ISOM 3390: Business Programming in R

Instructor: Bingjie Qian (PhD candidate)

Summer 2020, HKUST

Topic 8: `ggplot2` Plotting System

8.1 An Overview of `ggplot2`

In Topic 7, we learnt the base plotting system in R. The base plotting system has many useful functions that allow us to create a variety of statistical plots, e.g., scatter plots, boxplots, histograms, etc.

Although the base plotting system is suited to do quick plotting for data exploration, creating complex graphs and overlays with base plotting system can be very complicated and time consuming.

In this topic, we will learn another plotting system `ggplot2`, which is the data visualization piece of the `tidyverse` collection.

Generally speaking, `ggplot2` has a slightly steeper learning curve than the base plotting system, but once you get used to the plotting framework of `ggplot2`, it provides a very concise and powerful language for creating plots.

Grammar of Graphics

Human languages, like English, are built on grammars. In the following sentence, every word has a clear grammatical definition.

The	quick brown	fox	jumps	over	the	lazy	dog.
Article	Adjective	Noun	Verb	Preposition	Article	Adjective	Noun

In `ggplot2`, graphics are also built on an underlying grammar. The **grammar of graphics** is a plotting framework developed by Leland Wilkinson (The Grammar of Graphics, 1999).

There are **two principles** for the grammar of graphics:

1. Graphics are made up of distinct layers of grammatical elements.

2. Graphics are built around aesthectic mappings.

The layers are like the adjectives and nouns, and the aesthetic mappings are like the grammatical rules for how to assemble the vocabulary.

Aesthetic mappings describe how variables in the data are mapped to visual properties (aesthetics) of geometric objects (geoms).

Grammatical Elements:

- Essential grammatical elements
  - **Data**: the data being plotted
  - **Geometries**: Geometric objects used to plot each observation in the data, e.g., lines, points, bars, polygons, etc.
  - **Aesthetics**: Visual features onto which we map variables in the data, e.g., color, fill, shape, size, linetype, pointtype, etc.
- Optional grammatical elements
  - **Statistics**: Statistical transformations that summarise data in many useful ways, e.g., binning and counting observations to create a histogram, summarising a 2d relationship with a linear model, etc.
  - **Positions**: Position adjustments that deal with overlapping geometric objects, e.g., dodging objects side-to-side, jittering points, stacking objects on top of each another, etc.
  - **Scales**: Specify how values in the data space should be mapped to values in the aesthetic space.
  - **Facets**: Define how to break up the data into subsets and display those subsets as small multiples.
  - **Coordinates**: Describe how data coordinates are mapped to the plane of the graphic.
  - **Themes**: Control all non-data elements of a plot.

8.2 Geoms

In the following example, we will use `mpg`, a dataset that comes with the package `ggplot2`.

```
library(tidyverse)

## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──

## ✓ ggplot2 3.3.2     ✓ purrr   0.3.4
## ✓ tibble  3.0.1     ✓ dplyr   1.0.0
## ✓ tidyr   1.1.0     ✓ stringr 1.4.0
## ✓ readr   1.3.1     ✓ forcats 0.5.0

## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

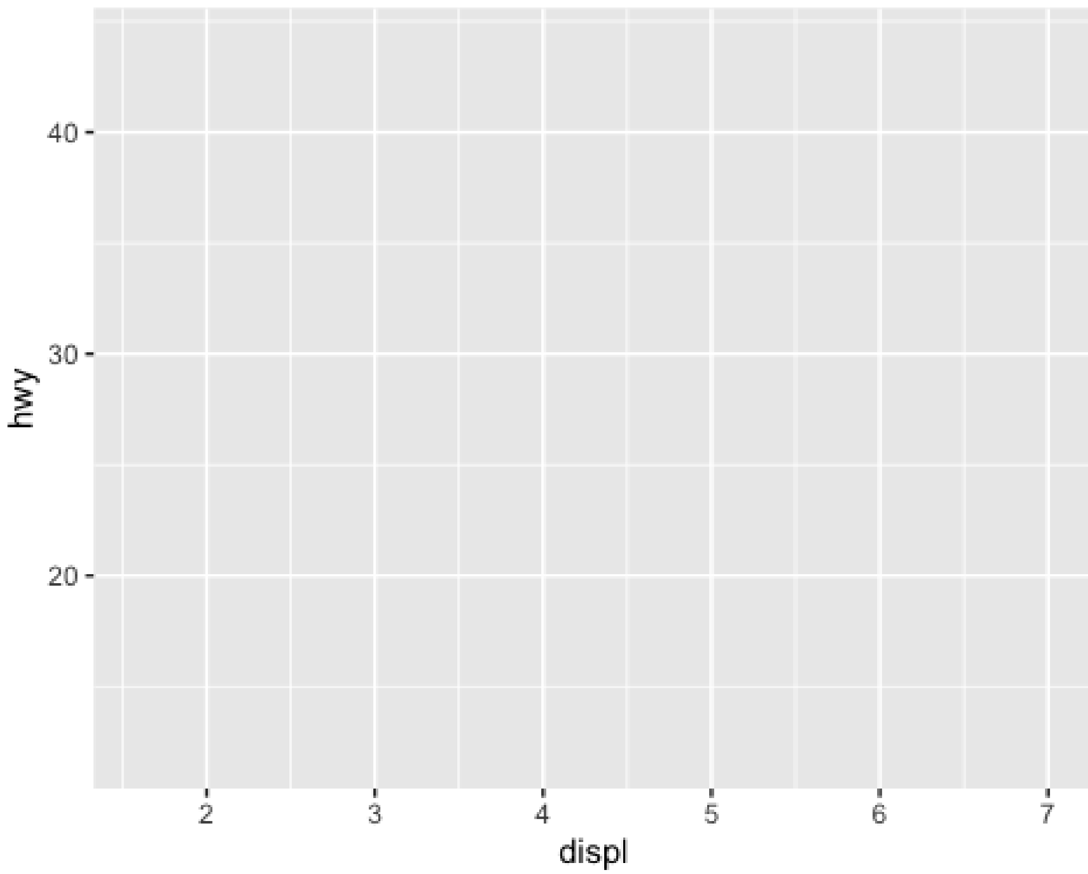
```
?mpg
head(mpg)

## # A tibble: 6 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(l5)   f        18    29 p
compa…
## 2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p
compa…
## 3 audi         a4      2    2008     4 manual(m6) f        20    31 p
compa…
## 4 audi         a4      2    2008     4 auto(av)   f        21    30 p
compa…
## 5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p
compa…
## 6 audi         a4      2.8  1999     6 manual(m5) f        18    26 p
compa…
```

The `ggplot()` Function

`ggplot()` *initializes* a ggplot object.

It can be used to declare the **input data frame** for a graphic, and to specify the set of **plot aesthetics** intended to be common throughout all subsequent layers unless specifically overridden.

```
p <- ggplot(data = mpg, mapping = aes(x = displ, y = hwy))
p
```



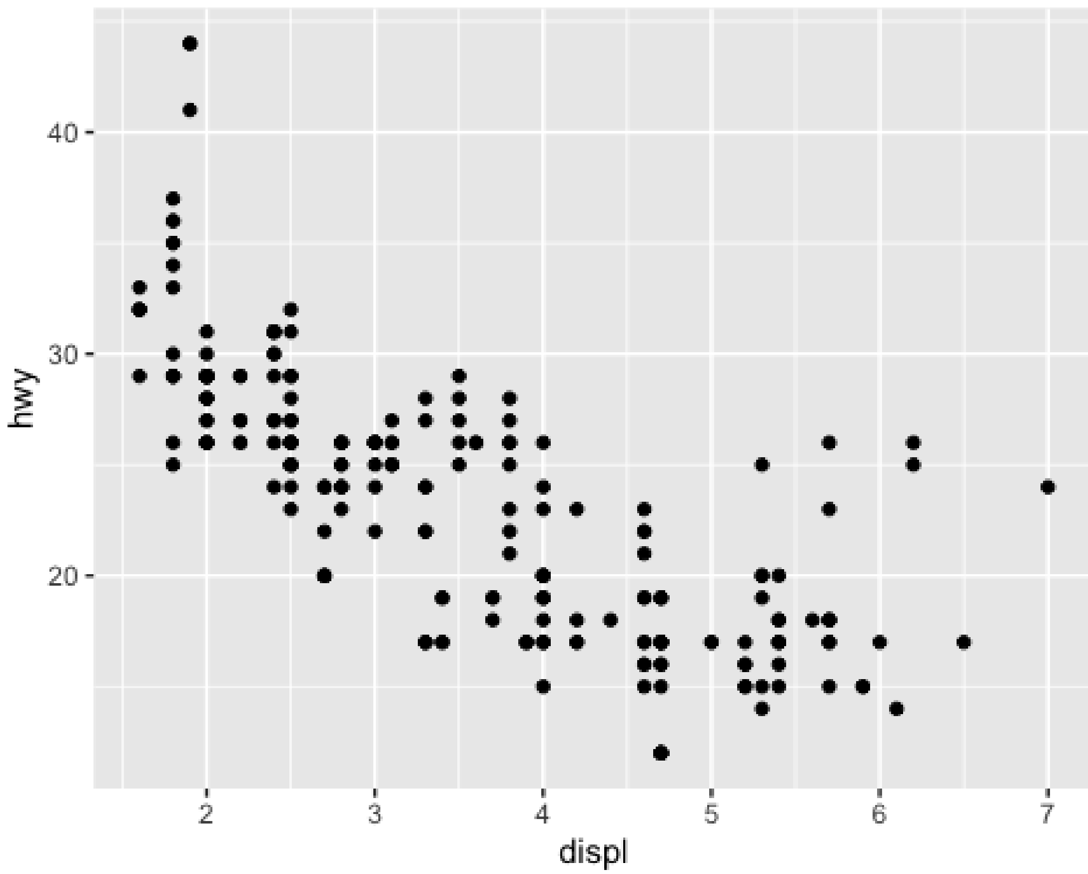
In the above code, we use `aes()` to define the aesthetic mappings.

We have initialized a `ggplot` object, but there's nothing in the plot until a layer of geom is added.

The `geom_*()` Functions

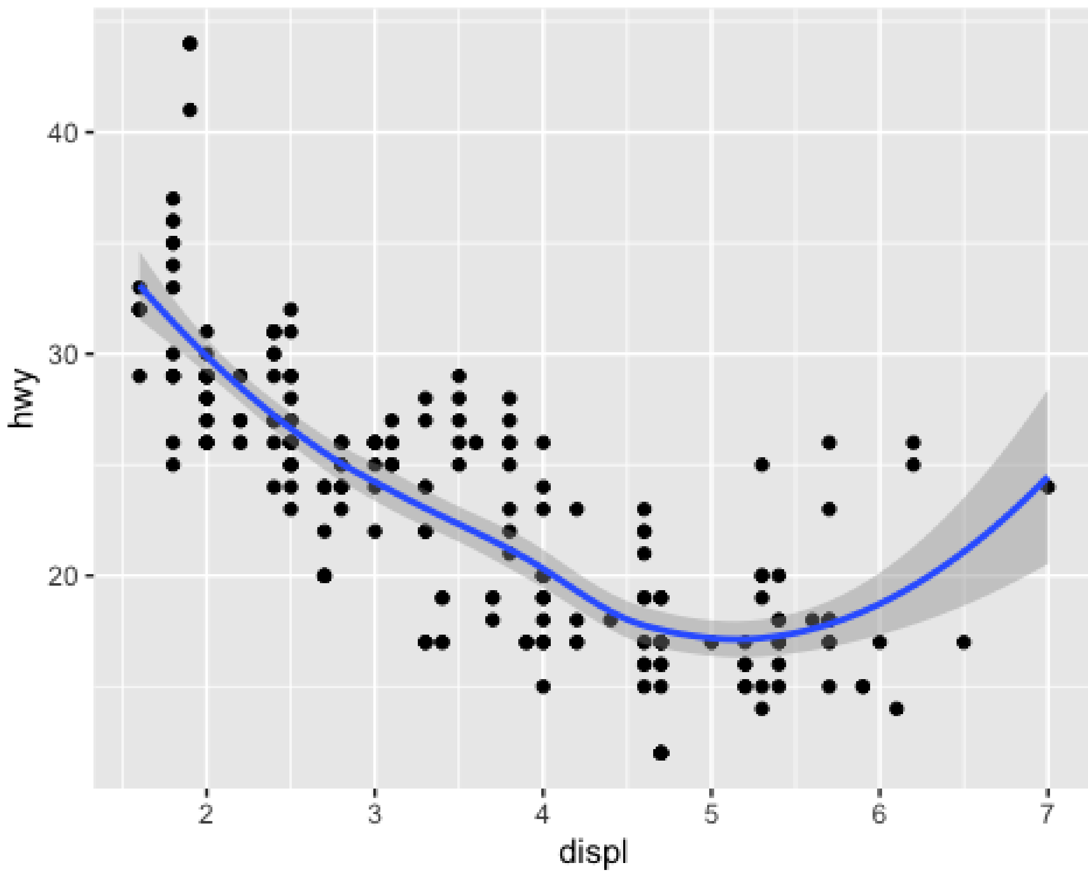
We can add a geom to a plot using the `+` operator and the `geom_*()` functions. Geoms perform the actual rendering of a layer and control the type of plot to be created.

```
# add a scatter plot:
p + geom_point()
```



We can continue to add more geoms:

```
# add a smoothed line to reveal the dominant pattern among the points:
p + geom_point() + geom_smooth()

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



The code snippet below shows a list of available geometric functions starting with geom_.

```
geom_fun <- str_extract(ls(pos = "package:ggplot2"), "^geom_.*") %>%
str_extract("^geom_.*")
geom_fun[!is.na(geom_fun)]

##  [1] "geom_abline"            "geom_area"              "geom_bar"
##  [4] "geom_bin2d"             "geom_blank"             "geom_boxplot"
##  [7] "geom_col"               "geom_contour"
"geom_contour_filled"
## [10] "geom_count"             "geom_crossbar"          "geom_curve"
## [13] "geom_density"           "geom_density_2d"
"geom_density_2d_filled"
## [16] "geom_density2d"         "geom_density2d_filled"  "geom_dotplot"
## [19] "geom_errorbar"          "geom_errorbarh"         "geom_freqpoly"
## [22] "geom_function"          "geom_hex"               "geom_histogram"
## [25] "geom_hline"             "geom_jitter"            "geom_label"
## [28] "geom_line"              "geom_linerange"         "geom_map"
## [31] "geom_path"              "geom_point"             "geom_pointrange"
## [34] "geom_polygon"           "geom_qq"                "geom_qq_line"
## [37] "geom_quantile"          "geom_raster"            "geom_rect"
## [40] "geom_ribbon"            "geom_rug"               "geom_segment"
## [43] "geom_sf"                "geom_sf_label"          "geom_sf_text"
```

```
## [46] "geom_smooth"              "geom_spoke"               "geom_step"
## [49] "geom_text"                "geom_tile"                "geom_violin"
## [52] "geom_vline"
```

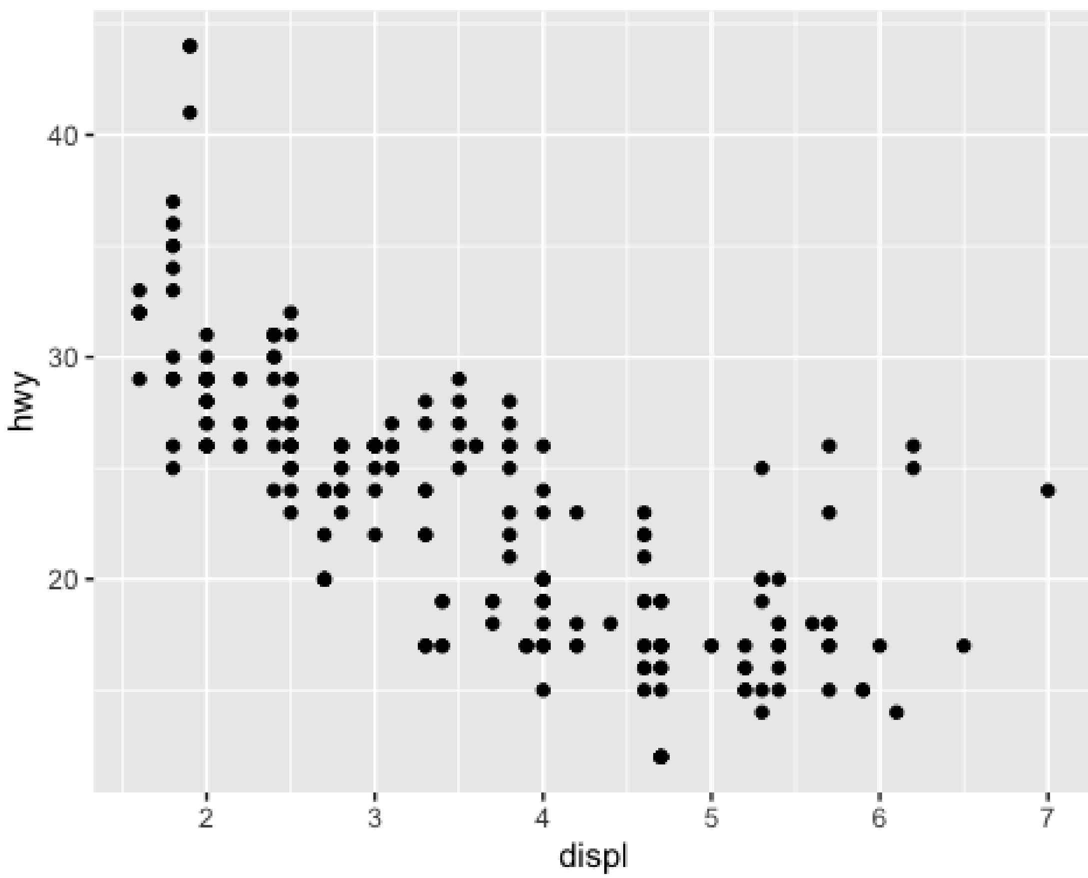
All `geom_*()` functions (like `geom_point()`) return a layer that contains a `Geom*` object (like `GeomPoint`). The `Geom*` object is responsible for rendering the data in the plot.

By adding more components, we can construct very sophisticated plots.

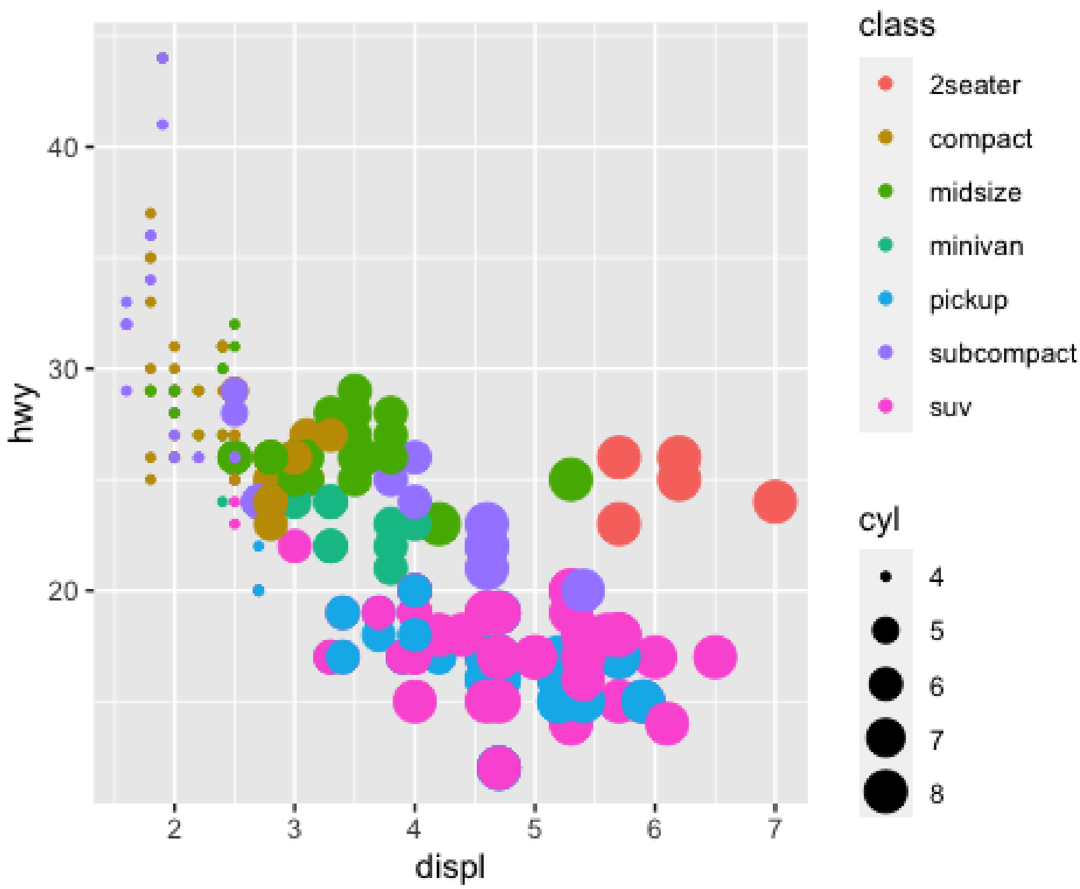
8.3 Aesthetics

To represent additional variables in the plot, we can use other aesthetics like colour, shape, and size, which are added into `aes()`:

```
ggplot(mpg, aes(displ, hwy)) + geom_point()
```



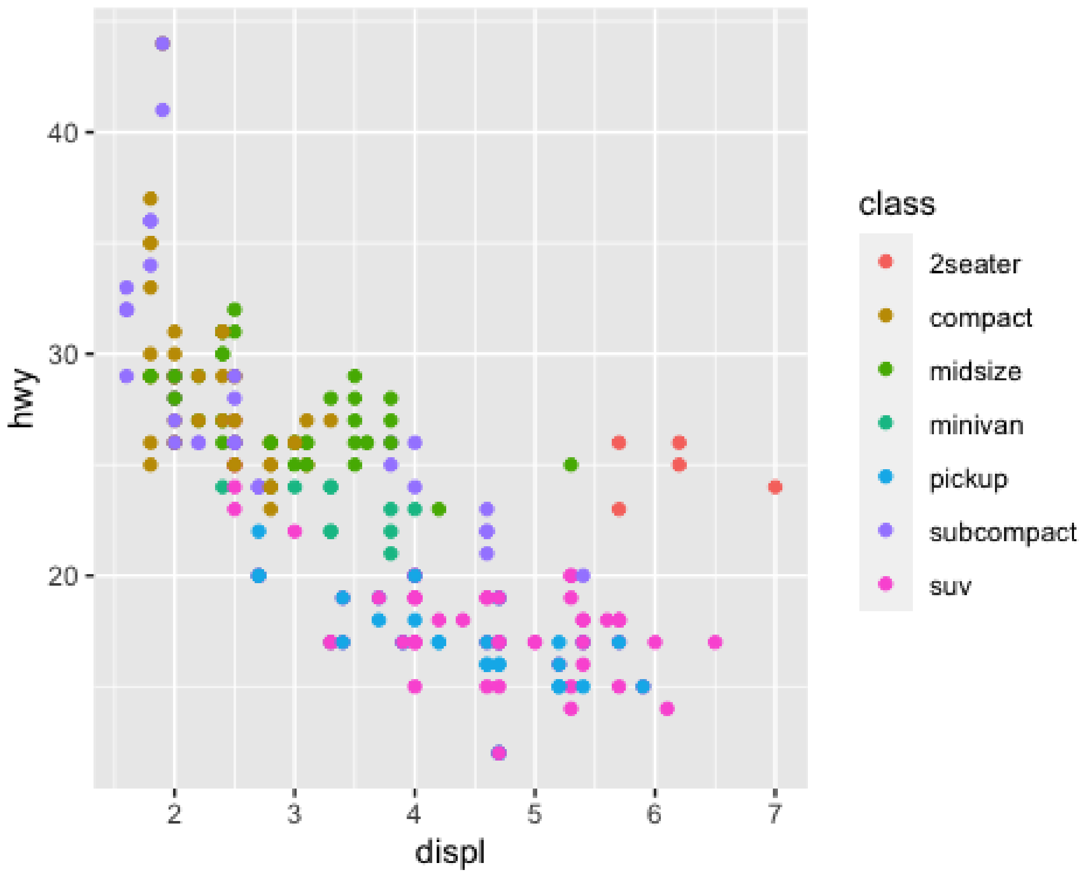

```
ggplot(mpg, aes(displ, hwy, colour = class, size = cyl)) + geom_point()
```



In the above plot, the variable `class` determines the colour of the points, and `cyl` determines the size of the points. The legend is generated automatically.

