R_For_Data_Science_Notes

Abhishek Dangol

12/30/2019

The mpg data frame

First we need to load the ggplot2 package. 'mpg' is a dataset in the ggplot2 library.

```
library(ggplot2)
```

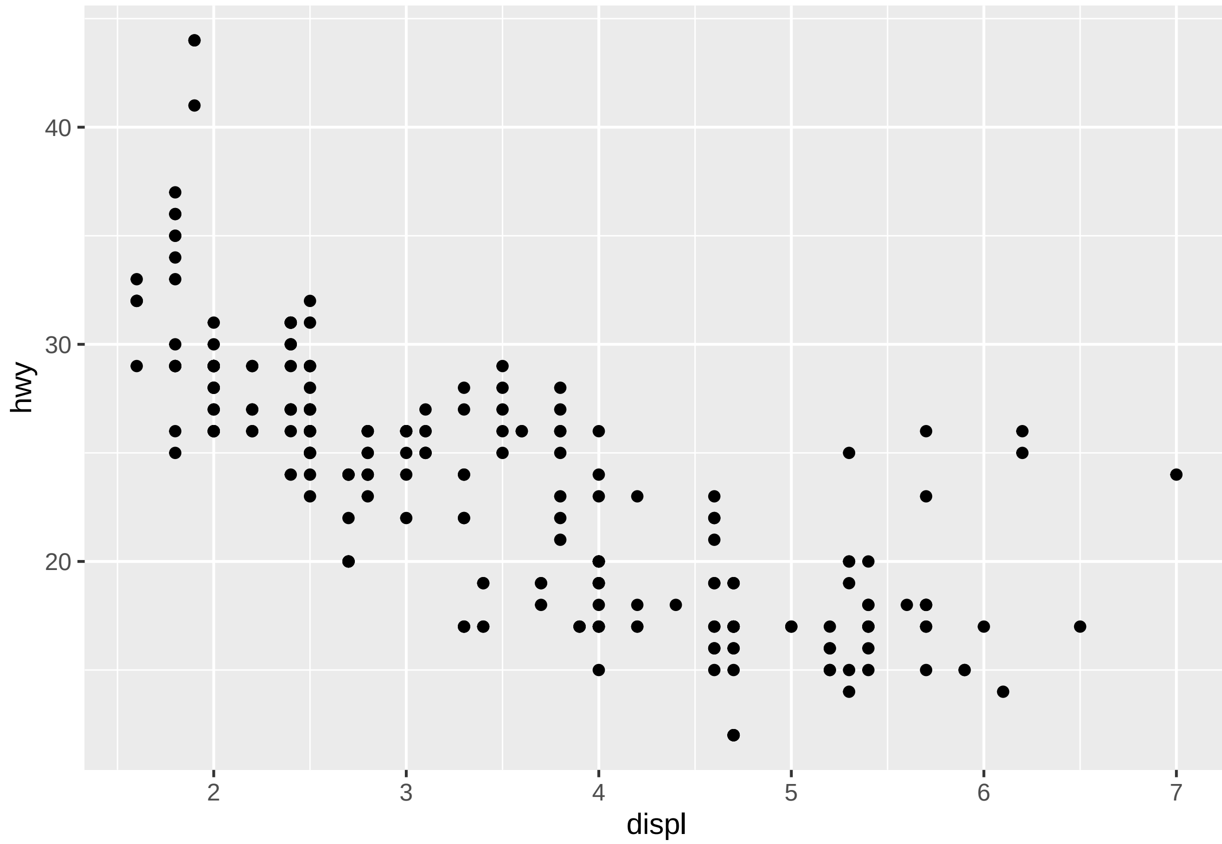
```
## Warning: package 'ggplot2' was built under R version 3.6.2
```

```
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer model displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4      1.8  1999     4 auto~ f        18    29 p     comp~
##  2 audi         a4      1.8  1999     4 manu~ f        21    29 p     comp~
##  3 audi         a4      2    2008     4 manu~ f        20    31 p     comp~
##  4 audi         a4      2    2008     4 auto~ f        21    30 p     comp~
##  5 audi         a4      2.8  1999     6 auto~ f        16    26 p     comp~
##  6 audi         a4      2.8  1999     6 manu~ f        18    26 p     comp~
##  7 audi         a4      3.1  2008     6 auto~ f        18    27 p     comp~
##  8 audi         a4 q~   1.8  1999     4 manu~ 4        18    26 p     comp~
##  9 audi         a4 q~   1.8  1999     4 auto~ 4        16    25 p     comp~
## 10 audi         a4 q~   2    2008     4 manu~ 4        20    28 p     comp~
## # ... with 224 more rows
```

We want to find out if cars with bigger engines consume more gas. The geom_point() adds a layer of points to the plot (scatterplot) Each geom function in ggplot takes a mapping as argument. This defines how variables in dataset are mapped to visual properties. The mapping argument is always paired with aes() and the x and y arguments of aes() specify which variables to map to the x and y axes.

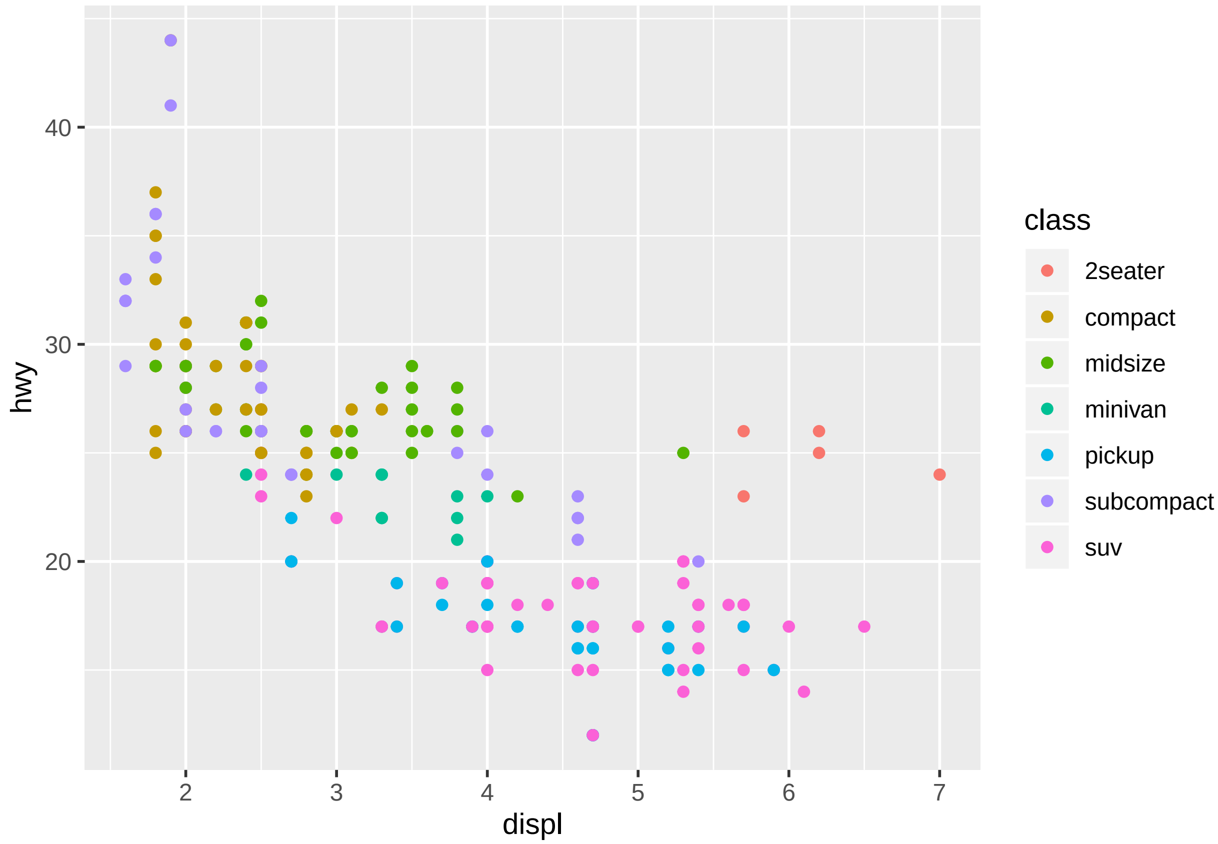
```
ggplot(data=mpg) + geom_point(mapping = aes(x = displ, y = hwy))
```



Aesthetic mapping

We can map the colors of our points to the 'class' variable to reveal the class of each car. As we cam see the red dots are sports cars and hence, they have better mileage despite having bigger engines.

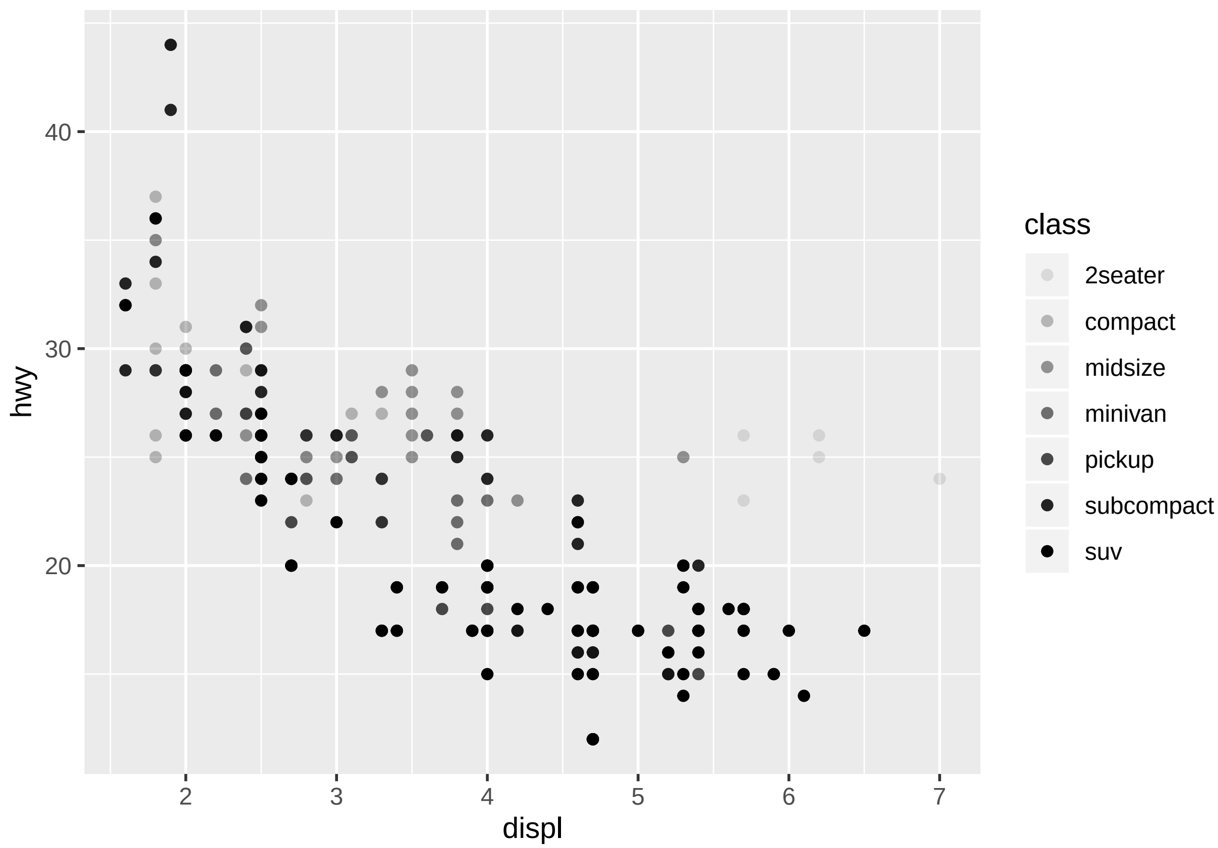
```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



Alternatively, we can also map class to the alpha aesthetic which controls the transparency of the points or to the shape aesthetic which controls the shape of the points/

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, alpha = class))
```

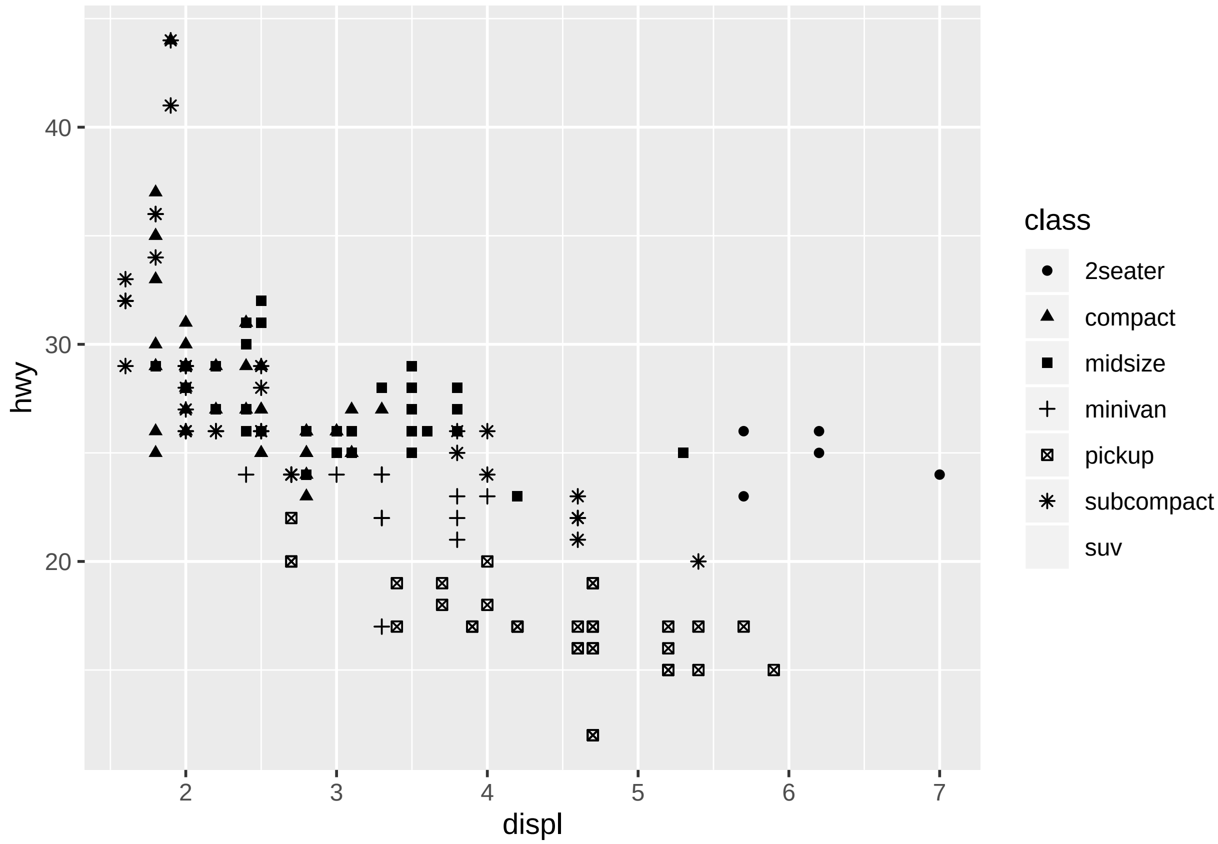
```
## Warning: Using alpha for a discrete variable is not advised.
```




```r
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```

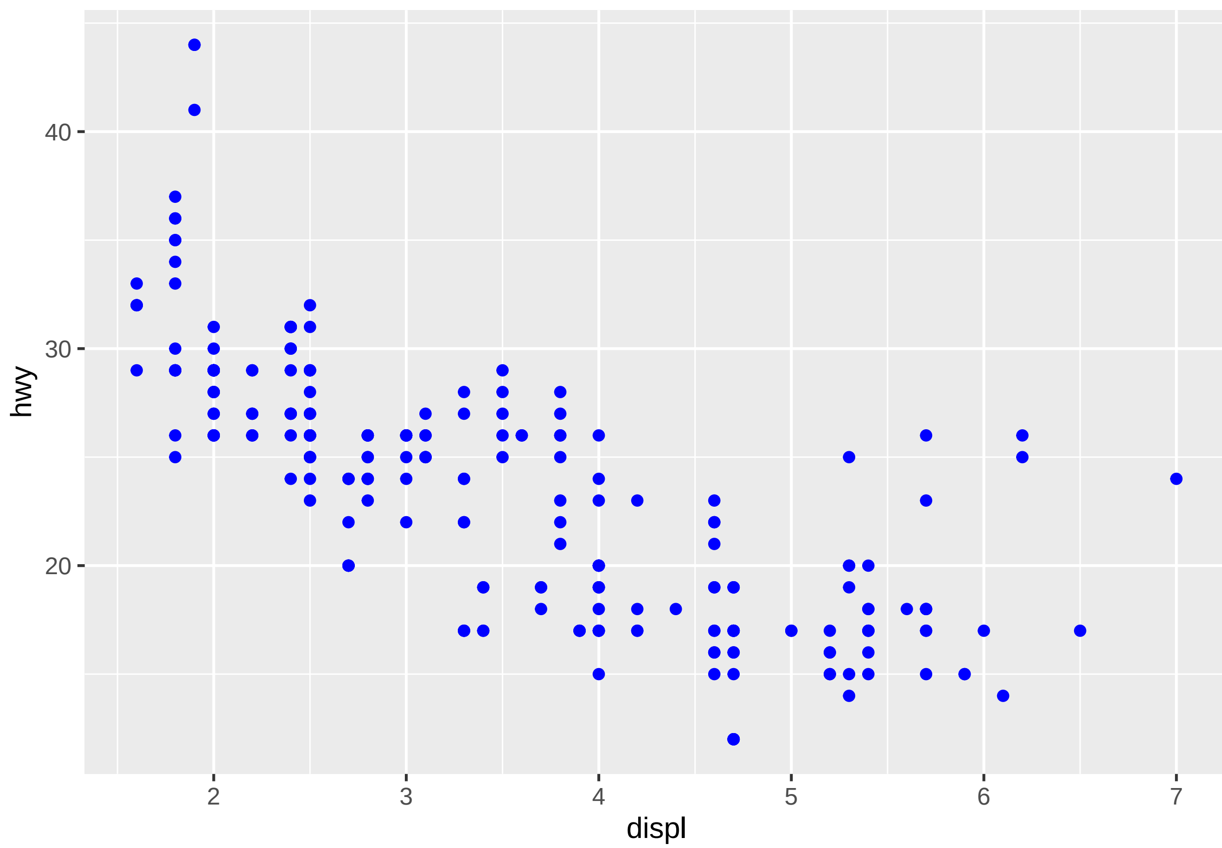
```
## Warning: The shape palette can deal with a maximum of 6 discrete values
## because more than 6 becomes difficult to discriminate; you have 7.
## Consider specifying shapes manually if you must have them.

## Warning: Removed 62 rows containing missing values (geom_point).
```



We can also set the aesthetic property of the geom manually. For example, we can make all of the points in our plot blue.

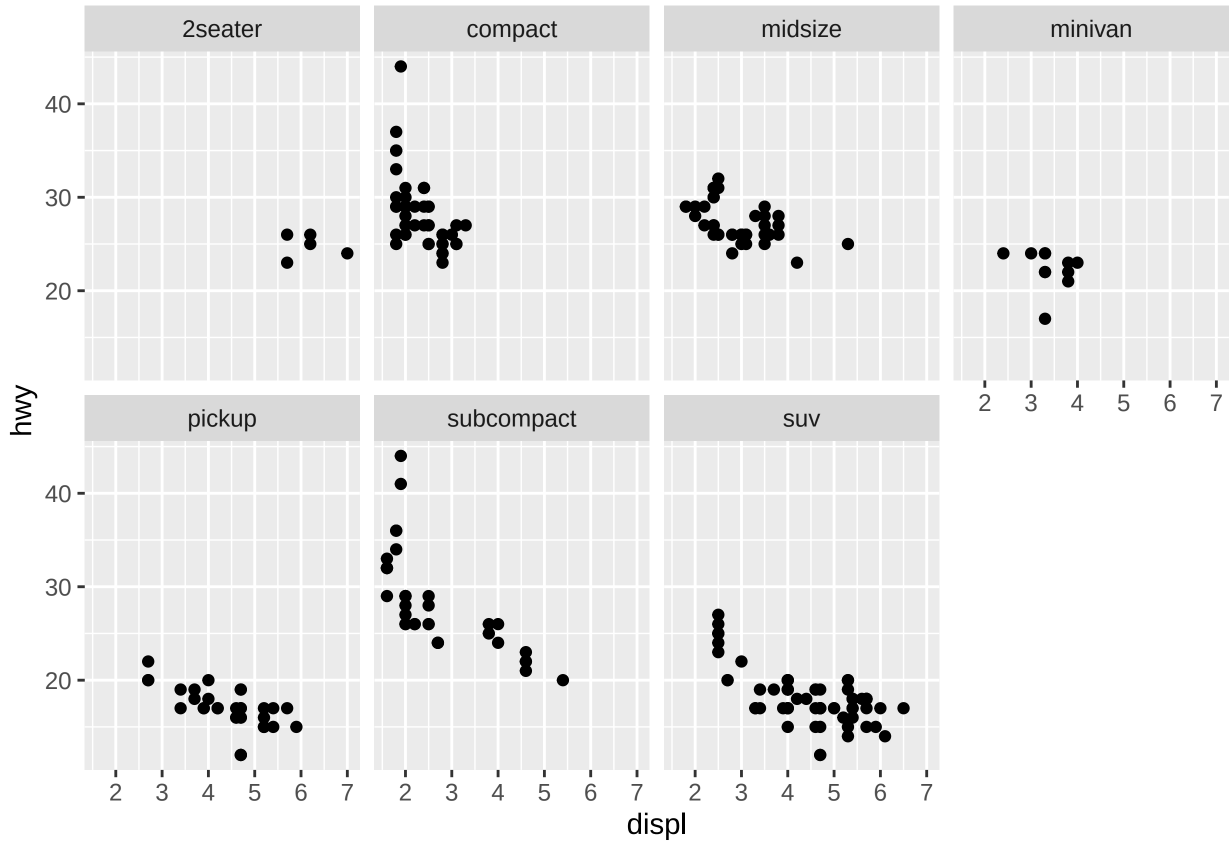
```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy), color = "blue")
```



Facets

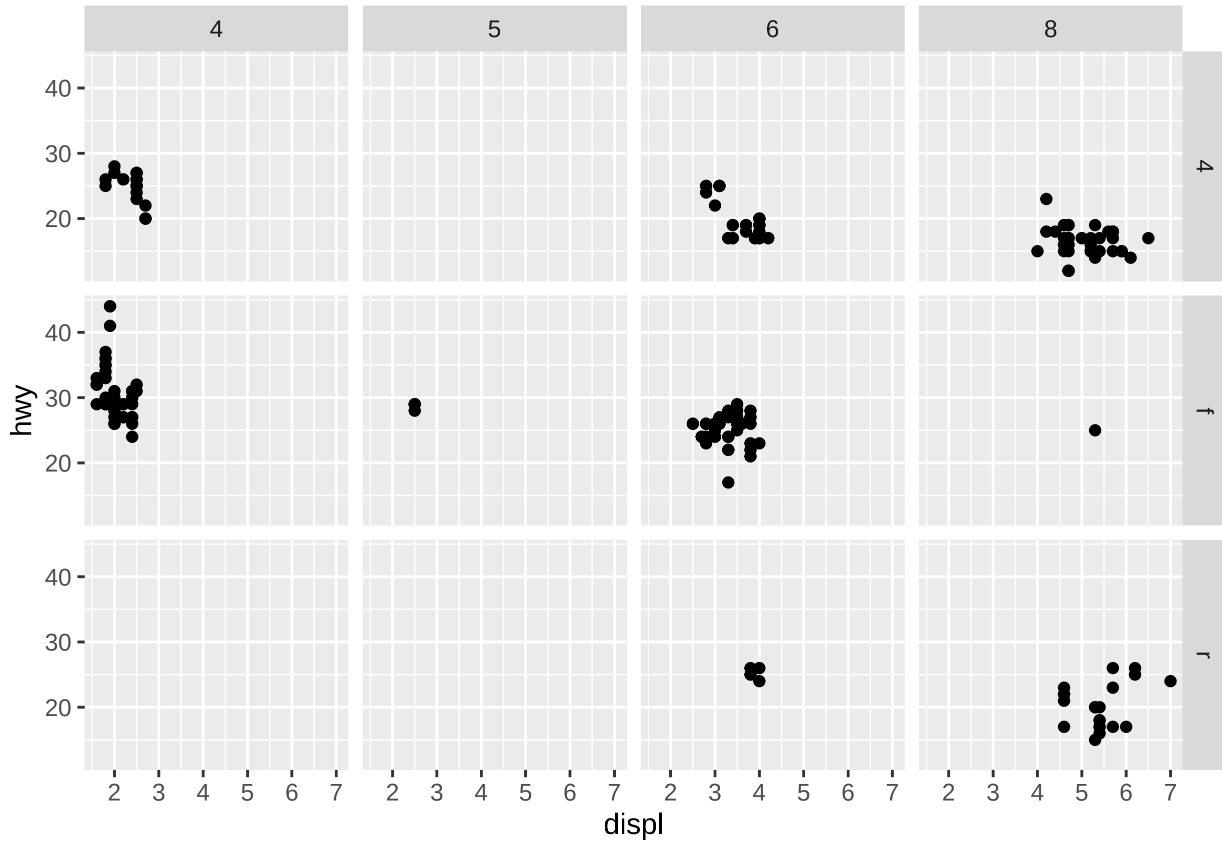
Another way, particularly useful for categorical variables is to split the plot into facets - subplots that each display one subset of the data. Use facet_wrap(). The first argument of facet_wrap() should be a formula which we can create with ~ followed by a variable name.

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) + facet_wrap(~ class, nrow = 2)
```



