Data Preparation

Rolf Bänziger, r.banziger@westminster.ac.uk

24/10/2022

In the last few weeks, we worked with data already available in R. These data sets were already suitable formats to visualise. Unfortunately, data in the real world is rarely in a form ready for analysing, and we often need to prepare data before we can use it.

In this tutorial, we learn how to load data, reshape, tidy and transform it in preparation for data analysis.

All libraries we need are part of the *tidyverse* (as is `ggplot2`). As usual, we need to install it first (note that this is a very large package that might take some time to install)…

In the last few weeks, we worked with data already available in R. These data sets were already suitable fo

```
install.packages("tidyverse")
```

…and load it (this loads all required packages, including `ggplot2`, so we don't need to load packages individually):

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Loading data

Data is often supplied in so-called CSV (comma separated values) files. CSV files are simple text files where each record is written on one line, and a comma separates each value.

We can use the function `read_csv()` from the `readr` package to read CSV files. This example loads the *chickens.csv* file (which must be in your current working directory, use the *Files* window in the bottom right corner of R Studio).

```
chickens <- read_csv("chickens.csv")
```

```
## Rows: 5 Columns: 4
## -- Column specification --------------------------------------------------------
```

```
## Delimiter: ","
## chr (3): chicken, sex, motto
## dbl (1): eggs_laid
##
## 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.
```

```
chickens
```

```
## # A tibble: 5 x 4
##   chicken                 sex     eggs_laid motto
##   <chr>                   <chr>       <dbl> <chr>
## 1 Foghorn Leghorn         rooster         0 That's a joke, ah say, that's a jok~
## 2 Chicken Little          hen             3 The sky is falling!
## 3 Ginger                  hen            12 Listen. We'll either die free chick~
## 4 Camilla the Chicken     hen             7 Bawk, buck, ba-gawk.
## 5 Ernie The Giant Chicken rooster         0 Put Captain Solo in the cargo hold.
```

In this example, we let `read_csv` guess the column types. You can use `spec()` to show the (guessed) column specification:

```
spec(chickens)
```

```
## cols(
##   chicken = col_character(),
##   sex = col_character(),
##   eggs_laid = col_double(),
##   motto = col_character()
## )
```

Now you can copy, paste, and tweak this, to create a more explicit readr call that expresses the desired column types. Here we express that sex should be a factor with levels rooster and hen, in that order, and that eggs_laid should be integer.

```
chickens <- read_csv(
  readr_example("chickens.csv"),
  col_types = cols(
    chicken   = col_character(),
    sex       = col_factor(levels = c("rooster", "hen")),
    eggs_laid = col_integer(),
    motto     = col_character()
  )
)
chickens
```

```
## # A tibble: 5 x 4
##   chicken                 sex     eggs_laid motto
##   <chr>                   <fct>       <int> <chr>
## 1 Foghorn Leghorn         rooster         0 That's a joke, ah say, that's a jok~
## 2 Chicken Little          hen             3 The sky is falling!
## 3 Ginger                  hen            12 Listen. We'll either die free chick~
## 4 Camilla the Chicken     hen             7 Bawk, buck, ba-gawk.
## 5 Ernie The Giant Chicken rooster         0 Put Captain Solo in the cargo hold.
```

Consult readr.tidyverse.org for a more comprehensive introduction to `read_csv` and its cousins `read_tsv()`, `read_delim()`, `read_fwf()`, `read_table()`and `read_log()`.

