



PHW251 Week 4 Reader

Topic 1: Reproducible Examples and How to Solve Your Own Problems

Lecture 1: Reproducible Examples and How to Solve Your Own Problems. 2

Tidyverse/reprex documentation. 11

R for Data Science (2e) Chapter 9: Workflow: getting help...... 15

Topic: Dates

Lecture 2: Dates. 18

R for Data Science (2e) Chapter 18: Dates & Times......27

Topic 1: Reproducible Examples and How to Solve Your Own Problems

How to solve your own problems

You have an issue, now what?

Programming often involves getting stuck. Balancing independent problem-solving with seeking external help is vital to becoming a successful programmer. It's important to challenge yourself to uncover issues, but avoid excessive frustration that hampers your productivity.

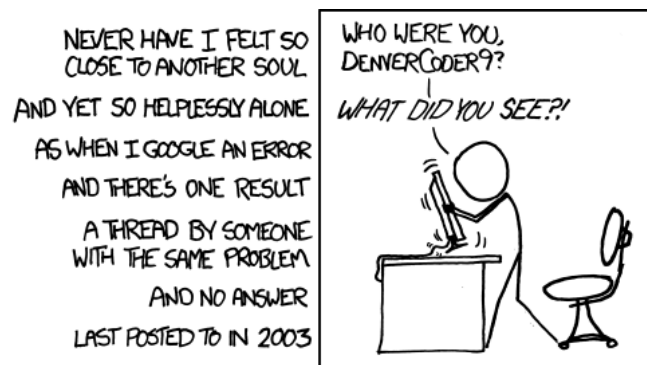
Where to get help [🔗](#)

You've exhausted your troubleshooting in R, re-read the help page and package documentation too many times, where to next? Luckily you have a few options! You may first want to start with your favorite search engine (e.g. Google).

Google / Search Engines

The art of Googling is a learned skill. For example, let's say I'm trying to filter rows in R and I don't know how or I've forgotten. I could try searching "r how to filter rows" and find this helpful [r-bloggers post answer that question](#). But if I had searched "r filter rows i'm stuck" I get less specific results, potentially missing out on solutions.

Though sometimes, you may find yourself in this situation.



Hello everyone. Today we'll be discussing a very important topic, and that's how to solve your own problems. As a programmer, it's crucial that you balance figuring out solutions to problems on your own with seeking external help.

Of course, you should challenge yourself. There's lots to gain from learning how to figure things out on your own, but you also need to avoid too much frustration, which will hamper your productivity. What happens when you get stuck? You've already tried maybe all of the troubleshooting steps in R and reread the help page and package documentation way too many times. What's next? Luckily you have quite a few options.

The first line of defense can be your favorite search engine like Google. Remember, the art of googling is a learned skill. The more specific your search terms and the more specific results that you'll get. For example, searching something like r how to filter rows might give you more precise results than just putting in r filter rows I'm stuck.

This very funny cartoon, one of my favorites that just wanted to include the longer that you Google things, I'm sure that you'll encounter. Let's say that you've exhausted where Google can help. That's where you can turn to online communities for help.

Stack Overflow

[Stack Overflow](#) is a public platform where users post coding questions and the community provides answers, voting for the “best” answer. You’ll find many of your future questions answered on the platform.

GitHub

GitHub is typically where developers store their open source code, allowing users (like us) to post issues. These issues are directly reviewed by the developers if that particular code base is active. For example, here’s a look at the tidyverse package on [GitHub with specific issues related to dplyr that users have](#).

Posit Community

Posit is the open-source data science company that created and maintains RStudio. They have built a community around supporting each other and have a [forum you can post questions to](#).

r4ds

You may also consider joining online communities such as the [r4ds slack channel](#), which is an R for data science online learning community.

ChatGPT

ChatGPT is a language model developed by OpenAI. It is designed to generate human-like text based on the inputs it receives. ChatGPT is another tool you can consider using when you hit a coding roadblock or are looking for inspiration. However, you will want to ensure you test out the code because the language model is not perfect. Moreover, you should never feed ChatGPT any kind of sensitive information.

Ed Discussions

For this course, you have access to our own learning community via [Ed Discussions](#). We encourage you to try and answer your own questions first. When you reach that point of needing outside help, please do reach out to our Piazza chat. However, we want you to submit your questions in the form of a reprex, detailed below.

Stack Overflow is a popular website where people post coding questions and can receive help from the community. You’ll be able to see that the best answers you can vote for plus one on the best answer that you find helpful and then when thousands of other people are doing that, also you can clearly see which is the best answer.

GitHub is also a great resource for answering questions. It’s often where R package documentation and source code are stored and you can potentially request feedback and help directly from the developers or if you find a bug that you can’t resolve, you can create an issue at GitHub and also see what other issues people have run into.

There’s also Posit the company that develops and maintains RStudio, which is an open source data science company. They have built a community around supporting each other, other users and they have a fairly robust online forum that you can participate in and post questions too.

There’s also r4ds, is similar to the Posit RStudio community, but it’s a slack channel which you can ask to join and it’s full of R users that can be helpful.

You may also consider using ChatGPT, which is a language model developed by OpenAI. However, just make sure that you test out the code that you receive from ChatGPT because the language model is not perfect and Version 3.5 which I think is the current free version is sourcing code and content from 2021, so it might be a little out of date. Furthermore, you should never feed ChatGPT any kind of sensitive data or confidential information about others or yourself because you’re not sure where that’s going.

For this course, we also have Ed Discussions, which we may encourage you to use once you’ve exhausted your other options.

How to get help

When you reach that tipping point of hitting your head against the wall, these resources listed above and more can assist. However, to avoid drawing the ire of these online communities, it's important to provide a **reproducible example** (reprex or MRE for Minimal Reproducible Example) with your question. A reprex allows potential helpers to replicate your issue and offer solutions. Moreover, a reprex shows you have put in the effort to try and answer your question rather than asking people to take responsibility for your question: [“Don't ask to ask, just ask.”](#)

Often by going through the process of creating a reprex will reveal the bug to you. It's unclear whether this results from altering your perspective to explain your problem to others, taking a mental break, or something else. But for whatever reason, it often works.

Reprex

Let's go through what is needed in a reprex and how to create one.

Simple example

I'm subsetting a data frame using base R's brackets syntax, but I always forget if the order is [column, row] or [row, column]. Let's say I'm trying to filter a big dataset to include rows with an age greater than 17 and columns 1 and 5 through 12. My code may look like this:

```
confidential_data[age > 17, c(1, 5:12)]
```

For some reason, my code is throwing an error and I can't figure out why! I've exhausted my options and I'm now ready to reach out for help. Before I post my question, I need to prepare a reprex with the following pieces:

When you use the Ed channel or any of the other online communities, please make sure to create a reproducible example or a reprex. When you're asking somebody for help, often the best way for them to diagnose an issue is to try and replicate the error on their own systems and then investigate a fix.

Making that process as easy on them as possible increases your chances of getting a helpful response. Getting somebody to help you in getting a response. That's where creating a reproducible example comes in. You're basically isolating the problem or the error that you're encountering, setting up your environment, providing your data or similar data, ensuring that your code is otherwise error free and explaining your problem clearly. Then all they have to do is copy what you've created, replicating the issue in their environment and they can get work helping. They don't have to figure things out-side. Let's take a look at an example.

This is just a very simple example. I'm subsetting a data frame based on basically age. I want to pull out any age greater than 17 and I'm looking for Columns 1 and 5 through 12. My code may look something like this. This doesn't actually run, but this is just an example. For some reason, my code is throwing an error, and I can't figure out why. I looked at the help, I've searched Google to try and find a solution, but I haven't been able to find one. Before I post my question, let's say to Ed, I need to prepare a reprex with the following pieces.

1. **Isolate the problem:** Try to identify the minimal amount of code necessary to reproduce the issue. If you're working with a large codebase, you may need to create a simplified version that demonstrates the problem.
2. **Setup your environment:** Include all necessary libraries and their versions, your R version, etc so that anyone running your code will be able to do so without having to guess what is needed.
3. **Provide your data:** You need to provide data to work with. While you can share your data, you need to make sure it's not confidential and easy to access. Instead, consider using built-in R datasets (mtcars) or publicly available ones such as [palmerpenguins](#).
4. **Ensure your code is error-free:** Before you post your reprex, make sure to run it one more time in a new R session to confirm it's error-free and still produces the problem you are trying to solve.
5. **Explain your problem:** Provide a clear explanation of what you're trying to accomplish, what you've tried, what you expected to happen, and what actually happened.

I want to isolate the problem in order to allow somebody to reproduce that issue. If this is a small part of a large program, I may want to just create a simplified version in the simplified dataset that demonstrates the problem. I want to make sure that I've set up my environment so I'm including my R version in all necessary libraries that are needed to run that code.

I want to provide my data, although we'll talk in a little bit about how to do that, because certainly if you're working with confidential data, you wouldn't want to share that. But there are ways that you can share similar datasets that are public that would allow you to share something similar to what you're working with and that will allow you to diagnose the problem once somebody has posted a response.

Then make sure that your code is otherwise error free. So before you're posting your reprex, you want to make sure that whatever is causing the problem is really the only problem in the dataset and to fix everything else just so that anyone helping you doesn't run into basically the wrong problem. You want to isolate what you are asking about.

Then the last bit is to just provide a clear explanation of what you're trying to accomplish, what you've tried, what you're expecting, and then what actually happened, so what the error code or whatever is going wrong happened.

Here's what a reprex may look like for the above example, using a publicly available dataset with similar goals. You can see that I modified the column I'm filtering for rows and changed the number of columns I'm subsetting, yet still capturing the same issue.

```
# Description: I am trying to pull out rows with bill length greater than 40 and columns

# Setup
# install.packages(palmerpenguins) # if not already installed
library(palmerpenguins)

# code to create the subset
penguins[bill_length_mm > 40, c(1, 7:8)]

# Error in `[.tbl_df`(penguins, bill_length_mm > 40, c(1, 7:8)) :
# object 'bill_length_mm' not found
```

We can make this reprex easier to share by using `reprex::reprex()` to create an output to copy and paste. Since the reprex automatically captures the error, we can actually remove the error comment. Try it out in RStudio!

```
reprex::reprex({
# Description: I am trying to pull out rows with bill length greater than 40 and columns

# Setup
# install.packages(palmerpenguins) # if not already installed
library(palmerpenguins)

# code to create the subset
penguins[bill_length_mm > 40, c(1, 7:8)]
})
```

Here's an example of a reprex, what that might look like using a publicly available dataset. I've changed this around a little bit because we're using the palmerpenguins dataset which you can load. It's a publicly available dataset. Just noting that I am using the pacman package that we've started using this year to make it easier to load those libraries.

I will run this chunk here and note that this error comes up. I'm not able to subset on bill length millimeters greater than 40. I'm trying to pull Column 1 plus 7 through 8. It's a similar problem that I was encountering up here. But if this is confidential data, I've mimicked the problem, what I'm running into with this public dataset. If I post this and somebody says, here's the solution, I can then apply that solution to my confidential data.

There is a package called reprex, which allows you to basically create the code that you've been encountering the problem in, wrap that around this reprex function and it will generate a nice output that you can then post to either, or stack overflow, or wherever you are posting your question. If I run this, instead of actually running the code, it's wrapping that reprex function around it and generating this output.

As long as I post on either ed or Stack, Overflow that I'm running this and R, I can just copy this part. This provides the description of what's going on, loading all the packages that I need and then running the code that's causing the error plus the error that is causing and I could post that on Ed and then the person who has volunteered to help me come up with a solution can just open up a new R script and put this all in and run it and exactly replicate what the issue is that I'm seeing and then can get digging into finding out a solution. That's a handy way of creating a reprex without too much effort.