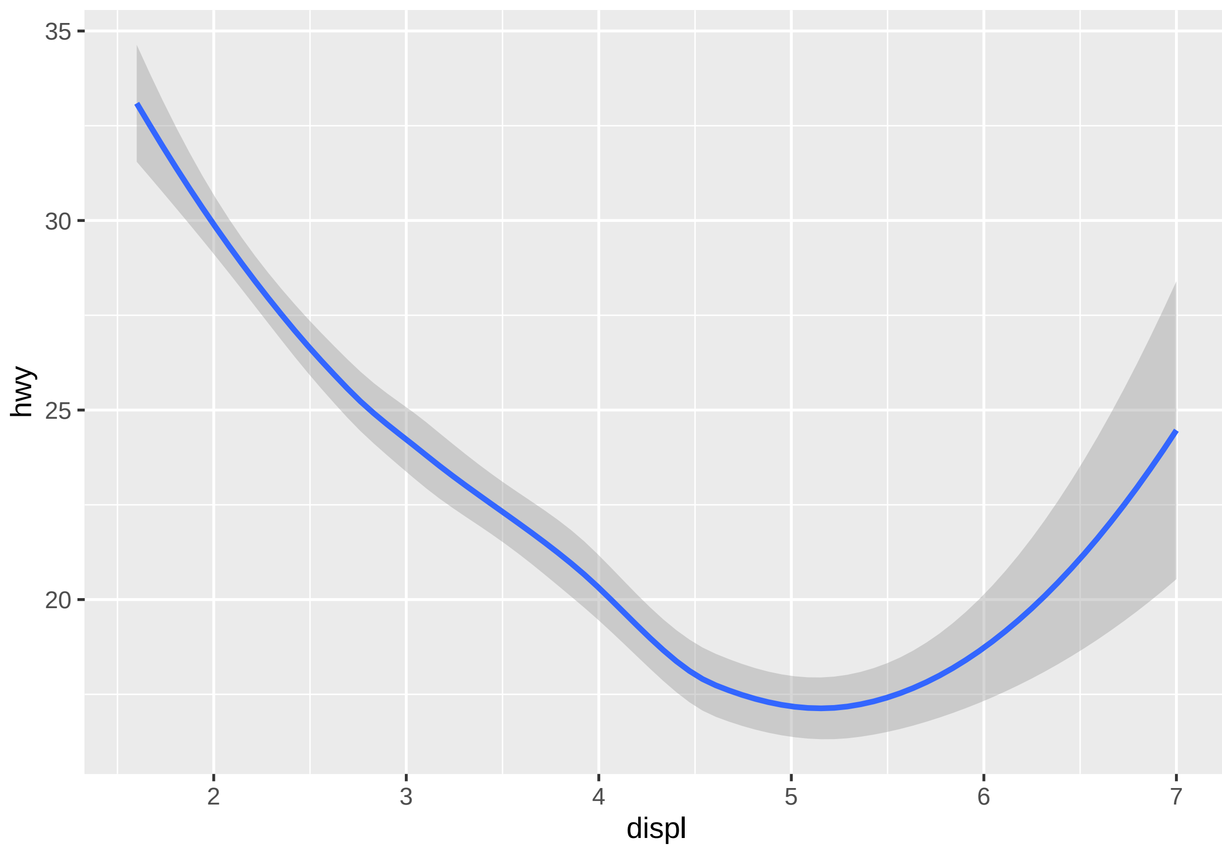
To facet the plot on the combination of two variables, add facet_grid() to plot call. To plot on the basis fo 'drv' and 'cyl'

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) + facet_grid(drv ~ cyl)
```



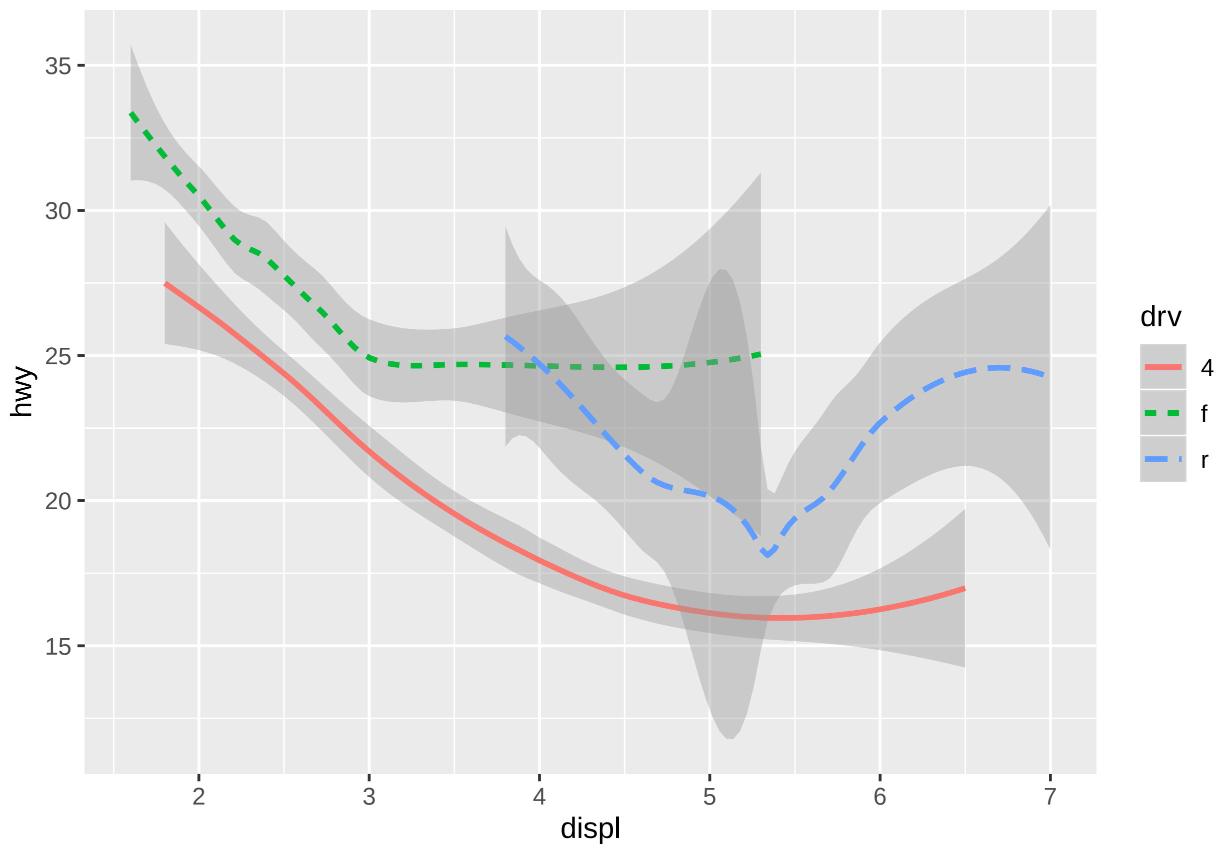
There are several different kinds of geoms such as bar geoms, line geoms, boxplot geoms, point geoms (as seen above) and smooth geom (as shown below)

```
ggplot(data = mpg) + geom_smooth(mapping = aes(x = displ, y = hwy))
```



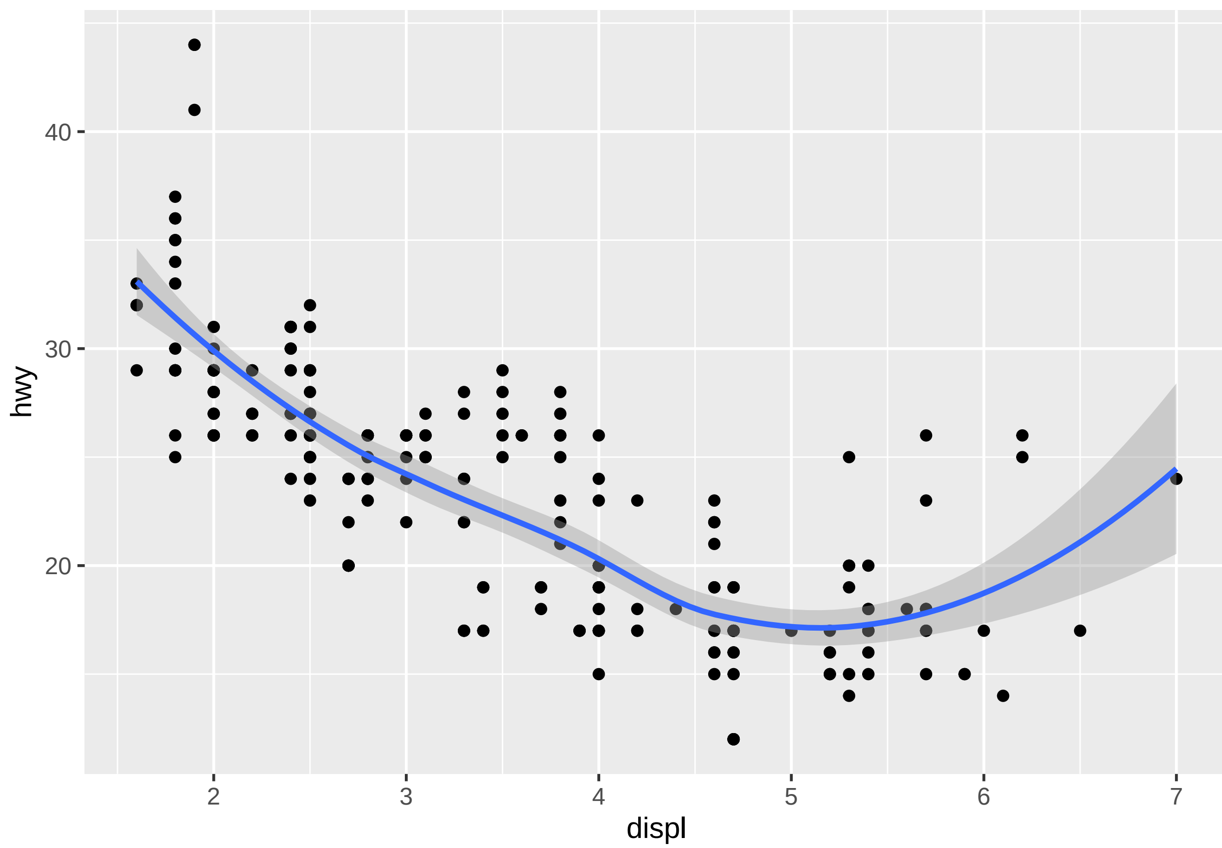
We can also set the linetype of a line. geom_smooth() will draw a different line, with a different linetype, for each unique value of the variable that you map to linetype.

```
ggplot(data = mpg) + geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv, color = drv))
```



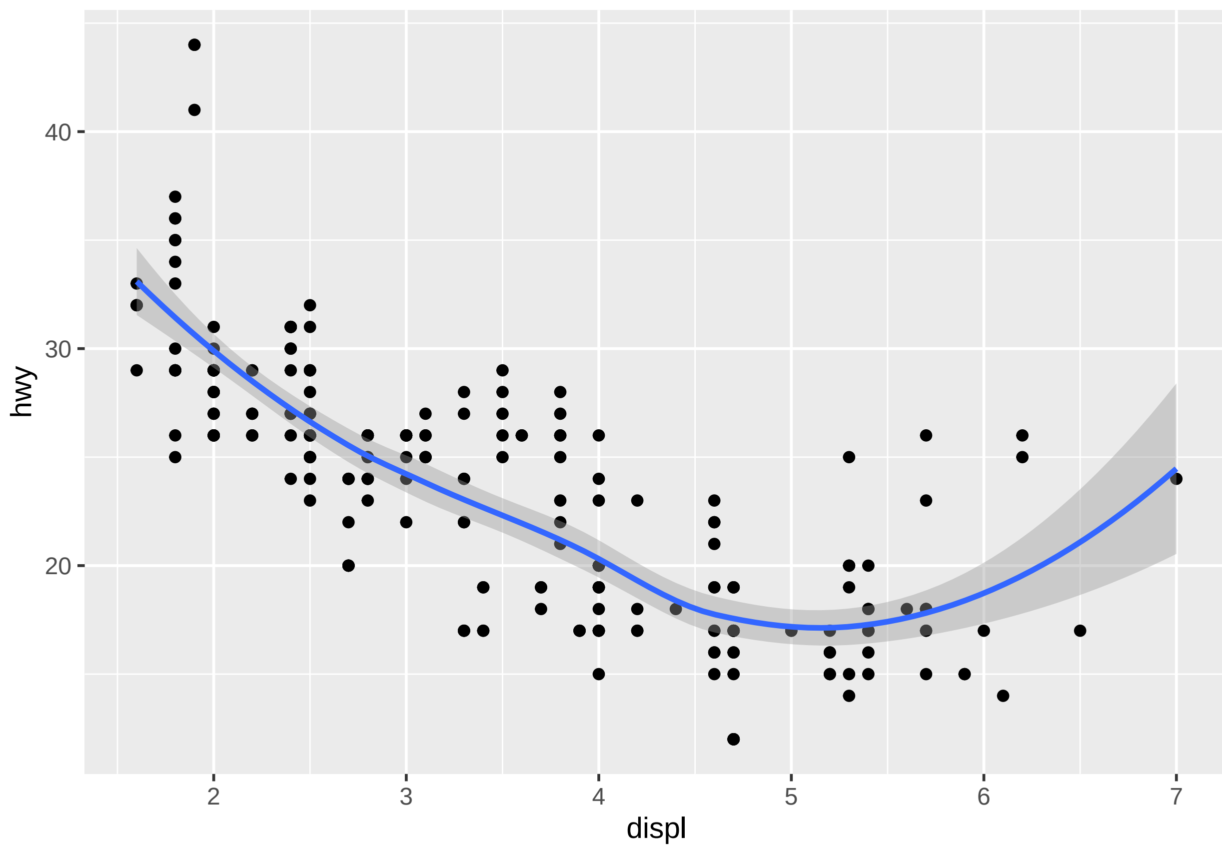
To display multple geoms in the same plot, add multiple geom functions to ggplot()“‘{r}

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) + geom_smooth(mapping = aes(x = displ
```



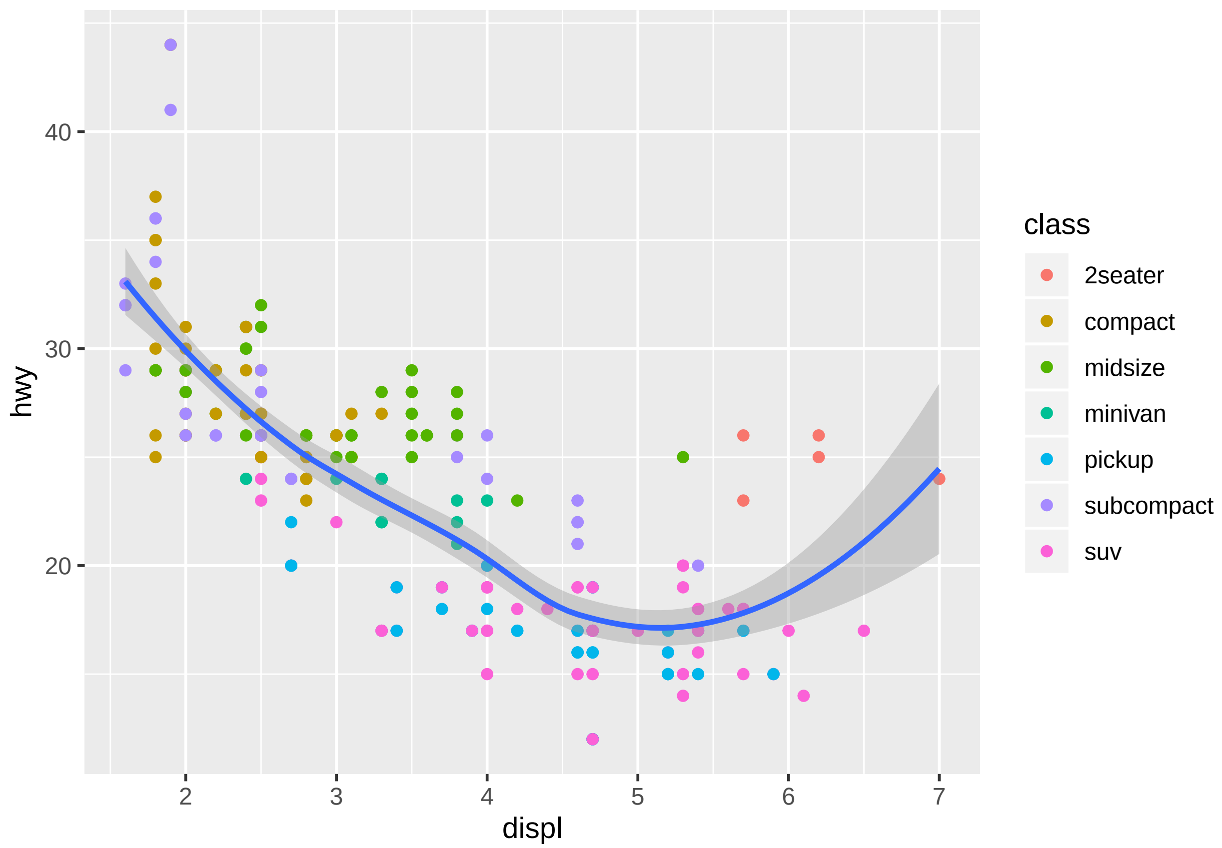
We can get rid of the duplication by using the following code instead

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom_point() + geom_smooth()
```



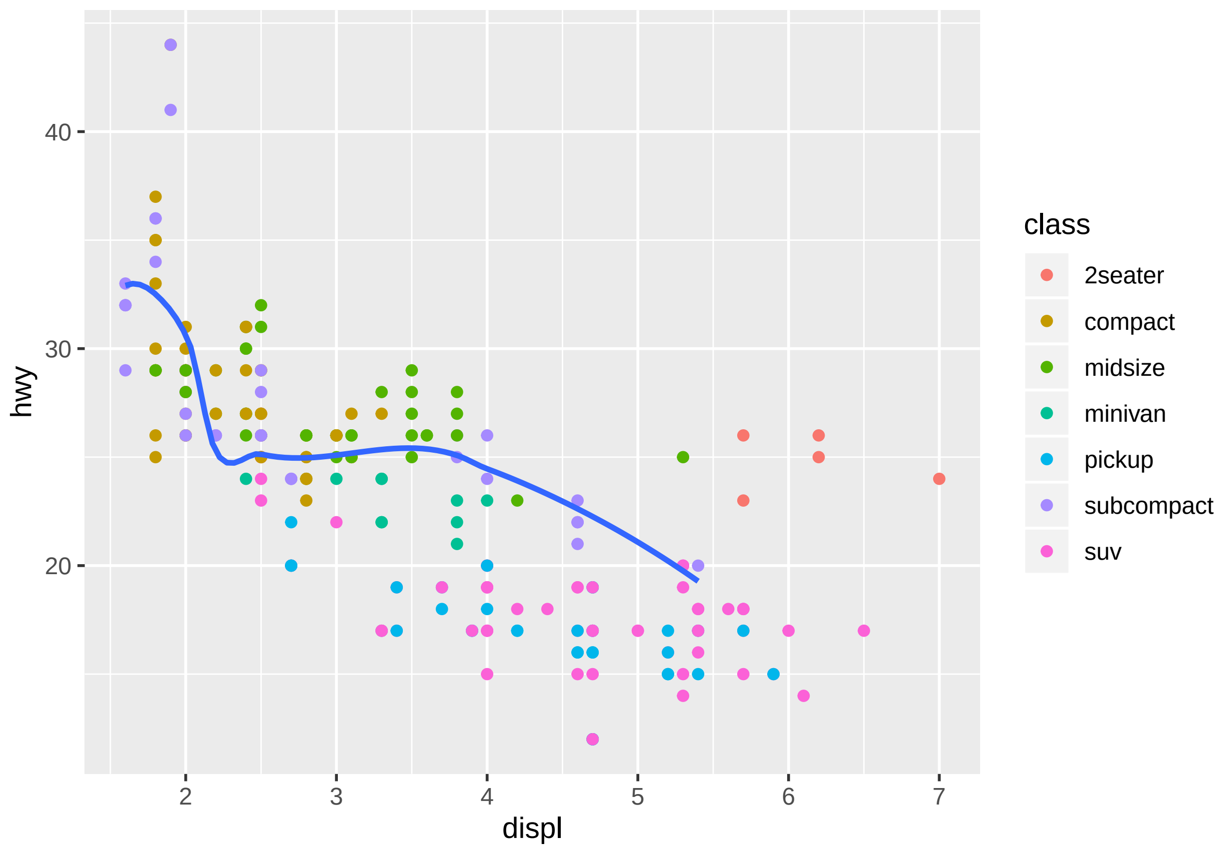
If we were to add a mapping tpa a geom function, it will add it to that layer only. This makes it possible to display different aesthetics in different layers

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom_point(mapping = aes(color = class)) + geom_
```



We can also specify data for each layer. Here, our smooth line displays just a subset of the mpg dataset, the subcompact cars. The local argument in geom_smooth() overrides the global argument in ggplot() for that layer only. We need to load dplyr package for the filter() function to work.

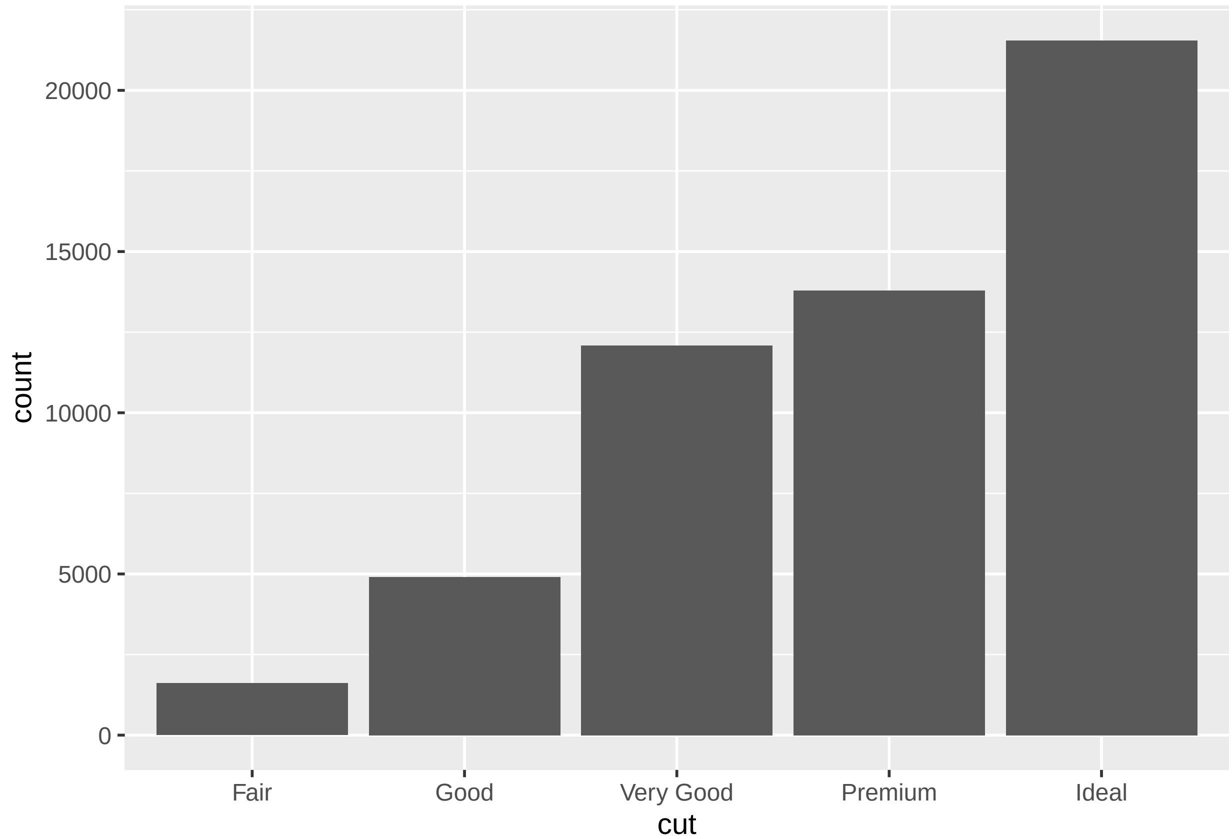
```
library(dplyr)
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom_point(mapping = aes(color = class)) + geom_
```



STATISTICAL TRANSFORMATIONS

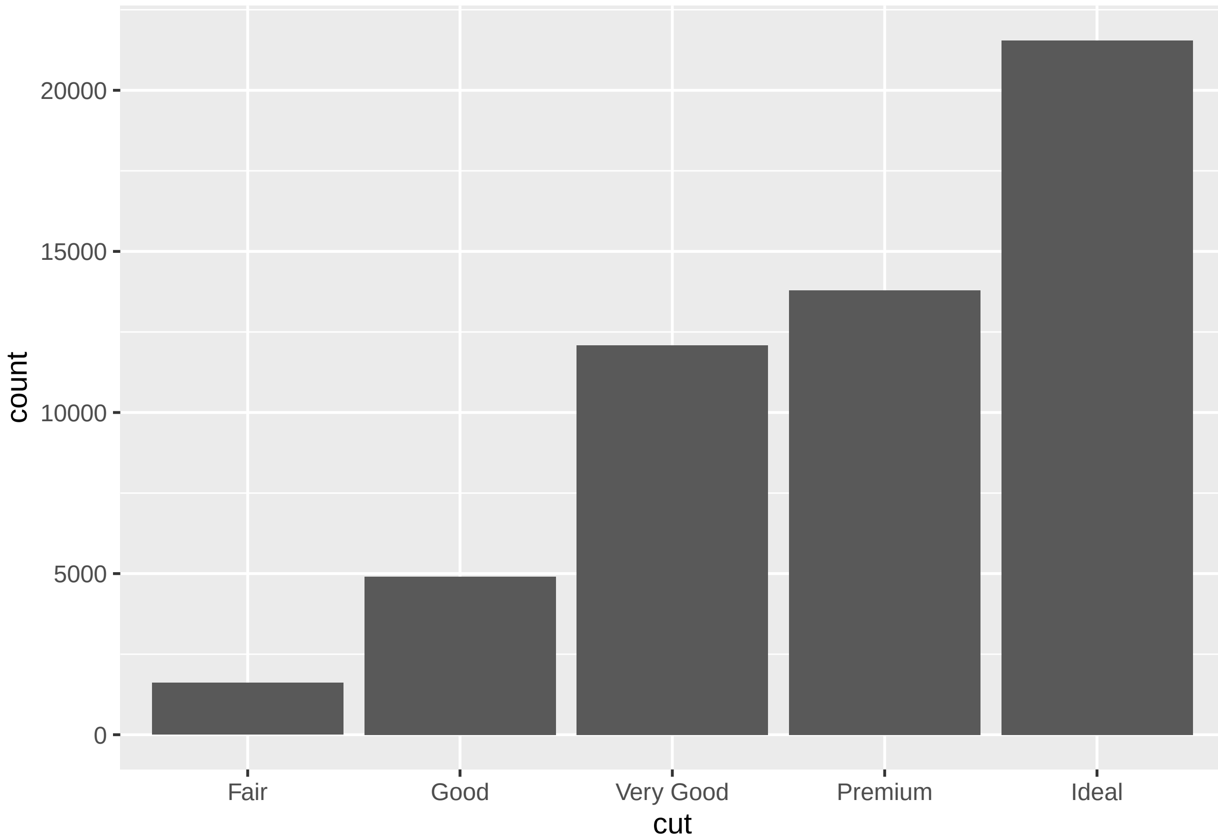
Even though we did not specify the y axis in the bar chart below, R can automatically calculate new values for the graph using stat or statistical transformations.

```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut))
```



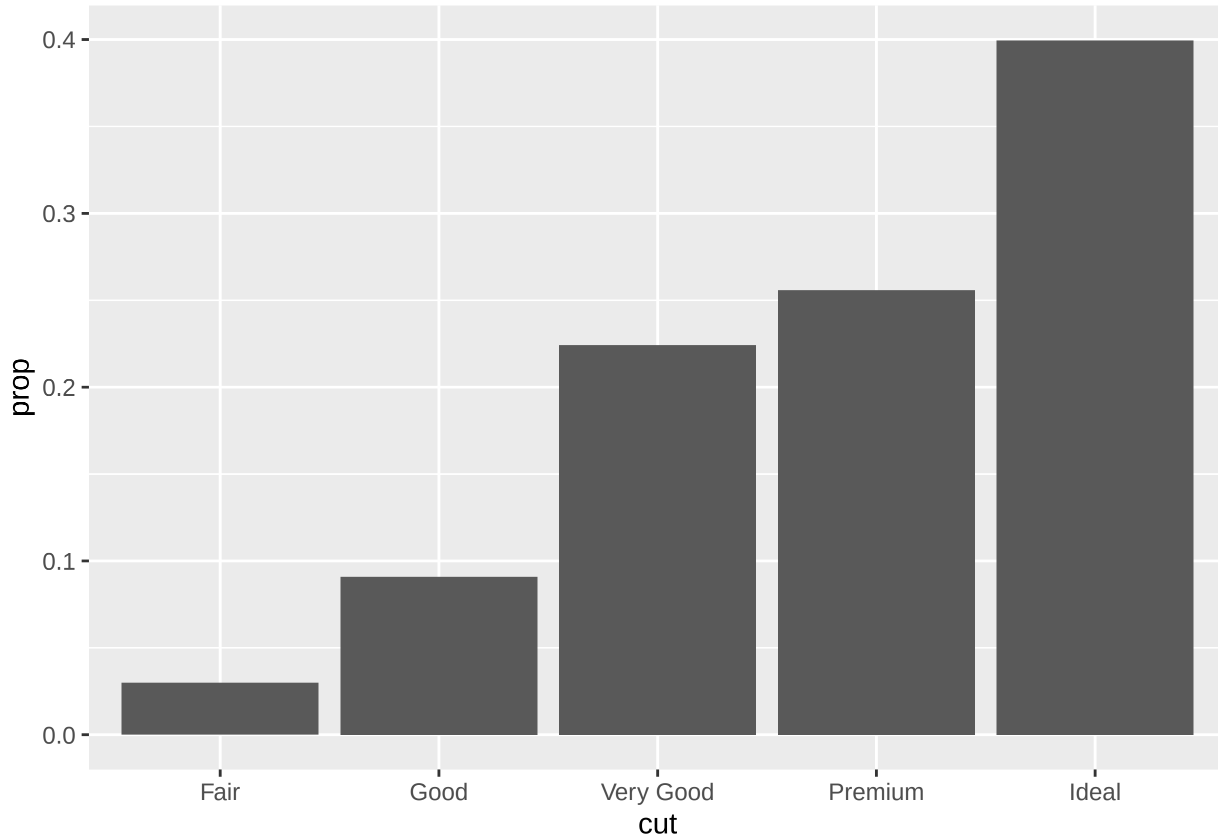
The default value for stat in geom_bar is count. We can interchangeably use stat_count() and geom_bar().

```
ggplot(data = diamonds) + stat_count(mapping = aes(x = cut))
```



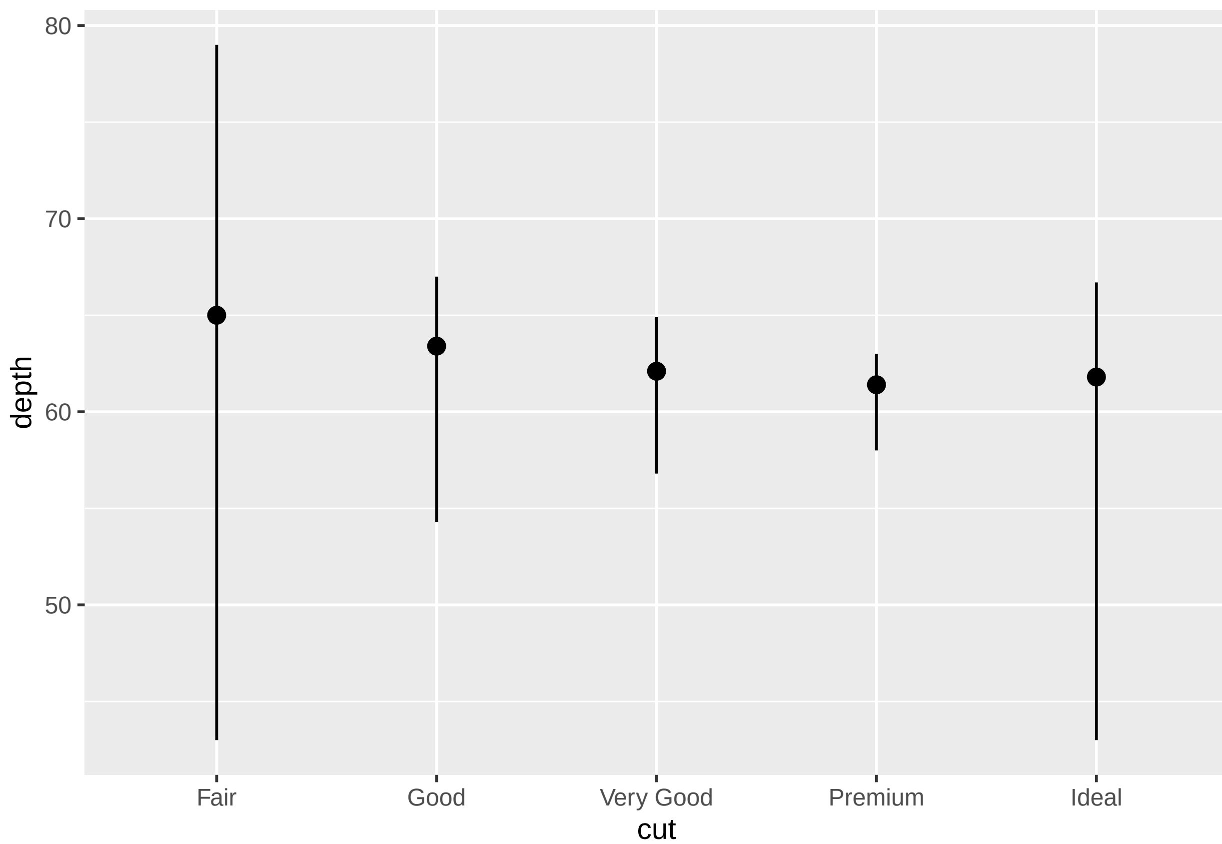
We can also override the default stat. Here we are using 'identity' instead of the default 'count'. Also, we can display a bar chart of proportion rather than count

```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, y = ..prop.., group = 1))
```



`stat_summary()` summarizes the y values for each unique x value, to draw attention to the summary that we are computing

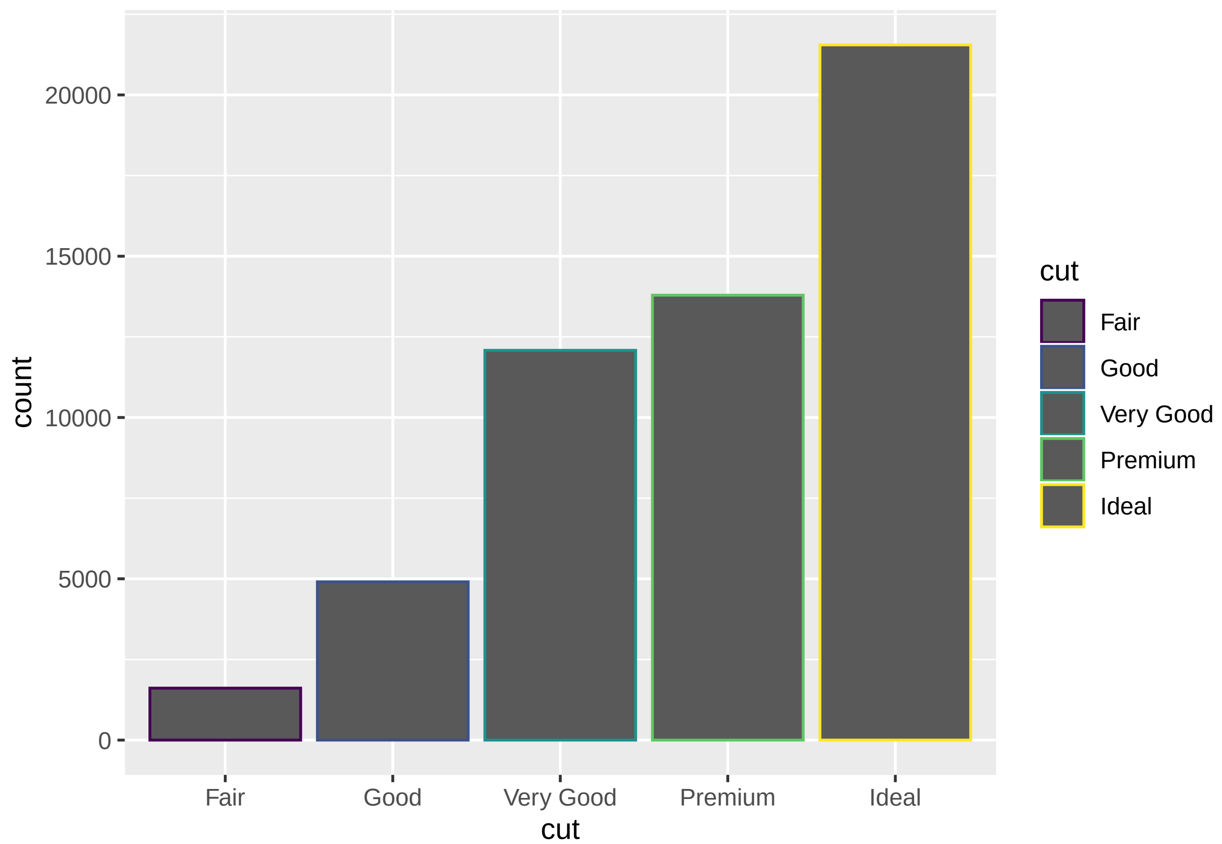
```
ggplot(data = diamonds) + stat_summary(
  mapping = aes(x = cut, y = depth),
  fun.ymin = min,
  fun.ymax = max,
  fun.y = median
)
```



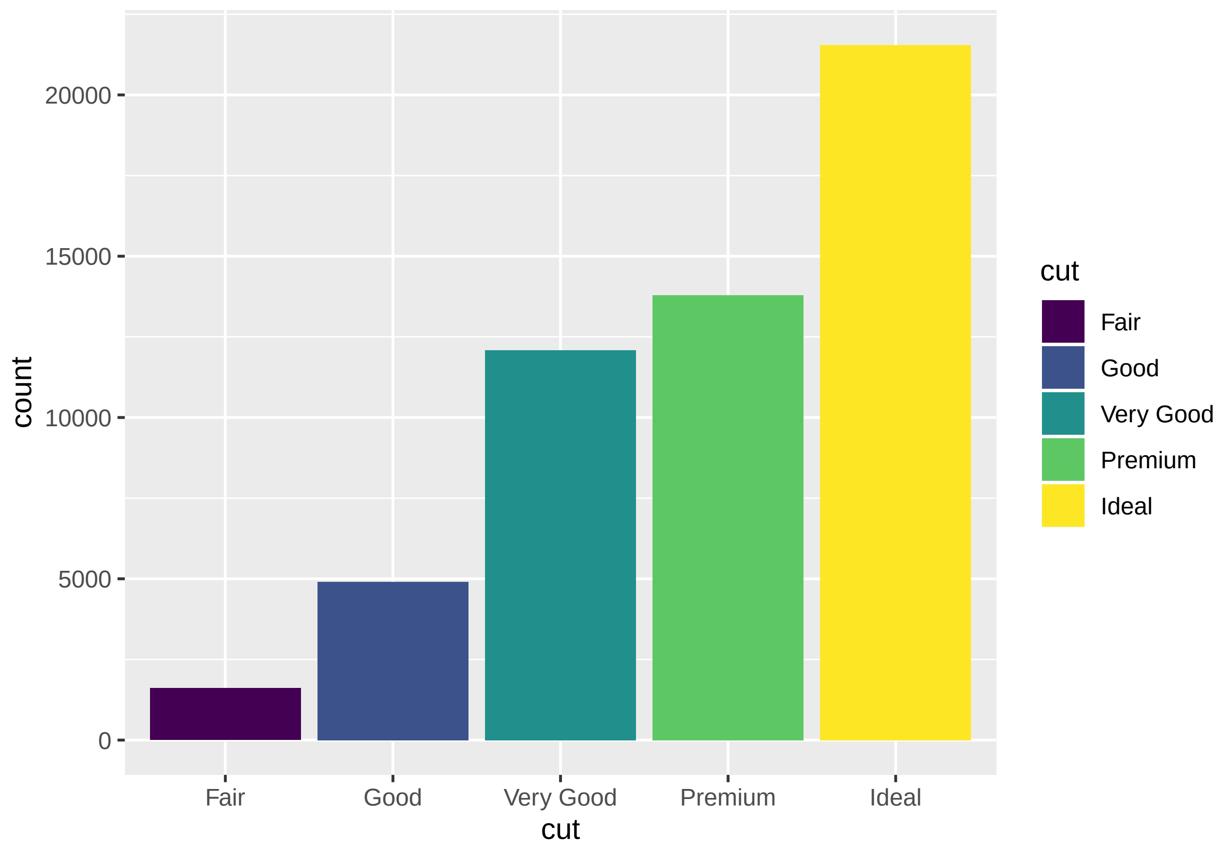
POSITION ADJUSTMENT

We can color a bar chart using either colour aesthetic or more usefully fill.

```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, color = cut))
```

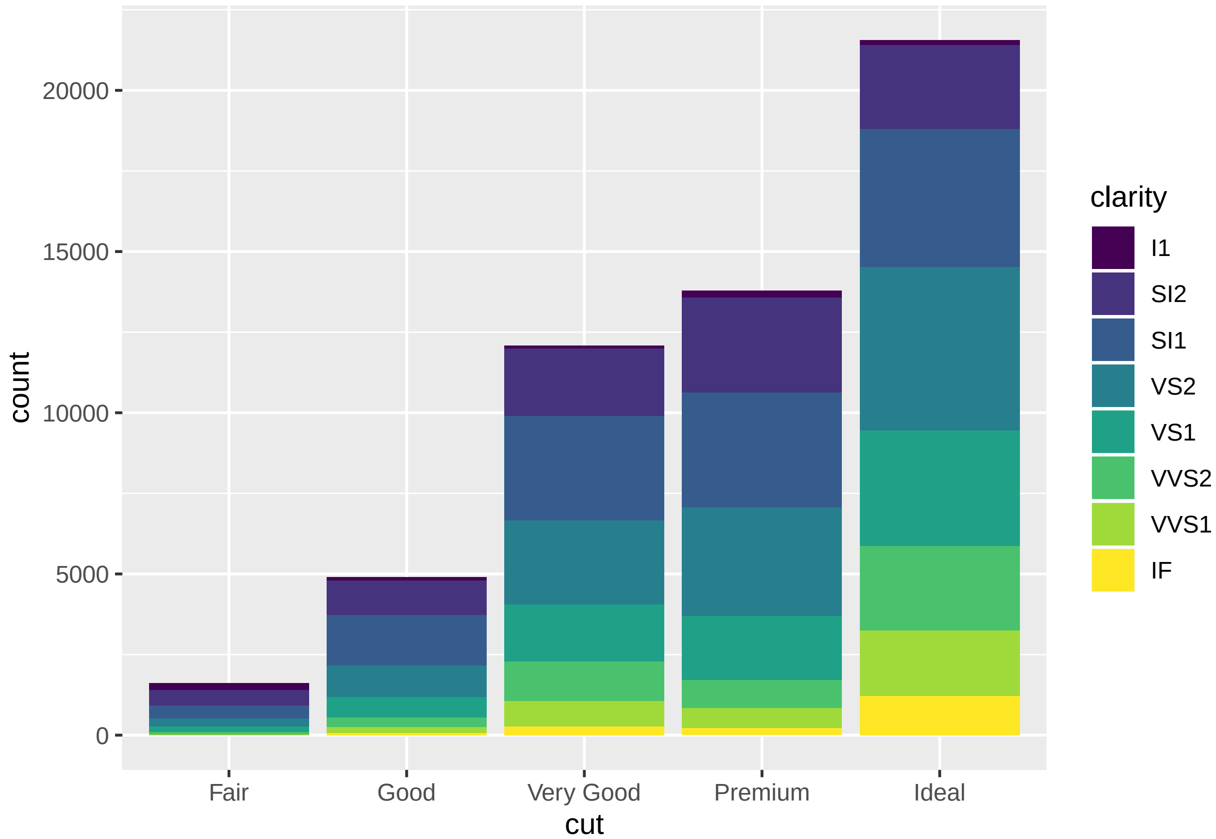



```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, fill = cut))
```



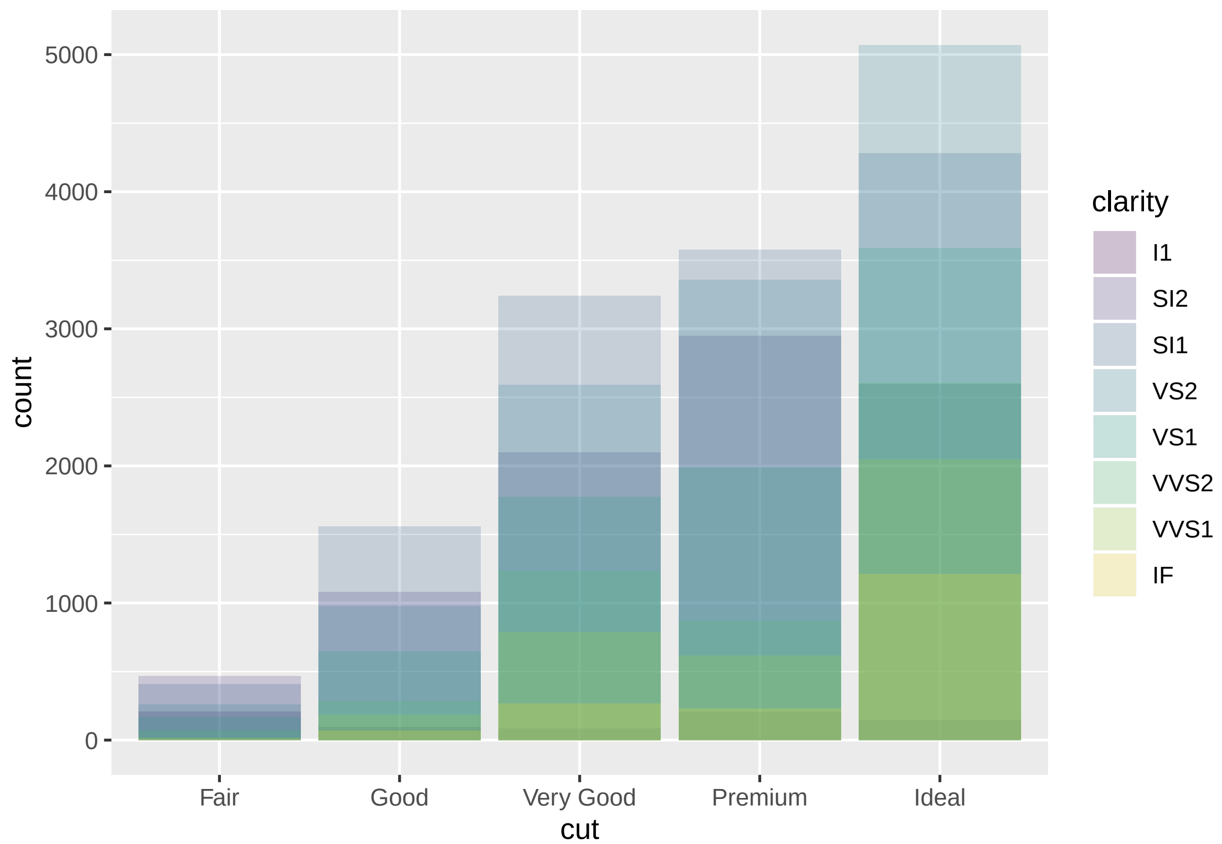
If we were to map the fill aesthetic to another variable like clarity, the bars are automatically stacked. Each colored rectangle represents a combination of cut and clarity

```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, fill = clarity))
```

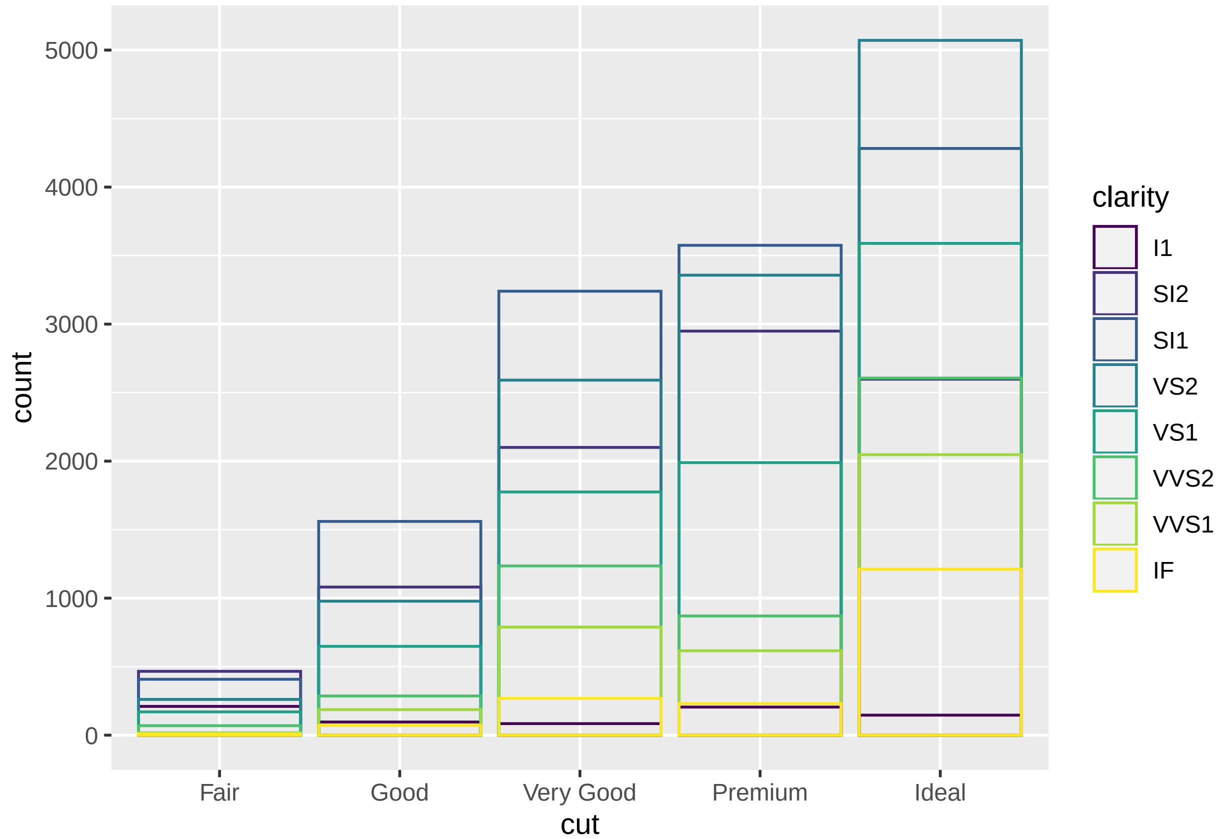


`position = "identity"` places each object exactly where it falls in the context of the graph. In bars, however, to see the overlapping we either need to make the bars slightly transparent by setting alpha to a small value, or completely transparent by setting `fill = NA`.

```
ggplot(data = diamonds, mapping = aes(x = cut, fill = clarity)) + geom_bar(alpha = 1/5, position = "iden
```

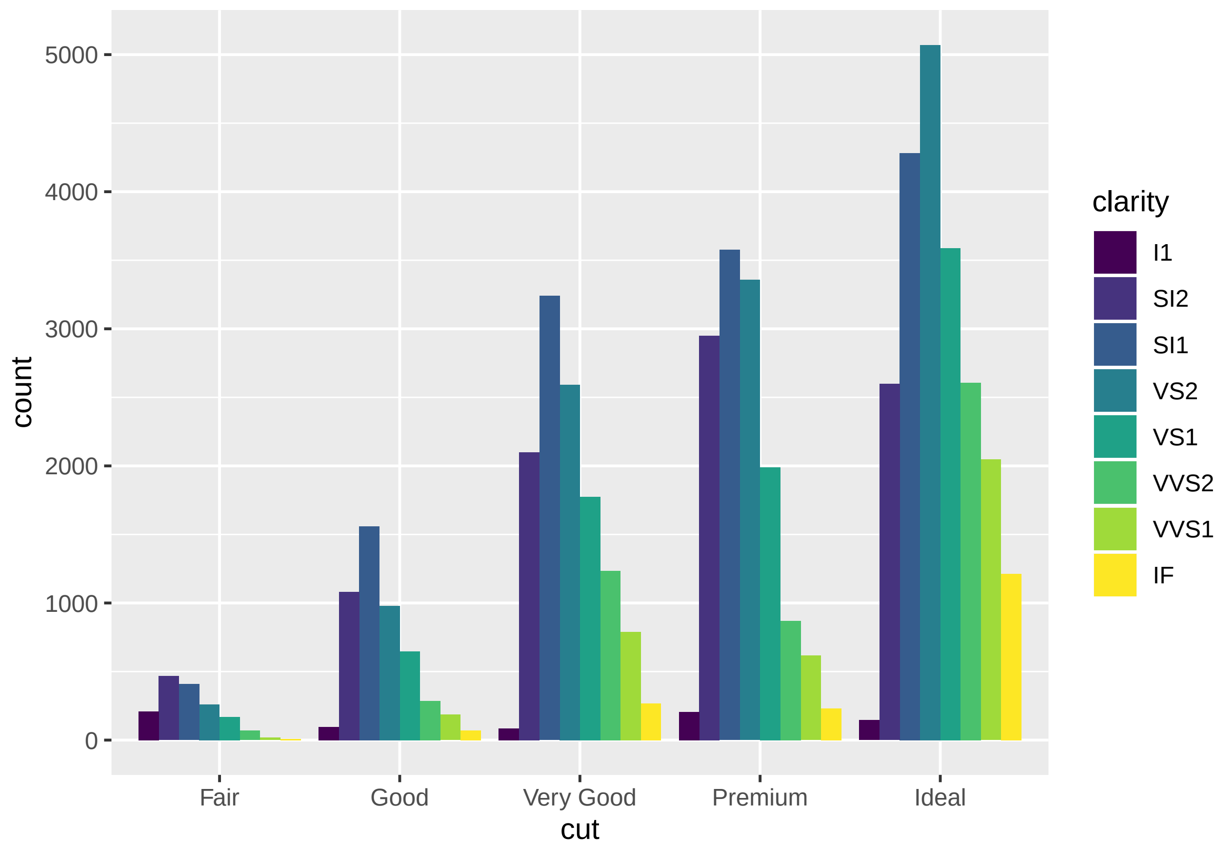



```
ggplot(data = diamonds, mapping = aes(x = cut, color = clarity)) + geom_bar(fill = NA, position = "iden
```



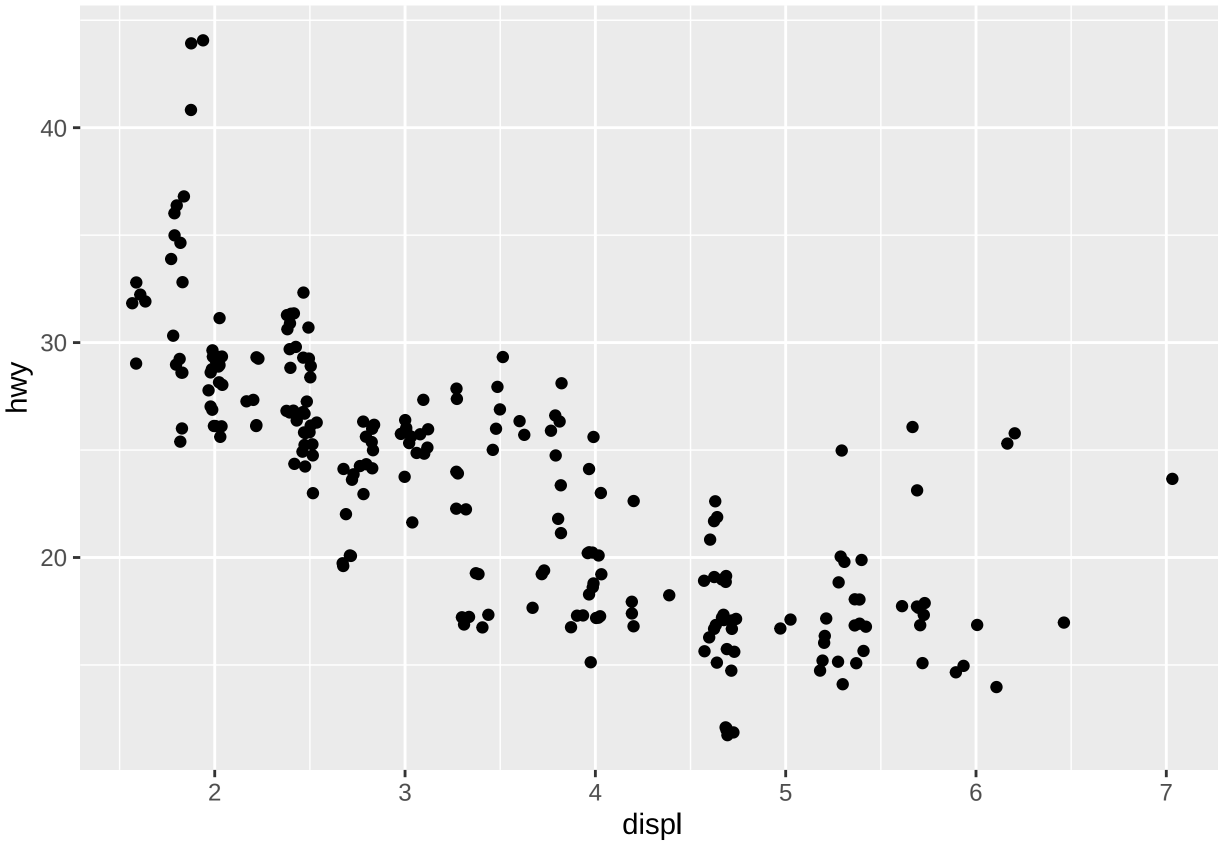
`position = "dodge"` places overlapping objects directly beside one another. This makes it easier to compare individual values

```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, fill = clarity), position = "dodge")
```



`position = "jitter` adds a small amount of random noise to each point. This spreads the points out because no two points are likely to receive the same amount of random noise

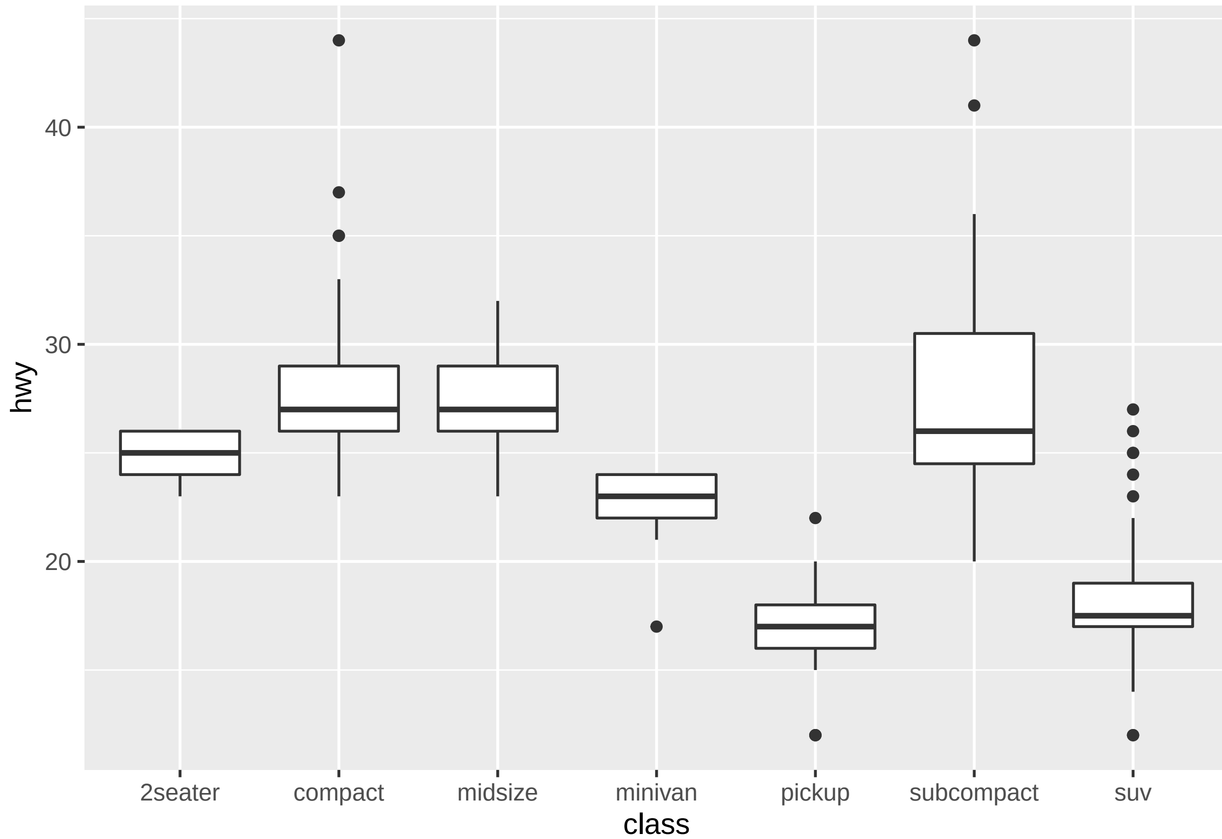
```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy), position = "jitter")
```



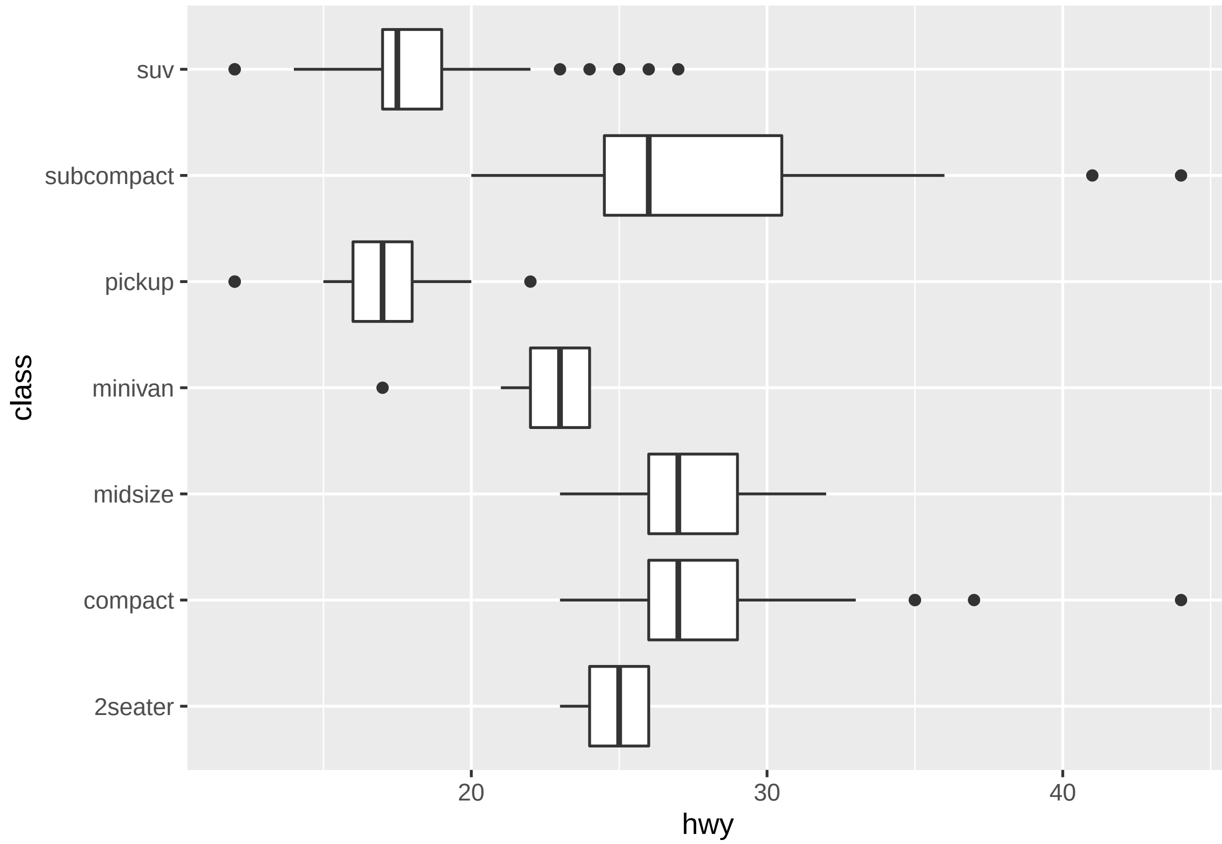
COORDINATE SYSTEMS

`coord_flip()` switches the x and y axes. This is useful for horizontal boxplots and for long labels

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) + geom_boxplot()
```




```r
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) + geom_boxplot() + coord_flip()
```



DATA TRANSFORMATION

```
library(nycflights13)
library(tidyverse)
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Filter rows with `filter()`

`filter()` allows us to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame. For example, we can select all flights on January 1st with:

```r
filter(flights, month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 832 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

`filter(flights, month == 11 | month == 12)` finds all flights that departed either in November or December `nov_dec <- filter(flights, month %in% c(11, 12))` also does the same thing.

Missing Values

`filter()` only includes rows where the condition is true; it excludes both false and NA values. If we want to preserve these missing values, we need to ask for them explicitly.

Arrange rows with `arrange()`

`arrange()` works similarly to filter() except that instead of selecting rows, it changes their order.

```r
arrange(flights, year, month, day)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
```

```
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Use 'desc()' to re-order a column in descending order

```r
arrange(flights, desc(dep_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     9      641            900      1301     1242
##  2  2013     6    15     1432           1935      1137     1607
##  3  2013     1    10     1121           1635      1126     1239
##  4  2013     9    20     1139           1845      1014     1457
##  5  2013     7    22      845           1600      1005     1044
##  6  2013     4    10     1100           1900       960     1342
##  7  2013     3    17     2321            810       911      135
##  8  2013     6    27      959           1900       899     1236
##  9  2013     7    22     2257            759       898      121
## 10  2013    12     5      756           1700       896     1058
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Select columns with `select()`

Select columns by name

```r
select(flights, year, month, day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Select all columns between year and day (inclusive)

```r
select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Select all columns except those from year to day (inclusive)

```r
select(flights, -(year:day))
```

```
## # A tibble: 336,776 x 16
##    dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
##       <int>          <int>     <dbl>    <int>          <int>     <dbl>
##  1      517            515         2      830            819        11
##  2      533            529         4      850            830        20
##  3      542            540         2      923            850        33
##  4      544            545        -1     1004           1022       -18
##  5      554            600        -6      812            837       -25
##  6      554            558        -4      740            728        12
##  7      555            600        -5      913            854        19
##  8      557            600        -3      709            723       -14
##  9      557            600        -3      838            846        -8
## 10      558            600        -2      753            745         8
## # ... with 336,766 more rows, and 10 more variables: carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
## #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

`starts_with("abc")` matches names that begin with "abc". `ends_with("xyz")` matches names that end with "xyz". `contains("xyz")` matches names that contain "xyz". `matches("(.)\\1")` selects variables that match a regular expression. `num_range("x", 1:3)` matches x1. x2 and x2.

`rename()` can be used to rename a variiable

```r
rename(flights, tail_num = tailnum)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
```

```
## 4  2013     1     1      544            545        -1     1004
## 5  2013     1     1      554            600        -6      812
## 6  2013     1     1      554            558        -4      740
## 7  2013     1     1      555            600        -5      913
## 8  2013     1     1      557            600        -3      709
## 9  2013     1     1      557            600        -3      838
## 10 2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tail_num <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Using `select()` with `everything()` helper allows us to move variables to the start of the data frame.

```r
select(flights, time_hour, air_time, everything())
```

```
## # A tibble: 336,776 x 19
##    time_hour           air_time  year month   day dep_time sched_dep_time
##    <dttm>                 <dbl> <int> <int> <int>    <int>          <int>
##  1 2013-01-01 05:00:00      227  2013     1     1      517            515
##  2 2013-01-01 05:00:00      227  2013     1     1      533            529
##  3 2013-01-01 05:00:00      160  2013     1     1      542            540
##  4 2013-01-01 05:00:00      183  2013     1     1      544            545
##  5 2013-01-01 06:00:00      116  2013     1     1      554            600
##  6 2013-01-01 05:00:00      150  2013     1     1      554            558
##  7 2013-01-01 06:00:00      158  2013     1     1      555            600
##  8 2013-01-01 06:00:00       53  2013     1     1      557            600
##  9 2013-01-01 06:00:00      140  2013     1     1      557            600
## 10 2013-01-01 06:00:00      138  2013     1     1      558            600
## # ... with 336,766 more rows, and 12 more variables: dep_delay <dbl>,
## #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, distance <dbl>,
## #   hour <dbl>, minute <dbl>
```

Add new variables with mutate()

It is often useful to add new columns that are functions of existing columns

```r
flights_small <- select(flights, year:day, ends_with("delay"), distance, air_time)
mutate(flights_small, gain = dep_delay - arr_delay, speed = distance / air_time * 60)
```

```
## # A tibble: 336,776 x 9
##     year month   day dep_delay arr_delay distance air_time  gain speed
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
##  1  2013     1     1         2        11     1400      227    -9  370.
##  2  2013     1     1         4        20     1416      227   -16  374.
##  3  2013     1     1         2        33     1089      160   -31  408.
##  4  2013     1     1        -1       -18     1576      183    17  517.
##  5  2013     1     1        -6       -25      762      116    19  394.
##  6  2013     1     1        -4        12      719      150   -16  288.
##  7  2013     1     1        -5        19     1065      158   -24  404.
##  8  2013     1     1        -3       -14      229       53    11  259.
```

```
## 9 2013 1 1 -3 -8 944 140 5 405.
## 10 2013 1 1 -2 8 733 138 -10 319.
## # ... with 336,766 more rows
```

We can also refer to columns that we have just created

```
mutate(flights_small, gain = dep_delay - arr_delay, hours = air_time / 60, gain_per_hour = gain / hours)
```

```
## # A tibble: 336,776 x 10
##     year month   day dep_delay arr_delay distance air_time  gain hours
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
##  1  2013     1     1         2        11     1400      227    -9 3.78
##  2  2013     1     1         4        20     1416      227   -16 3.78
##  3  2013     1     1         2        33     1089      160   -31 2.67
##  4  2013     1     1        -1       -18     1576      183    17 3.05
##  5  2013     1     1        -6       -25      762      116    19 1.93
##  6  2013     1     1        -4        12      719      150   -16 2.5
##  7  2013     1     1        -5        19     1065      158   -24 2.63
##  8  2013     1     1        -3       -14      229       53    11 0.883
##  9  2013     1     1        -3        -8      944      140     5 2.33
## 10  2013     1     1        -2         8      733      138   -10 2.3
## # ... with 336,766 more rows, and 1 more variable: gain_per_hour <dbl>
```

If we only want to keep the new variables, use transmute()

```
transmute(flights, gain = dep_delay - arr_delay, hours = air_time / 60, gain_per_hour = gain / hours)
```

```
## # A tibble: 336,776 x 3
##     gain hours gain_per_hour
##    <dbl> <dbl>         <dbl>
##  1    -9 3.78          -2.38
##  2   -16 3.78          -4.23
##  3   -31 2.67         -11.6
##  4    17 3.05           5.57
##  5    19 1.93           9.83
##  6   -16 2.5           -6.4
##  7   -24 2.63          -9.11
##  8    11 0.883         12.5
##  9     5 2.33           2.14
## 10   -10 2.3           -4.35
## # ... with 336,766 more rows
```

Grouped summaries with summarise()

`summarise()` collapses a dataframe to a single row.

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1  12.6
```

`summarise()` is only useful if we pair it with `group_by()`. To get the average delay per date:

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day delay
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
## 10  2013     1    10  2.84
## # ... with 355 more rows
```

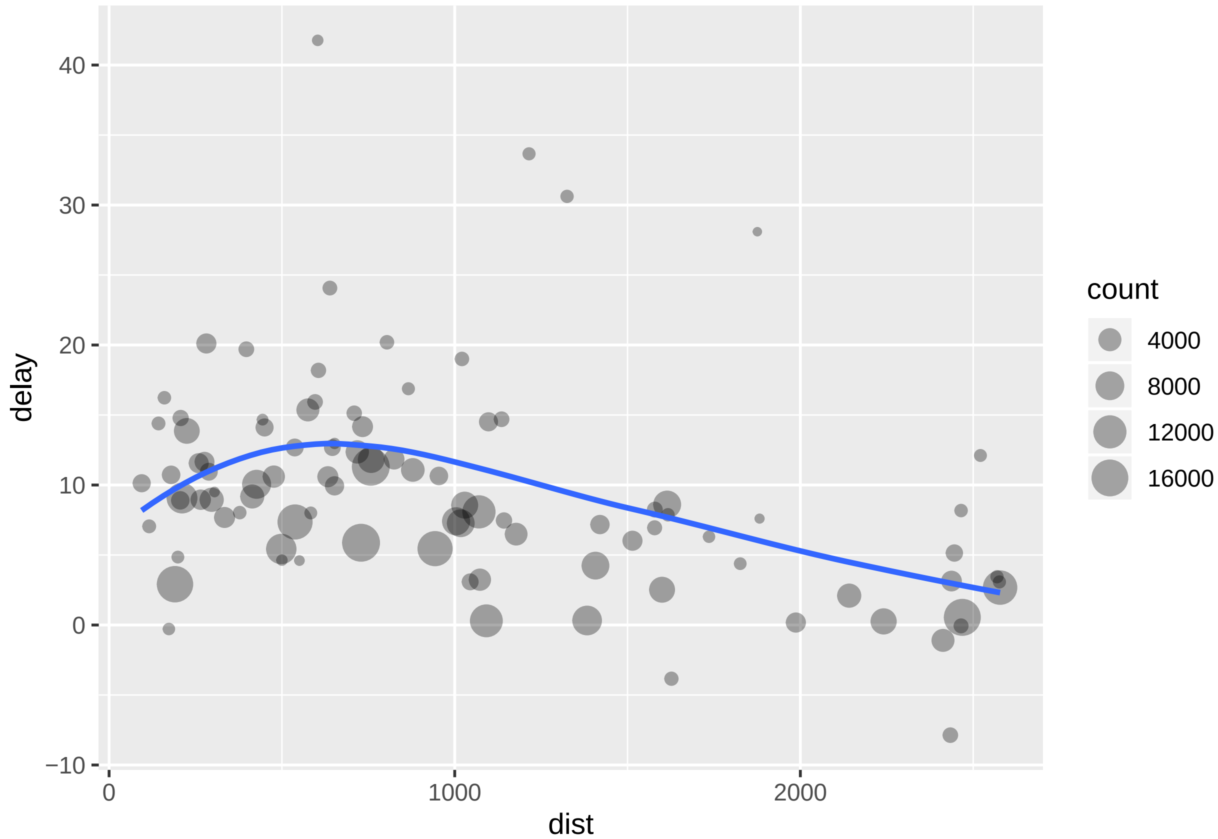
Combining multiple operations with pipe

We want to explore the relationship between the distance and average delay for each location. Our initial solution (without piping) would look like this:

```
by_dest <- group_by(flights, dest)
delay <- summarise(by_dest, count = n(), dist = mean(distance, na.rm = TRUE), delay = mean(arr_delay, na
delay <- filter(delay, count > 20, dest != "HNL")

# It looks like delays increase with distance up to ~750 miles and then decreases. This could be becaus

ggplot(data = delay, mapping = aes(x = dist, y = delay)) + geom_point(aes(size = count), alpha = 1/3) +
```



Using pipes, we can more efficiently write the code as:

```
delays <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  )%>%
  filter(count > 20, dest != "HNL")
```

Missing values

If we do not set the missing values:

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1    NA
##  2  2013     1     2    NA
```

```
##  3  2013     1     3    NA
##  4  2013     1     4    NA
##  5  2013     1     5    NA
##  6  2013     1     6    NA
##  7  2013     1     7    NA
##  8  2013     1     8    NA
##  9  2013     1     9    NA
## 10  2013     1    10    NA
## # ... with 355 more rows
```

If we do set the missing values:

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
## 10  2013     1    10  2.84
## # ... with 355 more rows
```

In the case where missing values represent cancelled flights, we could also first of all remove the cancelled flights.

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day  mean
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.4
##  2  2013     1     2 13.7
##  3  2013     1     3 10.9
##  4  2013     1     4  8.97
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
```

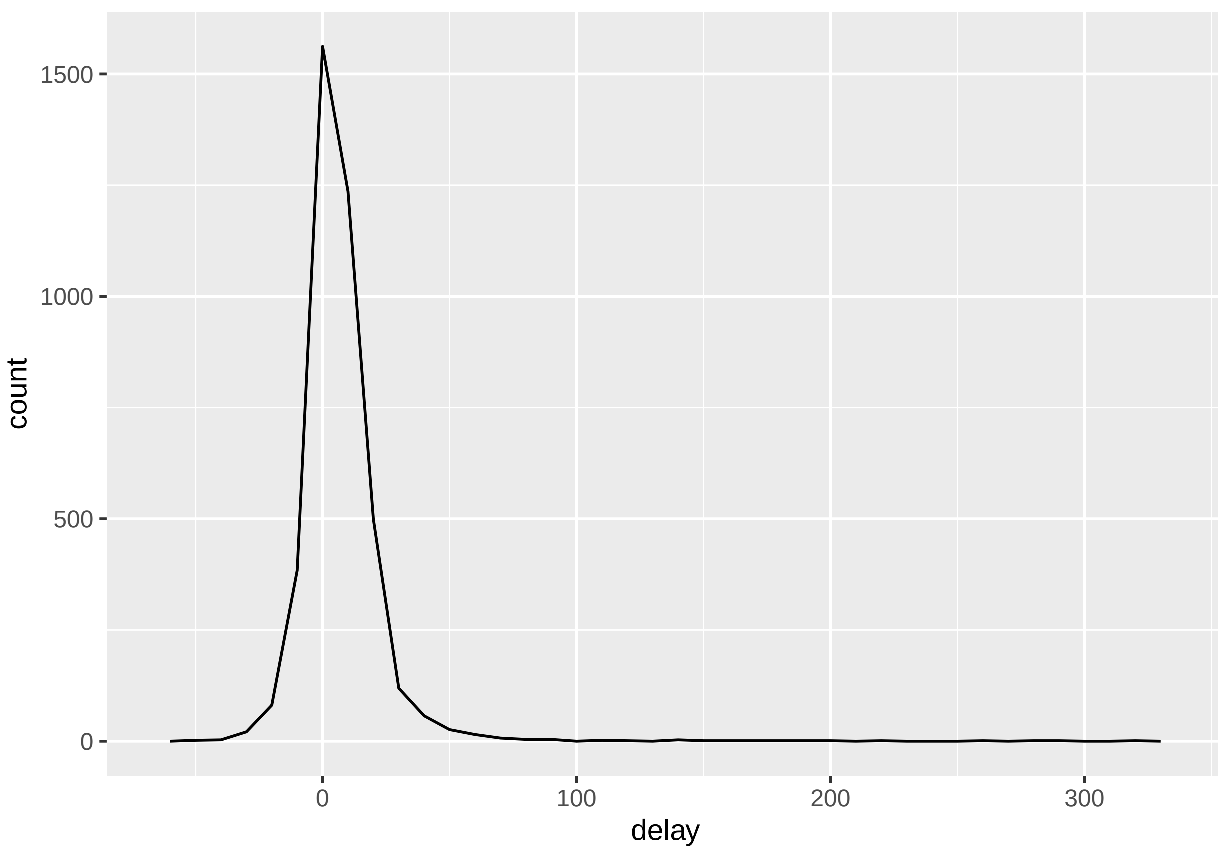
```
## 8  2013     1     8  2.56
## 9  2013     1     9  2.30
## 10 2013     1    10  2.84
## # ... with 355 more rows
```

Count

To look at planes (identified by their tail number) that have the highest average delays.

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay)
  )
```

```
ggplot(data = delays, mapping = aes(x = delay)) + geom_freqpoly(binwidth = 10)
```



To view when the first and last flights leave each day:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first = min(dep_time),
    last = max(dep_time)
  )
```

```
## # A tibble: 365 x 5
## # Groups:   year, month [12]
##     year month   day first  last
##    <int> <int> <int> <int> <int>
##  1  2013     1     1   517  2356
##  2  2013     1     2    42  2354
##  3  2013     1     3    32  2349
##  4  2013     1     4    25  2358
##  5  2013     1     5    14  2357
##  6  2013     1     6    16  2355
##  7  2013     1     7    49  2359
##  8  2013     1     8   454  2351
##  9  2013     1     9     2  2252
## 10  2013     1    10     3  2320
## # ... with 355 more rows
```

`n()` takes no arguments and returns the size of the current group. To count the number of non-missing values, use `sum(!is.na(x))`. To count the number of distinct or unique values, use `n_distinct(x)`.

```
# which destination has the most carriers
not_cancelled %>%
  group_by(dest) %>%
  summarise(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
```

```
## # A tibble: 104 x 2
##    dest  carriers
##    <chr>    <int>
##  1 ATL          7
##  2 BOS          7
##  3 CLT          7
##  4 ORD          7
##  5 TPA          7
##  6 AUS          6
##  7 DCA          6
##  8 DTW          6
##  9 IAD          6
## 10 MSP          6
## # ... with 94 more rows
```

Counts are so useful that dplyr provides a simple helper if all we want is count

```
not_cancelled %>%
  count(dest)
```

```
## # A tibble: 104 x 2
##    dest      n
##    <chr> <int>
##  1 ABQ     254
##  2 ACK     264
##  3 ALB     418
##  4 ANC       8
##  5 ATL   16837
```

```
##   6 AUS    2411
##   7 AVL     261
##   8 BDL     412
##   9 BGR     358
##  10 BHM     269
## # ... with 94 more rows
```

A weight variable is used if want to for example, count(sum) the total number of miles a plane flew

```r
not_cancelled %>%
  count(tailnum, wt = distance)
```

```
## # A tibble: 4,037 x 2
##    tailnum      n
##    <chr>    <dbl>
##  1 D942DN    3418
##  2 N0EGMQ  239143
##  3 N10156  109664
##  4 N102UW   25722
##  5 N103US   24619
##  6 N104UW   24616
##  7 N10575  139903
##  8 N105UW   23618
##  9 N107US   21677
## 10 N108UW   32070
## # ... with 4,027 more rows
```

How many flights left before 5 am?

```r
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(n_early = sum(dep_time < 500))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day n_early
##    <int> <int> <int>   <int>
##  1  2013     1     1       0
##  2  2013     1     2       3
##  3  2013     1     3       4
##  4  2013     1     4       3
##  5  2013     1     5       3
##  6  2013     1     6       2
##  7  2013     1     7       2
##  8  2013     1     8       1
##  9  2013     1     9       3
## 10  2013     1    10       3
## # ... with 355 more rows
```

What proportion of flights are delayed by more than one hour?

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(hour_perc = mean(arr_delay > 60))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day hour_perc
##    <int> <int> <int>     <dbl>
##  1  2013     1     1    0.0722
##  2  2013     1     2    0.0851
##  3  2013     1     3    0.0567
##  4  2013     1     4    0.0396
##  5  2013     1     5    0.0349
##  6  2013     1     6    0.0470
##  7  2013     1     7    0.0333
##  8  2013     1     8    0.0213
##  9  2013     1     9    0.0202
## 10  2013     1    10    0.0183
## # ... with 355 more rows
```

Grouping by multiple variables

```
daily <- group_by(flights, year, month, day)
(per_day <- summarise(daily, flights = n()))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day flights
##    <int> <int> <int>   <int>
##  1  2013     1     1     842
##  2  2013     1     2     943
##  3  2013     1     3     914
##  4  2013     1     4     915
##  5  2013     1     5     720
##  6  2013     1     6     832
##  7  2013     1     7     933
##  8  2013     1     8     899
##  9  2013     1     9     902
## 10  2013     1    10     932
## # ... with 355 more rows
```

```
per_month <- summarise(per_day, flights = sum(flights))
per_year <- summarise(per_month, flights = sum(flights))
```

Ungrouping

```
daily %>%
  ungroup() %>%
  summarise(flights = n())
```

```
## # A tibble: 1 x 1
##   flights
##     <int>
## 1  336776
```

Grouped mutates and filters

Find the worst members of each group

```
flights_small %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 10)
```

```
## # A tibble: 3,306 x 7
## # Groups:   year, month, day [365]
##     year month   day dep_delay arr_delay distance air_time
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
##  1  2013     1     1       853       851      184       41
##  2  2013     1     1       290       338     1134      213
##  3  2013     1     1       260       263      266       46
##  4  2013     1     1       157       174      213       60
##  5  2013     1     1       216       222      708      121
##  6  2013     1     1       255       250      589      115
##  7  2013     1     1       285       246     1085      146
##  8  2013     1     1       192       191      199       44
##  9  2013     1     1       379       456     1092      222
## 10  2013     1     2       224       207      550       94
## # ... with 3,296 more rows
```

Find all the groups bigger than a threshold

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
popular_dests
```

```
## # A tibble: 332,577 x 19
## # Groups:   dest [77]
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 332,567 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
```

```
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Standardise to compute per group metrics

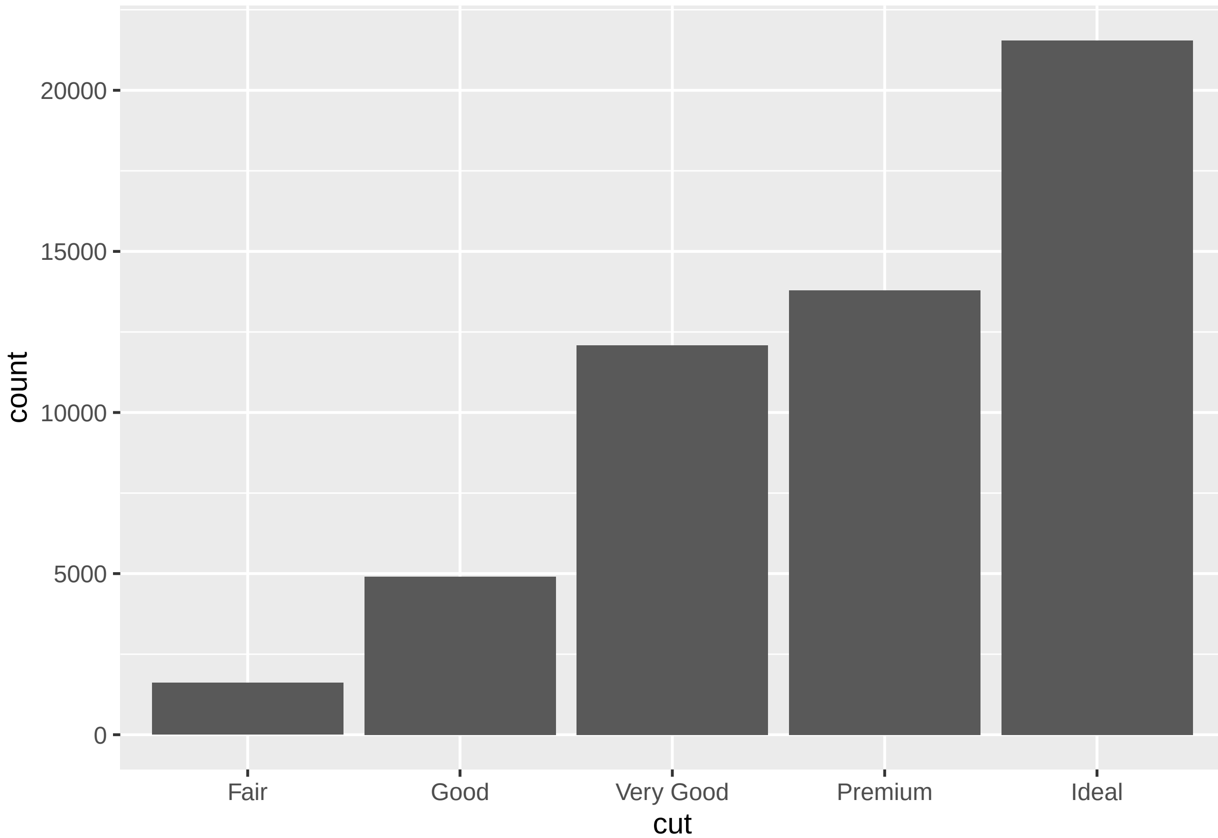
```
popular_dests %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
  select(year:day, dest, arr_delay, prop_delay)
```

```
## # A tibble: 131,106 x 6
## # Groups:   dest [77]
##     year month   day dest  arr_delay prop_delay
##    <int> <int> <int> <chr>     <dbl>      <dbl>
##  1  2013     1     1 IAH          11  0.000111
##  2  2013     1     1 IAH          20  0.000201
##  3  2013     1     1 MIA          33  0.000235
##  4  2013     1     1 ORD          12  0.0000424
##  5  2013     1     1 FLL          19  0.0000938
##  6  2013     1     1 ORD           8  0.0000283
##  7  2013     1     1 LAX           7  0.0000344
##  8  2013     1     1 DFW          31  0.000282
##  9  2013     1     1 ATL          12  0.0000400
## 10  2013     1     1 DTW          16  0.000116
## # ... with 131,096 more rows
```

EXPLORATORY DATA ANALYSIS

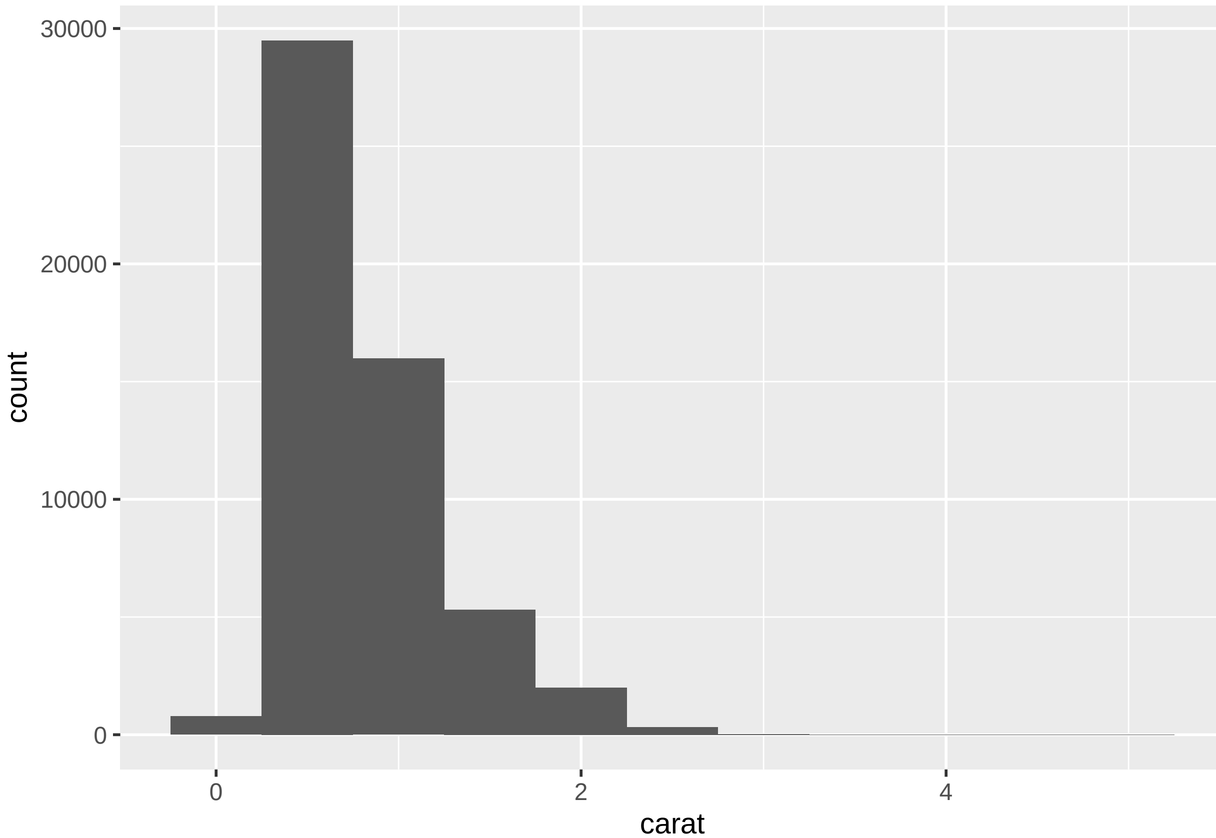
Categorical variables can only take a small set of values. In R, categorical variables are usually saved as factors or character vectors.

```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut))
```



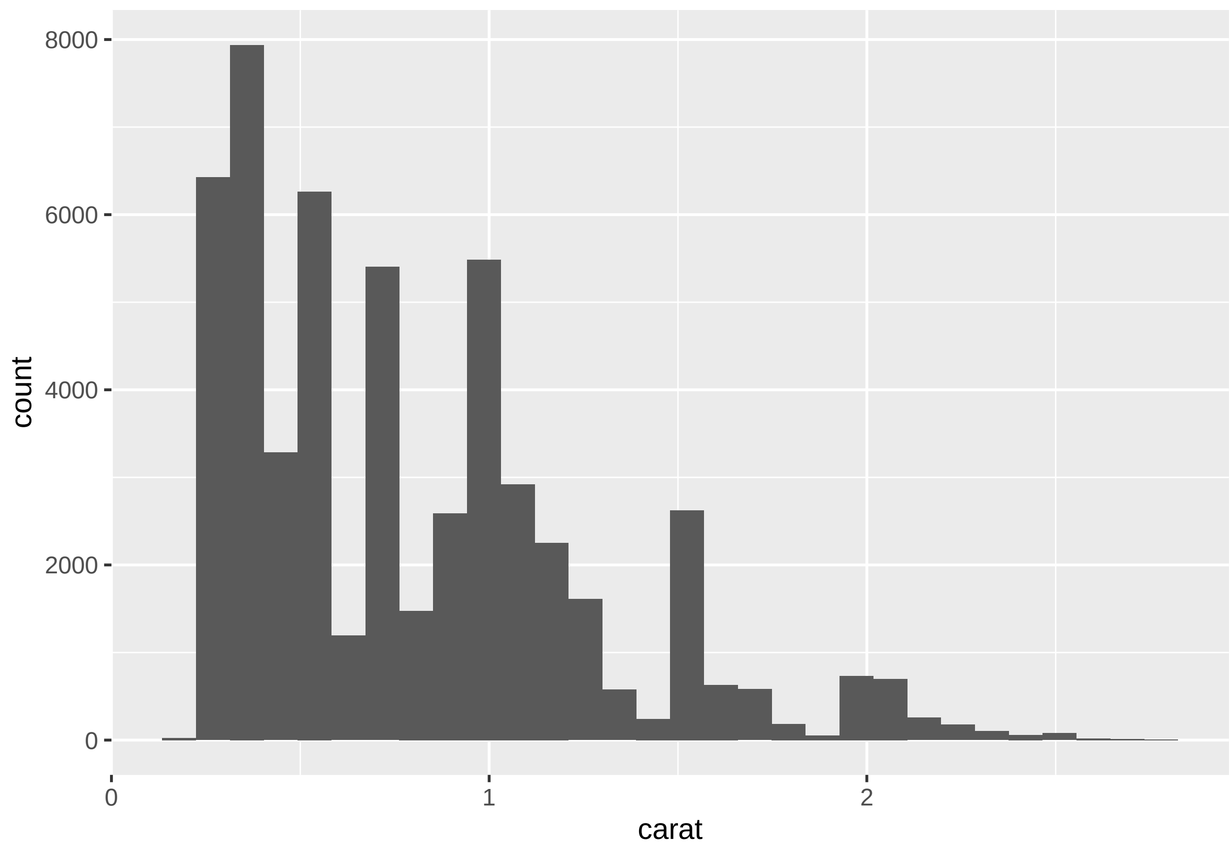
Continuous variables can take any of an infinite set of ordered values. Numbers and date-times are two examples of continuous variables.

```
ggplot(data = diamonds) + geom_histogram(mapping = aes(x = carat), binwidth = 0.5)
```



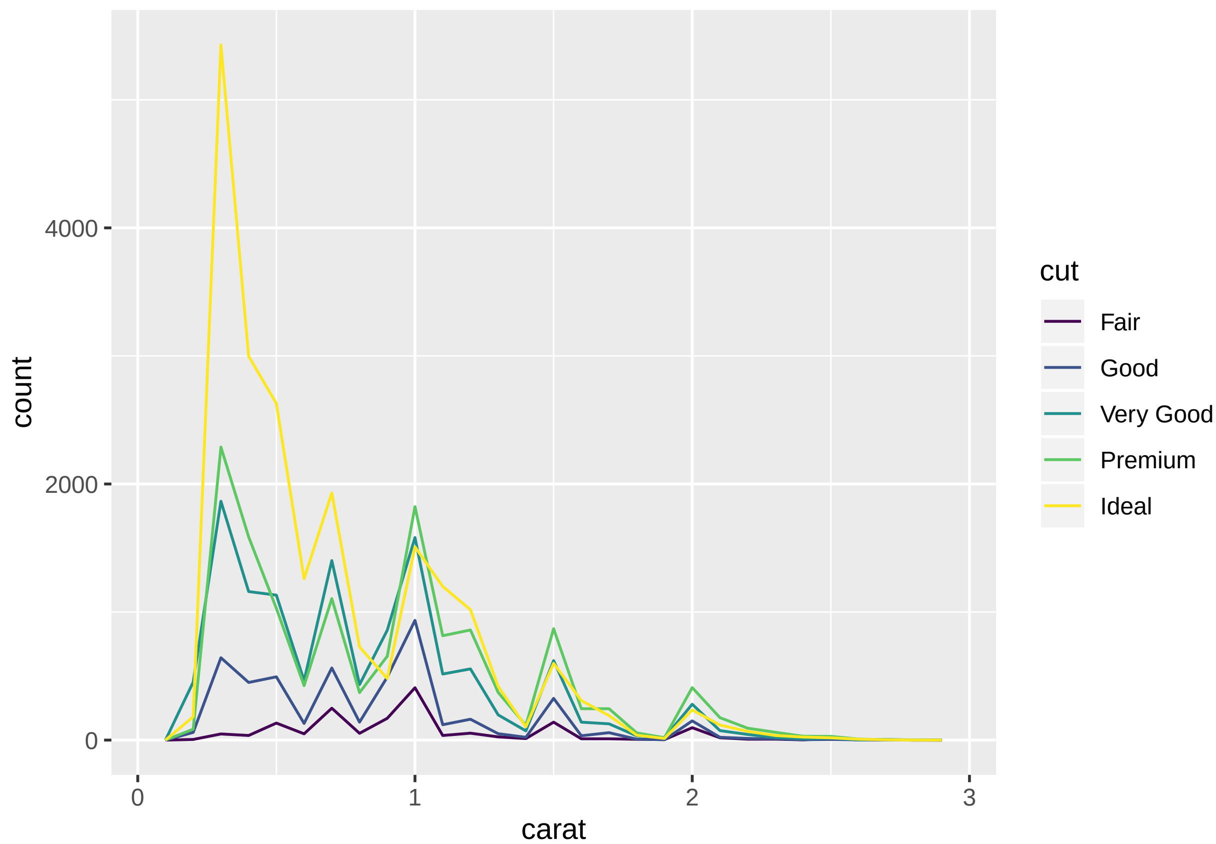
We should explore a variety of binwidths when working with histograms, as different binwidths can reveal different patters. For example, here is how the graph above looks when we zoom into just the diamonds with a size of less than three carats and choose a smaller binwidth.

```
smaller <- diamonds %>%
  filter(carat < 3)
ggplot(data = smaller, mapping = aes(x = carat)) + geom_histogram(bindwidth = 0.1)
```

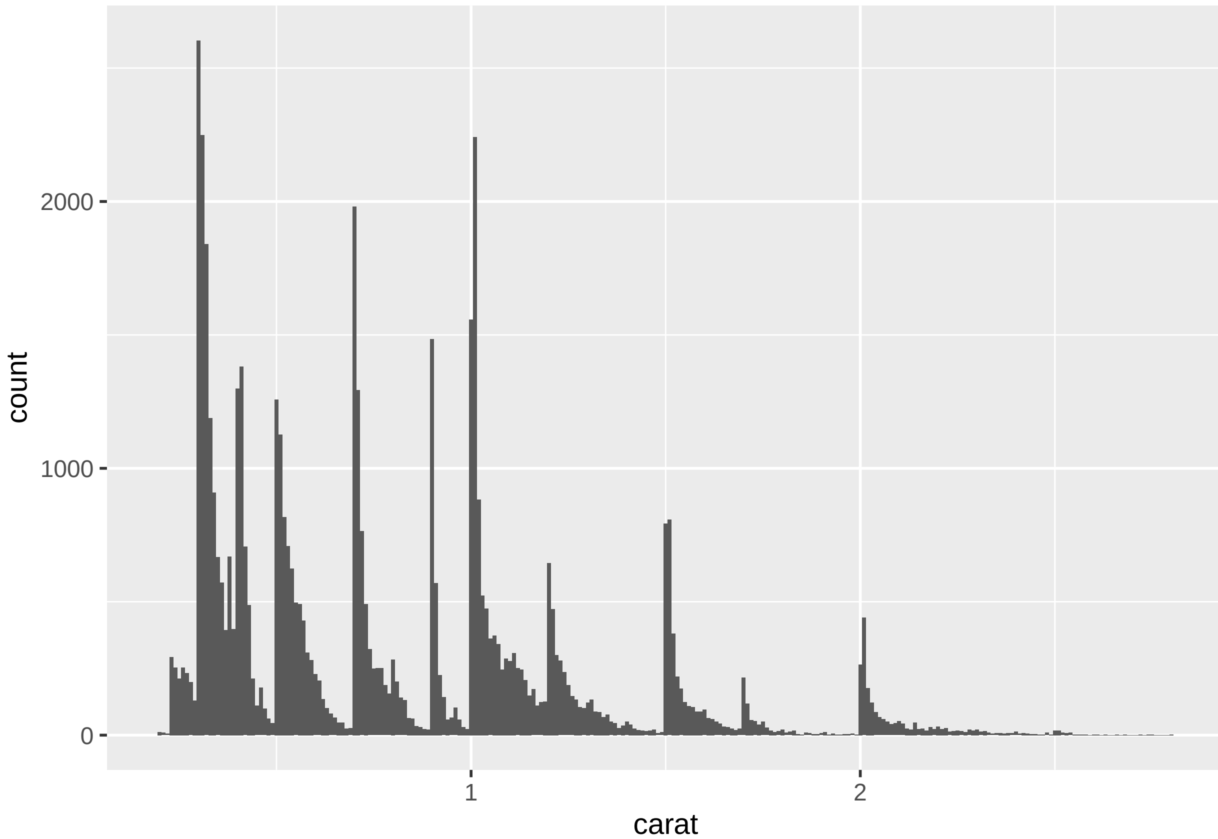


`geom_freqpoly()` performs the same calculation as `geom_histogram()` but instead of displaying the counts with bars, uses lines instead. It is much easier to understand overlapping lines than bars

```
ggplot(data = smaller, mapping = aes(x = carat, color = cut)) + geom_freqpoly(binwidth = 0.1)
```



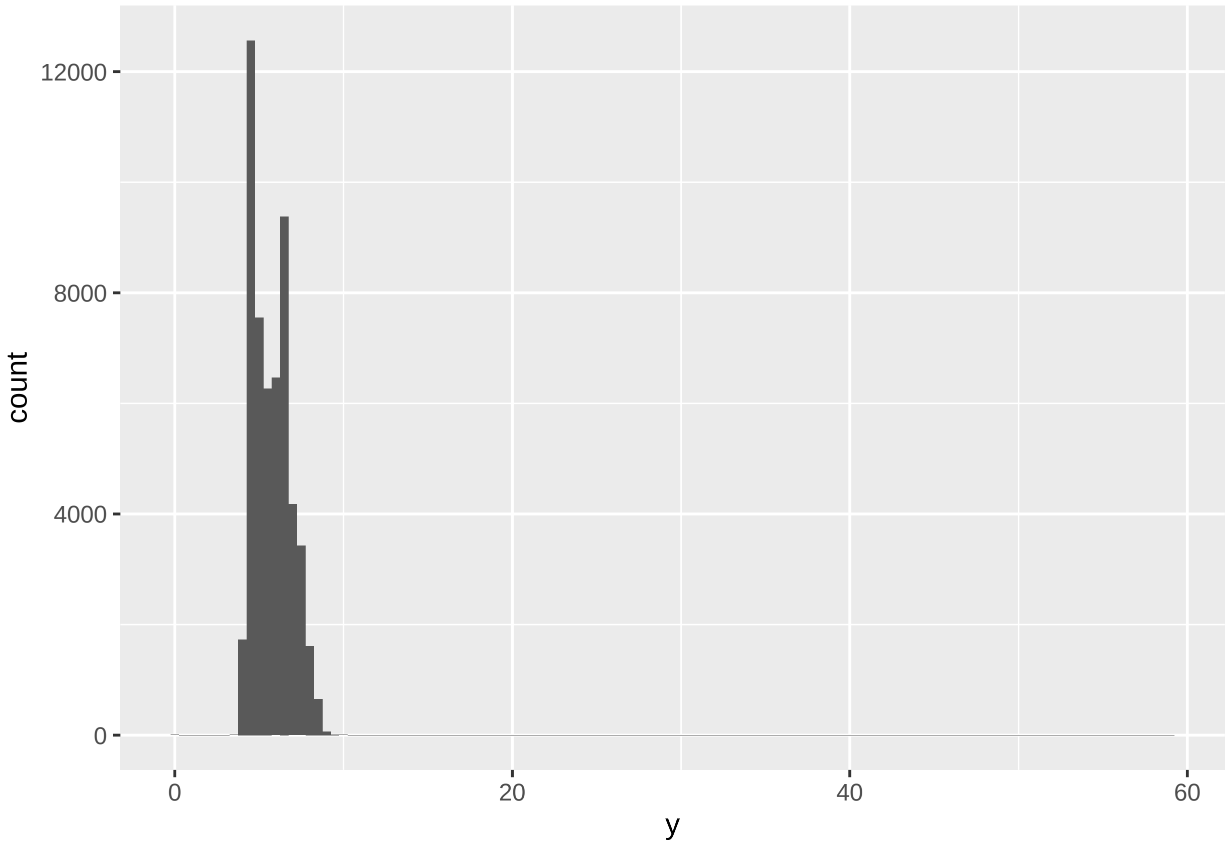

```r
ggplot(data = smaller, mapping = aes(x = carat)) + geom_histogram(binwidth = 0.01)
```



Outliers

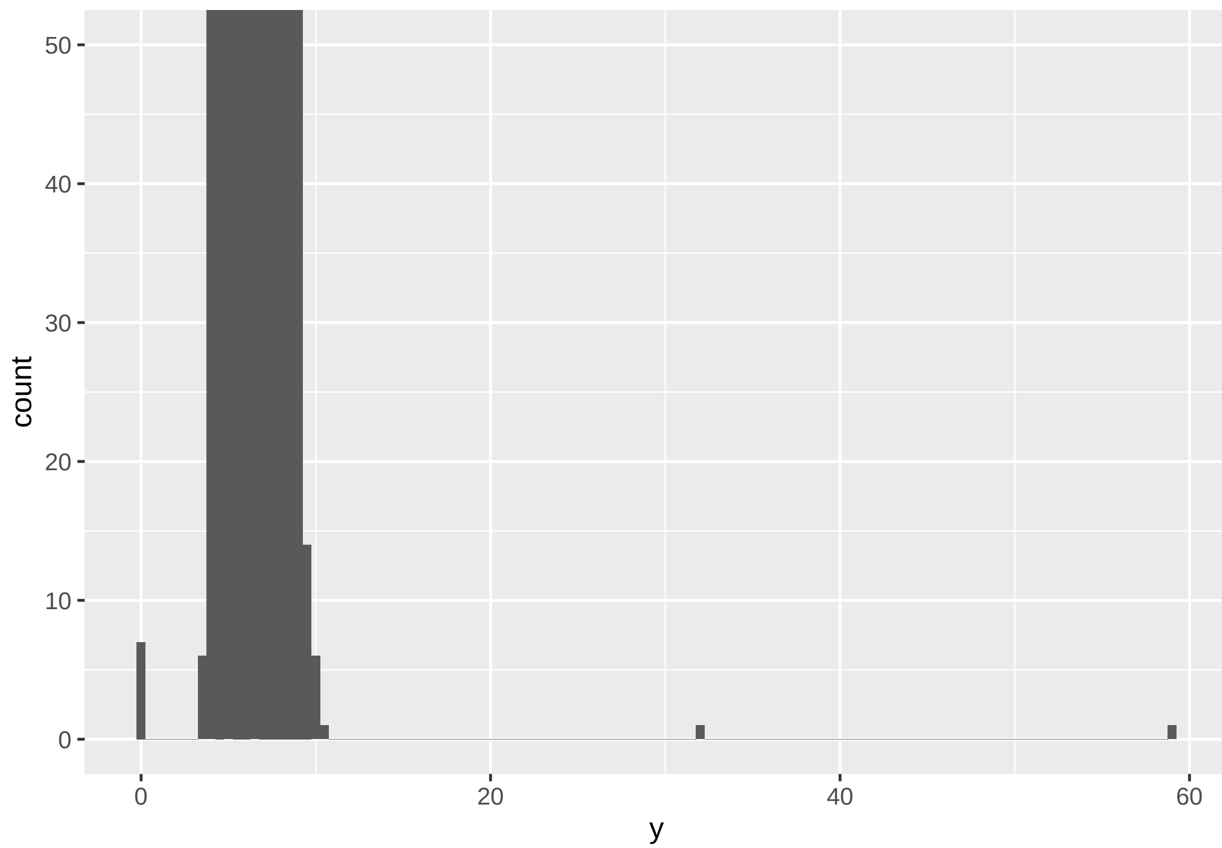
The only evidence of outliers is the unusually wide limits on the x-axis

```
ggplot(diamonds) + geom_histogram(mapping = aes(x = y), binwidth = 0.5)
```



To be able to see the outliers, we need to zoom to small values of the y-axis with `coord_cartesian()`

```
ggplot(diamonds) + geom_histogram(mapping = aes(x = y), binwidth = 0.5) + coord_cartesian(ylim = c(0, 5
```



This allows us to see that there are three unusual values: 0, 30 and 60. We can pluck them out with dplyr

```
unusual <- diamonds %>%
  filter(y < 3 | y > 20) %>%
  select(price, x, y, z) %>%
  arrange(y)
unusual
```

```
## # A tibble: 9 x 4
##   price     x     y     z
##   <int> <dbl> <dbl> <dbl>
## 1  5139  0      0    0
## 2  6381  0      0    0
## 3 12800  0      0    0
## 4 15686  0      0    0
## 5 18034  0      0    0
## 6  2130  0      0    0
## 7  2130  0      0    0
## 8  2075  5.15  31.8  5.12
## 9 12210  8.09  58.9  8.06
```

If we encounter unusual values in the dataset, we have two options: 1. Drop the entire row with the strange values

```r
diamonds2 <- diamonds %>%
  filter(between(y, 3, 20))
diamonds2
```

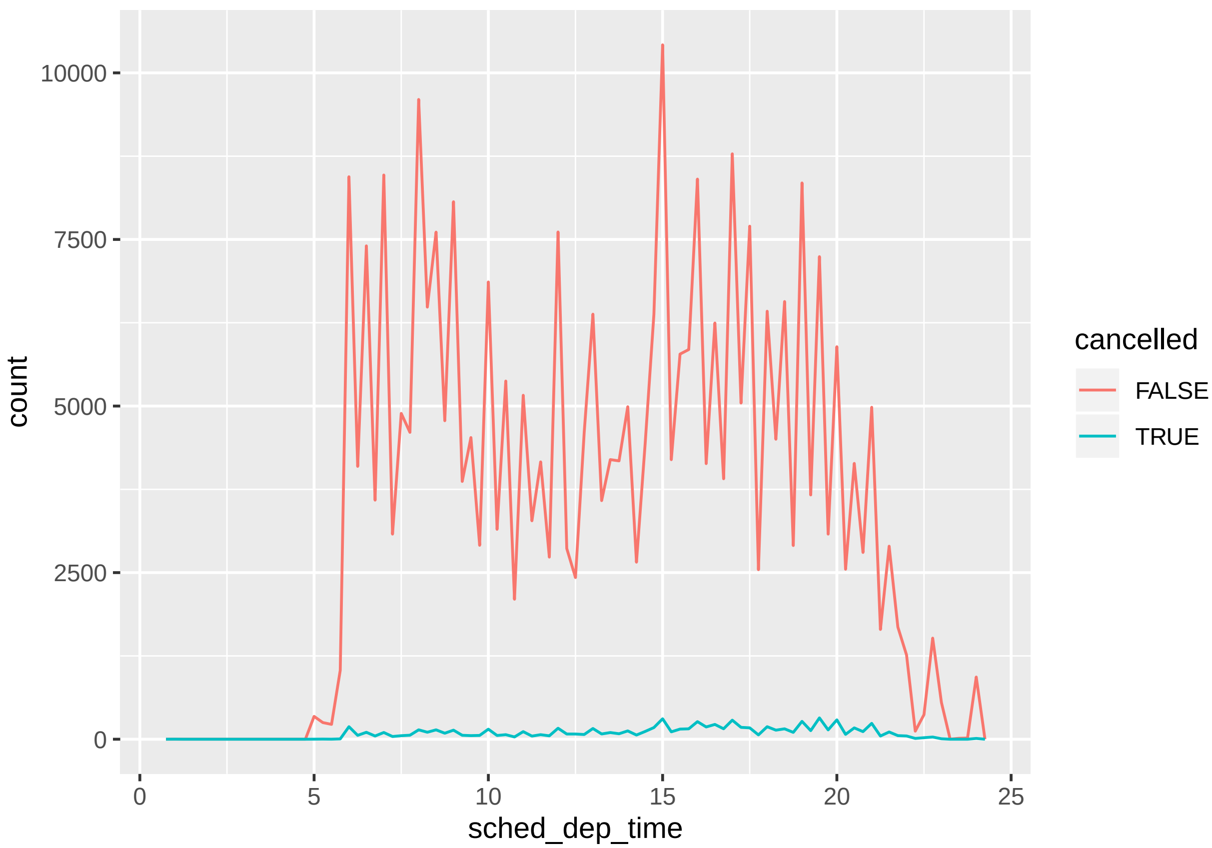
```
## # A tibble: 53,931 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7 0.24  Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8 0.26  Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9 0.22  Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10 0.23  Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,921 more rows
```

2. But the recommended way is to replace the unusual values with missing values. The easiest way to do this is to use `mutate()` to replace the variable with a modified copy. We can use `ifelse()` function to replace unusual values with NA;

```r
diamonds2 <- diamonds %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y))
```

We might want to compare the scheduled departure times for cancelled and non cancelled times. We can do this by making a new variable with `is.na()`

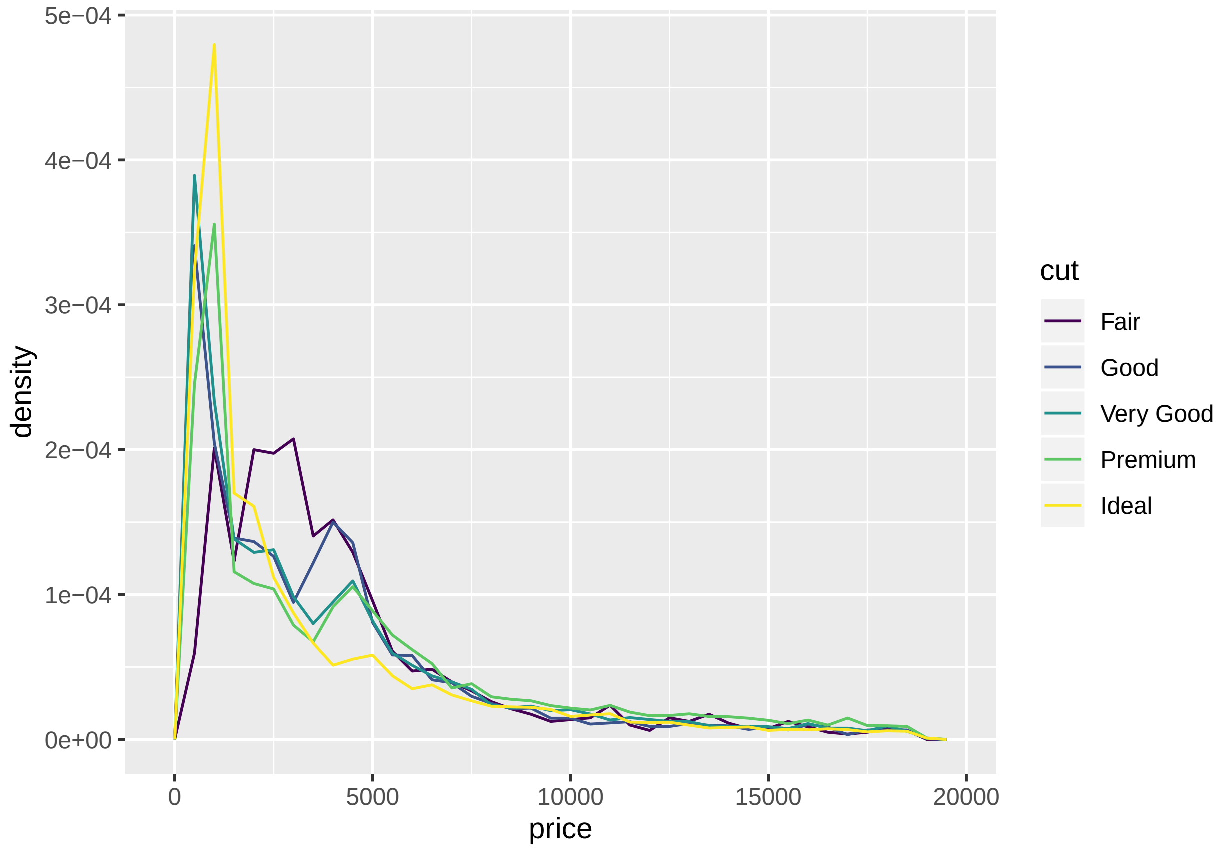
```r
nycflights13::flights %>%
  mutate(
    cancelled = is.na(dep_time),
    sched_hour = sched_dep_time %/% 100,
    sched_min = sched_dep_time %% 100,
    sched_dep_time = sched_hour + sched_min / 60
  )%>%
  ggplot(mapping = aes(sched_dep_time)) + geom_freqpoly(mapping = aes(color = cancelled), binwidth = 1/4
```



Covariation

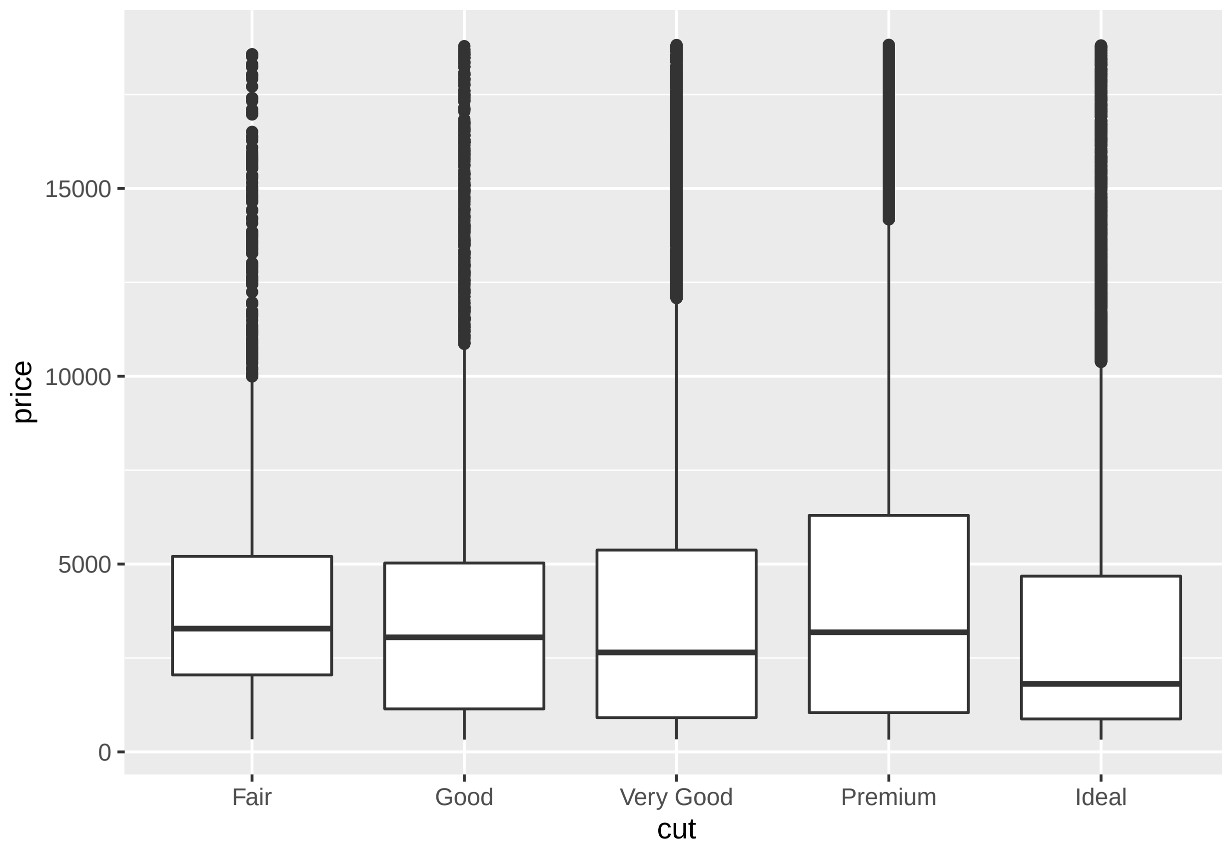
We want to explore how the price of a diamond varies with quality. Instead of displaying count, we will display density

```
ggplot(data = diamonds, mapping = aes(x = price, y = ..density..)) + geom_freqpoly(mapping = aes(color =
```



Distribution of price by cut using `geom_boxplot()`:

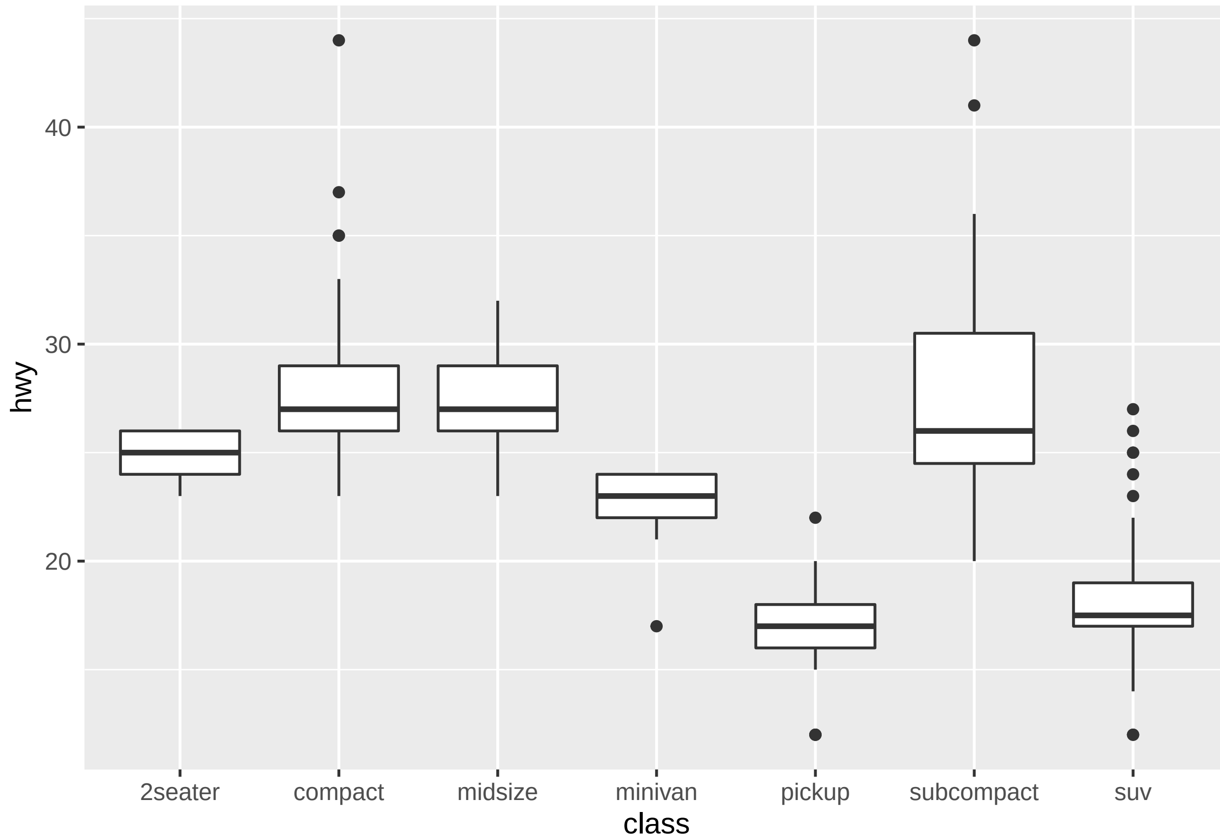
```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) + geom_boxplot()
```



Better quality diamonds are cheaper on average.

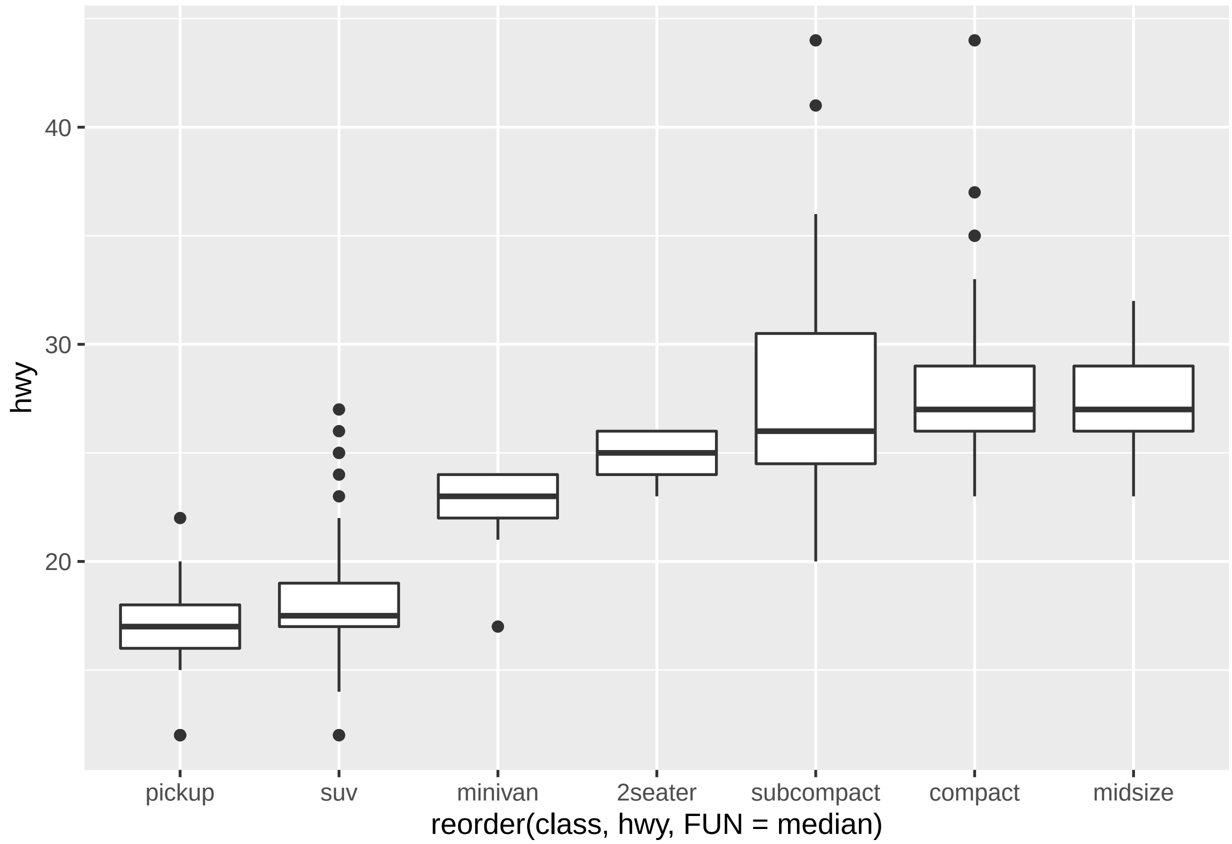
Many categorical variables are not ordered properly like fair < good < very good < premium < ideal. So we need to reorder them to make a more informative display. One way to do this is with `reorder()` function

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) + geom_boxplot()
```



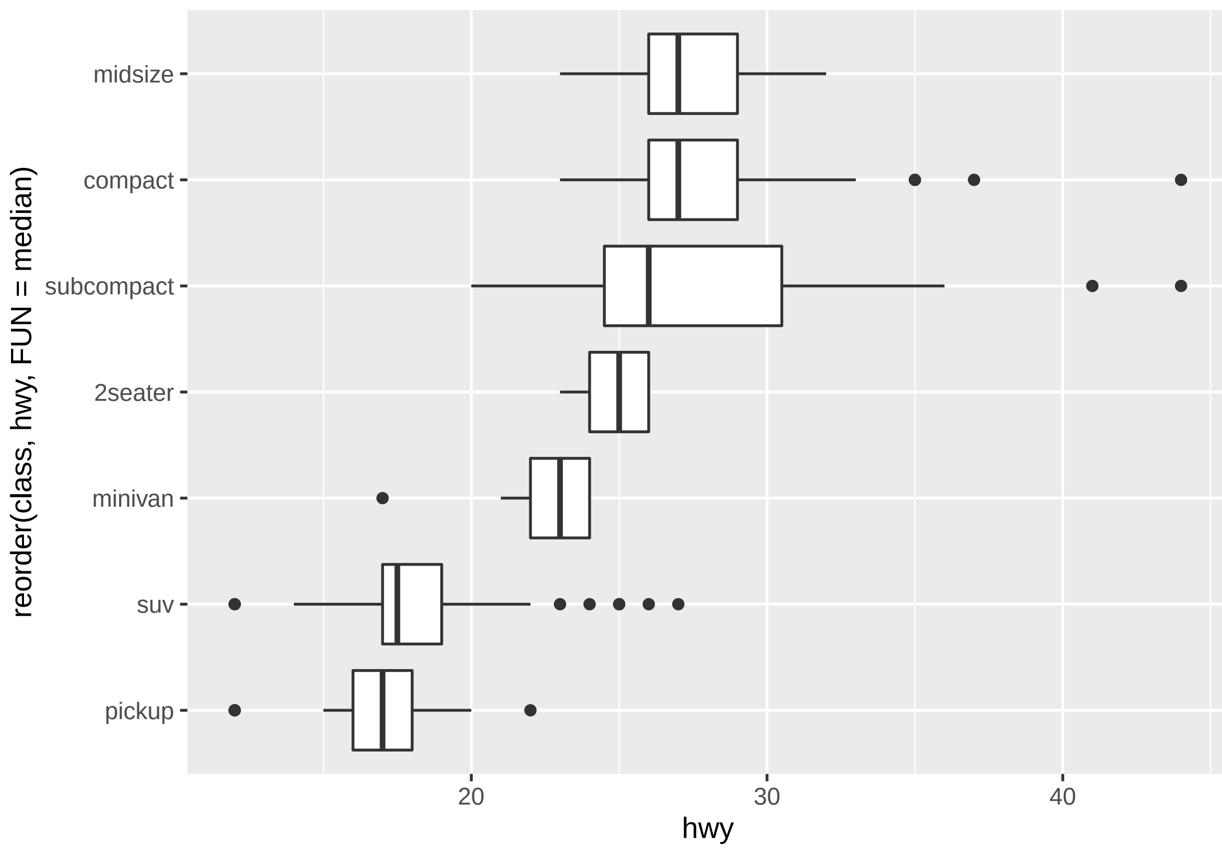
To make the trend easier to see, we can reorder class based on the median value of hwy.

```
ggplot(data = mpg) + geom_boxplot(mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy))
```



For long variable names, we can flip the axes:

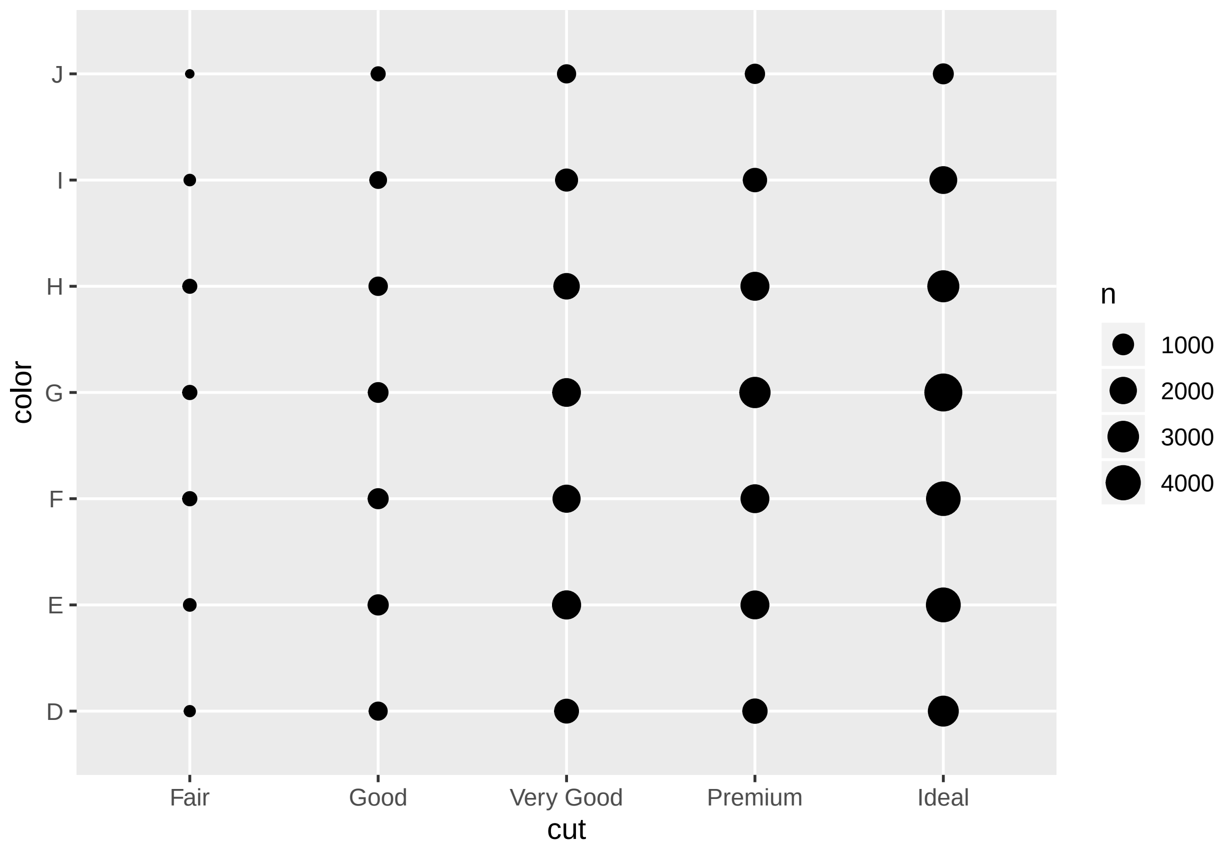
```
ggplot(data = mpg) + geom_boxplot(mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy)) + coord
```



Two categorical variables

To visualise the covariation between categorical variables, you will need to count the number of observations for each combination. One way to do that is to rely on the built in `geom-count()`

```
ggplot(data = diamonds) + geom_count(mapping = aes(x = cut, y = color))
```



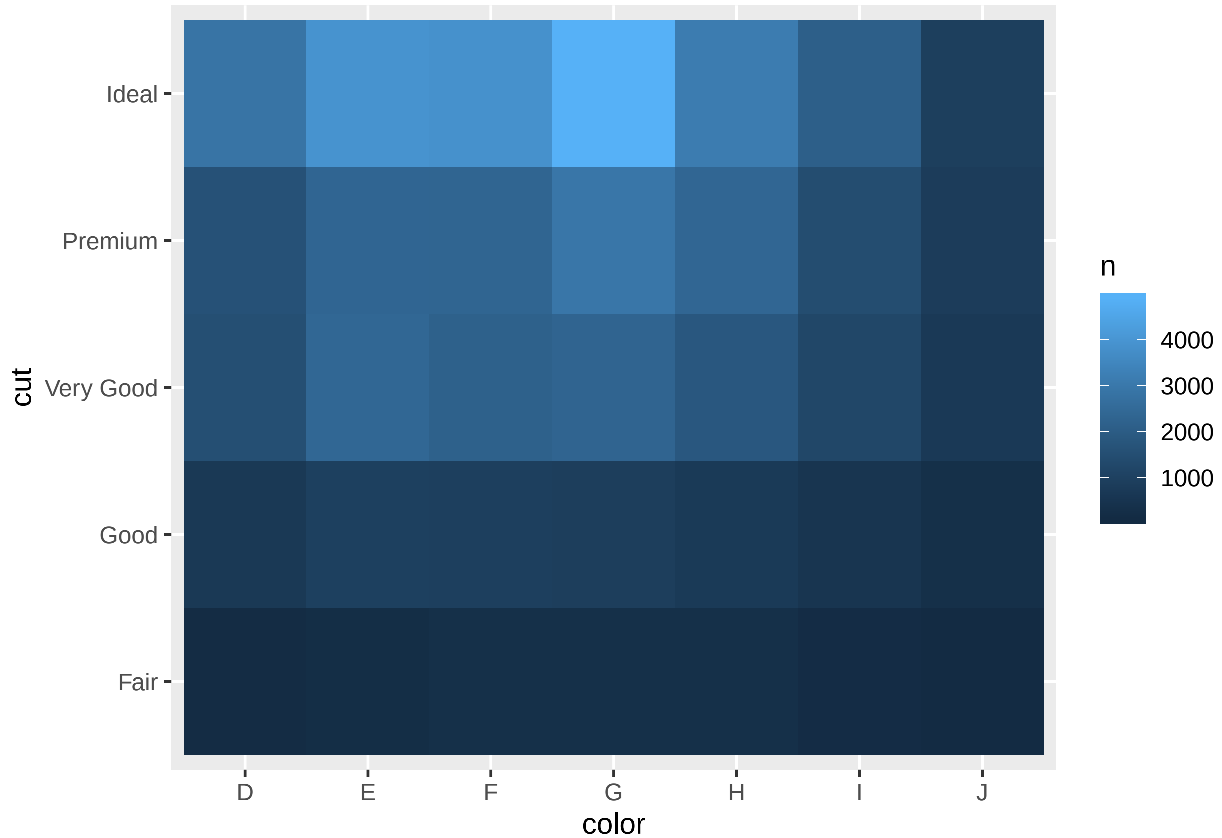
We can also compute the count with dplyr

```
diamonds %>%
  count(color, cut)
```

```
## # A tibble: 35 x 3
##    color cut           n
##    <ord> <ord>     <int>
##  1 D     Fair        163
##  2 D     Good        662
##  3 D     Very Good  1513
##  4 D     Premium    1603
##  5 D     Ideal      2834
##  6 E     Fair        224
##  7 E     Good        933
##  8 E     Very Good  2400
##  9 E     Premium    2337
## 10 E     Ideal      3903
## # ... with 25 more rows
```

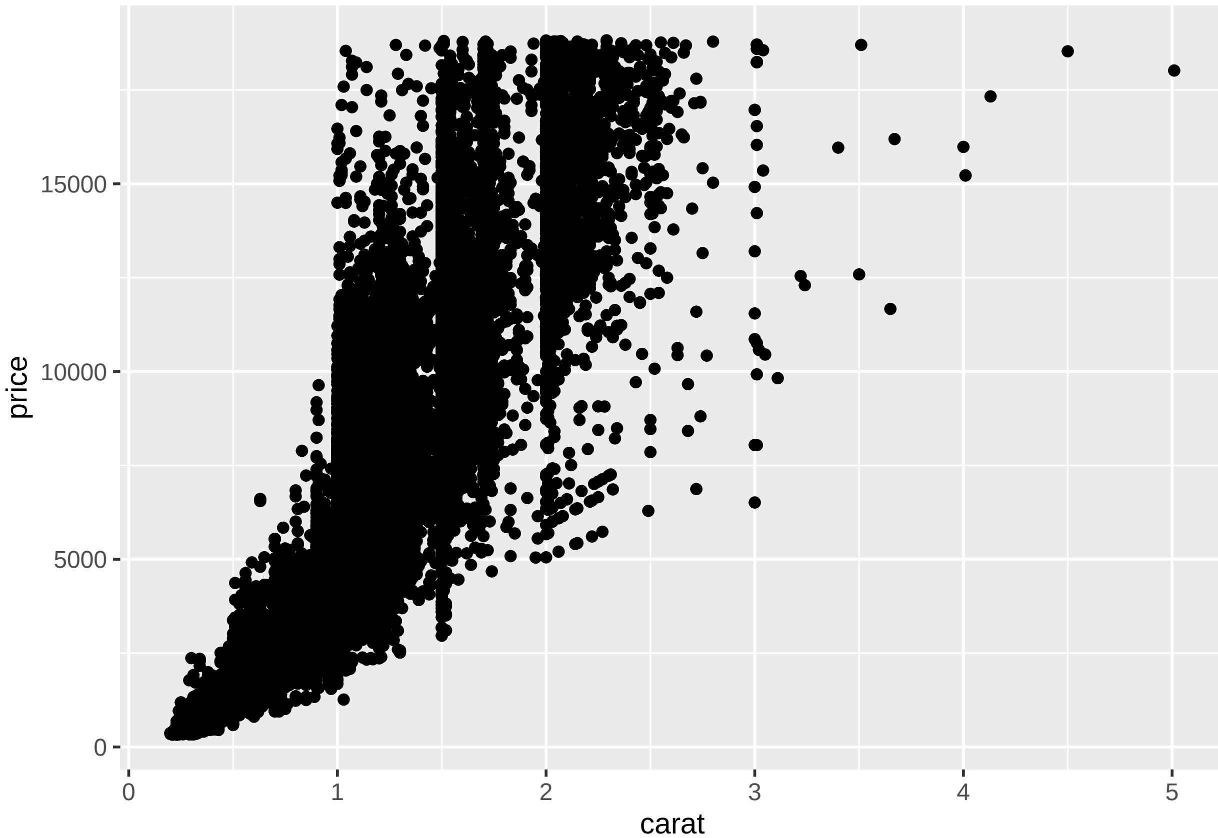
Then visualize with geom_tile() and the fill aesthetic:

```
diamonds %>%
  count(color, cut) %>%
  ggplot(mapping = aes(x = color, y = cut)) + geom_tile(mapping = aes(fill = n))
```



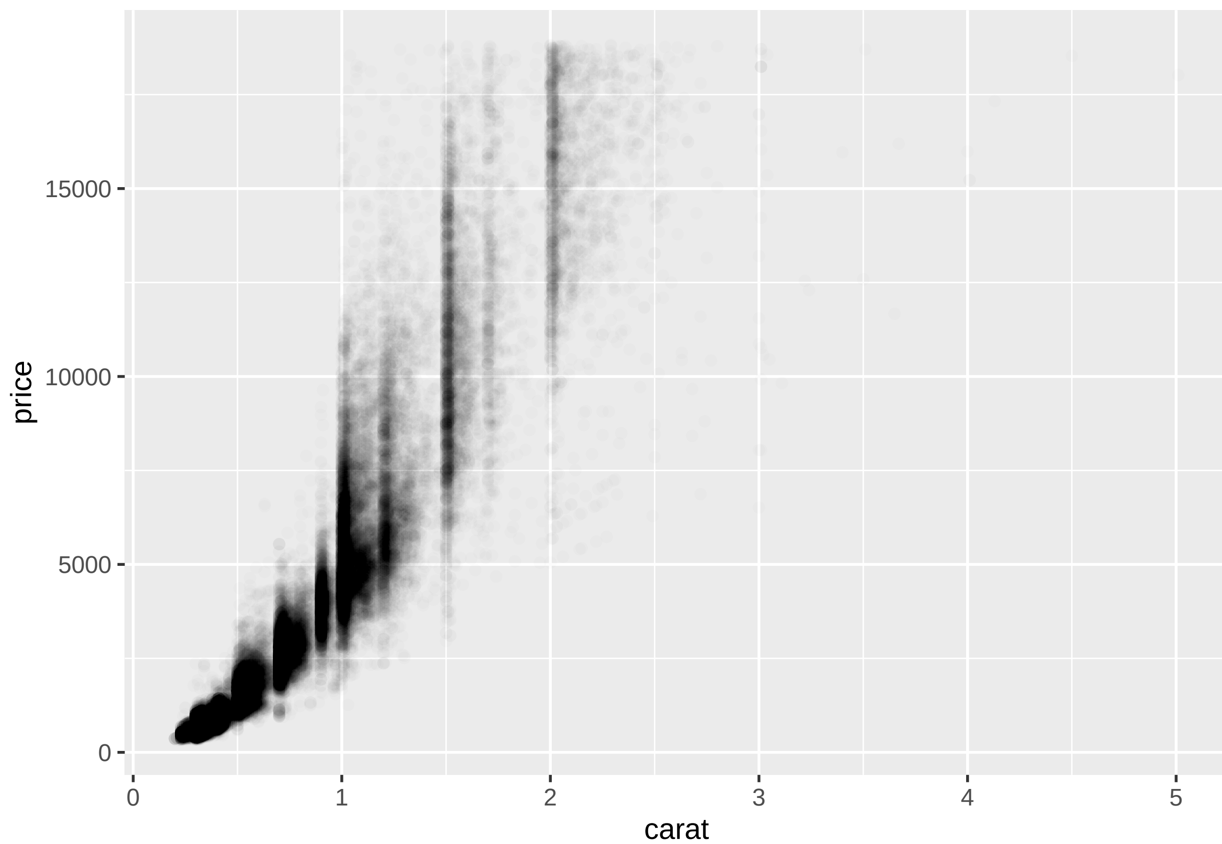
With a scatterplot, we can see an exponential relationship between carat size and price of diamonds

```
ggplot(data = diamonds) + geom_point(mapping = aes(x = carat, y = price))
```



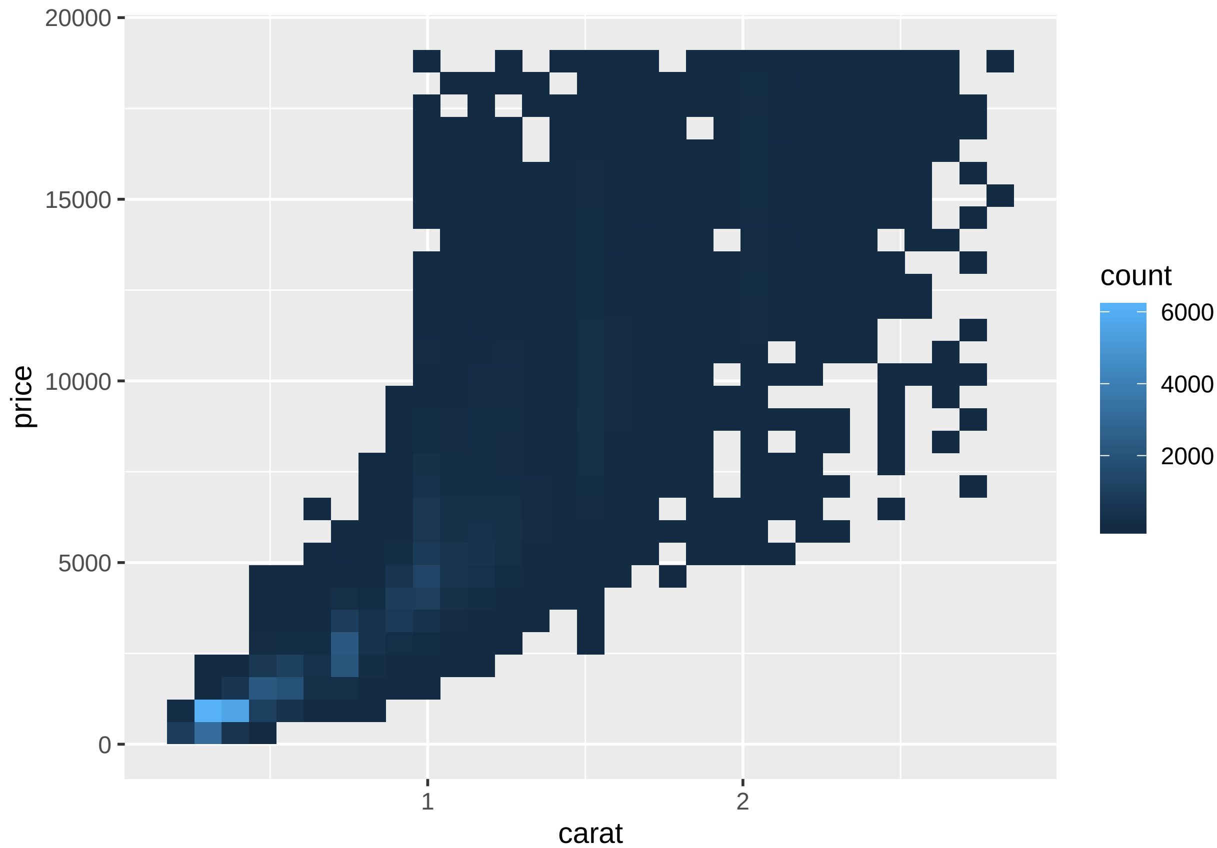
Use the `alpha` aesthetic to add the transparency

```
ggplot(data = diamonds) + geom_point(mapping = aes(x = carat, y = price), alpha = 1/100)
```

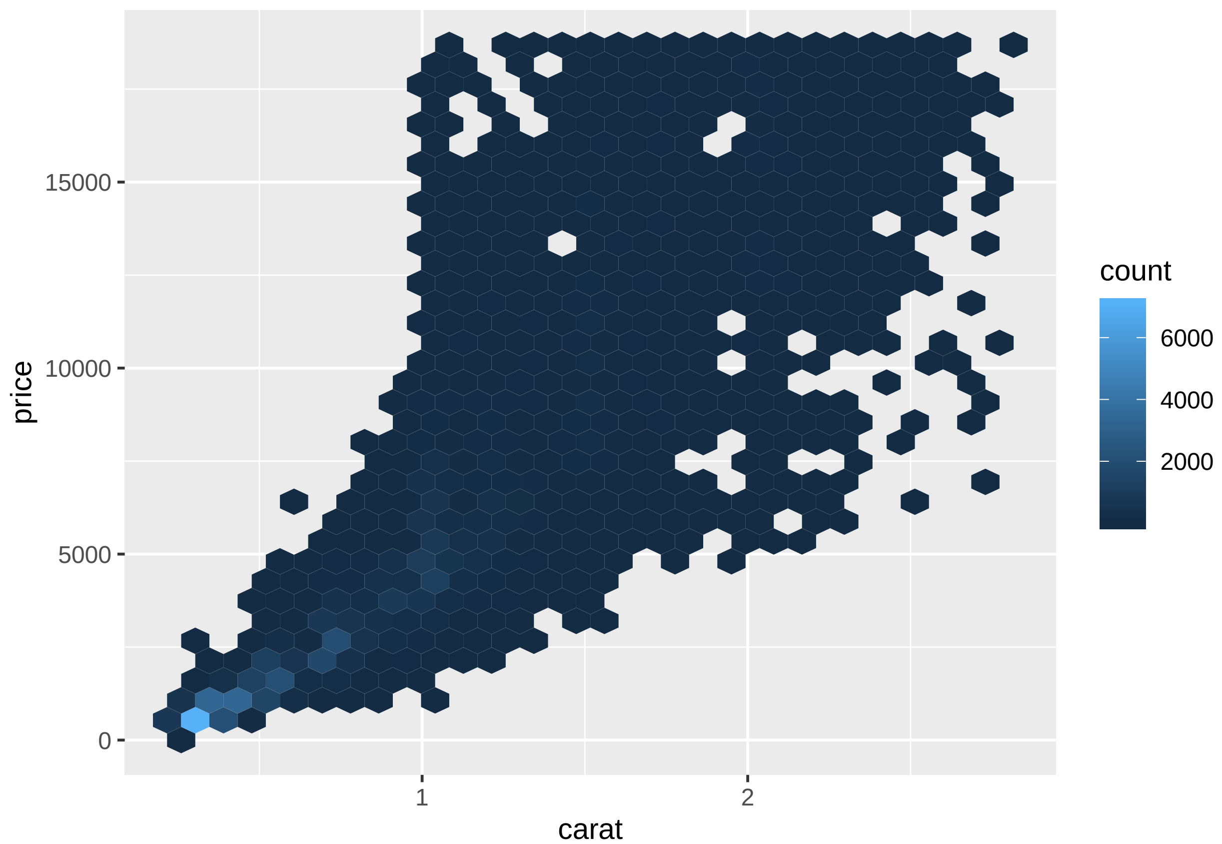


`geom_bin2d()` creates rectangular bins and `geom_hex()` creates hexagonal bins.

```
ggplot(data = smaller) + geom_bin2d(mapping = aes(x = carat, y = price))
```

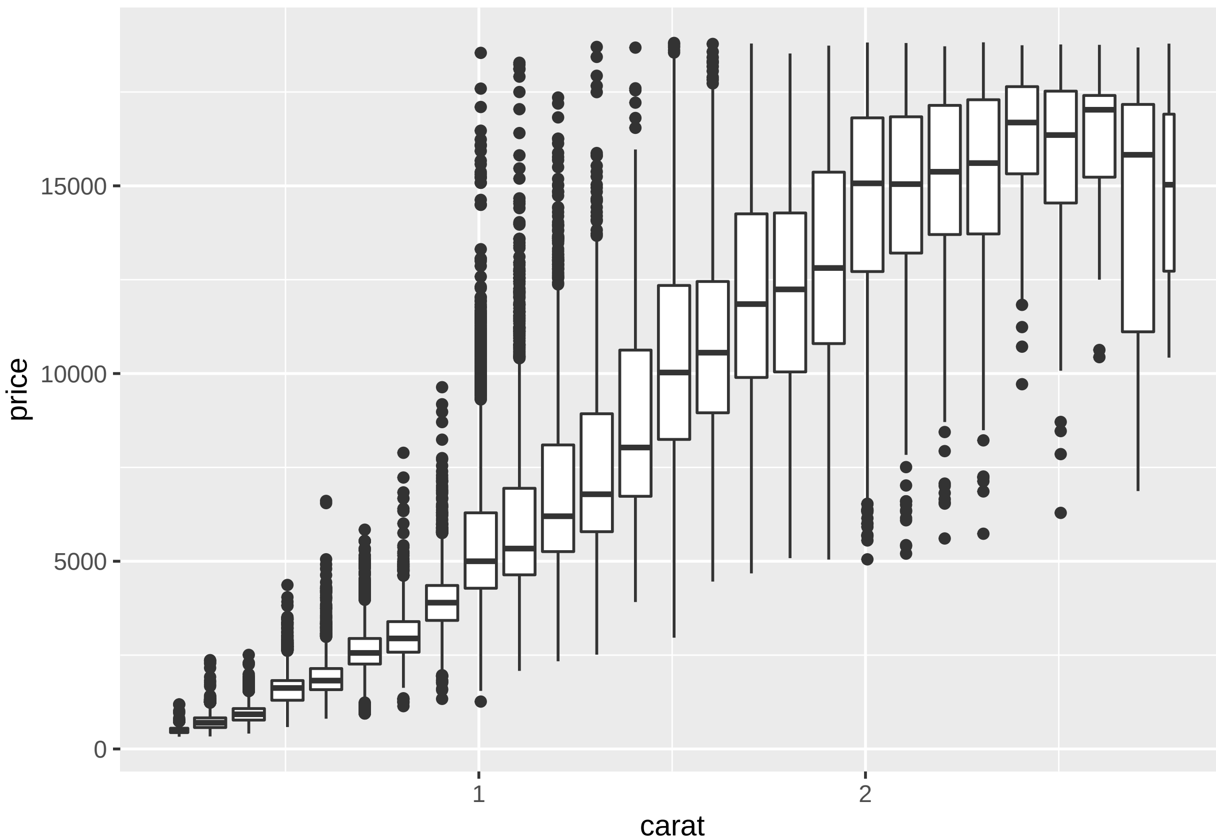



```r
ggplot(data = smaller) + geom_hex(mapping = aes(x = carat, y = price))
```



We could also bin carat and for each group display a boxplot.

```
ggplot(data = smaller, mapping = aes(x = carat, y = price)) + geom_boxplot(mapping = aes(group = cut_wi
```




```
ggplot(data = faithful, mapping = aes(x = eruptions)) + geom_freqpoly(binwdth = 0.25)
```

can be written more concisely as:

```
ggplot(faithful, aes(eruptions)) + geom_freqpoly(binwidth = 0.25)
```

DATA IMPORT

`read_csv()` In this case, `read_csv()` uses the first line of the data for the column names.

```
read_csv("a, b, c
          1, 2, 3
          4, 5, 6")
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Sometimes thera are a few lines of metadata at the top of the file. We can use skip = n to skip the first n lines or use comment = "#" to drop all lines that start with e.g. '#'.

```r
read_csv("The first line of metadata
         The second line of metadata
         x, y, z
         1, 2, 3", skip = 2)
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

```r
read_csv("# A comment I wnat to skip
         x, y, z
         1, 2, 3", comment = "#")
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

If the data does not have column names, we can use `col_names = FALSE` to tell `read_csv()` not to treat the first row as headings, and instead label them sequentially from x1 to xn.

```r
read_csv("1, 2, 3\n4, 5, 6", col_names = FALSE)
```

```
## # A tibble: 2 x 3
##      X1    X2    X3
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

We can also pass col_names a character vector which will be used as the column names

```r
read_csv("1, 2, 3\n4, 5, 6", col_names = c("x", "y", "z"))
```

```
## # A tibble: 2 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

`na` specifies the value or calues that are used to represent the missing values in the file.

```r
read_csv("a, b, c\n1, 2, .", na = ".")
```

```
## # A tibble: 1 x 3
##       a     b c
##   <dbl> <dbl> <lgl>
## 1     1     2 NA
```

`parse_*()` functions take a character vector and return a more specialised vector like a logical, integer or a date.

```r
str(parse_logical(c("TRUE", "FALSE", "NA")))
```

```
##  logi [1:3] TRUE FALSE NA
```

```r
str(parse_integer(c("1", "2", "3")))
```

```
##  int [1:3] 1 2 3
```

```r
str(parse_date(c("2010-01-01", "1979-10-14")))
```

```
##  Date[1:2], format: "2010-01-01" "1979-10-14"
```

The first argument is a character vector to parse and the `na` argument specifies which strings should be treated as missing.

```r
parse_integer(c("1", "231", ".", "456"), na = ".")
```

```
## [1]   1 231  NA 456
```

`parse_number()` ignores non-numeric characters before and after the number. This is particularly useful for currencies and percentages., but also works to extract numbers embedded in text.

```r
parse_number("$100")
```

```
## [1] 100
```

```r
parse_number("20%")
```

```
## [1] 20
```

```r
parse_number("It cost $123.45")
```

```
## [1] 123.45
```

```r
parse_number("$123,456,789")
```

```
## [1] 123456789
```

Factors

R uses factors to represent categorical variables that have a known set of possible values. Give parse_factor() a vector of known levels to generate a warning whenver an unexpected value is present.

