R for Data Science

Code

1. Introduction

Prerequisites

R, RStudio, a collection of R packages called the **tidyverse**, and a handful of other packages. Packages are the fundamental units of reproducible R code.

R

To download R, go to CRAN, the **c**omprehensive **R a**rchive **n**etwork. CRAN is composed of a set of mirror servers distributed around the world and is used to distribute R and R packages.

RStudio

RStudio is an integrated development environment, or IDE, for R programming. Download and install it from http://www.rstudio.com/download (http://www.rstudio.com/download).

The tidyverse

An R *package* is a collection of functions, data, and documentation that extends the capabilities of base R. Using packages is key to the successful use of R.

You can install the complete tidyverse with a single line of code:

Hide

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

On your own computer, type that line of code in the console, and then press enter to run it. R will download the packages from CRAN and install them on to your computer. If you have problems installing, make sure that you are connected to the internet, and that https://cloud.r-project.org/ (https://cloud.r-project.org/) isn't blocked by your firewall or proxy.

You will not be able to use the functions, objects, and help files in a package until you load it with `library()`. Once you have installed a package, you can load it with the `library()` function:

Hide

```
library(tidyverse)
```

This tells you that tidyverse is loading the ggplot2, tibble, tidyr, readr, purrr, and dplyr packages. These are considered to be the **core** of the tidyverse because you'll use them in almost every analysis.

Other packages

There are many other excellent packages that are not part of the tidyverse, because they solve problems in a different domain, or are designed with a different set of underlying principles. This doesn't make them better or worse, just different. In other words, the complement to the tidyverse is not the messyverse, but many other universes of interrelated packages. As you tackle more data science projects with R, you'll learn new packages and new ways of thinking about data.

Hide

```
install.packages(c("nycflights13", "gapminder", "Lahman"))
```

These packages provide data on airline flights, world development, and baseball that we'll use to illustrate key data science ideas.

Running R code

The previous section showed you a couple of examples of running R code. Code in the book looks like this:

Hide

```
1 + 2
#> [1] 3
```

If you run the same code in your local console, it will look like this:

```
> 1 + 2
[1] 3
```

Getting help and learning more

If you get stuck, start with Google. Typically adding "R" to a query is enough to restrict it to relevant results: if the search isn't useful, it often means that there aren't any R-specific results available. Google is particularly useful for error messages. If you get an error message and you have no idea what it means, try googling it! Chances are that someone else has been confused by it in the past, and there will be help somewhere on the web. (If the error message isn't in English, run `Sys.setenv(LANGUAGE = "en")` and re-run the code; you're more likely to find help for English error messages.)

If Google doesn't help, try stackoverflow (http://stackoverflow.com). Start by spending a little time searching for an existing answer, including `[R]` restrict your search to questions and answers that use R. If you don't find anything useful, prepare a minimal reproducible example or **reprex**. A good reprex makes it easier for other people to help you, and often you'll figure out the problem yourself in the course of making it.

There are three things you need to include to make your example reproducible: required packages, data, and code.

1. **Packages** should be loaded at the top of the script, so it's easy to see which ones the example needs. This is a good time to check that you're using the latest version of each package; it's possible you've discovered a bug that's been fixed since you installed the package. For packages in the tidyverse, the easiest way to check is to run `tidyverse_update()`.

2. The easiest way to include **data** in a question is to use `dput()` to generate the R code to recreate it. For example, to recreate the `mtcars` dataset in R, I'd perform the following steps:

    1. Run `dput(mtcars)` in R
    2. Copy the output
    3. In my reproducible script, type `mtcars <-` then paste.

    Try and find the smallest subset of your data that still reveals the problem.

3. Spend a little bit of time ensuring that your **code** is easy for others to read:

    - Make sure you've used spaces and your variable names are concise, yet informative.
    - Use comments to indicate where your problem lies.

- Do your best to remove everything that is not related to the problem.
  The shorter your code is, the easier it is to understand, and the easier it is to fix.

Finish by checking that you have actually made a reproducible example by starting a fresh R session and copying and pasting your script in.

You should also spend some time preparing yourself to solve problems before they occur. Investing a little time in learning R each day will pay off handsomely in the long run. One way is to follow what Hadley, Garrett, and everyone else at RStudio are doing on the RStudio blog (https://blog.rstudio.org).

To keep up with the R community more broadly, we recommend reading http://www.r-bloggers.com (http://www.r-bloggers.com): it aggregates over 500 blogs about R from around the world. If you're an active Twitter user, follow the `#rstats` hashtag. Twitter is one of the key tools that Hadley uses to keep up with new developments in the community.

2. Workflow: basics

Coding basics

Let's review some basics we've so far omitted in the interests of getting you plotting as quickly as possible. You can use R as a calculator:

Hide

```
1 / 200 * 30
(59 + 73 + 2) / 3
sin(pi / 2)
```

You can create new objects with `<-`:

Hide

```
x <- 3 * 4
```

All R statements where you create objects, **assignment** statements, have the same form:

Hide

```
object_name <- value
```

When reading that code say "object name gets value" in your head.

You will make lots of assignments and `<-` is a pain to type. Don't be lazy and use `=`: it will work, but it will cause confusion later. Instead, use RStudio's keyboard shortcut: Alt + - (the minus sign). Notice that RStudio automagically surrounds `<-` with spaces, which is a good code formatting practice. Code is miserable to read on a good day, so giveyoureyesabreak and use spaces.

What's in a name?

Object names must start with a letter, and can only contain letters, numbers, `_` and `.`. You want your object names to be descriptive, so you'll need a convention for multiple words. I recommend **snake_case** where you separate lowercase words with `_`.

Hide

```
i_use_snake_case
otherPeopleUseCamelCase
some.people.use.periods
And_aFew.People_RENOUNCEconvention
```

We'll come back to code style later, in [functions].

You can inspect an object by typing its name:

Hide

```
x
```

Make another assignment:

Hide

```
this_is_a_really_long_name <- 2.5
```

To inspect this object, try out RStudio's completion facility: type "this", press TAB, add characters until you have a unique prefix, then press return.

Ooops, you made a mistake! `this_is_a_really_long_name` should have value 3.5 not 2.5. Use another keyboard shortcut to help you fix it. Type "this" then press Cmd/Ctrl + ↑. That will list all the commands you've typed that start those letters. Use the arrow keys to navigate, then press enter to retype the command. Change 2.5 to 3.5 and rerun.

Make yet another assignment:

Hide

```
r_rocks <- 2 ^ 3
```

Let's try to inspect it:

Hide

```
r_rock
#> Error: object 'r_rock' not found
R_rocks
#> Error: object 'R_rocks' not found
```

There's an implied contract between you and R: it will do the tedious computation for you, but in return, you must be completely precise in your instructions. Typos matter. Case matters.

Calling functions

R has a large collection of built-in functions that are called like this:

Hide

```
function_name(arg1 = val1, arg2 = val2, ...)
```

Let's try using `seq()` which makes regular **seq**uences of numbers and, while we're at it, learn more helpful features of RStudio. Type `se` and hit TAB. A popup shows you possible completions. Specify `seq()` by typing more (a "q") to disambiguate, or by using ↑/↓ arrows to select. Notice the floating tooltip that pops up,

reminding you of the function's arguments and purpose. If you want more help, press F1 to get all the details in help tab in the lower right pane.

Press TAB once more when you've selected the function you want. RStudio will add matching opening (`(`) and closing (`)`) parentheses for you. Type the arguments `1, 10` and hit return.

Hide

```
seq(1, 10)
```

Type this code and notice similar assistance help with the paired quotation marks:

Hide

```
x <- "hello world"
```

Quotation marks and parentheses must always come in a pair. RStudio does its best to help you, but it's still possible to mess up and end up with a mismatch. If this happens, R will show you the continuation character "+":

```
> x <- "hello
+
```

The `+` tells you that R is waiting for more input; it doesn't think you're done yet. Usually that means you've forgotten either a `"` or a `)`. Either add the missing pair, or press ESCAPE to abort the expression and try again.

If you make an assignment, you don't get to see the value. You're then tempted to immediately double-check the result:

Hide

```
y <- seq(1, 10, length.out = 5)
y
```

This common action can be shortened by surrounding the assignment with parentheses, which causes assignment and "print to screen" to happen.