Now you may figure out your error as you're creating your reprex or someone on the forum you posted your reprex to helps you solve the issue. Turns out what was missing was the correct way to reference the `bill_length_mm` column! The correct syntax is: `penguins$bill_length_mm` or `confidential_data$age` in our "actual" data.

```
penguins[penguins$bill_length_mm > 40, c(1, 7:8)]
```

```
# A tibble: 244 × 3
  species sex    year
  <fct>   <fct> <int>
1 Adelie female 2007
2 <NA>    <NA>    NA
3 Adelie <NA>    2007
4 Adelie female 2007
5 Adelie male   2007
6 Adelie male   2007
7 Adelie male   2007
8 Adelie female 2007
9 Adelie male   2007
10 Adelie male   2007
# i 234 more rows
```

```
# confidential_data[confidential_data$age > 17, c(1, 5:12)]
```

Non-shareable data

In the above example we showed you how you may recreate your issue using publicly available data. Most of you will work with confidential, sensitive, or protected data during your careers. It's very tempting to just remove or mask columns that contain these sensitive data, but **that route is risky and strongly recommended against**. An accidental keystroke can lead to posting sensitive data where it does not belong with unpleasant and negative consequences.

The issue that this simple example was encountering was that I was trying to subset without actually calling the dataset of reference, so if I had posted that on Ed a fellow student might have said, all you got to do is add the data set and a dollar sign and you can get a successful program to run and sure enough, that works.

As mentioned above, when you're working with R, you're often going to be using data that you can't freely share. Whether you're working on a research project or in a public health department, you'll have confidential data that you're not really meant to share outside of that environment. Is certainly tempting to just remove or mask the columns that contain a sensitive data, but that can be fairly risky. We really strongly recommend against doing that. There's really not a lot that would prevent an accidental keystroke from posting that data. You might not realize that you're posting a bunch of confidential data until it's out there and too late.

Alternatives to public data

You can also create your own data to demonstrate your issue, such as with characters from your favorite TV shows.

```
sample_data <- data.frame(  
  first_name = c("Leslie", "Ron", "April", "Donna", "Greg"),  
  last_name = c("Knope", "Swanson", "Ludgate", "Meagle", "Pikitis"),  
  age = c(36, 42, 24, 26, 16),  
  occupation=c("Deputy Director", "Director", "Assistant", "Boss", "Student")  
)
```

sample_data

	first_name	last_name	age	occupation
1	Leslie	Knope	36	Deputy Director
2	Ron	Swanson	42	Director
3	April	Ludgate	24	Assistant
4	Donna	Meagle	26	Boss
5	Greg	Pikitis	16	Student

What we recommend is figuring out an alternative. One way of doing that would be instead of using patient names and birthdays or ages, what you could do is just replace the patient names with some names from your favorite TV show or book or something like that. Again, what you're trying to do is create something that you can share to get feedback on the problem that you're encountering. As long as it's similar enough, once somebody has volunteered to help and produced a fix, you can then apply that to your confidential dataset.

Shareable data

If your data is shareable, you may want to reduce the size of the data to make the reprex more digestible. For example, if your data exist online you can limit the number of rows.

```
ds_url <- "https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv"

readr::read_csv(ds_url, n_max = 200) # n_max allows you specify # of rows
```

Or if your data is not available online but is shareable, you can use a function called `dput()` to write code that creates your subset.

```
# creates code to rebuild my dataset
dput(penguins[1:10,])
```

```
structure(list(species = structure(c(1L, 1L, 1L, 1L, 1L, 1L,
1L, 1L, 1L, 1L), levels = c("Adelie", "Chinstrap", "Gentoo"), class = "factor"),
  island = structure(c(3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L), levels = c("Biscoe", "Dream", "Torgersen"), class = "factor"),
  bill_length_mm = c(39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9,
39.2, 34.1, 42), bill_depth_mm = c(18.7, 17.4, 18, NA, 19.3,
20.6, 17.8, 19.6, 18.1, 20.2), flipper_length_mm = c(181L,
186L, 195L, NA, 193L, 190L, 181L, 195L, 193L, 190L), body_mass_g = c(3750L,
3800L, 3250L, NA, 3450L, 3650L, 3625L, 4675L, 3475L, 4250L
), sex = structure(c(2L, 1L, 1L, NA, 1L, 2L, 1L, 2L, NA,
NA), levels = c("female", "male"), class = "factor"), year = c(2007L,
2007L, 2007L, 2007L, 2007L, 2007L, 2007L, 2007L, 2007L, 2007L
)), row.names = c(NA, -10L), class = c("tbl_df", "tbl", "data.frame"
))
```

Another tip that makes it easier for people to help you on a Stack Overflow is instead of sending an entire dataset that might be thousands and thousands of rows long, you can just create a dataset that it subsets maybe the first 200 rows.

Or if your data is not available online but it is sharable, you can use a function called `dput` to write code that creates your dataset. Say that I wanted to create a dataset for somebody to use that uses the first ten rows of the Penguins dataset, I can use this `dput` function, then just specify the first 10 rows and it will create this structure function and a list based on that dataset. So that I could take this bit of code here and put it into a completely unrelated R script file and that will generate, I need to actually put it in a data frame here. But that basically creates a data frame for me, replicates it in code, so that's a handy way of generating that. Again, only if you have data that is sharable.

I would then take this code output and use it to create my reprex. Now anyone with my reprex can run the code in their environment and replicate my error.

```
reprex::reprex({
# Description: I am trying to pull out rows with bill length greater than 40 and columns

# Setup
# install.packages(palmerpenguins) # if not already installed
penguins <- structure(list(species = structure(c(1L, 1L, 1L, 1L, 1L, 1L,
1L, 1L, 1L, 1L), levels = c("Adelie", "Chinstrap", "Gentoo"), class = "factor"),
island = structure(c(3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L, 3L),
3L), levels = c("Biscoe", "Dream", "Torgersen"), class = "factor"),
bill_length_mm = c(39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9,
39.2, 34.1, 42), bill_depth_mm = c(18.7, 17.4, 18, NA, 19.3,
20.6, 17.8, 19.6, 18.1, 20.2), flipper_length_mm = c(181L,
186L, 195L, NA, 193L, 190L, 181L, 195L, 193L, 190L), body_mass_g = c(3750L,
3800L, 3250L, NA, 3450L, 3650L, 3625L, 4675L, 3475L, 4250L
), sex = structure(c(2L, 1L, 1L, NA, 1L, 2L, 1L, 2L, NA,
NA), levels = c("female", "male"), class = "factor"), year = c(2007L,
2007L, 2007L, 2007L, 2007L, 2007L, 2007L, 2007L, 2007L
)), row.names = c(NA, -10L), class = c("tbl_df", "tbl", "data.frame"
))

# code to create the subset
penguins[bill_length_mm > 40, c(1, 7:8)]
})
```

How to create a good reprex

You can review sites like GitHub, Stack Overflow, and the RStudio community to see how other users produce their reprex and how they are received by other community members. Make sure your code is well organized and whether your code follows this [style guide](#).

Here are a few more resources on creating a good reprex.

- [The reprex package documentation](#)
- [A slide deck from Jenny Bryan](#)
- [Hadley Wickham's thoughts on reprex](#)
- [Stack Overflow - though not R specific](#)

Then when I'm creating Reprex, I would just put that Penguins dataset structure function within the reprex and then it will generate a good reprex for me. How to create a good Reprex. The sites that I've mentioned previously, GitHub, Stack, Overflow, and the RStudio or Posit communities. If you have spent any time on there, you have probably seen examples of users requesting that people posting, provide reprex.

There are also a lot of good examples of what people are looking for in reprex and how they create good ones. There are lots of resources that I've listed here. There's Hadley Wickham's thought on reprex. Jenny Bryan does a great slide deck on reprex and how that's useful.

There's also a Stack Overflow article about that though it's not our specific. We would encourage you to take a look at that if you're wanting to know how to create good reprexes. Remember creating a good reprex isn't just about solving your problem, it's about learning and contributing to the programming community. Not only by posting something or fixing this for yourself but that is an example that's going to stay on Stack Overflow or on the Posit community and other people who have your similar problem will be able to search and find the solution that somebody found for you.

You may also find that just by going through this process of creating a reprex for somebody else to help you with, you'll isolate the issue and you'll discover the source of the error on your own, so you won't actually need to post anything. I think as with anything with our practice, helps you improve.

Check out these additional resources and give it a try next time you run into an issue. We really encourage you to create a reference when you're posting on Ed. Please don't take screenshots. That really makes it difficult for anyone, students or instructors, to because it's really impossible for us to recreate your issue. Thank you.



reprex

Overview

Prepare reprexes for posting to [GitHub issues](#), [StackOverflow](#), in Slack [messages](#) or [snippets](#), or even to paste into PowerPoint or Keynote slides. What is a `reprex`? It's a **re**producible **e**xample, as coined by [Romain Francois](#).

Given R code on the clipboard, selected in RStudio, as an expression (quoted or not), or in a file ...

- run it via `rmarkdown::render()`,
- with deliberate choices re: `render()` arguments, knitr options, and Pandoc options.



Get resulting runnable code + output as

- Markdown, suitable for GitHub or Stack Overflow or Slack, or as
- R code, augmented with commented output, or as
- Plain HTML or (experimental) Rich Text

The result is returned invisibly, written to a file and, if possible, placed on the clipboard. Preview an HTML version in RStudio viewer or default browser.

Installation

Install from CRAN:

```
install.packages\("reprex"\)
```

On Linux, you probably want to install [xclip](#) or [xsel](#), so reprex can access the X11 clipboard. This is ‘nice to have’, but not mandatory. The usual `sudo apt-get install` or `sudo yum install` installation methods should work for both xclip and xsel.

Usage

Let’s say you copy this code onto your clipboard (or, on RStudio Server or Cloud, select it):

```
(y <- 1:4)
mean(y)
```

Then call `reprex()`, where the default target venue is GitHub:

`reprex()`

A nicely rendered HTML preview will display in RStudio’s Viewer (if you’re in RStudio) or your default browser otherwise.



The relevant bit of GitHub-flavored Markdown is ready to be pasted from your clipboard (on RStudio Server or Cloud, you will need to copy this yourself):

Here's what that Markdown would look like rendered in a GitHub issue:

```
(y <- 1:4)
#> [1] 1 2 3 4
mean(y)
#> [1] 2.5
```

Anyone else can copy, paste, and run this immediately.

In addition to GitHub, this Markdown also works on Stack Overflow and Discourse. Those venues can be formally requested via `venue = "so"` and `venue = "ds"`, but they are just aliases for `venue = "gh"`.

Instead of reading from the clipboard, you can:

- `reprex(mean(rnorm(10)))` to get code from expression.
- `reprex(input = "mean(rnorm(10))\n")` gets code from character vector (detected via length or terminating newline). Leading prompts are stripped from input source: `reprex(input = "> median(1:3)\n")` produces same output as `reprex(input = "median(1:3)\n")`
- `reprex(input = "my_reprex.R")` gets code from file
- Use one of the RStudio add-ins to use the selected text or current file.

But wait, there's more!

- Get slightly different Markdown, optimized for Slack messages, with `reprex(..., venue = "slack")`.
- Get a runnable R script, augmented with commented output, with `reprex(..., venue = "R")`. This is useful for Slack code snippets, email, etc.

- Prepare (un)rendered, syntax-highlighted code snippets to paste into Keynote or PowerPoint, with `reprex(..., venue = "rtf")`. This feature is still experimental; see the [associated article](#) for more.
- By default, figures are uploaded to [imgur.com](#) and the resulting URL is dropped into an inline image tag.
- If you really need to reprex in a specific directory, use the `wd` argument. For example, `reprex(wd = ".")` requests the current working directory.
- Append session info via `reprex(..., session_info = TRUE)`.
- Get clean, runnable code from wild-caught reprexes with
 - `reprex_invert()` = the opposite of `reprex()`
 - `reprex_clean()`, e.g. when you copy/paste from GitHub or Stack Overflow
 - `reprex_rescue()`, when you're dealing with copy/paste from R Console

LINKS

[View on CRAN](#)

[Browse source code](#)

[Report a bug](#)

LICENSE

[Full license](#)

[MIT](#) + file [LICENSE](#)

COMMUNITY

[Contributing guide](#)

9 Workflow: getting help

This book is not an island; there is no single resource that will allow you to master R. As you begin to apply the techniques described in this book to your own data, you will soon find questions that we do not answer. This section describes a few tips on how to get help and to help you keep learning.

