Dates 2: Working with Dates

GRAPH Network & WHO, supported by the Global Fund to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Intro

You now have a solid understanding of how dates are stored, displayed, and formatted in R. In this lesson, you will learn how to perform simple analyses with dates, such as calculating the time between date intervals and creating time series graphs! These skills are crucial for anyone working with health data, as they are the basis to understanding temporal patterns such as the spread of disease, changes in population-level health indicators, and the impact of preventive measures!

Learning Objectives

- You know how to calculate intervals between dates
- You know how to extract components from date columns
- You know how to round dates
- You are able to create simple time series graphs

Packages

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

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
               lubridate,
               ggplot2)
```

Dataset

The first dataset we will be working with is the IRS dataset from previous the previous lesson. Check out the first Dates lesson for more information about the contents of this dataset.

```
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 second dataset that we will be using contains data from the same study that the IRS data was taken from. Here, we have the average monthly incidence rate of malaria per 1000 people from 2015-2019 for villages that received IRS (cases) and villages that didn't receive IRS (controls). The dataset also contains the average minimum temperature and average maximum temperature in Illovo for each month from 2015-2019.

```
inc_temp <- read_csv(here("data/Illovo_ir_weather.csv"))
```

```
inc_temp
```

```
## # A tibble: 5 × 5
##   date       ir_case ir_control avg_min avg_max
##   <date>       <dbl>      <dbl>   <dbl>   <dbl>
## 1 2015-01-10    42.9       19.6    21.2    31.6
## 2 2015-02-03    61.0       10.1    21.5    32.9
## 3 2015-03-11    74.1       56.8    20.6    33.4
## 4 2015-04-15    95.2       34.7    18.5    32.3
## 5 2015-05-05    89.8       31.9    15.9    31.4
```

The variables included in the dataset are the following:

- `date`: Monthly time points ranging from 2015-2019, with the day of the month randomly generated

- `ir_case`: Average incidence rate of malaria per 1000 people for villages that received IRS
- `ir_control`: Average incidence rate of malaria per 1000 people for villages that did not receive IRS
- `avg_min`: Average monthly minimum temperature in Celsius
- `avg_min`: Average monthly maximum temperature in Celsius

WEATHER

```
weather <- read_csv(here("data/Illovo_weather.csv"))
```

```
weather
```

```
## # A tibble: 5 × 4
##   date       min_temp max_temp  rain
##   <date>        <dbl>    <dbl> <dbl>
## 1 2015-01-01     21.5     29.9  21.7
## 2 2015-01-02     19.6     30.4   2.2
## 3 2015-01-03     21.6     29.9  25.8
## 4 2015-01-04     20       29.5   1
## 5 2015-01-05     20       32.2  53
```

Calculating Date Intervals

To start off, we're going to look at two ways to calculate intervals, the first using the "-" operator in base R, and the second using the interval operator from the `lubridate` package. Let's take a look at both of these and compare.

Using the "-" operator

The first way to calculate time differences is using the "-" operator to subtract one date from another. Let's create two date variables and try it out!

```
date_1 <- as.Date("2000-01-01") # January 1st, 2000
date_2 <- as.Date("2000-01-31") # January 31st, 2000
date_2 - date_1
```

```
## Time difference of 30 days
```

It's that simple! Here we can see that R outputs the time difference in days.

Using the interval operator from `lubridate`

The second way to calculate time intervals is by using the `%--%` operator from the `lubridate` package. We can see here that the output is slightly different to the base R output.

```
date_1 %--% date_2
```

```
## [1] 2000-01-01 UTC--2000-01-31 UTC
```

Our output is an interval between two dates. If we want to know how long has passed in days, we have to use the `days()` function. The `(1)` here tells lubridate to count in increments of one day at a time.

```
date_1 %--% date_2/days(1)
```

```
## [1] 30
```

Technically, specifying `days(1)` isn't actually necessary, we can also leave the parentheses empty (ie. `days()`)and get the same result because lubridate's default is to count in increments of 1. However, if we want to count in increments of 5 days for example, we can specify `days(5)` and the result returned to us will be 6, because 5*6=30.

```
date_1 %--% date_2/days(5)
```

```
## [1] 6
```

Use all the three different methods to calculate the time between the dates below. Use the `weeks()` function in place of `days()` in the lubridate method to return the answer in weeks.

```
oct_31 <- as.Date("2023-10-31")
jul_20 <- as.Date("2023-07-20")
```

Comparison

So which of the methods is the best to use? Well, you can probably tell that lubridate has more flexibility, and thanks to how it handles dates (the specifics of which are beyond the scope of the course), it's more accurate!

