Intro to Tidyverse

Tonya Brunetti

4/5/2021

Getting all your dependencies in order

The first step is to make sure you have all your packages/libraries installed and loaded into your Rstudio environment. You only ever need to install packages one time per R version you use per computer OS, while you have to load libraries each time a new R session is started. In R, there are two ways to install packages since R has two main code repositories. One is called **CRAN** and the other is called **Bioconductor**. CRAN is the main basic R respository, while Bioconductor is where a lot of data science packages/libraries stored. We will be using both in this tutorial.

Installing Libraries

To install packages that are located in the **CRAN** repository, you can use the `install.packages()` command. You will notice I have set an extra parameter `dependencies = TRUE`. This means if the package you are installing requires other R software packages that are not already installed, R will automatically install them, if possible.

```
install.packages("BiocManager", dependencies = TRUE)
install.packages("tidyverse", dependencies = TRUE)
```

Although not used in this tutorial, if you have a to install a package that is located in the **Bioconductor** repository, you can use `BiocManager::install()` instead of `install.packages()`.

Loading Libraries

In order to load an installed package into your current Rstudio environment, you use the `library()` command. This needs to be done anytime you open a new Rstudio session or anytime you restart your Rstudio session. You only need to load in the the packages you will be using commands from.

```
    library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.0     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Your first command – setwd()

I strongly recommend setting the path to your working directory before beginning any coding. This is the main directory your code will run from so if you have files in your working directory, you can easily access those files and directories. By setting the working directory, R automatically inherits and fills in the beginning location with the location of your `setwd()` path command. For example, let me set my working directory to my `Downloads` folder:

```
setwd("/home/tonya/Downloads/")
```

You will notice as soon as you run the command, in your Console it know shows you where your Rstudio session is being run. That means if you save a file or import a file, it looks for the file in the `setwd()` location first or saves the files and images in the `setwd()` location. You can always append to the current path if you want to search or save files elsewhere. Sometimes you will notice the ~ symbol. This is shorthand for your home/username directory.

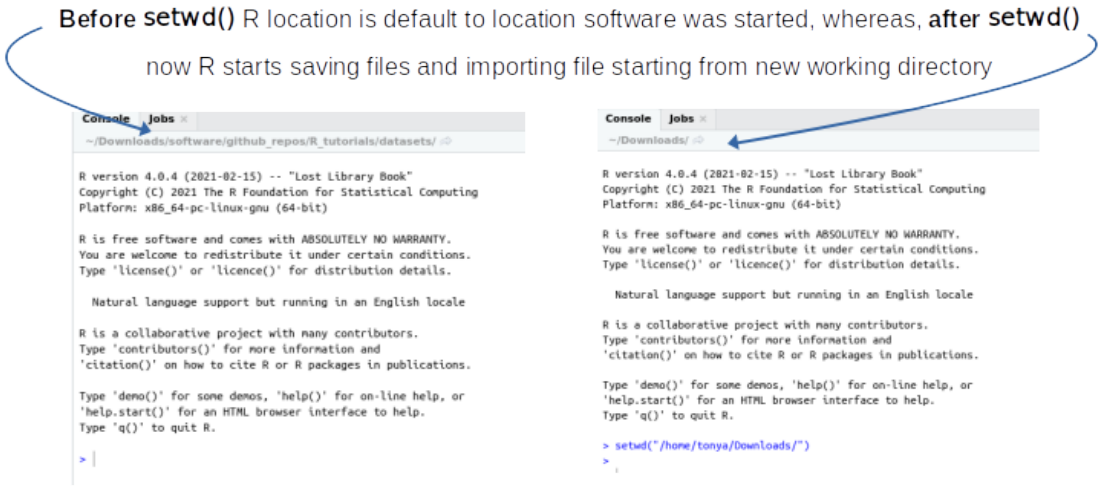


Figure 1: Setting a current working directory path

What does setting a current working directory path mean?

As stated, above this is the directory your current R session will look for and save files. If you load a file from this directory, R automatically looks for the file name in the listed directory, without having to give the full path location. Also, if you save a file, it will automatically save to this directory location.

Reading in a dataset

In the tidyverse world of R, you can read in csv, Excel spreadsheets, tab-delimited text files, etc... with many simple commands:

command	library	explanation	return type
read_csv()	readr	read in comma-separated value text file	tibble
read_tsv()	readr	read in tab-separated value text file	tibble
read_delim()	readr	read in a text file separated by input character	tibble
read_xls()	readxl	read in Excel file	tibble
read_xlsx()	readxl	read in Excel file	tibble
read_excel()	readxl	read in Excel file	tibble

```
    read_csv(file = "covid_19_clean_complete.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   `Province/State` = col_character(),
##   `Country/Region` = col_character(),
##   Lat = col_double(),
##   Long = col_double(),
##   Date = col_date(format = ""),
##   Confirmed = col_double(),
##   Deaths = col_double(),
##   Recovered = col_double(),
##   Active = col_double(),
##   `WHO Region` = col_character()
## )
```

```
## # A tibble: 49,068 x 10
##    `Province/State`    `Country/Region`   Lat   Long Date       Confirmed Deaths
##    <chr>               <chr>            <dbl>  <dbl> <date>         <dbl>  <dbl>
##  1 <NA>                Afghanistan       33.9  67.7  2020-01-22         0      0
##  2 <NA>                Albania           41.2  20.2  2020-01-22         0      0
##  3 <NA>                Algeria           28.0   1.66 2020-01-22         0      0
##  4 <NA>                Andorra           42.5   1.52 2020-01-22         0      0
##  5 <NA>                Angola           -11.2  17.9  2020-01-22         0      0
##  6 <NA>                Antigua and Bar~  17.1 -61.8  2020-01-22         0      0
##  7 <NA>                Argentina        -38.4 -63.6  2020-01-22         0      0
##  8 <NA>                Armenia           40.1  45.0  2020-01-22         0      0
##  9 Australian Capital~ Australia        -35.5 149.   2020-01-22         0      0
## 10 New South Wales     Australia        -33.9 151.   2020-01-22         0      0
## # ... with 49,058 more rows, and 3 more variables: Recovered <dbl>,
## #   Active <dbl>, WHO Region <chr>
```

Another way to do it is to specify which library the command is coming from followed by `::` :

```
    readr::read_csv(file = "covid_19_clean_complete.csv")
```

As we saw previously with the dataframe, it does print some data (not all of it unlike a dataframe) to the Console screen. However, we need to store this as a variable so we can apply functions to it. Let's call our dataset `covidData`.

```
    covidData = readr::read_csv(file = "covid_19_clean_complete.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   `Province/State` = col_character(),
##   `Country/Region` = col_character(),
##   Lat = col_double(),
##   Long = col_double(),
##   Date = col_date(format = ""),
##   Confirmed = col_double(),
##   Deaths = col_double(),
##   Recovered = col_double(),
##   Active = col_double(),
##   `WHO Region` = col_character()
```

```
## )
```

What is the difference between a tibble and a dataframe?

A tibble is basically the analagous to a dataframe in the tidyverse world. Personally, I think tibbles are easier to view. It have nice features that normal R dataframes do not have such as:

- printing a tibble prints out the header and first few lines for viewing – useful if you have large datasets
- the headers in a tibble let you know what type of data lives in each column
- tibbles have the capability to use all the tidyverse functions we are going to discuss in this tutorial

For example if I run the following:

```
myCOVIDtibble = read_csv(file = "covid_19_clean_complete.csv")
print(myCOVIDtibble)
```

Now let's convert the tibble into a dataframe so we can see the difference:

```
myCOVIDdf <- as.data.frame(myCOVIDtibble)
print(myCOVIDdf)
```

Similary, if you have a dataframe, you can covert it to a tibble by running the following:

```
myConvertedTibble = as_tibble(myCOVIDdf)
print(myConvertedTibble)
```

Selecting Columns in a Tibble – `select()`

The command to filter tibbles by rows is the `select()` command. You can use the same `&` and `|` commands as you were doing with the dataframe, except now, select() automatically makes it easier to subset by not having long `which()` joined which commands and makes the code easy to follow!

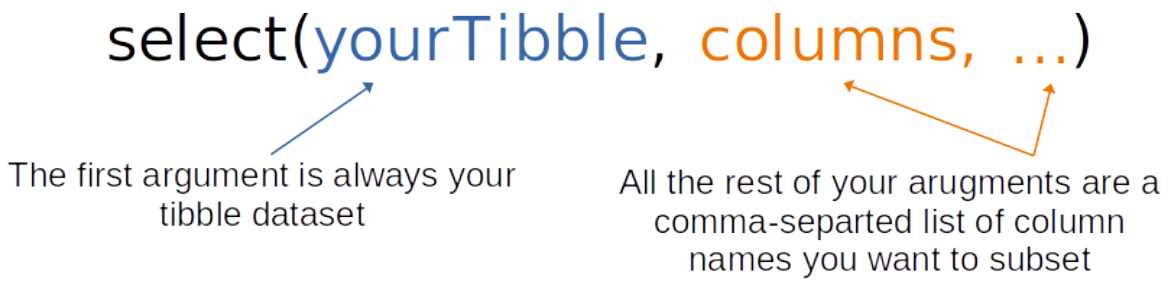


Figure 2: Using the select() command

In the dataset, let's go ahead and only select the column called `Confirmed`, which is the total number of confirmed COVID cases.

```
select(covidData, Confirmed)
```

What if I want to select a range of columns to subset?

Similar to a dataframe, you can still use the `:` symbol. For example, let's say we want to subset our data start from the latitude `Lat` to the `Recovered` variable. I can do that like this:

```
select(covidData, Lat:Recovered)
select(covidData, Lat, Date, Confirmed, Recovered)
```

How do I subset weirdly named columns?!

When I say "weirdly" I mean column names that have spaces or symbols, i.e. does not just contain 0-9 or A-Z values. For example, if we try to subset `Country/Region` we are going to have a problem because '/' is a special character! What do we do? We had backward single quotes:

```
select(covidData, `Country/Region`)
```

Same for anything with a space in the column name:

```
select(covidData, `WHO Region`)
```

Exercises

1. What is returned when running the following code:

   ```
   select(covidData, starts_with("Co"))
   ```

2. What is returned when running the following code:

   ```
   select(covidData, ends_with("n"))
   ```

3. What is returned when running the following code:

   ```
   select(covidData, contains("o"))
   ```

4. What is returned when running the following code:

   ```
   select(covidData, -`WHO Region`)
   ```

5. What is returned when running the following code:

   ```
   select(covidData, -starts_with("L"))
   ```

Selecting Rows in a Tibble – `filter()`

The `filter()` command is used to filter or select rows of a tibble based on an expression. Unlike the `select()` command which takes in column names to subset tibbles to certain columns, the `filter()` command takes in a expression for which the boolean returned is `TRUE`.

As an example, let's go ahead an select all rows of data where the data of occurence is 2020-01-22:

```
filter(covidData, Date == "2020-01-22")
```

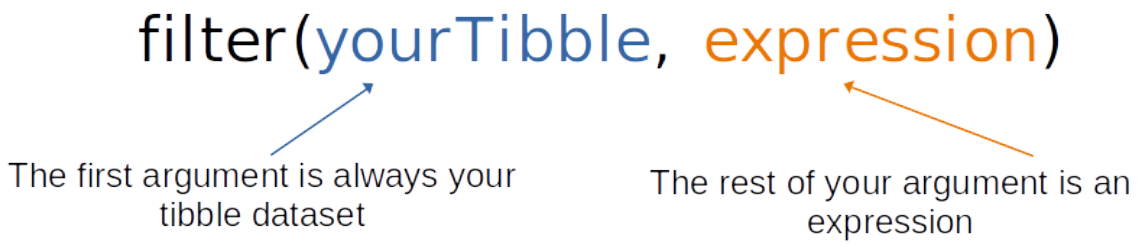


Figure 3: Using the select() command

You will notice the tibble contains 261 rows of data for which the expression `Date == "2020-01-22"` is `TRUE`. You can create much more complex expressions as well. Let's go ahead and select all row there the total confirmed COVID cases are at least 10 and located in Europe:

```
filter(covidData, (Confirmed > 10) & (`WHO Region` == "Europe"))
```

Introducing the %in% operator

Another useful operator in the tidyverse arena is the `%in%` operator. If you are looking to select lots of unique values from a column, instead of writing lots of `&` and `|` commands, you can list them and use `%in%`.

For example, if we wanted to select rows where the World Health Oraganization Region is the Western Pacific, Americas, and Africa. We could do it this way:

```
filter(covidData, (`WHO Region` == "Western Pacific") | (`WHO Region` == "Americas")
| (`WHO Region` == "Africa"))
```

However using the `%in%` operator we can shorten this and make it so that if you look at your code later, it is easier to understand what you are trying to do:

```
filter(covidData, `WHO Region` %in% c("Western Pacific", "Americas", "Africa"))
```

Exercises:

1. Using the `filter()` command, create a tibble where the the WHO Region is located in the Eastern Mediterranean or the Western Pacific and has at least 5 confirmed cases. (Hint: the final answer should have 12,005 rows/enties)

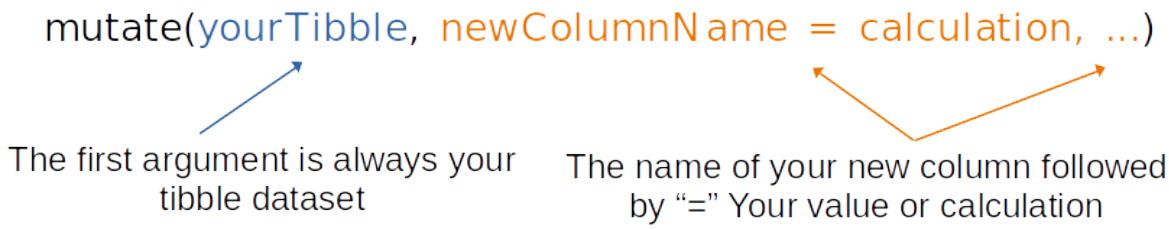
2. Using the `filter()` command, check which days, Alberta (in Province/State column), had at least 20 deaths (Hint: the final answer should have 114 rows/entries).

3. Using the `filter()` command find all the days in which French Guiana (in Province/State Column) had less than 100 active cases (Active column) while also having either least 12 recovered cases or least

1000 confirmed cases. (Hint: the final answer should have 49 rows/entries)

Adding New Data to a Tibble – `mutate()`

Let's say you have a tibble, but you wanted to add additional columns with new calculations or values without overwriting anything in your current tibble. This can easily be accomplished using the `mutate()` command.



In the `covidData` tibble we have a lot of raw values that are cumulative counts from the beginning of the year for each `Country/Region`. The number of active cases and recovered cases don't really make sense to compare since large populations will have large values and small populations will have small values. Instead, let's create two new columns that store those values of the percetage recovered in the population and the percentage active in the population:

```
    covidData <- mutate(covidData, percentRecovered = Recovered/Confirmed * 100,
                        percentActive = Active/Confirmed *100)
```

Notice we created two new columns: `percentRecovered` and `percentActive`. These values will now be added as columns to the `covidData` tibble so we can use them in downstream analysis.

```
    print(covidData)
```

```
## # A tibble: 49,068 x 12
##    `Province/State`    `Country/Region`   Lat   Long Date       Confirmed Deaths
##    <chr>               <chr>            <dbl>  <dbl> <date>         <dbl>  <dbl>
##  1 <NA>                Afghanistan       33.9  67.7  2020-01-22         0      0
##  2 <NA>                Albania           41.2  20.2  2020-01-22         0      0
##  3 <NA>                Algeria           28.0   1.66 2020-01-22         0      0
##  4 <NA>                Andorra           42.5   1.52 2020-01-22         0      0
##  5 <NA>                Angola           -11.2  17.9  2020-01-22         0      0
##  6 <NA>                Antigua and Bar~  17.1 -61.8  2020-01-22         0      0
##  7 <NA>                Argentina        -38.4 -63.6  2020-01-22         0      0
##  8 <NA>                Armenia           40.1  45.0  2020-01-22         0      0
##  9 Australian Capital~ Australia        -35.5 149.   2020-01-22         0      0
## 10 New South Wales     Australia        -33.9 151.   2020-01-22         0      0
## # ... with 49,058 more rows, and 5 more variables: Recovered <dbl>,
## #   Active <dbl>, WHO Region <chr>, percentRecovered <dbl>, percentActive <dbl>
```

Piping and Stringing Data Commands – the %>% operator

Many times in R, you are going to want to run a lot of these commands together. Rather than writing multiple lines of code and creating many variables that clutter your workspace, you can pipe and string commands together into one line of code. This is acheived by using the `%>%` command – called a pipe. Let's look at a basic example where we combine the `select()` command and the `filter()` commands.

Normally, we can do what we just learned above. For this example, first we want all columns execpt the latitude and longitude columns and then we want to filter by only taking World Health Organization Regions that are either African or South-East Asia. We could do this in two steps:

```
ignoreLatLong = select(covidData, -c(Lat, Long))
cleanedSet = filter(ignoreLatLong, `WHO Region` %in% c("Africa", "South-East Asia"))
```

This same set of commands can be written in one line by running the following:

```
cleanedSet <- select(covidData, -c(Lat, Long)) %>% filter(`WHO Region` %in% c("Africa", "South-East As
```

What did the piping do?

By using the `%>%` prior to a tidyverse command, R makes an assumption that the first arugment in each of the command is the output tibble of the previous command. Therefore, you no longer need to specify the first argument! You can string together as many of these as you would like.

Exercises

1. Create a new tibble called `subsetCovid`. Using the pipe operator write one line of code to select columns that are only `Country/Region`, `Deaths`, `Recovered`, and `Confirmed`, followed by rows where `Country/Region` is either Spain, Italy, France, or the United Kingdom. (Hint: the final tibble should have 4,512 row/entries)

2. Run the same one liner above, however, now add another piped command that creates a new column called `percentDeaths` calculated by taking the Deaths divided by Confirmed and multiplying by 100.

Visualizing Data – `ggplot()`

For many R users `ggplot()` is the go-to graphing library to generate pretty figures, graphs, and visualizations and it part of the tidyverse world. In this section we are going to explore a few different types of ways you can plot your data.

The `options...` section: Types of Graphs

These are just some of the basic graph types you can create using `ggplot`. There are many other add on packages you can install and use with ggplot to generate heatmaps, correlations plots, maps, etc... Some that may be of interest. One that many scientists use it the `heatmap()` function. This is a ggplot type, however, it doesn't follow the same `ggplot()` command structure, so just be sure to look up how to use these special use cases.

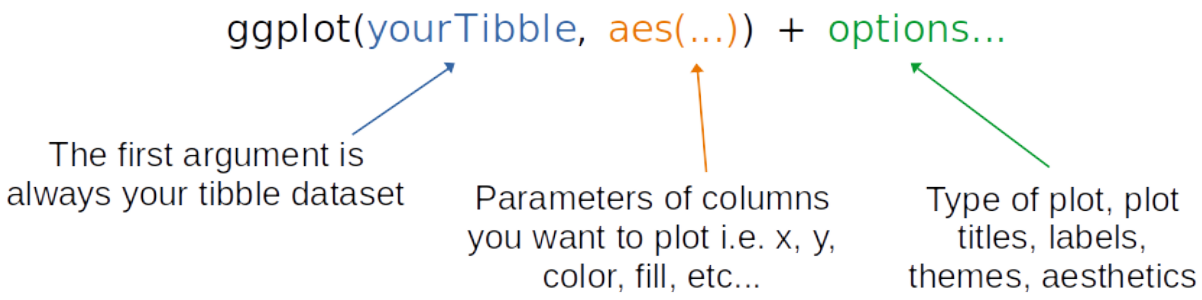


Figure 4: Plotting and visualizing data using the ggplot() command

command	explanation
+ geom_point()	scatter plot
+ geom_jitter()	jitter plot
+ geom_bar()	bar plot
+ geom_boxplot()	boxplot
+ geom_violin()	violin plot
+ geom_histogram()	histogram
+ geom_density()	kernel density plot
+ geom_dotplot()	dot plot
+ geom_line()	line plot

The `options...` section: Adding titles, labels, legends, plot types, themes, color palettes

These are some basic command commands you can add to your `ggplot()` commands to change colors, themes, and add titles/annotations.

command	explanation
+ ggtitle("Title of Your Graph")	add title to your ggplot
+ labs(y="y-axis label", x = "x-axis label")	add labels to x and y axis
+ theme()	ex: +theme(plot.title = element_text(hjust = 0.5))
+ scale_color_manual(values=c("#999999"))	provide a list of color manual colors to set points
+ scale_color_brewer(palette="Dark2")	palette is a set of prefined color palettes in R
+ scale_color_gradient(low="blue", high="red")	autoscale color gradient
+ scale_fill_manual(values=c("#999999"))	fill values manually for boxplots and barplots
+ scale_color_grey()	color points to be greyscale

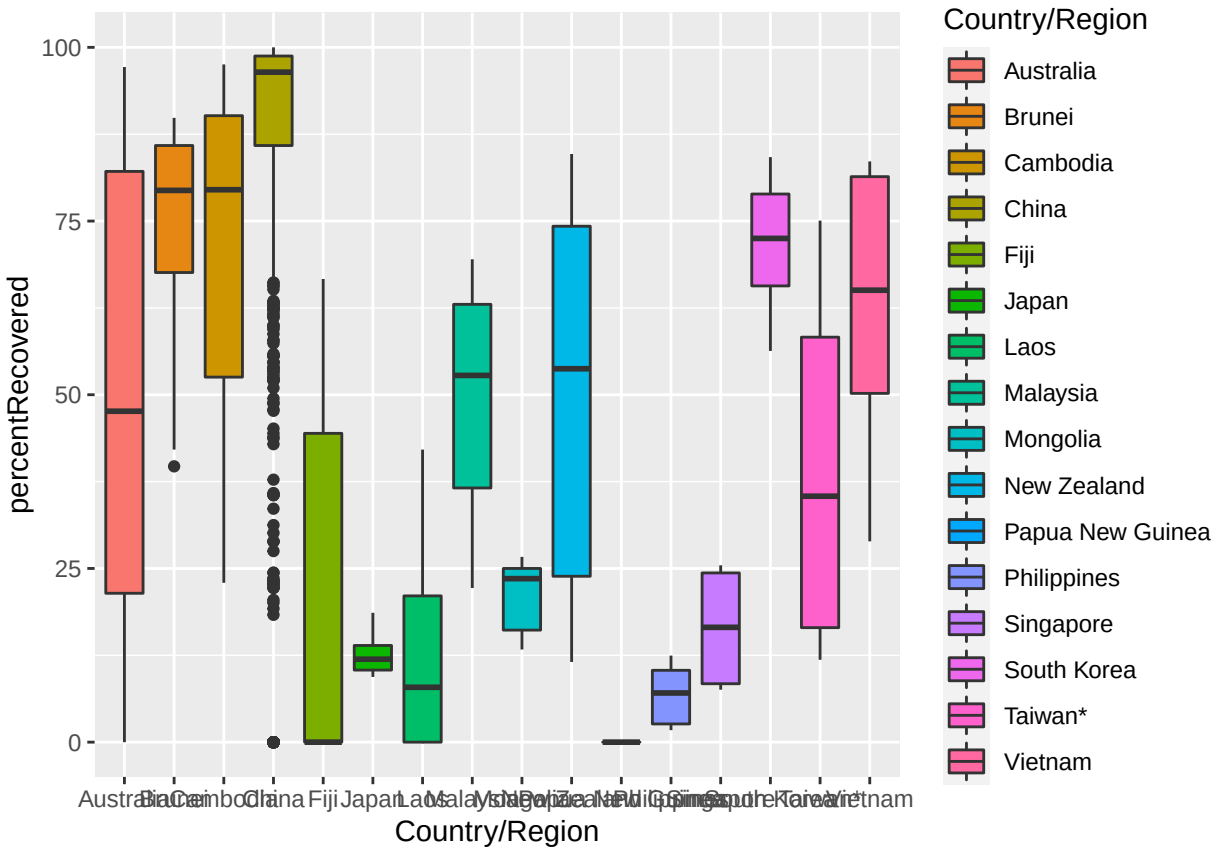
Example: Boxplot

Let's work through a simple boxplot example. First, let's subset the data we want to graph. In this case, let's filter the data so that it only selects data for the full month of April of 2020 and are located in the Western Pacific.

```
newData = filter(covidData, Date >= "2020-04-01" & Date <= "2020-04-30"
                 & `WHO Region` == "Western Pacific")
```

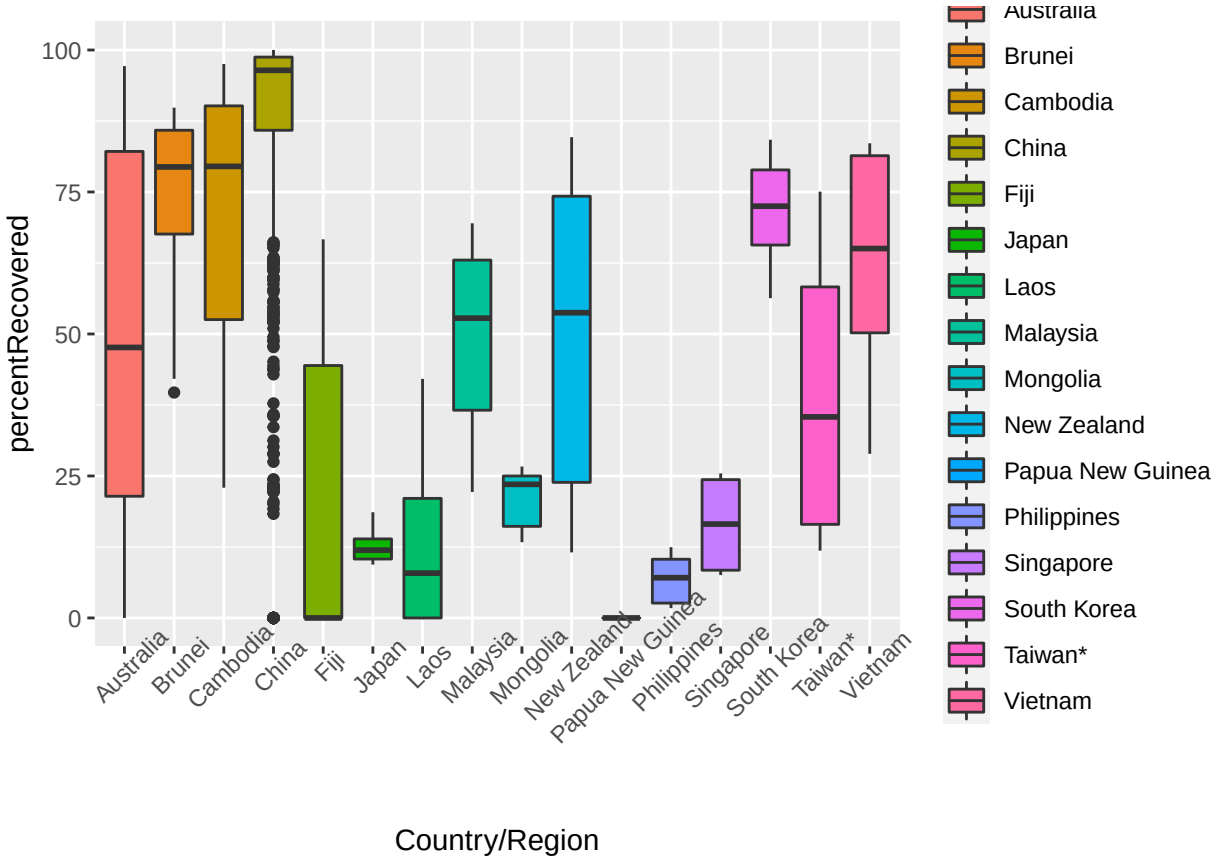
Now, let's use ggplot to create a boxplot with the `Country/Region` plotted on the x-axis and the `percentRecovered` plotted on the y-axis:

```
ggplot(newData, aes(x = `Country/Region`, y = percentRecovered, fill = `Country/Region`)) +
  geom_boxplot()
```



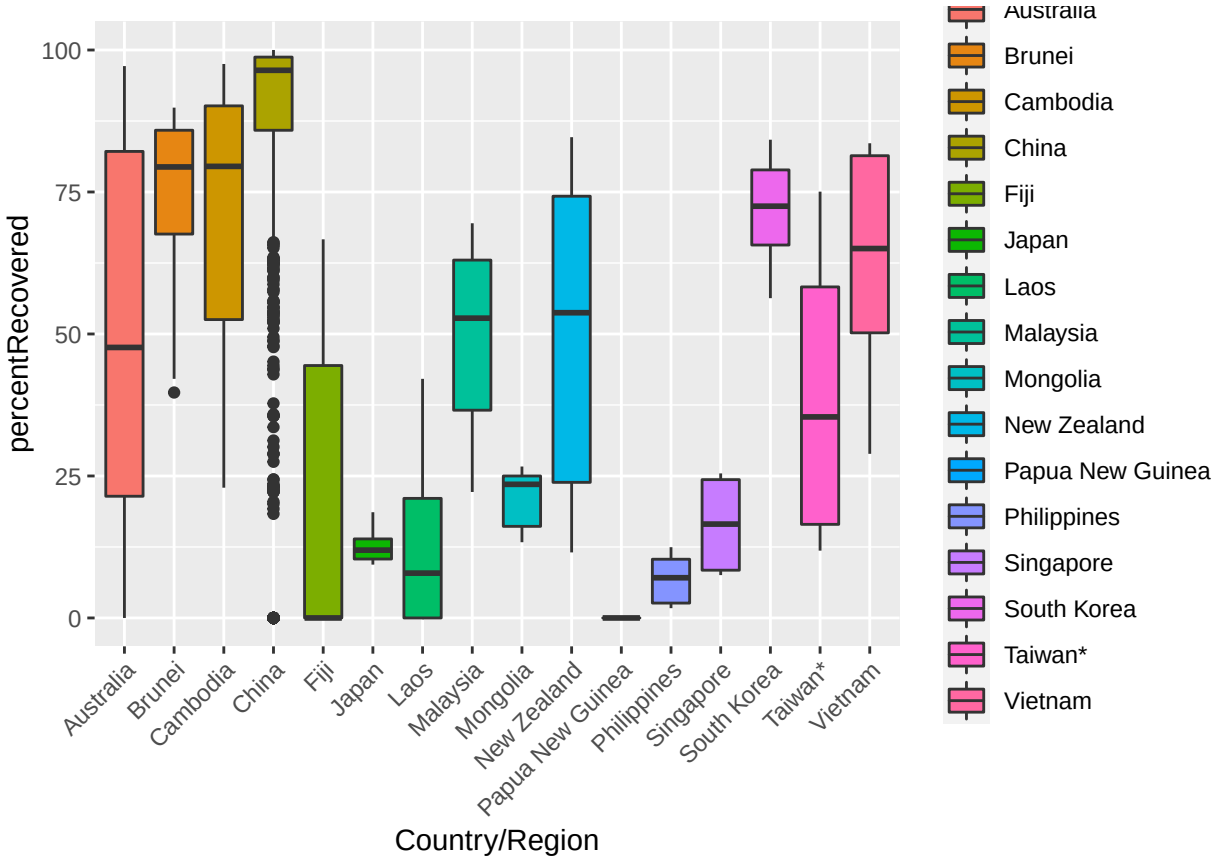
Hmmm, those x-axis labels are hard to read. Can we rotate them? Yes! Using `theme()` using `axis.text.x` keyword:

```
ggplot(newData, aes(x = `Country/Region`, y = percentRecovered, fill = `Country/Region`)) +
  geom_boxplot() + theme(axis.text.x = element_text(angle = 45))
```



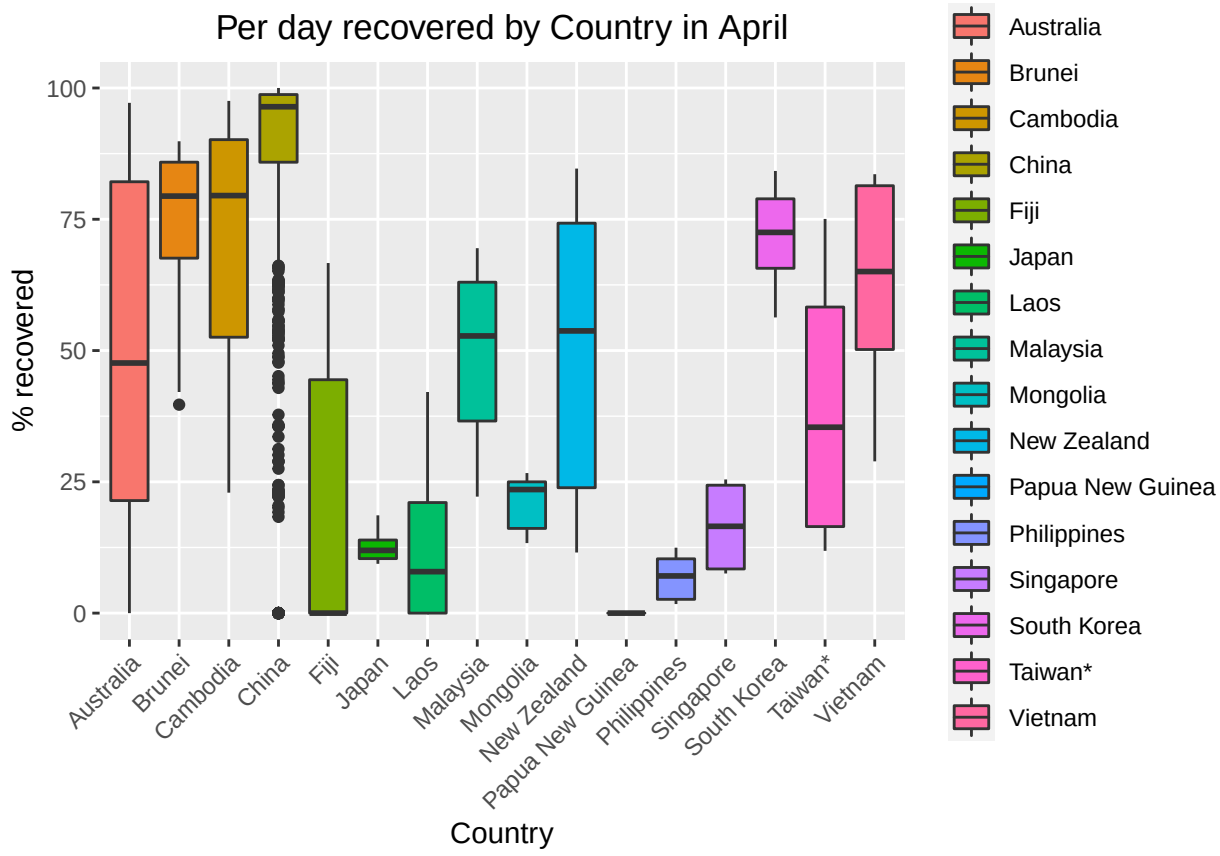
Blah! Now the text is invading the plot space. How can we fix it?

```
ggplot(newData, aes(x = `Country/Region`, y = percentRecovered, fill = `Country/Region`)) +
  geom_boxplot() + theme(axis.text.x = element_text(angle = 45, hjust=1))
```



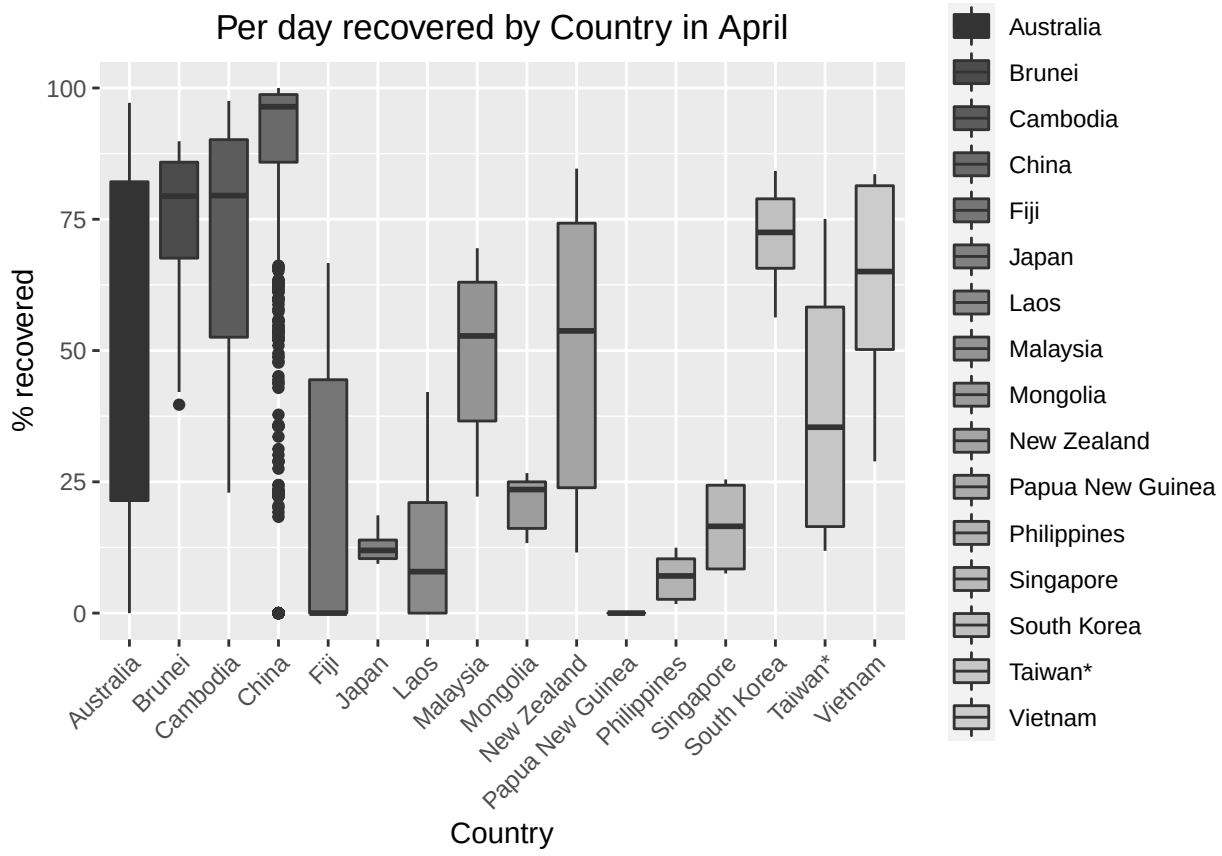
Let's go ahead and now add a title and change the x and y axis labels. You will notice many of these keywords in `theme()` have `hjust` and `vjust` paramters. The are the horizontal and vertical adjustments. Notice for the title, we have `hjust=0.5` meaning the title will be in the center.

```
ggplot(newData, aes(x = `Country/Region`, y = percentRecovered, fill = `Country/Region`)) +
  geom_boxplot() + ggtitle("Per day recovered by Country in April") +
  theme(axis.text.x = element_text(angle = 45, hjust=1),
  plot.title = element_text(hjust = 0.5)) +
  labs(x="Country", y="% recovered")
```



Now let's see if we can change the color scheme:

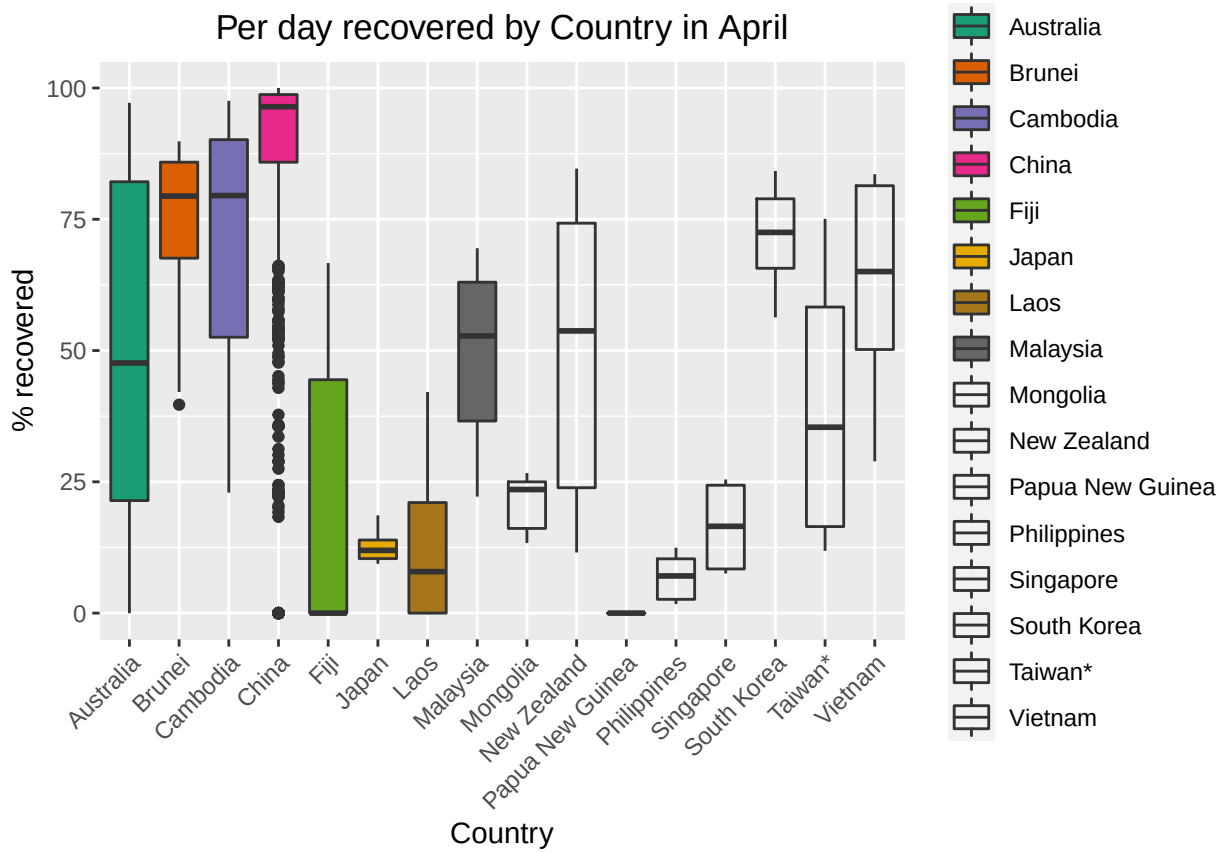
```
ggplot(newData, aes(x = `Country/Region`, y = percentRecovered, fill = `Country/Region`)) +
  geom_boxplot() + ggtitle("Per day recovered by Country in April") +
  theme(axis.text.x = element_text(angle = 45, hjust=1),
  plot.title = element_text(hjust = 0.5)) +
  labs(x="Country", y="% recovered") + scale_fill_grey()
```



Now let's see if we can change the color scheme using a predefined palette in R:

```
ggplot(newData, aes(x = `Country/Region`, y = percentRecovered, fill = `Country/Region`)) +
  geom_boxplot() + ggtitle("Per day recovered by Country in April") +
  theme(axis.text.x = element_text(angle = 45, hjust=1),
  plot.title = element_text(hjust = 0.5)) +
  labs(x="Country", y="% recovered") + scale_fill_brewer(palette="Dark2")
```

```
## Warning in RColorBrewer::brewer.pal(n, pal): n too large, allowed maximum for palette Dark2 is 8
## Returning the palette you asked for with that many colors
```



You will notice a lot of empty boxplots, and that is because if you use a predefined palette and there are fewer colors than there are categories, it runs out of colors to use.

`ggplot()` as a single piped command (Advanced)

Since `ggplot()` follows the same tidyverse format of the first argument always being a tibble, you can actually pipe all of you data into a single long command if you choose. This would look something like this:

```
filter(covidData, Date >= "2020-04-01" & Date <= "2020-04-30"
                  & `WHO Region` == "Western Pacific") %>%
ggplot(newData, aes(x = `Country/Region`, y = percentRecovered, fill = `Country/Region`)) +
  geom_boxplot() + ggtitle("Per day recovered by Country in April") +
  theme(axis.text.x = element_text(angle = 45, hjust=1),
  plot.title = element_text(hjust = 0.5)) +
  labs(x="Country", y="% recovered")
```

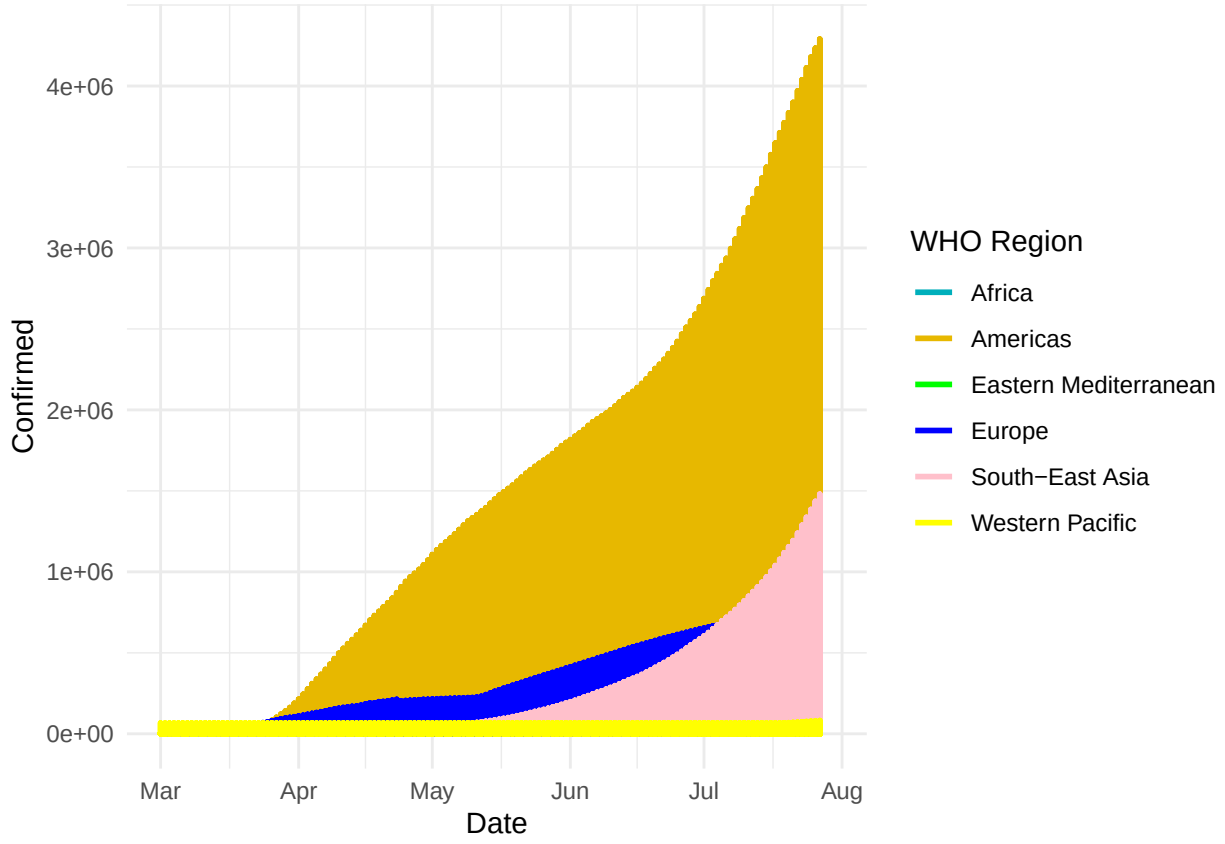
Example: Time Series

Just so we can see and example of how the tidyverse world handle the data type class called `Date`, here is an example of telling `ggplot()` to take our original data and plot `Date` on the x-axis and the total `Confirmed` cases on the y-axis and break down the colors by their World Health Organization classifications:

```
ggplot(covidData, aes(x = Date, y = Confirmed)) +
  geom_line(aes(color = `WHO Region`), size = 1) +
```

```
  scale_color_manual(values = c("#00AFBB", "#E7B800", "green", "blue", "pink", "yellow")) +
  theme_minimal() + scale_x_date(limits = c(as.Date("2020-03-01"), as.Date("2020-07-30")))
```

```
## Warning: Removed 10179 row(s) containing missing values (geom_path).
```



In the example above, you can see we introduce a new `ggplot()` option called `scale_x_date`, which allows you to plot the date between bounds. In my opinion, I would avoid this and just prefilter your dataset to include the date you wants, and then you never have to use this new command.

Excerises:

1. Create a scatter plot, where you first filter the data to only selet the `Country/Region` to be France. Then plot the Confirmed cases along the x-axis, the percent recovered on the y-axis, and the color to be set to the `Province/State`.

2. In the time series example above, add a title and center it along the top. Additionally change the y-axis label to read, "Total Confirmed Cases".

3. Run the boxplot code from an above example, except add an additional option to the end `+ geom_jitter()`. What changes?