16 Filters and selections

More often than not, we have more data than we want. Sometimes we need to be rid of that data. In `dplyr`, there's two ways to go about this: filtering and selecting.

Filtering creates a subset of the data based on criteria. All records where the amount is greater than 150,000. All records that match "College Park". Something like that. **Filtering works with rows – when we filter, we get fewer rows back than we start with.**

Selecting simply returns only the fields named. So if you only want to see city and amount, you select those fields. When you look at your data again, you'll have two columns. If you try to use one of your columns that you had before you used select, you'll get an error. **Selecting works with columns. You will have the same number of records when you are done, but fewer columns of data to work with.**

Let's continue to work with the UMD course data we used in the previous chapter. First, we need to load the tidyverse:

```
library(tidyverse)
```

```
umd_courses <- read_rds("data/umd_courses.rds")
```

If we want to see only those courses offered a particular department, we can use the `filter` function to isolate just those records. Filter works with something called a comparison operator. We need to filter all records equal to "Journalism". The comparison operators in R, like most programming languages, are == for equal to, != for not equal to, > for greater than, >= for greater than or equal to and so on.

Be careful: `=` is not `==` and `=` is not "equal to". `=` is an assignment operator in most languages – how things get named.

```
journalism_courses <- umd_courses |> filter(department == "Jour

head(journalism_courses)
```

```
# A tibble: 6 × 9
  id       title        description   term department sections
instructors seats
```

```
  <chr>    <chr>        <chr>        <dbl> <chr>         <dbl>
<chr>       <dbl>
1 JOUR282  Beyond Face… "Credit on… 202112 Journalism        1
Denitsa Yo…    35
2 JOUR698  Special Pro…  <NA>       202112 Journalism        0
<NA>            0
3 JOUR175  Media Liter… "Additiona… 202112 Journalism        1
Susan Moel…    20
4 JOUR199  Survey Appr…  <NA>       202112 Journalism        1
Karen Denny     5
5 JOUR130  Self-Presen… "Credit on… 202112 Journalism        1
Amber Moore    25
6 JOUR459W Special Top… "This cour… 202112 Journalism        1
Shannon Sc…    23
# ℹ 1 more variable: syllabus_count <dbl>
```

And just like that, we have just Journalism results, which we can verify looking at the head, the first six rows.

We also have more data than we might want. For example, we may only want to work with the course id and title.

To simplify our dataset, we can use select.

```
selected_journalism_courses <- journalism_courses |> select(id,

head(selected_journalism_courses)
```

```
# A tibble: 6 × 2
  id       title
  <chr>    <chr>
1 JOUR282  Beyond Facebook: How Social Media are Transforming
Society, Culture,…
2 JOUR698  Special Problems in Communication
3 JOUR175  Media Literacy
4 JOUR199  Survey Apprenticeship
5 JOUR130  Self-Presentation in the Age of YouTube
6 JOUR459W Special Topics in Journalism; Sports Media & Athlete
Branding
```

And now we only have two columns of data for whatever analysis we might want to do.

16.1 Combining filters

So let's say we wanted to see all the courses in the Theatre department

with at least 15 seats. We can do this a number of ways. The first is we can chain together a whole lot of filters.

```
theatre_seats_15 <- umd_courses |> filter(department == "Theatr

nrow(theatre_seats_15)
```

```
[1] 308
```

That gives us 308 records But that's repetitive, no? We can do better using a single filter and boolean operators – AND and OR. In this case, AND is `&` and OR is `|`.

The difference? With AND, all conditions must be true to be included. With OR, any of those conditions things can be true and it will be included.

Here's the difference.

```
and_theatre_seats_15 <- umd_courses |> filter(department == "The

nrow(and_theatre_seats_15)
```

```
[1] 308
```

So AND gives us the same answer we got before. What does OR give us?

```
and_theatre_seats_15 <- umd_courses |> filter(department == "The

nrow(and_theatre_seats_15)
```

```
[1] 54000
```

So there's 54,000 rows that are EITHER Theatre classes OR have at least 15 seats. OR is additive; AND is restrictive.