Hide

```
(y <- seq(1, 10, length.out = 5))
```

Here you can see all of the objects that you've created.

Practice

1. Why does this code not work?

Hide

```
my_variable <- 10
my_varıable
```

Look carefully! (This may seem like an exercise in pointlessness, but training your brain to notice even the tiniest difference will pay off when programming.)

3. Data transformation

Introduction

Visualisation is an important tool for insight generation, but it is rare that you get the data in exactly the right form you need. Often you'll need to create some new variables or summaries, or maybe you just want to rename the variables or reorder the observations in order to make the data a little easier to work with. You'll learn how to do all that (and more!) in this chapter, which will teach you how to transform your data using the dplyr package and a new dataset on flights departing New York City in 2013.

Prerequisites

In this chapter we're going to focus on how to use the dplyr package, another core member of the tidyverse. We'll illustrate the key ideas using data from the nycflights13 package, and use ggplot2 to help us understand the data.

Hide

```
library(nycflights13)
library(tidyverse)
```

Take careful note of the conflicts message that's printed when you load the tidyverse. It tells you that dplyr overwrites some functions in base R. If you want to use the base version of these functions after loading dplyr, you'll need to use their full names: `stats::filter()` and `stats::lag()`.

nycflights13

To explore the basic data manipulation verbs of dplyr, we'll use `nycflights13::flights`. This data frame contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics (http://www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=120&Link=0), and is documented in `?flights`.

Hide

```
flights
```

You might notice that this data frame prints a little differently from other data frames you might have used in the past: it only shows the first few rows and all the columns that fit on one screen. (To see the whole dataset, you can run `View(flights)` which will open the dataset in the RStudio viewer). It prints differently because it's a **tibble**. Tibbles are data frames, but slightly tweaked to work better in the tidyverse. For now, you don't need to worry about the differences; we'll come back to tibbles in more detail in wrangle.

You might also have noticed the row of three (or four) letter abbreviations under the column names. These describe the type of each variable:

- `int` stands for integers.
- `dbl` stands for doubles, or real numbers.
- `chr` stands for character vectors, or strings.
- `dttm` stands for date-times (a date + a time).

There are three other common types of variables that aren't used in this dataset but you'll encounter later in the book:

- `lgl` stands for logical, vectors that contain only `TRUE` or `FALSE`.
- `fctr` stands for factors, which R uses to represent categorical variables with fixed possible values.
- `date` stands for dates.

dplyr basics

In this chapter you are going to learn the five key dplyr functions that allow you to solve the vast majority of your data manipulation challenges:

- 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. Let's dive in and see how these verbs work.

Filter rows with `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:

Hide

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

When you run that line of code, dplyr executes the filtering operation and returns a new data frame. dplyr functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, `<-`:

Hide

```
jan1 <- filter(flights, month == 1, day == 1)
```

R either prints out the results, or saves them to a variable. If you want to do both, you can wrap the assignment in parentheses:

Hide

```
(dec25 <- filter(flights, month == 12, day == 25))
```

Comparisons

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).

When you're starting out with R, the easiest mistake to make is to use `=` instead of `==` when testing for equality. When this happens you'll get an informative error:

Hide

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

There's another common problem you might encounter when using `==`: floating point numbers. These results might surprise you!

Hide

```
sqrt(2) ^ 2 == 2
1/49 * 49 == 1
```

Computers use finite precision arithmetic (they obviously can't store an infinite number of digits!) so remember that every number you see is an approximation. Instead of relying on `==`, use `near()`:

Hide

```
near(sqrt(2) ^ 2,  2)
near(1 / 49 * 49, 1)
```

Logical operators

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". Figure @ref(fig:bool-ops) shows the complete set of Boolean operations.

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

Hide

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

The order of operations doesn't work like English. You can't write `filter(flights, month == 11 | 12)`, which you might literally translate into "finds all flights that departed in November or December". Instead it finds all months that equal `11 | 12`, an expression that evaluates to `TRUE`. In a numeric context (like here), `TRUE` becomes one, so this finds all flights in January, not November or December. This is quite confusing!

A useful short-hand for this problem is `x %in% y`. This will select every row where `x` is one of the values in `y`. We could use it to rewrite the code above:

Hide

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

Sometimes you can simplify complicated subsetting by remembering De Morgan's law: `!(x & y)` is the same as `!x | !y`, and `!(x | y)` is the same as `!x & !y`. For example, if you wanted to find flights that weren't delayed (on arrival or departure) by more than two hours, you could use either of the following two filters:

Hide

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

As well as `&` and `|`, R also has `&&` and `||`. Don't use them here! You'll learn when you should use them in [conditional execution].

Whenever you start using complicated, multipart expressions in `filter()`, consider making them explicit variables instead. That makes it much easier to check your work. You'll learn how to create new variables shortly.

Missing values

One important feature of R that can make comparison tricky are missing values, or `NA`s ("not availables"). `NA` represents an unknown value so missing values are "contagious": almost any operation involving an unknown value will also be unknown.

Hide

```
NA > 5
10 == NA
NA + 10
NA / 2
```

The most confusing result is this one:

Hide

```
NA == NA
```

It's easiest to understand why this is true with a bit more context:

Hide

```
# Let x be Mary's age. We don't know how old she is.
x <- NA

# Let y be John's age. We don't know how old he is.
y <- NA

# Are John and Mary the same age?
x == y
# We don't know!
```

If you want to determine if a value is missing, use `is.na()`:

Hide

```
is.na(x)
```

`filter()` only includes rows where the condition is `TRUE`; it excludes both `FALSE` and `NA` values. If you want to preserve missing values, ask for them explicitly:

Hide

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)
filter(df, is.na(x) | x > 1)
```

Exercises

1. Find all flights that
    1. Had an arrival delay of two or more hours
    2. Flew to Houston (`IAH` or `HOU`)
    3. Were operated by United, American, or Delta
    4. Departed in summer (July, August, and September)
2. How many flights have a missing `dep_time` ? What other variables are missing?

Arrange rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

Hide

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

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

Hide

```
arrange(flights, desc(arr_delay))
```

Missing values are always sorted at the end:

Hide

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
arrange(df, desc(x))
```

Select columns with `select()`

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:

Hide

```
# Select columns by name
select(flights, year, month, day)
# Select all columns between year and day (inclusive)
select(flights, year:day)
# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
```

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:

Hide

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

Another option is to use `select()` in conjunction with the `everything()` helper. This is useful if you have a handful of variables you’d like to move to the start of the data frame.

Hide

```
select(flights, time_hour, air_time, everything())
```

Exercises

1. Does the result of running the following code surprise you? How do the select helpers deal with case by default? How can you change that default?

Hide

```
select(flights, contains("TIME"))
?select_helper
```

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. Remember that when you’re in RStudio, the easiest way to see all the columns is `View()`.

Hide

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

Note that you can refer to columns that you've just created:

Hide

```
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```

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

Hide

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

Useful creation functions

There are many functions for creating new variables that you can use with `mutate()`. The key property is that the function must be vectorised: it must take a vector of values as input, return a vector with the same number of values as output. There's no way to list every possible function that you might use, but here's a selection of functions that are frequently useful:

- Arithmetic operators: `+`, `-`, `*`, `/`, `^`. These are all vectorised, using the so called "recycling rules". If one parameter is shorter than the other, it will be automatically extended to be the same length. This is most useful when one of the arguments is a single number: `air_time / 60`, `hours * 60 + minute`, etc.

  Arithmetic operators are also useful in conjunction with the aggregate functions you'll learn about later. For example, `x / sum(x)` calculates the proportion of a total, and `y - mean(y)` computes the difference from the mean.

- Modular arithmetic: `%/%` (integer division) and `%%` (remainder), where `x == y * (x %/% y) + (x %% y)`. Modular arithmetic is a handy tool because it allows you to break integers up into pieces. For example, in the flights dataset, you can compute `hour` and `minute` from `dep_time` with:

  Hide

```
transmute(flights,
  dep_time,
  hour = dep_time %/% 100,
  minute = dep_time %% 100
)
```

- Logs: `log()`, `log2()`, `log10()`. Logarithms are an incredibly useful transformation for dealing with data that ranges across multiple orders of magnitude. They also convert multiplicative relationships to additive, a feature we'll come back to in modelling.

  All else being equal, I recommend using `log2()` because it's easy to interpret: a difference of 1 on the log scale corresponds to doubling on the original scale and a difference of -1 corresponds to halving.

- Offsets: `lead()` and `lag()` allow you to refer to leading or lagging values. This allows you to compute running differences (e.g. `x - lag(x)`) or find when values change (`x != lag(x)`). They are most useful in conjunction with `group_by()`, which you'll learn about shortly.

Hide

