Lecture 5

Marc Kaufmann

10/9/2019

Wrapping up Chapter 5 of R4DS

5.6.4 Summary Functions

You should be aware of the following functions that you can use with `summarise()` should you need them:

```
x <- 1:100
median(x)
```

```
## [1] 50.5
```

```
sd(x)
```

```
## [1] 29.01149
```

```
IQR(x)
```

```
## [1] 49.5
```

```
mad(x)
```

```
## [1] 37.065
```

```
?mad
```

```
## starting httpd help server ... done
```

```
min(x)
```

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

```
max(x)
```

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

An important difference between the mean and the median is that the median is more robust to outliers. Often when people say 'the average person', they have in mind the 'median person', not the average person.

```
# Example
x <- 1:100
mean(x)
```

```
## [1] 50.5
```

```
median(x)
```

```
## [1] 50.5
```

```
x_with_outlier <- c(x, 1000000000)
mean(x_with_outlier)
```

```
## [1] 9901040
```

```
median(x_with_outlier)
```

```
## [1] 51
```

Sometimes you want the mean, not the median, but you have to be aware of what it tells you.

Quantile

An important one is `quantile()`:

```
# What does quantile do?
?quantile

# Not that helpful. Here's what I do when I am not sure
x <- 1:100
quantile(x, 0.25)
```

```
##   25%
## 25.75
```

```
quantile(x, 0.20)
```

```
##  20%
## 20.8
```

```
quantile(x, 0.50)
```

```
##  50%
## 50.5
```

```
y <- 1:5
quantile(y, 0.50)
```

```
## 50%
##   3
```

```
z <- c(0,0,0,0,0,1,2,100,100)
quantile(z, 0.90)
```

```
## 90%
## 100
```

```
quantile(z, 0.50)
```

```
## 50%
##   0
```

```
quantile(z, 0.40)
```

```
## 40%
##   0
```

Counts

You'll often want to count things.

```
# Count the number of flights to each destination
library(nycflights13)
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------------------------------
## v ggplot2 3.2.1     v purrr   0.3.2
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   0.8.3     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
## -- Conflicts ------------------------------------------------------------------------------------
```

```
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
not_missing <- flights %>%
  filter(!is.na(arr_time), !is.na(dep_time)) %>%
  filter(!is.na(arr_delay), !is.na(dep_delay))

not_missing %>%
  group_by(dest) %>%
  summarise(
    count = n()
  )
```

```
## # A tibble: 104 x 2
##    dest  count
##    <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
```

```
# Count the number of distinct carriers to each location
not_missing %>%
  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
```

Since they are so important and common, there is a shorthand for `group_by(...) %>% summarise(count = n())` called `count(...)`:

```
# Short hand
not_missing %>%
  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
```

This is good enough for simple counts, but you may want to weight the counting, or get sums, or get averages:

```
# This counts how many airmiles a given airplane did from NYC
not_missing %>%
  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
```

```
not_missing %>%
  count(tailnum, wt = distance) %>%
  arrange(desc(n))
```

```
## # A tibble: 4,037 x 2
##    tailnum      n
##    <chr>    <dbl>
##  1 N328AA  929090
##  2 N338AA  921172
##  3 N335AA  902271
##  4 N327AA  900482
##  5 N323AA  839468
##  6 N319AA  837924
##  7 N336AA  833136
##  8 N329AA  825826
##  9 N324AA  786159
## 10 N339AA  783648
## # ... with 4,027 more rows
```

```
## Number of flights each day before 5am
not_missing %>%
  group_by(year, month, day) %>%
  summarise(sum(dep_time < 500))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day `sum(dep_time < 500)`
##    <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
```

```
# How many flights are delayed each day by more than 1 hour?
not_missing %>%
  group_by(year, month, day) %>%
  summarise(one_hour_fq = mean(arr_delay > 60))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day one_hour_fq
##    <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
```

```
# Class Exercise: Why do I use the mean above? How does that get the proportion?
```

Ungrouping

If you want to get rid of earlier groupings, use `ungroup()`:

```
daily <- flights %>%
  group_by(year, month, day)

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

```
## # A tibble: 1 x 1
```

```
##    `n()`
##    <int>
## 1 336776
```

Question: Given the answer, what is the default grouping and how many groups are there?

5.7 Grouped Mutates (and filters)

We can use grouping to

```
# Get the worst 10 arrivers for every day
flights_small <- flights %>%
  select(year:day, starts_with("arr"), starts_with("dep"))

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

```
## # A tibble: 3,306 x 8
## # Groups:   year, month, day [365]
##     year month   day arr_time arr_delay dep_time dep_delay delay_rank
##    <int> <int> <int>    <int>     <dbl>    <int>     <dbl>      <dbl>
##  1  2013     1     1     1001       851      848       853          1
##  2  2013     1     1     2120       338     1815       290          3
##  3  2013     1     1     1958       263     1842       260          4
##  4  2013     1     1     2124       174     1942       157          9
##  5  2013     1     1     2230       222     2006       216          7
##  6  2013     1     1     2330       250     2115       255          5
##  7  2013     1     1       46       246     2205       285          6
##  8  2013     1     1       21       191     2312       192          8
##  9  2013     1     1      314       456     2343       379          2
## 10  2013     1     2     1431       207     1244       224          6
## # ... with 3,296 more rows
```

Question: Why did we have to group anything? What would have happened without grouping?