Let's take a look at an example of this by calculating an interval of 6 years. With base R, the results are returned in days. If we want to get the result in years, we have to use the `as.numeric()` function to transform the results into a number without the days unit, and divide by 365.25. We use 365.25 because there is one extra day every 4 years (i.e. leap

years), so on average there is 365.25 days per year. With lubridate, we can just specify `years()`. Let's see how each performs in the following example.

```
date_1 <- as.Date("2000-01-01") # January 1st, 2000
date_2 <- as.Date("2006-01-01") # January 1st, 2006
as.numeric(date_2-date_1)/365.25
```

```
## [1] 6.001369
```

```
date_1 %--% date_2/years()
```

```
## [1] 6
```

Ok, so they're very similar, but not exactly the same! Obviously the correct answer is that 6 years has passed between the two given dates. Yet, dates can get complicated since the number of days in a year, or even a month, isn't always the same. Trying to calculate date intervals with base R is an approximation rather than an exact answer. In the example above, we couldn't get exactly 6 years using the minus oprator in base R, even if we divide by 365, 365.25, or 366.

```
as.numeric(date_2-date_1)/365
```

```
## [1] 6.005479
```

```
as.numeric(date_2-date_1)/365.25
```

```
## [1] 6.001369
```

```
as.numeric(date_2-date_1)/366
```

```
## [1] 5.989071
```

Lubridate on the other hand can handle these complexities, so it will give an exact answer. The differences are slight, but in terms of flexibility and accuracy, lubridate is the clear winner!

SIDE NOTE

Although we don't cover date-times in this course, it's good to know that `lubridate` is particularly handy for dealing with time zones and daylight savings shifts.

Can you apply this to our IRS dataset? Create a new column called `spraying_time` and using lubridates `%--%` operator, calculate the number of days between `start_date_default` and `end_date_default`.

Extracting Date Components

Sometimes during your data cleaning or analysis, you may need to extract a specific component of your date variable. A set of useful functions within the `lubridate` package allows you to do exactly this. For example, if we wanted to create a column with just the month that spraying started at each interval, we could use the `month()` function in the following way:

```
irs %>%
  mutate(month_start = month(start_date_default))%>%
  select(village, start_date_default, month_start)
```

```
## # A tibble: 5 × 3
##   village     start_date_default month_start
##   <chr>       <date>                  <dbl>
## 1 Mess        2014-04-07                  4
## 2 Nkombedzi   2014-04-22                  4
## 3 B Compound  2014-05-13                  5
## 4 D Compound  2014-05-13                  5
## 5 Post Office 2014-05-13                  5
```

As we can see here, this function returns the month as a number from 1-12. For our first observation, the spraying started during the fourth month, so in April. It's that simple! .If we want to have R display the month written out rather than the number underneath it, we can use the `label=TRUE` argument.

```
irs %>%
  mutate(month_start = month(start_date_default, label=TRUE))%>%
  select(village, start_date_default, month_start)
```

```
## # A tibble: 5 × 3
##   village     start_date_default month_start
##   <chr>       <date>             <ord>
## 1 Mess        2014-04-07         Apr
## 2 Nkombedzi   2014-04-22         Apr
## 3 B Compound  2014-05-13         May
## 4 D Compound  2014-05-13         May
## 5 Post Office 2014-05-13         May
```

Likewise, if we wanted to extract the year, we would use the `year()` function.