9.1 Google is your friend

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. Additionally, adding package names like “tidyverse” or “ggplot2” will help narrow down the results to code that will feel more familiar to you as well, e.g., “how to make a boxplot in R” vs. “how to make a boxplot in R with ggplot2”. 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 [Stack Overflow](#). Start by spending a little time searching for an existing answer, including `[R]`, to restrict your search to questions and answers that use R.

9.2 Making a reprex

If your googling doesn’t find anything useful, it’s a really good idea to prepare a **reprex**, short for minimal **re**producible **e**xample. 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 two parts to creating a reprex:

- First, you need to make your code reproducible. This means that you need to capture everything, i.e. include any `library()` calls and create all necessary objects. The easiest way to make sure you’ve done this is using the reprex package.
- Second, you need to make it minimal. Strip away everything that is not directly related to your problem. This usually involves creating a much smaller and simpler R object than the one you’re facing in real life or even using built-in data.

That sounds like a lot of work! And it can be, but it has a great payoff:

- 80% of the time, creating an excellent reprex reveals the source of your problem. It’s amazing how often the process of writing up a self-contained and minimal example allows you to answer your own question.
- The other 20% of the time, you will have captured the essence of your problem in a way that is easy for others to play with. This substantially improves your chances of getting help!

When creating a reprex by hand, it’s easy to accidentally miss something, meaning your code can’t be run on

15 one else’s computer. Avoid this problem by using the reprex package, which is installed as part of the

tidyverse. Let's say you copy this code onto your clipboard (or, on RStudio Server or Cloud, select it):

```
y <- 1:4  
mean(y)
```

Then call `reprex()`, where the default output is formatted for GitHub:

```
reprex::reprex()
```

A nicely rendered HTML preview will display in RStudio's Viewer (if you're in RStudio) or your default browser otherwise. The reprex is automatically copied to your clipboard (on RStudio Server or Cloud, you will need to copy this yourself):

```
``` r  
y <- 1:4
mean(y)
#> [1] 2.5
```
```

This text is formatted in a special way, called Markdown, which can be pasted to sites like StackOverflow or Github and they will automatically render it to look like code. Here's what that Markdown would look like rendered on GitHub:

```
y <- 1:4  
mean(y)  
#> [1] 2.5
```

Anyone else can copy, paste, and run this immediately.

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; you may have discovered a bug that's been fixed since you installed or last updated the package. For packages in the tidyverse, the easiest way to check is to run `tidyverse_update()`.
2. The easiest way to include **data** is to use `dput()` to generate the R code needed to recreate it. For example, to recreate the `mtcars` dataset in R, perform the following steps:
 1. Run `dput(mtcars)` in R
 2. Copy the output
 3. In reprex, type `mtcars <-`, then paste.

Try to use 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:

- 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.

Creating reprexes is not trivial, and it will take some practice to learn to create good, truly minimal reprexes. However, learning to ask questions that include the code, and investing the time to make it reproducible will continue to pay off as you learn and master R.

9.3 Investing in yourself

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 the tidyverse team is doing on the [tidyverse blog](#). To keep up with the R community more broadly, we recommend reading [R Weekly](#): it's a community effort to aggregate the most interesting news in the R community each week.

9.4 Summary

This chapter concludes the Whole Game part of the book. You've now seen the most important parts of the data science process: visualization, transformation, tidying and importing. Now you've got a holistic view of the whole process, and we start to get into the details of small pieces.

The next part of the book, Visualize, does a deeper dive into the grammar of graphics and creating data visualizations with ggplot2, showcases how to use the tools you've learned so far to conduct exploratory data analysis, and introduces good practices for creating plots for communication.

Topic: Dates

Background

Dates are a special form of numeric that are stored as numeric but display in formats we're familiar with.

Like Microsoft (January 1st, 1900) and SAS (January 1st, 1960), R anchors to a specific date (January 1st, 1970).

Dates in R are stored as the **number of days** before or after 1/1/1970 and date/times are stored as the **number of seconds** before or after 1/1/1970.

Default display format in R is yyyy-mm-dd ('2020-09-17').

This video, we're going to talk about dates, and dates in R. Dates in R are a special form of a numeric. Dates are stored as a numeric value, but they're displayed in formats that we're more familiar.

With R similar to Microsoft, which uses origin date of January 1st, 1990, or SAS, which uses the origin date of January 1st, 1960, R anchors to a specific origin date, which is January 1st, 1970. Of course, they couldn't agree on one. But at least you can feel good that they all agreed to use January 1st.

Dates in R. There are two different types of dates that can be stored. There's date, which is just the day, or the date times, which is obviously date and time. For dates, those are stored as the number of days before or after January 1st, 1970, and for date times, they're stored as the number of seconds before or after the same date. The default display in R is a four-digit year and a dash, and a two-digit month and a dash and a two-digit day.

Dates using Base R

```
# Default format is yyyy-mm-dd - if you use that format, you don't have to specify
# You can assign a date to an object by using a date conversion function and pass
# Using base SAS function as.Date()
```

```
dob <- as.Date("2020-09-03")
dob
```

```
[1] "2020-09-03"
```

```
# Derive the number of days since 1/1/1970 by converting the date object to numeric
as.numeric(dob)
```

```
[1] 18508
```

```
#Confirming a different way
as.numeric(dob) - as.numeric(as.Date("1970-01-01"))
```

```
[1] 18508
```

```
# if you wanted to add 90 days to a date (or a column of dates):
as.Date("2020-09-04") + 90
```

```
[1] "2020-12-03"
```

```
# Converting strings to date or date/time

# If the date is stored in a different format, you just have to include a format
```

```
chr_dates <- c("03/05/1985", "09/21/1972")
dates <- as.Date(chr_dates, "%m/%d/%Y")
dates
```

```
[1] "1985-03-05" "1972-09-21"
```

```
# Another frequently used function is getting the current date
Sys.Date()
```

```
[1] "2024-06-21"
```

We are going to talk about both how to manipulate dates in base R, as well as using the lubridate package, which is part of the tidyverse. But we'll first talk about using date R. In order to create or assign an object a date data type, you can use in base R this function called `as.Date` and that'll basically take a character string and convert it into a date. We're going to do that for this `dob` here and double-check that it worked. Now you can see if we take a look at this. I think it will work.

If we do `str` on that, it'll show that it's a date format and show the format, which is the standard format. We can then see behind the curtain, if we use the `as.numeric` function to check out what the numeric value is being stored behind that. If we run that, we get 18,508, so that means that there are 18,508 days between September 3rd of 2020, and January 1st of 1970.

We can double-check that confirming a different way by subtracting the origin day. The `as.numeric` for the origin day is actually zero. You can confirm that. That's converting the anchor day to numeric, and that is zero because that's the starting day.

Once you've converted something into a date, you can add or subtract dates to those. In this situation, I'll convert that same date, but I'm going to add 90 days to it, and so that yields December 3rd, 2020 and then we can if we want to convert dates that come in a different format, so not in the standard R format already, all we have to do, we're going to do is store draw this as a vector.

Then if I want to convert that, all I have to do is add the format that I want R to use to parse. In this situation, these dates are the more American way of doing this, which is month- day- year. I would just specify the format month- day- year as the parsing format, and then R will convert those using my instructions and convert it into the format that it's most comfortable with.

Another frequently used date-related function is the `Sys.Date` which returns the system date. As long as your computer is configured correctly, it should be the date that we are in today. There are sometimes when a server is set up like the data server. Occasionally, when there's an update, it gets reset to GMT time and so the system date would be whatever it is in Greenwich Meridian Time. It's sometimes good to just double-check. That's correct.

Dates Using Lubridate

For working with dates in this course, we'll generally either use base R functions or the lubridate package functions from the tidyverse.



[lubridate reference page](#)

[lubridate cheat sheet](#)

```
# Using lubridate package functions (part of the tidyverse)

# Lubridate makes things a little easier in working with dates

dob <- as_date("2020-09-03")

# Using the mdy function easily converts the US date convention (or many other da
mdy("02/29/2020")
```

```
[1] "2020-02-29"
```

```
# if you wanted to add 90 days to a date (or a column of dates) it's basically th
as_date("2020-09-04") + 90
```

```
[1] "2020-12-03"
```

Now I'll talk about dates using the lubridate package. For most of what we do in this class, either days are or lubridate is perfectly fine. I prefer lubridate because it says it makes things a little easier in working with dates. The conversion function for lubridate is instead of as.Date, it's as_date and so it does the same thing. We now have dob as a day earlier. But only because it was specified a date earlier, not that there's anything about lubridate to do that.

You can also instead of including a format to parse by, what lubridate allows you to do is specify a function called mdy, and they have a number of different functions like this. But mdy is basically saying, I want to convert this month, day, year format to a date. It's a little bit easier that you don't have to include that parsing function and that successfully converted it to the right format. Similarly to base R, you can convert something to date format and then add 90 days.

```
# But if you need to add three months, lubridate makes it much easier
as_date("2020-09-04") + months(1)
```

```
[1] "2020-10-04"
```

```
# that works fine, but becomes a little problematic if the added month has fewer
as_date("2020-03-31") + months(1)
```

```
[1] NA
```

```
# the %m+% operator adds a month but reverts back to the last day of the month if
as_date("2020-03-31") %m+% months(1)
```

```
[1] "2020-04-30"
```

```
# On the off chance you need to convert from numeric:
lubridate::as_date(18239)
```

```
[1] "2019-12-09"
```

```
# You can also get current time (just have to be careful about time zone) using
with_tz(Sys.time(), tzone = "America/Los_Angeles")
```

```
[1] "2024-06-21 10:20:29 PDT"
```

```
# You can also use the today function and specify the time zone
today(tzone = "Australia/Sydney")
```

```
[1] "2024-06-22"
```

If you wanted to add one month, you can convert that and then add a month plus one. That month's function will add 30 days to that previous date. It can become a little problematic, especially if you're trying to add a month to a last day of a month.

In this case, April doesn't have 30 days, and so you're trying to add a month to March and April only has 30 days. It returns an NA. But one way around this is using this percent sign m plus percent sign operator, which adds a month but reverts back to the last day of the month if the expression yields a non-existent date.

In that scenario, I want to add a month, but if I use this operator instead of just the plus operator, then what we get is the last day of the next month is essentially what that's doing. If you ever have numeric values for dates, instead of like text value of date, you can convert that numeric value using the same as date function. If we do this, we will get 2019 December 9th. That can be useful, especially there are sometimes when you're importing data from Excel.

When you're importing data from Excel, it sometimes converts those dates into the numeric form of that. Then you just have to calculate the number of days between R's anchor date which is January 1st, 1970, and Microsoft's anchor date, which is January 1st, 1900, and then you'd be able to calculate the R date from there.

Then when you're talking about day times, you also have to be careful to include information about the time zone you're talking about. If I want to get the system time using the Pacific time zone, I need to specify that I'm asking for it in the Pacific time zone, and so it will give me that correct value.

You can also use this lubridate function called Today, which simplifies slightly the time zone that you're talking about. If I wanted to get today, right now in the time zone Australia/Sydney, that returns that's actually tomorrow in Sydney, Australia, so it gets that.