Let's focus on the worst 2 offenders per day to make it easier to see.

```
# Get the worst arriver for every day
flights_small %>%
  group_by(year, month, day) %>%
  mutate(delay_rank = rank(desc(arr_delay))) %>%
  filter(delay_rank <= 2)
```

```
## # A tibble: 722 x 8
## # Groups:   year, month, day [365]
##     year month   day arr_time arr_delay dep_time dep_delay delay_rank
##    <int> <int> <int>    <int>     <dbl>    <int>     <dbl>      <dbl>
##  1  2013     1     1     1001       851      848       853          1
##  2  2013     1     1      314       456     2343       379          2
##  3  2013     1     2     2003       368     1607       337          1
##  4  2013     1     2     2340       359     2131       379          2
##  5  2013     1     3     2339       270     2008       268          2
##  6  2013     1     3     2239       285     2056       291          1
##  7  2013     1     4        2       172     2058       208          2
##  8  2013     1     4     2332       276     2123       288          1
```

```
##  9  2013     1     5     1405       248     1232       257          2
## 10  2013     1     5     1635       308     1344       327          1
## # ... with 712 more rows
```

How many rows does this return? Why? How many rows were there before we filtered?

```
# Find all destinations that have more than one flight arriving (from NY) every day
popular_destinations <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
```

Question: Again, why did we have to group anything? Which function cares about this and works differently because of it?

Lesson: When you want to add information to rows that depends on other rows, you `group_by` subgroup before computing the new value. The summary function will then compute the mean, the rank, or anything else with respect to this subgroup and add this value to every row of the dataframe. Unlike `summarise`, it will return every row of the original dataframe, not only one row for every subgroup.

Class Exercises

The following chunks have most, but not all of the code necessary to achieve their goal. Fix as many of them as you can. **Note:** You'll have to switch off the `eval = FALSE`.

```
# Compute for every day how many minutes above the average delay a given flight is.

flights_small %>%
  group_by(...) %>%
  mutate(difference_from_daily_mean_dep_delay = <some-function-name>(...))
```

Now compute the daily standard deviation in departure delays. Explain in one sentence what it means for that to be higher or lower.

```
# Compute for every day how many minutes above the average delay a given flight is.

flights_small %>%
  group_by(...) %>%
  mutate(daily_sd_dep_delay = <some-function-name>(...))
```

Now combine the previous two commands with a third to compute how many standard deviations a given flight is from the mean. A value of -1 indicates that the flight had 1 sd less departure delay than the average flights that day, 1 indicates that it had 1 sd more departure delay than the average flight that day.

```
flights_small %>%
  group_by(...) %>%
  mutate(
    difference_from_daily_mean_dep_delay = ...,
    daily_sd_dep_delay  = ...,
    diff_from_daily_mean_in_sd = .../...
  )
```

And finally (if time permits, which is unlikely):

```
# What time of the day should you fly to avoid delays the most?
# Start with dep_time. Then realize this is bad.

not_missing %>%
  mutate(hour = dep_time %/% 100) %>%
  group_by(hour) %>%
```

```r
  summarise(delay = mean(arr_delay))

# Fix the variable, check what it computes before ranking
not_missing %>%
  mutate(hour = ...%/% 100) %>%
  group_by(hour) %>%
  summarise(delay = mean(arr_delay))

# Now rank (in reality you would change the earlier code, rather than repeat)
not_missing %>%
  mutate(hour = ...%/% 100) %>%
  group_by(hour) %>%
  summarise(delay = mean(arr_delay)) %>%
  mutate(rank_delay = rank(delay)) %>%
  arrange(rank_delay)
```

