Dates 1: Recognizing and Formatting Dates

Intro

Understanding how to manipulate dates is a crucial skill when working with health data. From patient admission dates to vaccination schedules, date-related data plays a vital role in epidemiological analyses. In this lesson, we will learn how R stores and displays dates, as well as how to effectively manipulate, parse, and format them. Let's get started!

Learning Objectives

- You understand how dates are stored and manipulated in R
- You understand how to coerce strings to dates
- You know how to handle messy dates
- You are able to change how dates are displayed

Packages

Please load the packages needed for this lesson with the code below:

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
               lubridate)
```

Datasets

IRS Malawi

The first dataset we will be using contains data related to indoor residual spraying (IRS) for malaria control efforts between 2014–2019 in Illovo, Malawi. Notice the dataset is in long format with each row representing a period of time in which IRS occurred within a village. Given the same village is sprayed multiple times at different time points, the village names repeat. Long format datasets are often used when dealing with time series data with repeated measurements, as they are easier to manipulate for analyses and visualization.

```
irs <- read_csv(here("data/Illovo_data.csv"))
```

```
irs
```

```
## # A tibble: 5 × 9
##   village     target_spray sprayed coverage_p
##   <chr>              <dbl>   <dbl>      <dbl>
## 1 Mess                  87      64       73.6
## 2 Nkombedzi            183     169       92.4
## 3 B Compound            16      16      100
## 4 D Compound             3       2       66.7
## 5 Post Office            6       3       50
## # ℹ 5 more variables: start_date_default <date>,
## #   end_date_default <date>, start_date_typical <chr>, …
```

The variables included in the dataset are the following:

- `village`: name of the village where the spraying occurred
- `target_spray`: number of structures targeted for spraying
- `sprayed`: number of structures actually sprayed
- `start_date_default`: day the spraying started, formatted as the default "YYYY-MM-DD"
- `end_date_default`: day the spraying ended, formatted as the default "YYYY-MM-DD"
- `start_date_typical`: day the spraying started, formatted "DD/MM/YYYY"
- `start_date_long`: day the spraying started, with the month written out fully, two-digit day, then four-digit year
- `start_date_messy`: day the spraying started with a mix of different formats.

Inpatient stays

The second dataset is mock data of simulated hospital inpatient stays. It contains admission and discharge dates for 150 patients. Similar to the IRS dataset, the admission

dates are formatted in multiple different ways so that you can practice your formatting skills.

```
ip <- read_csv(here("data/inpatient_data.csv"))
```

```
ip
```

```
## # A tibble: 5 × 6
##   patient_id adm_date_default adm_date_common
##        <dbl> <date>           <chr>
## 1          1 2021-05-23       05/23/2021
## 2          2 2022-12-07       12/07/2022
## 3          3 2022-03-27       03/27/2022
## 4          4 2022-04-28       04/28/2022
## 5          5 2023-06-28       06/28/2023
## # ℹ 3 more variables: adm_date_abbrev <chr>,
## #   adm_date_messy <chr>, disch_date_default <date>
```

Introduction to dates in R

In R, there is a specific class designed to handle dates, known as `Date`. The default format for this class is "YYYY-MM-DD". For example, December 31st 2000 would be represented as `2000-12-31`.

However, if you just enter in such a date string, R will initially consider this to be a character:

```
class("2000-12-31")
```

```
## [1] "character"
```

If we want to create a `Date`, we can use the `as.Date()` function and write in the date following the default format:

```
my_date <- as.Date("2000-12-31")
class(my_date)
```

```
## [1] "Date"
```

With the date format, you can now do things like find the difference between two dates:

```
as.Date("2000-12-31") - as.Date("2000-12-20")
```

```
## Time difference of 11 days
```

This would of course not be possible if you had bare characters:

```
"2000-12-31" - "2000-12-20"
```

```
Error in "2000-12-31" - "2000-12-20" :
  non-numeric argument to binary operator
```

Many other operations apply only to the `Date` class. We'll explore these later.

The default format for `as.Date()` is "YYYY-MM-DD". Other common formats like "MM/DD/YYYY" or "Month DD, YYYY" won't work by default:

```
as.Date("12/31/2000")
as.Date("Dec 31, 2000")
```

However, R will also accept "/" instead of "-" as long as the order is still "YYYY/MM/DD". The dates will be displayed in the default "YYYY-MM-DD" format though:

