Data Wrangling

Matthew McDonald

readr Functions

Most of readr's functions are concerned with turning flat files into data frames:

- `read_csv()` reads comma delimited files, `read_csv2()` reads semicolon separated files (common in countries where `,` is used as the decimal place), `read_tsv()` reads tab delimited files, and `read_delim()` reads in files with any delimiter.
- `read_fwf()` reads fixed width files. You can specify fields either by their widths with `fwf_widths()` or their position with `fwf_positions()`. `read_table()` reads a common variation of fixed width files where columns are separated by white space.
- `read_log()` reads Apache style log files. (But also check out webreadr which is built on top of `read_log()` and provides many more helpful tools.)

Other types of data

- **readxl** reads Excel files (both `.xls` and `.xlsx`).
- **googlesheets4** reads Google Sheets.
- **DBI**, along with a database specific backend (e.g. **RMySQL**, **RSQLite**, **RPostgreSQL** etc) allows you to run SQL queries against a database and return a data frame.
- **haven** reads SPSS, Stata, and SAS files.
- For hierarchical data: use **jsonlite** (by Jeroen Ooms) for json, and **xml2** for XML.

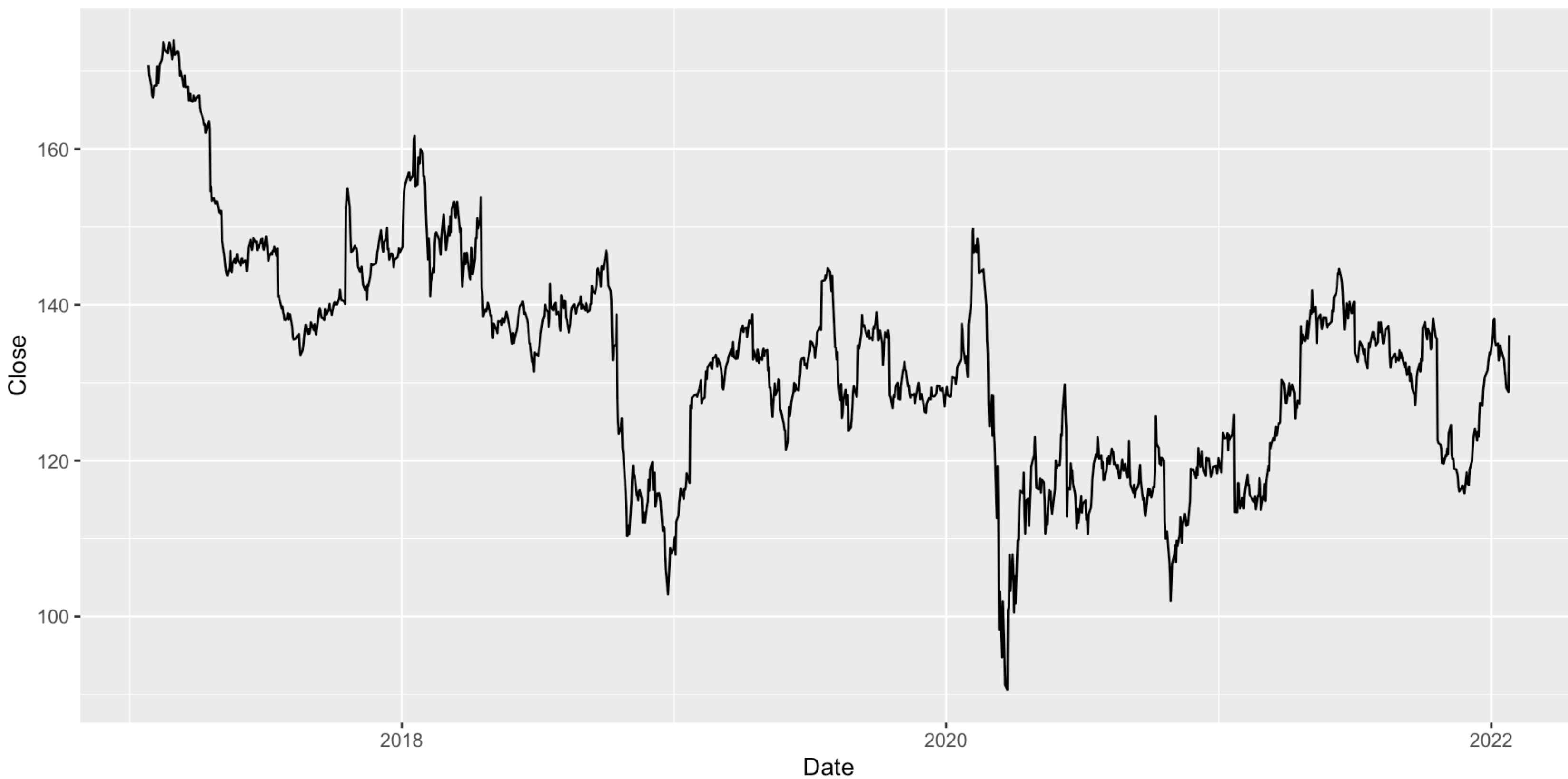
Fixing The IBM Data

```r
ibm <- read_csv('IBM.csv')
ibm <- ibm %>% mutate(Date=as.Date(Date, format="%m/%d/%Y"))

# spec(ibm)
ibm <- read_csv('IBM.csv',
                col_types = list(Date = col_date(format="%m/%d/%Y"),
                                  Open = col_double(),
                                  High = col_double(),
                                  Low = col_double(),
                                  Close = col_double(),
                                  `Adj Close` = col_double(),
                                  Volume = col_double()))

ibm %>% ggplot(aes(x=Date, y=Close)) + geom_line()
```