Loading data from Excel files

Another common file format to exchange data is, of course, Excel. We can use the `readxl` package to import Excel files in R. Even though it is part of the `tidyverse`, we need to explicitly load it:

```
library(readxl)
```

The `read_excel()` function imports an excel sheet:

```
people <- read_excel("people.xlsx")
people
```

```
## # A tibble: 10 x 6
##    Name      Profession   Age `Has kids` `Date of birth`     `Date of death`
##    <chr>     <chr>      <dbl> <lgl>      <dttm>              <dttm>
##  1 David Bo~ musician      69 TRUE       1947-01-08 00:00:00 2016-01-10 00:00:00
##  2 Carrie F~ actor         60 TRUE       1956-10-21 00:00:00 2016-12-27 00:00:00
##  3 Chuck Be~ musician      90 TRUE       1926-10-18 00:00:00 2017-03-18 00:00:00
##  4 Bill Pax~ actor         61 TRUE       1955-05-17 00:00:00 2017-02-25 00:00:00
##  5 Prince    musician      57 TRUE       1958-06-07 00:00:00 2016-04-21 00:00:00
##  6 Alan Ric~ actor         69 FALSE      1946-02-21 00:00:00 2016-01-14 00:00:00
##  7 Florence~ actor         82 TRUE       1934-02-14 00:00:00 2016-11-24 00:00:00
##  8 Harper L~ author        89 FALSE      1926-04-28 00:00:00 2016-02-19 00:00:00
##  9 Zsa Zsa ~ actor         99 TRUE       1917-02-06 00:00:00 2016-12-18 00:00:00
## 10 George M~ musician      53 FALSE      1963-06-25 00:00:00 2016-12-25 00:00:00
```

However, Excel files often have multiple sheets, and data doesn't always start in the first row. We can use the `sheet` and `range` parameters of `read_excel()` to specify exactly what we want to import:

```
people <- read_excel("people.xlsx", sheet= 2, range = "A5:F15")
people
```

```
## # A tibble: 10 x 6
##    Name      Profession   Age `Has kids` `Date of birth`     `Date of death`
##    <chr>     <chr>      <dbl> <lgl>      <dttm>              <dttm>
##  1 David Bo~ musician      69 TRUE       1947-01-08 00:00:00 2016-01-10 00:00:00
##  2 Carrie F~ actor         60 TRUE       1956-10-21 00:00:00 2016-12-27 00:00:00
##  3 Chuck Be~ musician      90 TRUE       1926-10-18 00:00:00 2017-03-18 00:00:00
##  4 Bill Pax~ actor         61 TRUE       1955-05-17 00:00:00 2017-02-25 00:00:00
##  5 Prince    musician      57 TRUE       1958-06-07 00:00:00 2016-04-21 00:00:00
##  6 Alan Ric~ actor         69 FALSE      1946-02-21 00:00:00 2016-01-14 00:00:00
##  7 Florence~ actor         82 TRUE       1934-02-14 00:00:00 2016-11-24 00:00:00
##  8 Harper L~ author        89 FALSE      1926-04-28 00:00:00 2016-02-19 00:00:00
##  9 Zsa Zsa ~ actor         99 TRUE       1917-02-06 00:00:00 2016-12-18 00:00:00
## 10 George M~ musician      53 FALSE      1963-06-25 00:00:00 2016-12-25 00:00:00
```

See readxl.tidyverse.org for a comprehensive documentation. Consult the Data Import Cheat Sheet for an overview of the data import functionality of tidyverse.

Exercise

Import the data in the file `incidents.xlsx`. The imported data should look like below:

```
## # A tibble: 26 x 13
##    Employee January February March April   May  June  July August September
##    <chr>      <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>     <dbl>
##  1 B-002          4        1     5     2     3     0     3      1         2
##  2 E-055          1        2     1     3     4     1     4      0         2
##  3 E-075-II      14       17    16    15    18    16    14     17        12
##  4 B-066          4        4     5     2     5     0     0      2         0
##  5 C-025-II      17       13    17    18    17    17    12     15        17
##  6 E-030          2        2     1     1     0     3     5      5         0
##  7 C-001-II      14       14    14    14    13    18    17     14        13
##  8 E-038          4        1     0     4     0     2     5      0         2
##  9 C-054          2        5     4     4     2     3     0      5         5
## 10 A-081          3        2     4     5     2     2     2      4         1
## # ... with 16 more rows, and 3 more variables: October <dbl>, November <dbl>,
## #   December <dbl>
```