```
as.Date("2000/12/31")
```

```
## [1] "2000-12-31"
```

So in summary, the only formats that work by default are "YYYY-MM-DD" and "YYYY/MM/DD". Later on in this lesson we will learn how to handle different date formats, and give tips on how to coerce dates that are imported as strings to the `Date` class. For now, the important thing is to understand that dates have their own class which has its own formatting properties.

SIDE NOTE

There is one other data class used for dates, called `POSIXct`. This class specifically handles dates and times together, and the default format is "YYYY-MM-DD HH:MM:SS". However for the purpose of this course, we won't be thinking too much about date-times as that level of analysis is much less common in the world of public health.

Coercing strings to dates

Let's return to our IRS dataset and take a look at how R has classified our date variables!

```
irs %>%
  select(contains("date"))
```

```
## # A tibble: 112 × 5
##    start_date_default end_date_default start_date_typical
##    <date>             <date>           <chr>
##  1 2014-04-07         2014-04-17       07/04/2014
##  2 2014-04-22         2014-04-27       22/04/2014
##  3 2014-05-13         2014-05-13       13/05/2014
##  4 2014-05-13         2014-05-13       13/05/2014
##  5 2014-05-13         2014-05-13       13/05/2014
##  6 2014-05-15         2014-05-26       15/05/2014
##  7 2014-05-27         2014-05-27       27/05/2014
##  8 2014-05-27         2014-05-27       27/05/2014
##  9 2014-05-28         2014-06-16       28/05/2014
## 10 2014-06-18         2014-06-27       18/06/2014
## # ℹ 102 more rows
## # ℹ 2 more variables: start_date_long <chr>, …
```

As we can see, the two columns that were recognized as dates are `start_date_default` and `end_date_default`, which follow R's "YYYY-MM-DD" format:

```
   start_date_default  end_date_default  start_date_typical
   👉<date>👈          👉<date>👈        <chr>
 1 2014-04-07          2014-04-17        07/04/2014
 2 2014-04-22          2014-04-27        22/04/2014
```

All other date columns in our dataset were imported as character strings ("chr"), and if we want to coerce them into dates we will need to tell R that they are dates, as well as specifying the order of the date components.

You may be wondering why it's necessary to specify the order. Well imagine that we have a date written `01–02–03`. Is that January 2nd 2003? February 1st 2003? Or maybe March 2nd 2001? There are so many different conventions for writing dates that if R were to guess the format, there would inevitably be instances where it guessed wrong.

To tackle this, there are two main ways to coerce strings to dates that involve specifying the component order. The first approach relies on base R, and the second uses a package called `lubridate` from the `tidyverse` library. Let's take a look at the base R function first!

Base R

We saw the function used to convert strings to dates using base R in the introduction, the `as.Date()` function. Let's try to use this on our `start_date_typical`column without specifying the order of components to see what happens.

```
irs %>%
  mutate(start_date_typical = as.Date(start_date_typical)) %>%
  select(start_date_typical)
```

```
## # A tibble: 5 × 1
##   start_date_typical
##   <date>
## 1 0007-04-20
## 2 0022-04-20
## 3 0013-05-20
## 4 0013-05-20
## 5 0013-05-20
```

Obviously, this is not at all what we wanted! If we take a look at the original variable, we can see that it's formatted "DD/MM/YYYY". R tried to apply it's default format to these dates, giving us these odd results.

Often R will throw an error if you try to coerce ambiguous strings to dates without specifying the order of its components. But, as we have just seen, this isn't always the case! Always double-check that your code has run how you expected and never rely on error messages alone to ensure that your data transformations have worked correctly.

For R to correctly interpret our dates, we have to use the `format` option and specify the components of our date using a series of symbols. The table below shows the symbols for the most common format components:

Component	Symbol	Example
Year (numeric, with century)	%Y	2023
Year (numeric, without century)	%y	23
Month (numeric, 01-12)	%m	01
Month (written out fully)	%B	January
Month (abbreviated)	%b	Jan
Day of the month	%d	31
Day of the week (numeric, 1-7 with Sunday being 1)	%u	5
Day of the week (written out fully)	%A	Friday
Day of the week (abbreviated)	%a	Fri

If we come back to our original `start_date_typical` variable, we see it's formatted as "DD/MM/YYYY" which is the day of the month, followed by the month represented as a number (01-12), and then the four-digit year. If we use these symbols, we should get the results we're looking for.