```
(x <- 1:10)
lag(x)
lead(x)
```

- Cumulative and rolling aggregates: R provides functions for running sums, products, mins and maxes: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`; and dplyr provides `cummean()` for cumulative means. If you need rolling aggregates (i.e. a sum computed over a rolling window), try the RcppRoll package.

Hide

```
x
cumsum(x)
cummean(x)
```

- Logical comparisons, `<`, `<=`, `>`, `>=`, `!=`, which you learned about earlier. If you're doing a complex sequence of logical operations it's often a good idea to store the interim values in new variables so you can check that each step is working as expected.

Grouped summaries with `summarise()`

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

Hide

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

(We'll come back to what that `na.rm = TRUE` means very shortly.)

`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:

Hide

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

Together `group_by()` and `summarise()` provide one of the tools that you'll use most commonly when working with dplyr: grouped summaries. But before we go any further with this, we need to introduce a powerful new idea: the pipe.

Combining multiple operations with the pipe

Imagine that we want to explore the relationship between the distance and average delay for each location. Using what you know about dplyr, you might write code like this:

Hide

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

# It looks like delays increase with distance up to ~750 miles
# and then decrease. Maybe as flights get longer there's more
# ability to make up delays in the air?
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
```

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.

This code is a little frustrating to write because we have to give each intermediate data frame a name, even though we don't care about it. Naming things is hard, so this slows down our analysis.

There's another way to tackle the same problem with the pipe, `%>%` :

Hide

```
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")
```

This focuses on the transformations, not what's being transformed, which makes the code easier to read. You can read it as a series of imperative statements: group, then summarise, then filter. As suggested by this reading, a good way to pronounce `%>%` when reading code is "then".

Behind the scenes, `x %>% f(y)` turns into `f(x, y)`, and `x %>% f(y) %>% g(z)` turns into `g(f(x, y), z)` and so on. You can use the pipe to rewrite multiple operations in a way that you can read left-to-right, top-to-bottom. We'll use piping frequently from now on because it considerably improves the readability of code, and we'll come back to it in more detail in [pipes].

Working with the pipe is one of the key criteria for belonging to the tidyverse. The only exception is ggplot2: it was written before the pipe was discovered. Unfortunately, the next iteration of ggplot2, ggvis, which does use the pipe, isn't quite ready for prime time yet.

Missing values

You may have wondered about the `na.rm` argument we used above. What happens if we don't set it?

Hide

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

We get a lot of missing values! That's because aggregation functions obey the usual rule of missing values: if there's any missing value in the input, the output will be a missing value. Fortunately, all aggregation functions have an `na.rm` argument which removes the missing values prior to computation:

Hide

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

In this case, where missing values represent cancelled flights, we could also tackle the problem by first removing the cancelled flights. We'll save this dataset so we can reuse in the next few examples.

Hide

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

Counts

Whenever you do any aggregation, it's always a good idea to include either a count (`n()`), or a count of non-missing values (`sum(!is.na(x))`). That way you can check that you're not drawing conclusions based on very small amounts of data. For example, let's look at the planes (identified by their tail number) that have the highest average delays:

Hide

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay)
  )

ggplot(data = delays, mapping = aes(x = delay)) +
  geom_freqpoly(binwidth = 10)
```

Wow, there are some planes that have an *average* delay of 5 hours (300 minutes)!

The story is actually a little more nuanced. We can get more insight if we draw a scatterplot of number of flights vs. average delay:

Hide

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

ggplot(data = delays, mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```

Not surprisingly, there is much greater variation in the average delay when there are few flights. The shape of this plot is very characteristic: whenever you plot a mean (or other summary) vs. group size, you'll see that the variation decreases as the sample size increases.

When looking at this sort of plot, it's often useful to filter out the groups with the smallest numbers of observations, so you can see more of the pattern and less of the extreme variation in the smallest groups. This is what the following code does, as well as showing you a handy pattern for integrating ggplot2 into dplyr flows. It's a bit painful that you have to switch from `%>%` to `+`, but once you get the hang of it, it's quite convenient.

Hide

```
delays %>%
  filter(n > 25) %>%
  ggplot(mapping = aes(x = n, y = delay)) +
    geom_point(alpha = 1/10)
```

RStudio tip: a useful keyboard shortcut is Cmd/Ctrl + Shift + P. This resends the previously sent chunk from the editor to the console. This is very convenient when you're (e.g.) exploring the value of `n` in the example above. You send the whole block once with Cmd/Ctrl + Enter, then you modify the value of `n` and press Cmd/Ctrl + Shift + P to resend the complete block.

Useful summary functions

Just using means, counts, and sum can get you a long way, but R provides many other useful summary functions:

- Measures of location: we've used `mean(x)`, but `median(x)` is also useful. The mean is the sum divided by the length; the median is a value where 50% of `x` is above it, and 50% is below it.

It's sometimes useful to combine aggregation with logical subsetting. We haven't talked about this sort of subsetting yet, but you'll learn more about it in subsetting.

Hide

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    avg_delay1 = mean(arr_delay),
    avg_delay2 = mean(arr_delay[arr_delay > 0]) # the average positive delay
  )
```

- Measures of spread: `sd(x)`, `IQR(x)`, `mad(x)`. The mean squared deviation, or standard deviation or sd for short, is the standard measure of spread. The interquartile range `IQR()` and median absolute deviation `mad(x)` are robust equivalents that may be more useful if you have outliers.

Hide

```
# Why is distance to some destinations more variable than to others?
not_cancelled %>%
  group_by(dest) %>%
  summarise(distance_sd = sd(distance)) %>%
  arrange(desc(distance_sd))
```

- Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`. Quantiles are a generalisation of the median. For example, `quantile(x, 0.25)` will find a value of `x` that is greater than 25% of the values, and less than the remaining 75%.

Hide

```
# When do the first and last flights leave each day?
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first = min(dep_time),
    last = max(dep_time)
  )
```

- Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`. These work similarly to `x[1]`, `x[2]`, and `x[length(x)]` but let you set a default value if that position does not exist (i.e. you're trying to get the 3rd element from a group that only has two elements). For example, we can find the first and last departure for each day:

Hide

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first_dep = first(dep_time),
    last_dep = last(dep_time)
  )
```

```
These functions are complementary to filtering on ranks. Filtering gives
you all variables, with each observation in a separate row:
```

Hide

```
not_cancelled %>%
  group_by(year, month, day) %>%
  mutate(r = min_rank(desc(dep_time))) %>%
  filter(r %in% range(r))
```

- Counts: You've seen `n()`, which takes no arguments, and returns the size of the current group. To count the number of non-missing values, use `sum(!is.na(x))`. To count the number of distinct (unique) values, use `n_distinct(x)`.

Hide

```
# Which destinations have the most carriers?
not_cancelled %>%
  group_by(dest) %>%
  summarise(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
```

```
Counts are so useful that dplyr provides a simple helper if all you want is
a count:
```

Hide

```
not_cancelled %>%
  count(dest)
```

```
You can optionally provide a weight variable. For example, you could use
this to "count" (sum) the total number of miles a plane flew:
```

Hide

```
not_cancelled %>%
  count(tailnum, wt = distance)
```

- Counts and proportions of logical values: `sum(x > 10)`, `mean(y == 0)`. When used with numeric functions, `TRUE` is converted to 1 and `FALSE` to 0. This makes `sum()` and `mean()` very useful: `sum(x)` gives the number of `TRUE`s in `x`, and `mean(x)` gives the proportion.

Hide

```
# How many flights left before 5am? (these usually indicate delayed
# flights from the previous day)
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(n_early = sum(dep_time < 500))
```

Hide

```
# What proportion of flights are delayed by more than an hour?
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(hour_perc = mean(arr_delay > 60))
```

Grouping by multiple variables

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll up a dataset:

Hide