Aesthetic mappings can be supplied in the initial `ggplot()` call, in individual layers, or in the combination of both. The following four expressions generate the same plot:

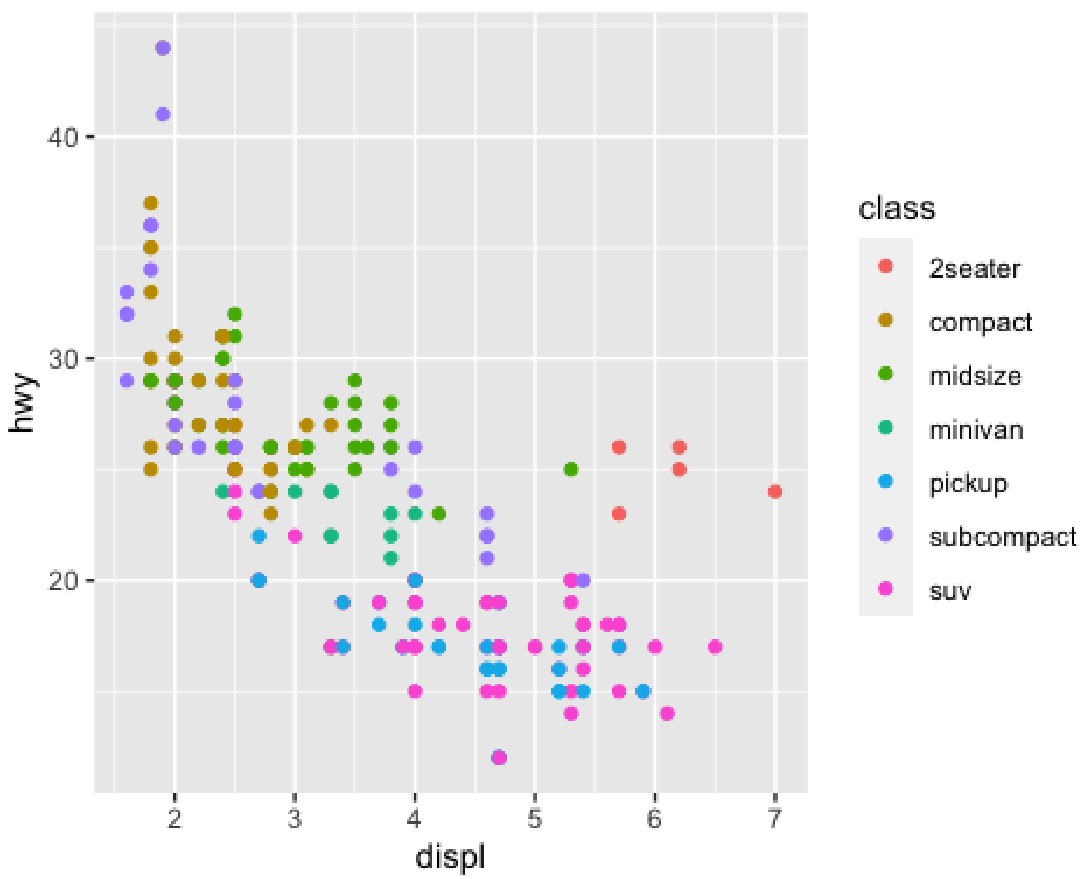
```
ggplot(mpg, aes(displ, hwy, colour = class)) + geom_point()
```



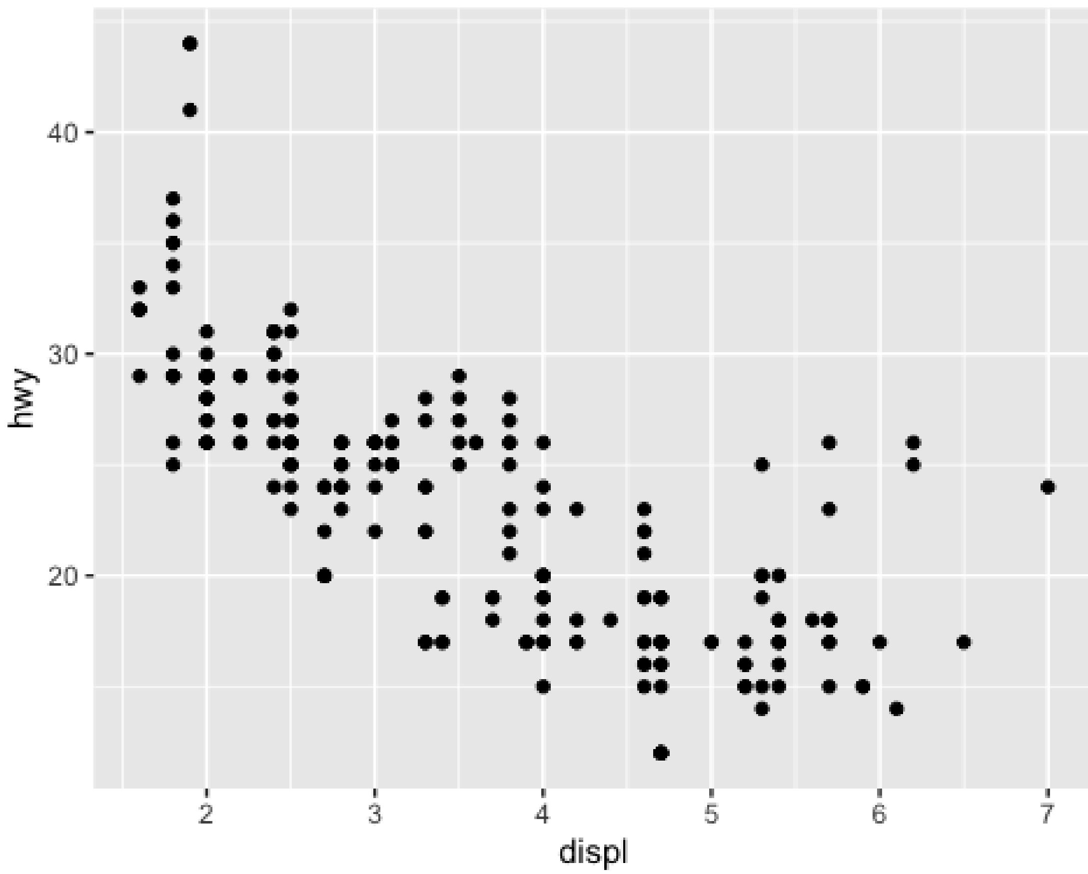

```
# ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = class))
# ggplot(mpg, aes(displ)) + geom_point(aes(y = hwy, colour = class))
# ggplot(mpg) + geom_point(aes(displ, hwy, colour = class))
```

The aesthetic mappings supplied in individual layers (e.g., `geom_point`) can add, remove, or override the aesthetic mappings supplied in the initial `ggplot()` call.

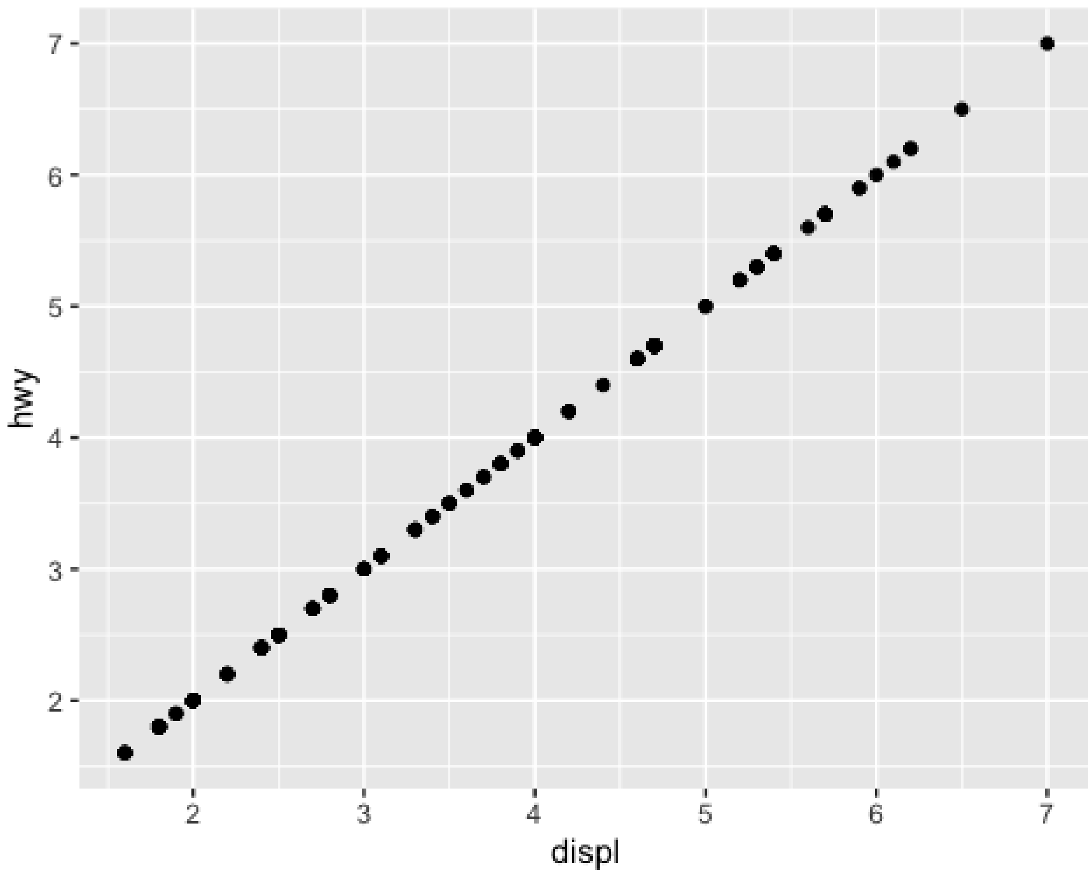
```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = class))  # add
```




```
ggplot(mpg, aes(displ, hwy, colour = class)) + geom_point(aes(colour = NULL))
# remove
```

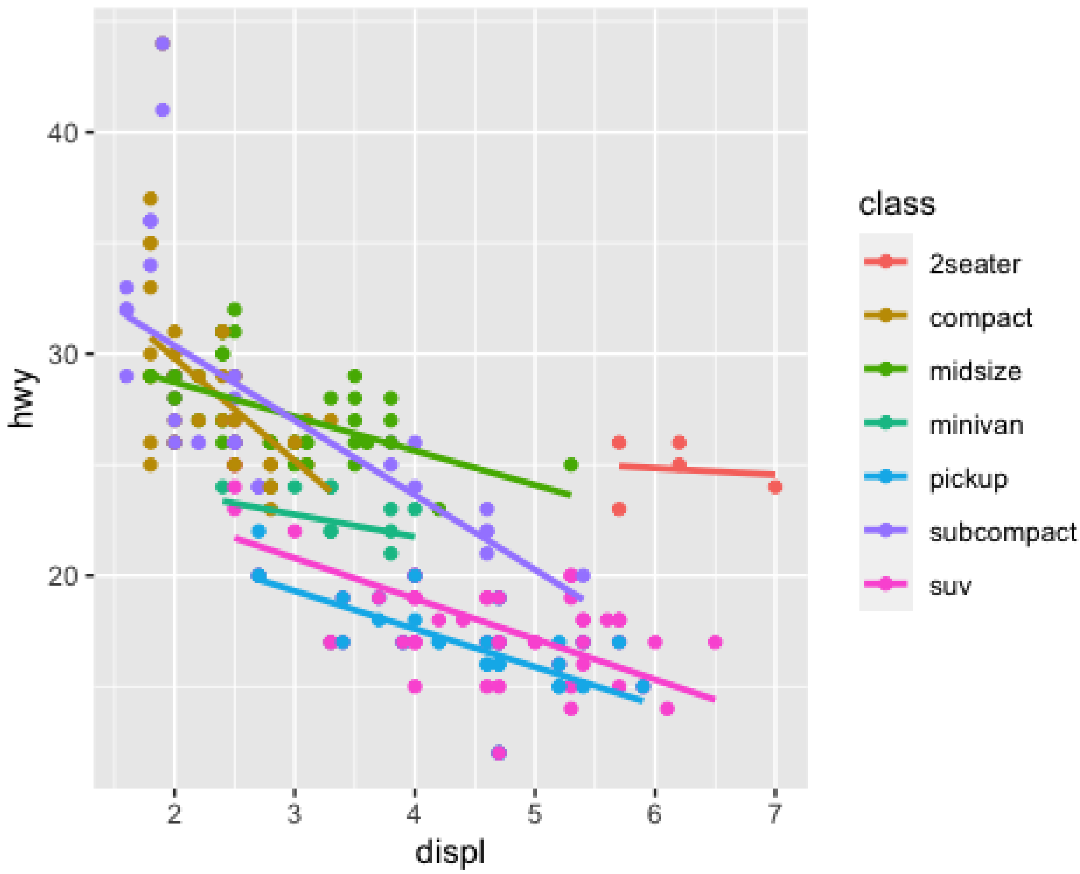



```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(y = displ))       # override (y
axis label unchanged)
```

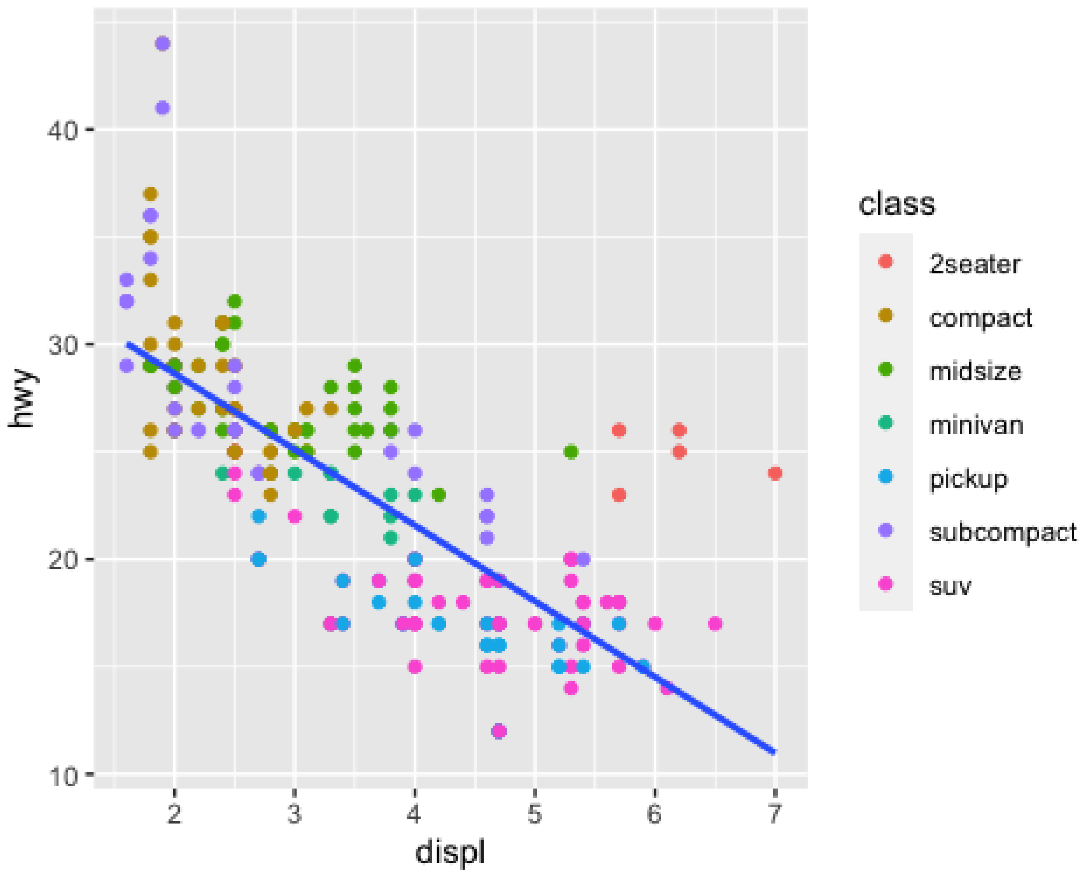


The *scope* of the aesthetic mappings supplied in the initial plot and in the layers are different. Aesthetic mappings in the initial plot will affect all subsequent layers.

```
# `colour = class` supplied in the initial plot:
ggplot(mpg, aes(displ, hwy, colour = class)) + geom_point() +
  geom_smooth(method = "lm", se = FALSE)

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




```r
# `colour = class` supplied in the layer geom_point:
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = class)) +
  geom_smooth(method = "lm", se = FALSE)

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



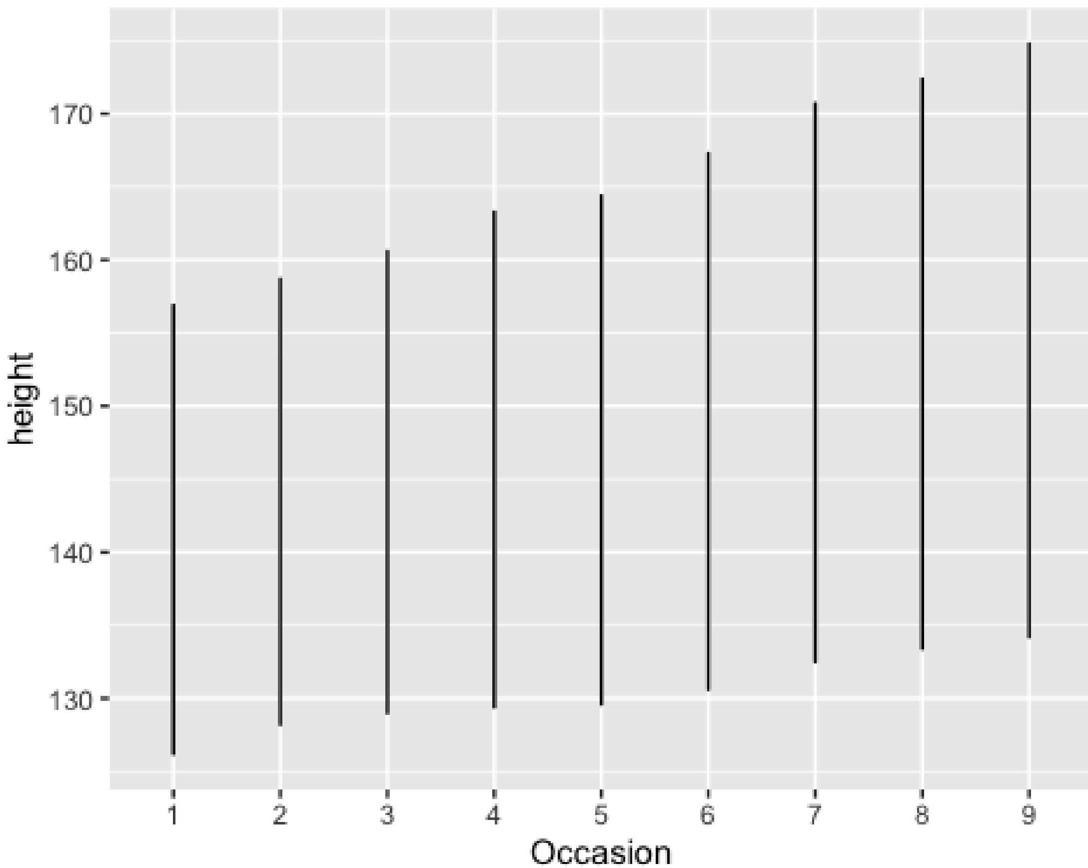
The group Aesthetic

In the following example, we use the data set `Oxboys` from the package `nlme`, which stores the height of 26 boys, each measured on 9 occasions (at different ages).

```
head(nlme::Oxboys)

## Grouped Data: height ~ age | Subject
##   Subject     age height Occasion
## 1       1 -1.0000  140.5        1
## 2       1 -0.7479  143.4        2
## 3       1 -0.4630  144.8        3
## 4       1 -0.1643  147.1        4
## 5       1 -0.0027  147.7        5
## 6       1  0.2466  150.2        6
```

Suppose we want to plot a growth trajectory for each boy by connecting the height records of each boy at different ages (different occasions). The function `geom_line()` connects the observations in order of the variable on the x axis.

```
h <- ggplot(nlme::Oxboys, aes(Occasion, height))
h + geom_line()
```



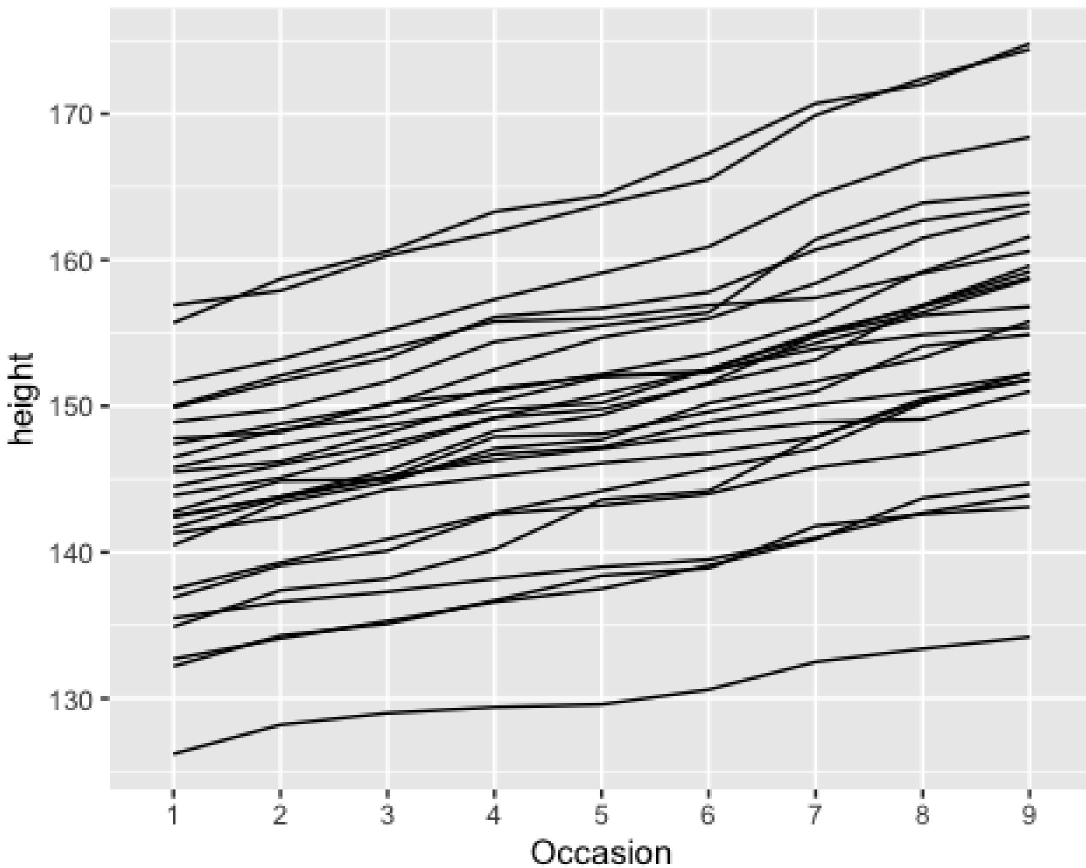
By default, the `group` aesthetic is set to *the interaction of all discrete variables* in the plot.

The above plot involves two variables, `Occasion` (discrete) and `height` (continuous). Therefore, the `group` aesthetic is mapped to `Occasion`, and the `geom_line` layer inherits the group aesthetic from the initial plot.

```
# check the variable types
class(nlme::Oxboys$Occasion)

## [1] "ordered" "factor"

class(nlme::Oxboys$height)

## [1] "numeric"
```

When the default grouping does not partition the data correctly, we need to override the default grouping by setting the `group` aesthetic explicitly.

```
h + geom_line(aes(group = Subject))
```



[Task 1: Plotting Billboard Ranking]

Download the **billboard.csv** file from Canvas, read and save it as a tibble. Drop the columns of meta data and last several weeks, select a sample of 11 songs by 5 artists, and name the resulting tibble `billboard_sample`:

```
billboard <- read_csv("billboard.csv")

## Parsed with column specification:
## cols(
##   .default = col_double(),
##   artist = col_character(),
##   track = col_character(),
##   time = col_time(format = ""),
##   genre = col_character(),
##   date.entered = col_character(),
##   date.peaked = col_character()
## )

## See spec(...) for full column specifications.
```

```
billboard_sample <- billboard %>% select(-c(3:6, 39:70)) %>% filter(artist
%in% c("Eminem", "3 Doors Down", "Carey, Mariah", "Creed", "Aaliyah"))
billboard_sample
```

```
## # A tibble: 11 x 34
##    artist track week1 week2 week3 week4 week5 week6 week7 week8 week9
week10
##    <chr>  <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
<dbl>
##  1 Creed  With…    84    78    76    74    70    68    74    75    69
74
##  2 Aaliy… Try …    59    53    38    28    21    18    16    14    12
10
##  3 Carey… Than…    82    68    50    50    41    37    26    22    22
2
##  4 3 Doo… Kryp…    81    70    68    67    66    57    54    53    51
51
##  5 Eminem The …    70    32    20    16    11     7     6     4     5
4
##  6 Creed  High…    81    77    73    63    61    58    56    52    56
57
##  7 Carey… Cryb…    28    34    48    62    77    90    95    NA    NA
NA
##  8 Aaliy… I Do…    84    62    51    41    38    35    35    38    38
36
##  9 Eminem Stan     78    67    57    57    51    51    51    57    55
70
## 10 3 Doo… Loser    76    76    72    69    67    65    55    59    62
61
## 11 Eminem The …    87    74    59    65    59    58    59    62    89
86
## # … with 22 more variables: week11 <dbl>, week12 <dbl>, week13 <dbl>,
## #   week14 <dbl>, week15 <dbl>, week16 <dbl>, week17 <dbl>, week18 <dbl>,
## #   week19 <dbl>, week20 <dbl>, week21 <dbl>, week22 <dbl>, week23 <dbl>,
## #   week24 <dbl>, week25 <dbl>, week26 <dbl>, week27 <dbl>, week28 <dbl>,
## #   week29 <dbl>, week30 <dbl>, week31 <dbl>, week32 <dbl>
```

(a) Convert `billboard_sample` to long format. Name the new tibble `billboard_long`.

The expected output is

```
# A tibble: 231 x 4
   artist track                   week  position
   <chr>  <chr>                   <fct>    <dbl>
 1 Creed  With Arms Wide Open 1           84
 2 Creed  With Arms Wide Open 2           78
 3 Creed  With Arms Wide Open 3           76
 4 Creed  With Arms Wide Open 4           74
 5 Creed  With Arms Wide Open 5           70
 6 Creed  With Arms Wide Open 6           68
 7 Creed  With Arms Wide Open 7           74
```

```
 8 Creed  With Arms Wide Open 8          75
 9 Creed  With Arms Wide Open 9          69
10 Creed  With Arms Wide Open 10         74
# . with 221 more rows
```

Please pay attention to the type of week values in `billboard_long`.

Tips:

1. Use `names_prefix` and `names_ptypes` of `pivot_longer()`; specifically, set `names_ptypes = list(week = factor())`;
2. Drop missing values by setting `values_drop_na`.

(b) Create a plot to show how the ranking positions of these 11 songs vary across weeks. Use a distinct color for each song. The expected output is as follows:

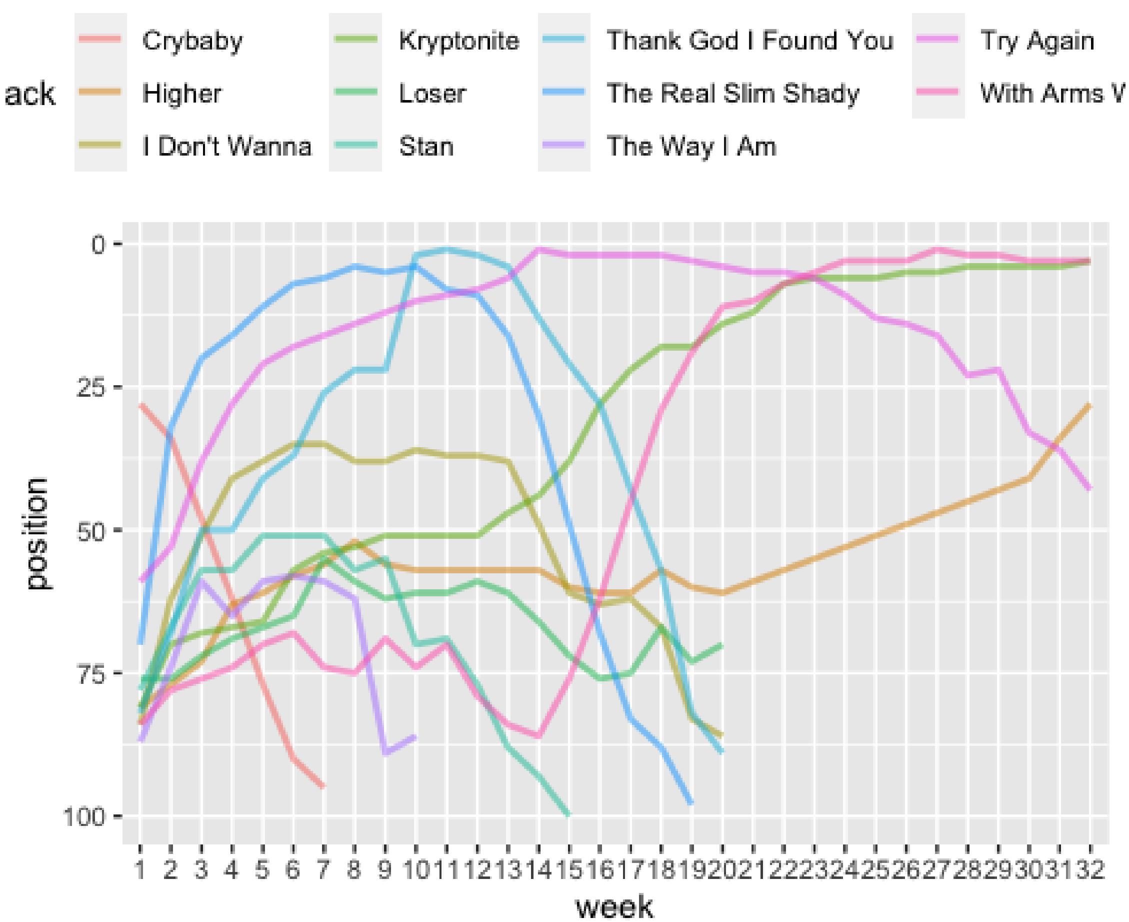


Figure 1. Task 1 (b)

Tips:

1. Use the group aesthetic;
2. `size = 1, alpha = 0.5` for lines;

3. + `scale_y_reverse()`; this is to reverse the y scale
4. + `theme(legend.position = "top")`; this is to put the legend on the top

(c) Modify the code above to use different colors to represent songs of different aritists. The expected output is as follows:

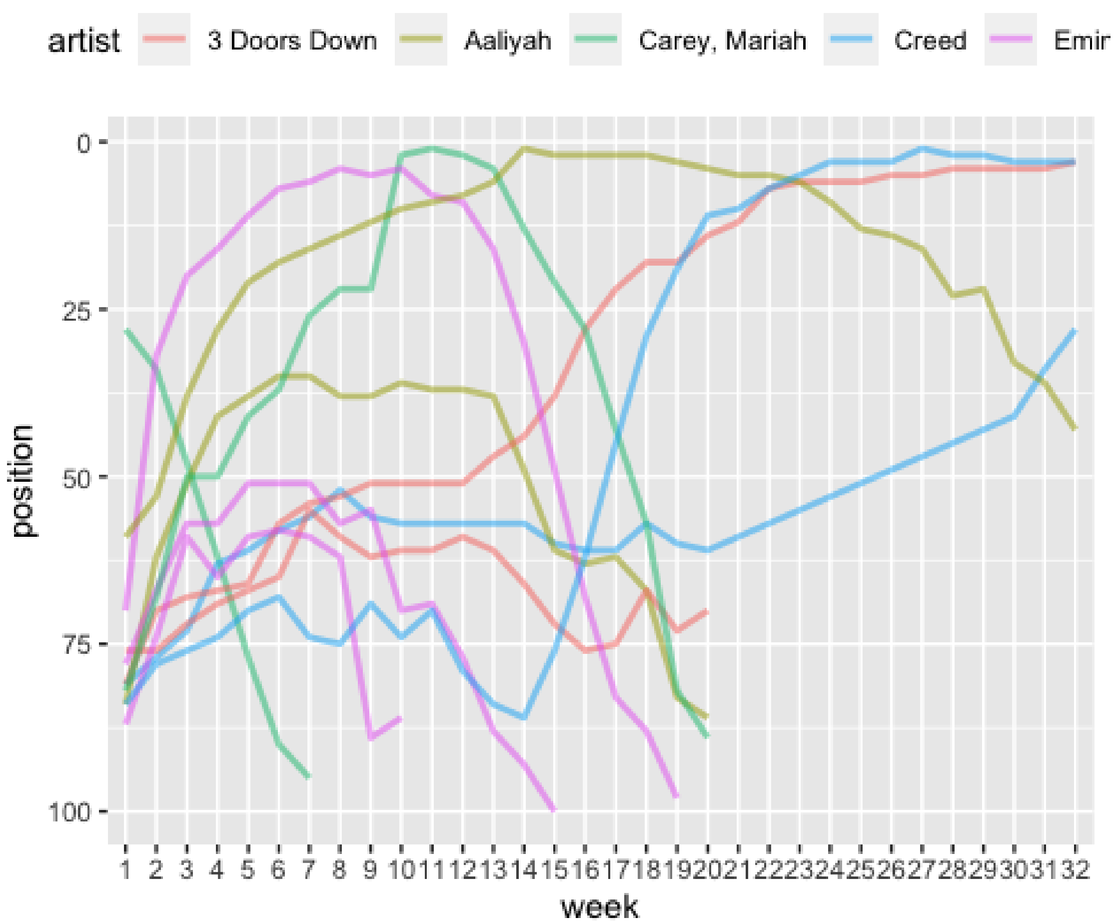


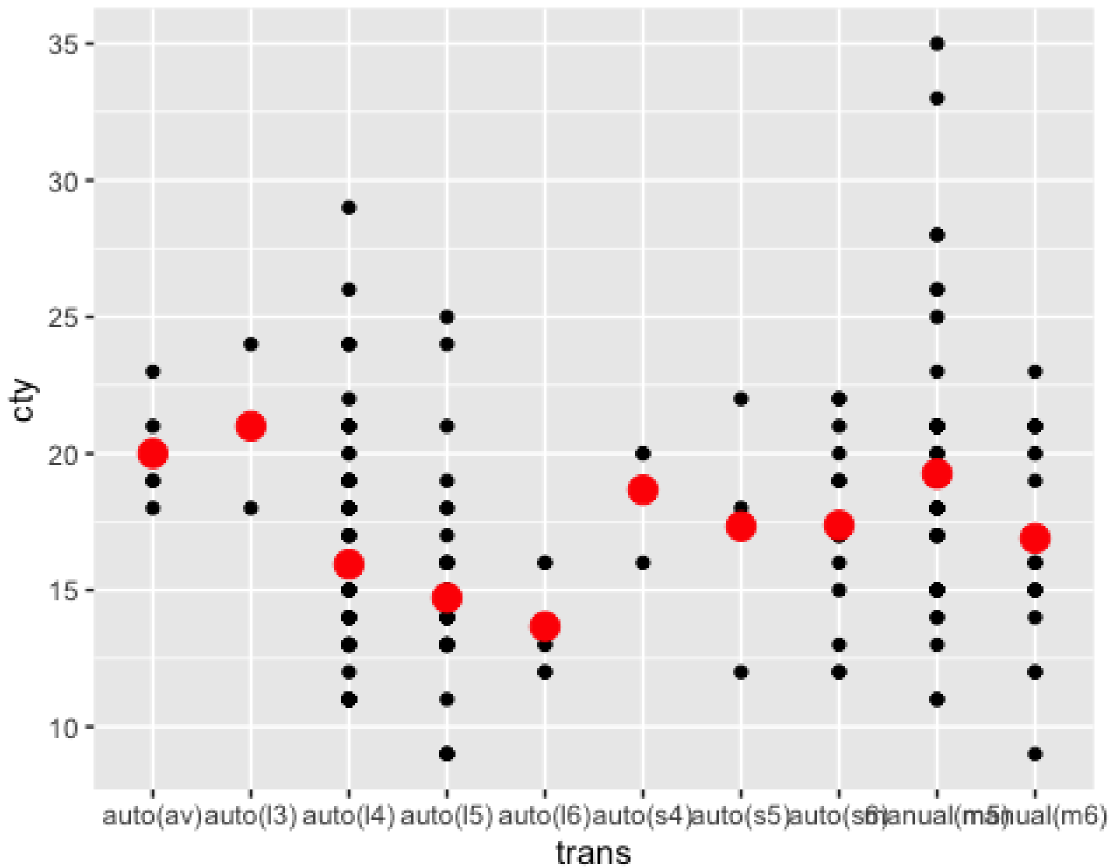
Figure 2. Task 1 (c)

[End of Task 1]

8.4 Stats

In some situations, rather than the raw data, we want to plot some **statistical transformations** of the data.

```
ggplot(mpg, aes(trans, cty)) + geom_point() + geom_point(stat = "summary",
fun = mean, colour = "red", size = 4)
```



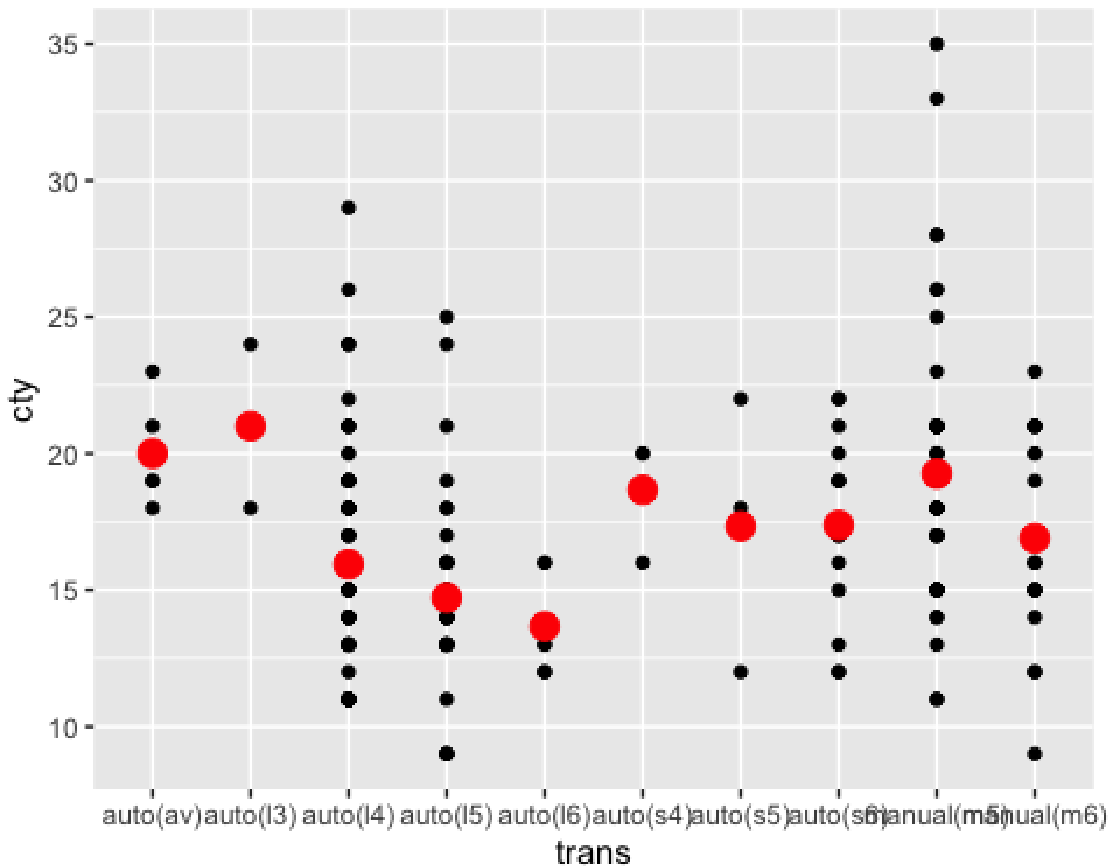
The first geom_point layer plots the raw data, whereas the second geom_point layer plots the mean of cty for each trans by setting stat = "summary" and fun = mean.

```
str(geom_point)   # the default value of `stat` is "identity", which means
the raw data

## function (mapping = NULL, data = NULL, stat = "identity", position =
"identity",
##     ..., na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)
```

The stat_*() Functions

Alternatively, we can use a distinct grammatical element stat, which provides a class of more specialized functions for displaying statistical quantities.

```
ggplot(mpg, aes(trans, cty)) + geom_point() + stat_summary(geom = "point",
fun = mean, colour = "red", size = 4)
```



Here, instead of using a `geom` function which focuses on the *visual appearance*, we use a `stat` function that draws more attention to the *statistical transformation*. It gives us the same layer of red points as the previous `geom` function.

```
str(stat_summary)
```

```
## function (mapping = NULL, data = NULL, geom = "pointrange", position =
"identity",
##     ..., fun.data = NULL, fun = NULL, fun.max = NULL, fun.min = NULL,
fun.args = list(),
##     na.rm = FALSE, orientation = NA, show.legend = NA, inherit.aes = TRUE,
##     fun.y, fun.ymin, fun.ymax)
```

The function `stat_summary()` summarises y values at unique/binned x values.

The code below shows a list of available `stat` functions starting with `stat_`.

```
stat_fun <- ls(pos = "package:ggplot2") %>% str_extract("^stat_.*")
stat_fun[!is.na(stat_fun)]
```

```
##  [1] "stat_bin"              "stat_bin_2d"           "stat_bin_hex"
##  [4] "stat_bin2d"            "stat_binhex"           "stat_boxplot"
##  [7] "stat_contour"          "stat_contour_filled"   "stat_count"
## [10] "stat_density"          "stat_density_2d"
```

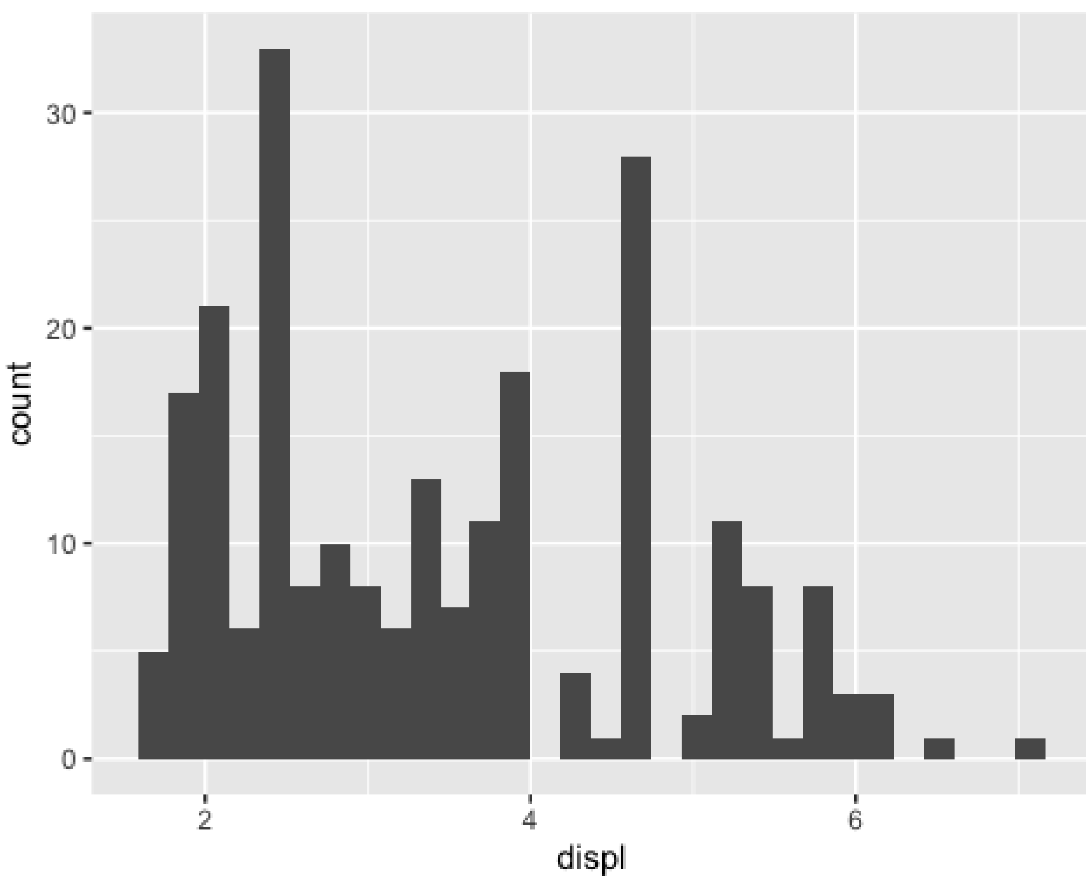
```
"stat_density_2d_filled"
## [13] "stat_density2d"          "stat_density2d_filled"  "stat_ecdf"
## [16] "stat_ellipse"            "stat_function"          "stat_identity"
## [19] "stat_qq"                 "stat_qq_line"           "stat_quantile"
## [22] "stat_sf"                 "stat_sf_coordinates"    "stat_smooth"
## [25] "stat_spoke"              "stat_sum"               "stat_summary"
## [28] "stat_summary_2d"         "stat_summary_bin"       "stat_summary_hex"
## [31] "stat_summary2d"          "stat_unique"            "stat_ydensity"
```

All `stat_*` functions (like `stat_bin`) return a layer that contains a `Stat*` object (like `StatBin`).

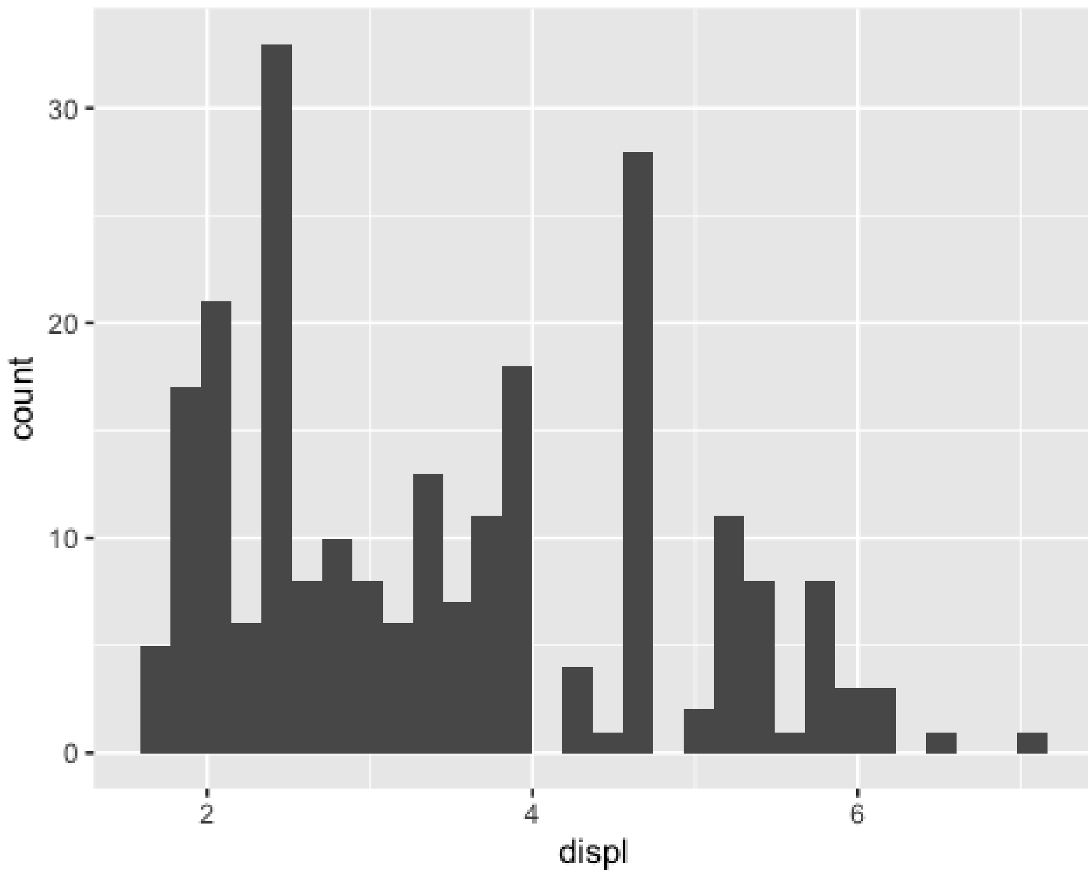
Like `geom`, `stat` is also responsible for rendering plotting layers.

We can use both of them to overlay data elements in a plot. Many `stat` functions have equivalent `geom` functions.

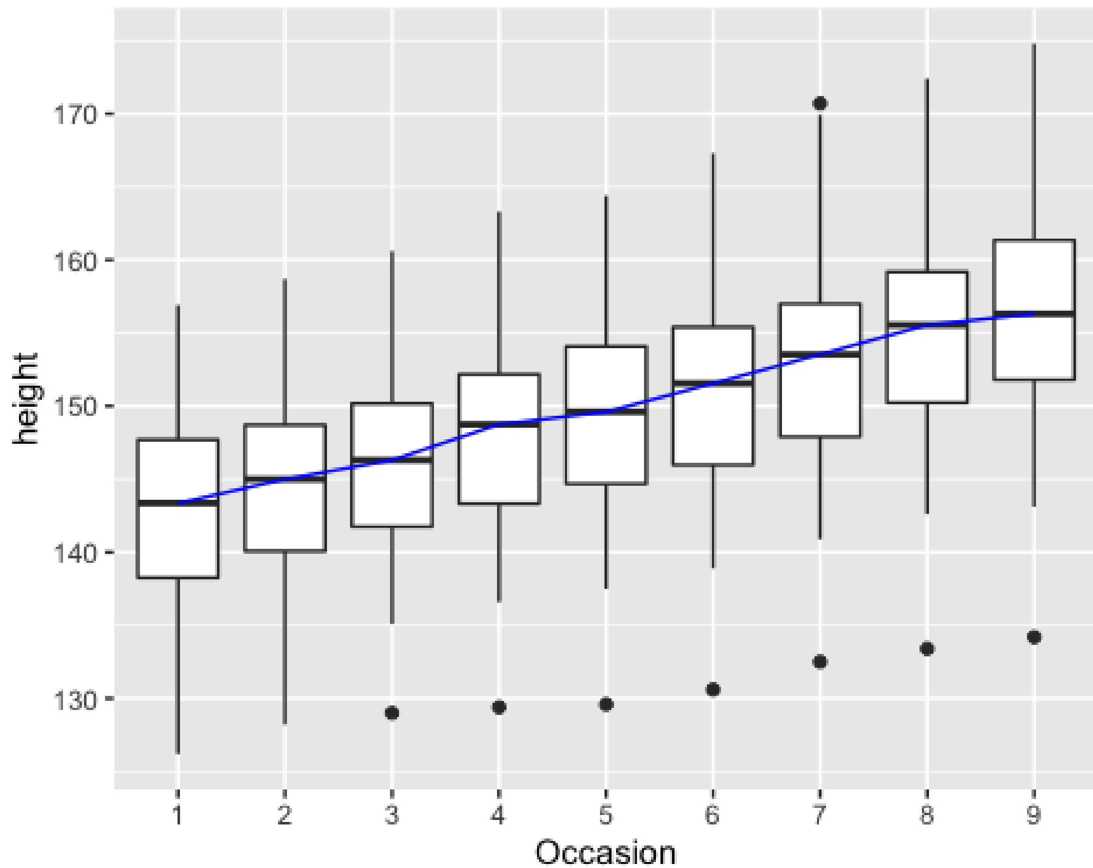
```
ggplot(mpg, aes(displ)) + geom_histogram()

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



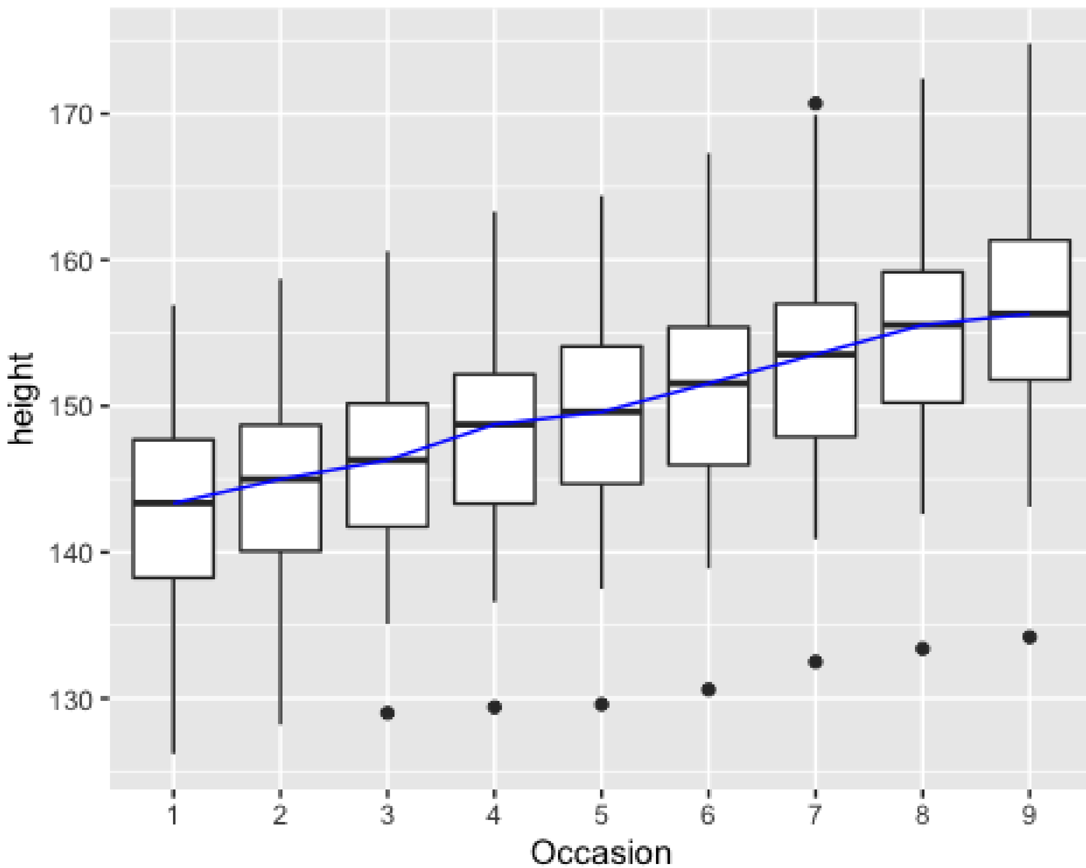

```
ggplot(mpg, aes(displ)) + stat_bin()

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
# h <- ggplot(nlme::Oxboys, aes(Occasion, height))
h + geom_boxplot() + geom_line(group = 1, stat = "summary", colour = "blue",
fun = median)
```



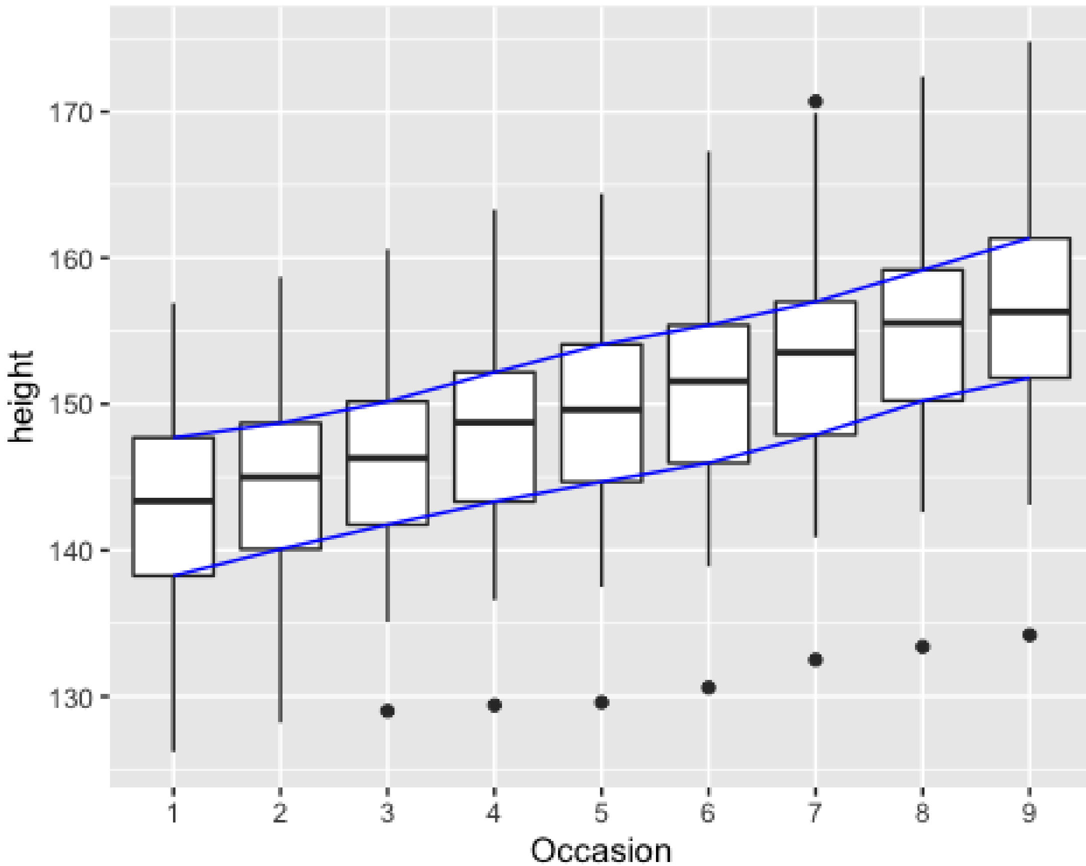

```
h + geom_boxplot() + stat_summary(group = 1, geom = "line", colour = "blue",
fun = median)
```



In the above code, we set `group = 1` to ungroup the default grouping (group = Occasion).

We can add multiple layers of `stat` to a plot:

```
h + geom_boxplot() + stat_summary(group = 1, geom = "line", colour = "blue",
fun = quantile, fun.args = list(probs = 0.25)) + stat_summary(group = 1, geom
= "line", colour = "blue", fun = quantile, fun.args = list(probs = 0.75))
```



geom functions have default stat, and stat functions also have default geom.

```
str(stat_bin)        # the default `geom` is "bar"
```

```
## function (mapping = NULL, data = NULL, geom = "bar", position = "stack",
##     ..., binwidth = NULL, bins = NULL, center = NULL, boundary = NULL,
##     breaks = NULL, closed = c("right", "left"), pad = FALSE, na.rm =
FALSE,
##     orientation = NA, show.legend = NA, inherit.aes = TRUE)
```

```
str(geom_histogram)  # the default `stat` is "bin"
```

```
## function (mapping = NULL, data = NULL, stat = "bin", position = "stack",
##     ..., binwidth = NULL, bins = NULL, na.rm = FALSE, orientation = NA,
##     show.legend = NA, inherit.aes = TRUE)
```

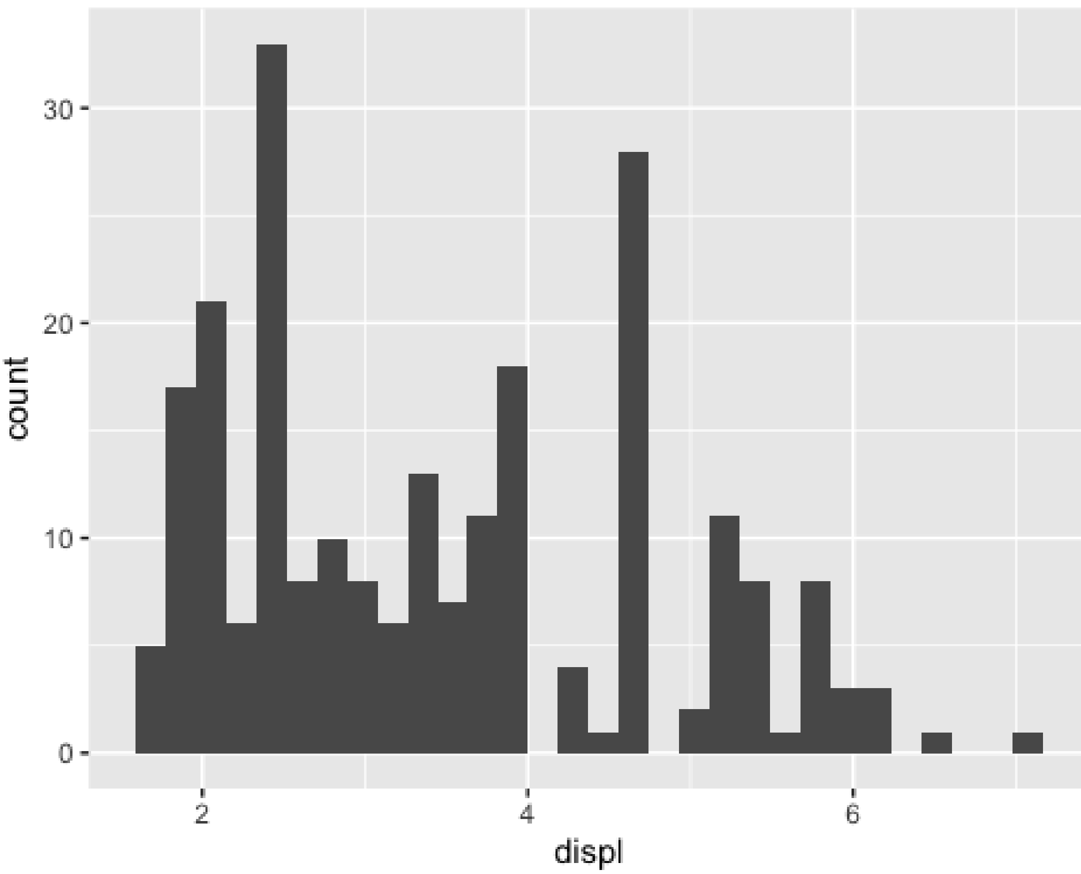
Change Plotted Variables

More than one variable can be generated by a geom or stat.

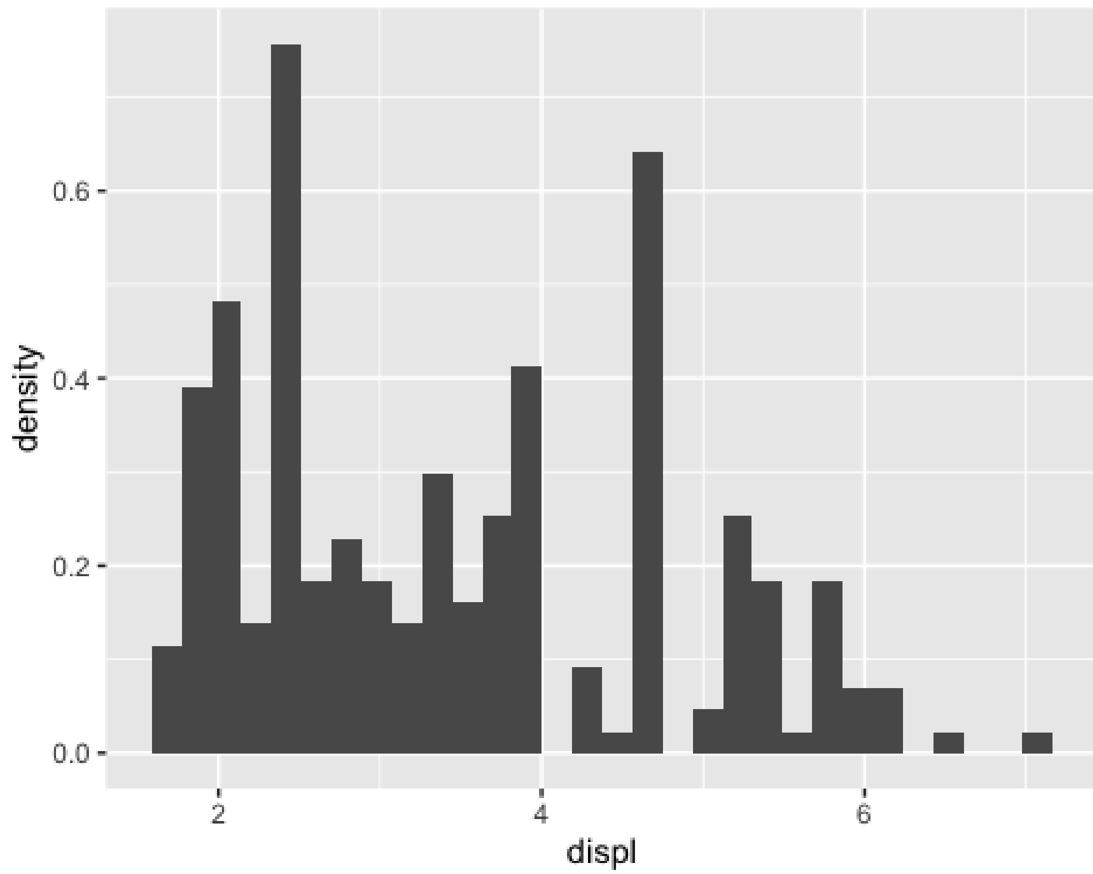
Use aes() to change the default mapping with the variable name surrounded by ..:

```
ggplot(mpg, aes(displ)) + geom_histogram()   # y axis defaults to count
```

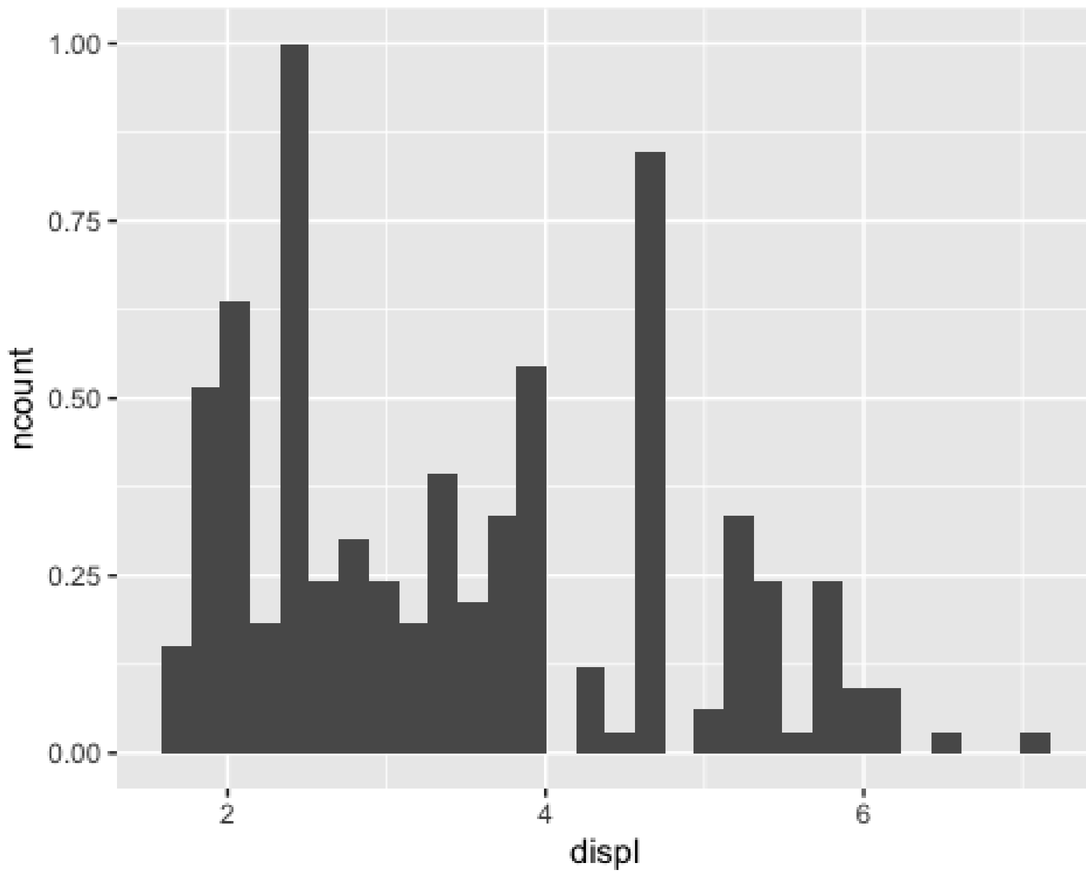
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
ggplot(mpg, aes(displ)) + geom_histogram(aes(y = ..density..))  # density of
points in bin

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
ggplot(mpg, aes(displ)) + geom_histogram(aes(y = ..ncount..))   # count, scaled to maximum of 1

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



To check the available variables computed by the `geom` or `stat`, go the help document. For example, `help(geom_histogram)`, refer to the section "Computed variables".

We use a pair of double dots to surround the variable name to avoid confusion (in case the dataset contains a variable with the same name).

[Task 2: Plotting Billboard Ranking, Continued]

(a) Use `stat_summary()` to display the best weekly records for each artist; that is, the highest position her/his songs have reached for each week after release. The expected output is as follows:

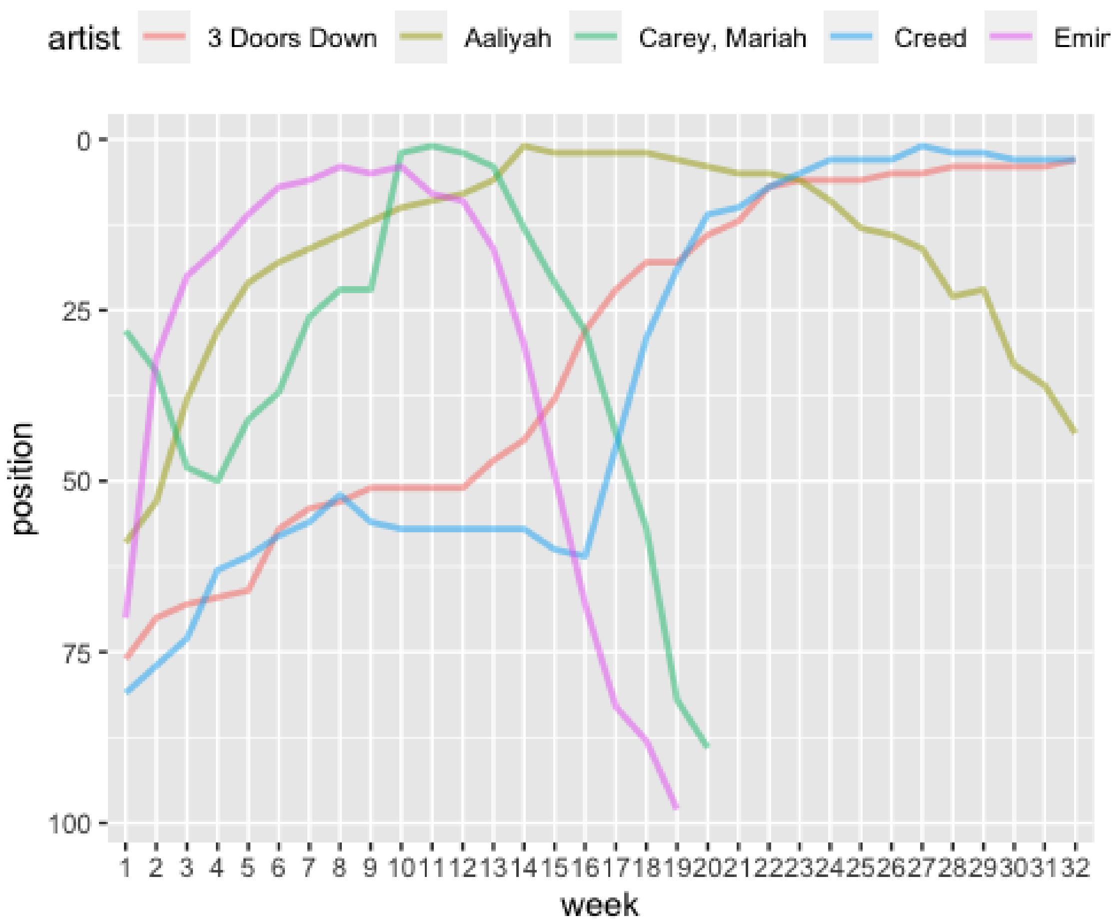


Figure 3. Task 2 (a)

Tips: `fun = max` for `stat_summary()`

(b) Use the `billboard_long` tibble to create a box plot to show the distribution of weekly ranking positions of all songs, and overlay the best weekly records for each of the 5 artists on it. The expected output is as follows:

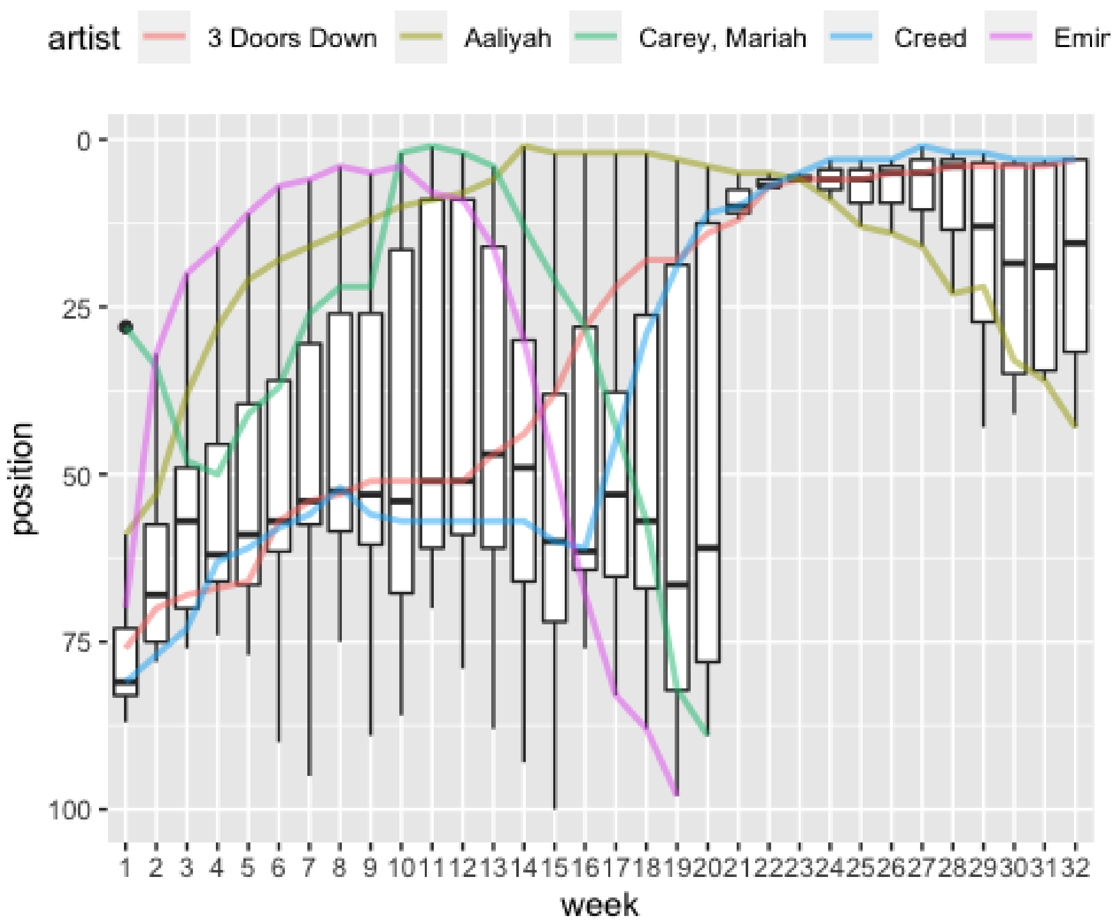


Figure 4. Task 2 (b)

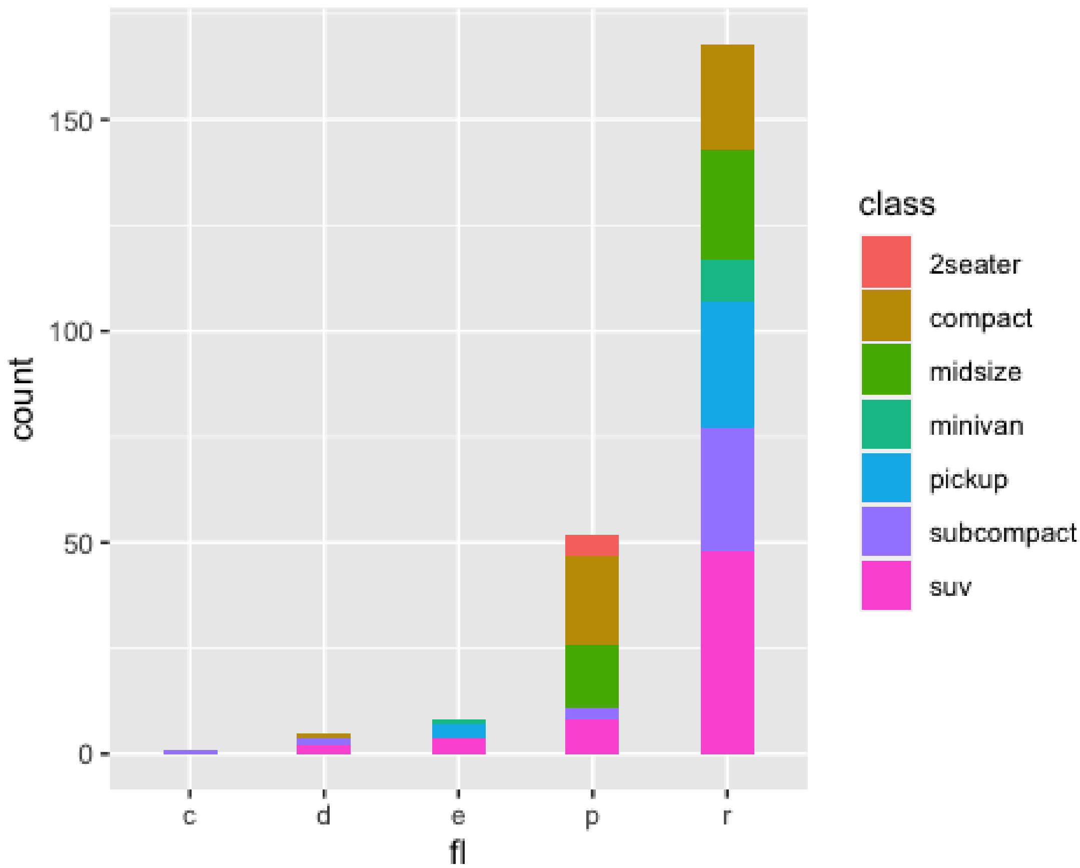
[End of Task 2]

8.5 Positions

Position Adjustment

All layers have a *position adjustment* that resolves *overlapping geoms* within a layer.

```
ggplot(mpg, aes(fl, fill = class)) + geom_bar(width = 0.4)
```

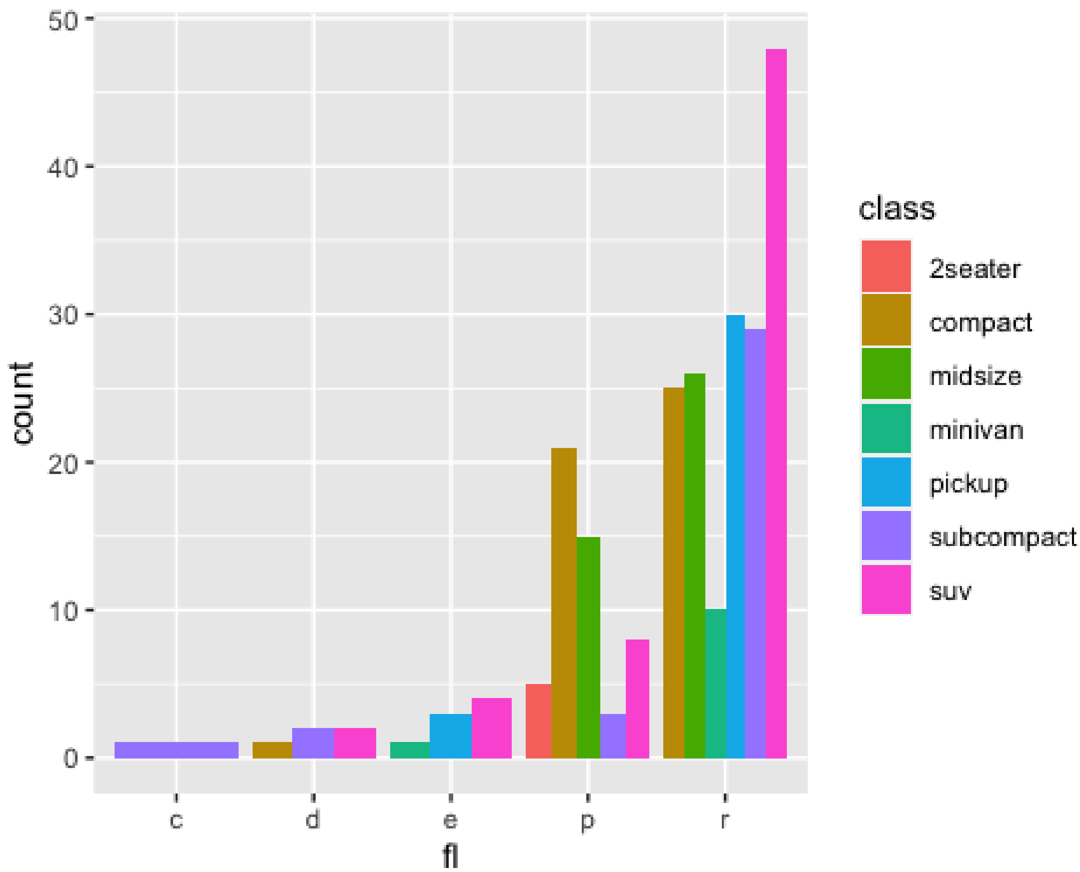



```
str(geom_bar)   # default position is "stack"

## function (mapping = NULL, data = NULL, stat = "count", position = "stack",
##     ..., width = NULL, na.rm = FALSE, orientation = NA, show.legend = NA,
##     inherit.aes = TRUE)
```

In the above bar plot, within each level of `fl`, the obesrvations are further partitioned by `class`, and a bar is created for each partition and colored differently. The default way to position these bars is to "stack" them.

We can override the default by setting the `position` argument of the `geom` or `stat` function:

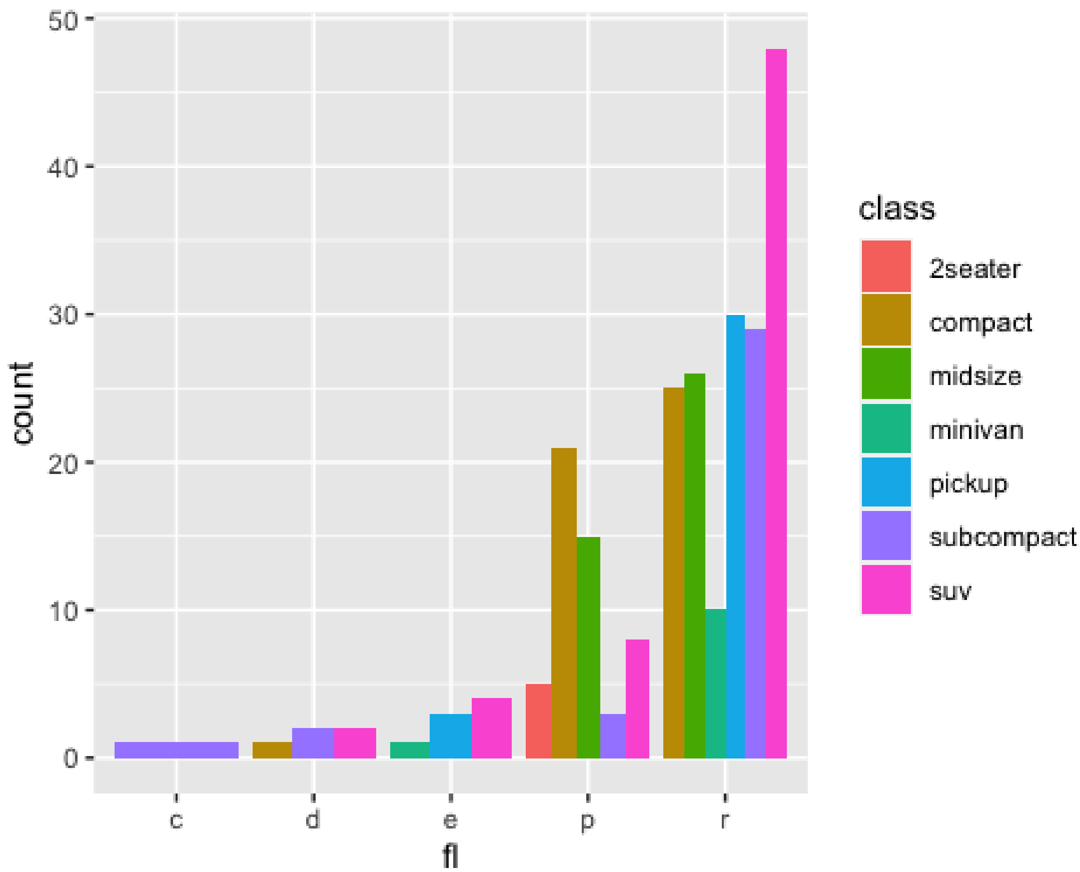
```
ggplot(mpg, aes(fl, fill = class)) + geom_bar(position = "dodge")
```



The `position_*()` Functions

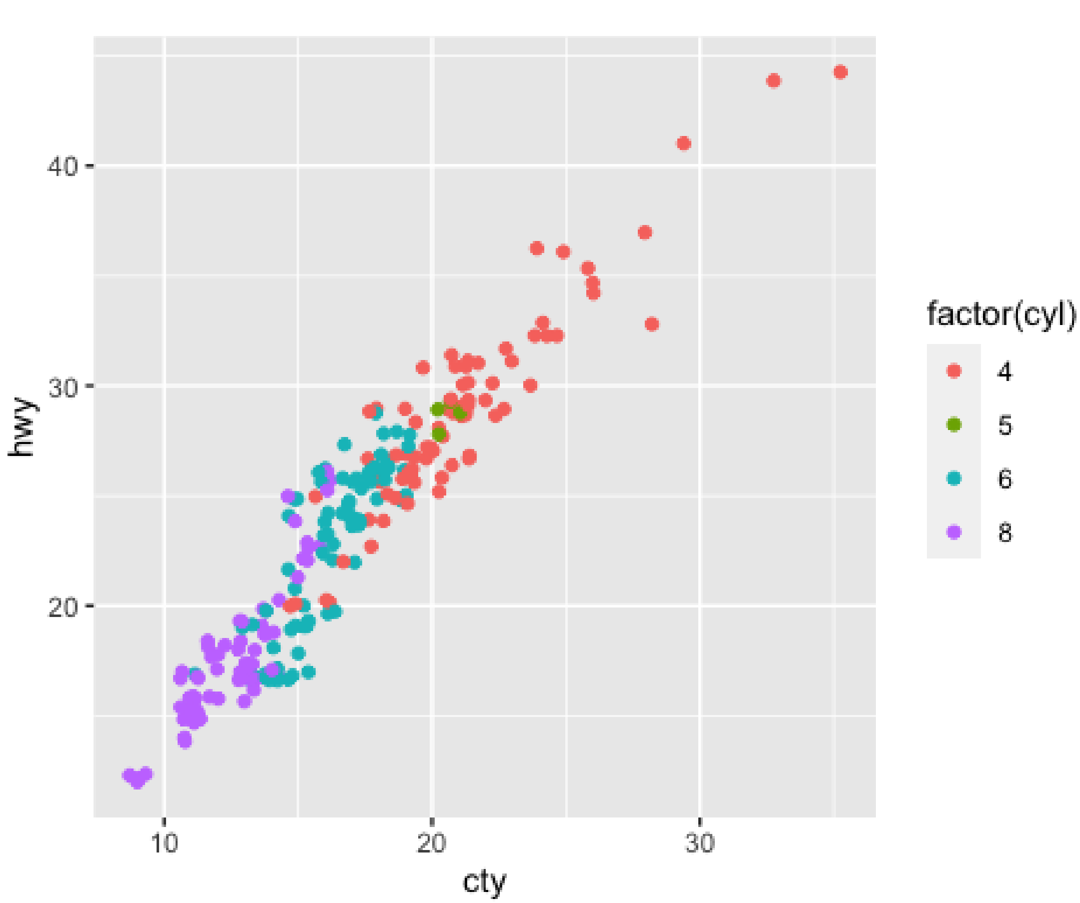
Alternatively, we can use a distinct grammatical element `position`:

```
ggplot(mpg, aes(fl, fill = class)) + geom_bar(position = position_dodge())
```

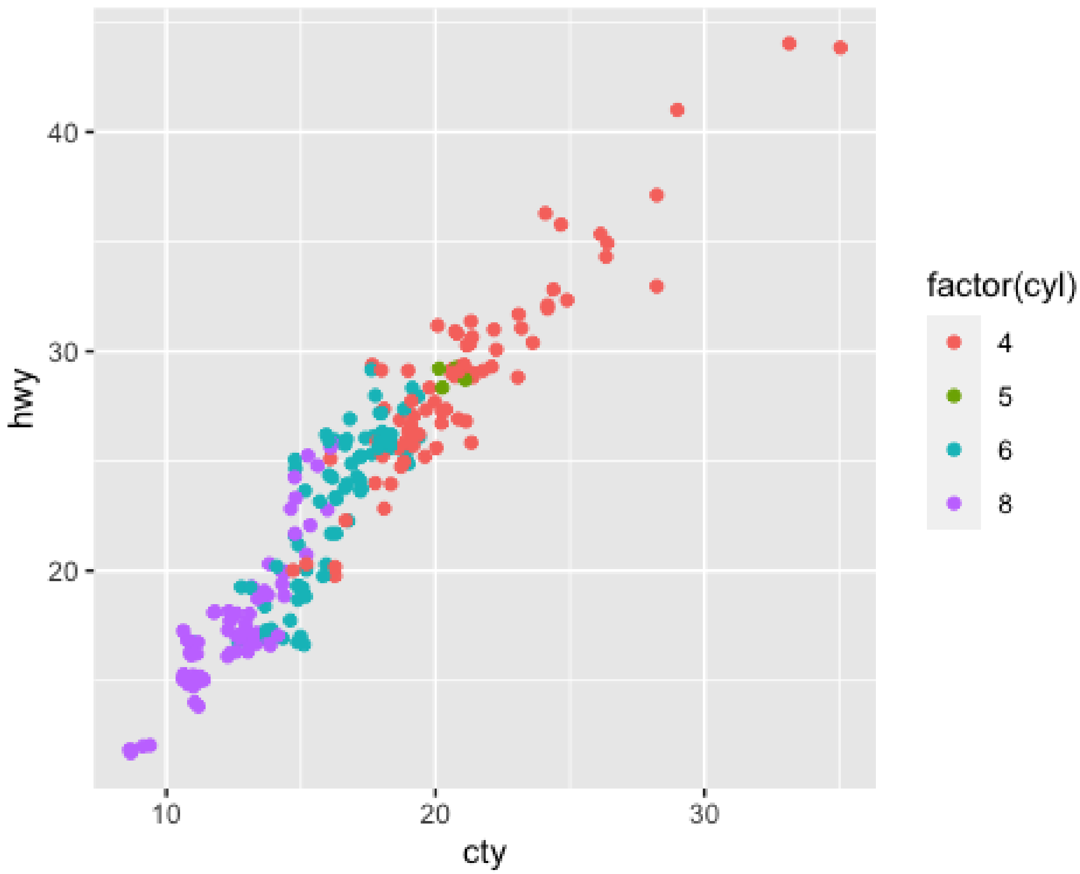


- The `position` functions that apply primarily to *bars*:
  - `position_dodge()`, `position_dodge2()`: Dodge overlapping objects side-to-side.
  - `position_stack()`, `position_fill()`: Stack overlapping objects on top of each another.
- The `position` functions primarily useful for *points*:
  - `position_jitter()`: Jitter points to avoid overplotting (add random noise to the location of each point).
  - `position_jitterdodge()`: Simultaneously dodge and jitter.
  - `position_nudge()`: Nudge points a fixed distance.

```
ggplot(mpg, aes(cty, hwy)) + geom_point(aes(colour = factor(cyl)), position = "jitter")
```

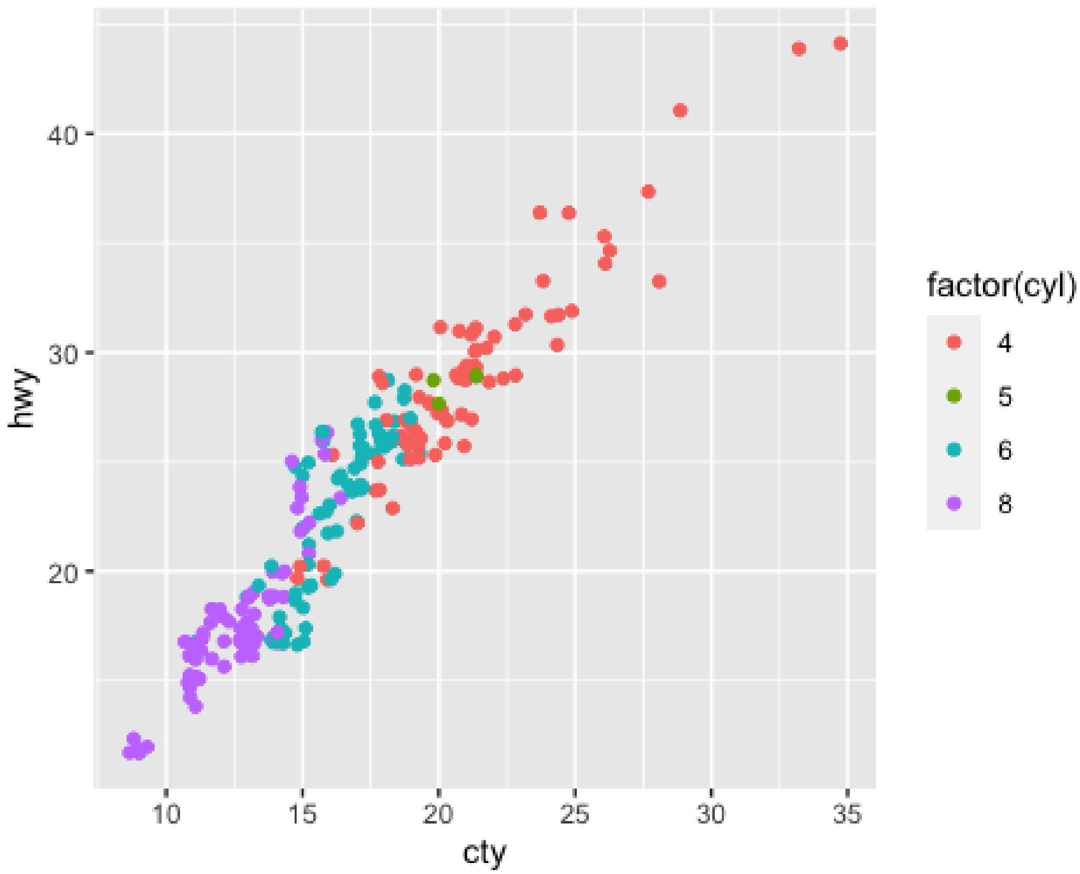



```
ggplot(mpg, aes(cty, hwy)) + geom_point(aes(colour = factor(cyl)), position =
position_jitter())
```



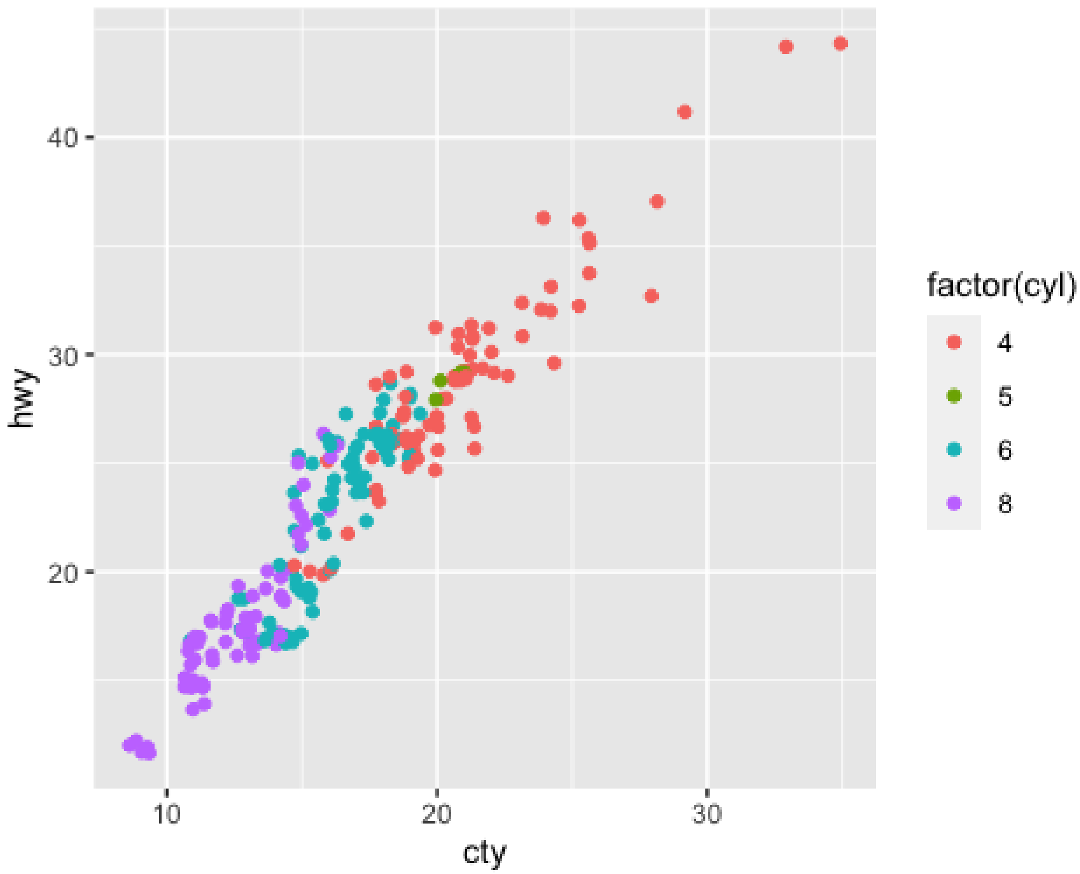
We can set the parameters of `position_jitter()` to control the jitter. For example, set `seed` to make the jitter reproducible:

```
ggplot(mpg, aes(cty, hwy)) + geom_point(aes(colour = factor(cyl)), position =
position_jitter(seed = 1))
```



geom_jitter is a convenient shortcut for geom_point(position = "jitter"):

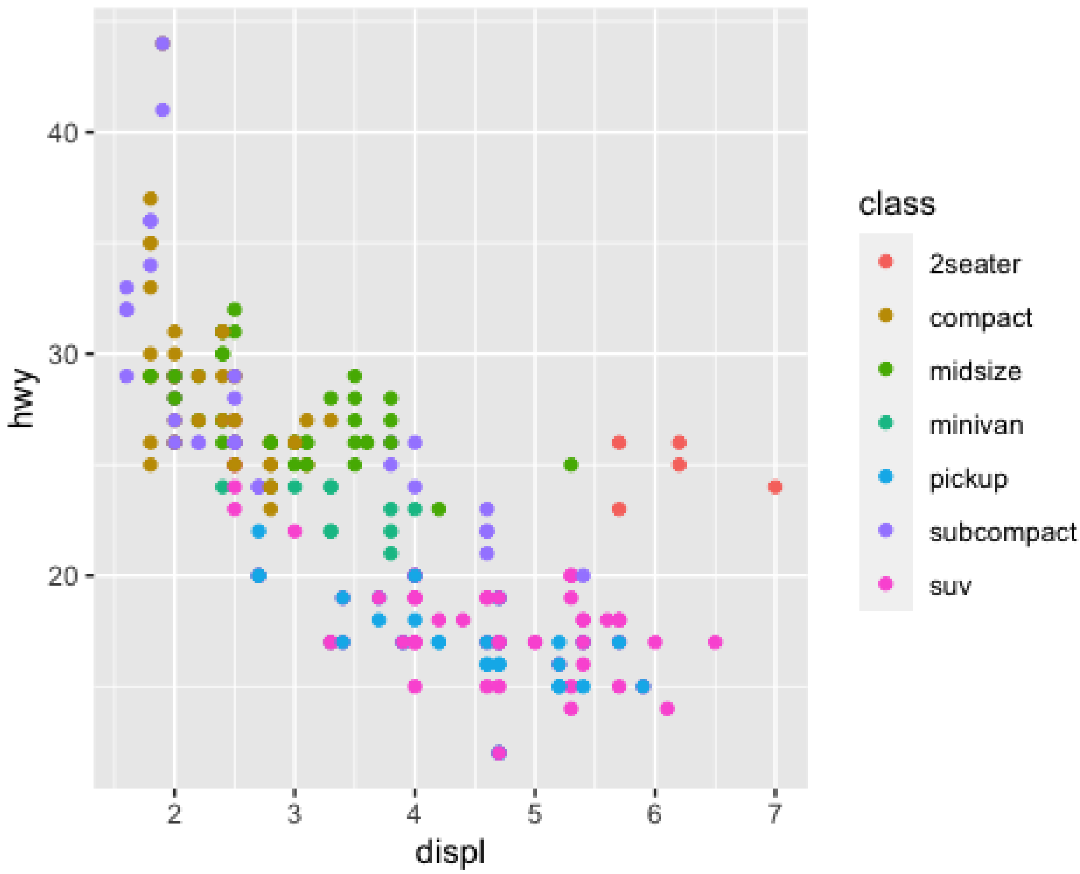
```
ggplot(mpg, aes(cty, hwy)) + geom_jitter(aes(colour = factor(cyl)))
```



8.6 Scales

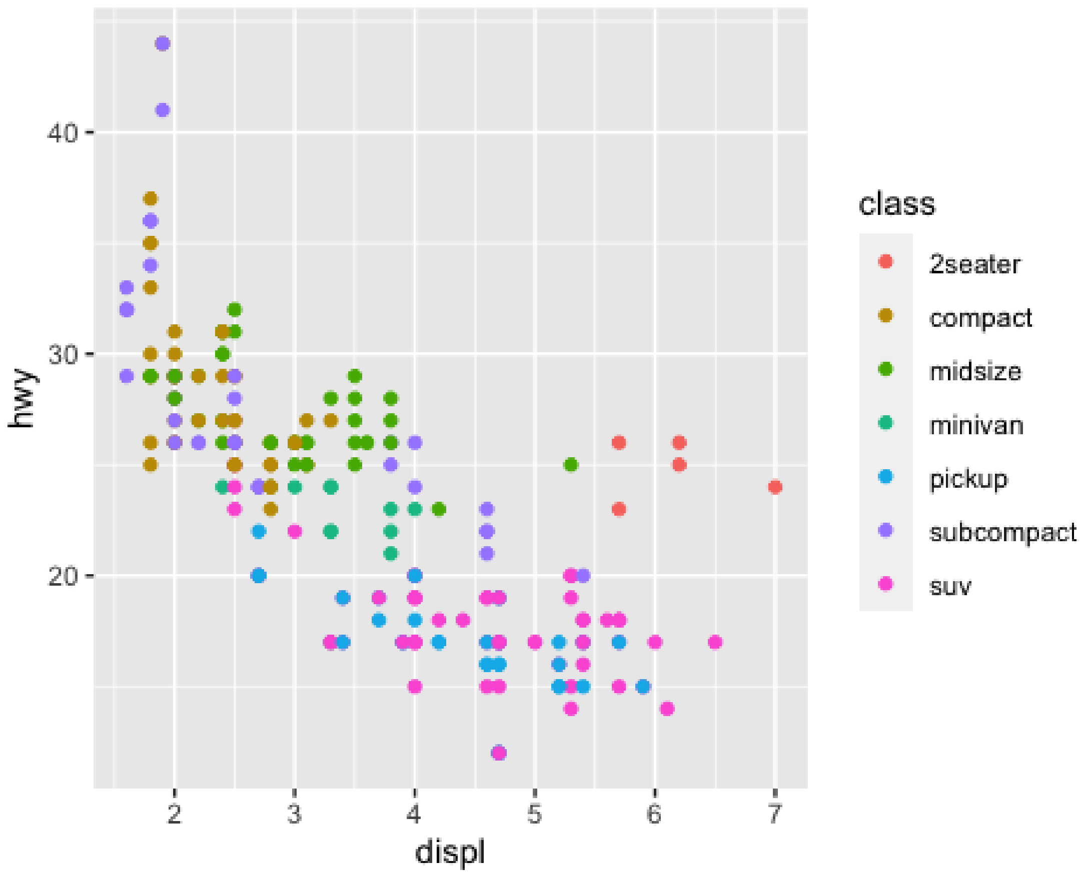
Scales control the mapping from data to aesthetics and provide the tools (i.e., the axes and legends) that allow us to read the plot:

```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = class))
```



What actually happens behind the scence is:

```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = class)) +
scale_x_continuous() + scale_y_continuous() + scale_colour_hue()
```

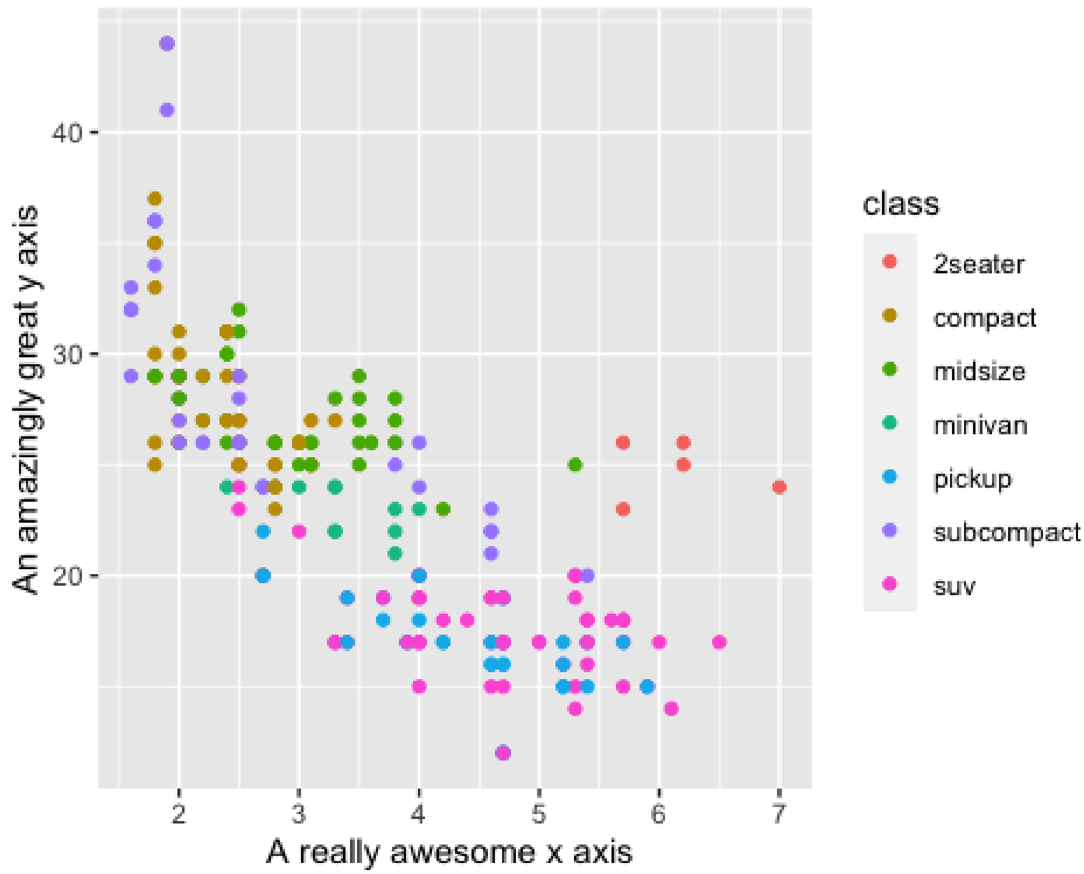


The default `scales` are generated automatically by R.

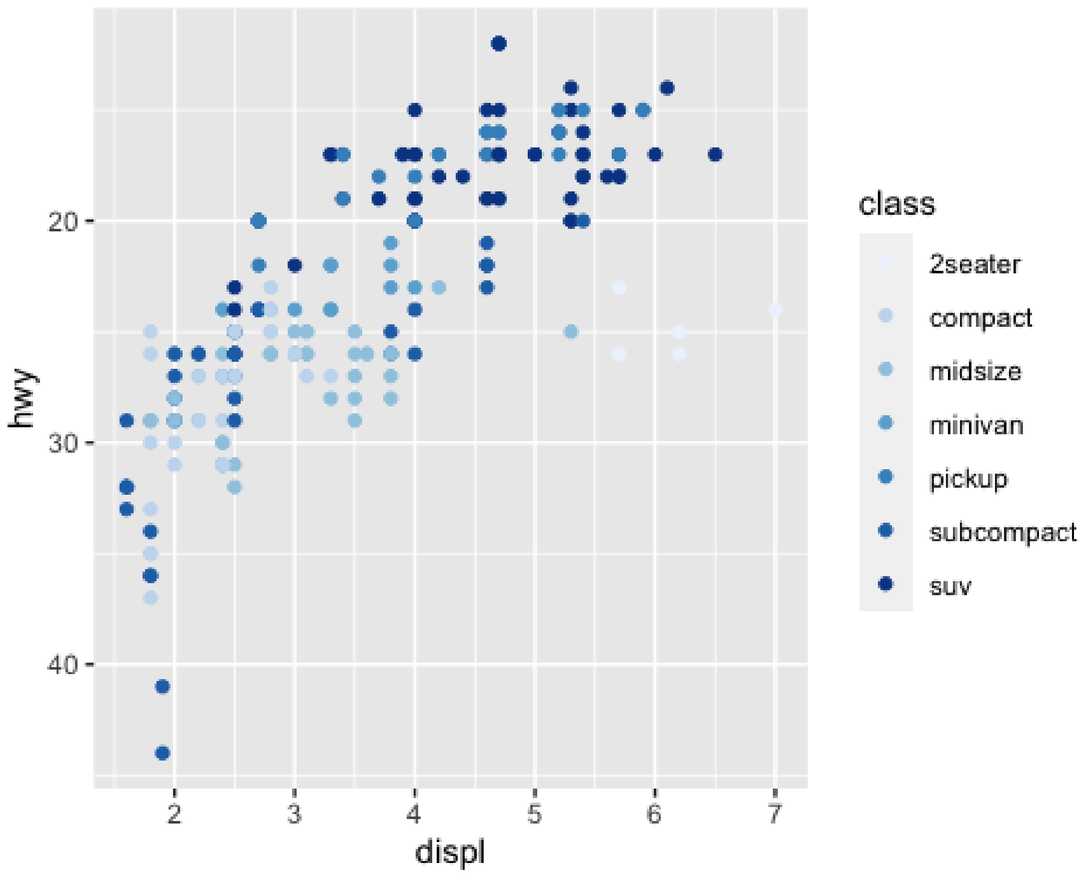
The naming convention for scales: "scale_aestheticName_scaleName", e.g., `scale_y_continuous()`, `scale_colour_hue()`, etc.

We can explicitly call a `scale_*()` function to override the defaults:

```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = class)) +
scale_x_continuous("A really awesome x axis ") + scale_y_continuous("An
amazingly great y axis")
```




```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = class)) +
scale_y_reverse() + scale_colour_brewer()
```



The `scale_*()` Functions

The following code gives a list of all `scale` functions we can use:

```
scale_fun <- ls(pos = "package:ggplot2") %>% str_extract("^scale_.*")
scale_fun[!is.na(scale_fun)]
```

```
##   [1] "scale_alpha"                "scale_alpha_binned"
##   [3] "scale_alpha_continuous"     "scale_alpha_date"
##   [5] "scale_alpha_datetime"       "scale_alpha_discrete"
##   [7] "scale_alpha_identity"       "scale_alpha_manual"
##   [9] "scale_alpha_ordinal"        "scale_color_binned"
##  [11] "scale_color_brewer"         "scale_color_continuous"
##  [13] "scale_color_date"           "scale_color_datetime"
##  [15] "scale_color_discrete"       "scale_color_distiller"
##  [17] "scale_color_fermenter"      "scale_color_gradient"
##  [19] "scale_color_gradient2"      "scale_color_gradientn"
##  [21] "scale_color_grey"           "scale_color_hue"
##  [23] "scale_color_identity"       "scale_color_manual"
##  [25] "scale_color_ordinal"        "scale_color_steps"
##  [27] "scale_color_steps2"         "scale_color_stepsn"
##  [29] "scale_color_viridis_c"      "scale_color_viridis_d"
##  [31] "scale_colour_binned"        "scale_colour_brewer"
##  [33] "scale_colour_continuous"    "scale_colour_date"
```

```
##  [35] "scale_colour_datetime"     "scale_colour_discrete"
##  [37] "scale_colour_distiller"    "scale_colour_fermenter"
##  [39] "scale_colour_gradient"     "scale_colour_gradient2"
##  [41] "scale_colour_gradientn"    "scale_colour_grey"
##  [43] "scale_colour_hue"          "scale_colour_identity"
##  [45] "scale_colour_manual"       "scale_colour_ordinal"
##  [47] "scale_colour_steps"        "scale_colour_steps2"
##  [49] "scale_colour_stepsn"       "scale_colour_viridis_b"
##  [51] "scale_colour_viridis_c"    "scale_colour_viridis_d"
##  [53] "scale_continuous_identity" "scale_discrete_identity"
##  [55] "scale_discrete_manual"     "scale_fill_binned"
##  [57] "scale_fill_brewer"         "scale_fill_continuous"
##  [59] "scale_fill_date"           "scale_fill_datetime"
##  [61] "scale_fill_discrete"       "scale_fill_distiller"
##  [63] "scale_fill_fermenter"      "scale_fill_gradient"
##  [65] "scale_fill_gradient2"      "scale_fill_gradientn"
##  [67] "scale_fill_grey"           "scale_fill_hue"
##  [69] "scale_fill_identity"       "scale_fill_manual"
##  [71] "scale_fill_ordinal"        "scale_fill_steps"
##  [73] "scale_fill_steps2"         "scale_fill_stepsn"
##  [75] "scale_fill_viridis_b"      "scale_fill_viridis_c"
##  [77] "scale_fill_viridis_d"      "scale_linetype"
##  [79] "scale_linetype_binned"     "scale_linetype_continuous"
##  [81] "scale_linetype_discrete"   "scale_linetype_identity"
##  [83] "scale_linetype_manual"     "scale_radius"
##  [85] "scale_shape"               "scale_shape_binned"
##  [87] "scale_shape_continuous"    "scale_shape_discrete"
##  [89] "scale_shape_identity"      "scale_shape_manual"
##  [91] "scale_shape_ordinal"       "scale_size"
##  [93] "scale_size_area"           "scale_size_binned"
##  [95] "scale_size_binned_area"    "scale_size_continuous"
##  [97] "scale_size_date"           "scale_size_datetime"
##  [99] "scale_size_discrete"       "scale_size_identity"
## [101] "scale_size_manual"         "scale_size_ordinal"
## [103] "scale_type"                "scale_x_binned"
## [105] "scale_x_continuous"        "scale_x_date"
## [107] "scale_x_datetime"          "scale_x_discrete"
## [109] "scale_x_log10"             "scale_x_reverse"
## [111] "scale_x_sqrt"              "scale_x_time"
## [113] "scale_y_binned"            "scale_y_continuous"
## [115] "scale_y_date"              "scale_y_datetime"
## [117] "scale_y_discrete"          "scale_y_log10"
## [119] "scale_y_reverse"           "scale_y_sqrt"
## [121] "scale_y_time"
```

The component of the scale that we're most likely to modify is the *axis* or *legend* associated with a scale, collectively referred to as the *guide*.

In `ggplot2`, guides are generated automatically based on the layers in the plot, which is different from base R graphics where we need to draw the legend by hand.

Axes calibrate the scales of the position aesthetics, while legends calibrate the scales of non-position aesthetics.

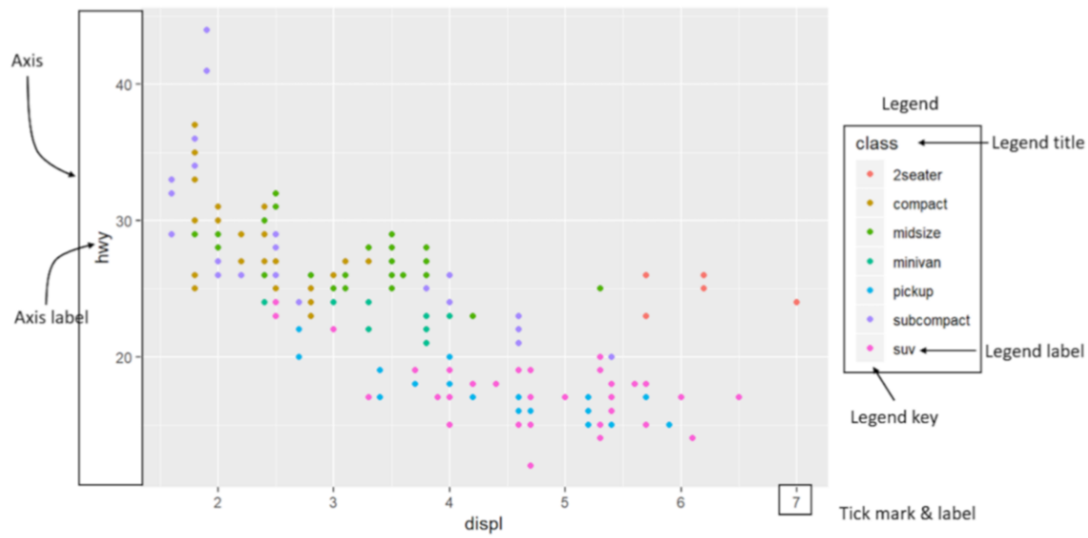
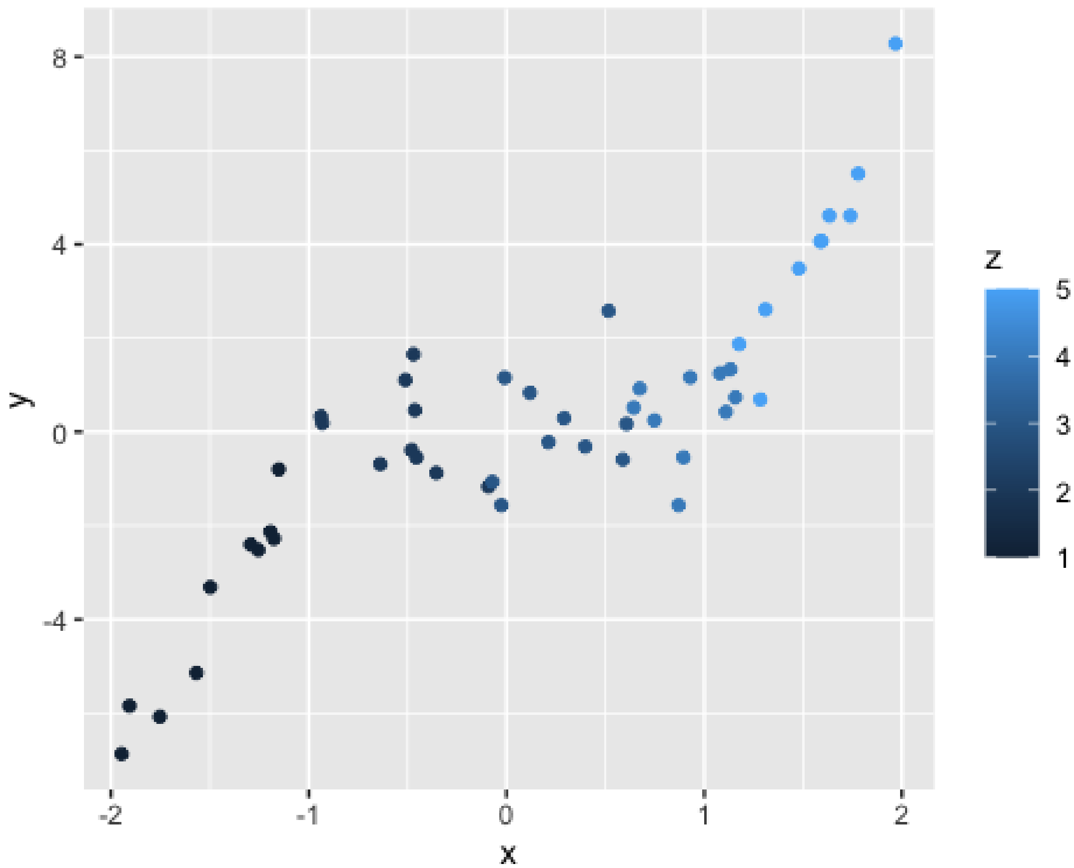


Fig. 5 Axis and Legend of a Scale

name: Axis Label and Legend Title

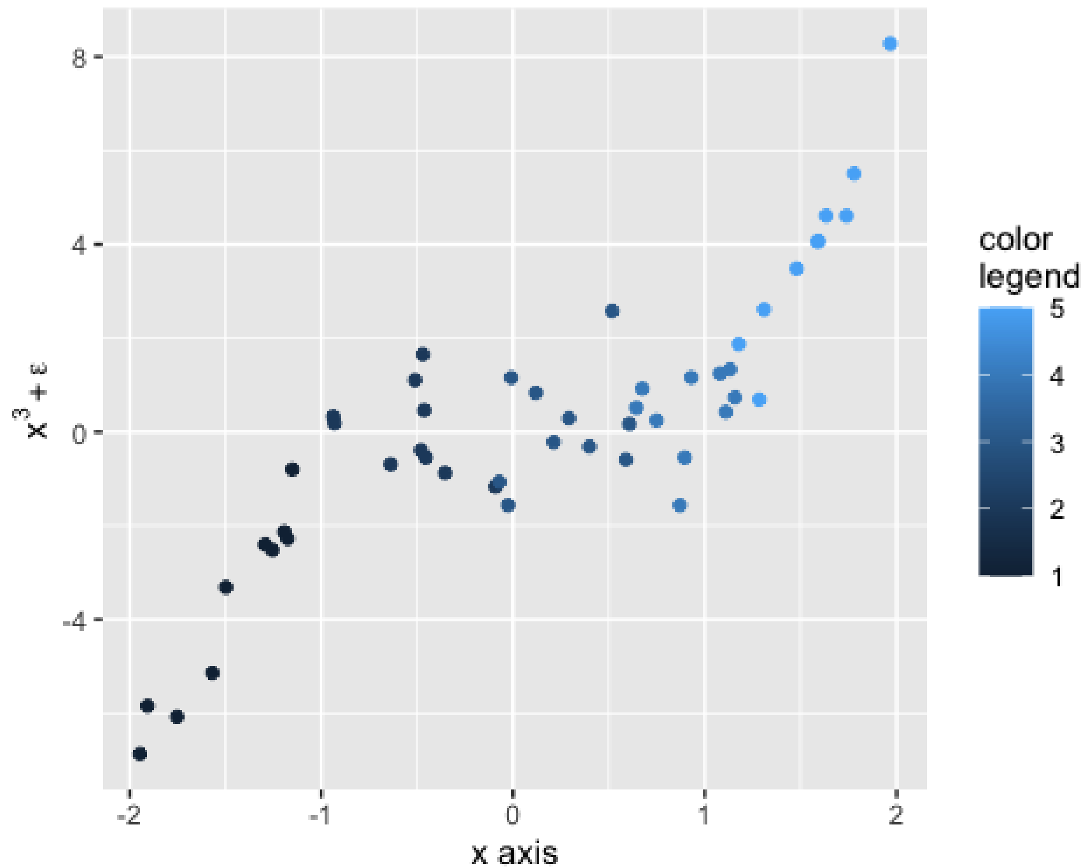
```
set.seed(0)
df <- tibble(x = sort(runif(50, min = -2, max = 2)), y = x^3 + rnorm(50), z =
rep(1:5, times = rep(10, times = 5)))
p1 <- ggplot(df, aes(x, y, colour = z))
p1 + geom_point()
```




```
# equivalent to:
# p1 + geom_point() + scale_x_continuous() + scale_y_continuous() +
scale_color_continuous()
```

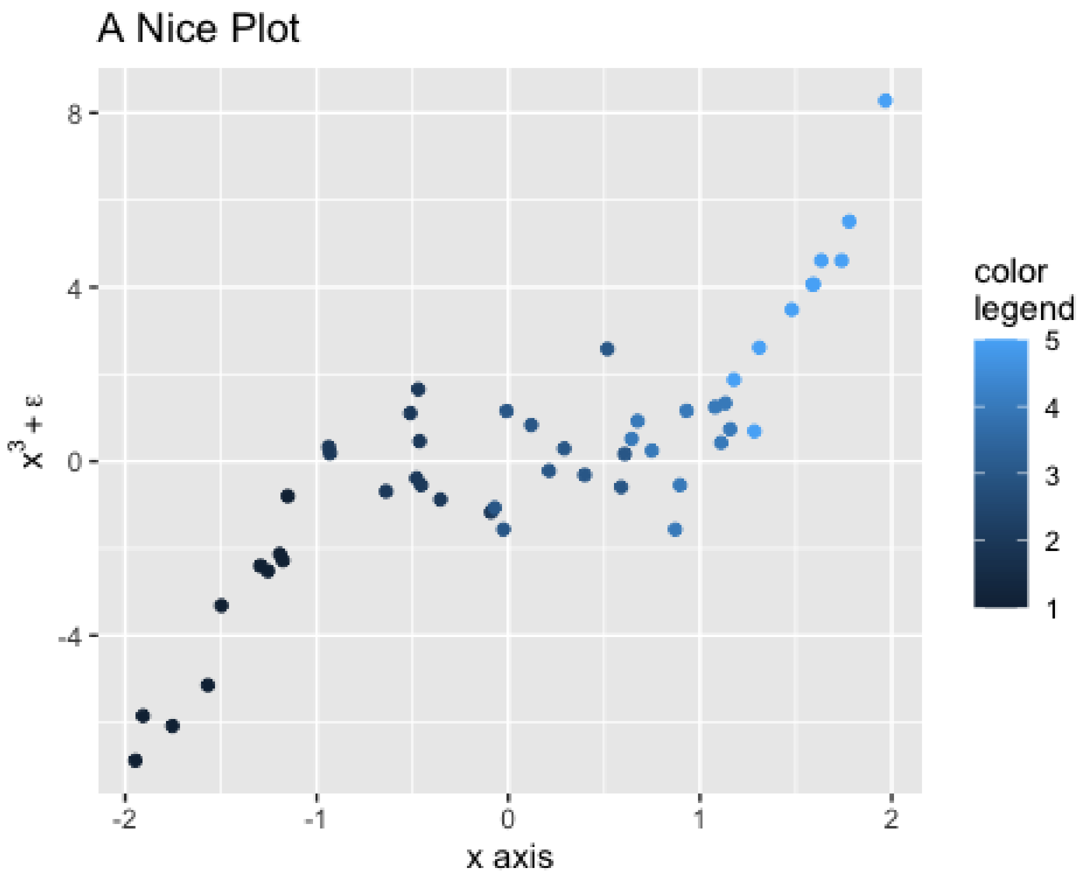
The first argument to the `scale_*()` function, `name`, controls the axis label or legend title, which can be text strings, mathematical expressions enclosed by quotes.

```
p1 + geom_point() + scale_x_continuous("x axis") +
scale_y_continuous(quote(x^3 + epsilon)) +
scale_color_continuous("color\nlegend")   # `\n` indicates a line break
```



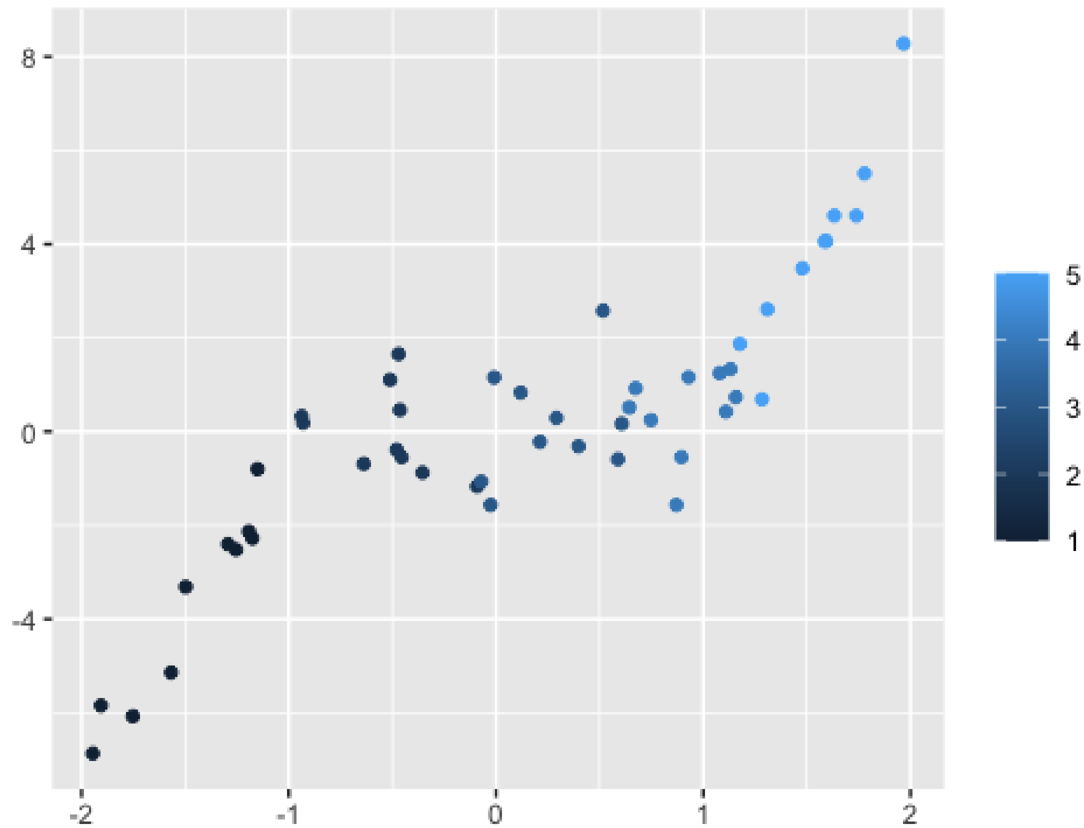
Because modifying these labels is such a common task, `ggplot2` provides three helper functions: `xlab()`, `ylab()` and `labs()`:

```
p1 + geom_point() + labs(x = "x axis", y = quote(x^3 + epsilon), title = "A
Nice Plot", color = "color\nlegend")
```



We can remove the axis label and legend title by setting it to `""` or `NULL`:

```
p1 + geom_point() + labs(x = "", y = NULL, colour = NULL)
```

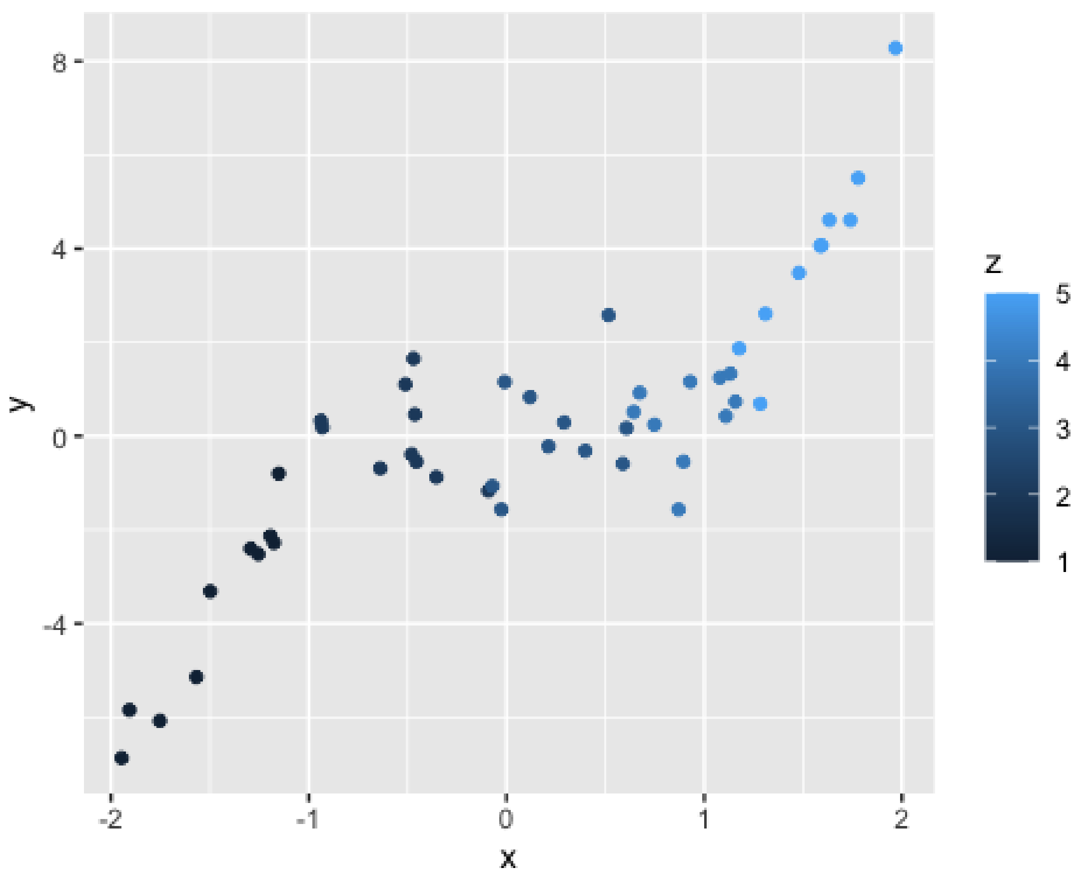


`breaks` and `labels`

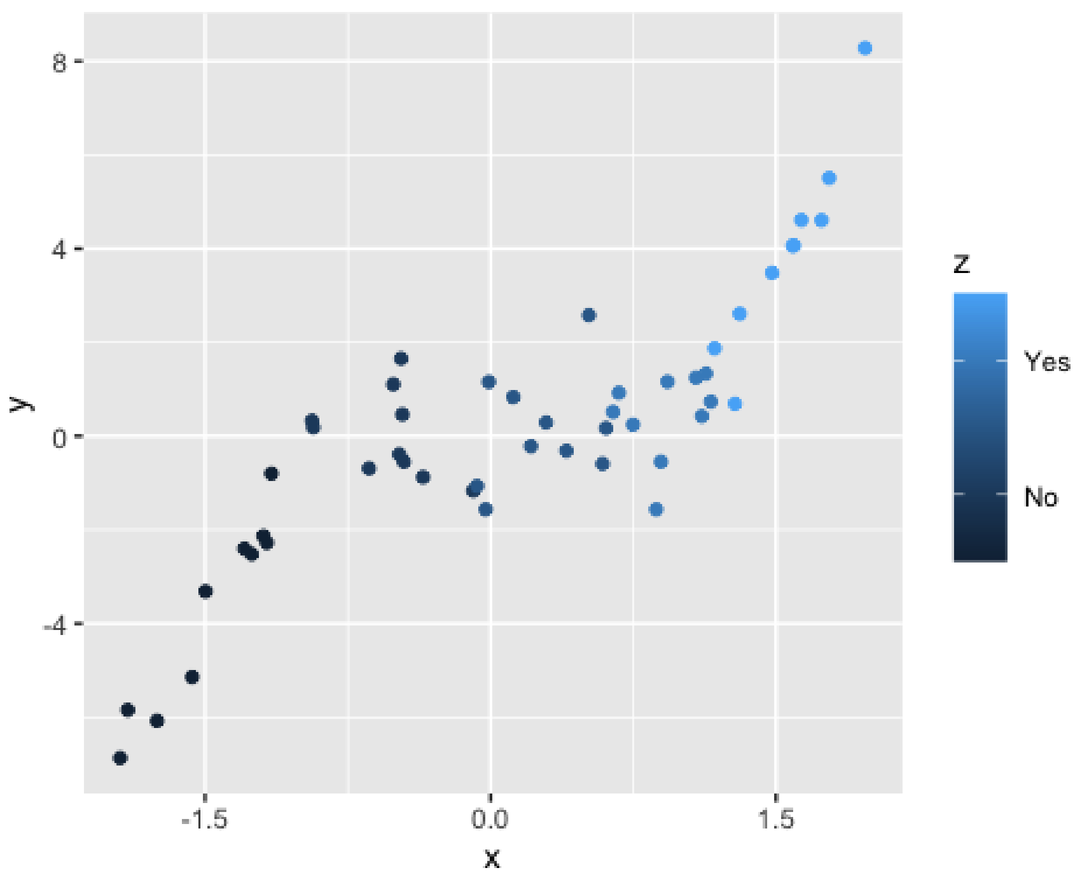
The `breaks` argument controls *which values appear as tick marks* on axes and keys on legends.

Each break has an associated label, controlled by the `labels` argument. If we set `labels`, we must also set `breaks`.

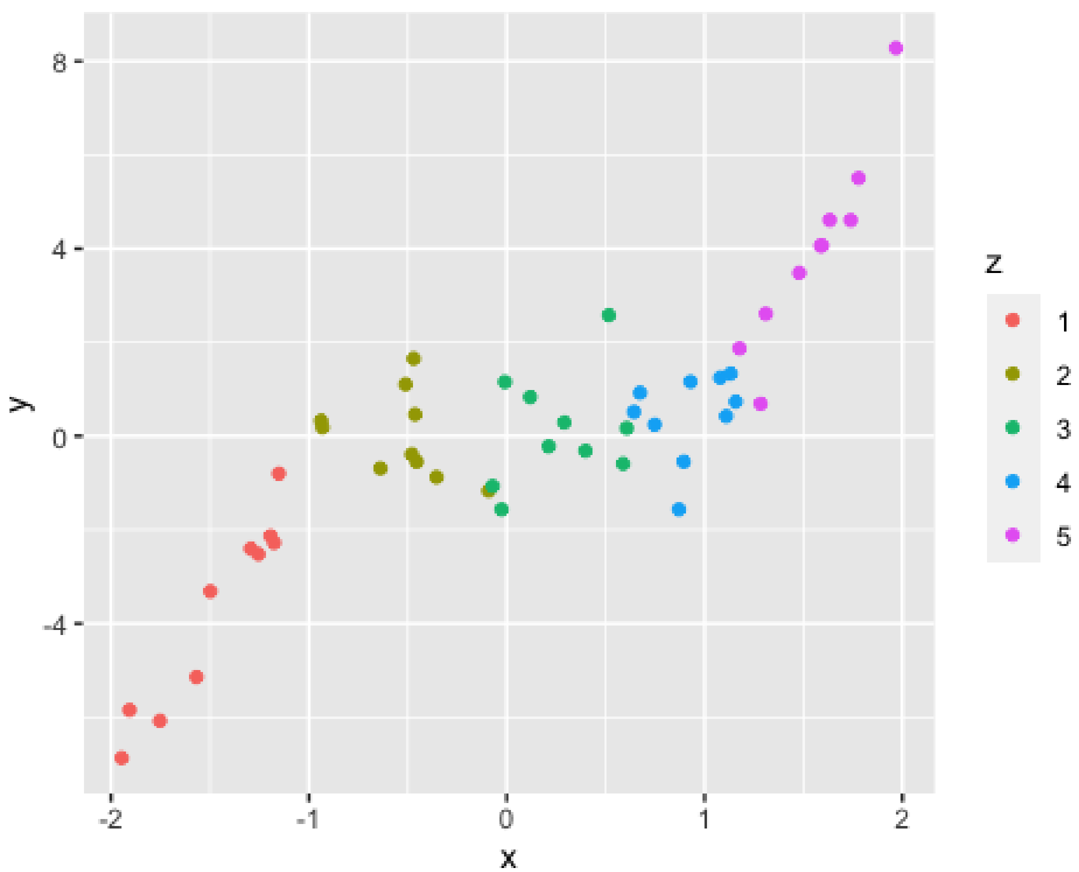
```
p1 + geom_point()
```



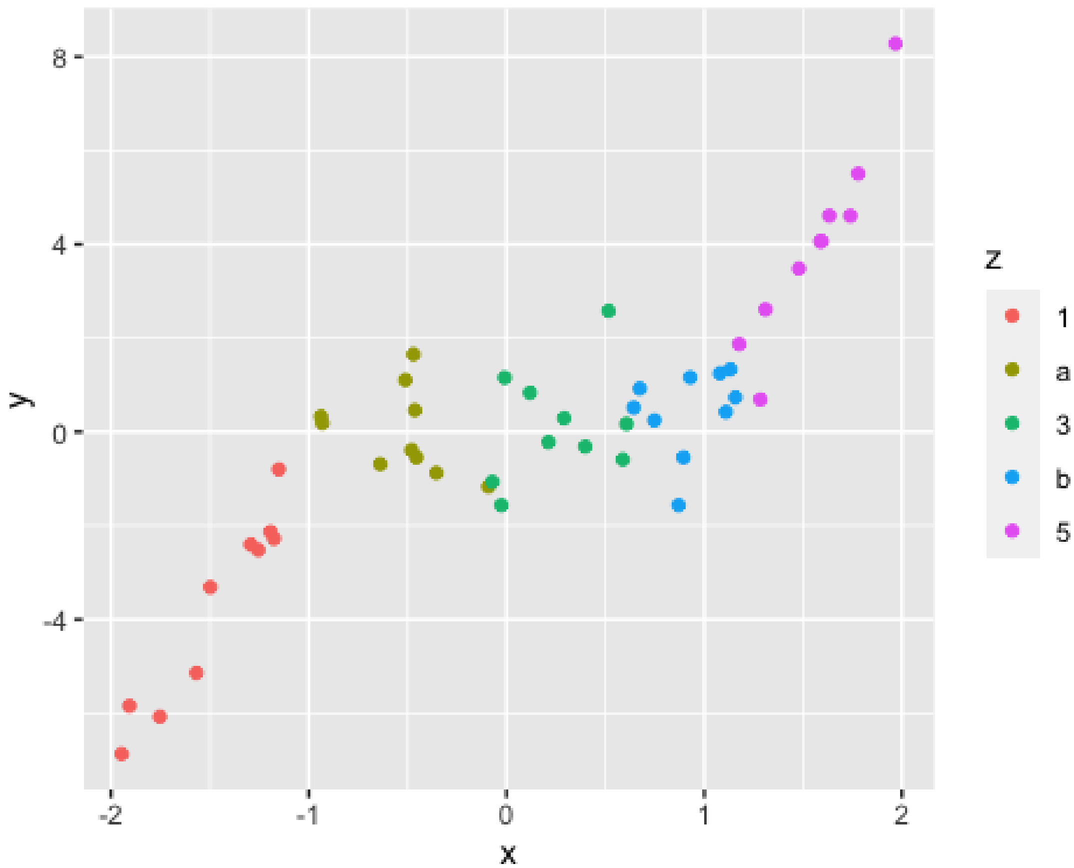

```
# equivalent to:
# p1 + geom_point() + scale_x_continuous() + scale_y_continuous() +
scale_color_continuous()
# set the `breaks` argument:
p1 + geom_point() + scale_x_continuous(breaks = c(-1.5, 0, 1.5)) +
scale_color_continuous(breaks = c(2, 4), labels = c("No", "Yes"))
```




```
p1 + geom_point(aes(colour = as.factor(z)))
```




```
# equivalent to:
# p1 + geom_point(aes(colour = as.factor(z))) + scale_x_continuous() +
scale_y_continuous() + scale_colour_hue()
# set the `labels` argument:
p1 + geom_point(aes(colour = as.factor(z))) + scale_colour_hue(labels = c("2"
= "a", "4" = "b"))
```



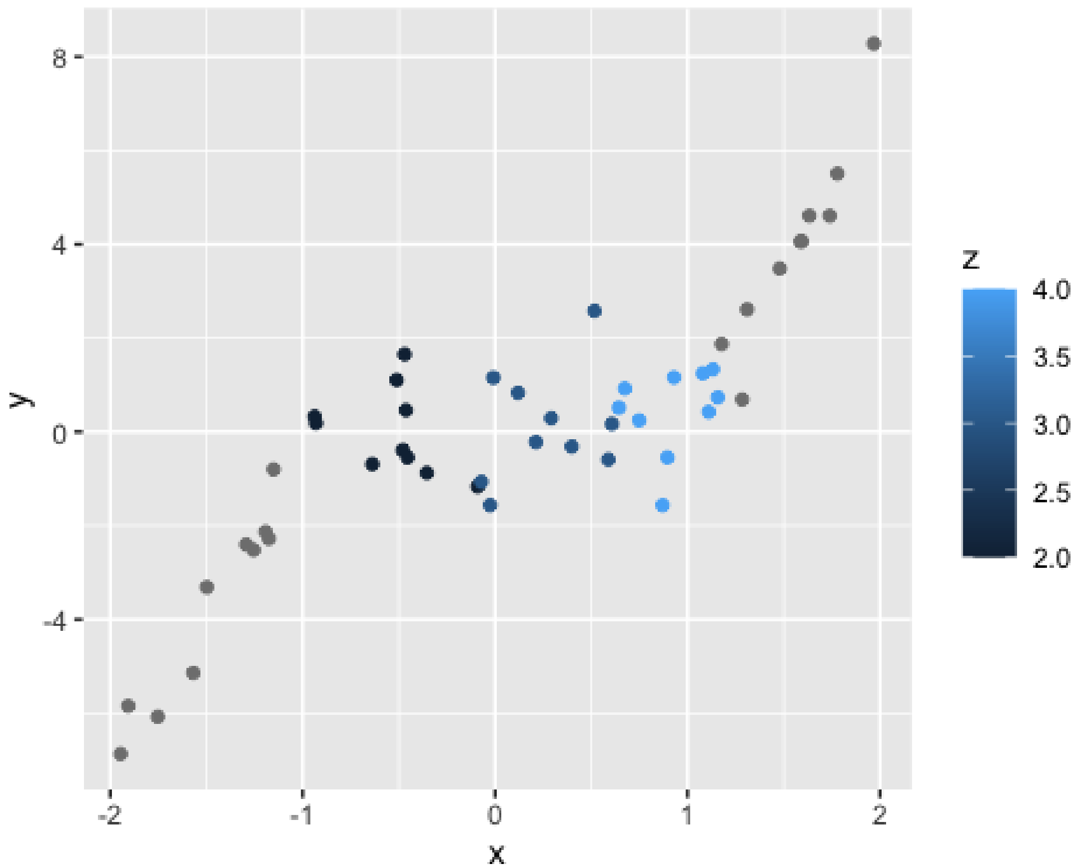
Setting `breaks` or `labels` to `NULL` can suppress them.

limits

The `limits` of a scale controls *the range of values to be mapped on the scale*.

For a *continuous* scale, `limits` is specified by a *numeric vector* of length 2, representing the upper and lower limits.

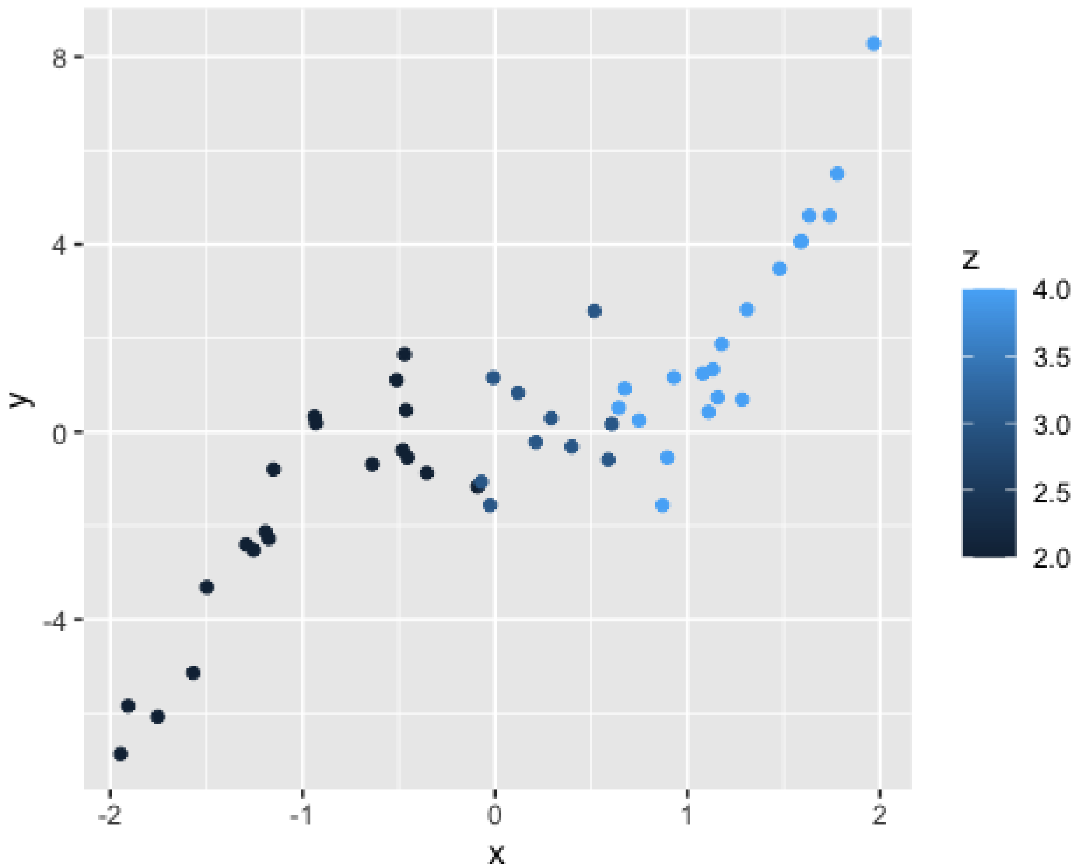
```
p1 + geom_point() + scale_colour_continuous(limits = c(2, 4))
```



In the above plot, z values that fall outside the range specified by `limits` will not be mapped to the color scale. They are colored *grey* in the plot.

This can be overridden by the `oob` (out of bounds) argument.

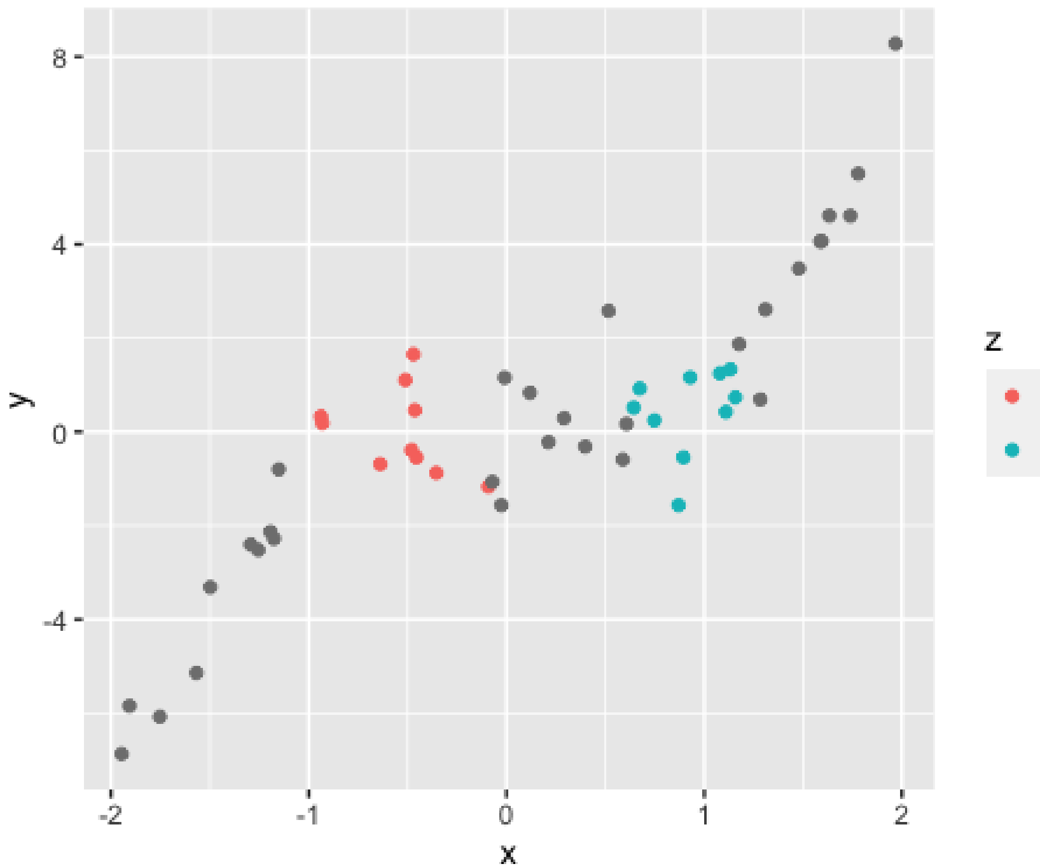
```
p1 + geom_point() + scale_colour_continuous(limits = c(2, 4), oob =
scales::squish)
```



In the above plot, the out-of-bound values are colored with *extreme colors of the scale*.

For *discrete* scales, `limits` is set by a *character vector* which enumerates all possible values:

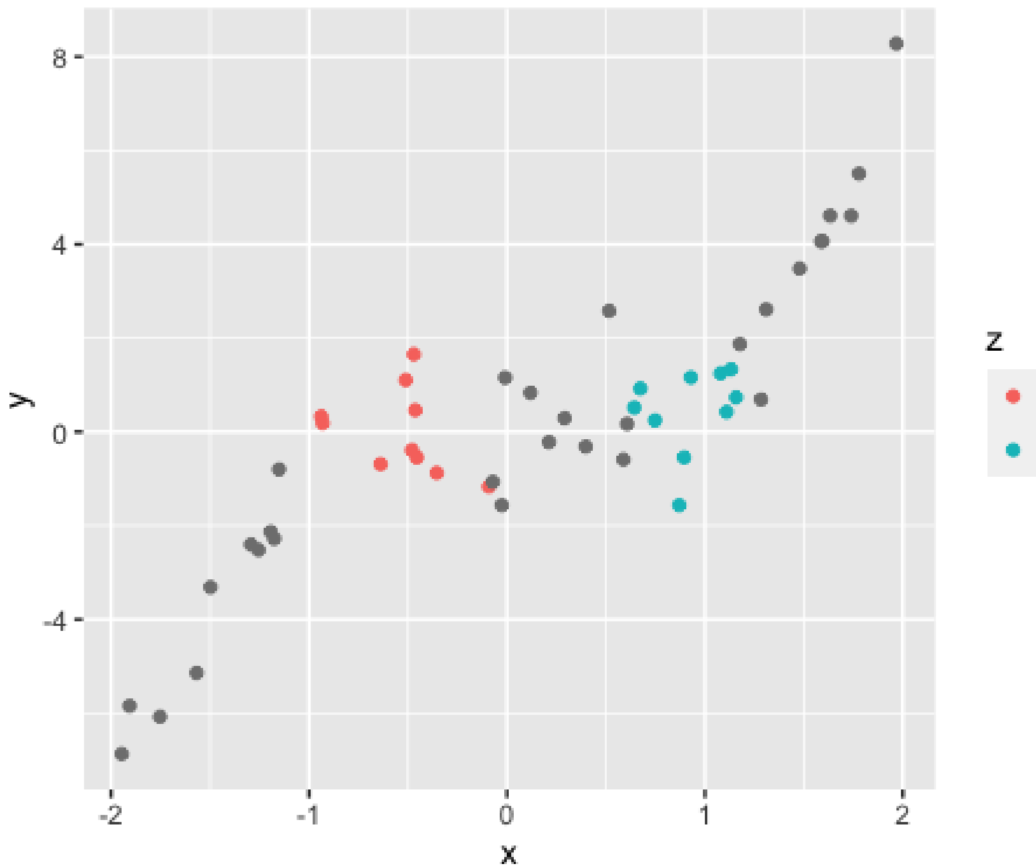
```
p1 + geom_point(aes(colour = as.factor(z))) + scale_colour_hue(limits =
c("2", "4"))
```



In the above plot, only the observations corresponding to the two `z` values specified in `limits` are colored.

Because modifying the limits is so common in practice, `ggplot2` provides some helpers: `xlim()`, `ylim()` and `lims()`.