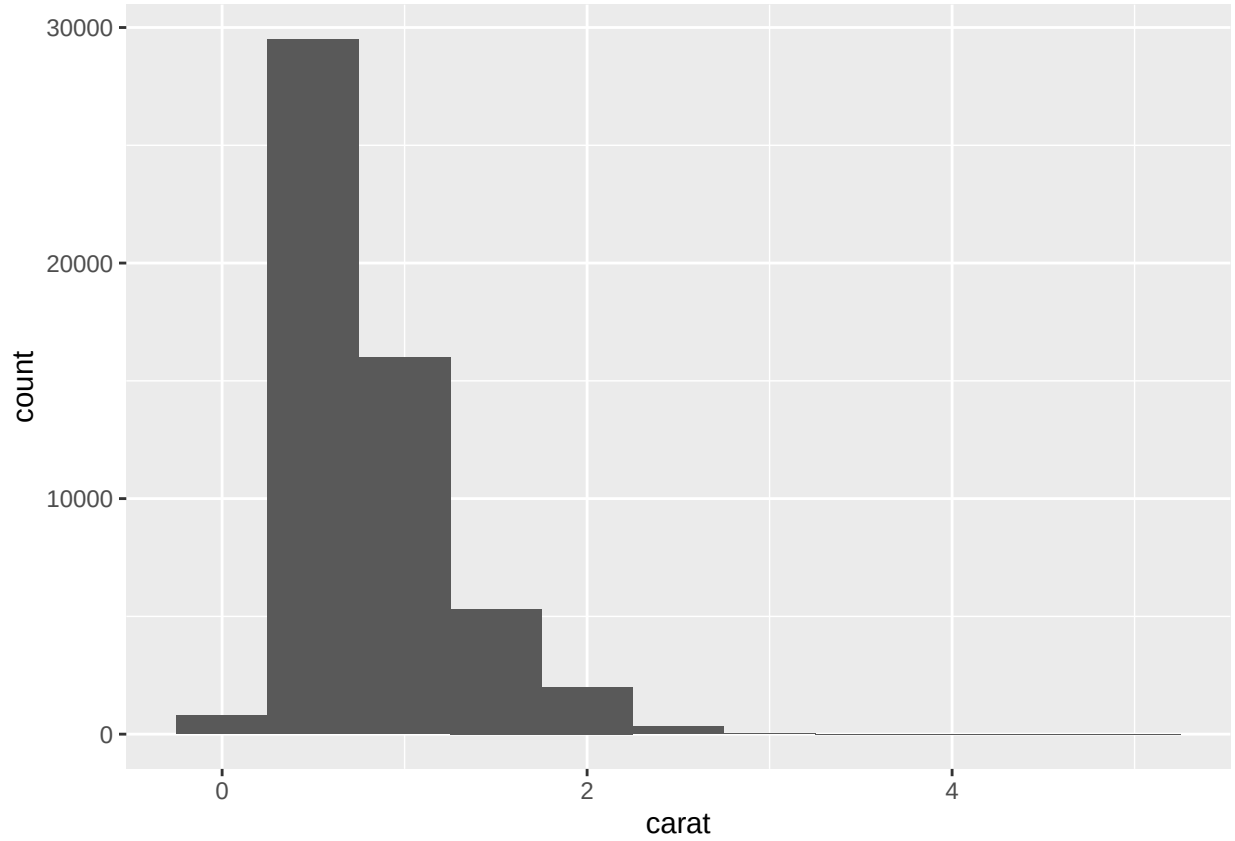
Chapter 7: Exploratory Data Analysis

We will not be able to cover much of chapter 7 in class, but you should work through it yourself in the next few weeks. Some things you should be aware of:

cut_width()

`cut_width()` allows you to turn a continuous range into a set of bins, which is useful if you want to make boxplots (or other plots) that require discrete bins for a continuous range. It is what `geom_histogram` uses.

```r
diamonds %>%
  ggplot(mapping = aes(x = carat)) +
  geom_histogram(binwidth = 0.5)
```



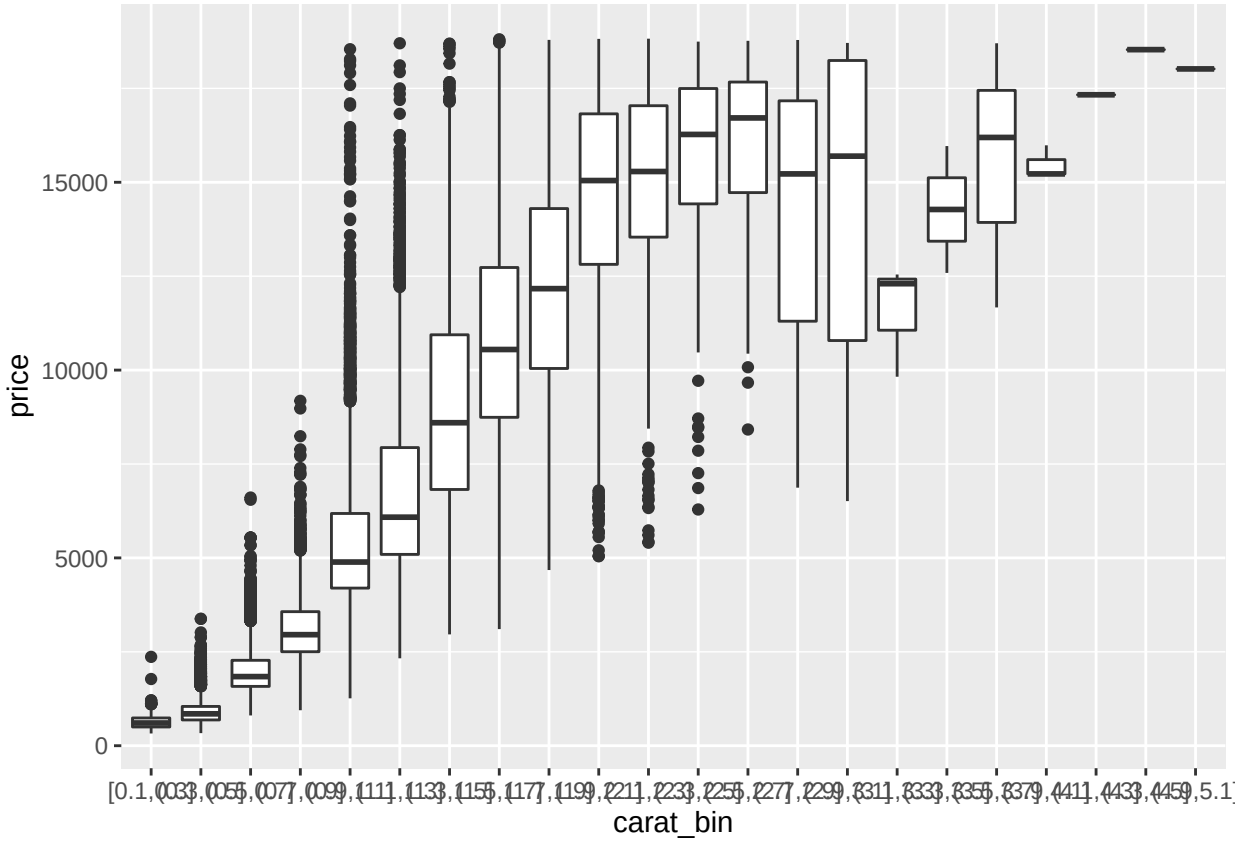

```
# If we wanted to generate the data directly that ggplot computes, use `cut_width()`

diamonds %>%
  mutate(interval = cut_width(carat, 0.5)) %>%
  group_by(interval) %>%
  summarise(n = n())
```

```
## # A tibble: 11 x 2
##    interval         n
##    <fct>        <int>
##  1 [-0.25,0.25]   785
##  2 (0.25,0.75]  29498
##  3 (0.75,1.25]  15977
##  4 (1.25,1.75]   5313
##  5 (1.75,2.25]   2002
##  6 (2.25,2.75]    322
##  7 (2.75,3.25]     32
##  8 (3.25,3.75]      5
##  9 (3.75,4.25]      4
## 10 (4.25,4.75]      1
## 11 (4.75,5.25]      1
```