Fixing The IBM Data



Tidy Data

There are three interrelated rules which make a dataset tidy:

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

Examples of Different Data Sets

```
table1
```

```
# A tibble: 6 × 4
  country      year  cases population
  <chr>       <dbl>  <dbl>      <dbl>
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 × 4
   country      year type            count
   <chr>       <dbl> <chr>           <dbl>
 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
```

Examples of Different Data Sets (2)

```
1  table3
```

```
# A tibble: 6 × 3
  country      year rate
  <chr>       <dbl> <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
```

```
1  table4a
```

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

```
1  table4b
```

```
# A tibble: 3 × 3
  country         `1999`     `2000`
  <chr>            <dbl>      <dbl>
1 Afghanistan   19987071   20595360
2 Brazil       172006362  174504898
3 China       1272915272 1280428583
```

Pivoting

Common problems:

1. One variable might be spread across multiple columns.
2. One observation might be scattered across multiple rows.

table4a

In `table4a` the column names `1999` and `2000` represent values of the `year` variable, the values in the `1999` and `2000` columns represent values of the `cases` variable, and each row represents two observations, not one.

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: `year`.
- The name of the variable to move the column values to: `cases`.

Together those parameters generate the call to `pivot_longer()`:

pivot_longer()

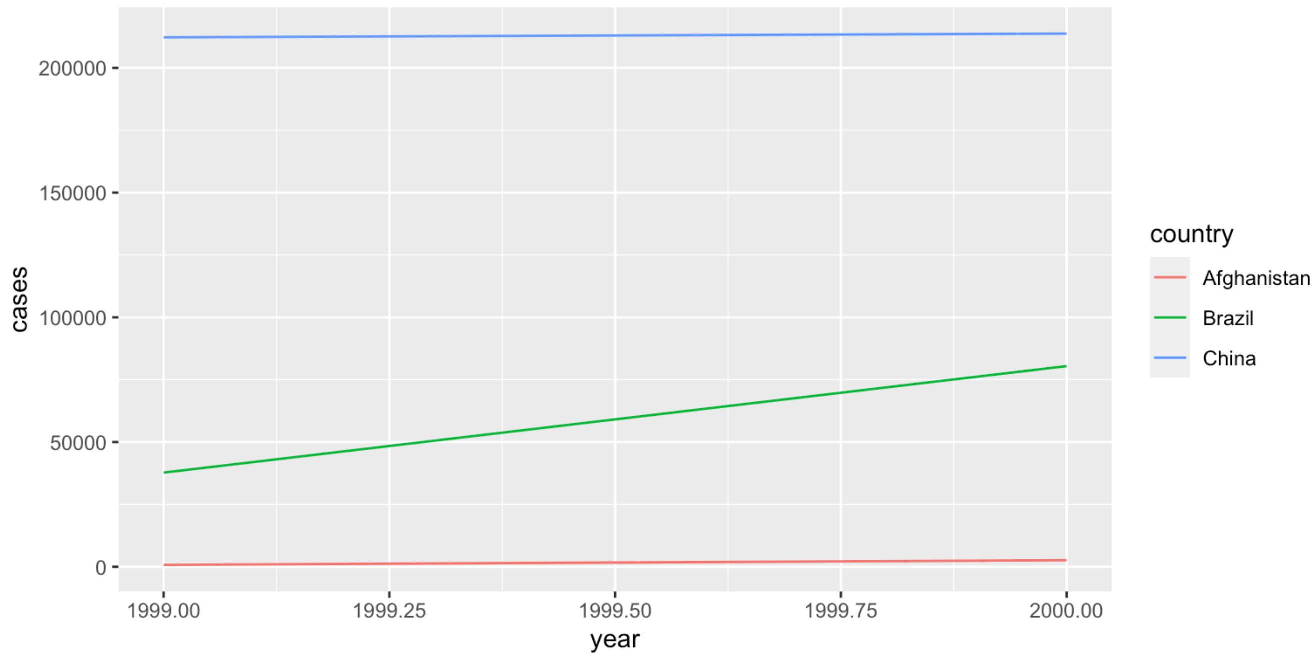
Together those parameters generate the call to `pivot_longer()`:

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

```
# A tibble: 6 × 3
  country     year   cases
  <chr>       <chr>  <dbl>
1 Afghanistan 1999     745
2 Afghanistan 2000    2666
3 Brazil      1999   37737
4 Brazil      2000   80488
5 China       1999  212258
6 China       2000  213766
```

table4 ggplot

```r
table4a %>% pivot_longer(cols = c(`1999`, `2000`),
                         names_to = "year",
                         values_to = "cases") %>%
  mutate(year = as.integer(year)) %>%
  ggplot(aes(x = year, y = cases)) +
  geom_line(aes(color = country))
```



fixing table2

```
table2
```

```
# A tibble: 12 × 4
   country      year type            count
   <chr>       <dbl> <chr>           <dbl>
 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
```

pivot_wider()

We need a data frame with `cases` and `population` as separate columns, and in those columns, each cell will hold the values of the relevant `count`s. Let's analyse the representation in similar way to `pivot_longer()`. This time, however, we only need two parameters:

- The column to take variable names from: `type`.
- The column to take values from: `count`.

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

```
# A tibble: 6 × 4
  country      year  cases population
  <chr>       <dbl>  <dbl>      <dbl>
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
```

Relational Data

It’s rare that a data analysis involves only a single data frame. Typically you have many data frames, and you must combine them to answer the questions that you’re interested in. Collectively, multiple data frames are called **relational data** because it is the relations, not just the individual datasets, that are important.

Relations are always defined between a pair of data frames. All other relations are built up from this simple idea: the relations of three or more data frames are always a property of the relations between each pair. Sometimes both elements of a pair can be the same data frame! This is needed if, for example, you have a data frame of people, and each person has a reference to their parents.

Relational Data Verbs

To work with relational data you need verbs that work with pairs of data frames. There are three families of verbs designed to work with relational data:

- **Mutating joins**, which add new variables to one data frame from matching observations in another.
- **Filtering joins**, which filter observations from one data frame based on whether or not they match an observation in the other data frame.
- **Set operations**, which treat observations as if they were set elements.

RDBMS

The most common place to find relational data is in a *relational* database management system (or RDBMS), a term that encompasses almost all modern databases.

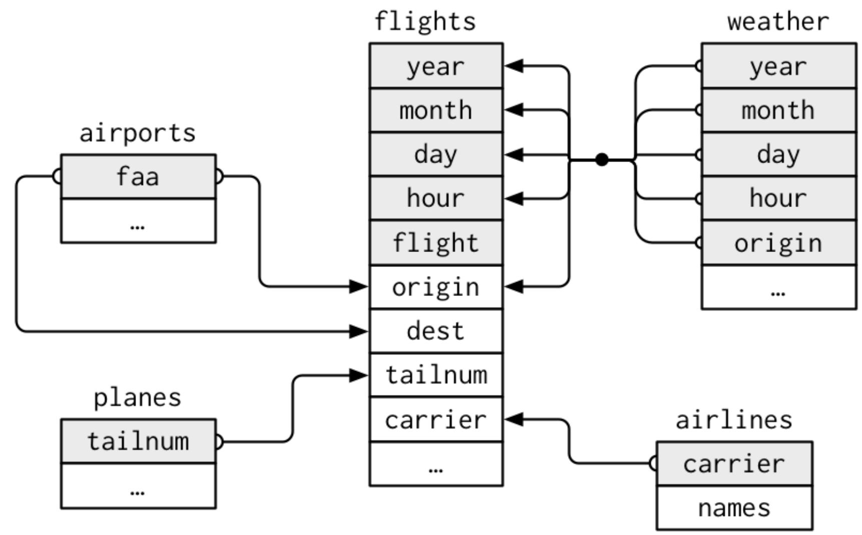
If you've used a database before, you've almost certainly used SQL.