```
daily <- group_by(flights, year, month, day)
(per_day   <- summarise(daily, flights = n()))
(per_month <- summarise(per_day, flights = sum(flights)))
(per_year  <- summarise(per_month, flights = sum(flights)))
```

Be careful when progressively rolling up summaries: it's OK for sums and counts, but you need to think about weighting means and variances, and it's not possible to do it exactly for rank-based statistics like the median. In other words, the sum of groupwise sums is the overall sum, but the median of groupwise medians is not the overall median.

Ungrouping

If you need to remove grouping, and return to operations on ungrouped data, use `ungroup()`.

Hide

```
daily %>%
  ungroup() %>%             # no longer grouped by date
  summarise(flights = n())  # all flights
```

Exercises

1. Brainstorm at least 5 different ways to assess the typical delay characteristics of a group of flights. Consider the following scenarios:
    - A flight is 15 minutes early 50% of the time, and 15 minutes late 50% of the time.
    - A flight is always 10 minutes late.
    - A flight is 30 minutes early 50% of the time, and 30 minutes late 50% of the time.
    - 99% of the time a flight is on time. 1% of the time it's 2 hours late.

    Which is more important: arrival delay or departure delay?
2. Come up with another approach that will give you the same output as `not_cancelled %>% count(dest)` and `not_cancelled %>% count(tailnum, wt = distance)` (without using `count()`).
3. Our definition of cancelled flights (`is.na(dep_delay) | is.na(arr_delay)`) is slightly suboptimal. Why? Which is the most important column?
4. Look at the number of cancelled flights per day. Is there a pattern? Is the proportion of cancelled flights related to the average delay?
5. Which carrier has the worst delays? Challenge: can you disentangle the effects of bad airports vs. bad carriers? Why/why not? (Hint: think about `flights %>% group_by(carrier, dest) %>% summarise(n())`)
6. For each plane, count the number of flights before the first delay of greater than 1 hour.
7. What does the `sort` argument to `count()` do. When might you use it?

Grouped mutates (and filters)

Grouping is most useful in conjunction with `summarise()`, but you can also do convenient operations with `mutate()` and `filter()`:

- Find the worst members of each group:

Hide

```
flights_sml %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 10)
```

- Find all groups bigger than a threshold:

Hide

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
popular_dests
```

- Standardise to compute per group metrics:

Hide

```
popular_dests %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
  select(year:day, dest, arr_delay, prop_delay)
```

A grouped filter is a grouped mutate followed by an ungrouped filter. I generally avoid them except for quick and dirty manipulations: otherwise it's hard to check that you've done the manipulation correctly.

Functions that work most naturally in grouped mutates and filters are known as window functions (vs. the summary functions used for summaries). You can learn more about useful window functions in the corresponding vignette: `vignette("window-functions")`.

Exercises

1. Refer back to the table of useful mutate and filtering functions. Describe how each operation changes when you combine it with grouping.
2. Which plane (`tailnum`) has the worst on-time record?
3. What time of day should you fly if you want to avoid delays as much as possible?
4. For each destination, compute the total minutes of delay. For each, flight, compute the proportion of the total delay for its destination.
5. Delays are typically temporally correlated: even once the problem that caused the initial delay has been resolved, later flights are delayed to allow earlier flights to leave. Using `lag()` explore how the delay of a flight is related to the delay of the immediately preceding flight.
6. Look at each destination. Can you find flights that are suspiciously fast? (i.e. flights that represent a potential data entry error). Compute the air time a flight relative to the shortest flight to that destination. Which flights were most delayed in the air?
7. Find all destinations that are flown by at least two carriers. Use that information to rank the carriers.

4. Workflow: scripts

So far you've been using the console to run code. That's a great place to start, but you'll find it gets cramped pretty quickly as you create more complex ggplot2 graphics and dplyr pipes. To give yourself more room to work, it's a great idea to use the script editor. Open it up either by clicking the File menu, and selecting New File, then R script, or using the keyboard shortcut Cmd/Ctrl + Shift + N. Now you'll see four panes:

The script editor is a great place to put code you care about. Keep experimenting in the console, but once you have written code that works and does what you want, put it in the script editor. RStudio will automatically save the contents of the editor when you quit RStudio, and will automatically load it when you re-open. Nevertheless, it's a good idea to save your scripts regularly and to back them up.

Running code

The script editor is also a great place to build up complex ggplot2 plots or long sequences of dplyr manipulations. The key to using the script editor effectively is to memorise one of the most important keyboard shortcuts: Cmd/Ctrl + Enter. This executes the current R expression in the console. For example, take the code below. If your cursor is at ▉, pressing Cmd/Ctrl + Enter will run the complete command that generates `not_cancelled`. It will also move the cursor to the next statement (beginning with `not_cancelled %>%`). That makes it easy to run your complete script by repeatedly pressing Cmd/Ctrl + Enter.

Hide

```
library(dplyr)
library(nycflights13)

not_cancelled <- flights %>%
  filter(!is.na(dep_delay)▉, !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

Instead of running expression-by-expression, you can also execute the complete script in one step: Cmd/Ctrl + Shift + S. Doing this regularly is a great way to check that you've captured all the important parts of your code in the script.

I recommend that you always start your script with the packages that you need. That way, if you share your code with others, they can easily see what packages they need to install. Note, however, that you should never include `install.packages()` or `setwd()` in a script that you share. It's very antisocial to change settings on someone else's computer!

When working through future chapters, I highly recommend starting in the editor and practicing your keyboard shortcuts. Over time, sending code to the console in this way will become so natural that you won't even think about it.

Practice

1. Go to the RStudio Tips twitter account, https://twitter.com/rstudiotips (https://twitter.com/rstudiotips) and find one tip that looks interesting. Practice using it!

2. What other common mistakes will RStudio diagnostics report? Read https://support.rstudio.com/hc/en-us/articles/205753617-Code-Diagnostics (https://support.rstudio.com/hc/en-us/articles/205753617-Code-Diagnostics) to find out.

5. Tibbles

Introduction

Throughout this book we work with "tibbles" instead of R's traditional `data.frame`. Tibbles *are* data frames, but they tweak some older behaviours to make life a little easier. R is an old language, and some things that were useful 10 or 20 years ago now get in your way. It's difficult to change base R without breaking existing code, so most innovation occurs in packages. Here we will describe the **tibble** package, which provides opinionated data frames that make working in the tidyverse a little easier. In most places, I'll use the term tibble and data frame interchangeably; when I want to draw particular attention to R's built-in data frame, I'll call them `data.frame`s.

If this chapter leaves you wanting to learn more about tibbles, you might enjoy `vignette("tibble")`.

Prerequisites

In this chapter we'll explore the **tibble** package, part of the core tidyverse.

Creating tibbles

Almost all of the functions that you'll use in this book produce tibbles, as tibbles are one of the unifying features of the tidyverse. Most other R packages use regular data frames, so you might want to coerce a data frame to a tibble. You can do that with `as_tibble()`:

Hide

```
as_tibble(iris)
```

You can create a new tibble from individual vectors with `tibble()`. `tibble()` will automatically recycle inputs of length 1, and allows you to refer to variables that you just created, as shown below.

Hide

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)
```

If you're already familiar with `data.frame()`, note that `tibble()` does much less: it never changes the type of the inputs (e.g. it never converts strings to factors!), it never changes the names of variables, and it never creates row names.

It's possible for a tibble to have column names that are not valid R variable names, aka **non-syntactic** names. For example, they might not start with a letter, or they might contain unusual characters like a space. To refer to these variables, you need to surround them with backticks, `` ` ``:

Hide

```
tb <- tibble(
  `:)` = "smile",
  ` ` = "space",
  `2000` = "number"
)
tb
```

You'll also need the backticks when working with these variables in other packages, like ggplot2, dplyr, and tidyr.

Tibbles vs. data.frame

There are two main differences in the usage of a tibble vs. a classic `data.frame` : printing and subsetting.