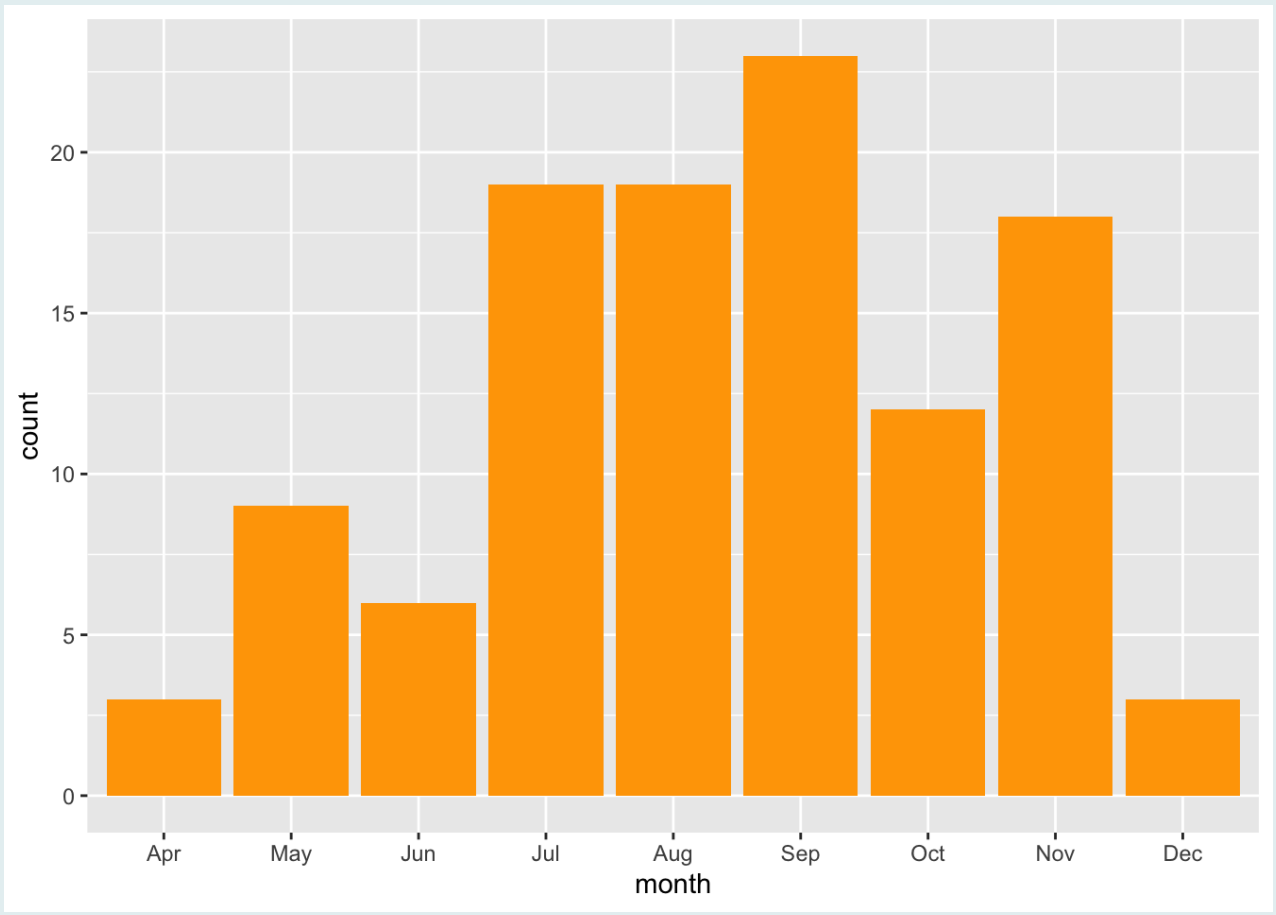
```
irs %>%
  mutate(year_start = year(start_date_default)) %>%
  select(village, start_date_default, year_start)
```

```
## # A tibble: 5 × 3
##   village     start_date_default year_start
##   <chr>       <date>                  <dbl>
## 1 Mess        2014-04-07               2014
## 2 Nkombedzi   2014-04-22               2014
## 3 B Compound  2014-05-13               2014
## 4 D Compound  2014-05-13               2014
## 5 Post Office 2014-05-13               2014
```

Create a new variable called `wday_start` and extract the day of the week that the spraying started in the same way as above but with the `wday()` function. Try to display the days of the week written out rather than numerically.