Tidy data

> "Happy families are all alike; every unhappy family is unhappy in its own way."
> — Leo Tolstoy "Tidy datasets are all alike, but every messy dataset is messy in its own way."
> — Hadley Wickham

You can represent the same underlying data in multiple ways. The example below shows the same data organised in four different ways. Each dataset shows the same values of four variables country, year, population, and cases, but each dataset organises the values in a different way.

```
table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset, will be much easier to work with inside the tidyverse.

There are three interrelated rules which make a dataset tidy:

Each variable must have its own column. Each observation must have its own row. Each value must have its own cell.

In the example above, only one dataset is tidy. Which one is it?

Most functions of the `tidyverse` are designed to work with tidy data.

See R for Data Science, Chapter 12: Tidy Data and the Data Tidying Cheat Sheet for more information.

Pivoting

A common problem of datasets is that either variables are spread across multiple columns, or an observation is spread across multiple rows. To fix these problems, you'll need the two most important functions in tidyr: `pivot_longer()` and `pivot_wider()`.

Longer

Consider the example below, where some of the column names are not names of variables, but values of a variable. The column name `1999` and `2000` represent values of the `year` variable, the values in the columns represent values of the `cases` variable, and each row represents two observations, not one.

```
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

To tidy a dataset like this, we need to pivot the offending columns into a new pair of variables. To describe that operation we need three parameters:

- The set of columns whose names are values, not variables. In this example, those are the columns 1999 and 2000.
- The name of the variable to move the column names to. Here it is year.
- The name of the variable to move the column values to. Here it's cases.

Together those parameters generate the call to pivot_longer():

```
table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
```

```
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

Wider

`pivot_wider()` is the opposite of `pivot_longer()`. You use it when an observation is scattered across multiple rows. For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

```
table2 %>% head()
```

```
## # A tibble: 6 x 4
##   country      year type           count
##   <chr>       <int> <chr>          <int>
## 1 Afghanistan  1999 cases            745
## 2 Afghanistan  1999 population  19987071
## 3 Afghanistan  2000 cases           2666
## 4 Afghanistan  2000 population  20595360
## 5 Brazil       1999 cases          37737
## 6 Brazil       1999 population 172006362
```

To tidy this up, we first analyse the representation in similar way to `pivot_longer()`. This time, however, we only need two parameters:

- The column to take variable names from. Here, it's type.
- The column to take values from. Here it's count.

Once we've figured that out, we can use pivot_wider():

```
table2 %>%
    pivot_wider(names_from = type, values_from = count)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Separating and uniting

So far you've learned how to tidy table2 and table4, but not table3. table3 has a different problem: we have one column (rate) that contains two variables (cases and population). To fix this problem, we'll need the separate() function. You'll also learn about the complement of separate(): unite(), which you use if a single variable is spread across multiple columns.

`separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take `table3`:

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

The rate column contains both cases and population variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into:

```
table3 %>%
  separate(rate, into = c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country      year cases  population
##   <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Consult the help file for `separate()` to find out how to specify separator characters, or separate a specific number of characters.

`unite()` is the inverse of `separate()`: it combines multiple columns into a single column. You'll need it much less frequently than `separate()`, but it's still a useful tool to have in your back pocket.

We can use `unite()` to rejoin the *century* and *year* columns in `table5`.

```
table5
```

```
## # A tibble: 6 x 4
##   country     century year  rate
## * <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

`unite()` takes a data frame, the name of the new variable to create, and a set of columns to combine:

```
table5 %>%
  unite(new, century, year)
```

```
## # A tibble: 6 x 3
##   country     new   rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 19_99 745/19987071
## 2 Afghanistan 20_00 2666/20595360
## 3 Brazil      19_99 37737/172006362
## 4 Brazil      20_00 80488/174504898
## 5 China       19_99 212258/1272915272
## 6 China       20_00 213766/1280428583
```

Exercise

Use the `incidents` data you imported earlier. Tidy the dataset. Which steps do you need to undertake?