One other major terminology difference between databases and R is that what we generally refer to as data frames in R while the same concept is referred to as "table" in databases.

nycflights13

- `flights` 2013 NYC flight data
- `airlines` lets you look up the full carrier name from its abbreviated code:
- `airports` gives information about each airport, identified by the `faa` airport code
- `planes` gives information about each plane, identified by its `tailnum`
- `weather` gives the weather at each NYC airport for each hour

One way to show the relationships between the different data frames is with a diagram:



Keys

The variables used to connect each pair of data frames are called **keys**.

A key is a variable (or set of variables) that uniquely identifies an observation.

In simple cases, a single variable is sufficient to identify an observation.

For example, each plane is uniquely identified by its `tailnum`.

In other cases, multiple variables may be needed.

For example, to identify an observation in `weather` you need five variables: `year`, `month`, `day`, `hour`, and `origin`.

Types of Keys

There are two types of keys:

- A **primary key** uniquely identifies an observation in its own data frame. For example, `planes$tailnum` is a primary key because it uniquely identifies each plane in the `planes` data frame.
- A **foreign key** uniquely identifies an observation in another data frame. For example, `flights$tailnum` is a foreign key because it appears in the `flights` data frame where it matches each flight to a unique plane.

A variable can be both a primary key *and* a foreign key. For example, `origin` is part of the `weather` primary key, and is also a foreign key for the `airports` data frame.

Relations

A primary key and the corresponding foreign key in another data frame form a **relation**. Relations are typically one-to-many.

For example, each flight has one plane, but each plane has many flights.

You can model many-to-many relations with a many-to-1 relation plus a 1-to-many relation.

For example, in this data there's a many-to-many relationship between airlines and airports: each airline flies to many airports; each airport hosts many airlines.

Mutating Joins

A mutating join allows you to combine variables from two data frames. It first matches observations by their keys, then copies across variables from one data frame to the other.

Imagine you want to add the full airline name to the `flights2` data. You can combine the `airlines` and `flights2` data frames with `left_join()`:

```r
flights2 %>%
  select(-origin, -dest) %>%
  left_join(airlines, by = "carrier") %>% head()
```

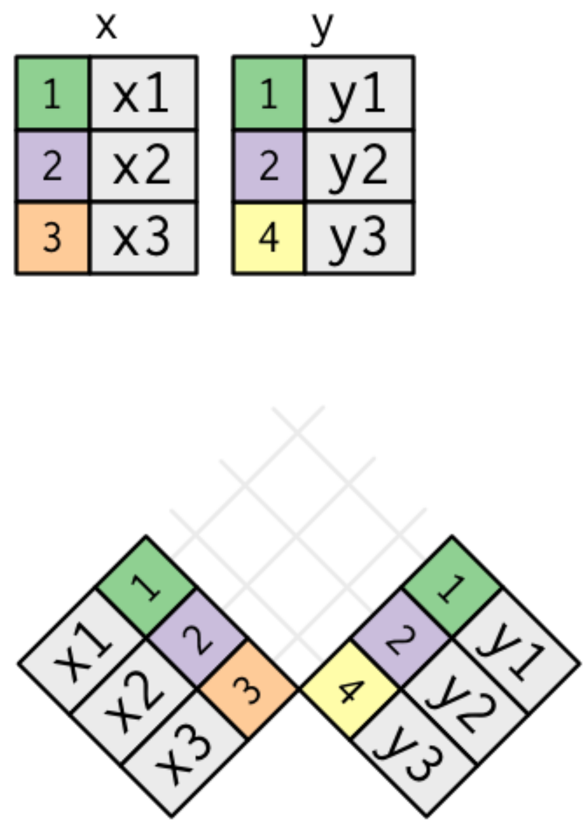
```
# A tibble: 6 × 7
   year month   day  hour tailnum carrier name
  <int> <int> <int> <dbl> <chr>   <chr>   <chr>
1  2013     1     1     5 N14228  UA      United Air Lines Inc.
2  2013     1     1     5 N24211  UA      United Air Lines Inc.
3  2013     1     1     5 N619AA  AA      American Airlines Inc.
4  2013     1     1     5 N804JB  B6      JetBlue Airways
5  2013     1     1     6 N668DN  DL      Delta Air Lines Inc.
6  2013     1     1     5 N39463  UA      United Air Lines Inc.
```