Printing

Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen. This makes it much easier to work with large data. In addition to its name, each column reports its type, a nice feature borrowed from `str()` :

Hide

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
```

Tibbles are designed so that you don't accidentally overwhelm your console when you print large data frames. But sometimes you need more output than the default display. There are a few options that can help.

First, you can explicitly `print()` the data frame and control the number of rows (`n`) and the `width` of the display. `width = Inf` will display all columns:

Hide

```
nycflights13::flights %>%
  print(n = 10, width = Inf)
```

You can also control the default print behaviour by setting options:

- `options(tibble.print_max = n, tibble.print_min = m)` : if more than `m` rows, print only `n` rows. Use `options(dplyr.print_min = Inf)` to always show all rows.
- Use `options(tibble.width = Inf)` to always print all columns, regardless of the width of the screen.

You can see a complete list of options by looking at the package help with `package?tibble` .

A final option is to use RStudio's built-in data viewer to get a scrollable view of the complete dataset. This is also often useful at the end of a long chain of manipulations.

Hide

```
nycflights13::flights %>%
  View()
```

Subsetting

So far all the tools you've learned have worked with complete data frames. If you want to pull out a single variable, you need some new tools, `$` and `[[` . `[[` can extract by name or position; `$` only extracts by name but is a little less typing.

Hide

```
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)

# Extract by name
df$x
df[["x"]]

# Extract by position
df[[1]]
```

To use these in a pipe, you'll need to use the special placeholder `.` :

Hide

```
df %>% .$x
df %>% .[["x"]]
```

Compared to a `data.frame` , tibbles are more strict: they never do partial matching, and they will generate a warning if the column you are trying to access does not exist.

Interacting with older code

Some older functions don't work with tibbles. If you encounter one of these functions, use `as.data.frame()` to turn a tibble back to a `data.frame` :

Hide

```
class(as.data.frame(tb))
```

The main reason that some older functions don't work with tibble is the `[` function. We don't use `[` much in this book because `dplyr::filter()` and `dplyr::select()` allow you to solve the same problems with clearer code (but you will learn a little about it in vector subsetting). With base R data frames, `[` sometimes returns a data frame, and sometimes returns a vector. With tibbles, `[` always returns another tibble.

Exercises

1. How can you tell if an object is a tibble? (Hint: try printing `mtcars` , which is a regular data frame).
2. Compare and contrast the following operations on a `data.frame` and equivalent tibble. What is different? Why might the default data frame behaviours cause you frustration?

Hide

```
df <- data.frame(abc = 1, xyz = "a")
df$x
df[, "xyz"]
df[, c("abc", "xyz")]
```

1. If you have the name of a variable stored in an object, e.g. `var <- "mpg"` , how can you extract the reference variable from a tibble?
2. Practice referring to non-syntactic names in the following data frame by:

1. Extracting the variable called `1`.
2. Plotting a scatterplot of `1` vs `2`.
3. Creating a new column called `3` which is `2` divided by `1`.
4. Renaming the columns to `one`, `two` and `three`.

Hide

```
annoying <- tibble(
  `1` = 1:10,
  `2` = `1` * 2 + rnorm(length(`1`))
)
```

1. What does `tibble::enframe()` do? When might you use it?
2. What option controls how many additional column names are printed at the footer of a tibble?

6. Data import

Introduction

Working with data provided by R packages is a great way to learn the tools of data science, but at some point you want to stop learning and start working with your own data. In this chapter, you'll learn how to read plain-text rectangular files into R. Here, we'll only scratch the surface of data import, but many of the principles will translate to other forms of data. We'll finish with a few pointers to packages that are useful for other types of data.

Prerequisites

In this chapter, you'll learn how to load flat files in R with the **readr** package, which is part of the core tidyverse.

Getting started

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 (https://github.com/Ironholds/webreadr) which is built on top of `read_log()` and provides many more helpful tools.)

These functions all have similar syntax: once you've mastered one, you can use the others with ease. For the rest of this chapter we'll focus on `read_csv()`. Not only are csv files one of the most common forms of data storage, but once you understand `read_csv()`, you can easily apply your knowledge to all the other functions in readr.

The first argument to `read_csv()` is the most important: it's the path to the file to read.

Hide

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

When you run `read_csv()` it prints out a column specification that gives the name and type of each column. That's an important part of readr, which we'll come back to in parsing a file.

You can also supply an inline csv file. This is useful for experimenting with readr and for creating reproducible examples to share with others:

Hide

```
read_csv("a,b,c
1,2,3
4,5,6")
```

In both cases `read_csv()` uses the first line of the data for the column names, which is a very common convention. There are two cases where you might want to tweak this behaviour:

1. Sometimes there are a few lines of metadata at the top of the file. You can use `skip = n` to skip the first `n` lines; or use `comment = "#"` to drop all lines that start with (e.g.) `#`.

```
```r
read_csv("The first line of metadata
  The second line of metadata
  x,y,z
  1,2,3", skip = 2)

read_csv("# A comment I want to skip
  x,y,z
  1,2,3", comment = "#")
```
```

1. The data might not have column names. You can use `col_names = FALSE` to tell `read_csv()` not to treat the first row as headings, and instead label them sequentially from `X1` to `Xn`:

```
```r
read_csv("1,2,3\n4,5,6", col_names = FALSE)
```
```

```
(`"\n"` is a convenient shortcut for adding a new line. You'll learn more
about it and other types of string escape in [string basics].)

Alternatively you can pass `col_names` a character vector which will be
used as the column names:
```

```
```r
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
```
```

Another option that commonly needs tweaking is `na`: this specifies the value (or values) that are used to represent missing values in your file:

Hide

```
read_csv("a,b,c\n1,2,.", na = ".")
```

This is all you need to know to read ~75% of CSV files that you'll encounter in practice. You can also easily adapt what you've learned to read tab separated files with `read_tsv()` and fixed width files with `read_fwf()`. To read in more challenging files, you'll need to learn more about how readr parses each column, turning them into R vectors.

Compared to base R

If you've used R before, you might wonder why we're not using `read.csv()`. There are a few good reasons to favour readr functions over the base equivalents:

- They are typically much faster (~10x) than their base equivalents. Long running jobs have a progress bar, so you can see what's happening. If you're looking for raw speed, try `data.table::fread()`. It doesn't fit quite so well into the tidyverse, but it can be quite a bit faster.
- They produce tibbles, they don't convert character vectors to factors, use row names, or munge the column names. These are common sources of frustration with the base R functions.
- They are more reproducible. Base R functions inherit some behaviour from your operating system and environment variables, so import code that works on your computer might not work on someone else's.

Exercises

1. What function would you use to read a file where fields were separated with "|"?
2. Apart from `file`, `skip`, and `comment`, what other arguments do `read_csv()` and `read_tsv()` have in common?
3. What are the most important arguments to `read_fwf()`?
4. Sometimes strings in a CSV file contain commas. To prevent them from causing problems they need to be surrounded by a quoting character, like `"` or `'`. By convention, `read_csv()` assumes that the quoting character will be `"`, and if you want to change it you'll need to use `read_delim()` instead. What arguments do you need to specify to read the following text into a data frame?

````
```r
"x,y\n1,'a,b'"
```
````

1. Identify what is wrong with each of the following inline CSV files. What happens when you run the code?

````
```r
read_csv("a,b\n1,2,3\n4,5,6")
read_csv("a,b,c\n1,2\n1,2,3,4")
read_csv("a,b\n\"1")
read_csv("a,b\n1,2\na,b")
read_csv("a;b\n1;3")
```
````

Parsing a vector

Before we get into the details of how readr reads files from disk, we need to take a little detour to talk about the `parse_*()` functions. These functions take a character vector and return a more specialised vector like a logical, integer, or date:

Hide

```
str(parse_logical(c("TRUE", "FALSE", "NA")))
str(parse_integer(c("1", "2", "3")))
str(parse_date(c("2010-01-01", "1979-10-14")))
```

These functions are useful in their own right, but are also an important building block for readr. Once you've learned how the individual parsers work in this section, we'll circle back and see how they fit together to parse a complete file in the next section.

Like all functions in the tidyverse, the `parse_*()` functions are uniform: the first argument is a character vector to parse, and the `na` argument specifies which strings should be treated as missing:

Hide

```
parse_integer(c("1", "231", ".", "456"), na = ".")
```

If parsing fails, you'll get a warning:

Hide

```
x <- parse_integer(c("123", "345", "abc", "123.45"))
```

And the failures will be missing in the output:

Hide

```
x
```

If there are many parsing failures, you'll need to use `problems()` to get the complete set. This returns a tibble, which you can then manipulate with dplyr.

Hide

```
problems(x)
```

Using parsers is mostly a matter of understanding what's available and how they deal with different types of input. There are eight particularly important parsers:

1. `parse_logical()` and `parse_integer()` parse logicals and integers respectively. There's basically nothing that can go wrong with these parsers so I won't describe them here further.
2. `parse_double()` is a strict numeric parser, and `parse_number()` is a flexible numeric parser. These are more complicated than you might expect because different parts of the world write numbers in different ways.
3. `parse_character()` seems so simple that it shouldn't be necessary. But one complication makes it quite important: character encodings.
4. `parse_factor()` create factors, the data structure that R uses to represent categorical variables with fixed and known values.
5. `parse_datetime()`, `parse_date()`, and `parse_time()` allow you to parse various date & time specifications. These are the most complicated because there are so many different ways of writing dates.

The following sections describe these parsers in more detail.

Numbers

It seems like it should be straightforward to parse a number, but three problems make it tricky:

1. People write numbers differently in different parts of the world. For example, some countries use `.` in between the integer and fractional parts of a real number, while others use `,`.
2. Numbers are often surrounded by other characters that provide some context, like “$1000” or “10%”.
3. Numbers often contain “grouping” characters to make them easier to read, like “1,000,000”, and these grouping characters vary around the world.

To address the first problem, readr has the notion of a “locale”, an object that specifies parsing options that differ from place to place. When parsing numbers, the most important option is the character you use for the decimal mark. You can override the default value of `.` by creating a new locale and setting the `decimal_mark` argument:

Hide

```
parse_double("1.23")
parse_double("1,23", locale = locale(decimal_mark = ","))
```

readr’s default locale is US-centric, because generally R is US-centric (i.e. the documentation of base R is written in American English). An alternative approach would be to try and guess the defaults from your operating system. This is hard to do well, and, more importantly, makes your code fragile: even if it works on your computer, it might fail when you email it to a colleague in another country.

`parse_number()` addresses the second problem: it ignores non-numeric characters before and after the number. This is particularly useful for currencies and percentages, but also works to extract numbers embedded in text.

Hide

```
parse_number("$100")
parse_number("20%")
parse_number("It cost $123.45")
```

The final problem is addressed by the combination of `parse_number()` and the locale as `parse_number()` will ignore the “grouping mark”:

Hide

```
# Used in America
parse_number("$123,456,789")

# Used in many parts of Europe
parse_number("123.456.789", locale = locale(grouping_mark = "."))

# Used in Switzerland
parse_number("123'456'789", locale = locale(grouping_mark = "'"))
```

Strings

It seems like `parse_character()` should be really simple — it could just return its input. Unfortunately life isn't so simple, as there are multiple ways to represent the same string. To understand what's going on, we need to dive into the details of how computers represent strings. In R, we can get at the underlying representation of a string using `charToRaw()`:

Hide

```
charToRaw("Hadley")
```

Each hexadecimal number represents a byte of information: `48` is H, `61` is a, and so on. The mapping from hexadecimal number to character is called the encoding, and in this case the encoding is called ASCII. ASCII does a great job of representing English characters, because it's the **American** Standard Code for Information Interchange.

Things get more complicated for languages other than English. In the early days of computing there were many competing standards for encoding non-English characters, and to correctly interpret a string you needed to know both the values and the encoding. For example, two common encodings are Latin1 (aka ISO-8859-1, used for Western European languages) and Latin2 (aka ISO-8859-2, used for Eastern European languages). In Latin1, the byte `b1` is "±", but in Latin2, it's "ą"! Fortunately, today there is one standard that is supported almost everywhere: UTF-8. UTF-8 can encode just about every character used by humans today, as well as many extra symbols (like emoji!).

readr uses UTF-8 everywhere: it assumes your data is UTF-8 encoded when you read it, and always uses it when writing. This is a good default, but will fail for data produced by older systems that don't understand UTF-8. If this happens to you, your strings will look weird when you print them. Sometimes just one or two characters might be messed up; other times you'll get complete gibberish. For example:

Hide

```
x1 <- "El Ni\xf1o was particularly bad this year"
x2 <- "\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"

x1
x2
```

To fix the problem you need to specify the encoding in `parse_character()`:

Hide

```
parse_character(x1, locale = locale(encoding = "Latin1"))
parse_character(x2, locale = locale(encoding = "Shift-JIS"))
```

How do you find the correct encoding? If you're lucky, it'll be included somewhere in the data documentation. Unfortunately, that's rarely the case, so readr provides `guess_encoding()` to help you figure it out. It's not foolproof, and it works better when you have lots of text (unlike here), but it's a reasonable place to start. Expect to try a few different encodings before you find the right one.

Hide

```
guess_encoding(charToRaw(x1))
guess_encoding(charToRaw(x2))
```

The first argument to `guess_encoding()` can either be a path to a file, or, as in this case, a raw vector (useful if the strings are already in R).

Encodings are a rich and complex topic, and I've only scratched the surface here. If you'd like to learn more I'd recommend reading the detailed explanation at http://kunststube.net/encoding/ (http://kunststube.net/encoding/).

Factors

R uses factors to represent categorical variables that have a known set of possible values. Give `parse_factor()` a vector of known `levels` to generate a warning whenever an unexpected value is present:

Hide

```
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
```

But if you have many problematic entries, it's often easier to leave as character vectors and then use the tools you'll learn about in strings and factors to clean them up.

Dates, date-times, and times

You pick between three parsers depending on whether you want a date (the number of days since 1970-01-01), a date-time (the number of seconds since midnight 1970-01-01), or a time (the number of seconds since midnight). When called without any additional arguments:

- `parse_datetime()` expects an ISO8601 date-time. ISO8601 is an international standard in which the components of a date are organised from biggest to smallest: year, month, day, hour, minute, second.

````
```r
parse_datetime("2010-10-01T2010")
# If time is omitted, it will be set to midnight
parse_datetime("20101010")
```
````

```
This is the most important date/time standard, and if you work with
dates and times frequently, I recommend reading
<https://en.wikipedia.org/wiki/ISO_8601>
```

- `parse_date()` expects a four digit year, a `-` or `/`, the month, a `-` or `/`, then the day:

````
```r
parse_date("2010-10-01")
```
````

- `parse_time()` expects the hour, `:`, minutes, optionally `:` and seconds, and an optional am/pm specifier:

````
```r
library(hms)
parse_time("01:10 am")
parse_time("20:10:01")
```
````

```
Base R doesn't have a great built in class for time data, so we use
the one provided in the hms package.
```

If these defaults don't work for your data you can supply your own date-time `format`, built up of the following pieces:

Year

`%Y` (4 digits).

`%y` (2 digits); 00-69 -> 2000-2069, 70-99 -> 1970-1999.

Month

`%m` (2 digits).

`%b` (abbreviated name, like "Jan").

`%B` (full name, "January").

Day

`%d` (2 digits).

`%e` (optional leading space).

Time

`%H` 0-23 hour.

`%I` 0-12, must be used with `%p`.

`%p` AM/PM indicator.

`%M` minutes.

`%S` integer seconds.

`%OS` real seconds.

`%Z` Time zone (as name, e.g. `America/Chicago`). Beware of abbreviations: if you're American, note that "EST" is a Canadian time zone that does not have daylight savings time. It is *not* Eastern Standard Time! We'll come back to this [time zones].

`%z` (as offset from UTC, e.g. `+0800`).

Non-digits

`%.` skips one non-digit character.

`%*` skips any number of non-digits.

The best way to figure out the correct format is to create a few examples in a character vector, and test with one of the parsing functions. For example:

Hide

```
parse_date("01/02/15", "%m/%d/%y")
parse_date("01/02/15", "%d/%m/%y")
parse_date("01/02/15", "%y/%m/%d")
```

If you're using `%b` or `%B` with non-English month names, you'll need to set the `lang` argument to `locale()`. See the list of built-in languages in `date_names_langs()`, or if your language is not already included, create your own with `date_names()`.

Hide

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
```

Exercises

1. What are the most important arguments to `locale()`?
2. What happens if you try and set `decimal_mark` and `grouping_mark` to the same character? What happens to the default value of `grouping_mark` when you set `decimal_mark` to ","? What happens to the default value of `decimal_mark` when you set the `grouping_mark` to "."?
3. I didn't discuss the `date_format` and `time_format` options to `locale()`. What do they do? Construct an example that shows when they might be useful.

4. If you live outside the US, create a new locale object that encapsulates the settings for the types of file you read most commonly.
5. What's the difference between `read_csv()` and `read_csv2()`?
6. What are the most common encodings used in Europe? What are the most common encodings used in Asia? Do some googling to find out.
7. Generate the correct format string to parse each of the following dates and times:

```
```r
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```
```

Parsing a file

Now that you've learned how to parse an individual vector, it's time to return to the beginning and explore how readr parses a file. There are two new things that you'll learn about in this section:

1. How readr automatically guesses the type of each column.
2. How to override the default specification.

Strategy

readr uses a heuristic to figure out the type of each column: it reads the first 1000 rows and uses some (moderately conservative) heuristics to figure out the type of each column. You can emulate this process with a character vector using `guess_parser()`, which returns readr's best guess, and `parse_guess()` which uses that guess to parse the column:

Hide

```
guess_parser("2010-10-01")
guess_parser("15:01")
guess_parser(c("TRUE", "FALSE"))
guess_parser(c("1", "5", "9"))
guess_parser(c("12,352,561"))

str(parse_guess("2010-10-10"))
```

The heuristic tries each of the following types, stopping when it finds a match:

- logical: contains only "F", "T", "FALSE", or "TRUE".
- integer: contains only numeric characters (and `-`).
- double: contains only valid doubles (including numbers like `4.5e-5`).
- number: contains valid doubles with the grouping mark inside.
- time: matches the default `time_format`.
- date: matches the default `date_format`.
- date-time: any ISO8601 date.

If none of these rules apply, then the column will stay as a vector of strings.

Problems

These defaults don't always work for larger files. There are two basic problems:

1. The first thousand rows might be a special case, and readr guesses a type that is not sufficiently general. For example, you might have a column of doubles that only contains integers in the first 1000 rows.
2. The column might contain a lot of missing values. If the first 1000 rows contain only `NA` s, readr will guess that it's a character vector, whereas you probably want to parse it as something more specific.

readr contains a challenging CSV that illustrates both of these problems:

Hide

```
challenge <- read_csv(readr_example("challenge.csv"))
```

(Note the use of `readr_example()` which finds the path to one of the files included with the package)

There are two printed outputs: the column specification generated by looking at the first 1000 rows, and the first five parsing failures. It's always a good idea to explicitly pull out the `problems()` , so you can explore them in more depth:

Hide

```
problems(challenge)
```

A good strategy is to work column by column until there are no problems remaining. Here we can see that there are a lot of parsing problems with the `x` column - there are trailing characters after the integer value. That suggests we need to use a double parser instead.

To fix the call, start by copying and pasting the column specification into your original call:

Hide

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_integer(),
    y = col_character()
  )
)
```

Then you can tweak the type of the `x` column:

Hide

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_character()
  )
)
```

That fixes the first problem, but if we look at the last few rows, you'll see that they're dates stored in a character vector:

Hide

```
tail(challenge)
```

You can fix that by specifying that `y` is a date column:

Hide

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_date()
  )
)
tail(challenge)
```

Every `parse_xyz()` function has a corresponding `col_xyz()` function. You use `parse_xyz()` when the data is in a character vector in R already; you use `col_xyz()` when you want to tell readr how to load the data.

I highly recommend always supplying `col_types`, building up from the print-out provided by readr. This ensures that you have a consistent and reproducible data import script. If you rely on the default guesses and your data changes, readr will continue to read it in. If you want to be really strict, use `stop_for_problems()`: that will throw an error and stop your script if there are any parsing problems.

Other strategies

There are a few other general strategies to help you parse files:

- In the previous example, we just got unlucky: if we look at just one more row than the default, we can correctly parse in one shot:

```
```r
challenge2 <- read_csv(readr_example("challenge.csv"), guess_max = 1001)
challenge2
```
```

- Sometimes it's easier to diagnose problems if you just read in all the columns as character vectors:

```
```r
challenge2 <- read_csv(readr_example("challenge.csv"),
  col_types = cols(.default = col_character())
)
```
```

```
This is particularly useful in conjunction with `type_convert()`,
which applies the parsing heuristics to the character columns in a data
frame.
```

```r
df <- tribble(
  ~x,  ~y,
  "1", "1.21",
  "2", "2.32",
  "3", "4.56"
)
df

# Note the column types
type_convert(df)
```

- If you're reading a very large file, you might want to set `n_max` to a smallish number like 10,000 or 100,000. That will accelerate your iterations while you eliminate common problems.
- If you're having major parsing problems, sometimes it's easier to just read into a character vector of lines with `read_lines()`, or even a character vector of length 1 with `read_file()`. Then you can use the string parsing skills you'll learn later to parse more exotic formats.

Writing to a file

readr also comes with two useful functions for writing data back to disk: `write_csv()` and `write_tsv()`. Both functions increase the chances of the output file being read back in correctly by:

- Always encoding strings in UTF-8.
- Saving dates and date-times in ISO8601 format so they are easily parsed elsewhere.

If you want to export a csv file to Excel, use `write_excel_csv()` — this writes a special character (a "byte order mark") at the start of the file which tells Excel that you're using the UTF-8 encoding.

The most important arguments are `x` (the data frame to save), and `path` (the location to save it). You can also specify how missing values are written with `na`, and if you want to `append` to an existing file.

Hide

```
write_csv(challenge, "challenge.csv")
```

Note that the type information is lost when you save to csv:

Hide

```
challenge
write_csv(challenge, "challenge-2.csv")
read_csv("challenge-2.csv")
```

This makes CSVs a little unreliable for caching interim results—you need to recreate the column specification every time you load in. There are two alternatives:

1. `write_rds()` and `read_rds()` are uniform wrappers around the base functions `readRDS()` and `saveRDS()`. These store data in R's custom binary format called RDS:

```r
write_rds(challenge, "challenge.rds")
read_rds("challenge.rds")
```

1. The feather package implements a fast binary file format that can be shared across programming languages:

```r
library(feather)
write_feather(challenge, "challenge.feather")
read_feather("challenge.feather")
#> # A tibble: 2,000 x 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

Feather tends to be faster than RDS and is usable outside of R. RDS supports list-columns (which you'll learn about in [many models]); feather currently does not.

Other types of data

To get other types of data into R, we recommend starting with the tidyverse packages listed below. They're certainly not perfect, but they are a good place to start. For rectangular data:

- **haven** reads SPSS, Stata, and SAS files.
- **readxl** reads excel files (both `.xls` and `.xlsx`).
- **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.

For hierarchical data: use **jsonlite** (by Jeroen Ooms) for json, and **xml2** for XML. Jenny Bryan has some excellent worked examples at https://jennybc.github.io/purrr-tutorial/examples.html (https://jennybc.github.io/purrr-tutorial/examples.html).

For other file types, try the R data import/export manual (https://cran.r-project.org/doc/manuals/r-release/R-data.html) and the **rio** (https://github.com/leeper/rio) package.

7. Tidy data

Introduction

> "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

In this chapter, you will learn a consistent way to organise your data in R, an organisation called **tidy data**. Getting your data into this format requires some upfront work, but that work pays off in the long term. Once you have tidy data and the tidy tools provided by packages in the tidyverse, you will spend much less time munging data from one representation to another, allowing you to spend more time on the analytic questions at hand.

This chapter will give you a practical introduction to tidy data and the accompanying tools in the **tidyr** package. If you'd like to learn more about the underlying theory, you might enjoy the *Tidy Data* paper published in the Journal of Statistical Software, http://www.jstatsoft.org/v59/i10/paper (http://www.jstatsoft.org/v59/i10/paper).

Prerequisites

In this chapter we'll focus on tidyr, a package that provides a bunch of tools to help tidy up your messy datasets. tidyr is a member of the core tidyverse.

Tidy data

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.

Hide

```
table1
table2
table3

# Spread across two tibbles
table4a  # cases
table4b  # population
```

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:

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

Figure @ref(fig:tidy-structure) shows the rules visually.

These three rules are interrelated because it's impossible to only satisfy two of the three. That interrelationship leads to an even simpler set of practical instructions:

1. Put each dataset in a tibble.
2. Put each variable in a column.

In this example, only `table1` is tidy. It's the only representation where each column is a variable.

Why ensure that your data is tidy? There are two main advantages:

1. There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.

2. There's a specific advantage to placing variables in columns because it allows R's vectorised nature to shine. As you learned in mutate and summary functions, most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

dplyr, ggplot2, and all the other packages in the tidyverse are designed to work with tidy data. Here are a couple of small examples showing how you might work with `table1`.

Hide