A general tip about using filter: it's easier to work your way towards the filter syntax you need rather than try and write it once and trust the result. Each time you modify your filter, check the results to see if they make sense. This adds a little time to your process but you'll thank yourself for doing it because it helps avoid mistakes.

17 Working with dates

One of the most frustrating things in data is working with dates. Everyone has a different opinion on how to record them, and every software package on the planet has to sort it out. Dealing with it can be a little … confusing. And every dataset has something new to throw at you. So consider this an introduction.

First, there's the right way to display dates in data. Most of the rest of the world knows how to do this, but Americans aren't taught it. The correct way to display dates is the following format: YYYY-MM-DD, or 2022-09-15. Any date that looks different should be converted into that format when you're using R.

Luckily, this problem is so common that the Tidyverse has an entire library for dealing with it: lubridate.

We're going to do this two ways. First I'm going to show you how to use base R to solve a tricky problem. And then we'll use a library called `lubridate` to solve a more common and less tricky problem. And then we'll use a new library to solve most of the common problems before they start. If it's not already installed, just run `install.packages('lubridate')`

17.1 Making dates dates again

First, we'll import `tidyverse` like we always do and our newly-installed lubridate.

```
library(tidyverse)
library(lubridate)
```

Let's start with a dataset of campaign expenses from Maryland political committees:

```
maryland_expenses <- read_csv("data/maryland_expenses.csv")
```

```
Rows: 97912 Columns: 14
── Column specification ────────────────────────────────────────
Delimiter: ","
chr (12): expenditure_date, payee_name, address, payee_type,
```

```
committee_name,...
dbl  (1): amount
lgl  (1): expense_toward

ℹ Use `spec()` to retrieve the full column specification for
this data.
ℹ Specify the column types or set `show_col_types = FALSE` to
quiet this message.
```

```
head(maryland_expenses)
```

```
# A tibble: 6 × 14
  expenditure_date payee_name           address payee_type
amount committee_name
  <chr>            <chr>                <chr>   <chr>
<dbl> <chr>
1 3/12/2021        <NA>                 <NA>    Reimburse
350 Salling   Joh…
2 3/29/2021        Dundalk Eagle Newsp… PO Box… Business/…
329 Salling   Joh…
3 4/29/2021        Dundalk Eagle Newsp… PO Box… Business/…
400 Salling   Joh…
4 5/18/2021        Dundalk Eagle Newsp… PO Box… Business/…
350 Salling   Joh…
5 6/9/2021         Dundalk Heritage Fa… Dundal… Business/…
200 Salling   Joh…
6 6/9/2021         Dundalk Heritage Fa… Dundal… Business/…
250 Salling   Joh…
# ℹ 8 more variables: expense_category <chr>, expense_purpose
<chr>,
#   expense_toward <lgl>, expense_method <chr>, vendor <chr>,
fundtype <chr>,
#   comments <chr>, x14 <chr>
```