```r
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
```

```
## [1] apple  banana <NA>
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected           actual
##   <int> <int> <chr>              <chr>
## 1     3    NA value in level set bananana
## Levels: apple banana
```

Parsing a file

R uses heursistics to figure out the type of each column. We can emulate this process with a character vector using `guess_parser()` which returns the best guess and `parse_guess()` which uses that guess to parse the column.

```
guess_parser("2010-10-01")
```

```
## [1] "date"
```

```
guess_parser("15:01")
```

```
## [1] "time"
```

```
guess_parser(c("TRUE", "FALSE"))
```

```
## [1] "logical"
```

```
guess_parser(c("1", "5", "9"))
```

```
## [1] "double"
```

```
guess_parser(c("12,352,561"))
```

```
## [1] "number"
```

```
str(parse_guess("2010-01-01"))
```

```
##  Date[1:1], format: "2010-01-01"
```

Gathering

used to tidy a dataset where 1999 and 2000 are column names. We want to put them in a single column called "year". `tabel4a %>% gather(`1999,2000`, key = "year", value = "cases")` Here, we need to use backticks for 1999 and 2000 because they are non-syntatic names or they do not start with a letter.

Similarly, we can use `gather()` to tidy table4b in a similar fashion. `table4b %>% gather(`1999,2000‘, key = "year", value = "population")

To combine the tidied versions of table4a and table4b into a single tibble, we need to use `left_join()` `left_join(tidy4a, tidy4b)`

Spreading

It is the opposite of gathering. We can use it when an observation is scattered across multiple rows. `table2 %>% spread(key = type, value = count)`

Separating

If the rate column containds both cases and population variables, we can separate it into two variables. `table3 %>% separate(rate, into = c("cases", "poulation"))` By defualt, separate() will separate values wherever it sees an alphanumeric character. We could also write the code as: `table2 %>% separate(rate, into = c("cases", "poulation"), sep = "/")` separate() leaves the type of column as is. It not very useful in this case however, as those are actuually number. We can ask separate() to try and convert better types using convert = TRUE ‘tabel3 %>% separate(rate, into = c(“cases”, “population”), convert = TRUE)

We can also separate the last two digits of each year. `table3 %>% separate(year, into = c("century", "year"), sep = 2)`

Unite

It is the inverse of separate. It combines multiple columns into a single column. We can use unite() to rejoin century and year columns that we created in the last example. `table5 %>% unite(new, century, year)` The default will place an underscore (_) between the values from different columns. If we do not want any separator, we use sep = ”. ‘table5 %>% unite(new, century, year, sep = "")

MISSING VALUES

They can be missing in one of two possible ways: Explicilty: flagged with NA, or Implicitly: simply not present in the data

```
stocks <- tibble(
  year = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr = c(    1,    2,    3,    4,    2,    3,     4),
  return = c(1.88, 0.59, 0.35,  NA, 0.92, 0.17, 2.66)
)
stocks
```

```
## # A tibble: 7 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     2   0.92
## 6  2016     3   0.17
## 7  2016     4   2.66
```

The return for the fourth quarter of 2015 is explicitly missing, whereas the return for the first quarter of 2016 is implicitly missing because it simply does not appear in the dataset. We can make the implicit missing value explicit by putting years in the columns.

```
stocks %>% spread(year, return)
```

```
## # A tibble: 4 x 3
##     qtr `2015` `2016`
```

```
##   <dbl>  <dbl>  <dbl>
## 1     1   1.88  NA   
## 2     2   0.59   0.92
## 3     3   0.35   0.17
## 4     4  NA      2.66
```

We can set na.rm = TRUE in gather() to turn explicit missing values implicit:

```
stocks %>%
  spread(year, return) %>%
  gather(year, return, `2015`:`2016`, na.rm = TRUE)
```

```
## # A tibble: 6 x 3
##     qtr year  return
##   <dbl> <chr>  <dbl>
## 1     1 2015    1.88
## 2     2 2015    0.59
## 3     3 2015    0.35
## 4     2 2016    0.92
## 5     3 2016    0.17
## 6     4 2016    2.66
```

Another way to make missing values explicit in tidy data is `complete()`.

```
stocks %>% complete(year, qtr)
```

```
## # A tibble: 8 x 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA   
## 5  2016     1  NA   
## 6  2016     2   0.92
## 7  2016     3   0.17
## 8  2016     4   2.66
```

Sometimes when a data source has been primarily used for data entry, missing values indicate that the previous value has been carried forward. We can fill these missing values with `fill()`

Case Study

```
tidyr::who
```

```
## # A tibble: 7,240 x 60
##    country iso2  iso3   year new_sp_m014 new_sp_m1524 new_sp_m2534
##    <chr>   <chr> <chr> <int>       <int>        <int>        <int>
##  1 Afghan~ AF    AFG    1980          NA           NA           NA
```

```
##  2 Afghan~ AF    AFG    1981          NA           NA           NA
##  3 Afghan~ AF    AFG    1982          NA           NA           NA
##  4 Afghan~ AF    AFG    1983          NA           NA           NA
##  5 Afghan~ AF    AFG    1984          NA           NA           NA
##  6 Afghan~ AF    AFG    1985          NA           NA           NA
##  7 Afghan~ AF    AFG    1986          NA           NA           NA
##  8 Afghan~ AF    AFG    1987          NA           NA           NA
##  9 Afghan~ AF    AFG    1988          NA           NA           NA
## 10 Afghan~ AF    AFG    1989          NA           NA           NA
## # ... with 7,230 more rows, and 53 more variables: new_sp_m3544 <int>,
## #   new_sp_m4554 <int>, new_sp_m5564 <int>, new_sp_m65 <int>,
## #   new_sp_f014 <int>, new_sp_f1524 <int>, new_sp_f2534 <int>,
## #   new_sp_f3544 <int>, new_sp_f4554 <int>, new_sp_f5564 <int>,
## #   new_sp_f65 <int>, new_sn_m014 <int>, new_sn_m1524 <int>,
## #   new_sn_m2534 <int>, new_sn_m3544 <int>, new_sn_m4554 <int>,
## #   new_sn_m5564 <int>, new_sn_m65 <int>, new_sn_f014 <int>,
## #   new_sn_f1524 <int>, new_sn_f2534 <int>, new_sn_f3544 <int>,
## #   new_sn_f4554 <int>, new_sn_f5564 <int>, new_sn_f65 <int>,
## #   new_ep_m014 <int>, new_ep_m1524 <int>, new_ep_m2534 <int>,
## #   new_ep_m3544 <int>, new_ep_m4554 <int>, new_ep_m5564 <int>,
## #   new_ep_m65 <int>, new_ep_f014 <int>, new_ep_f1524 <int>,
## #   new_ep_f2534 <int>, new_ep_f3544 <int>, new_ep_f4554 <int>,
## #   new_ep_f5564 <int>, new_ep_f65 <int>, newrel_m014 <int>,
## #   newrel_m1524 <int>, newrel_m2534 <int>, newrel_m3544 <int>,
## #   newrel_m4554 <int>, newrel_m5564 <int>, newrel_m65 <int>,
## #   newrel_f014 <int>, newrel_f1524 <int>, newrel_f2534 <int>,
## #   newrel_f3544 <int>, newrel_f4554 <int>, newrel_f5564 <int>,
## #   newrel_f65 <int>
```

It looks like `country`, `iso2` and `iso3` are three variables that redundantly specify the country. `year` is clearly also a variable. We don't know what all the other columns are yet but given the structure in the variable names, these are likely to be values, not variables. So, we need to gather together all the columns from `new_sp_m014` to `newrel_f65`. We do not know what those values represent yet, so we will give them the generic name "key". We know the cells represent the count of cases so we will use the variable `cases`.

```
who1 <- who %>%
  gather(new_sp_m014:newrel_f65, key = "key", value = "cases", na.rm = TRUE)
who1
```

```
## # A tibble: 76,046 x 6
##    country     iso2  iso3   year key          cases
##    <chr>       <chr> <chr> <int> <chr>        <int>
##  1 Afghanistan AF    AFG    1997 new_sp_m014      0
##  2 Afghanistan AF    AFG    1998 new_sp_m014     30
##  3 Afghanistan AF    AFG    1999 new_sp_m014      8
##  4 Afghanistan AF    AFG    2000 new_sp_m014     52
##  5 Afghanistan AF    AFG    2001 new_sp_m014    129
##  6 Afghanistan AF    AFG    2002 new_sp_m014     90
##  7 Afghanistan AF    AFG    2003 new_sp_m014    127
##  8 Afghanistan AF    AFG    2004 new_sp_m014    139
##  9 Afghanistan AF    AFG    2005 new_sp_m014    151
## 10 Afghanistan AF    AFG    2006 new_sp_m014    193
## # ... with 76,036 more rows
```

We can get some hint of the structure of the values in the new `key` column by counting them.

```
who1 %>%
  count(key)
```

```
## # A tibble: 56 x 2
##    key              n
##    <chr>        <int>
##  1 new_ep_f014   1032
##  2 new_ep_f1524  1021
##  3 new_ep_f2534  1021
##  4 new_ep_f3544  1021
##  5 new_ep_f4554  1017
##  6 new_ep_f5564  1017
##  7 new_ep_f65    1014
##  8 new_ep_m014   1038
##  9 new_ep_m1524  1026
## 10 new_ep_m2534  1020
## # ... with 46 more rows
```

We are going to replace the characters "newrel" with "new_rel" to make the variable names consistent.

```
who2 <- who1 %>%
  mutate(key = stringr::str_replace(key, "newrel", "new_rel"))
who2
```

```
## # A tibble: 76,046 x 6
##    country     iso2  iso3   year key          cases
##    <chr>       <chr> <chr> <int> <chr>        <int>
##  1 Afghanistan AF    AFG    1997 new_sp_m014      0
##  2 Afghanistan AF    AFG    1998 new_sp_m014     30
##  3 Afghanistan AF    AFG    1999 new_sp_m014      8
##  4 Afghanistan AF    AFG    2000 new_sp_m014     52
##  5 Afghanistan AF    AFG    2001 new_sp_m014    129
##  6 Afghanistan AF    AFG    2002 new_sp_m014     90
##  7 Afghanistan AF    AFG    2003 new_sp_m014    127
##  8 Afghanistan AF    AFG    2004 new_sp_m014    139
##  9 Afghanistan AF    AFG    2005 new_sp_m014    151
## 10 Afghanistan AF    AFG    2006 new_sp_m014    193
## # ... with 76,036 more rows
```

We can separate the values in each code with two passes of `separate()`. The first pass will split the codes at each underscore.

```
who3 <- who2 %>%
  separate(key, c("new", "type", "sexage"), sep = "_")
who3
```

```
## # A tibble: 76,046 x 8
##    country     iso2  iso3   year new   type  sexage cases
##    <chr>       <chr> <chr> <int> <chr> <chr> <chr>  <int>
##  1 Afghanistan AF    AFG    1997 new   sp    m014       0
```

```
##  2 Afghanistan AF    AFG    1998 new   sp    m014      30
##  3 Afghanistan AF    AFG    1999 new   sp    m014       8
##  4 Afghanistan AF    AFG    2000 new   sp    m014      52
##  5 Afghanistan AF    AFG    2001 new   sp    m014     129
##  6 Afghanistan AF    AFG    2002 new   sp    m014      90
##  7 Afghanistan AF    AFG    2003 new   sp    m014     127
##  8 Afghanistan AF    AFG    2004 new   sp    m014     139
##  9 Afghanistan AF    AFG    2005 new   sp    m014     151
## 10 Afghanistan AF    AFG    2006 new   sp    m014     193
## # ... with 76,036 more rows
```

We might as well drop the `new` column because it is constant in the dataset. While we are dropping columns, we can also drop iso2 and iso3 since they are redundant.

```
who3 %>% count(new)
```

```
## # A tibble: 1 x 2
##   new       n
##   <chr> <int>
## 1 new   76046
```

```
who4 <- who3 %>% select(-new, -iso2, -iso3)
who4
```

```
## # A tibble: 76,046 x 5
##    country      year type  sexage cases
##    <chr>       <int> <chr> <chr>  <int>
##  1 Afghanistan  1997 sp    m014       0
##  2 Afghanistan  1998 sp    m014      30
##  3 Afghanistan  1999 sp    m014       8
##  4 Afghanistan  2000 sp    m014      52
##  5 Afghanistan  2001 sp    m014     129
##  6 Afghanistan  2002 sp    m014      90
##  7 Afghanistan  2003 sp    m014     127
##  8 Afghanistan  2004 sp    m014     139
##  9 Afghanistan  2005 sp    m014     151
## 10 Afghanistan  2006 sp    m014     193
## # ... with 76,036 more rows
```

Next we can separate sexage into sex and age by splitting after the first character.

```
who5 <- who4 %>%
  separate(sexage, c("sex", "age"), sep = 1)
who5
```

```
## # A tibble: 76,046 x 6
##    country      year type  sex   age   cases
##    <chr>       <int> <chr> <chr> <chr> <int>
##  1 Afghanistan  1997 sp    m     014       0
##  2 Afghanistan  1998 sp    m     014      30
##  3 Afghanistan  1999 sp    m     014       8
##  4 Afghanistan  2000 sp    m     014      52
```

```
##  5 Afghanistan  2001 sp    m     014     129
##  6 Afghanistan  2002 sp    m     014      90
##  7 Afghanistan  2003 sp    m     014     127
##  8 Afghanistan  2004 sp    m     014     139
##  9 Afghanistan  2005 sp    m     014     151
## 10 Afghanistan  2006 sp    m     014     193
## # ... with 76,036 more rows
```

Relational Data

```
library(tidyverse)
library(nycflights13)
airlines
```

```
## # A tibble: 16 x 2
##    carrier name
##    <chr>   <chr>
##  1 9E      Endeavor Air Inc.
##  2 AA      American Airlines Inc.
##  3 AS      Alaska Airlines Inc.
##  4 B6      JetBlue Airways
##  5 DL      Delta Air Lines Inc.
##  6 EV      ExpressJet Airlines Inc.
##  7 F9      Frontier Airlines Inc.
##  8 FL      AirTran Airways Corporation
##  9 HA      Hawaiian Airlines Inc.
## 10 MQ      Envoy Air
## 11 OO      SkyWest Airlines Inc.
## 12 UA      United Air Lines Inc.
## 13 US      US Airways Inc.
## 14 VX      Virgin America
## 15 WN      Southwest Airlines Co.
## 16 YV      Mesa Airlines Inc.
```

```
airports
```

```
## # A tibble: 1,458 x 8
##    faa   name                   lat    lon   alt    tz dst   tzone
##    <chr> <chr>                <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
##  1 04G   Lansdowne Airport     41.1  -80.6  1044    -5 A     America/New_~
##  2 06A   Moton Field Municipa~ 32.5  -85.7   264    -6 A     America/Chic~
##  3 06C   Schaumburg Regional   42.0  -88.1   801    -6 A     America/Chic~
##  4 06N   Randall Airport       41.4  -74.4   523    -5 A     America/New_~
##  5 09J   Jekyll Island Airport 31.1  -81.4    11    -5 A     America/New_~
##  6 0A9   Elizabethton Municip~ 36.4  -82.2  1593    -5 A     America/New_~
##  7 0G6   Williams County Airp~ 41.5  -84.5   730    -5 A     America/New_~
##  8 0G7   Finger Lakes Regiona~ 42.9  -76.8   492    -5 A     America/New_~
##  9 0P2   Shoestring Aviation ~ 39.8  -76.6  1000    -5 U     America/New_~
## 10 0S9   Jefferson County Intl 48.1 -123.    108    -8 A     America/Los_~
## # ... with 1,448 more rows
```

```r
planes
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type       manufacturer  model   engines seats speed engine
##    <chr>   <int> <chr>      <chr>         <chr>     <int> <int> <int> <chr>
##  1 N10156   2004 Fixed win~ EMBRAER       EMB-1~        2    55    NA Turbo~
##  2 N102UW   1998 Fixed win~ AIRBUS INDUS~ A320-~~       2   182    NA Turbo~
##  3 N103US   1999 Fixed win~ AIRBUS INDUS~ A320-~~       2   182    NA Turbo~
##  4 N104UW   1999 Fixed win~ AIRBUS INDUS~ A320-~~       2   182    NA Turbo~
##  5 N10575   2002 Fixed win~ EMBRAER       EMB-1~        2    55    NA Turbo~
##  6 N105UW   1999 Fixed win~ AIRBUS INDUS~ A320-~~       2   182    NA Turbo~
##  7 N107US   1999 Fixed win~ AIRBUS INDUS~ A320-~~       2   182    NA Turbo~
##  8 N108UW   1999 Fixed win~ AIRBUS INDUS~ A320-~~       2   182    NA Turbo~
##  9 N109UW   1999 Fixed win~ AIRBUS INDUS~ A320-~~       2   182    NA Turbo~
## 10 N110UW   1999 Fixed win~ AIRBUS INDUS~ A320-~~       2   182    NA Turbo~
## # ... with 3,312 more rows
```

```r
weather
```

```
## # A tibble: 26,115 x 15
##    origin  year month   day  hour  temp  dewp humid wind_dir wind_speed
##    <chr>  <int> <int> <int> <int> <dbl> <dbl> <dbl>    <dbl>      <dbl>
##  1 EWR     2013     1     1     1  39.0  26.1  59.4      270      10.4
##  2 EWR     2013     1     1     2  39.0  27.0  61.6      250       8.06
##  3 EWR     2013     1     1     3  39.0  28.0  64.4      240      11.5
##  4 EWR     2013     1     1     4  39.9  28.0  62.2      250      12.7
##  5 EWR     2013     1     1     5  39.0  28.0  64.4      260      12.7
##  6 EWR     2013     1     1     6  37.9  28.0  67.2      240      11.5
##  7 EWR     2013     1     1     7  39.0  28.0  64.4      240      15.0
##  8 EWR     2013     1     1     8  39.9  28.0  62.2      250      10.4
##  9 EWR     2013     1     1     9  39.9  28.0  62.2      260      15.0
## 10 EWR     2013     1     1    10  41    28.0  59.6      260      13.8
## # ... with 26,105 more rows, and 5 more variables: wind_gust <dbl>,
## #   precip <dbl>, pressure <dbl>, visib <dbl>, time_hour <dttm>
```

To identify the primary keys in the tables, we can use `count()` and look for entries where n is greater than one:

```r
planes %>%
  count(tailnum) %>%
  filter(n>1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: tailnum <chr>, n <int>
```

```r
weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n>1)
```

```
## # A tibble: 3 x 6
##    year month   day  hour origin     n
```

```
##    <int> <int> <int> <int> <chr>  <int>
## 1  2013    11     3     1 EWR        2
## 2  2013    11     3     1 JFK        2
## 3  2013    11     3     1 LGA        2
```

Mutating joins

`mutate()` allows us to combine variables from two tables.

```
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
flights2
```

```
## # A tibble: 336,776 x 8
##     year month   day  hour origin dest  tailnum carrier
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
##  1  2013     1     1     5 EWR    IAH   N14228  UA
##  2  2013     1     1     5 LGA    IAH   N24211  UA
##  3  2013     1     1     5 JFK    MIA   N619AA  AA
##  4  2013     1     1     5 JFK    BQN   N804JB  B6
##  5  2013     1     1     6 LGA    ATL   N668DN  DL
##  6  2013     1     1     5 EWR    ORD   N39463  UA
##  7  2013     1     1     6 EWR    FLL   N516JB  B6
##  8  2013     1     1     6 LGA    IAD   N829AS  EV
##  9  2013     1     1     6 JFK    MCO   N593JB  B6
## 10  2013     1     1     6 LGA    ORD   N3ALAA  AA
## # ... with 336,766 more rows
```

If we want to add full airline name to flights2 data, we can combine `airlines` and `flights2` dataframe with left_join():

```r
flights2 %>%
  select(-origin, -dest) %>%
  left_join(airlines, by = "carrier")
```

```
## # A tibble: 336,776 x 7
##     year month   day  hour tailnum carrier name
##    <int> <int> <int> <dbl> <chr>   <chr>   <chr>
##  1  2013     1     1     5 N14228  UA      United Air Lines Inc.
##  2  2013     1     1     5 N24211  UA      United Air Lines Inc.
##  3  2013     1     1     5 N619AA  AA      American Airlines Inc.
##  4  2013     1     1     5 N804JB  B6      JetBlue Airways
##  5  2013     1     1     6 N668DN  DL      Delta Air Lines Inc.
##  6  2013     1     1     5 N39463  UA      United Air Lines Inc.
##  7  2013     1     1     6 N516JB  B6      JetBlue Airways
##  8  2013     1     1     6 N829AS  EV      ExpressJet Airlines Inc.
##  9  2013     1     1     6 N593JB  B6      JetBlue Airways
## 10  2013     1     1     6 N3ALAA  AA      American Airlines Inc.
## # ... with 336,766 more rows
```

We could have also obtained the same result with `mutate`

```r
flights2 %>%
  select(-origin, -dest) %>%
  mutate(name = airlines$name[match(carrier, airlines$carrier)])
```

```
## # A tibble: 336,776 x 7
##     year month   day  hour tailnum carrier name
##    <int> <int> <int> <dbl> <chr>   <chr>   <chr>
##  1  2013     1     1     5 N14228  UA      United Air Lines Inc.
##  2  2013     1     1     5 N24211  UA      United Air Lines Inc.
##  3  2013     1     1     5 N619AA  AA      American Airlines Inc.
##  4  2013     1     1     5 N804JB  B6      JetBlue Airways
##  5  2013     1     1     6 N668DN  DL      Delta Air Lines Inc.
##  6  2013     1     1     5 N39463  UA      United Air Lines Inc.
##  7  2013     1     1     6 N516JB  B6      JetBlue Airways
##  8  2013     1     1     6 N829AS  EV      ExpressJet Airlines Inc.
##  9  2013     1     1     6 N593JB  B6      JetBlue Airways
## 10  2013     1     1     6 N3ALAA  AA      American Airlines Inc.
## # ... with 336,766 more rows
```

Left join is the most widely used type of join: it preserves all entries in the left table.

The default `by = NULL` uses all variables that appear in both tables, the so called natural join. The flights and weather tables match on their common variables: year, month, day, hour and origin.

```r
flights2 %>%
  left_join(weather)
```

```
## # A tibble: 336,776 x 18
##     year month   day  hour origin dest  tailnum carrier  temp  dewp humid
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <dbl> <dbl> <dbl>
##  1  2013     1     1     5 EWR    IAH   N14228  UA       39.0  28.0  64.4
##  2  2013     1     1     5 LGA    IAH   N24211  UA       39.9  25.0  54.8
##  3  2013     1     1     5 JFK    MIA   N619AA  AA       39.0  27.0  61.6
##  4  2013     1     1     5 JFK    BQN   N804JB  B6       39.0  27.0  61.6
##  5  2013     1     1     6 LGA    ATL   N668DN  DL       39.9  25.0  54.8
##  6  2013     1     1     5 EWR    ORD   N39463  UA       39.0  28.0  64.4
##  7  2013     1     1     6 EWR    FLL   N516JB  B6       37.9  28.0  67.2
##  8  2013     1     1     6 LGA    IAD   N829AS  EV       39.9  25.0  54.8
##  9  2013     1     1     6 JFK    MCO   N593JB  B6       37.9  27.0  64.3
## 10  2013     1     1     6 LGA    ORD   N3ALAA  AA       39.9  25.0  54.8
## # ... with 336,766 more rows, and 7 more variables: wind_dir <dbl>,
## #   wind_speed <dbl>, wind_gust <dbl>, precip <dbl>, pressure <dbl>,
## #   visib <dbl>, time_hour <dttm>
```

flights and planes have year variables but they mean different things, so we only want to join by tailnum.

```
flights2 %>%
  left_join(planes, by="tailnum")
```

```
## # A tibble: 336,776 x 16
##    year.x month   day  hour origin dest  tailnum carrier year.y type
##     <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>    <int> <chr>
##  1   2013     1     1     5 EWR    IAH   N14228  UA        1999 Fixe~
##  2   2013     1     1     5 LGA    IAH   N24211  UA        1998 Fixe~
##  3   2013     1     1     5 JFK    MIA   N619AA  AA        1990 Fixe~
##  4   2013     1     1     5 JFK    BQN   N804JB  B6        2012 Fixe~
##  5   2013     1     1     6 LGA    ATL   N668DN  DL        1991 Fixe~
##  6   2013     1     1     5 EWR    ORD   N39463  UA        2012 Fixe~
##  7   2013     1     1     6 EWR    FLL   N516JB  B6        2000 Fixe~
##  8   2013     1     1     6 LGA    IAD   N829AS  EV        1998 Fixe~
##  9   2013     1     1     6 JFK    MCO   N593JB  B6        2004 Fixe~
## 10   2013     1     1     6 LGA    ORD   N3ALAA  AA          NA <NA>
## # ... with 336,766 more rows, and 6 more variables: manufacturer <chr>,
## #   model <chr>, engines <int>, seats <int>, speed <int>, engine <chr>
```

base::merge() can perform all types of mutating joins: inner_join(x, y) —> merge(x, y) left_join(x, y) —> merge(x, y, all.x = TRUE) right_join(x, y) —> merge(x, y, all.y = TRUE) full_joi(x, y)—> merge(x, y, all.x = TRUE, all.y = TRUE)

Strings

Use writeLines() to print a string

```
library(tidyverse)
library(stringr)
string1 <- "This is a string"
string2 <- 'If I want to include a "quote" inside a string, I use single quotes'
writeLines(string1)
```

```
## This is a string
```

```
writeLines(string2)
```

```
## If I want to include a "quote" inside a string, I use single quotes
```

Multiple strings are often stored in a character vector which we can create with `c()`:

```
c("one", "two", "three")
```

```
## [1] "one"   "two"   "three"
```

The Base R functions can be inconsistent, hence we should use the functions from `stringr` `str_length()` tells us the number of characters in a string.

```
str_length(c("a", "R for data science", NA))
```

```
## [1]  1 18 NA
```

To combine two strings, we can use `str_c()`:

```
str_c("x", "y")
```

```
## [1] "xy"
```

```
str_c("x", "y", "z")
```

```
## [1] "xyz"
```

Use `sep` argument to control how they are separated:

```
str_c("x", "y", sep = ", ")
```

```
## [1] "x, y"
```

Subsetting strings()

As well as the string, `str_sub()` takes the start amd emd arguments which give the inclusive position of the substring.

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
```

```
## [1] "App" "Ban" "Pea"
```

```r
str_sub(x, -3, -1)
```

```
## [1] "ple" "ana" "ear"
```

Regular Expressions

Detect matches:

```r
x <- c("apple", "banana", "pear")
str_detect(x, "e")
```

```
## [1]  TRUE FALSE  TRUE
```