Understanding Joins

The colored column represents the “key” variable: these are used to match the rows between the data frames.

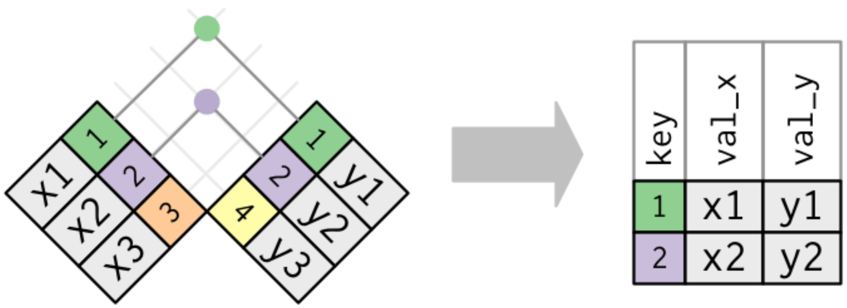
The grey column represents the “value” column that is carried along for the ride.

A join is a way of connecting each row in x to zero, one, or more rows in y.



Inner Join

The simplest type of join is the **inner join**. An inner join matches pairs of observations whenever their keys are equal:




```
x %>%
  inner_join(y, by = "key")
```

```
# A tibble: 2 × 3
    key val_x val_y
  <dbl> <chr> <chr>
1     1 x1    y1
2     2 x2    y2
```

In an inner join, unmatched rows are not included in the result. Inner joins are usually not appropriate for use in analysis because it's too easy to lose observations.

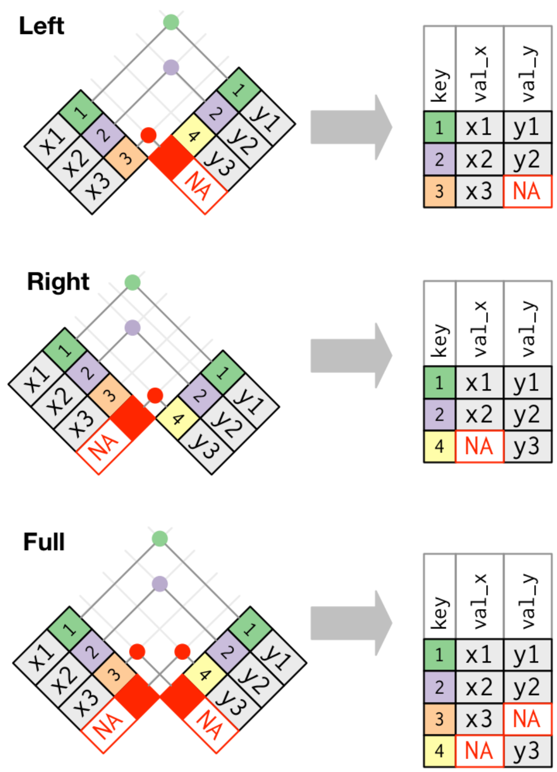
Outer Joins

An inner join keeps observations that appear in both data frames. An **outer join** keeps observations that appear in at least one of the data frames. There are three types of outer joins:

- A **left join** keeps all observations in `x`.
- A **right join** keeps all observations in `y`.
- A **full join** keeps all observations in `x` and `y`.

These joins work by adding an additional "virtual" observation to each data frame. This observation has a key that always matches (if no other key matches), and a value filled with `NA`.