One reason why you may want to extract specific date components is when you want to visualize your data, which is very simple to do in R using the `ggplot2` package! For example, let's say we wanted to compare the months when spraying starts, we can do this by creating a new month variable with the `month()` function, and plotting a bar graph with `geom_bar`.

```
irs %>%
  mutate(month = month(start_date_default, label = TRUE)) %>%
  ggplot(aes(x = month)) +
  geom_bar(fill = "orange")
```



Here we can see that most spraying campaigns started between July and November, with none occurring in the first three months of the year. The authors of the paper that this data was drawn from have stated that spraying campaigns aimed to finish just as the rainy season (November-April) in Malawi started. This was both for practical reasons and in anticipation of higher malaria transmission. We can see this temporal spraying pattern reflected in our graph!

Create a new graph of months when the spraying campaign ended and compare it to the graph of when they started. Does this match your expectations considering the `spraying_time`variable that you created in the exercise of the previous section?

Rounding

Sometimes it's necessary to round our dates up or down if we want to analyze or vizualize our data in a meaningful way. First, let's see what we mean by rounding with a few simple examples.

Let's take the date March 17th 2012. If we wanted to round down to the nearest month, then we would use the `floor_date()` function from `lubridate` with the `unit="month"`

```
my_date_down <- as.Date("2012-03-17")
floor_date(my_date_down, unit="month")
```

```
## [1] "2012-03-01"
```

As we can see, our date is now March 1st, 2012.

If we wanted to round up, we can use the `ceiling_date()` function. Let's try this on out on the date January 3rd 2020.

```
my_date_up <- as.Date("2020-01-03")
ceiling_date(my_date_up, unit="month")
```

```
## [1] "2020-02-01"
```

With `ceiling_date()`, January 3rd had been rounded up to February 1st.

Finally, we can also simply round without specifying up or down and the dates are automatically round to the nearest specified unit.

```
my_dates <- as.Date(c("2000-11-03", "2000-11-27"))
round_date(my_dates, unit="month")
```

```
## [1] "2000-11-01" "2000-12-01"
```

Here we can see that by rounding to the nearest month, November 3rd is round down to November 1st, and November 27th is round up to December 1st.

We can also round up or down to the nearest year. What do you think the output would be if we round down the date November 29th 2001 to the nearest year:

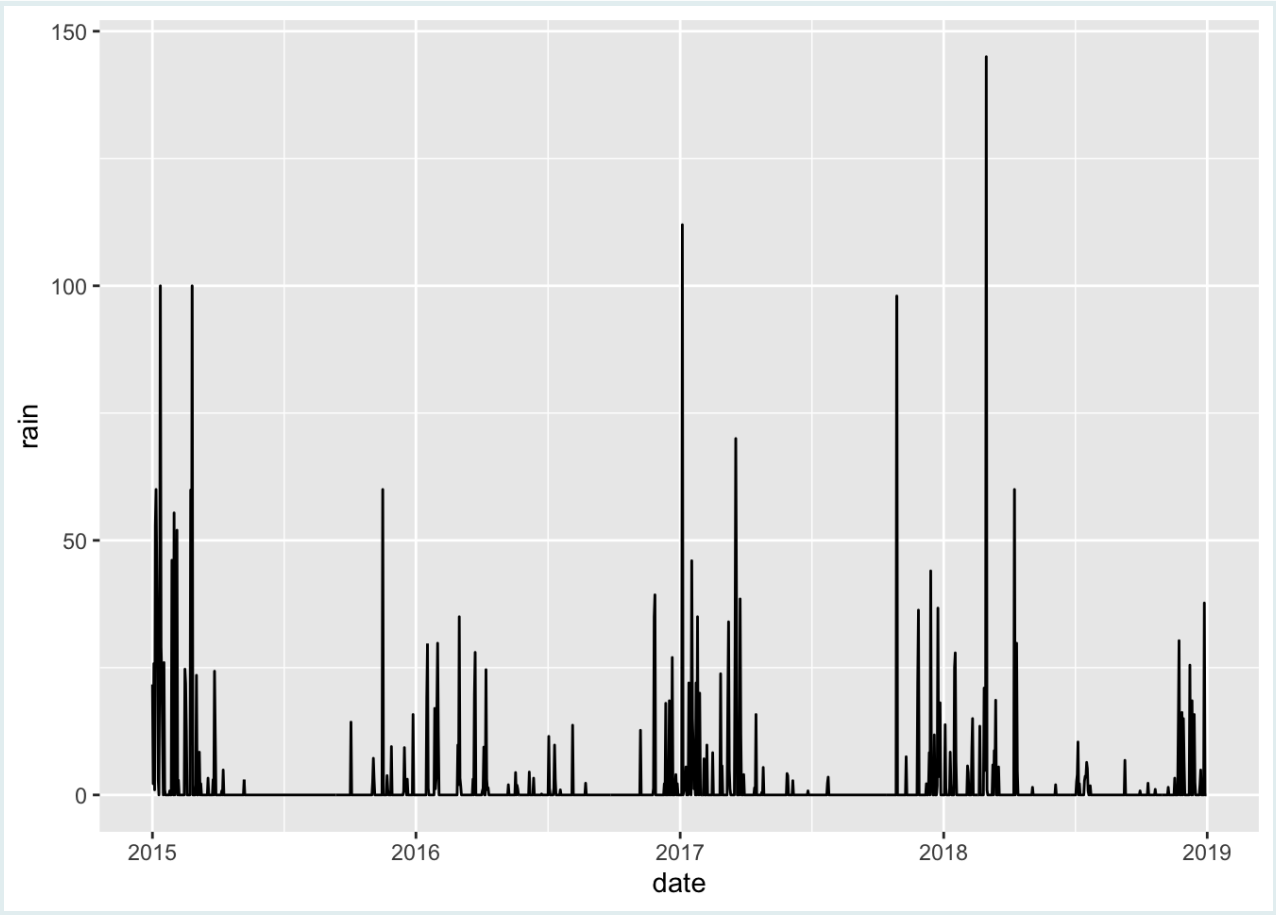
```
date_round <- as.Date("2001-11-29")
floor_date(date_round, unit="year")
```

Hopefully what we mean by rounding is a little bit more clear! So why could this be helpful with our data? Well, let's now turn to our weather data.

```
weather
```

As we can see, our weather data is recorded daily, but this level of detail isn't ideal for studying how weather patterns affect malaria transmission, which follows a seasonal pattern. Daily weather data can be quite noisy given the significant variation from one day to the next. For example, a graph of daily rainfall wouldn't be very informative. Let's try it out to see:

```
weather %>%
  ggplot()+
  geom_line(aes(date, rain))
```



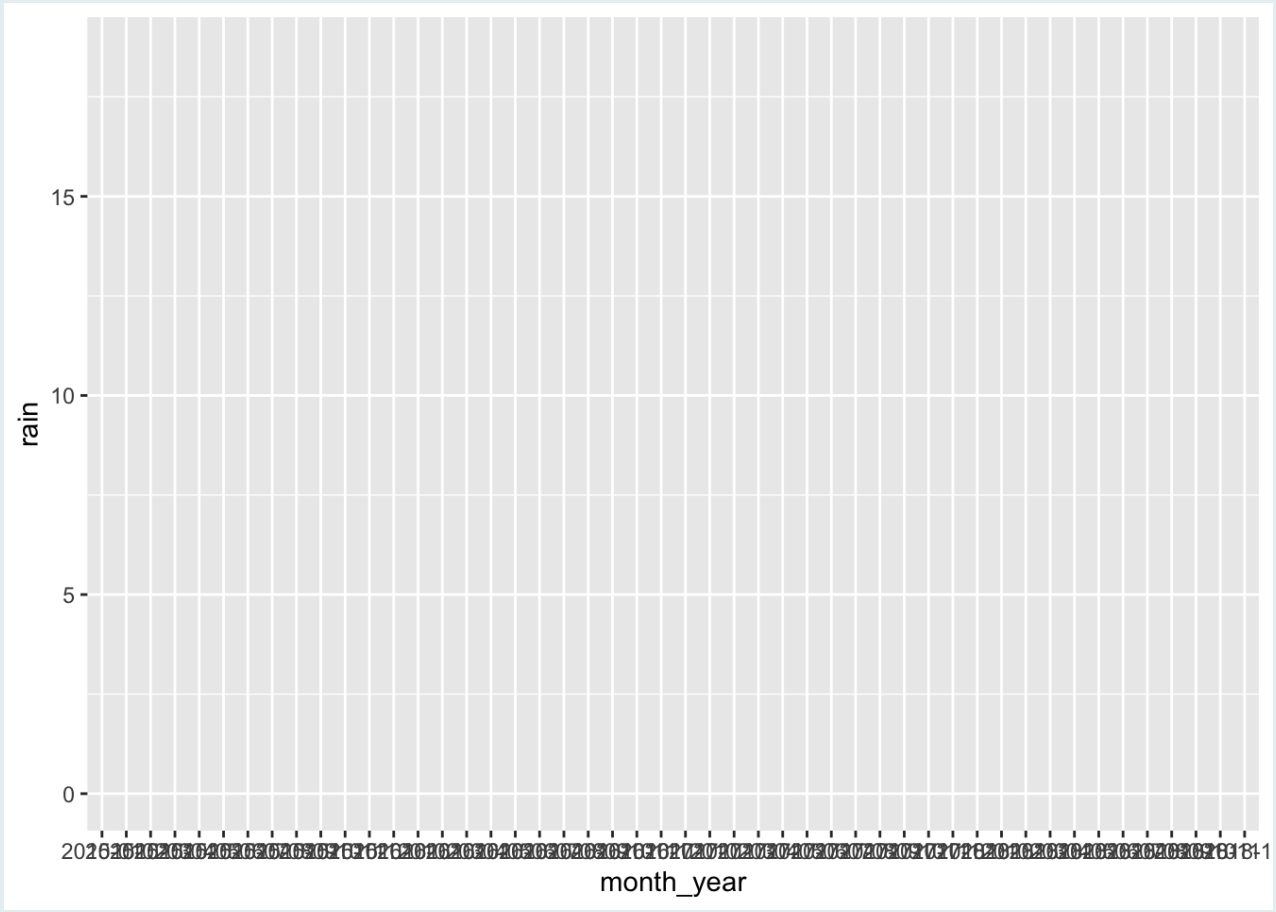
Aside from being visually messy, this graph fails to illustrate seasonal trends effectively. Monthly aggregation is a more effective approach for capturing seasonal variations and reducing noise. If we wanted to plot the monthly average rainfall, our first attempt may be to use the `str_sub()` function to extract the first seven characters of our date (the month and year component).

```
weather_bad <- weather %>%
  mutate(month_year=str_sub(date, 1, 7)) %>%
  group_by(month_year) %>%
  summarise(rain=mean(rain))
weather_bad
```

However, if we try to plot this, our new variable `month_year` is no longer a date variable, so we can't plot graphs over time as it's not continuous!

```
 weather_bad %>%
  ggplot() +
  geom_line(aes(month_year, rain))
```

```
## `geom_line()`: Each group consists of only one observation.
## ℹ Do you need to adjust the group aesthetic?
```

