295 days            4336 days
 3 4505 days            4546 days
 4 4659 days            4700 days
 5 4383 days            4424 days
 6 4537 days            4578 days
 7 4232 days            4273 days
 8 4505 days            4546 days
 9 4628 days            4669 days
10 4323 days            4364 days
# … with 73 more rows
```

As with strings and regular expressions, we've only scratched the surface of working with dates in R. For example, you can also work with times (hours, minutes, seconds, *etc*). Now that you've been introduced, however, you should have a starting point for working with panel and administrative data that includes strings and dates that you need to process before conducting your analyses.