Visualizing Outer Joins



Outer Joins in dplyr

```r
left_join(x, y, by = "key")
```

```
# A tibble: 3 × 3
    key val_x val_y
  <dbl> <chr> <chr>
1     1 x1    y1
2     2 x2    y2
3     3 x3    <NA>
```

```r
right_join(x, y, by = "key")
```

```
# A tibble: 3 × 3
    key val_x val_y
  <dbl> <chr> <chr>
1     1 x1    y1
2     2 x2    y2
3     4 <NA>  y3
```

```r
full_join(x, y, by = "key")
```

```
# A tibble: 4 × 3
    key val_x val_y
  <dbl> <chr> <chr>
1     1 x1    y1
2     2 x2    y2
3     3 x3    <NA>
4     4 <NA>  y3
```

Flight Data Joins

```
flights2 %>%
  left_join(airports, c("origin" = "faa")) %>% head()
```

```
# A tibble: 6 × 15
   year month   day  hour origin dest  tailnum carrier name      lat   lon   alt
  <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr>   <dbl> <dbl> <dbl>
1  2013     1     1     5 EWR    IAH   N14228  UA      Newark…  40.7 -74.2    18
2  2013     1     1     5 LGA    IAH   N24211  UA      La Gua…  40.8 -73.9    22
3  2013     1     1     5 JFK    MIA   N619AA  AA      John F…  40.6 -73.8    13
4  2013     1     1     5 JFK    BQN   N804JB  B6      John F…  40.6 -73.8    13
5  2013     1     1     6 LGA    ATL   N668DN  DL      La Gua…  40.8 -73.9    22
6  2013     1     1     5 EWR    ORD   N39463  UA      Newark…  40.7 -74.2    18
# ℹ 3 more variables: tz <dbl>, dst <chr>, tzone <chr>
```

```
flights2 %>%
  left_join(airports, c("origin" = "faa")) %>%
  filter(is.na(name))
```

```
# A tibble: 0 × 15
# ℹ 15 variables: year <int>, month <int>, day <int>, hour <dbl>, origin <chr>,
#   dest <chr>, tailnum <chr>, carrier <chr>, name <chr>, lat <dbl>, lon <dbl>,
#   alt <dbl>, tz <dbl>, dst <chr>, tzone <chr>
```

Working with Dates & Datetimes

There are three types of date/time data that refer to an instant in time:

- A **date**. Tibbles print this as `<date>`.
- A **time** within a day. Tibbles print this as `<time>`.
- A **date-time** is a date plus a time: it uniquely identifies an instant in time (typically to the nearest second). Tibbles print this as `<dttm>`.
- To get the current date or date-time you can use `today()` or `now()`

lubridate

The **lubridate** package makes it easier to work with dates and times in R.

lubridate is not part of core tidyverse because you only need it when you're working with dates/times.

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

Creating Dates From Strings

The helpers provided by lubridate automatically work out the format once you specify the order of the component.

To use them, identify the order in which year, month, and day appear in your dates, then arrange “y”, “m”, and “d” in the same order. That gives you the name of the lubridate function that will parse your date.

```r
ymd("2017-01-31")
```

```
[1] "2017-01-31"
```

```r
mdy("January 31st, 2017")
```

```
[1] "2017-01-31"
```

```r
dmy("31-Jan-2017")
```

```
[1] "2017-01-31"
```

Creating Dates from Components

To create a date/time from this sort of input, use `make_date()` for dates, or `make_datetime()` for date-times:

```
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))
```

```
# A tibble: 336,776 × 6
    year month   day  hour minute departure
   <int> <int> <int> <dbl>  <dbl> <dttm>
 1  2013     1     1     5     15 2013-01-01 05:15:00
 2  2013     1     1     5     29 2013-01-01 05:29:00
 3  2013     1     1     5     40 2013-01-01 05:40:00
 4  2013     1     1     5     45 2013-01-01 05:45:00
 5  2013     1     1     6      0 2013-01-01 06:00:00
 6  2013     1     1     5     58 2013-01-01 05:58:00
 7  2013     1     1     6      0 2013-01-01 06:00:00
 8  2013     1     1     6      0 2013-01-01 06:00:00
 9  2013     1     1     6      0 2013-01-01 06:00:00
10  2013     1     1     6      0 2013-01-01 06:00:00
# ℹ 336,766 more rows
```