```
irs %>%
  mutate(start_date_typical = as.Date(start_date_typical, format="%d%m%Y"))
```

```
## # A tibble: 5 × 9
##   village     target_spray sprayed coverage_p
##   <chr>              <dbl>   <dbl>      <dbl>
## 1 Mess                  87      64       73.6
## 2 Nkombedzi            183     169       92.4
## 3 B Compound            16      16      100
## 4 D Compound             3       2       66.7
## 5 Post Office            6       3       50
## # ℹ 5 more variables: start_date_default <date>,
## #   end_date_default <date>, start_date_typical <date>, …
```

Ok so it's still not what we wanted. Do you have an idea why that may be? It's because the components of our dates are separated by a slash "/", which we have to include into our format option. Let's try it again!

```
irs %>%
  mutate(start_date_typical = as.Date(start_date_typical,
format="%d/%m/%Y"))%>%
  select(start_date_typical)
```

```
## # A tibble: 5 × 1
##   start_date_typical
##   <date>
## 1 2014-04-07
## 2 2014-04-22
## 3 2014-05-13
## 4 2014-05-13
## 5 2014-05-13
```

This time it worked perfectly! Now we know how to coerce strings to dates in base R using the `as.Date()` function with the `format` option.

Coerce long date

Try to coerce the column `start_date_long` from the IRS dataset to the `Date` class. Don't forget to include all elements into the format option, including the symbols that separate the components of the date!

Find code errors

Can you find all the errors in the following code?

```
as.Date("June 26, 1987", format = "%b%d%y")
```

```
## [1] NA
```

Lubridate

The `lubridate` package provides a much more user-friendly way of coercing strings to dates than base R. With this package, all that's necessary is that you specify the order in which the year, month, and day appear using "y", "m", and "d", respectively. With these functions, specifying the characters that separate the different components of the date isn't necessary.

Let's take a look at a few examples:

```
dmy("30/04/2002")
```

```
## [1] "2002-04-30"
```

```
mdy("April 30th 2002")
```

```
## [1] "2002-04-30"
```

```
ymd("2002-04-03")
```

```
## [1] "2002-04-03"
```

That was easy enough! And as we can see, our dates are displayed using the default R format. Now that we understand how to use the functions in the lubridate package, let's try to apply them to the `start_day_long` variable from our dataset.

```
irs %>%
  mutate(start_date_long = mdy(start_date_long)) %>%
  select(start_date_long)
```

```
## # A tibble: 5 × 1
##   start_date_long
##   <date>
## 1 2014-04-07
## 2 2014-04-22
## 3 2014-05-13
```

```
## 4 2014-05-13
## 5 2014-05-13
```

Perfect, that's exactly what we wanted!

Coerce typical date

Try to coerce the column `start_date_typical` from the IRS dataset to the `Date` class using the functions in the lubridate package.

Base and lubridate formatting The following table contains the formats found in the `adm_date_abbr` and `adm_date_messy` formats from our inpatient dataset. See if you can fill in the blank cells:

Date example	Base R	Lubridate
Dec 07, 2022		
03-27-2022		mdy
28.04.2022		
	%Y/%m/%d	

Now we know two ways to coerce strings to the date class by specifying the order of the components! But what if we have multiple date formats within the same column? Let's move on to the next section to find out!

Handling messy dates with `lubridate::parse_date_time()`

When working with dates, sometimes you'll have various different formats within the same column. Luckily, lubridate has a useful function just for this purpose! The `parse_date_time()` function is similar to the previous functions we saw in the lubridate package, but with more flexibility and the possibility of including multiple date formats into the same call using the `orders` argument. Let's quickly take a look at how it works with a few simple examples.

To get an understanding of how to use `parse_date_time()`, let's apply it to a single string that we want to coerce to a date.