Wrangling with Dates

It makes sense to keep dates as numeric behind the scene, since you'll often want to be able to compare dates, identify number of days, weeks, months, and years between dates, and see if something happened early or later.

```
# Difference in days
Sys.Date() - as.Date("2020-09-03")
```

Time difference of 1387 days

```
# which date is earlier, later etc
Sys.Date() < as.Date("2020-09-03")
```

[1] FALSE

```
# another method of determining days between two dates, but also the option for c
as.numeric(difftime(Sys.Date(), as.Date("2020-09-03"), units = "days"))
```

[1] 1387

```
as.numeric(difftime(Sys.Date(), as.Date("2020-09-03"), units = "hours"))
```

[1] 33288

Talking a bit more about wrangling with dates. It does make sense to keep dates numeric behind the scenes. Again, we talked in a previous lecture about numerics in R being more efficient to process and more efficient to compare.

We've talked about, you can say, does this date happen before this other date and do conditional logic based on that. It's easier if that is happening as a numeric. But obviously, we want to look at something and not have to calculate what 18,754 means. We just want to see that date.

We can use subtraction as we've talked about, we can see the difference in days between the current system date and some specified date. I will run this and we can see the time difference of 1,332 days between today, which is April 26th, and September 3rd, 2020. Then we can also use comparison. If we want to know which of those is earlier, the system date is not earlier than this date because we're in future according to that date.

Another way of differentiating between two dates, we can specify the type of measurement that we want to use between those two dates. If we're looking at the difference between today and September 3rd of 2020, and we want to know the number of days, we can specify that in the function, and we get 1,332.

Or if we wanted hours, we could specify that instead and get 31,968. You can do years or weeks or minutes or seconds or whatever you need.


```
# If you need to extract a date part out of a date object:  
test <- as_date("1999-04-02")  
day(test)
```

```
[1] 2
```

```
month(test)
```

```
[1] 4
```

```
year(test)
```

```
[1] 1999
```

```
# day of the week  
wday(test, label = TRUE, abbr = TRUE)
```

```
[1] Fri
```

```
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

```
# day of the year (julian day)  
yday(test)
```

```
[1] 92
```

```
# key time measure for public health built into lubridate!  
epiweek(test)
```

```
[1] 13
```

You can also use functions to extract parts of a date. We're going to assign test this 1999, April 2nd date. If I want to pull the day out of that, I run the day function and I get the 2nd. But when I pull the month out of that, I use the month function and get the 4th.

Then if I want only the year, I would get 1999. Then you can also get day of the week for any date in history by using the weekday and so we now know that April 2nd 1999, was a Friday.

You can also get the Julian day, which is the day of the year so that is the 92nd day of the year. Specifically for us in Public Health, lubridate has a conversion to epi week. We know that the epi week for that date in 1999 was the the 13th week of the year.

You can also sort and join (next week) a dataset by dates more efficiently because it has numeric properties. As well as querying the most recent even in a dataset, grouped by people.

When you import dataset, most packages will make an attempt to guess that a column contains a date and automatically convert. But for the sake of learning how to manually manipulate, we'll use base R import

```
cdcr_nodt <- read.csv("https://raw.githubusercontent.com/datadesk/california-corc  
  
# note: readr::read_csv will guess the column data types, so you wouldn't need to  
str(cdcn_nodt)
```

```
'data.frame': 41370 obs. of 13 variables:  
 $ date      : chr  "2020-03-10" "2020-03-11" "2020-03-12" "2020-03-13" ...  
 $ code      : chr  "ASP" "ASP" "ASP" "ASP" ...  
 $ name      : chr  "Avenal State Prison" "Avenal State Prison" "Avenal State  
Prison" "Avenal State Prison" ...  
 $ city      : chr  "Avenal" "Avenal" "Avenal" "Avenal" ...  
 $ county    : chr  "Kings" "Kings" "Kings" "Kings" ...  
 $ fips      : int   31 31 31 31 31 31 31 31 31 31 ...  
 $ zipcode   : int   93204 93204 93204 93204 93204 93204 93204 93204 93204 ...  
 ...  
 $ x         : num  -120 -120 -120 -120 -120 ...  
 $ y         : num   36 36 36 36 36 ...  
 $ confirmed_cases : int   0 0 0 0 0 0 0 0 0 ...  
 $ new_confirmed_cases: int  NA 0 0 0 0 0 0 0 0 ...  
 $ deaths    : int   0 0 0 0 0 0 0 0 0 ...  
 $ new_deaths : int   NA 0 0 0 0 0 0 0 0 ...
```

You can also sort and join. We'll talk about join in a few weeks, joining data. But you can use dates to join as well as filtering or querying based on the most recent date in a dataset. We'll talk about that when we talk about aggregating data in a couple of weeks.

The other thing that you'll run into is when you import a dataset, most import packages like read CSV for Base or read CSV for Tidyverse, they'll make an attempt to guess at the column if the column contains a date and automatically convert that. But sometimes it gets it wrong, and so you need to be able to convert that manually.

We'll take a look at this. This is a dataset from CDCHA which is the California Department of Corrections Health Agency. If we take a look at that dataset, we can see that the date was brought in correctly. Oh, no, sorry, it was not. It was brought in as a character value. It's stored as character. Some programs might have converted that to date, but this one did not.

```
# In a dataset with dates as character, convert to date

cdc_nodt$date_mod <- ymd(cdc_nodt$date) #as_date would also work

# Getting the most recent date and earliest date
max(cdc_nodt$date_mod)
```

```
[1] "2023-06-04"
```

```
min(cdc_nodt$date_mod)
```

```
[1] "2020-03-10"
```

```
# and epi week

cdc_nodt$epiweek <- epiweek(cdc_nodt$date_mod)
cdc_nodt$year <- year(cdc_nodt$date_mod)

cdc_epiweek <- cdc_nodt |>
  filter(year == max(year) - 1) |>
  group_by(year, epiweek) |>
  summarize(cases = sum(confirmed_cases)) |>
  ungroup() |>
  as.data.frame()
```

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

```
str(cdc_epiweek)
```

```
'data.frame': 52 obs. of 3 variables:
 $ year : num 2022 2022 2022 2022 2022 ...
 $ epiweek: num 1 2 3 4 5 6 7 8 9 10 ...
 $ cases : int 375088 398210 427825 458349 479708 492936 500660 505432 508598 510824 ...
```

```
bind_rows(slice_head(cdc_epiweek, n=5),
          slice_tail(cdc_epiweek, n=5)) |>
  gt(auto_align = TRUE) |>
  tab_header(
    title = "CDCR Cases by Epi Week",
    subtitle = "First and Last 5 weeks displayed"
  )
```

Since it's a character, we now have to create a new column called date mod and convert that using the YMD function, and so we'll do that now. Then if we go back and look at this dataset, we can now see the date mod specified correctly as a date.

Now that we've done that, we could do something like calculate the epi week in the year. Because this is a multi year dataset, we can't just convert all of the dates to epi week. We also need the year because epi week 13 2020 is different than epi week 13 for 2021.

Then what I'm basically doing here is creating a dataset, I just really want to show you what we can do with epi week. This dataset contains cases by date, and I am then aggregating that to be able to look at cases by epi week in the highest year in this dataset. If you take a look at it is 2023, but it's only a partial year, I think data only goes through May.

If we wanted to look at the last full year, we would just do the max of the year minus 1. It's going to filter just year 2022. Then again, you'll get into this group by and summarize functions in a couple of weeks.

CDCR Cases by Epi Week

First and Last 5 weeks displayed

| year | epiwk | cases |
|------|-------|--------|
| 2022 | 1 | 375088 |
| 2022 | 2 | 398210 |
| 2022 | 3 | 427825 |
| 2022 | 4 | 458349 |
| 2022 | 5 | 479708 |
| 2022 | 48 | 616450 |
| 2022 | 49 | 619492 |
| 2022 | 50 | 623162 |
| 2022 | 51 | 626577 |

CDCR Cases by Epi Week

First and Last 5 weeks displayed

| year | epiwk | cases |
|------|-------|--------|
| 2022 | 52 | 682266 |

But this is just basically to show you what you can do with dates, and we'll take a look at what comes up and so basically we're looking at only the year of 2022, but we can see that this is the total number of cases for epi week 1. This is the total number of cases for epi week 2, etc. Great. Thank you for listening.

18 Dates and times

18.1 Introduction

This chapter will show you how to work with dates and times in R. At first glance, dates and times seem simple. You use them all the time in your regular life, and they don't seem to cause much confusion. However, the more you learn about dates and times, the more complicated they seem to get!

To warm up think about how many days there are in a year, and how many hours there are in a day. You probably remembered that most years have 365 days, but leap years have 366. Do you know the full rule for determining if a year is a leap year¹? The number of hours in a day is a little less obvious: most days have 24 hours, but in places that use daylight saving time (DST), one day each year has 23 hours and another has 25.

Dates and times are hard because they have to reconcile two physical phenomena (the rotation of the Earth and its orbit around the sun) with a whole raft of geopolitical phenomena including months, time zones, and DST. This chapter won't teach you every last detail about dates and times, but it will give you a solid grounding of practical skills that will help you with common data analysis challenges.

We'll begin by showing you how to create date-times from various inputs, and then once you've got a date-time, how you can extract components like year, month, and day. We'll then dive into the tricky topic of working with time spans, which come in a variety of flavors depending on what you're trying to do. We'll conclude with a brief discussion of the additional challenges posed by time zones.

18.1.1 Prerequisites

This chapter will focus on the **lubridate** package, which makes it easier to work with dates and times in R. As of the latest tidyverse release, lubridate is part of core tidyverse. We will also need nycflights13 for practice data.

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

18.2 Creating date/times

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 `<dtm>`. Base R calls these POSIXct, but doesn't exactly trip off the tongue.

In this chapter we are going to focus on dates and date-times as R doesn't have a native class for storing times. If you need one, you can use the **hms** package.

You should always use the simplest possible data type that works for your needs. That means if you can use a date instead of a date-time, you should. Date-times are substantially more complicated because of the need to handle time zones, which we'll come back to at the end of the chapter.

To get the current date or date-time you can use `today()` or `now()`:

```
today()
#> [1] "2023-09-05"
now()
#> [1] "2023-09-05 23:09:46 UTC"
```

Otherwise, the following sections describe the four ways you're likely to create a date/time:

- While reading a file with `readr`.
- From a string.
- From individual date-time components.
- From an existing date/time object.

18.2.1 During import

If your CSV contains an ISO8601 date or date-time, you don't need to do anything; `readr` will automatically recognize it:

```
csv <- "
  date,datetime
  2022-01-02,2022-01-02 05:12
"
read_csv(csv)
#> # A tibble: 1 × 2
#>   date      datetime
#>   <date>    <dtm>
#> 1 2022-01-02 2022-01-02 05:12:00
```

If you haven't heard of **ISO8601** before, it's an international standard² for writing dates where the components of a date are organized from biggest to smallest separated by `-`. For example, in ISO8601 May 3 2022 is `2022-05-03`. ISO8601 dates can also include times, where hour, minute, and second are separated by `:`, and the date and time components are separated by either a `T` or a space. For example, you could write 4:26pm on May 3 2022 as either `2022-05-03 16:26` or `2022-05-03T16:26`.

For other date-time formats, you'll need to use `col_types` plus `col_date()` or `col_datetime()` along with a date-time format. The date-time format used by `readr` is a standard used across many programming languages, describing a date component with a `%` followed by a single character. For example, `%Y-%m-%d` specifies a date that's a year, `-`, month (as number) `-`, day. Table [Table 18.1](#) lists all the options.

Table 18.1: All date formats understood by readr

| Type | Code | Meaning | Example |
|-------|------|--------------------------------|-----------------|
| Year | %Y | 4 digit year | 2021 |
| | %y | 2 digit year | 21 |
| Month | %m | Number | 2 |
| | %b | Abbreviated name | Feb |
| | %B | Full name | February |
| Day | %d | Two digits | 02 |
| | %e | One or two digits | 2 |
| | %e | One or two digits | 2 |
| Time | %H | 24-hour hour | 13 |
| | %I | 12-hour hour | 1 |
| | %p | AM/PM | pm |
| | %M | Minutes | 35 |
| | %S | Seconds | 45 |
| | %OS | Seconds with decimal component | 45.35 |
| | %Z | Time zone name | America/Chicago |
| Other | %z | Offset from UTC | +0800 |
| | %. | Skip one non-digit | : |
| | %* | Skip any number of non-digits | |

And this code shows a few options applied to a very ambiguous date:

```
csv <- "
  date
  01/02/15
"

read_csv(csv, col_types = cols(date = col_date("%m/%d/%y")))
#> # A tibble: 1 × 1
#>   date
#>   <date>
#> 1 2015-01-02
```



```

read_csv(csv, col_types = cols(date = col_date("%d/%m/%y")))
#> # A tibble: 1 × 1
#>   date
#>   <date>
#> 1 2015-02-01

read_csv(csv, col_types = cols(date = col_date("%y/%m/%d")))
#> # A tibble: 1 × 1
#>   date
#>   <date>
#> 1 2001-02-15

```

Note that no matter how you specify the date format, it's always displayed the same way once you get it into R.

If you're using %b or %B and working with non-English dates, you'll also need to provide a `locale()`. See the list of built-in languages in `date_names_langs()`, or create your own with `date_names()`,

18.2.2 From strings

The date-time specification language is powerful, but requires careful analysis of the date format. An alternative approach is to use lubridate's helpers which attempt to automatically determine 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. For example:

```

ymd("2017-01-31")
#> [1] "2017-01-31"
mdy("January 31st, 2017")
#> [1] "2017-01-31"
dmy("31-Jan-2017")
#> [1] "2017-01-31"

```

`ymd()` and friends create dates. To create a date-time, add an underscore and one or more of "h", "m", and "s" to the name of the parsing function:

```

ymd_hms("2017-01-31 20:11:59")
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
#> [1] "2017-01-31 08:01:00 UTC"

```

You can also force the creation of a date-time from a date by supplying a timezone:

```

ymd("2017-01-31", tz = "UTC")
#> [1] "2017-01-31 UTC"

```

Here I use the UTC³ timezone which you might also know as GMT, or Greenwich Mean Time, the time at 0° longitude⁴. It doesn't use daylight saving time, making it a bit easier to compute with.

18.2.3 From individual components

Instead of a single string, sometimes you'll have the individual components of the date-time spread across multiple columns. This is what we have in the `flights` data:

```
flights |>
  select(year, month, day, hour, minute)
#> # A tibble: 336,776 × 5
#>   year month   day hour minute
#>   <int> <int> <int> <dbl> <dbl>
#> 1  2013     1     1     5     15
#> 2  2013     1     1     5     29
#> 3  2013     1     1     5     40
#> 4  2013     1     1     5     45
#> 5  2013     1     1     6      0
#> 6  2013     1     1     5     58
#> # i 336,770 more rows
```

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> <dtm>
#> 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
#> # i 336,770 more rows
```

Let's do the same thing for each of the four time columns in `flights`. The times are represented in a slightly odd format, so we use modulus arithmetic to pull out the hour and minute components. Once we've created the date-time variables, we focus in on the variables we'll explore in the rest of the chapter.

```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

flights_dt <- flights |>
  filter(!is.na(dep_time), !is.na(arr_time)) |>
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(year, month, day, sched_dep_time),
```

```

    sched_arr_time = make_datetime_100(year, month, day, sched_arr_time)
  ) |>
  select(origin, dest, ends_with("delay"), ends_with("time"))

flights_dt
#> # A tibble: 328,063 × 9
#>   origin dest dep_delay arr_delay dep_time          sched_dep_time
#>   <chr>  <chr>    <dbl>    <dbl> <dtm>          <dtm>
#> 1 EWR    IAH         2        11 2013-01-01 05:17:00 2013-01-01 05:15:00
#> 2 LGA    IAH         4        20 2013-01-01 05:33:00 2013-01-01 05:29:00
#> 3 JFK    MIA         2        33 2013-01-01 05:42:00 2013-01-01 05:40:00
#> 4 JFK    BQN        -1       -18 2013-01-01 05:44:00 2013-01-01 05:45:00
#> 5 LGA    ATL        -6       -25 2013-01-01 05:54:00 2013-01-01 06:00:00
#> 6 EWR    ORD        -4        12 2013-01-01 05:54:00 2013-01-01 05:58:00
#> # i 328,057 more rows
#> # i 3 more variables: arr_time <dtm>, sched_arr_time <dtm>, ...

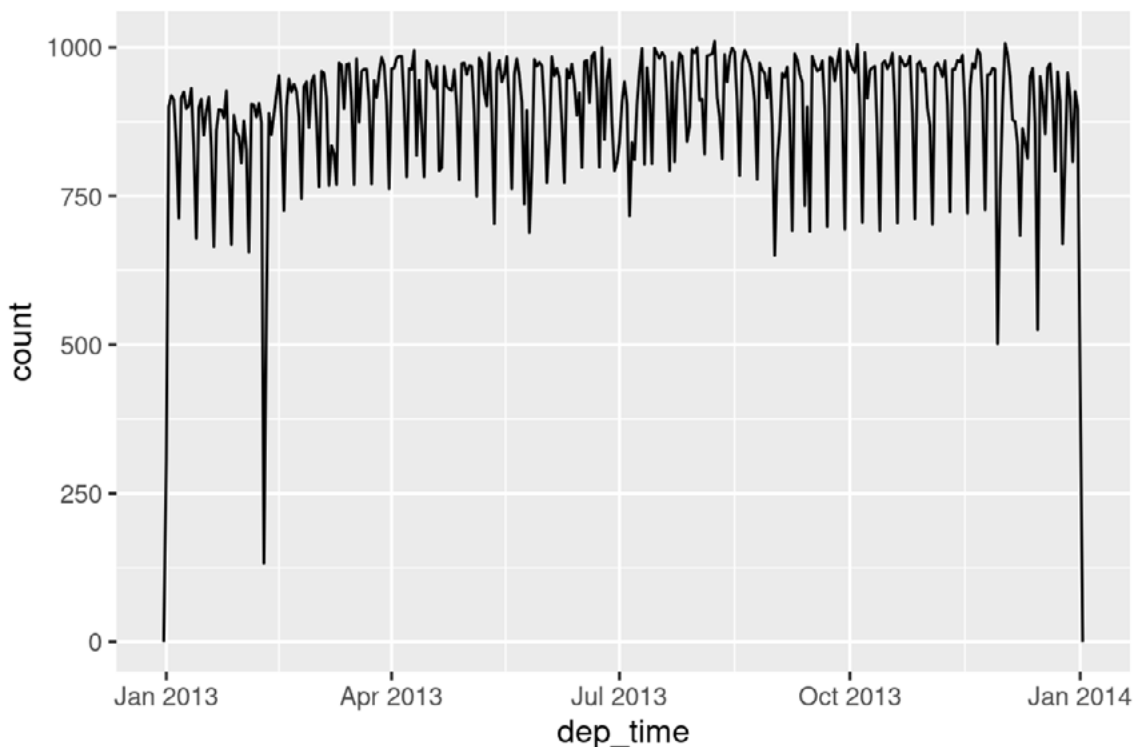
```

With this data, we can visualize the distribution of departure times across the year:

```

flights_dt |>
  ggplot(aes(x = dep_time)) +
  geom_freqpoly(binwidth = 86400) # 86400 seconds = 1 day

```

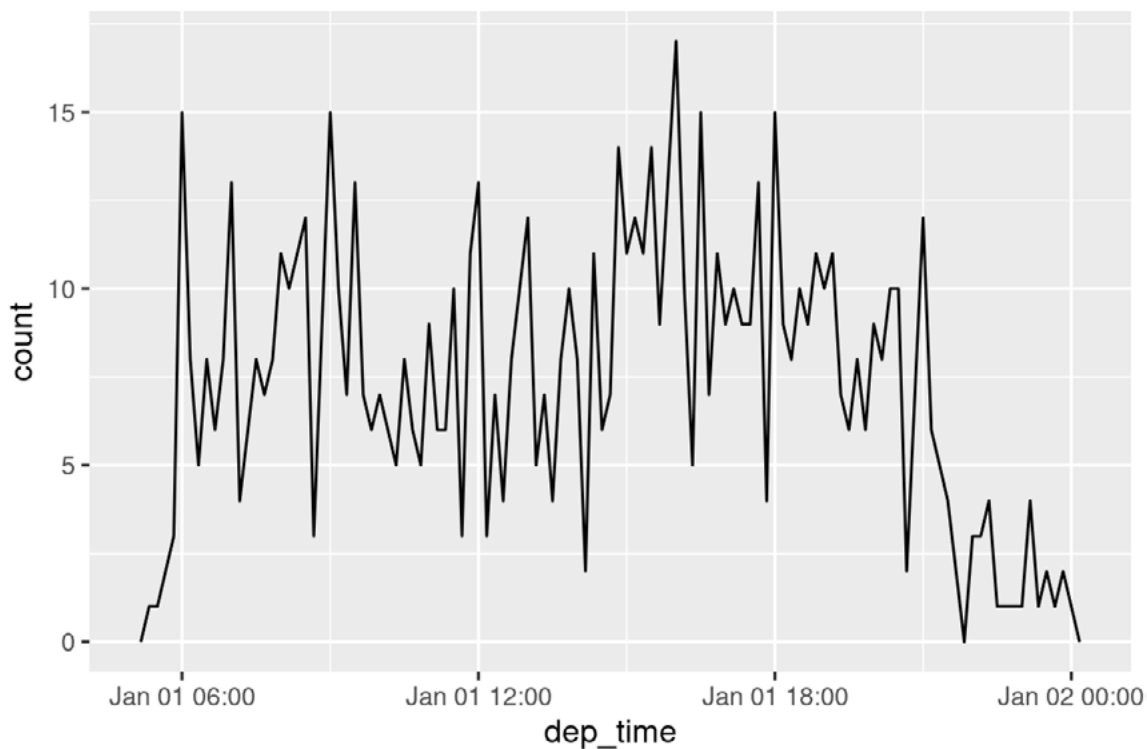


Or within a single day:

```

flights_dt |>
  filter(dep_time < ymd(20130102)) |>
  ggplot(aes(x = dep_time)) +
  geom_freqpoly(binwidth = 600) # 600 s = 10 minutes

```



Note that when you use date-times in a numeric context (like in a histogram), 1 means 1 second, so a binwidth of 86400 means one day. For dates, 1 means 1 day.

18.2.4 From other types

You may want to switch between a date-time and a date. That's the job of `as_datetime()` and `as_date()`:

```
as_datetime(today())
#> [1] "2023-09-05 UTC"
as_date(now())
#> [1] "2023-09-05"
```

Sometimes you'll get date/times as numeric offsets from the "Unix Epoch", 1970-01-01. If the offset is in seconds, use `as_datetime()`; if it's in days, use `as_date()`.

```
as_datetime(60 * 60 * 10)
#> [1] "1970-01-01 10:00:00 UTC"
as_date(365 * 10 + 2)
#> [1] "1980-01-01"
```

18.2.5 Exercises

1. What happens if you parse a string that contains invalid dates?

```
ymd(c("2010-10-10", "bananas"))
```

2. What does the `tz` argument to `today()` do? Why is it important?

3. For each of the following date-times, show how you'd parse it using a readr column specification and a lubridate function.

```
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"
```

18.3 Date-time components

Now that you know how to get date-time data into R's date-time data structures, let's explore what you can do with them. This section will focus on the accessor functions that let you get and set individual components. The next section will look at how arithmetic works with date-times.