HINT: The dataset should look like this in this end:

```
## Warning: Expected 3 pieces. Missing pieces filled with `NA` in 216 rows [1, 2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

```
## # A tibble: 6 x 5
##   Location `Employee Id` Tier  Month    Incidents
##   <chr>    <chr>         <chr> <chr>        <dbl>
## 1 B        002           <NA>  January          4
## 2 B        002           <NA>  February         1
## 3 B        002           <NA>  March            5
## 4 B        002           <NA>  April            2
## 5 B        002           <NA>  May              3
## 6 B        002           <NA>  June             0
```

Data transformation

In this section, we're using `dplyr` to transform data. `dplyr` provides five key functions to manipulate data: * Pick observations by their values (`filter()`). * Reorder the rows (`arrange()`). * Pick variables by their names (`select()`). * Create new variables with functions of existing variables (`mutate()`). * Collapse many values down to a single summary (`summarise()`).

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

We will be using the `flights` dataset from the `nycflights13` package.

```
install.packages("nycflights13")
```

```
library(nycflights13)
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

filter

`filter()` allows you to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame. For example, we can select all flights on January 1st with:

```
filter(flights, month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal). Don't confuse `=` with `==`!

Multiple arguments to `filter()` are combined with "and": every expression must be true in order for a row to be included in the output. For other types of combinations, you'll need to use Boolean operators yourself: `&` is "and", `|` is "or", and `!` is "not".

The following code finds all flights that departed in November or December:

```
filter(flights, month == 11 | month == 12)
```

```
## # A tibble: 55,403 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013    11     1        5           2359         6      352            345
##  2  2013    11     1       35           2250       105      123           2356
##  3  2013    11     1      455            500        -5      641            651
##  4  2013    11     1      539            545        -6      856            827
##  5  2013    11     1      542            545        -3      831            855
##  6  2013    11     1      549            600       -11      912            923
##  7  2013    11     1      550            600       -10      705            659
##  8  2013    11     1      554            600        -6      659            701
##  9  2013    11     1      554            600        -6      826            827
## 10  2013    11     1      554            600        -6      749            751
## # ... with 55,393 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

We could also rewrite the code above using the `%in%` operator:

```
filter(flights, month %in% c(11, 12))
```

```
## # A tibble: 55,403 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
```

```
##  1  2013    11     1        5           2359         6      352            345
##  2  2013    11     1       35           2250       105      123           2356
##  3  2013    11     1      455            500        -5      641            651
##  4  2013    11     1      539            545        -6      856            827
##  5  2013    11     1      542            545        -3      831            855
##  6  2013    11     1      549            600       -11      912            923
##  7  2013    11     1      550            600       -10      705            659
##  8  2013    11     1      554            600        -6      659            701
##  9  2013    11     1      554            600        -6      826            827
## 10  2013    11     1      554            600        -6      749            751
## # ... with 55,393 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Arrange

`arrange()` changes the order of rows, by sorting it:

```
arrange(flights, year, month, day)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Use desc() to re-order by a column in descending order:

```
arrange(flights, desc(dep_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     9      641            900      1301     1242           1530
##  2  2013     6    15     1432           1935      1137     1607           2120
##  3  2013     1    10     1121           1635      1126     1239           1810
##  4  2013     9    20     1139           1845      1014     1457           2210
##  5  2013     7    22      845           1600      1005     1044           1815
##  6  2013     4    10     1100           1900       960     1342           2211
##  7  2013     3    17     2321            810       911      135           1020
##  8  2013     6    27      959           1900       899     1236           2226
```

```
##  9  2013     7    22     2257            759       898      121           1026
## 10  2013    12     5      756           1700       896     1058           2020
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Select columns

It's not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

`select()` is not terribly useful with the flights data because we only have 19 variables, but you can still get the general idea:

```
select(flights, year, month, day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

There are a number of helper functions you can use within select():

- `starts_with("abc")`: matches names that begin with "abc".
- `ends_with("xyz")`: matches names that end with "xyz".
- `contains("ijk")`: matches names that contain "ijk".
- `matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters. You'll learn more about regular expressions in strings.
- `num_range("x", 1:3)`: matches x1, x2 and x3.