```
parse_date_time("30/07/2001", orders="dmy")
```

```
## [1] "2001-07-30 UTC"
```

That worked perfectly! Using the function like this is equivalent to using `dmy()`. However, the real power in the function is when we have multiple date strings that have different formats.

The "UTC" part is the default time zone used to parse the date. This can be changed in with the `tz=` argument, but changing the default time zone is rarely necessary when dealing with dates alone, as opposed to date-times.

Let's take a look at another example but with two different formats:

```
parse_date_time(c("1 Jan 2000", "July 30th 2001"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-01-01 UTC" "2001-07-30 UTC"
```

Note that this specific example will still work if you change the order in which you present the formats:

```
parse_date_time(c("1 Jan 2000", "July 30th 2001"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-01 UTC" "2001-07-30 UTC"
```

The last code chunk still works because `parse_date_time()` checks each format specified in the `orders` argument until it finds a match. This means whether you list "dmy" first or "mdy" first, it will try both formats on each date string to see which one fits. The order doesn't matter for distinct date strings that can only match one format.

However, when dealing with ambiguous dates like "01/02/2000" and "01/03/2000", which could be interpreted as either January 2nd and January 3rd or February 1st and March 1st respectively, the order in `orders` really does matter:

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-02 UTC" "2000-01-03 UTC"
```

In the example above, because "mdy" is listed first, the function interprets the dates as January 2nd and January 3rd. But, if you switched the order and listed "dmy" first, it would interpret the dates as February 1st and March 1st:

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-02-01 UTC" "2000-03-01 UTC"
```

Hence, when there's potential ambiguity in the date strings, the order in which you specify the formats becomes vital.

Using parse_date_time

The dates in the code below are November 9th 2002, December 4th 2001, and June 5th 2003. Complete the code to coerce them from strings to dates.

```
parse_date_time(c("11/09/2002", "12/04/2001", "2003-06-05"), orders=c(...))
```

Let's return to our dataset, this time to the `start_date_messy` column.

```
irs %>%
  select("start_date_messy")
```

```
## # A tibble: 5 × 1
##   start_date_messy
##   <chr>
## 1 04/07/14
## 2 April 22 2014
## 3 May 13 2014
## 4 13-05-2014
## 5 May 13 2014
```

Given that this column that was created specifically for this course, we know the different date formats in this column. In your own work, always be sure that you know what format your dates are in, as we know some can be ambiguous.

Here, we're working with four different formats, specifically:

- YYYY/MM/DD
- Month DD YYYY
- DD-MM-YYYY
- MM/DD/YYYY

Let's see how this looks in lubridate compared to base R:

Example date	Base R	Lubridate
2014/05/13	%Y/%m/%d	ymd
May 13 2014	%B %d %Y	mdy
27-05-2014	%d-%m-%Y	dmy
07/21/14	%m/%d/%y	mdy

Here, lubridate considers there to be only three different formats ("ymd", "mdy", and "dmy"). Now that we know how our data is formatted, we can use the `parse_date_time()` function to clean it up.

```
irs %>%
  select(start_date_messy) %>%
  mutate(start_date_messy = parse_date_time(start_date_messy, orders =
c("mdy", "dmy", "ymd")))
```

start_date_messy
2014-04-07
2014-04-22
2014-05-13
2014-05-13
2014-05-13

That's much better! R has correctly formatted our column and it is now recognized as a date variable. You may be wondering if the ordering of the formats is necessary in this case. Let's try a different order and find out!

```
irs %>%
  select(start_date_messy) %>%
  mutate(start_date_messy = parse_date_time(start_date_messy, orders =
c("dmy", "mdy", "ymd")))
```

start_date_messy
2014-04-07
2014-04-22
2014-05-13
2014-05-13
2014-05-13

That didn't seem to make a difference, the dates are still correctly formatted! If you're wondering why the order mattered in our previous example but not here, it's to do with how the parse_date_times() function works. When given multiple orders, the function tries to find the best fit for a subset of observations by considering the dates separators' and favoring the order in which the formats were supplied. In our last example, both dates were separated by a "/" and both supplied formats ("dmy" and "mdy") were possible formats, so the function favored the first one given.

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-02 UTC" "2000-01-03 UTC"
```