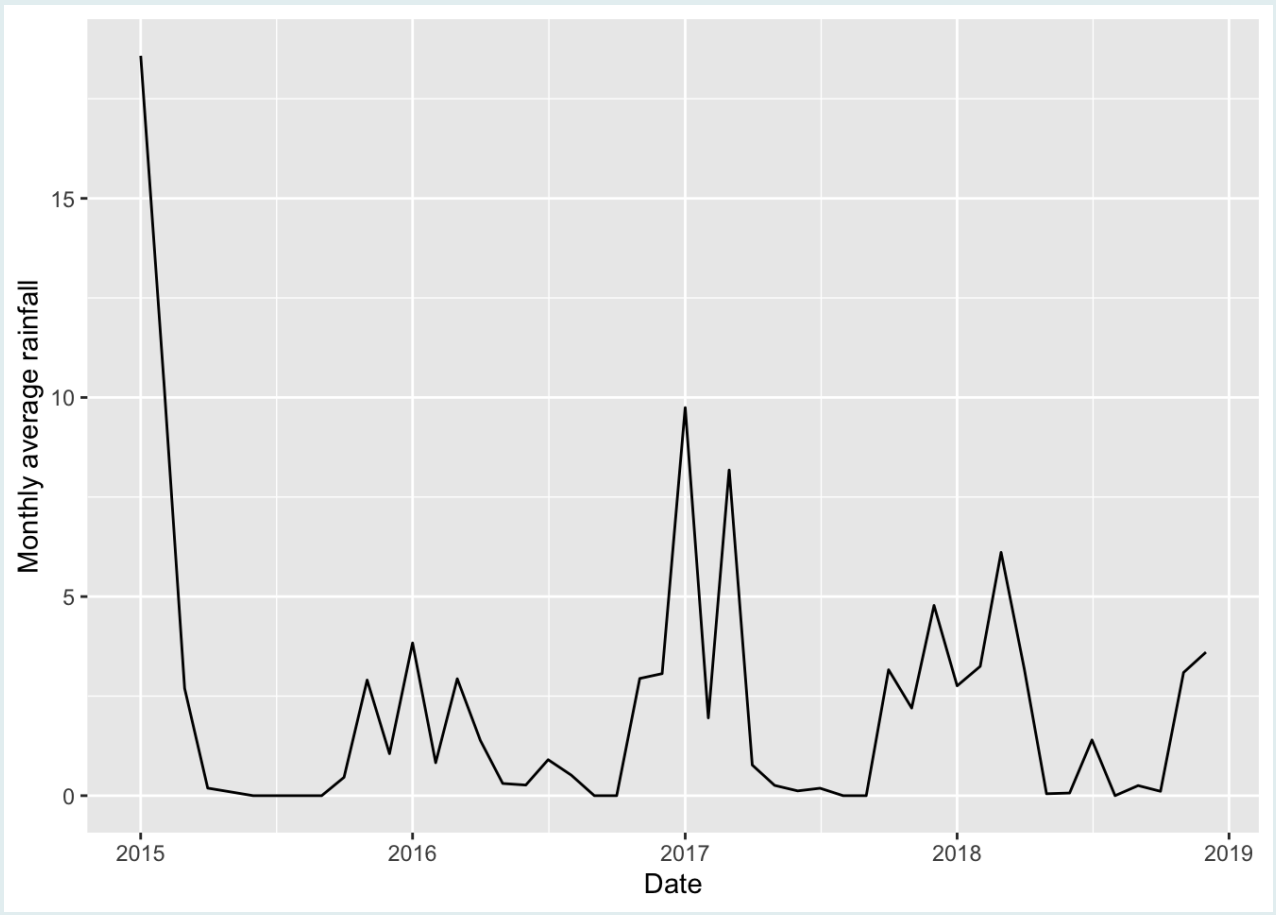


The best way to do this is to first round our dates down to the month using `floor_date()` function, then group our data by our new `month_year` variable, and then calculate the monthly average. Let's try it out now.

```
weather <- weather %>%
  mutate(month_year=floor_date(date, unit="month")) %>%
  group_by(month_year) %>%
  summarise(rain=mean(rain))
weather
```

Now we can plot our data and we'll have a graph of the average monthly rainfall over the 4 year spraying period.

```
 weather %>%
  ggplot() +
  geom_line(aes(month_year, rain)) +
  labs(x="Date", y="Monthly average rainfall")
```



That looks much better! Now we get a much clearer picture of seasonal trends and yearly variations.

Using the weather data, create a new graph plotting the average monthly minimum and maximum temperatures from 2015-2019.

WRAP UP!

[XXX NICE WRAP UP MESSAGE OR SUMMARY IF NEEDED HERE XXX]

Answer Key

```
irs %>%
  mutate(wday_start= wday(start_date_default, label=TRUE)) %>%
  select(village, start_date_default, wday_start)
```

```
## # A tibble: 5 × 3
##   village     start_date_default wday_start
##   <chr>       <date>             <ord>
## 1 Mess        2014-04-07         Mon
## 2 Nkombedzi   2014-04-22         Tue
## 3 B Compound  2014-05-13         Tue
## 4 D Compound  2014-05-13         Tue
## 5 Post Office 2014-05-13         Tue
```

Contributors

The following team members contributed to this lesson:

(make sure to update the contributor list accordingly!)