```
# Compute rate per 10,000
table1 %>%
  mutate(rate = cases / population * 10000)

# Compute cases per year
table1 %>%
  count(year, wt = cases)

# Visualise changes over time
library(ggplot2)
ggplot(table1, aes(year, cases)) +
  geom_line(aes(group = country), colour = "grey50") +
  geom_point(aes(colour = country))
```

Exercises

1. Using prose, describe how the variables and observations are organised in each of the sample tables.
2. Compute the `rate` for `table2`, and `table4a + table4b`. You will need to perform four operations:
    1. Extract the number of TB cases per country per year.
    2. Extract the matching population per country per year.
    3. Divide cases by population, and multiply by 10000.
    4. Store back in the appropriate place.

    Which representation is easiest to work with? Which is hardest? Why?
3. Recreate the plot showing change in cases over time using `table2` instead of `table1`. What do you need to do first?

Spreading and gathering

The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons:

1. Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.
2. Data is often organised to facilitate some use other than analysis. For example, data is often organised to make entry as easy as possible.

This means for most real analyses, you'll need to do some tidying. The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. The second step is to resolve one of two common problems:

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

Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in tidyr: `gather()` and `spread()`.

Gathering

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take `table4a`: the column names `1999` and `2000` represent values of the `year` variable, and each row represents two observations, not one.

Hide

```
table4a
```

To tidy a dataset like this, we need to **gather** those columns into a new pair of variables. To describe that operation we need three parameters:

- The set of columns that represent values, not variables. In this example, those are the columns `1999` and `2000`.
- The name of the variable whose values form the column names. I call that the `key`, and here it is `year`.
- The name of the variable whose values are spread over the cells. I call that `value`, and here it's the number of `cases`.

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

Hide

```
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
```

The columns to gather are specified with `dplyr::select()` style notation. Here there are only two columns, so we list them individually. Note that "1999" and "2000" are non-syntactic names so we have to surround them in backticks. To refresh your memory of the other ways to select columns, see select.

In the final result, the gathered columns are dropped, and we get new `key` and `value` columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in Figure @ref(fig:tidy-gather). We can use `gather()` to tidy `table4b` in a similar fashion. The only difference is the variable stored in the cell values:

Hide

```
table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
```

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we need to use `dplyr::left_join()`, which you'll learn about in [relational data].

Hide

```
tidy4a <- table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
tidy4b <- table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
left_join(tidy4a, tidy4b)
```

Spreading

Spreading is the opposite of gathering. 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.

Hide

```
table2
```

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

- The column that contains variable names, the `key` column. Here, it's `type` .
- The column that contains values forms multiple variables, the `value` column. Here it's `count` .

Once we've figured that out, we can use `spread()` , as shown programmatically below, and visually in Figure @ref(fig:tidy-spread).

Hide

```
spread(table2, key = type, value = count)
```

As you might have guessed from the common `key` and `value` arguments, `spread()` and `gather()` are complements. `gather()` makes wide tables narrower and longer; `spread()` makes long tables shorter and wider.

Exercises

1. Why are `gather()` and `spread()` not perfectly symmetrical?
   Carefully consider the following example:

````
```r
stocks <- tibble(
  year   = c(2015, 2015, 2016, 2016),
  half  = c(   1,    2,     1,    2),
  return = c(1.88, 0.59, 0.92, 0.17)
)
stocks %>%
  spread(year, return) %>%
  gather("year", "return", `2015`:`2016`)
```
````

```
(Hint: look at the variable types and think about column _names_.)

Both `spread()` and `gather()` have a `convert` argument. What does it
do?
```

1. Why does this code fail?

````
```r
table4a %>%
  gather(1999, 2000, key = "year", value = "cases")
```
````

1. Why does spreading this tibble fail? How could you add a new column to fix the problem?

```r
people <- tribble(
  ~name,             ~key,    ~value,
  #-----------------|--------|------
  "Phillip Woods",   "age",       45,
  "Phillip Woods",   "height",   186,
  "Phillip Woods",   "age",       50,
  "Jessica Cordero", "age",       37,
  "Jessica Cordero", "height",   156
)
```

1. Tidy the simple tibble below. Do you need to spread or gather it? What are the variables?

```r
preg <- tribble(
  ~pregnant, ~male, ~female,
  "yes",     NA,    10,
  "no",      20,    12
)
```

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

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

Hide

```
table3
```

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, as shown in Figure @ref(fig:tidy-separate) and the code below.

Hide

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

By default, `separate()` will split values wherever it sees a non-alphanumeric character (i.e. a character that isn't a number or letter). For example, in the code above, `separate()` split the values of `rate` at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the `sep` argument of `separate()`. For example, we could rewrite the code above as:

Hide

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

(Formally, `sep` is a regular expression, which you'll learn more about in strings.)

Look carefully at the column types: you'll notice that `case` and `population` are character columns. This is the default behaviour in `separate()`: it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using `convert = TRUE`:

Hide

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

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far-left of the strings; negative value start at -1 on the far-right of the strings. When using integers to separate strings, the length of `sep` should be one less than the number of names in `into`.

You can use this arrangement to separate the last two digits of each year. This make this data less tidy, but is useful in other cases, as you'll see in a little bit.

Hide

```
table3 %>%
  separate(year, into = c("century", "year"), sep = 2)
```

Unite

`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 that we created in the last example. That data is saved as `tidyr::table5`. `unite()` takes a data frame, the name of the new variable to create, and a set of columns to combine, again specified in `dplyr::select()` style:

Hide

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

In this case we also need to use the `sep` argument. The default will place an underscore (_) between the values from different columns. Here we don't want any separator so we use `""`:

Hide

```
table5 %>%
  unite(new, century, year, sep = "")
```

Exercises

1. What do the `extra` and `fill` arguments do in `separate()`? Experiment with the various options for the following two toy datasets.

```r
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"))

tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"))
```

1. Both `unite()` and `separate()` have a `remove` argument. What does it do? Why would you set it to `FALSE`?
2. Compare and contrast `separate()` and `extract()`. Why are there three variations of separation (by position, by separator, and with groups), but only one unite?

Missing values

Changing the representation of a dataset brings up an important subtlety of missing values. Surprisingly, a value can be missing in one of two possible ways:

- **Explicitly**, i.e. flagged with `NA`.
- **Implicitly**, i.e. simply not present in the data.

Let's illustrate this idea with a very simple data set:

Hide

```
stocks <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c(   1,    2,    3,    4,    2,    3,    4),
  return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
```

There are two missing values in this dataset:

- The return for the fourth quarter of 2015 is explicitly missing, because the cell where its value should be instead contains `NA`.
- The return for the first quarter of 2016 is implicitly missing, because it simply does not appear in the dataset.

One way to think about the difference is with this Zen-like koan: An explicit missing value is the presence of an absence; an implicit missing value is the absence of a presence.

The way that a dataset is represented can make implicit values explicit. For example, we can make the implicit missing value explicit by putting years in the columns:

Hide

```
stocks %>%
  spread(year, return)
```

Because these explicit missing values may not be important in other representations of the data, you can set `na.rm = TRUE` in `gather()` to turn explicit missing values implicit:

Hide

```
stocks %>%
  spread(year, return) %>%
  gather(year, return, `2015`:`2016`, na.rm = TRUE)
```

Another important tool for making missing values explicit in tidy data is `complete()`:

Hide

```
stocks %>%
  complete(year, qtr)
```

`complete()` takes a set of columns, and finds all unique combinations. It then ensures the original dataset contains all those values, filling in explicit `NA`s where necessary.

There's one other important tool that you should know for working with missing values. Sometimes when a data source has primarily been used for data entry, missing values indicate that the previous value should be carried forward:

Hide

```
treatment <- tribble(
  ~ person,           ~ treatment, ~response,
  "Derrick Whitmore", 1,           7,
  NA,                 2,           10,
  NA,                 3,           9,
  "Katherine Burke",  1,           4
)
```

You can fill in these missing values with `fill()`. It takes a set of columns where you want missing values to be replaced by the most recent non-missing value (sometimes called last observation carried forward).

Hide

```
treatment %>%
  fill(person)
```

Exercises

1. Compare and contrast the `fill` arguments to `spread()` and `complete()`.
2. What does the direction argument to `fill()` do?