18.3.1 Getting components

You can pull out individual parts of the date with the accessor functions `year()`, `month()`, `mday()` (day of the month), `yday()` (day of the year), `wday()` (day of the week), `hour()`, `minute()`, and `second()`. These are effectively the opposites of `make_datetime()`.

```
datetime <- ymd_hms("2026-07-08 12:34:56")

year(datetime)
#> [1] 2026
month(datetime)
#> [1] 7
mday(datetime)
#> [1] 8

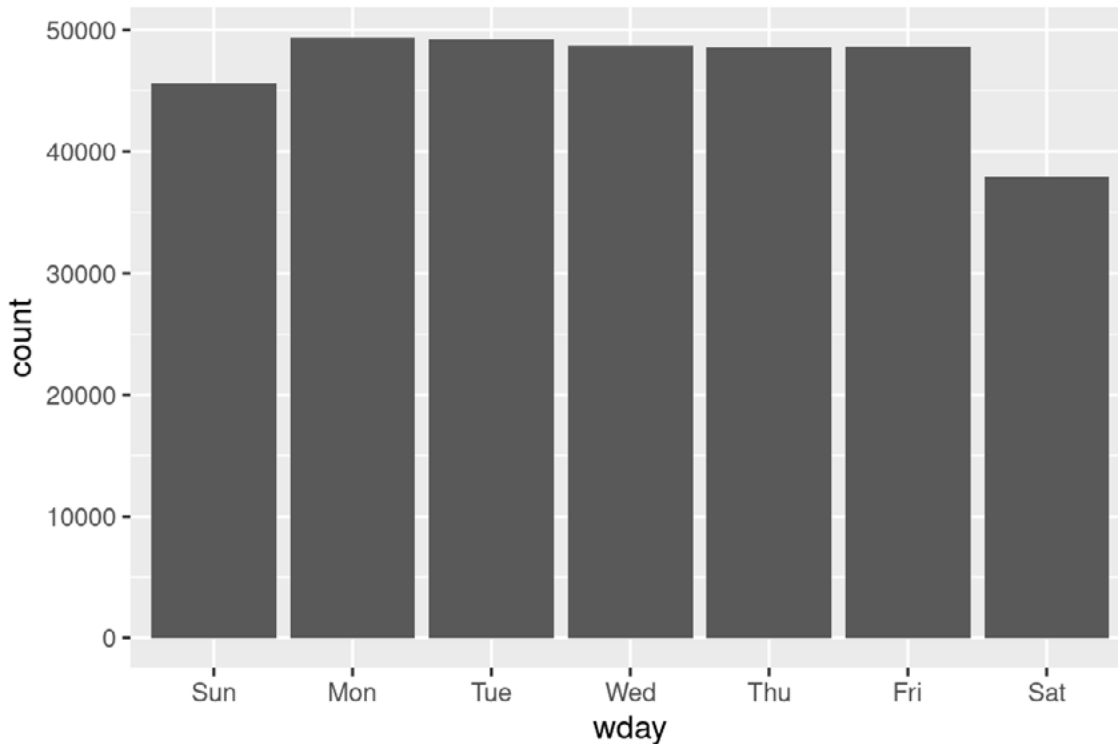
yday(datetime)
#> [1] 189
wday(datetime)
#> [1] 4
```

For `month()` and `wday()` you can set `label = TRUE` to return the abbreviated name of the month or day of the week. Set `abbr = FALSE` to return the full name.

```
month(datetime, label = TRUE)
#> [1] Jul
#> 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
wday(datetime, label = TRUE, abbr = FALSE)
#> [1] Wednesday
#> 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

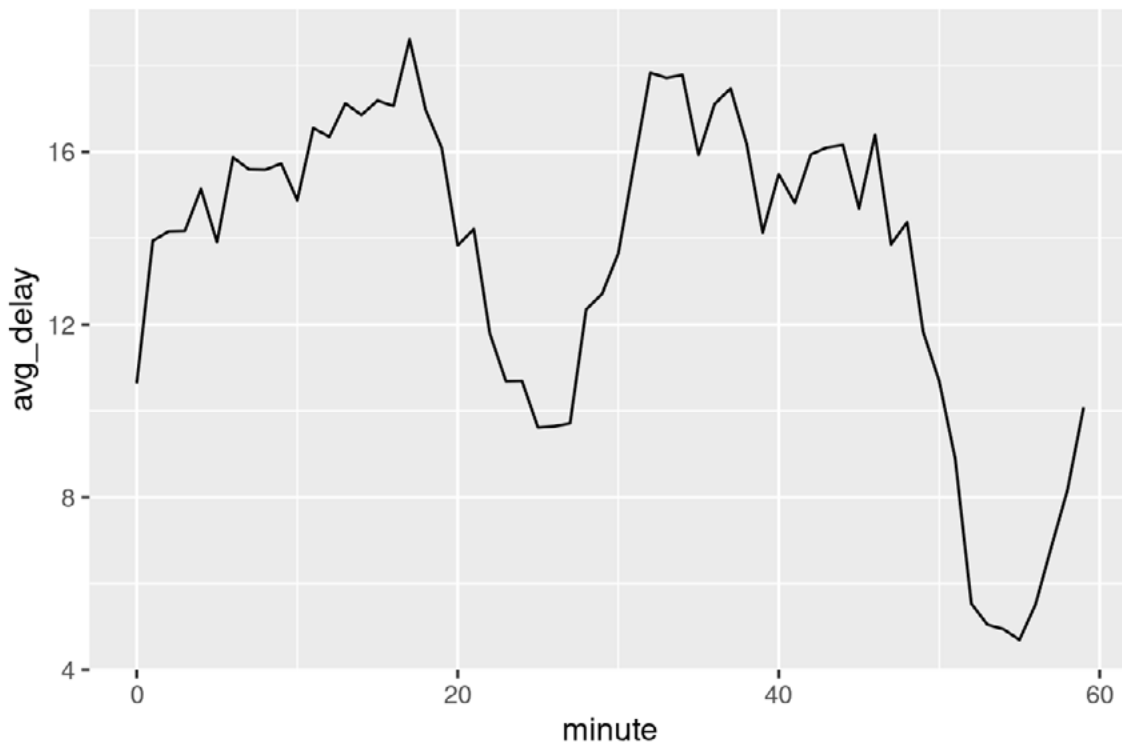
We can use `wday()` to see that more flights depart during the week than on the weekend:

```
flights_dt |>
  mutate(wday = wday(dep_time, label = TRUE)) |>
  ggplot(aes(x = wday)) +
  geom_bar()
```



We can also look at the average departure delay by minute within the hour. There's an interesting pattern: flights leaving in minutes 20-30 and 50-60 have much lower delays than the rest of the hour!

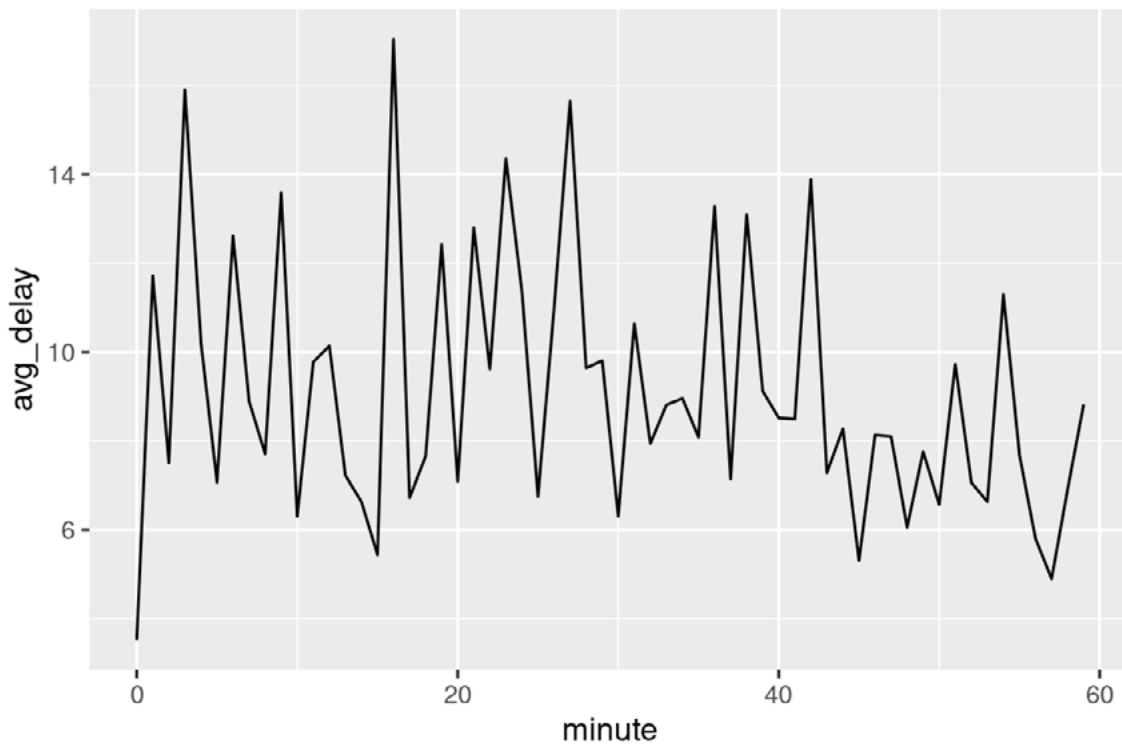
```
flights_dt |>
  mutate(minute = minute(dep_time)) |>
  group_by(minute) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  ) |>
  ggplot(aes(x = minute, y = avg_delay)) +
  geom_line()
```



Interestingly, if we look at the *scheduled* departure time we don't see such a strong pattern:

```
sched_dep <- flights_dt |>
  mutate(minute = minute(sched_dep_time)) |>
  group_by(minute) |>
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