See `?select` for more details.

`select()` can be used to rename variables, but it's rarely useful because it drops all of the variables not explicitly mentioned. Instead, use `rename()`, which is a variant of `select()` that keeps all the variables that aren't explicitly mentioned:

```
rename(flights, tail_num = tailnum)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
```

```
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tail_num <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Add new variables with `mutate()`

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`.

`mutate()` always adds new columns at the end of your dataset so we'll start by creating a narrower dataset so we can see the new variables.

```
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
mutate(flights_sml,
  gain = dep_delay - arr_delay,
  speed = distance / air_time * 60
)
```

```
## # A tibble: 336,776 x 9
##     year month   day dep_delay arr_delay distance air_time  gain speed
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
##  1  2013     1     1         2        11     1400      227    -9  370.
##  2  2013     1     1         4        20     1416      227   -16  374.
##  3  2013     1     1         2        33     1089      160   -31  408.
##  4  2013     1     1        -1       -18     1576      183    17  517.
##  5  2013     1     1        -6       -25      762      116    19  394.
##  6  2013     1     1        -4        12      719      150   -16  288.
##  7  2013     1     1        -5        19     1065      158   -24  404.
##  8  2013     1     1        -3       -14      229       53    11  259.
##  9  2013     1     1        -3        -8      944      140     5  405.
## 10  2013     1     1        -2         8      733      138   -10  319.
## # ... with 336,766 more rows
```

If you only want to keep the new variables, use transmute():

```
transmute(flights,
  gain = dep_delay - arr_delay,
  hours = air_time / 60,
```

```
  gain_per_hour = gain / hours
)
```

```
## # A tibble: 336,776 x 3
##     gain hours gain_per_hour
##    <dbl> <dbl>         <dbl>
##  1    -9 3.78          -2.38
##  2   -16 3.78          -4.23
##  3   -31 2.67         -11.6
##  4    17 3.05           5.57
##  5    19 1.93           9.83
##  6   -16 2.5           -6.4
##  7   -24 2.63          -9.11
##  8    11 0.883         12.5
##  9     5 2.33           2.14
## 10   -10 2.3           -4.35
## # ... with 336,766 more rows
```

Grouped summaries with `summarise()`

The last key verb is `summarise()`. It collapses a data frame to a single row:

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1  12.6
```

`summarise()` is not terribly useful unless we pair it with `group_by()`. This changes the unit of analysis from the complete dataset to individual groups. Then, when you use the dplyr verbs on a grouped data frame they'll be automatically applied "by group". For example, if we applied exactly the same code to a data frame grouped by date, we get the average delay per date:

```
flights %>%
  group_by(year, month, day) %>%
  summarise(delay = mean(dep_delay, na.rm = TRUE))
```

```
## `summarise()` has grouped output by 'year', 'month'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day delay
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
```

```
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
## 10  2013     1    10  2.84
## # ... with 355 more rows
```

Imagine that we want to explore the relationship between the distance and average delay for each location.