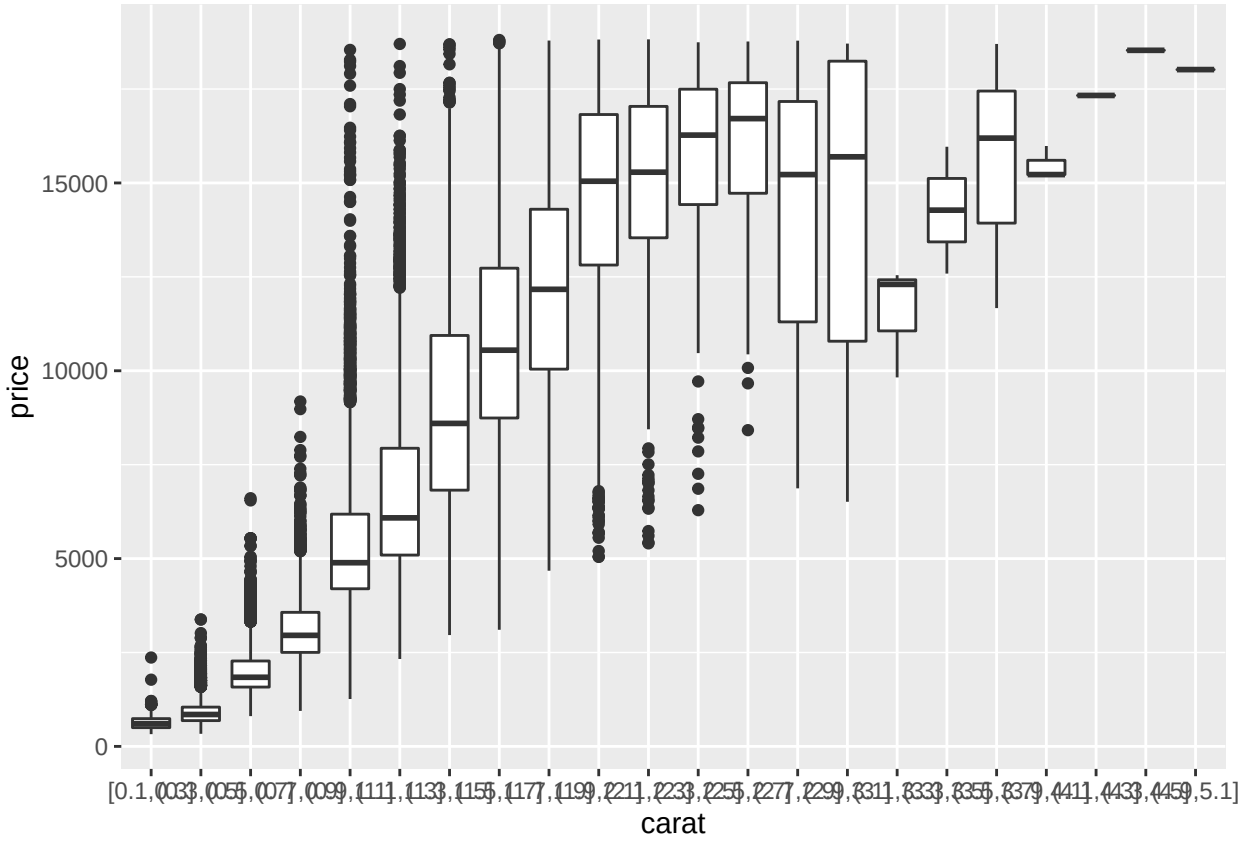
```
# Remember, count(<expression-to-group-on>) is equivalent to
# mutate(<etgo>) %>% group_by(<etgo>) summarise(n())

diamonds %>%
  count(cut_width(carat, 0.5))
```

```
## # A tibble: 11 x 2
##    `cut_width(carat, 0.5)`     n
##    <fct>                   <int>
##  1 [-0.25,0.25]              785
##  2 (0.25,0.75]             29498
##  3 (0.75,1.25]             15977
##  4 (1.25,1.75]              5313
##  5 (1.75,2.25]              2002
##  6 (2.25,2.75]               322
##  7 (2.75,3.25]                32
##  8 (3.25,3.75]                 5
##  9 (3.75,4.25]                 4
## 10 (4.25,4.75]                 1
## 11 (4.75,5.25]                 1
```