Take a look at that first column, expenditure_date. It *looks* like a date, but see the `<chr` right below the column name? That means R thinks it's actually a character column. What we need to do is make it into an actual date column, which lubridate is very good at doing. It has a variety of functions that match the format of the data you have. In this case, the current format is `m/d/y`, and the lubridate function is called `mdy` that we can use with mutate:

```
maryland_expenses <- maryland_expenses |> mutate(expenditure_da
```

```
Warning: There was 1 warning in `mutate()`.
ℹ In argument: `expenditure_date = mdy(expenditure_date)`.
```

```
Caused by warning:
!  18 failed to parse.
```

```
head(maryland_expenses)
```

```
# A tibble: 6 × 14
  expenditure_date payee_name           address payee_type
amount committee_name
  <date>           <chr>                <chr>   <chr>
<dbl> <chr>
1 2021-03-12       <NA>                 <NA>    Reimburse
350 Salling   Joh…
2 2021-03-29       Dundalk Eagle Newsp… PO Box… Business/…
329 Salling   Joh…
3 2021-04-29       Dundalk Eagle Newsp… PO Box… Business/…
400 Salling   Joh…
4 2021-05-18       Dundalk Eagle Newsp… PO Box… Business/…
350 Salling   Joh…
5 2021-06-09       Dundalk Heritage Fa… Dundal… Business/…
200 Salling   Joh…
6 2021-06-09       Dundalk Heritage Fa… Dundal… Business/…
250 Salling   Joh…
# ℹ 8 more variables: expense_category <chr>, expense_purpose
<chr>,
#   expense_toward <lgl>, expense_method <chr>, vendor <chr>,
fundtype <chr>,
#   comments <chr>, x14 <chr>
```

Now look at the expenditure_date column: R says it's a date column and it looks like we want it to: YYYY-MM-DD. Accept no substitutes.

Lubridate has functions for basically any type of character date format: mdy, ymd, even datetimes like ymd_hms.

That's less code and less weirdness, so that's good.

But to get clean data, I've installed a library and created a new field so I can now start to work with my dates. That seems like a lot, but don't think your data will always be perfect and you won't have to do these things.

Still, there's got to be a better way. And there is.

Fortunately, `readr` anticipates some date formatting and can automatically handle many of these issues (indeed it uses lubridate under the hood). When you are importing a CSV file, be sure to use `read_csv`, not `read.csv`.

But you're not done with lubridate yet. It has some interesting pieces parts we'll use elsewhere.

For example, in spreadsheets you can extract portions of dates - a month, day or year - with formulas. You can do the same in R with lubridate. Let's say we wanted to add up the total amount spent in each month in our Maryland expenses data.

We could use formatting to create a Month field but that would group all the Aprils ever together. We could create a year and a month together, but that would give us an invalid date object and that would create problems later. Lubridate has something called a floor date that we can use.

So to follow along here, we're going to use mutate to create a month field, group by to lump them together, summarize to count them up and arrange to order them. We're just chaining things together.

```
maryland_expenses |>
  mutate(month = floor_date(expenditure_date, "month")) |>
  group_by(month) |>
  summarise(total_amount = sum(amount)) |>
  arrange(desc(total_amount))
```

```
# A tibble: 25 × 2
   month      total_amount
   <date>            <dbl>
 1 2022-10-01    15827467.
 2 2022-09-01     6603431.
 3 2022-08-01     5892055.
 4 2022-11-01     4715694.
 5 2021-07-01     2242692.
 6 2021-09-01     2212083.
 7 2021-08-01     2086313.
 8 2021-06-01     1827400.
 9 2021-05-01     1341210.
10 2021-01-01      772923.
# ℹ 15 more rows
```

So the month of June 2022 had the most expenditures by far in this data.

18 Mutating data

Often the data you have will prompt questions that it doesn't immediately answer. Election results, for example, have raw vote totals but we often don't use those to make comparisons between candidates unless the numbers are small. We need percentages!

To do that in R, we can use `dplyr` and `mutate` to calculate new metrics in a new field using existing fields of data. That's the essence of `mutate` - using the data you have to answer a new question.

So first we'll import the tidyverse so we can read in our data and begin to work with it.

```
library(tidyverse)
```

Now we'll import a dataset of county-level gubernatorial results from Maryland's 2022 general election that is in the data folder in this chapter's pre-lab directory. We'll use this to explore ways to create new information from existing data.

```
general_22 <- read_csv('data/md_gov_county.csv')
```

```
Rows: 24 Columns: 8
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr (1): county
dbl (7): fips_code, cox, moore, lashar, wallace, harding, write_ins

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Let's add a column called `percent_moore` for the percentage of votes that went to Wes Moore, the Democratic candidate who won the election, in each county. The code to calculate a percentage is pretty simple. Remember, with `summarize`, we used `n()` to count things. With `mutate`, we use very similar syntax to calculate a new value – a new column of data – using other values in our dataset.

To calculate a percentage, we need both the number of votes for Moore but also the total number of votes. We'll use mutate to create both columns. The first will be total votes. The key here is to save the dataframe to itself so that our changes stick.