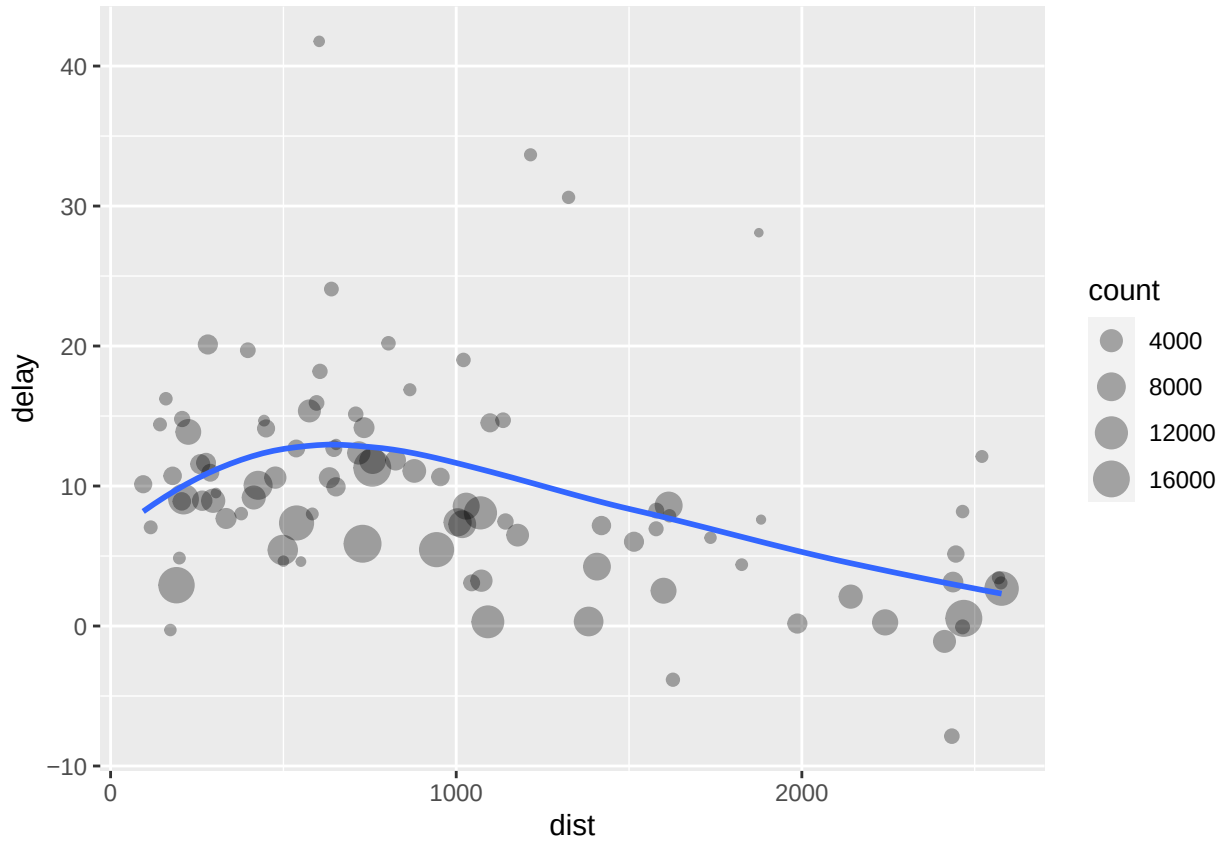
There are three steps to prepare this data:

1. Group flights by destination.
2. Summarise to compute distance, average delay, and number of flights.
3. Filter to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport.

```
delays <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")

ggplot(data = delays, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



See Chapter 5: Data transformation of R for Data Science and the [Data Transformation Cheat] Sheet(https://github.com/rstudio/cheatsheets/blob/main/data-transformation.pdf) for more information.

Exercise

1. Use the modfied `incidents` dataset from the earlier exercise. Replace the `Tier` column with a new column that contains "I" and "2"II". Use `mutate()` and `replace_na()`.

```
## # A tibble: 312 x 5
##    Location `Employee Id` Tier  Month     Incidents
##    <chr>    <chr>         <chr> <chr>         <dbl>
##  1 B        002           I     January           4
##  2 B        002           I     February          1
##  3 B        002           I     March             5
##  4 B        002           I     April             2
##  5 B        002           I     May               3
##  6 B        002           I     June              0
##  7 B        002           I     July              3
##  8 B        002           I     August            1
##  9 B        002           I     September         2
## 10 B        002           I     October           0
## # ... with 302 more rows
```

2. What is the number of resolved incidents by tier 2 employees in location E by Month? Order the list by the highest number of incidents first.

```
## # A tibble: 12 x 2
##    Month     Total
##    <chr>     <dbl>
##  1 May          52
##  2 April        49
##  3 August       47
##  4 February     47
##  5 December     45
##  6 October      45
##  7 June         43
##  8 March        43
##  9 November     42
## 10 July         41
## 11 January      40
## 12 September    38
```

Solutions

Import

```
incidents <- read_excel('incidents.xlsx', sheet = "Resolved Incidents", range = "B4:N30")
incidents
```

```
## # A tibble: 26 x 13
##    Employee January February March April   May  June  July August September
##    <chr>      <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>     <dbl>
##  1 B-002          4        1     5     2     3     0     3      1         2
##  2 E-055          1        2     1     3     4     1     4      0         2
##  3 E-075-II      14       17    16    15    18    16    14     17        12
##  4 B-066          4        4     5     2     5     0     0      2         0
##  5 C-025-II      17       13    17    18    17    17    12     15        17
##  6 E-030          2        2     1     1     0     3     5      5         0
##  7 C-001-II      14       14    14    14    13    18    17     14        13
##  8 E-038          4        1     0     4     0     2     5      0         2
##  9 C-054          2        5     4     4     2     3     0      5         5
## 10 A-081          3        2     4     5     2     2     2      4         1
## # ... with 16 more rows, and 3 more variables: October <dbl>, November <dbl>,
## #   December <dbl>
```

Tidy

```
tidy_incidents <- incidents %>%
  pivot_longer(!Employee, names_to = "Month", values_to = "Incidents") %>%
  separate(Employee, into = c("Location", "Employee Id", "Tier"))
```

```
## Warning: Expected 3 pieces. Missing pieces filled with `NA` in 216 rows [1, 2,
## 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...].
```

```
head(tidy_incidents)
```

```
## # A tibble: 6 x 5
##   Location `Employee Id` Tier  Month    Incidents
##   <chr>    <chr>         <chr> <chr>        <dbl>
## 1 B        002           <NA>  January          4
## 2 B        002           <NA>  February         1
## 3 B        002           <NA>  March            5
## 4 B        002           <NA>  April            2
## 5 B        002           <NA>  May              3
## 6 B        002           <NA>  June             0
```

Transform

```
transformed_incidents <- tidy_incidents %>%
  mutate(Tier = replace_na(Tier, "I"))

transformed_incidents
```

```
## # A tibble: 312 x 5
##    Location `Employee Id` Tier  Month     Incidents
##    <chr>    <chr>         <chr> <chr>         <dbl>
##  1 B        002           I     January           4
##  2 B        002           I     February          1
##  3 B        002           I     March             5
##  4 B        002           I     April             2
##  5 B        002           I     May               3
##  6 B        002           I     June              0
##  7 B        002           I     July              3
##  8 B        002           I     August            1
##  9 B        002           I     September         2
## 10 B        002           I     October           0
## # ... with 302 more rows
```

```
transformed_incidents %>%
  filter(Tier == "II", Location == "E") %>%
  group_by(Month) %>%
  summarise(Total = sum(Incidents)) %>%
  arrange(desc(Total))
```

```
## # A tibble: 12 x 2
##    Month     Total
##    <chr>     <dbl>
##  1 May          52
##  2 April        49
##  3 August       47
##  4 February     47
##  5 December     45
##  6 October      45
##  7 June         43
##  8 March        43
##  9 November     42
## 10 July         41
## 11 January      40
## 12 September    38
```