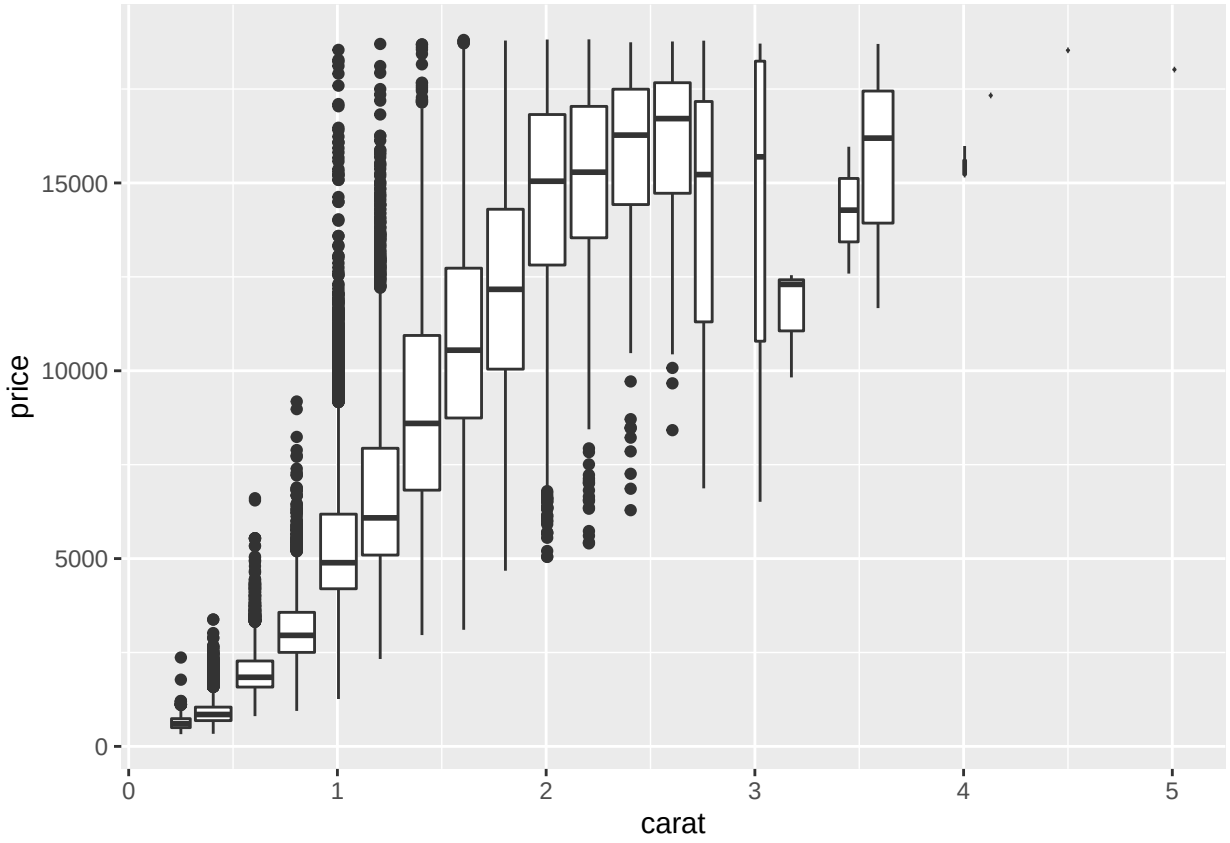
```
# Why do we care? The following should be done changing the code, rather than copy-pasting.

# Bin the carats, and make box plots for every bin

diamonds %>%
  mutate(carat_bin = cut_width(carat, 0.2)) %>%
  ggplot(mapping = aes(x = carat_bin, y = price)) +
  geom_boxplot()
```



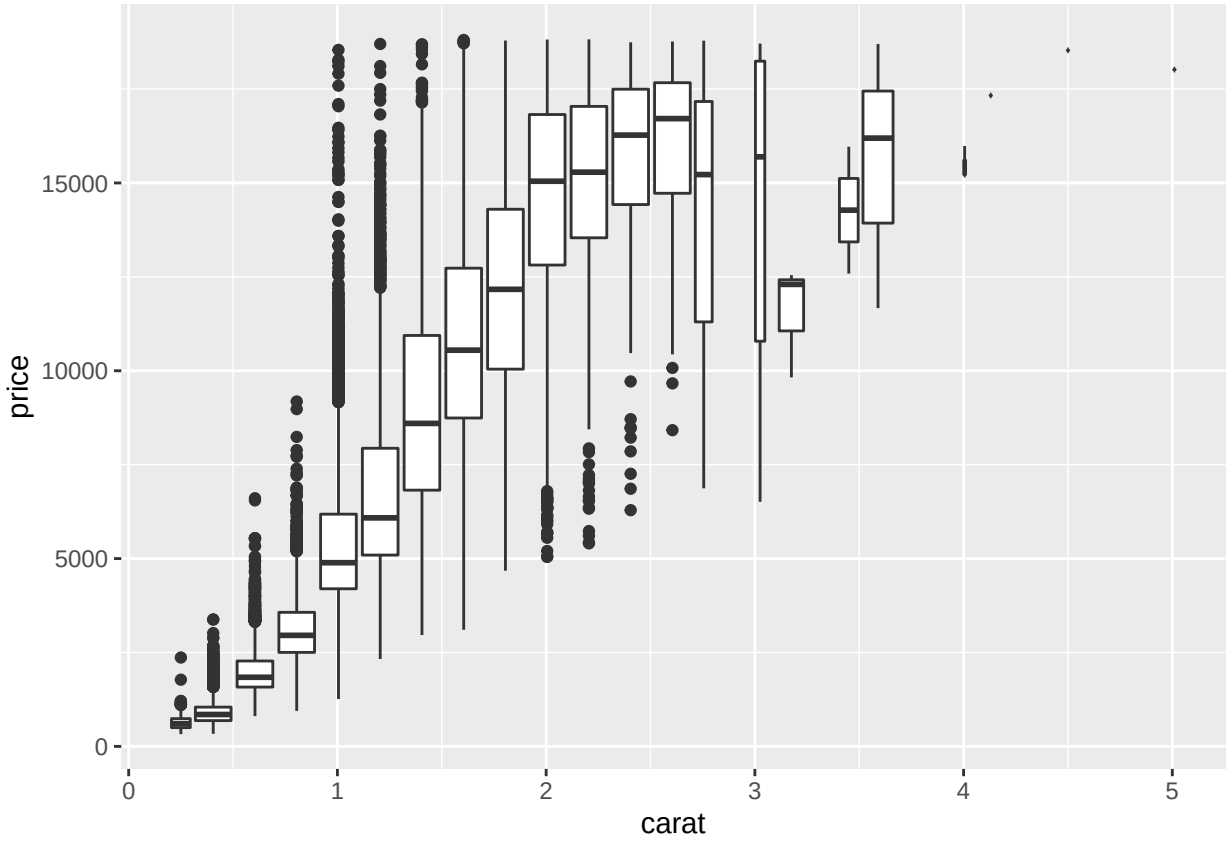

```
ggplot(data = diamonds,
       mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(x = cut_width(carat, 0.2)))
```