```
general_22 <- general_22 |>
  mutate(
    total_votes = cox + moore + lashar + wallace + write_ins,
    pct_moore = moore/total_votes
  )
```

But what do you see right away? Do those numbers look like we expect them to? No. They're a decimal expressed as a percentage. So let's fix that by multiplying by 100. Since we're replacing the contents of our new `pct_moore` column, we can just update our previous code and run it again:

```
general_22 <- general_22 |>
  mutate(
    pct_moore = (moore/total_votes)*100
  )
```

Now, does this ordering do anything for us? No. Let's fix that with arrange.

```
general_22 <- general_22 |>
  mutate(
    pct_moore = (moore/total_votes)*100
  ) |>
  arrange(desc(pct_moore))
```

So now we have results ordered by `pct_moore` with the highest percentage first. To see the lowest percentage first, we can reverse that `arrange` function - we don't need to recalculate the column:

```
general_22 |>
  arrange(pct_moore)
```

```
# A tibble: 24 × 10
   fips_code county       cox moore lashar wallace harding
write_ins total_votes
       <dbl> <chr>      <dbl> <dbl>  <dbl>   <dbl>   <dbl>
<dbl>       <dbl>
 1     24023 Garrett     8195  2402    194      52     138
```

```
0       10843
 2     24001 Allegany   13833  6390    279     143     267
0       20645
 3     24015 Cecil      18159  8691    472     155     311
0       27477
 4     24011 Caroline    6727  3276    176      56     140
0       10235
 5     24039 Somerset    3974  2254     67      46      81
0        6341
 6     24043 Washington 26943 15723    614     252     472
0       43532
 7     24013 Carroll    38969 25155   1515     436     561
0       66075
 8     24047 Worcester  13433  8550    273     121     161
0       22377
 9     24037 Saint Mar… 20279 13291    661     253     346
0       34484
10     24035 Queen Ann… 12840  8577    416     120     212
0       21953
# ℹ 14 more rows
# ℹ 1 more variable: pct_moore <dbl>
```

Moore had his weakest performance in Garrett County, at the far western edge of the state.

18.1 Another use of mutate

Mutate is also useful for standardizing data - for example, making different spellings of, say, campaign spending recipients.

Let's load some Maryland state campaign expenditures into a `maryland_expenses` dataframe, and focus in particular on the `payee_name` column.

```
maryland_expenses <- read_csv("data/maryland_expenses.csv")
```

```
Rows: 97912 Columns: 14
── Column specification ────────────────────────────────────────
Delimiter: ","
chr (12): expenditure_date, payee_name, address, payee_type,
committee_name,...
dbl  (1): amount
lgl  (1): expense_toward

ℹ Use `spec()` to retrieve the full column specification for
```

```
this data.
ℹ Specify the column types or set `show_col_types = FALSE` to
quiet this message.
```

