Data Wrangling with dplyr

Verbs to Data Wrangle

This is the first section on the collection of R packages included in the **tidyverse**. As noted before, make sure to begin each R session by loading the **tidyverse** package into R:

```
# Load tidyverse
library(tidyverse)
```

We'll use the **tidyverse** here for the purposes of data wrangling: the process of transforming our data into a usable format. In analytics, data wrangling is, roughly, 90% of the job (which basically means it's the job). The reason why so much time must be dedicated to transforming data to another, more appropriate format is because *data never come in the "right" format.* The organizations, entities, algorithms . . . the who- or whatever are behind the collection of the data you work with, don't think of your particular objectives when collecting the data. So you'll be spending lots of time doing the unsexy, but absolutely essential, work of data wrangling.

To give you an idea the problems you'll be facing, import the county-level covid dataset and name it `covid_df`. Since this is a new dataset, use `glimpse()` to see the data types of the variables and its general structure. Notice any problems?

You probably noticed that some vectors, like `vaccine_rate`, are character vectors when they should be numeric vectors. The first values of these vectors are `"."` and not numbers. You'll need convert these vectors to double vectors if you're going to work with them. But, before we can do that, you'll need to first learn some essential data-wrangling tools.

Filter

One of the most common data-wrangling techniques involves subsetting a data frame so that only certain rows remain. This type of subsetting is all the more important when working with big datasets in which, for some given task, much of the data are irrelevant. You need a quick way to separate what's important from what's not and that's what `filter()` does; it filters the dataset for rows that meet specific conditions. Like the other 4 data-wrangling verbs, the first argument of `filter()` is the data frame you want to transform. The second argument is the condition that we want the rows of some variable to satisfy. Here's one generic way to call the `filter()` function:

```
# Generic syntax for filter()
filter(data frame_name, variable_1 >= some_condition,
                        variable_2 == some_condition)
```

Note that we aren't limited to one condition per variable with `filter()`.

As an example, continue working with `covid_df`.[1] Suppose we're only concerned with data from 2020 and only for Tennessee (`statefips == 47`). Then we'll filter the dataset as follows:

```
# Filter rows for 2020 and state of TN
filter(covid_df, year == 2020,
               statefips == 47)
```

[1] If the covid dataset you're working with isn't named "covid," rename it using the assignment operator (`<-`) to conform with the examples in these notes.

A few things to notice:

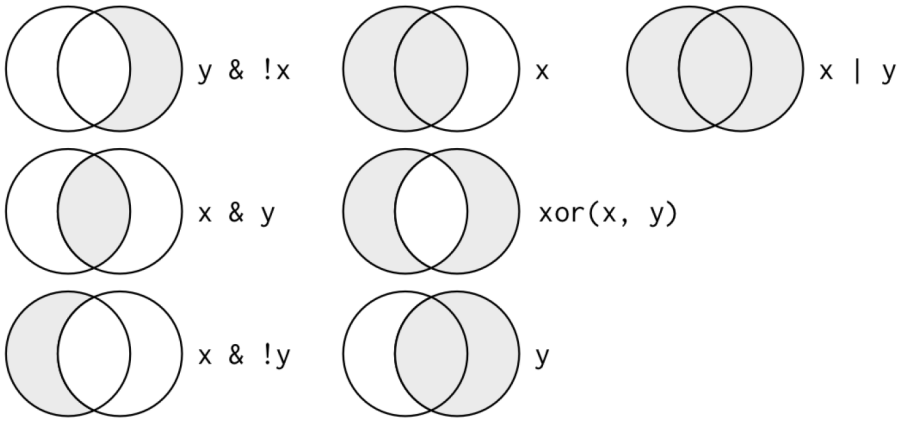
1. The first argument of `filter()` (and all 5 data-wrangling verbs) is a data frame. As a result, you don't call a variable by the usual `data frame$variable` way. R already knows which data frame the variable comes. But remember that when R doesn't know where the variable comes from, you'll need specify the data frame the variable belongs to, followed by `$`and then the variable's name.
2. When stating conditions (e.g., `year == 2020`) use the double-equality sign, `==`. The single `=` has two applications in R, one of which we'll never use.[2] The only use of `=` we need is to specify arguments inside functions, like we do in `ggplot()`, for example, by writing `x = x_variable`. The `==` is used in conditional (true/false) statements. In the previous command any row in which the year was not 2020 was dropped, while every row with the year 2020 was kept.
3. R didn't save the resulting data frame from the previous command; it showed up in the Console. If you want to store the data frame for later use (and you usually will), then you need to assign it a name with `data_frame_name <-` before the `filter()` command.
4. R returned a new type of data frame in the Console. This is called a *tibble* and it's essentially a more general type of data frame because it doesn't can consists of lists and not just vectors.

To effectively filter data, you'll need to make use of *comparison operators* and *logical operators*. Comparison operators are used to return rows that are greater than, less than, not equal, equal and so on, to some magnitude or characteristic. Comparison operators include `>, <, >=, <=, !=, ==`. The comparison operator `!=` means not equal to, while, again, `==` means equal to. The inequalities should be self-explanatory.

Suppose you want to find all observations from Tennessee (`statefips == 47`) or from Kentucky (`statefips == 21`) in the `covid_df` data frame. This time you can't separate these two conditions with a comma. (Try it.) You'll need a logical operator signifying "or," which in R is denoted by a vertical bar, `|`. Here's the code you'll need:

```
# Filter rows for state of TN or state of KY
filter(covid_df, statefips == 47 | statefips == 21)
```

These Venn diagrams depict the logical operators we'll need to use. Note how each is denoted and how it subsets.



Arrange

The `arrange()` function allows you to organize columns in a particular way, which typically means either in ascending or descending order. Compared to the other 4 data-wrangling functions, we'll use `arrange()` the least. But you should know what it does because it will come up again. One example will suffice to show what it does. Using the `covid_df` data frame again,

[2] The single `=` can be used as a weaker version of `<-`, but we'll never need to use it that way.

The `arrange()` function defaults to sorting in an ascending manner. To sort in a descending manner use `desc()` before the variable's name like so:

```
# Sort new_case_count descending
arrange(covid_df, desc(new_case_count))
```

Select

Big datasets typically aren't big just because they have lots of rows (observations); they also tend to come with lots of columns (variables). The `select()` function allows you to subset a big dataset by selecting only those columns you need to work with. The first argument is the data frame you want to subset. (Note the pattern). The other arguments are the variables you want to retain.

To create and save a new data frame called `cases` that consists of just the variables `year`, `month`, `day`, `statefips` and `new_case_count`, run the following code:

```
# Create a data frame called cases with 5 variables
cases <- select(covid_df, year, month, day, statefips, new_case_count)
```

You should verify that `cases` has the same number observations, but only 5 variables in the Environment tab.

Digression: Indexing Columns of a data frame

Because we wanted consecutive columns, we could index the columns to simplify our code. R indexes columns in a data frame just like it indexes the positions of elements in vectors and lists. Since `year` is the 1st column in the `covid_df` data frame, and `new_case_count` is the 5th column, instead of listing every column (as done in the last command), you can simply use `1:5` as the argument to generate the same data frame. Equivalently, you can name the first and last columns of a consecutive grouping of columns as the argument. In the present example, input `year:new_case_count` as the argument and obtain the same data frame. Each of the following two lines of code are equivalent to the lengthier one above:

```
# Create a data frame called cases with 5 variables (by indexing)
cases <- select(covid_df, 1:5)

cases <- select(covid_df, year:new_case_count)
```

To select variables that fall outside the , continue adding them inside the `select()`. For example, note that the `covid_df` data frame has a handful of variables with word "rate" contained in the column name (e.g., `vaccine_rate`, `test_rate`). To create a new data frame consisting of timing information and state information (the first 4 columns) and all columns containing the word "rate," index the first 4 columns as before and use the `contains()` function:

```
# Create df called covid_rate with 1st 4 columns and rate variables
covid_rate <- select(covid_df, 1:4, contains("rate"))
```

Mutate

You'll often need to create new variables based on the information in your dataset; this information gets *mutated* to become a new variable. The function `mutate()` allows you to add a new column. The data frame is the first argument `mutate()` takes, followed by the name of the new column and its definition.

To see how `mutute()` works let's use with the `covid_df` dataset, but let's filter it so that we're only working with Tennessee data (`statefips == 47`).

```
# Filter rows for state of TN; call df covid_tn
covid_tn <- filter(covid_df, statefips == 47)
```

Let's create variable giving the daily percent change in new covid cases (`new_case_count`) by using `mutate()`. This will it another column to the `covid_tn` data frame. First, recall how to calculate a percent change:

$$\text{Percent Change in } x_t = \frac{x_t - x_{t-1}}{x_{t-1}} \times 100,$$

where the t subscript in present case refers to a day because we're calculating daily percent changes.

Before proceeding, glance at the `covid_tn` data frame to confirm the `day` column proceeds from the earliest day to later days. If the ordering were different, then the percent change calculation wouldn't return the daily percent change.

Now, to construct this new variable we'll need to lag `new_case_count` since the previous day's new cases are in the numerator and denominator. To lag a variable, use the `lag()` function putting the variable to be lagged as the argument. It's of utmost importance that, when dealing with complicated formulas full of functions like in the present case, you're cognizant of your brackets. To determine which bracket belongs to which, put your cursor behind the closed bracket, `)`, which then prompts R to highlight its open, `(`, partner. If an open or closed bracket doesn't have a partner, then red icons appear in the left margin of your script with warnings like "unmatched bracket." It's a good idea to do some bracket bookkeeping when your executed code returns errors.

Now, because percent changes are so often used, and also because of the bracket complexity the `lag()` function introduces, it'd be wise to create a function that returns the percent change of *any* variable. We can then use that function to compute the percent change function of `new_case_count`. This function will do it:

```
# Define percent change function
percent_change <- function(x){
                    ((x - lag(x))/lag(x))*100
}
```

Now, instead of `((test_count - lag(test_count))/lag(test_count))*100`, we'll use `percent_change(test_count)` in the definition inside `mutute()`.

Because `mutute()` adds a new column we need to give it a name. Let's call it `test_percent`. Putting all this together, run the following:

```
# Add percent change of new cases to covid_tn
covid_tn <- mutate(covid_tn, test_percent = percent_change(test_count))
```

One inconvenient feature of the `covid_df` data frame is that there's no date variable; a single variable combining the `year`, `month` and `day`. To see why this is inconvenient, plot a time series of `new_cases_count` using the `covid_tn` data frame. Which time variable should you put on the x-axis? Year? Month? Day? This produces the plot using `day`:

```
# Plot of TN covid cases by day
ggplot(data = covid_tn) +
  geom_point(aes(x = day, y = new_case_count))
```

Days of the month range from 1 to 31 and so this plot stacks case counts on a particular day for different months.

What we need is a date variable that, for any given state, is unique. The problem with the `year`, `month` and `day` variables in this dataset is that they reoccur, and so this "stacking problem" manifests itself no matter which variable we put on the x-axis.

To create a unique date variable we need the `make_date()` function found in the **lubridate** package. So let's first install and load **lubridate**, and then add a date variable to the `covid_df` data frame using `mutate()`.

```
# Load lubridate into R
library(lubridate)

# Add vector of dates from year, month, day to covid
covid_df <- mutate(covid_df, date = make_date(year, month, day))
```

To obtain the `covid_tn` data frame with the new `date` variable you'll need to rerun the earlier command that filtered for Tennessee observations (`statefips == 47`). After you've updated `covid_tn`, generate the same plot as before but replacing your previous time variable on the x-axis with `date`.

Summarize

The `summarize()` function reduces a data frame to a single row by returning summary information of chosen variables. When you think of `summarize()`, think *summary statistics*, that is, `mean()`, `median()`, `max()`, `sd()` and so on. By way of example, to compute the mean number of covid tests and cases in Tennessee, run the following:

```
# Find mean number of TN covid tests (remove all na entries)
summarize(covid_tn, mean_test = mean(new_test_count, na.rm = TRUE),
                    mean_case = mean(new_case_count, na.rm = TRUE))
```

Suppose instead that you need to compute the means of *all* variables in the data frame. Listing out each variable would be laborious, but fortunately the `summarize_all()` function does the job. The arguments of `summarize_all()` in this case are the relevant data frame and the summary statistic you want to compute.

```
# Find means of all variables in covid_tn
summarize_all(covid_tn, mean, na.rm = TRUE)
```

The `summarize_all()` function is useful when your entire data frame consists of numeric variables. This is rarely the case, though, as most real-world data will include qualitative variables and quantitative variables for which numerical summaries are meaningless. (Did finding out that the mean year in the `covid_tn` data frame is 2020.221 move our analysis forward?)

A more general approach to finding summary statistics is to use `summarize_at()`. This function is general in 2 ways: It allows you to select which variables to summarize and to select more than one summary statistic. After specifying the data frame, use the `vars()` function to select which variables to summarize and the `funs()` function to select which summary statistics you want computed.

As an example, let's select the variables from `case_count` to `test_count`, and compute their medians and standard deviations. As you'll see, the only cost for greater generality is how we specify the summary statistics inside `funs()`. Since we're computing the median and the standard deviation, each statistic needs to be specified separately as a function (i.e., with brackets: `median()` and `sd()`). And, because you tell R which variables to summarize inside `vars()`, you don't have to restate them inside `median()` and `sd()`. Instead you put a placeholder for the variables, `.`, inside these summary functions. Here's the code:

```
# Find medians and sds of select variables in covid_tn
summarize_at(covid_tn,
             vars(case_count:test_count),
             funs(median(., na.rm = TRUE), sd(., na.rm = TRUE)))
```

The previous examples show how you can quickly extract summary statistics from a dataset. But they fail to demonstrate the power of `summarize()`. So, why should we care about `summarize()`? The importance of `summarize()` is revealed when used in conjunction with the `group_by()` function. To see why, let's create a new data frame, call it `by_month`, from the `covid_tn` data frame, which groups the `year` and `month` columns. I'll explain what "groups" means shortly.

```
# Create df named by_month that groups covid_tn by year and month
by_month <- group_by(covid_tn, year, month)
```

Now, if you compare `by_month` and `covid_tn`, you won't see any apparent difference between the data frames, but there is a fundamental difference. What `group_by()` did was create groupings according to the unique values in the `year` and `month` columns. So, built into the structure of the `by_month` data frame are 2 year groups, 2020 and 2021, since those are the only unique values in the `year` column. There are 12 unique values

for the `month` column, the 12 months of the year, each of which is now its own group in the `by_month` data frame.

Now, let's see what happens when run the same `summarize()` but with the `by_month` data frame.

```
# Find mean number of TN covid tests by month (ignore all NA entries)
summarize(by_month, mean_test = mean(new_test_count, na.rm = TRUE),
                    mean_case = mean(new_case_count, na.rm = TRUE))
```

Grouping by `year` and `month` allows us to obtain average test and case counts for each month in each year. Creating these groups gives us a much better understanding into how tests and cases varied across months. Averaging across the whole dataset, which was what we did originally, returned an overly aggregated summary that was essentially useless because of the variability of tests and cases across the months.

To test your understanding of the `group_by()` function, ask yourself, what if, instead of grouping by `year` and `month`, we only grouped by `year`? Think about this before reading ahead.

Grouping only by `year` would create 2 groups, 2020 and 2021, and thus we could get annual averages. Like the original ungrouped example, this would return insufficiently disaggregated summaries of the real situation.

Also, what if we had grouped the data only by `month`? Think about this before reading ahead.

In this case, the summary average would be completely useless. Why? Because, without the year groups, all observations in, say, April get grouped together. That is, the average number of cases and tests in April would use the cases and tests in April 2020 and April 2021. Grouping by both `year` and `month` generated separate averages for April 2020 and April 2021, which is what we want given the nature of the problem we're dealing with.

The Pipe Operator

Now that you have some practice with the 5 data-wrangling verbs, it's time to learn how to join them together in a single line of code using the pipe operator. The pipe operator allows you to execute multiple operations in a sequence with operations joined by the pipe (denoted by `%>%`). Not only is the pipe operator an efficient way to execute code; it makes code easier to read and understand. Once you get the hang of piping, you'll also start to discern the underlying logic of data transformation in clearer terms.

Okay. So how do we pipe? Let's return to a previous example. Recall taking the mean test and case counts in the `covid_tn` data frame by first grouping the `year` and `month` variables. There were 2 operations executed in 2 separate lines of code:

```
by_month <- group_by(covid_tn, year, month)

summarize(by_month, mean_test = mean(new_test_count, na.rm = TRUE),
                    mean_case = mean(new_case_count, na.rm = TRUE))
```

The pipe operator allows us to collapse these 2 lines of code into 1 line. To do this, start with the data frame you want to transform, which, in the present example, is `covid_tn`. Then use the pipe operator, `%>%`, to group `covid_tn` by `year` and `month`. You'll then introduce the pipe operator again to take the means in the `summarize()` function. Altogether

```
covid_tn %>% group_by(year, month) %>%
             summarize(mean_test = mean(new_test_count, na.rm = TRUE),
                       mean_case = mean(new_case_count, na.rm = TRUE))
```

Some things to notice:

1. A data frame starts the sequence of operations. Every time you use the pipe operator, your code will start with the data frame you want to transform. A piped statement is an order of operations, but it all starts with the data frame you want to manipulate.

2. After telling R the data frame to transform in starting the piped statement, you don't use it again as an argument inside subsequent functions. From above, you know that a data frame is the first argument inside all of the 5 data-wrangling verbs. But this is true only when these verbs are executed in a *non*-piped statement. If these verbs are part of a piped statement, like in the previous example, do not repeat the data frame in the function. The same is true for other functions that require a data frame as an argument (e.g., `group_by()`). The point: By starting your piped statement with a data frame you don't have to remind R which data frame to transform in subsequent commands; R knows. (And R will return errors if you do specify the data frame inside a function.)

3. A data frame always begins a piped statement. But when you want to save the transformed data frame, which is usually the case, you'll start the statement by assigning the data frame a name. For example, if you wanted to store the previous example's data frame, as you did in the original 2 line example, you'll execute:

```
by_month <- covid_tn %>% group_by(year, month) %>%
                  summarize(mean_test = mean(new_test_count, na.rm = TRUE),
                            mean_case = mean(new_case_count, na.rm = TRUE))
```

4. Typing out `%>%` is a pain. Simplify your life and use these shortcuts for the pipe operator:

   Ctrl + Shift + M (**Windows**) or Cmd + Shift + M (**Mac**).

One way to think about the pipe operator is in analogy to the `+` when producing plots with `ggplot()`. After specifying a plot's aesthetics and some geometry, then you can stylize your plot by inserting `+` before every new feature. The analogy between `%>%` and `+` breaks down, though, because data transformation, in contrast to creating plots, often requires the order of transformations be respected. Some transformations can only be done after earlier transformations, while in most plots it doesn't matter if you, say, add a regression line before you stylize the plot's title or after. Still, it can be useful when first using the pipe operator to think of it like the `+` from `ggplot()`.

These notes started by pointing out a problem with the `covid_df`, namely, that R interpreted a handful of its vectors as character vectors because some of their entries consisted of `"."`. Let's use the pipe operator to convert these vectors to double vectors.

Before doing so, let's think about why these `"."` are there in the first place. All the other covid-related vectors have 0s as their first entries. Why not these vaccine vectors, too? As you know, in late January 2020 covid-19 vaccines were not on the radar in the US, as covid-19 itself wasn't on the radar then. You can filter to see when the first vaccine-related data starts trickling in; it's around late February 2021. So the vaccine data are pretty useless until 2021, which further means that entering, for example, 0s in 2020 would be to attribute meaning to data when there isn't any to attribute.

This is important for us, too. You can convert a vector's data type, but what you should you put in place of the `"."`? Putting 0s would, again, attribute meaning to vaccines in the earlier days of the pandemic. By way of example, taking the mean vaccine rate in Tennessee across all observations, with these 0s, would give a very wrong impression about the true vaccine situation. For reasons like this, when you don't have a good idea about how to replace a non-numeric placeholder you should replace it with `NA`.

Fortunately, you don't have to explicitly instruct R to convert the `"."` to `NA` because, by changing the data type of the vector from character to double, any non-numeric entry is automatically replaced with `NA`. So, you can accomplish the task with a single pipe, in our case using `mutate_if()`:

```
# Convert character vectors to doubles
covid_df <- covid_df %>%
  mutate_if(is.character, as.double)
```

Consider another example using the pipe operator. Suppose you need to find the monthly means of case counts and test counts for all states. It's often helpful when crafting piped statements to reverse engineer. So let's work our way from the end of the statement back to the data frame that begins it. So, because you need to compute a mean, a summary statistic, for multiple variables your last command will call the

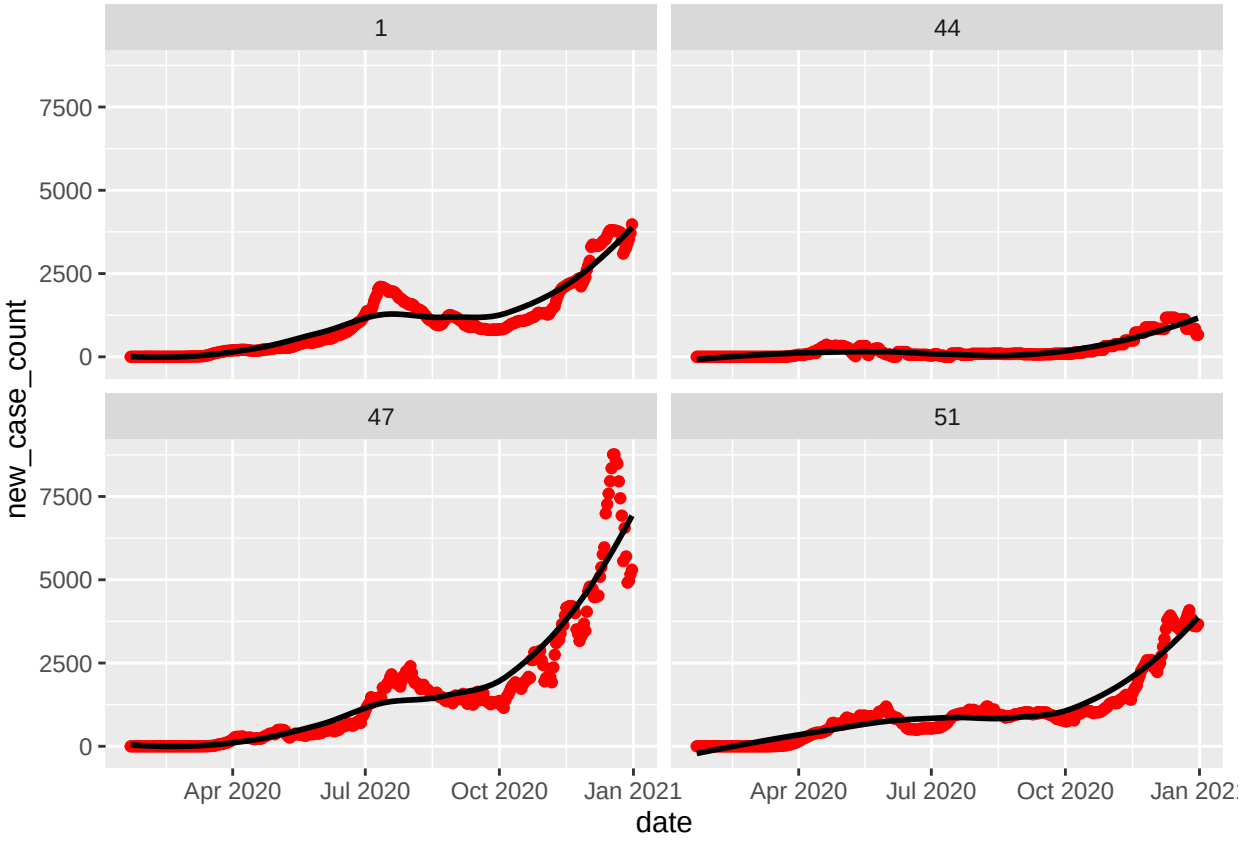
`summarize_at()` function. But now think about the data: It covers all months in 2020 and the first few months of 2021 for every state. You're going to have to insert a `group_by()` command before you compute means. You want means for each state so `statefips` will be one argument of `group_by()`. You also want monthly means, so add `month` as a second argument. Are we finished adding arguments to `group_by`? Recall that the data cover 2020 and 2021, so if we don't also include `year`, observations from, say March 2020 and March 2021 will be group together to form a single average. You want separate averages for each different month, and March 2020 and March 2021, as far as covid cases and tests are concerned, are different months. This is all to say you need to also include `year` as an argument of `group_by()`. Let's save this data frame, so let's begin the piped statement by `covid_means <- covid_df`. Putting all this together we have:

```
covid_means <- covid_df %>%
               group_by(statefips, year, month) %>%
               summarise_at(vars(new_case_count, new_test_count),
                            funs(mean(., na.rm=TRUE)))
```

Let's unlock further possibilities with the pipe operator. Below is some code that will generate a plot. Before running it, first read it and try to determine what the code will produce.

```
covid_df %>%
  filter(year != 2021 & (statefips == 1 | statefips == 44 |
                         statefips == 47 | statefips == 51)) %>%
  ggplot(aes(x = date, y = new_case_count)) +
    geom_point(color = "red") +
    geom_smooth(method = loess, se = FALSE, color = "black") +
    facet_wrap(~ statefips, nrow = 2)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



A plot of some state's average monthly new case counts:

```
covid_df %>%
  group_by(statefips, year, month) %>%
  summarize(mean_cases = mean(new_case_count, na.rm=TRUE)) %>%
  filter(year != 2021 & (statefips == 1 | statefips == 44 |
         statefips == 47 | statefips == 48)) %>%
  ggplot(aes(x = as.factor(month), y = mean_cases, color = as.factor(statefips))) +
    geom_point() +
    theme(legend.position = "bottom")
```

```
## `summarise()` has grouped output by 'statefips', 'year'. You can override using the `.groups` argumen
```