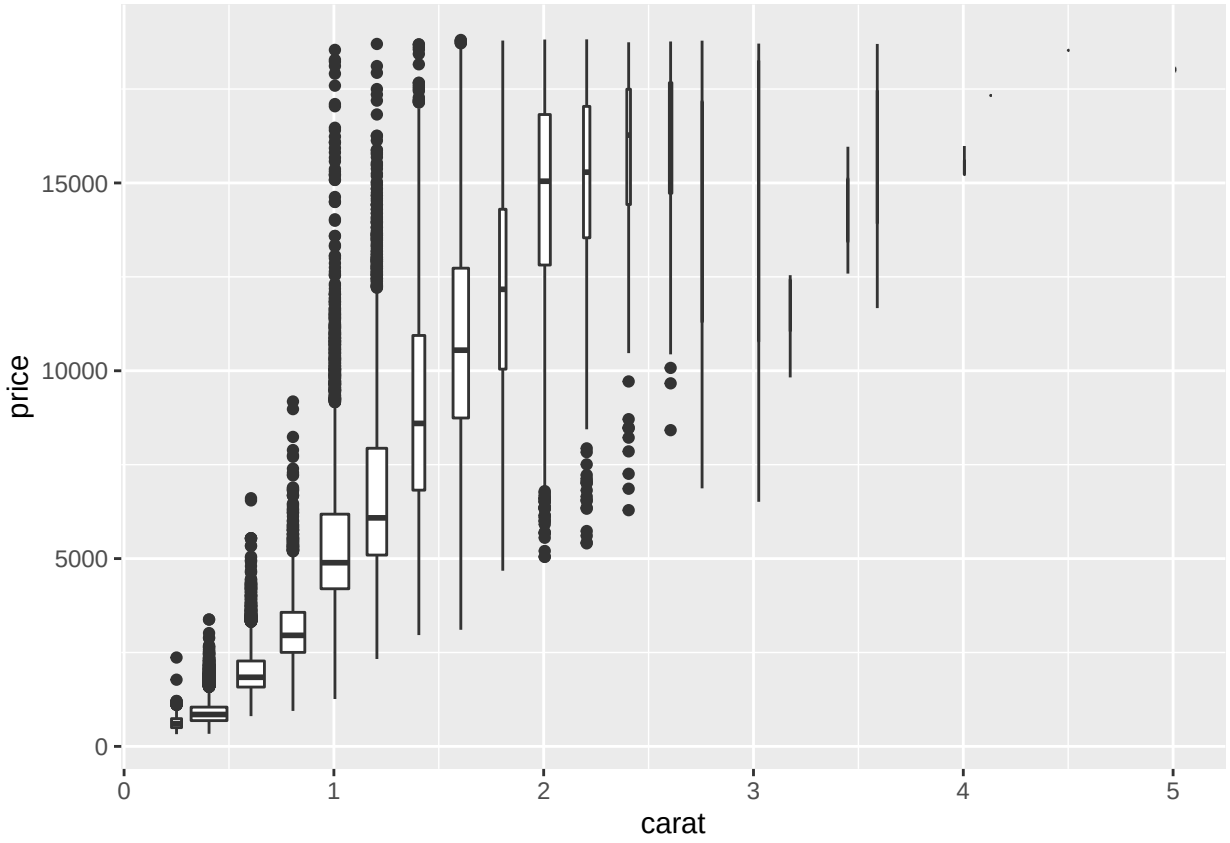

```
ggplot(data = diamonds,
       mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_width(carat, 0.2)))
```




```
bin_diamonds <- diamonds %>%
  mutate(carat_bin = cut_width(carat, 0.2))

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




```
ggplot(data = bin_diamonds,
       mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = carat_bin), varwidth = TRUE)
```



ifelse() and case_when()

```
x <- 1:10
ifelse(x %% 2 == 0, 'even', 'odd')
```

```
##  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

```
ifelse(1:10 %% 2 == 1, 'odd', 'even')
```

```
##  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

```
# There is also case_when
?case_when()
case_when(
  x %% 3 == 0 ~ "divisible by 3",
  x %% 3 == 1 ~ "+1",
  x %% 3 == 2 ~ "-1"
)
```

```
##  [1] "+1"             "-1"             "divisible by 3" "+1"
##  [5] "-1"             "divisible by 3" "+1"             "-1"
##  [9] "divisible by 3" "+1"
```

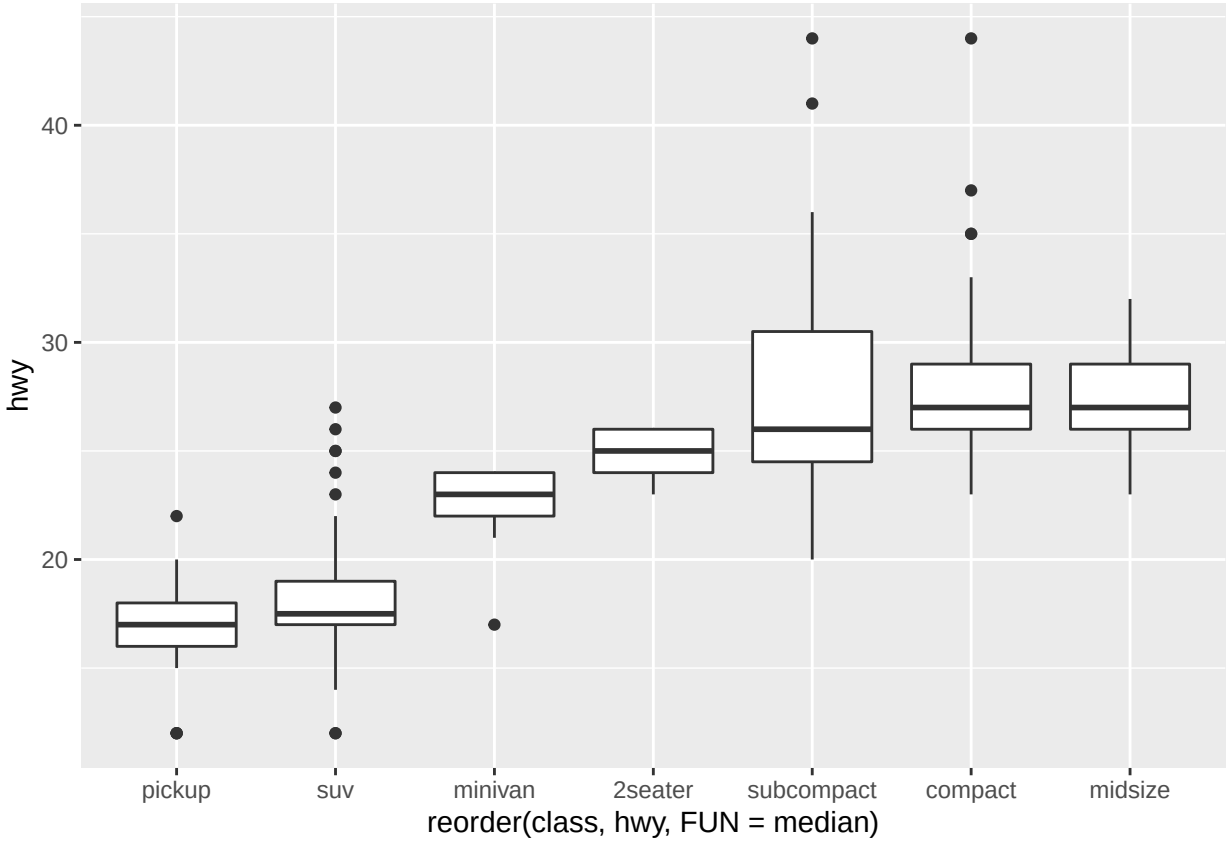
This is vectorized, so you can use it in mutates:

```
# Let'd deal with weird variables in diamonds, which happen to be for y below 3 and above 20
diamonds2 <- diamonds %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y))
```

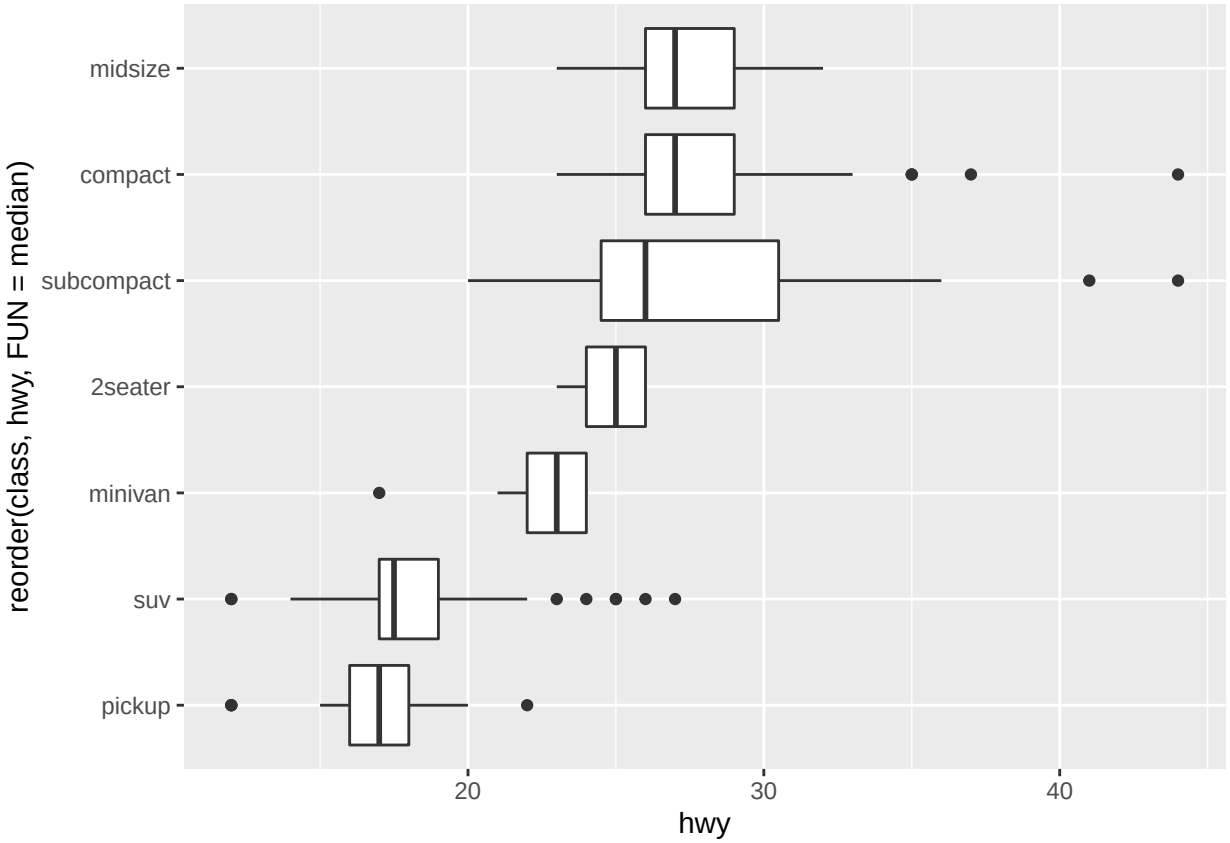
geom_boxplot

Look at the code below and try to think about what it does before running it, in particular the `FUN = median`.

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




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



There are several other geom_* that you should know of and the assignment asks you to find out more about them.

Assignment 6

Notes:

1. Please highlight the start of each exercise properly, as well as where your answer starts and ends.
2. Upload **only your exercises** and answers to Moodle, not all of the lecture notes.

Exercise 0: Manage your time. If you spend too much time on the assignment, it may be too long, and you may benefit from skipping an exercise. Since this lesson is lost on most people, exercise 0 requires you to skip one of the other exercises this week.[1] That is, write down which exercise you want to skip and why ("saves the most of time because it is the hardest" or "it's the most useless as you are quite confident in material"). Do not work on it. Of course, if you skip any additional exercise due to lack of time, highlight this, but that will cost some part of the grade, whereas skipping an exercise as part of exercise 0 will not.

Exercise 1: Using the nycflights13 data. Note that it also contains a tibble called `airports` (as well as others). Use these two dataframes to find the answer to 4 of the following, and print them out in a separate chunk (i.e. the chunk should print the tibble, thus showing the first 10 lines of each):

- The number of flights (in the whole year) to each destination
- The number and list of distinct airports in the US
- The number and list of distinct airports that have at least one flight in the whole year from NYC
- The number and list of distinct airports that have at least one flight **per day** from NYC
- The number airports that are further south than NYC (Hint: look up longitude and latitude.)
- The top 5 carriers that have the lowest average delay times.

[1] If you want to be cute, skip exercise 0.

- What is the worst day of the year to fly in terms of arrival delay?
- What is the best day of the year to fly in terms of departure delay?

Reminder: Pick only 3 of the 8 possible ones.

Exercise 2: Find all the rows with NA values in two specific columns (of your choosing) for the following datasets:

- diamonds
- flights
- mtcars

If you can't find a dataset, do `??<dataset>` to find out more about it and what library you need to load.

If you had to find all the rows containing `NA` values in some column, how would use filter for that? Don't write this command out, but you should realize that this would be unpleasant (and errorprone).

Exercise 3: Look up `filter_all` and look at the examples at the end of the documentation. Use this (with some google-fu or discourse help) to find all the rows with NA values in *any* column for the following datasets:

- diamonds
- flights
- mtcars

Thus, the output should be those rows that *do* contain NA values.

Exercise 4: Pick your favourite dataset. Use it to illustrate the following types of plots, and describe briefly (2-3 sentences) what each plot means. I.e. for the boxplot, what do different lines mean in general, and thus what do they say about the specific data, such as the mean?

- boxplot
- violin plot
- boxploth (Hint: You need to import the right library for this)
- bin2d
- hex

Thus the code for each plot should be `ggplot(data = <your-data-set>, mapping = aes(...) + geom_...()` and is not the main challenge. The main part is for you to look at and understand the data.

Exercise 5: Come up with an exercise to help you – and others – learn `summarise` and `group_by` better. The more confused you are, the more you should simply try to come up with, or even copy, an example from somewhere and highlight what confuses you. Is it the order or arguments? Their role? If you are less confused, try to find a (non-obvious) use. Mention any resources used.

Exercise 6: Work through sections 11.1 and 11.2 (skip exercises).