ggplot(sched_dep, aes(x = minute, y = avg_delay)) +
  geom_line()
```

So why do we see that pattern with the actual departure times? Well, like much data collected by humans, there's a strong bias towards flights leaving at "nice" departure times, as [Figure 18.1](#) shows. Always be alert for this sort of pattern whenever you work with data that involves human judgement!

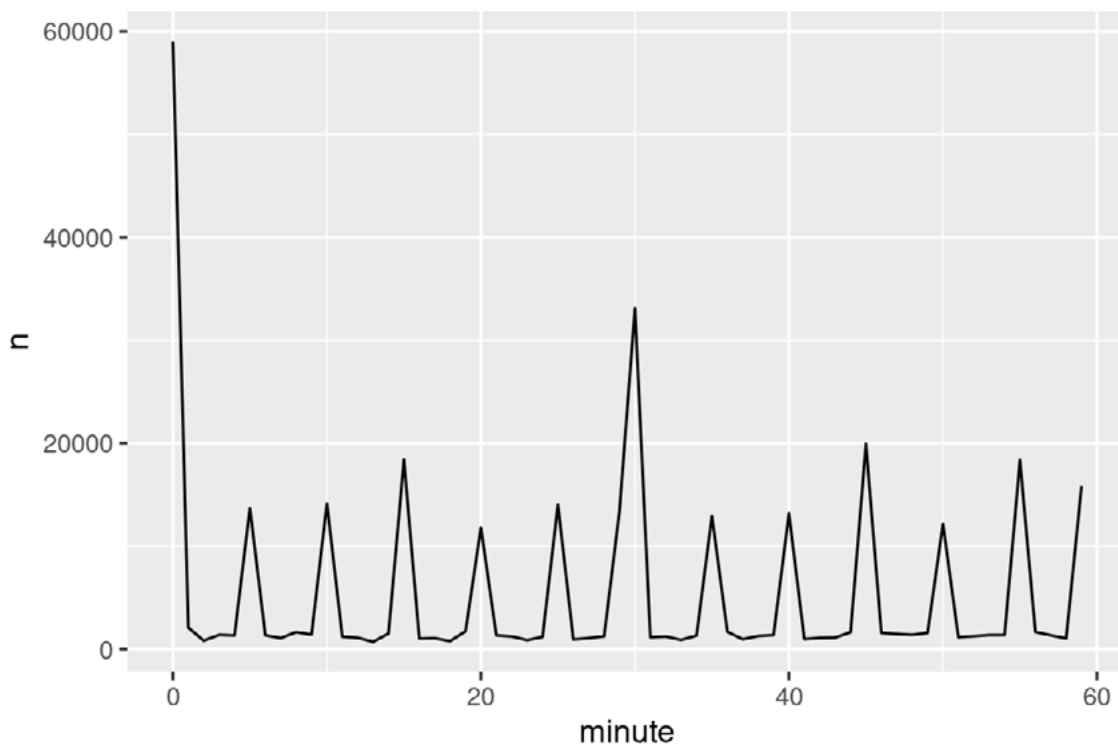
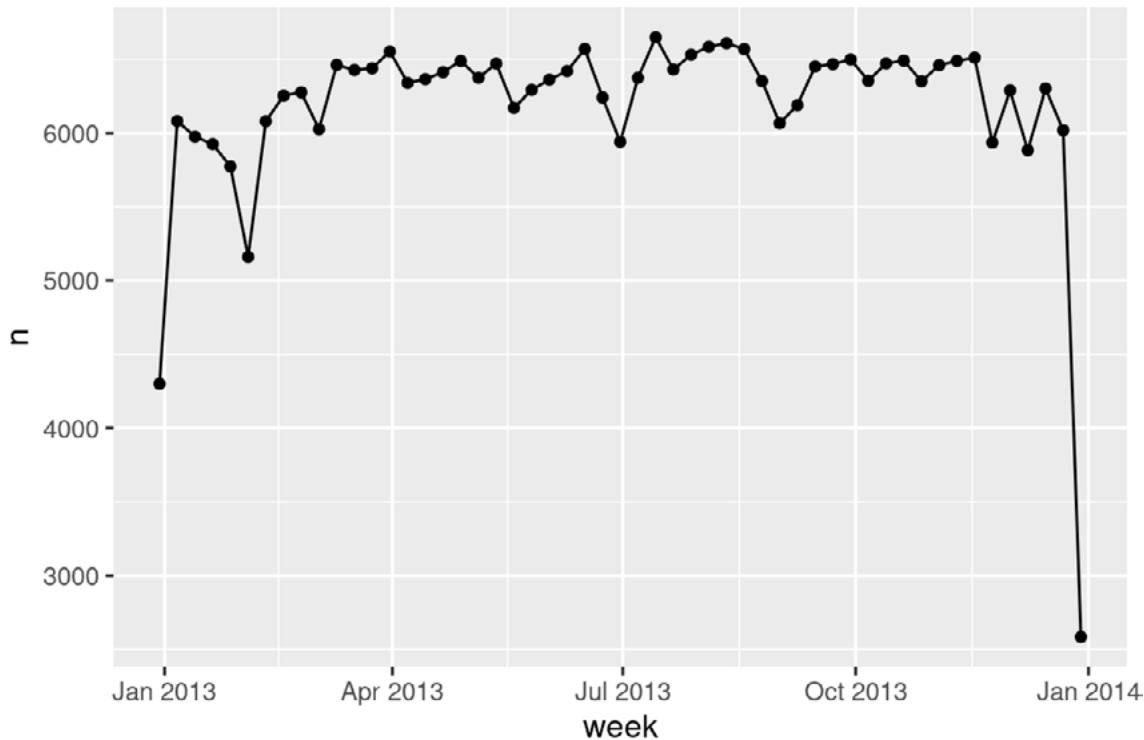


Figure 18.1: A frequency polygon showing the number of flights scheduled to depart each hour. You can see a strong preference for round numbers like 0 and 30 and generally for numbers that are a multiple of five.

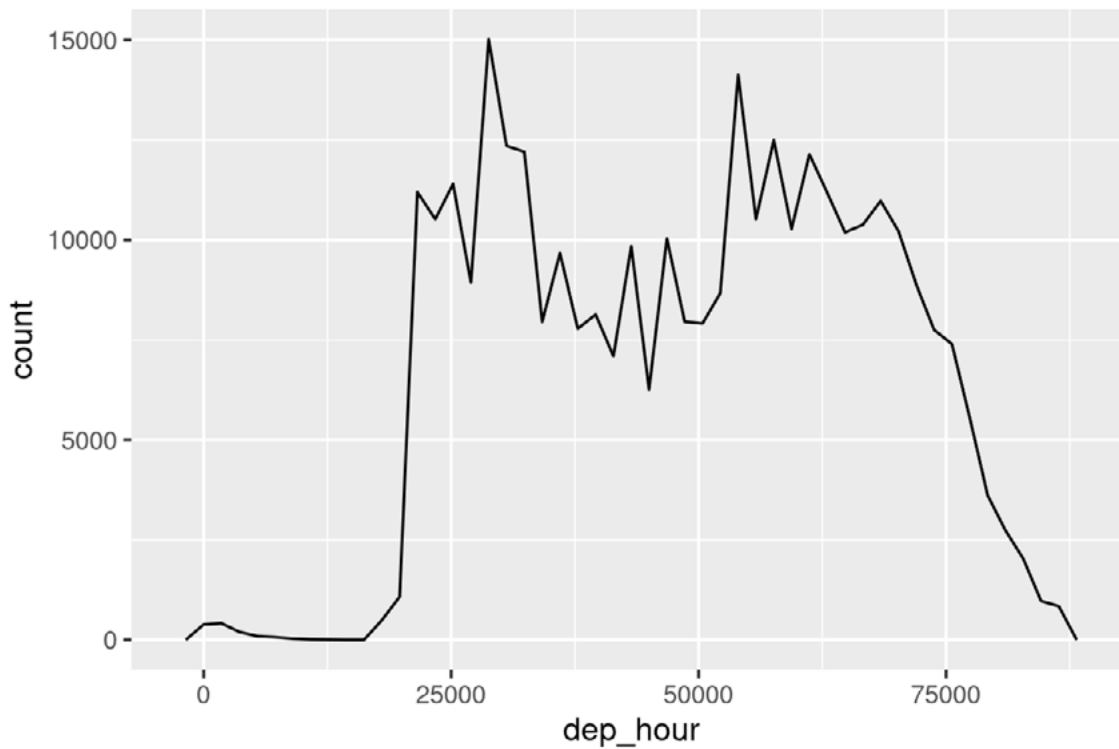
An alternative approach to plotting individual components is to round the date to a nearby unit of time, with `floor_date()`, `round_date()`, and `ceiling_date()`. Each function takes a vector of dates to adjust and then the name of the unit to round down (floor), round up (ceiling), or round to. This, for example, allows us to plot the number of flights per week:

```
flights_dt |>
  count(week = floor_date(dep_time, "week")) |>
  ggplot(aes(x = week, y = n)) +
  geom_line() +
  geom_point()
```



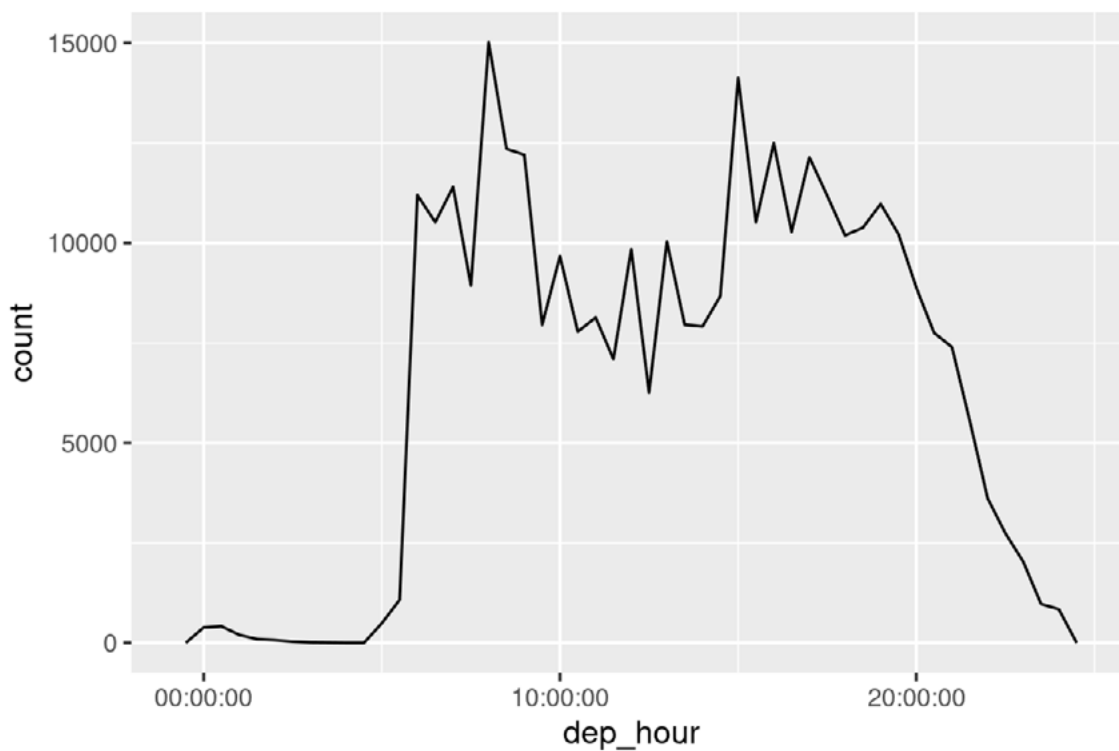
You can use rounding to show the distribution of flights across the course of a day by computing the difference between `dep_time` and the earliest instant of that day:

```
flights_dt |>
  mutate(dep_hour = dep_time - floor_date(dep_time, "day")) |>
  ggplot(aes(x = dep_hour)) +
  geom_freqpoly(binwidth = 60 * 30)
#> Don't know how to automatically pick scale for object of type <difftime>.
#> Defaulting to continuous.
```



Computing the difference between a pair of date-times yields a `difftime` (more on that in [Section 18.4.3](#)). We can convert that to an `hms` object to get a more useful x-axis:

```
flights_dt |>
  mutate(dep_hour = hms::as_hms(dep_time - floor_date(dep_time, "day"))) |>
  ggplot(aes(x = dep_hour)) +
  geom_freqpoly(binwidth = 60 * 30)
```



18.3.3 Modifying components

You can also use each accessor function to modify the components of a date/time. This doesn't come up much in data analysis, but can be useful when cleaning data that has clearly incorrect dates.

```
(datetime <- ymd_hms("2026-07-08 12:34:56"))
#> [1] "2026-07-08 12:34:56 UTC"

year(datetime) <- 2030
datetime
#> [1] "2030-07-08 12:34:56 UTC"
month(datetime) <- 01
datetime
#> [1] "2030-01-08 12:34:56 UTC"
hour(datetime) <- hour(datetime) + 1
datetime
#> [1] "2030-01-08 13:34:56 UTC"
```

Alternatively, rather than modifying an existing variable, you can create a new date-time with `update()`. This also allows you to set multiple values in one step:

```
update(datetime, year = 2030, month = 2, mday = 2, hour = 2)
#> [1] "2030-02-02 02:34:56 UTC"
```

If values are too big, they will roll-over:

```
update(ymd("2023-02-01"), mday = 30)
#> [1] "2023-03-02"
update(ymd("2023-02-01"), hour = 400)
#> [1] "2023-02-17 16:00:00 UTC"
```

18.3.4 Exercises

1. How does the distribution of flight times within a day change over the course of the year?
2. Compare `dep_time`, `sched_dep_time` and `dep_delay`. Are they consistent? Explain your findings.
3. Compare `air_time` with the duration between the departure and arrival. Explain your findings. (Hint: consider the location of the airport.)
4. How does the average delay time change over the course of a day? Should you use `dep_time` or `sched_dep_time`? Why?
5. On what day of the week should you leave if you want to minimise the chance of a delay?
6. What makes the distribution of `diamonds$carat` and `flights$sched_dep_time` similar?

7. Confirm our hypothesis that the early departures of flights in minutes 20-30 and 50-60 are caused by scheduled flights that leave early. Hint: create a binary variable that tells you whether or not a flight was delayed.

18.4 Time spans

Next you'll learn about how arithmetic with dates works, including subtraction, addition, and division. Along the way, you'll learn about three important classes that represent time spans:

- **Durations**, which represent an exact number of seconds.
- **Periods**, which represent human units like weeks and months.
- **Intervals**, which represent a starting and ending point.

How do you pick between duration, periods, and intervals? As always, pick the simplest data structure that solves your problem. If you only care about physical time, use a duration; if you need to add human times, use a period; if you need to figure out how long a span is in human units, use an interval.

18.4.1 Durations

In R, when you subtract two dates, you get a `difftime` object:

```
# How old is Hadley?  
h_age <- today() - ymd("1979-10-14")  
h_age  
#> Time difference of 16032 days
```

A `difftime` class object records a time span of seconds, minutes, hours, days, or weeks. This ambiguity can make `diff`times a little painful to work with, so `lubridate` provides an alternative which always uses seconds: the **duration**.