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-02-01 UTC" "2000-03-01 UTC"
```

In our IRS dataset, we also had formats that could be ambiguous such as DD-MM-YYYY and MM/DD/YYYY. But here the function can use the separators as a hint to find formatting rules and distinguish between different formats. For example, if we have an ambiguous date such as `01-02-2000`, but also a date with the same separator that is not ambiguous such as `30-05-2000`, the function will determine that the most likely answer is that all dates separated by a "-" are in DD-MM-YYYY format, and apply this rule recursively to the input data. If you want to learn more about the details of the `parse_date_time()` function, click here or run `?parse_date_time` in R!

Using parse_date_time with adm_date_messy With the help of the table you completed from the exercise in **Section 6.2 Lubridate**, use the `parse_date_time()` function to clean up the `adm_date_messy` column in the inpatient dataset, `ip`!

Changing how dates are displayed

Up until now we have been coercing strings of various formats to the `Date` class which follows a default "YYYY-MM-DD" format. But what if we want our dates to be displayed in a specific format that's different from this default, such as when we're creating reports or graphs? This is made possible by converting the dates back into strings using using the `format()` function in base R!

The `format()` function gives you lots of freedom to customize the appearance of your dates. You can do this by using the same symbols we've encountered before with the `as.Date()` function, ordering them to match how you want your date to look. Let's go back to the table to refresh our memory on how different date parts are represented in base R.

Component	Symbol	Example
Year (numeric, with century)	%Y	2023
Year (numeric, without century)	%y	23
Month (numeric, 01-12)	%m	01
Month (written out fully)	%B	January
Month (abbreviated)	%b	Jan
Day of the month	%d	31
Day of the week (numeric, 1-7 with Monday being 1)	%u	5
Day of the week (written out fully)	%A	Friday
Day of the week (abbreviated)	%a	Fri

Great, now let's try to apply this function to a single date. Let's say we want the date `2000-01-31` to be displayed as "Jan 31, 2000".

```
my_date <- as.Date("2000-01-31")
format(my_date, "%b %d, %Y")
```

```
## [1] "Jan 31, 2000"
```

Create date vector Format the date below to MM/DD/YYYY using the format function:

```
my_date <- as.Date("2018-05-07")
```

Now, let's try using it on our dataset. Let's create a new variable called `start_date_char` from the `start_date_default` column in our dataset. We'll format it to be displayed as DD/MM/YYYY.

```
irs %>%
  mutate(start_date_char = format(start_date_default, "%d/%m/%Y")) %>%
  select(start_date_char)
```

```
## # A tibble: 5 × 1
##   start_date_char
##   <chr>
## 1 07/04/2014
## 2 22/04/2014
## 3 13/05/2014
## 4 13/05/2014
## 5 13/05/2014
```

Looking great! Let's do one last example using our `end_date_default` variable and formatting it as Month DD, YYYY.

```
irs %>%
  mutate(end_date_char = format(end_date_default, "%B %d, %Y")) %>%
  select(end_date_char)
```

```
## # A tibble: 5 × 1
##   end_date_char
##   <chr>
## 1 April 17, 2014
## 2 April 27, 2014
## 3 May 13, 2014
## 4 May 13, 2014
## 5 May 13, 2014
```

Looks great!

Wrap Up!

Congratulations on finishing the first Dates lesson! Now that you understand how Dates are stored, displayed, and formatted in R, you can move on to the next section where you'll learn how to perform manipulations with dates and how to create basic time series graphs.

Answer Key

Coerce long date

```
irs <- irs %>%
  mutate(start_date_long = as.Date(start_date_long, format="%B %d %Y"))
```

Find code errors

```
as.Date("June 26, 1987", format = "%B %d, %Y")
```

Coerce typical date

```
irs %>%
  mutate(start_date_typical = dmy(start_date_typical))
```

Base and lubridate formatting

Date example	Base R	Lubridate
Dec 07, 2022	%b %d, %Y	mdy
03-27-2022	%m-%d-%Y	mdy
28.04.2022	%d.%m.%Y	dmy
2021/05/23	%Y/%m/%d	ymd

Using parse_date_time

```
parse_date_time(c("11/09/2002", "12/04/2001", "2003-06-05"), orders=c("mdy",
"ymd"))
```

Using parse_date_time with adm_date_messy

```
ip %>%
  mutate(adm_date_messy = parse_date_time(adm_date_messy, orders = c("mdy",
"dmy", "ymd")))
```

Format date vector

```r
my_date <- as.Date("2018-05-07")
format(my_date, "%m/%d/%Y")
```

Contributors

The following team members contributed to this lesson:

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement