Notes 3 - The Tidyverse and ggplot2

Rick Brown
Southern Utah University

Math 3190

Section 1

Tidyverse

Introduction

Modern **R** users are migrating away from many base **R** packages and functions to instead work in the **tidyverse**.

The tidyverse is both a philosophy for coding and organizing data as well as a collection of packages in **R**.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.4.4     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ----------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/
```

Tidy Format (murders data)

We say that a data table is in **tidy** format if each row represents one observation and columns represent the different variables available for each of these observations. For example, the following data set is in tidy format:

```r
library(dslabs)
data(murders)
head(murders)
```

```
##         state abb region population total
## 1     Alabama  AL  South    4779736   135
## 2      Alaska  AK   West     710231    19
## 3     Arizona  AZ   West    6392017   232
## 4    Arkansas  AR  South    2915918    93
## 5 California  CA   West   37253956  1257
## 6    Colorado  CO   West    5029196    65
```

Not Tidy Format (fertility)

The following dataset is organized, but not tidy. Why?

```
##       country 1960 1961 1962
## 1     Germany 2.41 2.44 2.47
## 2 South Korea 6.16 5.99 5.79
```

Tidy Format (fertility)

Here is what the data would look like in tidy format:

```
##        country year fertility
## 1      Germany 1960      2.41
## 2 South Korea 1960      6.16
## 3      Germany 1961      2.44
## 4 South Korea 1961      5.99
## 5      Germany 1962      2.47
## 6 South Korea 1962      5.79
```

The same information is provided, but there are important differences in the format. For the **tidyverse** packages to be optimally used, data need to be reshaped into 'tidy' format. The advantage of working in tidy format allows the data analyst to focus on more important aspects of the analysis rather than the format of the data.

Tibbles

A **tibble** is a modern version of a data.frame.

```
library(tidyverse)
dat1 <- tibble(x = 1:4, y = 5:8, z = c("A", "B", "C", "D"))
```

Or convert a data frame to a tibble

```
dat <- data.frame(x=1:4, y = 5:8, z = c("A", "B", "C", "D"))
dat1 <- as_tibble(dat)
dat1
```

```
## # A tibble: 4 x 3
##       x     y z
##   <int> <int> <chr>
## 1     1     5 A
## 2     2     6 B
## 3     3     7 C
## 4     4     8 D
```

Tibbles

Important characteristics that make tibbles unique:

1. Tibbles are primary data structure for the `tidyverse`
2. Tibbles display better and printing is more readable
3. Tibbles can be grouped
4. Subsets of tibbles are tibbles
5. Tibbles can have complex entries–numbers, strings, logicals, lists, functions, other tibbles, etc.

Subsetting Tibbles

Note: tibbles work just like data frames in just about every way except one. With data frames, using brackets [] will give vectors and with tibbles, using [] will give tibbles.

```
class(dat[,1])  # Class of first column from a data frame
```

```
## [1] "integer"
```

```
class(dat1[,1]) # Class of first column from a tibble
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
mean(dat[,1])
```

```
## [1] 2.5
```

```
mean(dat1[,1])
```

```
## Warning in mean.default(dat1[, 1]): argument is not
## numeric or logical: returning NA
## [1] NA
```

Subsetting Tibbles

We can choose the columns of a tibble as a vector using the $ operator or by putting double brackets `[[]]`.

```
dat1$x    # Gives the first column (whose name is x)
```

```
## [1] 1 2 3 4
```

```
dat1[[1]] # Gives the first column
```

```
## [1] 1 2 3 4
```

```
mean(dat1[[1]])
```

```
## [1] 2.5
```

This subsetting is not a problem for rows. Rows of data frames are data frames and rows of tibbles are tibbles, so nothing of note changes there.

Data Import in the Tidyverse

The `tidyverse` has its own functions that will read in data sets as tibbles. They are the functions

- `read_csv()`
- `read_table()`

These work very much like the `read.csv()` and `read.table()` functions. The primary difference is that `read.csv()` and `read.table()` read in the data has a data frame whereas `read_csv()` and `read_table()` read in the data as a tibble.

The `read_excel()` function in the `readxl` library also reads in data files as tibbles even though the `readxl` library is not technically part of the tidyverse.

dplyr Functions

One of the most useful packages in the tidyverse is the **dplyr** package that is used for data wrangling. dplyr is called that since it is a tool (like a set of pliers) for data frames (or tibbles).

The dplyr package has the following useful functions:

- mutate() adds new variables that are functions of existing variables.
- filter() picks cases based on their values. Selects rows.
- select() picks variables based on their names. Selects columns.
- summarize() or summarise() reduces multiple values down to a single summary.
- arrange() changes the ordering of the rows.
- group_by() allows you to perform any operation "by group"

Note an important point: most dplyr functions (and most functions in the tidyverse) input a tibble and then output a modified tibble, although many can also work with data frames.

Mutate

The function **mutate** takes the data frame or tibble, the instructions for the new columns in next arguments, and returns a modified data frame. For example:

```
murders <- as_tibble(murders)
head(murders)
```

```
## # A tibble: 6 x 5
##   state      abb   region population total
##   <chr>      <chr> <fct>       <dbl> <dbl>
## 1 Alabama    AL    South     4779736   135
## 2 Alaska     AK    West       710231    19
## 3 Arizona    AZ    West      6392017   232
## 4 Arkansas   AR    South     2915918    93
## 5 California CA    West     37253956  1257
## 6 Colorado   CO    West      5029196    65
```

Mutate

To add murder rates, we mutate as follows:

```
murdersRate <- mutate(murders,
  rate = total / population * 100000
)
head(murdersRate)
```

```
## # A tibble: 6 x 6
##   state      abb   region population total  rate
##   <chr>      <chr> <fct>       <dbl> <dbl> <dbl>
## 1 Alabama    AL    South     4779736   135  2.82
## 2 Alaska     AK    West       710231    19  2.68
## 3 Arizona    AZ    West      6392017   232  3.63
## 4 Arkansas   AR    South     2915918    93  3.19
## 5 California CA    West     37253956  1257  3.37
## 6 Colorado   CO    West      5029196    65  1.29
```

Filter

Now suppose that we want to filter the data table to only show the entries for which the murder rate is lower than 0.71. We do this as follows:

```
filter(murdersRate, rate <= 0.71)
murdersRate[murdersRate$rate <= 0.71,] # How do get the same
                                       # result without
                                       # filter function
```

```
## # A tibble: 5 x 6
##   state         abb   region        population total  rate
##   <chr>         <chr> <fct>              <dbl> <dbl> <dbl>
## 1 Hawaii        HI    West             1360301     7 0.515
## 2 Iowa          IA    North Central    3046355    21 0.689
## 3 New Hampshire NH    Northeast        1316470     5 0.380
## 4 North Dakota  ND    North Central     672591     4 0.595
## 5 Vermont       VT    Northeast         625741     2 0.320
```

Select

If we want to view just a few of our columns, we can use the following:

```
murdersRate <- mutate(murders, rate = total / population * 100000)
murdersRateSelect <- select(murdersRate, state, rate)
filter(murdersRateSelect, rate <= 0.71)
# The above is the same as the following
murdersRate[murdersRate$rate <= 0.71, c("state", "rate")]
```

```
## # A tibble: 5 x 2
##   state          rate
##   <chr>         <dbl>
## 1 Hawaii        0.515
## 2 Iowa          0.689
## 3 New Hampshire 0.380
## 4 North Dakota  0.595
## 5 Vermont       0.320
```

Nesting Functions

Instead of defining new objects along the way, we could do everything in one complex nested function:

```
filter(select(mutate(murders, rate = total / population * 100000),
              state, rate), rate <= 0.71)
```

```
## # A tibble: 5 x 2
##   state          rate
##   <chr>         <dbl>
## 1 Hawaii        0.515
## 2 Iowa          0.689
## 3 New Hampshire 0.380
## 4 North Dakota  0.595
## 5 Vermont       0.320
```

This is fairly concise but a little confusing. Is there a better, clearer way?

Pipes

In the previous example, we performed the following wrangling operations:

original data $\rightarrow$ mutate $\rightarrow$ select $\rightarrow$ filter

We can perform a series of operations in **R** by sending the results of one function to another using the **pipe operator**: `|>` that was added in **R** version 4.1.

There is also a pipe (that was actually added first) in the `magrittr` package that is loaded with the tidyverse with the syntax `%>%`. This `magrittr` pipe can do a few things the native pipe cannot[1], but for the vast majority of cases they work the same, so it is recommended to use the native pipe since it is built-in and runs slightly faster.

[1] https://magrittr.tidyverse.org

Pipes

The pipe is a combination of characters that when used properly does two things: *It shortens and simplifies the code* and it makes the code more intuitive to read.

There is a keyboard shortcut in RStudio for inserting the pipe. While you can always just type |>, you can also type:

Mac: Command-Shift-M
Windows: Control-Shift-M

However, this will not give you the default pipe unless you change a setting in RStudio. Go to
Tools → Global Options → Code → Select "Use native pipe operator".

Pipes

All the pipe does is provide **forward application** of an object to the first argument of a function. The pipe sends left side of the input to the function to the right of the pipe. For example, if we wanted to calculate

$$\log_2(\sqrt{16})$$

We could use:

```
16 |> sqrt() |> log2()
```

```
## [1] 2
```

Since the pipe sends values to the first argument, we can define other arguments as follows:

```
16 |> sqrt() |> log(base = 2)
```

```
## [1] 2
```

While piping works the way it is formatted above, it is better practice to use a new line after each pipe.

Pipes (murders)

Creating the prior tibble operation using pipes:

```
murders |>
  mutate(rate = total / population * 100000) |>
  select(state, rate) |>
  filter(rate <= 0.71)
```

```
## # A tibble: 5 x 2
##   state          rate
##   <chr>         <dbl>
## 1 Hawaii        0.515
## 2 Iowa          0.689
## 3 New Hampshire 0.380
## 4 North Dakota  0.595
## 5 Vermont       0.320
```

Piping into Other Arguments

By default, pipes will send the object being piped to the first argument of the next command, but we can send it to another argument by using the underscore (_) placeholder and specifying the argument.

```
murdersRate |>
  lm(rate ~ population, data = _)
```

```
##
## Call:
## lm(formula = rate ~ population, data = murdersRate)
##
## Coefficients:
## (Intercept)   population
##   2.575e+00    3.363e-08
```

We can also use the pipe placeholder along with $, [], and [[]]:

```
murdersRate |> _$rate |> head(5)
```

```
## [1] 2.824424 2.675186 3.629527 3.189390 3.374138
```

Arrange

We know about the **order** and **sort** functions, but for ordering entire tables, the **arrange** function is much more useful. For example, here we order the tibble by the state's murder rate:

```
murdersRate |>
  arrange(rate) |>
  head()
```

```
## # A tibble: 6 x 6
##   state         abb   region        population total  rate
##   <chr>         <chr> <fct>              <dbl> <dbl> <dbl>
## 1 Vermont       VT    Northeast         625741     2 0.320
## 2 New Hampshire NH    Northeast        1316470     5 0.380
## 3 Hawaii        HI    West             1360301     7 0.515
## 4 North Dakota  ND    North Central     672591     4 0.595
## 5 Iowa          IA    North Central    3046355    21 0.689
## 6 Idaho         ID    West             1567582    12 0.766
```

Arrange (descending order)

Note that the default behavior is to order in ascending order. The function **desc** transforms a vector so that it is in descending order. To sort the table in descending order, we can type:

```
murdersRate |>
  arrange(desc(rate)) |>
  head()
```

```
## # A tibble: 6 x 6
##   state                abb   region        population total  rate
##   <chr>                <chr> <fct>              <dbl> <dbl> <dbl>
## 1 District of Columbia DC    South             601723    99 16.5
## 2 Louisiana            LA    South            4533372   351  7.74
## 3 Missouri             MO    North Central    5988927   321  5.36
## 4 Maryland             MD    South            5773552   293  5.07
## 5 South Carolina       SC    South            4625364   207  4.48
## 6 Delaware             DE    South             897934    38  4.23
```

Nested sorting

If we are ordering by a column with ties, we can use a second (or third) column to break the tie. for example:

```
murdersRate |>
  arrange(region, rate) |>
  head()
```

```
## # A tibble: 6 x 6
##   state         abb   region    population total  rate
##   <chr>         <chr> <fct>          <dbl> <dbl> <dbl>
## 1 Vermont       VT    Northeast     625741     2 0.320
## 2 New Hampshire NH    Northeast    1316470     5 0.380
## 3 Maine         ME    Northeast    1328361    11 0.828
## 4 Rhode Island  RI    Northeast    1052567    16 1.52
## 5 Massachusetts MA    Northeast    6547629   118 1.80
## 6 New York      NY    Northeast   19378102   517 2.67
```

Summarize

The **summarize** function computes summary statistics in an intuitive way. The 'heights' dataset includes heights and sex reported by students in an in-class survey.

```
data(heights)
heights |>
  filter(sex == "Female") |>
  summarize(
    avg = mean(height),
    std_dev = sd(height)
  )
```

```
##          avg  std_dev
## 1 64.93942 3.760656
```

Group then summarize with group_by()

A common operation in data exploration is to first split data into groups and then compute summaries for each group. For example, we may want to compute the average and standard deviation for men's and women's heights separately. We can do the following

```
heights |>
  group_by(sex) |>
  summarize(
    average = mean(height),
    standard_deviation = sd(height)
  )
```

```
## # A tibble: 2 x 3
##   sex    average standard_deviation
##   <fct>    <dbl>              <dbl>
## 1 Female    64.9               3.76
## 2 Male      69.3               3.61
```

pivot_longer() Function

Sometimes it is the case that the data need to be manually put into tidy format. This is where the pivot_longer() function can help.

```r
prices <- read.csv("data/houseprice.txt")
head(prices, 10)
```

```
##    gainesville orlando tampa
## 1        173.0   243.9 230.7
## 2        145.5   201.1 115.7
## 3        190.6   185.3 211.0
## 4        186.3   187.5 203.5
## 5        248.7   207.9 149.9
## 6        206.4   234.8 166.8
## 7         86.8   253.2 134.1
## 8        204.6   144.7 214.2
## 9        174.5      NA 105.5
## 10       220.0      NA 216.2
```

pivot_longer() Function

```
new_prices <- pivot_longer(prices, cols = everything())
head(new_prices)
```

```
## # A tibble: 6 x 2
##   name        value
##   <chr>       <dbl>
## 1 gainesville  173
## 2 orlando      244.
## 3 tampa        231.
## 4 gainesville  146.
## 5 orlando      201.
## 6 tampa        116.
```

This looks much better! Except the column names are just set to the default of `name` and `value`. Also, there are some NA values as we saw in the original data set.

pivot_longer() Function

```
house_prices <- pivot_longer(prices, cols = everything()) |>
  na.omit() |>
  rename(city = name, price = value) |>
  arrange(city)
head(house_prices)
```

```
## # A tibble: 6 x 2
##   city        price
##   <chr>       <dbl>
## 1 gainesville  173
## 2 gainesville  146.
## 3 gainesville  191.
## 4 gainesville  186.
## 5 gainesville  249.
## 6 gainesville  206.
```

pivot_wider() Function

It's relatively rare to need `pivot_wider()` to make tidy data, but it can be useful for creating summary tables for presentation, or data in a format needed by other tools. We won't focus too much on `pivot_wider()`, but here is an example. We do need an "id" or a "row number" variable to make this work.

```
price_wide <- house_prices |>
  mutate(row_num = c(1:11, 1:8, 1:10)) |>
  pivot_wider(names_from = city, values_from = price)
head(price_wide, 3)
```

```
## # A tibble: 3 x 4
##   row_num gainesville orlando tampa
##     <int>       <dbl>   <dbl> <dbl>
## 1       1         173    244.  231.
## 2       2         146.   201.  116.
## 3       3         191.   185.  211
```

More on the tidyverse

There are some other tidyverse operations, including the `inner_join()`, `left_join()`, `right_join()`, `full_join()`, `pull()`, `dot()`, `reframe()`, `nest_by()`, and `pick()` functions. We will work with a few of these throughout the course.

Section 2

Graping with `ggplot2`

ggplot2 Introduction

Note: these slides were adapted from slides created by Aubrey Odom.

While knowing how to plot using the base **R** packages is important, many **R** users are using the `ggplot2` package (which is part of the tidyverse) more and more for making better-looking plots.

Advantages of ggplot2

- It's consistent! gg = "grammar of graphics"; easy base system for adding/removing plot elements, with room for being fancy too
- Very flexible
- Themes available to polish plot appearance
- Active maintenance/development = getting better all the time!
- It can do quick-and-dirty and complex, so you only need one system
- Plots, or whole parts of plots, can be saved as objects
- Easy to add complexity or revert to earlier plot

Introduction

Disadvantages of ggplot2

- Sometimes more complicated than base **R** plotting
- Difficult to work with in iterated functions
- No 3-D graphics
- ggplot is often slower than base graphics
- The default colors can be difficult to change
- You might need to change the structure of your data frame to make certain plots (use `tidyr::pivot_longer()`)

ggplot Basics

There are three primary components to plotting with `ggplot2`:

- The **data** component. This is what data set and variables we are actually plotting.
- The **geometry** component. This describes what it is we are plotting. Examples include barplots, scatter plots, histograms, smooth densities, qqplots, boxplots, etc.
- The **aesthetic mapping** or just the **mapping**. The two most important cues in this plot are the point positions on the x-axis and y-axis. Each point represents a different observation, and we map data about these observations to visual cues like x- and y-scale. Color is another visual cue that we map to region. How this is defined depends on what type of geometry we are using.

Example dataset: Diamonds

```
install.packages("ggplot2")
```

```
library(ggplot2)
```

Variable	Description	Values
price	price in US dollars	$326-$18,823
carat	weight of the diamond	0.2-5.01
cut	quality of the cut	Fair, Good, Very Good, Premium, Ideal
color	diamond color	J (worst) to D (best)
clarity	measurement of how clear the diamond is	I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best)
x	length in mm	0-10.74
y	width in mm	0-58.9
z	depth in mm	0-31.8
depth	total depth percentage	43-79
table	width of top of diamond relative to widest point	43-95

Figure 1: Diamonds

Example dataset: Diamonds

```
head(diamonds)
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl>
## 1  0.23 Ideal     E     SI2      61.5    55   326  3.95
## 2  0.21 Premium   E     SI1      59.8    61   326  3.89
## 3  0.23 Good      E     VS1      56.9    65   327  4.05
## 4  0.29 Premium   I     VS2      62.4    58   334  4.2
## 5  0.31 Good      J     SI2      63.3    58   335  4.34
## 6  0.24 Very Good J     VVS2     62.8    57   336  3.94
## # i 2 more variables: y <dbl>, z <dbl>
```

Example dataset: Diamonds

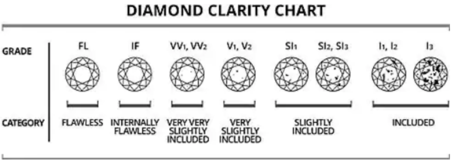


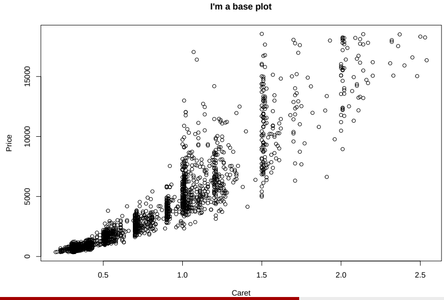
Figure 2: Diamond clarity is a measure of the purity and rarity of the stone, graded by the visibility of these characteristics under 10-power magnification. A stone is graded as flawless if, under 10-power magnification, no inclusions (internal flaws) and no blemishes (external imperfections) are visible.

- The dataset contains information about 53,940 round-cut diamonds
- There are 10 variables measuring various pieces of information about the diamonds.
- There are 3 variables with an ordered factor structure: cut, color, & clarity

Example in Base Plotting

There is essentially just one primary function to know: `ggplot()`. However, `ggplot()` needs lots of other basic functions.

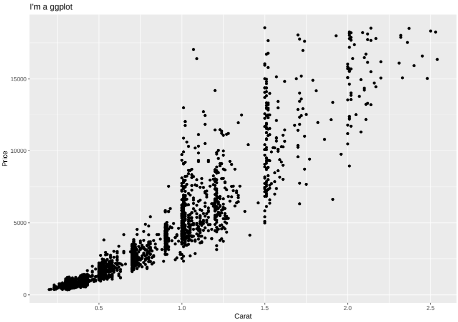
```
# load the diamonds dataset and take a sample of 2000
data(diamonds); set.seed(2023)
diam <- diamonds[sample(1:53940,2000),]
plot(diam$carat, diam$price, main = "I'm a base plot",
     xlab = "Caret", ylab = "Price")
```



Example in ggplot2

In a ggplot, we need to begin with the ggplot() function and then add on (literally with a + sign) to that plot using other commands. In this case, I put geom_point() to add those solid dots.

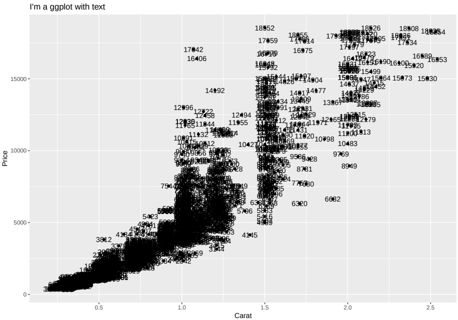
```
ggplot(data = diam) +
  geom_point(aes(x = carat, y = price)) +
  ggtitle("I'm a ggplot") + labs(x = "Carat", y = "Price")
```



Example in ggplot2

Here is an example adding text to the plot.

```
ggplot(data = diam) +
  geom_point(aes(x = carat, y = price)) +
  geom_text(aes(x = carat, y = price, label = price)) +
  ggtitle("I'm a ggplot with text") +
  labs(x = "Carat", y = "Price")
```



Global vs Local Aesthetic Mapping

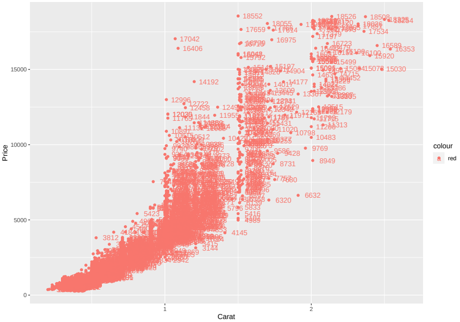
Instead of putting the `x` and `y` in the `geom_()` function, we can put it in the `ggplot()` and it will apply everywhere.

Anything put into the `ggplot()` function will apply globally to the entire plot whereas anything put into the geometry will only apply to that geometry. Some options, like size, can only be put into the geometry.

Example in ggplot2

This will make both the points and text red. The `nudge_x` option will move the text to the right 0.1 units.

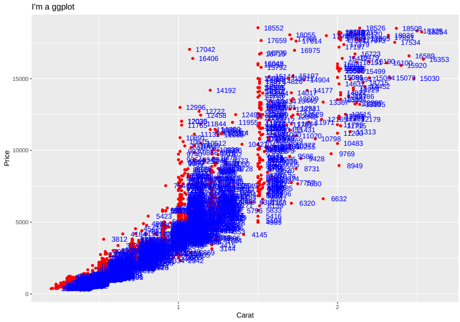
```
ggplot(data = diam, aes(x = carat, y = price, color = "red"))
  geom_point() +
  geom_text(aes(label = price), nudge_x = 0.1) +
  labs(x = "Carat", y = "Price")
```



Example in ggplot2

This will make the points red, but the text blue.

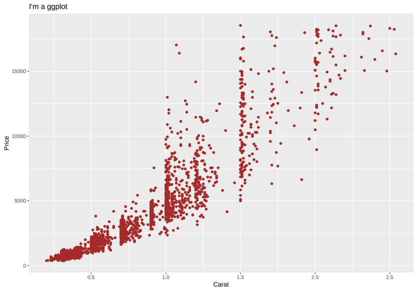
```
ggplot(data = diam, aes(x = carat, y = price)) +
  geom_point(color = "red") +
  geom_text(aes(label = price), nudge_x = 0.1, color="blue") +
  ggtitle("I'm a ggplot") +
  labs(x = "Carat", y = "Price")
```



Piping in ggplot2

Pipes work very well with ggplot also. Remember that pipes are a part of the tidyverse (in thr dplyr package) and by default, piping puts the thing being piped into the first argument of the function.

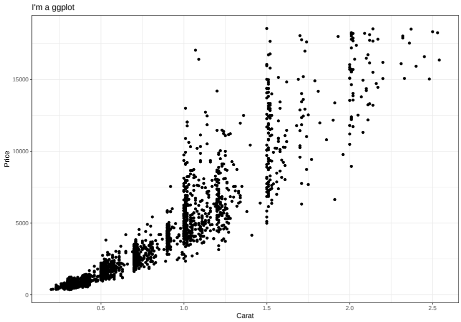
```
library(tidyverse)
diam |>
  ggplot(aes(carat, price)) + geom_point(col = "brown") +
  ggtitle("I'm a ggplot") + labs(x = "Carat", y = "Price")
```



Example in ggplot2

We can change the background using `theme_...()`. There are `theme_bw()`, `theme_dark()`, `theme_classic()`, and more.

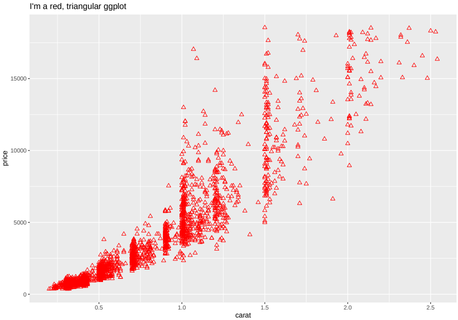
```
ggplot(data = diam, aes(x = carat, y = price)) +
  geom_point() + ggtitle("I'm a ggplot") +
  labs(x = "Carat", y = "Price") + theme_bw()
```



Changing the Graph Options in ggplot2

We can change the type of points added in the geom_point() function.

```
ggplot(data = diam, aes(x = carat, y = price)) +
  geom_point(size = 2, color = "red", shape = 2) +
  ggtitle("I'm a red, triangular ggplot")
```

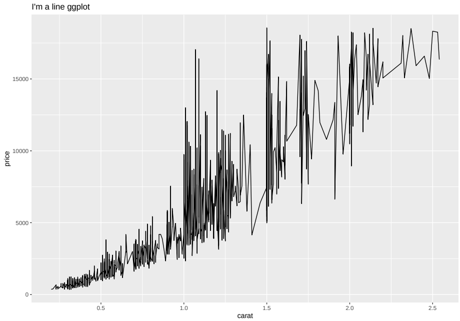


The shape argument works just like pch in base plotting.

Changing the Graph Type in ggplot2

Instead of adding `geom_point()`, we can add something else, like `geom_line()`

```
ggplot(data = diam, aes(x = carat, y = price)) +
  geom_line() + ggtitle("I'm a line ggplot")
```

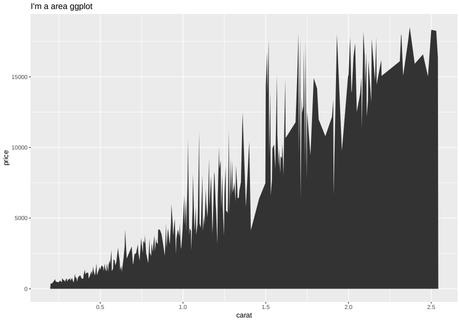


This looks strange for this plot, though.

Changing the Graph Type in ggplot2

Or even something like shading the area under the points.

```
ggplot(data = diam, aes(x = carat, y = price)) +
  geom_area() + ggtitle("I'm a area ggplot")
```

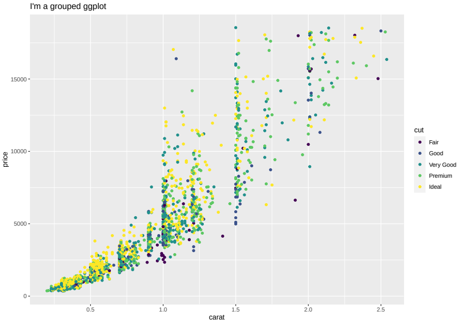


This looks even stranger in this case.

Grouping by Another Variable

ggplot makes it easy to split the data using another variable. Simply put the `col` argument in the `aes()` function in `ggplot`. This will automatically add a legend as well. We can change the shape based on another variable too.

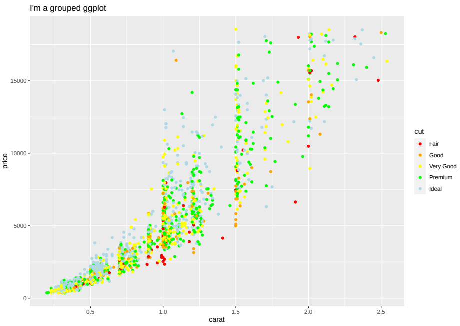
```
ggplot(data = diam, aes(x = carat, y = price, col = cut)) +
  geom_point() + ggtitle("I'm a grouped ggplot")
```



Grouping by Another Variable

We can manually change the colors using the `values` option in the `scale_color_manual()` add on function.

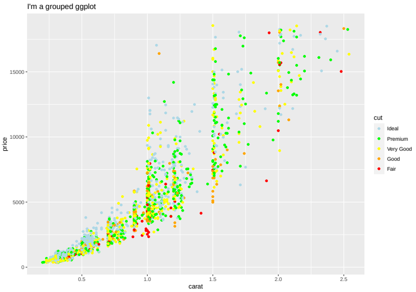
```
ggplot(data = diam, aes(x = carat, y = price, col = cut)) +
  geom_point() + ggtitle("I'm a grouped ggplot") +
  scale_color_manual(values = c("red", "orange", "yellow",
                                "green","lightblue"))
```



Grouping by Another Variable

We can manually change the order of the categories in the legend using the breaks option in the scale_color_manual() add on function.

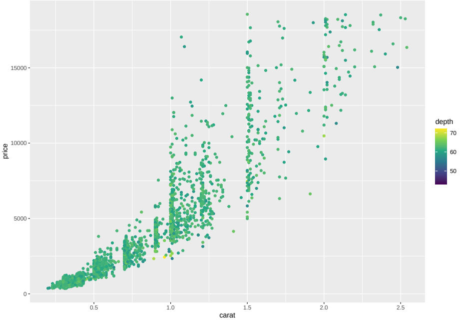
```
ggplot(data = diam, aes(x = carat, y = price, col = cut)) +
  geom_point() + ggtitle("I'm a grouped ggplot") +
  scale_color_manual(
    breaks = c("Ideal","Premium","Very Good","Good","Fair"),
    values = c("lightblue", "green","yellow","orange","red"))
```



Grouping by Another Variable

We can also group by a continuous variable. In this case, we can change the color based on the depth variable.

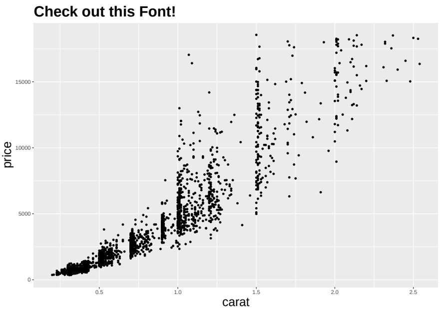
```
# color by a continuous variable
ggplot(data = diam, aes(x = carat,
          y = price, color = depth)) + geom_point() +
  scale_colour_continuous(type = "viridis")
```



Changing Font Size and Type

We can change font size and type in the `theme()` function:

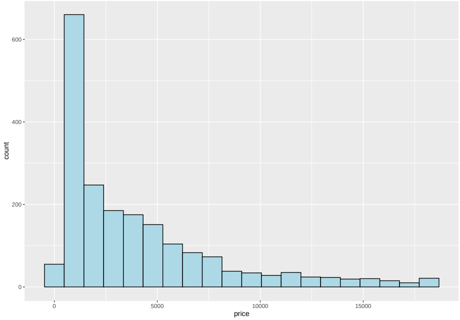
```
# color by a continuous variable
ggplot(data = diam, aes(x = carat, y = price)) +
  geom_point(shape = 16, size = 1.5) +
  ggtitle("Check out this Font!") +
  theme(axis.title = element_text(size = 20),
        plot.title = element_text(size = 24, face = "bold"))
```



ggplot2 for a Single Quantitative Variable: Histogram

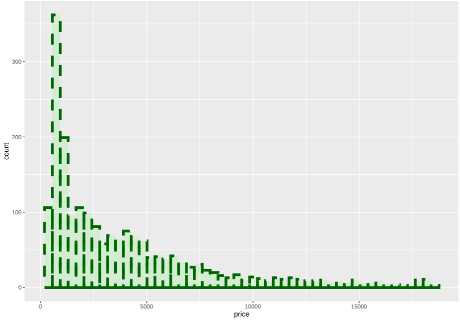
Of course, we can also use `ggplot()` for plotting a single variable. We can make histograms, boxplots, dotplots, etc.

```
# Save the base plot as an object p. Then add to p.
p <- ggplot(data = diam, aes(x = price))
p + geom_histogram(color = "black", fill = "lightblue",
                   bins = 20)
```



ggplot2 for a Single Quantitative Variable: Histogram

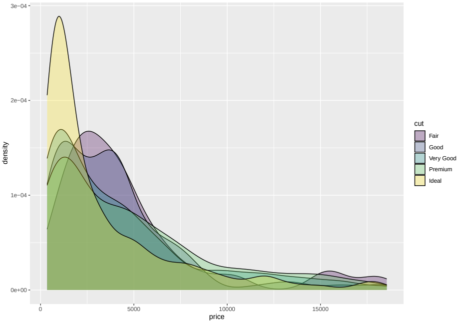
```
# We already created p, so we can add other options to it.
# Smaller alpha makes the plot more transparent.
p + geom_histogram(color = "darkgreen", fill = "green",
                   bins = 50, alpha = 0.1, lty = 2, lwd = 2)
```



ggplot2 for a Single Quant. Variable: Layered Histograms

A layered histogram is a good way to compare the distribution of a variable across groups. geom_histogram works well for two groups, but geom_density is easier to look at for several groups.

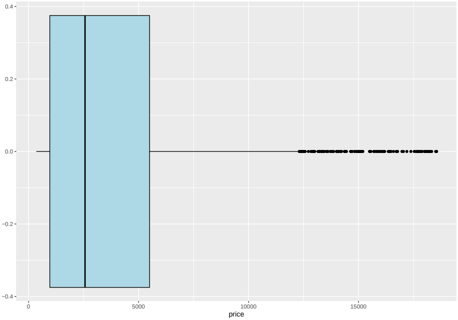
```
ggplot(data = diam, aes(x = price, fill = cut)) +
  geom_density(alpha = 0.3)
```



ggplot2 for a Single Quantitative Variable: Boxplot

Creating a basic boxplot. We can also make it vertical by putting `y = price` in the `aes()` function instead of `x = price`.

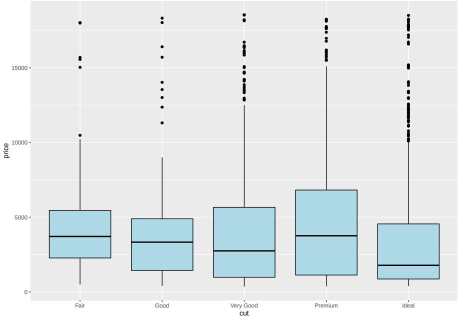
```
ggplot(data = diam, aes(x = price)) +
  geom_boxplot(color = "black", fill = "lightblue")
```



ggplot2 for a Single Quant. Variable: Side-by-Side Boxplots

We can make side-by-side boxplots grouped by a categorical variable as x (or as y if you want side-by-side horizontal boxplots).

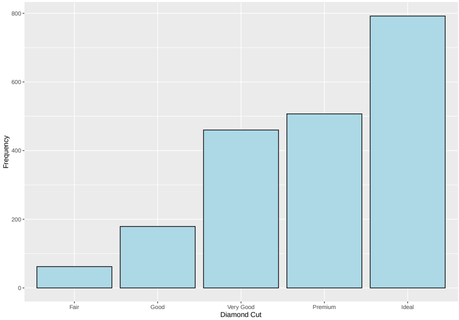
```
ggplot(data = diam, aes(x = cut, y = price)) +
  geom_boxplot(color = "black", fill = "lightblue")
```



ggplot2 for a Single Categorical Variable: Barplot

We can make plots for categorical variables as well.

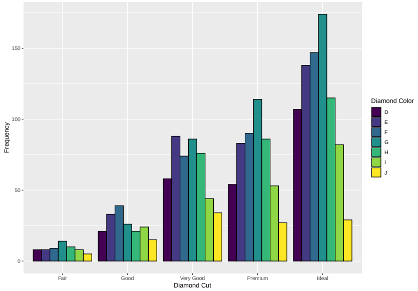
```
ggplot(data = diam, aes(x = cut)) +
  geom_bar(color = "black", fill = "lightblue") +
  labs(x = "Diamond Cut", y = "Frequency")
```



ggplot2 for a Single Categ. Variable: Side-by-Side Barplots

We can split the bars by another variable. In this case, we will make a plot of diamond cut and break it up by diamond color and put the bars side by side.

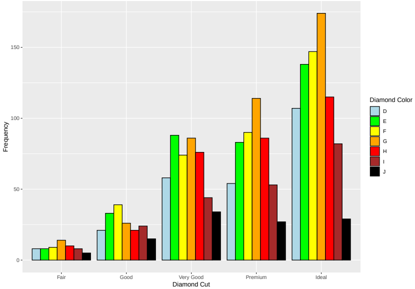
```
ggplot(data = diam, aes(x = cut, fill = color)) +
  geom_bar(color = "black", position = "dodge") +
  labs(x = "Diamond Cut", y = "Frequency",
       fill = "Diamond Color")
```



ggplot2 for a Single Categ. Variable: Side-by-Side Barplots

We can change the fill colors using scale_fill_manual().

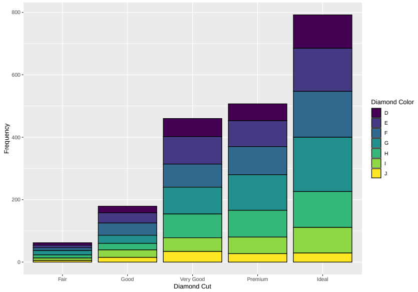
```
ggplot(data = diam, aes(x = cut, fill = color)) +
  geom_bar(color = "black", position = "dodge") +
  labs(x = "Diamond Cut", y = "Frequency",
       fill = "Diamond Color") +
  scale_fill_manual(values = c("lightblue","green","yellow",
                               "orange","red","brown","black"))
```



ggplot2 for a Single Categorical Variable: Stacked Barplot

We can split the bars by another variable. In this case, we will make a plot of diamond cut and break it up by diamond color and stack the bars.

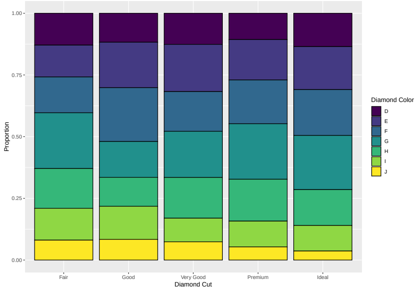
```
ggplot(data = diam, aes(x = cut, fill = color)) +
  geom_bar(color = "black", position = "stack") +
  labs(x = "Diamond Cut", y = "Frequency",
       fill = "Diamond Color")
```



ggplot2 for a Single Categorical Variable: Stacked Barplot

We can split the bars by another variable. In this case, we will make a plot of diamond cut and break it up by diamond color, stack the bars, and adjust them so each bar totals 100%.

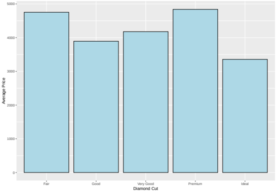
```
ggplot(data = diam, aes(x = cut, fill = color)) +
  geom_bar(color = "black", position = "fill") +
  labs(x = "Diamond Cut", y = "Proportion",
       fill = "Diamond Color")
```



ggplot2 Barplot Identity

We often want to use a column of a data frame or tibble as the heights of our bar plot instead of having ggplot tabulate them for us. For this, we need to put `stat = identity` in the `geom_bar()` function.

```
diam |> group_by(cut) |>
  summarize(avg_price = mean(price)) |>
  ggplot(aes(x = cut, y = avg_price)) +
  geom_bar(color = "black", stat = "identity",
           fill = "lightblue") +
  labs(x = "Diamond Cut", y = "Average Price")
```



Using ggplot2 for more complex plots

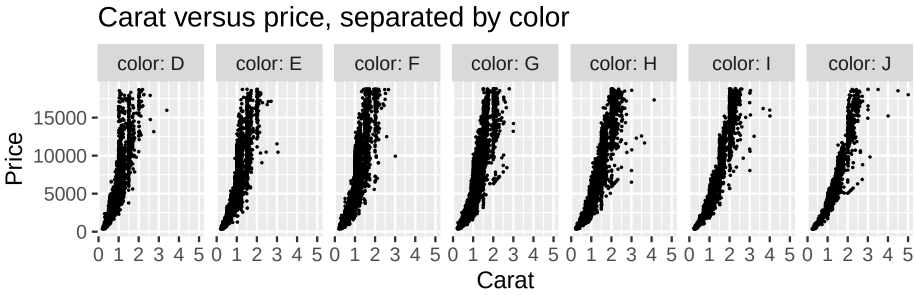
Use facet_grid() or facet_wrap() to create a separate plot for each value of a factor variable. We don't have to change any of the original plotting code, just add the facet command to it. Faceting can also be done on more than one categorical variable to create a grid of plots.

Additionally, it is sometimes helpful to save a simpler version of a plot, and then add onto it later with additional layers (for example, an if/else statement that plots different layers dependent on if a criterion is met or not).

We might want to summarize the data in the previous plot with a smoother on top of the points. With ggplot, we can simply add the geom_smooth command. Each geom just adds another layer to the plot.

Using ggplot2 for more complex plots

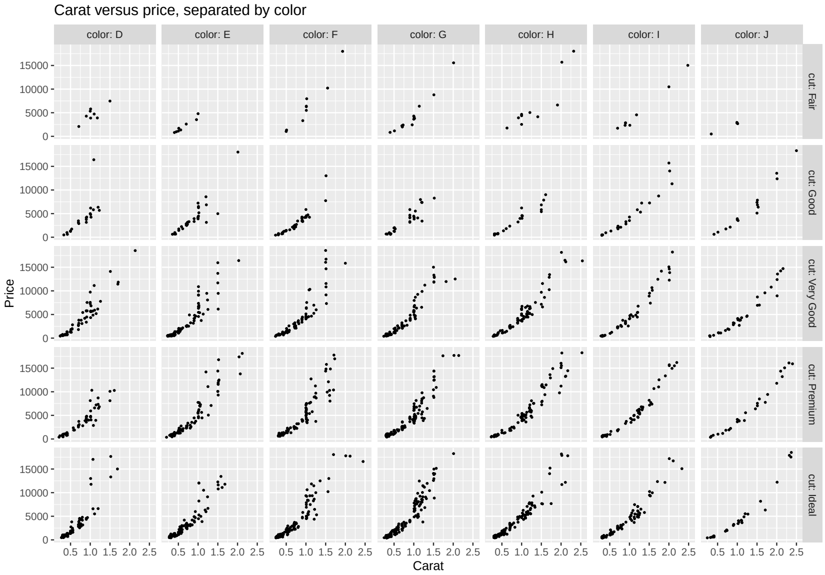
```
# make the basis for a plot using ggplot save it as p
p <- ggplot(data = diamonds, aes(x = carat, y = price))
# add a geom (points) and display the plot
p + geom_point(size=0.1) + facet_grid(cols = vars(color),
    labeller = label_both) + labs(x = "Carat", y = "Price",
      title = "Carat versus price, separated by color")
```



Using ggplot2 for more complex plots

```r
# make the basis for a plot using ggplot save it as p
p <- ggplot(data = diam, aes(x = carat, y = price))
# add a geom (points) and display the plot
p + geom_point(size=0.1) + facet_grid(rows = vars(cut),
        cols = vars(color), labeller = label_both) +
  labs(x = "Carat", y = "Price",
       title = "Carat versus price, separated by color")
```

Using ggplot2 for more complex plots



Summary

The syntax of a ggplot is `ggplot(data, aes(x, y))` and you add on to the plot with + at the end of each line.

The most useful functions to add onto a ggplot are:

- `geom_point()`, `geom_line()`, `geom_histogram()`, `geom_boxplot()`, `geom_text()`, etc.
- `labs()` for labels.
- `ggtitle()` for a plot title.
- `lims()` for limits.
- `theme()` for text size and visually changing other things.
- `scale_color_manual()` or `scale_fill_manual()` for changing the color or fill of the plot manually.

Further Resources & Assistance

- Cheat sheet for data visualization with ggplot2 (accessible in Rstudio by going to Help -> Cheat Sheets -> Data visualization with ggplot2)
- ggplot2 documentation
- Google
- Stack overflow
- Hadley Wickham's book: https://ggplot2-book.org/
- Rafael Irizarry's book: http://rafalab.dfci.harvard.edu/dsbook-part-1/dataviz/ggplot2.html.
- Note: these slides were adapted from slides created by Aubrey Odom.

Session info

```
sessionInfo()
```

```
## R version 4.3.2 (2023-10-31)
## Platform: aarch64-apple-darwin20 (64-bit)
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
##
## other attached packages:
##  [1] lubridate_1.9.3 forcats_1.0.0   stringr_1.5.1
##  [4] dplyr_1.1.4     purrr_1.0.2     readr_2.1.5
##  [7] tidyr_1.3.0     tibble_3.2.1    ggplot2_3.4.4
## [10] tidyverse_2.0.0 dslabs_0.7.6
##
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.4      highr_0.10
##  [3] compiler_4.3.2    tidyselect_1.2.0
##  [5] scales_1.3.0      yaml_2.3.8
##  [7] fastmap_1.1.1     R6_2.5.1
##  [9] labeling_0.4.3    generics_0.1.3
## [11] knitr_1.45        munsell_0.5.0
```