```
as.duration(h_age)  
#> [1] "1385164800s (~43.89 years)"
```

Durations come with a bunch of convenient constructors:

```
dseconds(15)  
#> [1] "15s"  
dminutes(10)  
#> [1] "600s (~10 minutes)"  
dhours(c(12, 24))  
#> [1] "43200s (~12 hours)" "86400s (~1 days)"  
ddays(0:5)  
#> [1] "0s" "86400s (~1 days)" "172800s (~2 days)"  
#> [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"  
dweeks(3)  
#> [1] "1814400s (~3 weeks)"
```

```
dyears(1)
#> [1] "31557600s (~1 years)"
```

Durations always record the time span in seconds. Larger units are created by converting minutes, hours, days, weeks, and years to seconds: 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 7 days in a week. Larger time units are more problematic. A year uses the “average” number of days in a year, i.e. 365.25. There’s no way to convert a month to a duration, because there’s just too much variation.

You can add and multiply durations:

```
2 * dyears(1)
#> [1] "63115200s (~2 years)"
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38869200s (~1.23 years)"
```

You can add and subtract durations to and from days:

```
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)
```

However, because durations represent an exact number of seconds, sometimes you might get an unexpected result:

```
one_am <- ymd_hms("2026-03-08 01:00:00", tz = "America/New_York")

one_am
#> [1] "2026-03-08 01:00:00 EST"
one_am + ddays(1)
#> [1] "2026-03-09 02:00:00 EDT"
```

Why is one day after 1am March 8, 2am March 9? If you look carefully at the date you might also notice that the time zones have changed. March 8 only has 23 hours because it’s when DST starts, so if we add a full days worth of seconds we end up with a different time.

18.4.2 Periods

To solve this problem, lubridate provides **periods**. Periods are time spans but don’t have a fixed length in seconds, instead they work with “human” times, like days and months. That allows them to work in a more intuitive way:

```
one_am
#> [1] "2026-03-08 01:00:00 EST"
one_am + days(1)
#> [1] "2026-03-09 01:00:00 EDT"
```

Like durations, periods can be created with a number of friendly constructor functions.

```
hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
```

```

days(7)
#> [1] "7d 0H 0M 0S"
months(1:6)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
#> [5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"

```

You can add and multiply periods:

```

10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"

```

And of course, add them to dates. Compared to durations, periods are more likely to do what you expect:

```

# A leap year
ymd("2024-01-01") + dyears(1)
#> [1] "2024-12-31 06:00:00 UTC"
ymd("2024-01-01") + years(1)
#> [1] "2025-01-01"

# Daylight saving time
one_am + ddays(1)
#> [1] "2026-03-09 02:00:00 EDT"
one_am + days(1)
#> [1] "2026-03-09 01:00:00 EDT"

```

Let's use periods to fix an oddity related to our flight dates. Some planes appear to have arrived at their destination *before* they departed from New York City.

```

flights_dt |>
  filter(arr_time < dep_time)
#> # A tibble: 10,633 × 9
#>   origin dest dep_delay arr_delay dep_time sched_dep_time
#>   <chr>  <chr>    <dbl>    <dbl> <dtm>          <dtm>
#> 1 EWR    BQN         9        -4 2013-01-01 19:29:00 2013-01-01 19:20:00
#> 2 JFK    DFW        59         NA 2013-01-01 19:39:00 2013-01-01 18:40:00
#> 3 EWR    TPA        -2         9 2013-01-01 20:58:00 2013-01-01 21:00:00
#> 4 EWR    SJU        -6       -12 2013-01-01 21:02:00 2013-01-01 21:08:00
#> 5 EWR    SFO        11       -14 2013-01-01 21:08:00 2013-01-01 20:57:00
#> 6 LGA    FLL       -10        -2 2013-01-01 21:20:00 2013-01-01 21:30:00
#> # i 10,627 more rows
#> # i 3 more variables: arr_time <dtm>, sched_arr_time <dtm>, ...

```

These are overnight flights. We used the same date information for both the departure and the arrival times, but these flights arrived on the following day. We can fix this by adding `days(1)` to the arrival time of each overnight flight.

```
flights_dt <- flights_dt |>
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight),
    sched_arr_time = sched_arr_time + days(overnight)
  )
```

Now all of our flights obey the laws of physics.

```
flights_dt |>
  filter(arr_time < dep_time)
#> # A tibble: 0 × 10
#> #   10 variables: origin <chr>, dest <chr>, dep_delay <dbl>,
#> #   arr_delay <dbl>, dep_time <dtm>, sched_dep_time <dtm>, ...
```

18.4.3 Intervals

What does `dyears(1) / ddays(365)` return? It's not quite one, because `dyears()` is defined as the number of seconds per average year, which is 365.25 days.

What does `years(1) / days(1)` return? Well, if the year was 2015 it should return 365, but if it was 2016, it should return 366! There's not quite enough information for lubridate to give a single clear answer. What it does instead is give an estimate:

```
years(1) / days(1)
#> [1] 365.25
```

If you want a more accurate measurement, you'll have to use an **interval**. An interval is a pair of starting and ending date times, or you can think of it as a duration with a starting point.

You can create an interval by writing `start %--% end`:

```
y2023 <- ymd("2023-01-01") %--% ymd("2024-01-01")
y2024 <- ymd("2024-01-01") %--% ymd("2025-01-01")

y2023
#> [1] 2023-01-01 UTC--2024-01-01 UTC
y2024
#> [1] 2024-01-01 UTC--2025-01-01 UTC
```

You could then divide it by `days()` to find out how many days fit in the year:

```
y2023 / days(1)
#> [1] 365
y2024 / days(1)
#> [1] 366
```


18.4.4 Exercises

1. Explain `days(!overnight)` and `days(overnight)` to someone who has just started learning R. What is the key fact you need to know?
2. Create a vector of dates giving the first day of every month in 2015. Create a vector of dates giving the first day of every month in the *current* year.
3. Write a function that given your birthday (as a date), returns how old you are in years.
4. Why can't `(today() %---% (today() + years(1))) / months(1)` work?

18.5 Time zones

Time zones are an enormously complicated topic because of their interaction with geopolitical entities. Fortunately we don't need to dig into all the details as they're not all important for data analysis, but there are a few challenges we'll need to tackle head on.

The first challenge is that everyday names of time zones tend to be ambiguous. For example, if you're American you're probably familiar with EST, or Eastern Standard Time. However, both Australia and Canada also have EST! To avoid confusion, R uses the international standard IANA time zones. These use a consistent naming scheme `{area}/{location}`, typically in the form `{continent}/{city}` or `{ocean}/{city}`. Examples include "America/New_York", "Europe/Paris", and "Pacific/Auckland".

You might wonder why the time zone uses a city, when typically you think of time zones as associated with a country or region within a country. This is because the IANA database has to record decades worth of time zone rules. Over the course of decades, countries change names (or break apart) fairly frequently, but city names tend to stay the same. Another problem is that the name needs to reflect not only the current behavior, but also the complete history. For example, there are time zones for both "America/New_York" and "America/Detroit". These cities both currently use Eastern Standard Time but in 1969-1972 Michigan (the state in which Detroit is located), did not follow DST, so it needs a different name. It's worth reading the raw time zone database (available at <https://www.iana.org/time-zones>) just to read some of these stories!

You can find out what R thinks your current time zone is with `Sys.timezone()`:

```
Sys.timezone()  
#> [1] "UTC"
```

(If R doesn't know, you'll get an `NA`.)

And see the complete list of all time zone names with `OlsonNames()`:

```
length(OlsonNames())  
#> [1] 597  
head(OlsonNames())
```

```
#> [1] "Africa/Abidjan"      "Africa/Accra"        "Africa/Addis_Ababa"
#> [4] "Africa/Algiers"      "Africa/Asmara"       "Africa/Asmera"
```

In R, the time zone is an attribute of the date-time that only controls printing. For example, these three objects represent the same instant in time:

```
x1 <- ymd_hms("2024-06-01 12:00:00", tz = "America/New_York")
x1
#> [1] "2024-06-01 12:00:00 EDT"

x2 <- ymd_hms("2024-06-01 18:00:00", tz = "Europe/Copenhagen")
x2
#> [1] "2024-06-01 18:00:00 CEST"

x3 <- ymd_hms("2024-06-02 04:00:00", tz = "Pacific/Auckland")
x3
#> [1] "2024-06-02 04:00:00 NZST"
```

You can verify that they're the same time using subtraction:

```
x1 - x2
#> Time difference of 0 secs
x1 - x3
#> Time difference of 0 secs
```

Unless otherwise specified, lubridate always uses UTC. UTC (Coordinated Universal Time) is the standard time zone used by the scientific community and is roughly equivalent to GMT (Greenwich Mean Time). It does not have DST, which makes a convenient representation for computation. Operations that combine date-times, like `c()`, will often drop the time zone. In that case, the date-times will display in the time zone of the first element:

```
x4 <- c(x1, x2, x3)
x4
#> [1] "2024-06-01 12:00:00 EDT" "2024-06-01 12:00:00 EDT"
#> [3] "2024-06-01 12:00:00 EDT"
```

You can change the time zone in two ways:

- Keep the instant in time the same, and change how it's displayed. Use this when the instant is correct, but you want a more natural display.

```
x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
x4a
#> [1] "2024-06-02 02:30:00 +1030" "2024-06-02 02:30:00 +1030"
#> [3] "2024-06-02 02:30:00 +1030"
x4a - x4
#> Time differences in secs
#> [1] 0 0 0
```

(This also illustrates another challenge of times zones: they're not all integer hour offsets!)




- Change the underlying instant in time. Use this when you have an instant that has been labelled with the incorrect time zone, and you need to fix it.

```
x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
x4b
#> [1] "2024-06-01 12:00:00 +1030" "2024-06-01 12:00:00 +1030"
#> [3] "2024-06-01 12:00:00 +1030"
x4b - x4
#> Time differences in hours
#> [1] -14.5 -14.5 -14.5
```

18.6 Summary

This chapter has introduced you to the tools that lubridate provides to help you work with date-time data. Working with dates and times can seem harder than necessary, but hopefully this chapter has helped you see why — date-times are more complex than they seem at first glance, and handling every possible situation adds complexity. Even if your data never crosses a day light savings boundary or involves a leap year, the functions need to be able to handle it.

The next chapter gives a round up of missing values. You've seen them in a few places and have no doubt encounter in your own analysis, and it's now time to provide a grab bag of useful techniques for dealing with them.

1. A year is a leap year if it's divisible by 4, unless it's also divisible by 100, except if it's also divisible by 400. In other words, in every set of 400 years, there's 97 leap years. 
2. <https://xkcd.com/1179/> 
3. You might wonder what UTC stands for. It's a compromise between the English "Coordinated Universal Time" and French "Temps Universel Coordonné". 
4. No prizes for guessing which country came up with the longitude system. 