```
maryland_expenses
```

```
# A tibble: 97,912 × 14
   expenditure_date payee_name          address payee_type
amount committee_name
   <chr>            <chr>               <chr>   <chr>
<dbl> <chr>
 1 3/12/2021        <NA>                <NA>    Reimburse   350
Salling   Joh…
 2 3/29/2021        Dundalk Eagle News… PO Box… Business/…  329
Salling   Joh…
 3 4/29/2021        Dundalk Eagle News… PO Box… Business/…  400
Salling   Joh…
 4 5/18/2021        Dundalk Eagle News… PO Box… Business/…  350
Salling   Joh…
 5 6/9/2021         Dundalk Heritage F… Dundal… Business/…  200
Salling   Joh…
 6 6/9/2021         Dundalk Heritage F… Dundal… Business/…  250
Salling   Joh…
 7 6/1/2021         Neighborhood Signs  6655 a… Business/…
77.4 Salling   Joh…
 8 4/16/2021        <NA>                <NA>    Reimburse   150
Salling   Joh…
 9 7/1/2021         MSP CUSTOM SOL      1000 P… Business/…
238.  Salling   Joh…
10 7/2/2021         Squire's Restaurant 6723 H… Business/…  260
Salling   Joh…
# ℹ 97,902 more rows
# ℹ 8 more variables: expense_category <chr>, expense_purpose
<chr>,
#   expense_toward <lgl>, expense_method <chr>, vendor <chr>,
fundtype <chr>,
#   comments <chr>, x14 <chr>
```

You'll notice that there's a mix of styles: lower-case and upper-case names like "Anedot" and "ANEDOT", for example. R will think those are two different payees, and that will mean that any aggregates we create based on payee_name won't be accurate.

So how can we fix that? Mutate - it's not just for math! And a function called `str_to_upper` that will convert a character column into all uppercase.

```
standardized_maryland_expenses <- maryland_expenses |>
  mutate(
    payee_upper = str_to_upper(payee_name)
)
```

There are lots of potential uses for standardization - addresses, zip codes, anything that can be misspelled or abbreviated.

18.2 A more powerful use

Mutate is even more useful when combined with some additional functions. Let's keep rolling with our expenditure data. Take a look at the address column: it contains a full address, including the state, spelled out. It would be useful to have a separate `state` column with an abbreviation. We can check to see if a state name is contained in that column and then populate a new column with the value we want, using the functions `str_detect` and `case_when`. We can identify the state by the following pattern: a space, followed by the full name, followed by another space. So, " Maryland ". The `case_when` function handles multiple variations, such as if the state is Maryland or the state is Texas, etc. Crucially, we can tell R to populate the new column with `NA` if it doesn't find a match.

```
maryland_expenses_with_state <- maryland_expenses |>
  mutate(
    state = case_when(
        str_detect(address, " Maryland ") ~ "MD",
        str_detect(address, " California ") ~ "CA",
        str_detect(address, " Washington ") ~ "WA",
        str_detect(address, " Louisiana ") ~ "LA",
        str_detect(address, " Florida ") ~ "FL",
        str_detect(address, " North Carolina ") ~ "NC",
        str_detect(address, " Massachusetts ") ~ "MA",
        str_detect(address, " West Virginia ") ~ "WV",
        str_detect(address, " Virginia ") ~ "VA",
        .default = NA
      )
  )
```

There's a lot going on here, so let's unpack it. It starts out as a typical mutate statement, but `case_when` introduces some new things. Each line checks to see if the pattern is contained in the address column, followed by `~` and then a value for the new column for records that match that check. You can read it like this: "If we find ' Maryland ' in the address

column, then put 'MD' in the state column" for Maryland and then a handful of states, and if we don't match any state we're looking for, make state `NA`.

We can then use our new `state` column in group_by statements to make summarizing easier.

```
maryland_expenses_with_state |>
  group_by(state) |>
  summarize(total = sum(amount)) |>
  arrange(desc(total))
```

```
# A tibble: 10 × 2
   state     total
   <chr>     <dbl>
 1 MD    77723146.
 2 WA    15552127.
 3 VA    10519646.
 4 CA     3370284.
 5 FL     1470592.
 6 MA     1264728.
 7 NC      691006.
 8 LA      255522.
 9 WV       41088.
10 <NA>        NA
```

Most expenditures seem to have occurred in Maryland, which makes sense, although we haven't assigned a state for every transaction.

Mutate is there to make your data more useful and to make it easier for you to ask more and better questions of it.