```
p1 + geom_point(aes(colour = as.factor(z))) + lims(color = c("2", "4"))
```

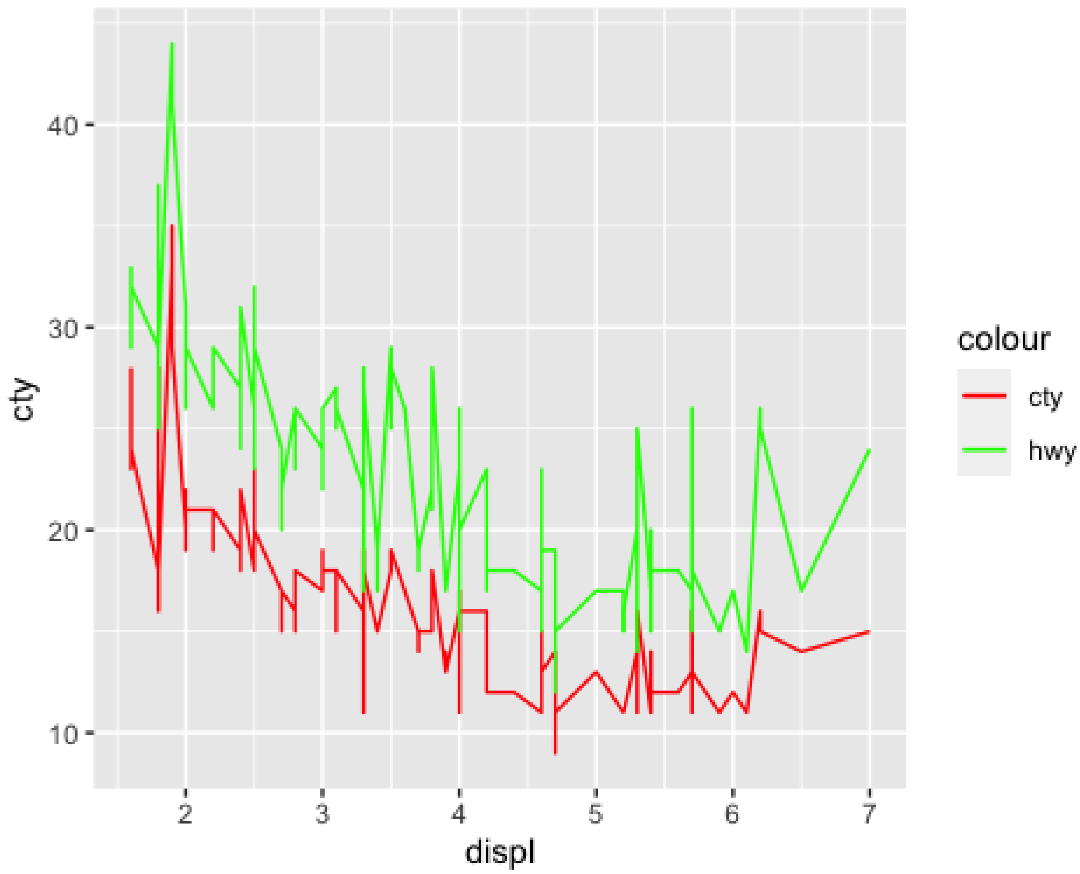


The `scale_*_manual` Functions

The `scale_*_manual` functions allow us to specify our own set of mappings from levels in the data to aesthetic values.

The argument `values` specifies the values that the scale should produce. It usually takes the form of named vectors.

```
ggplot(mpg, aes(displ)) + geom_line(aes(y = cty, colour = "cty")) +
geom_line(aes(y = hwy, colour = "hwy")) + scale_colour_manual(values =
c("cty" = "red", "hwy" = "green"))
```



[Task 3: Plotting Multiple Lines]

In the above example, we used two `geom_line` functions to add two lines to the plot one by one. However, it would be very tedious if we want to plot many lines.

Another way to do this is to convert the data frame to a long format and use the `color` aesthetic.

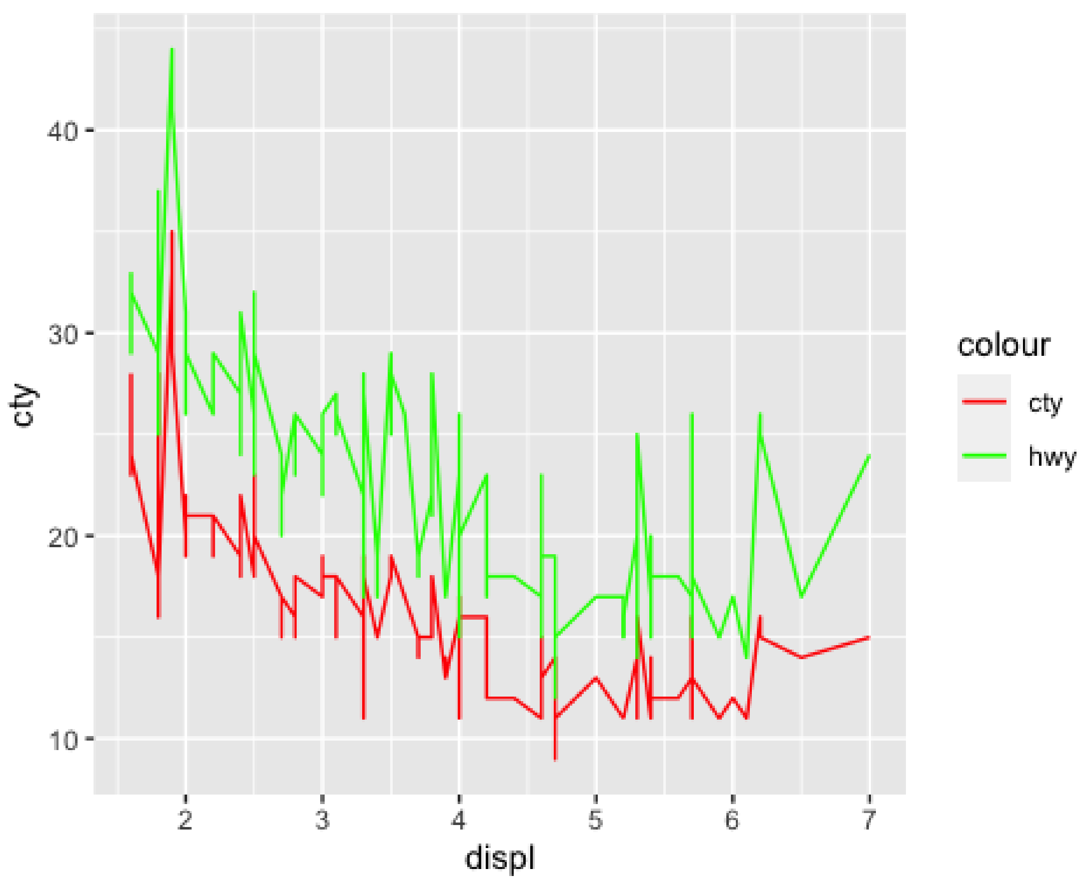
(a) Convert the data frame `mpg` to a long format `mpg_l`, where the column `type` indicates whether the record is `cty` or `hwy`, and the column `miles` stores the corresponding value of `cty` or `hwy`. The expected output is as follows.

```
# A tibble: 468 x 11
   manufacturer model displ  year   cyl trans      drv   fl    class   type
miles
   <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <chr> <chr>   <chr>
<int>
 1 audi         a4      1.8  1999     4 auto(l5)   f     p     compact cty
18
 2 audi         a4      1.8  1999     4 auto(l5)   f     p     compact hwy
29
 3 audi         a4      1.8  1999     4 manual(m5) f     p     compact cty
```

```
21
 4 audi         a4      1.8   1999      4 manual(m5) f      p      compact hwy
29
 5 audi         a4      2     2008      4 manual(m6) f      p      compact cty
20
 6 audi         a4      2     2008      4 manual(m6) f      p      compact hwy
31
 7 audi         a4      2     2008      4 auto(av)   f      p      compact cty
21
 8 audi         a4      2     2008      4 auto(av)   f      p      compact hwy
30
 9 audi         a4      2.8   1999      6 auto(l5)   f      p      compact cty
16
10 audi         a4      2.8   1999      6 auto(l5)   f      p      compact hwy
26
# … with 458 more rows
```

(b) Use `mpg_l` to reproduce the same plot created by the following code using `mpg`.

```
ggplot(mpg, aes(displ)) + geom_line(aes(y = cty, colour = "cty")) +
geom_line(aes(y = hwy, colour = "hwy")) + scale_colour_manual(values =
c("cty" = "red", "hwy" = "green"))
```



[End of Task 3]

8.7 Facets

Faceting generates small panels, each showing a different subset of the data.

It is very useful to investigate whether patterns are different across conditions and provides an alternative to displaying categorical variables on a plot.

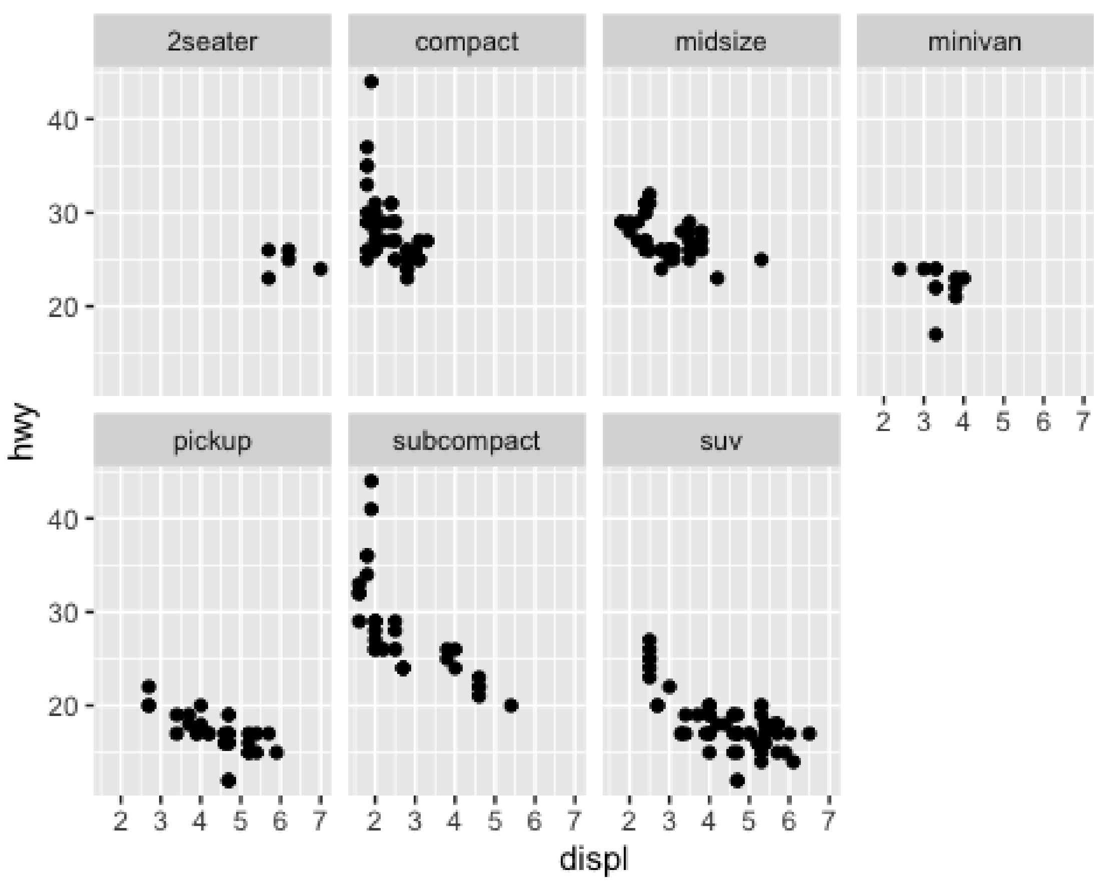
To create facets, we need to specify which variables should be used to split up the data and how the result of faceting should be arranged.

There two types of faceting: **wrap** and **grid**.

facet_wrap()

The function `facet_wrap()` wraps a sequence of panels into 2 dimensions.

```
ggplot(mpg, aes(displ, hwy)) + geom_point() + facet_wrap(~ class, ncol = 4)
# or nrow
```



The faceting rule is specified by a formula argument (using a tilde). In the above plot, the variable `class` is used to split the data. Because `class` has 7 unique values, we end up with 7 panels, and these panels are arranged into a grid of 4 columns.

```r
unique(mpg$class)
```

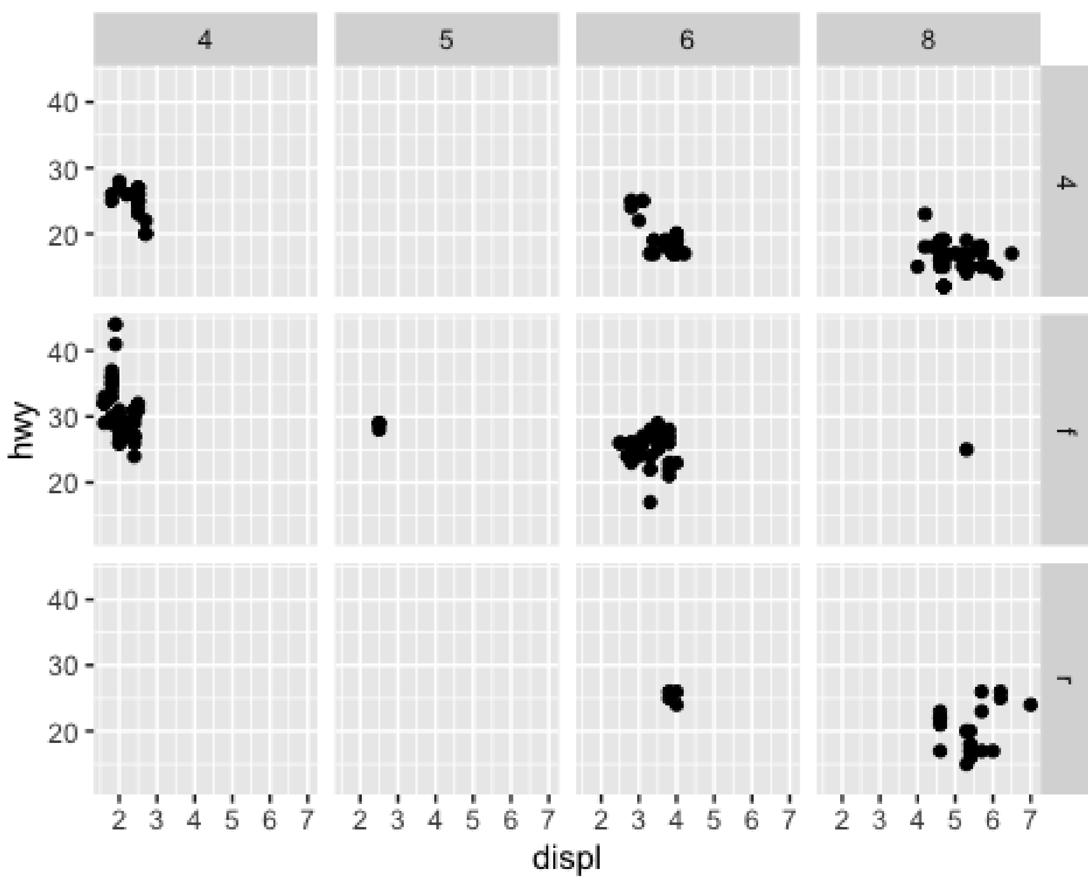
```
## [1] "compact"    "midsize"    "suv"        "2seater"    "minivan"
## [6] "pickup"     "subcompact"
```

facet_grid()

`facet_grid()` forms a matrix of panels defined by row and column faceting variables.

It is most useful when you have two discrete variables, and all combinations of the variables exist in the data. If you have only one variable with many levels, use `facet_wrap()` instead.

```r
ggplot(mpg, aes(displ, hwy)) + geom_point() + facet_grid(drv ~ cyl)
```



The faceting rule is also specified by a formula, with the rows (of the tabular display) on the LHS and the columns (of the tabular display) on the RHS.

Because `drv` has 3 unique values and `cyl` has 4 unique values, we end up with 12 panels, arranged into a grid of 3 rows and 4 columns.

```r
unique(mpg$drv)
```

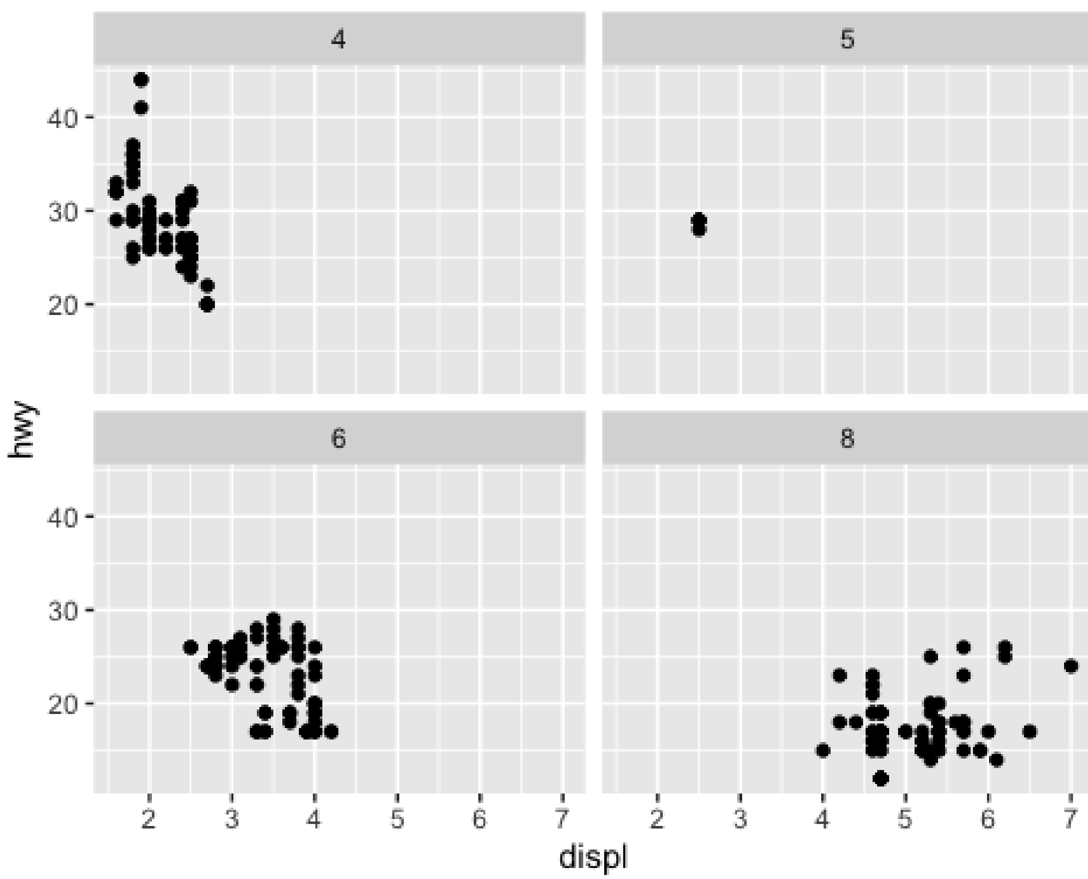
```
## [1] "f" "4" "r"
```

```
unique(mpg$cyl)

## [1] 4 6 8 5
```

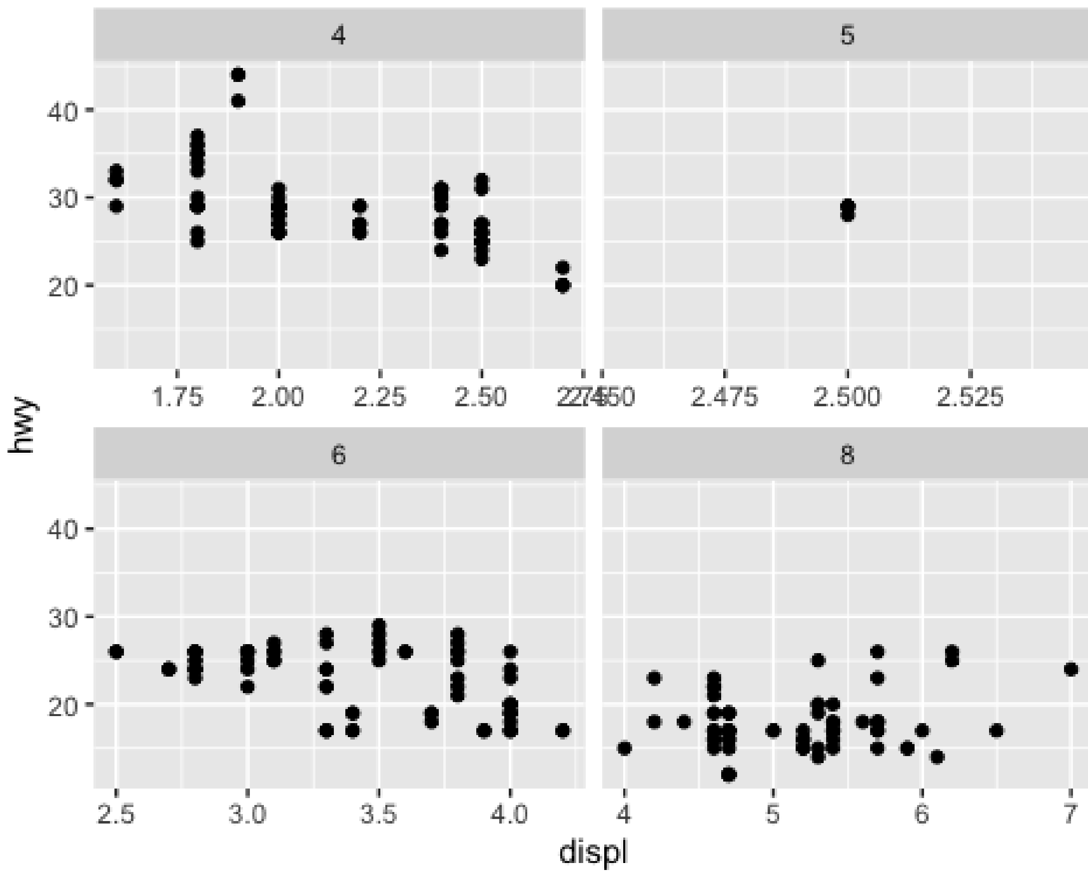
Controlling Scales

Both `facet_wrap()` and `facet_grid()` have the `scales` parameter that controls whether the position scales are the same in all panels (`fixed`) or allowed to vary between panels (`free`, `free_x`, `free_y`).

```
# by default, scales = "fixed":
ggplot(mpg, aes(displ, hwy)) + geom_point() + facet_wrap(~ cyl)
```




```
# set `scales = "free_x"` to allow different scales for the x axis:
ggplot(mpg, aes(displ, hwy)) + geom_point() + facet_wrap(~ cyl, scales =
"free_x")
```

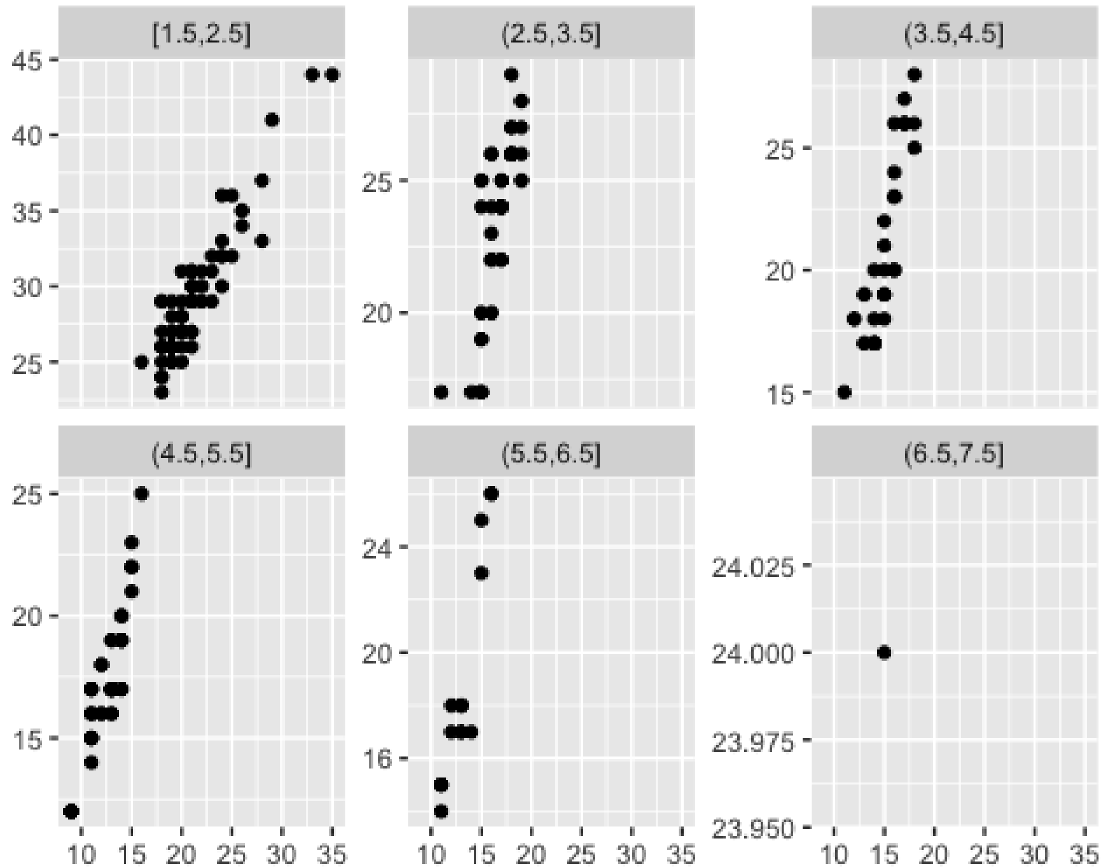


Faceting Continuous Variables

We must first *discretize* continuous variables so as to facet them.

ggplot2 provides 3 helper functions to discretize continuous variables: `cut_interval(x, n)`, `cut_width(x, width)`, and `cut_number(x, n = 10)`.

```
ggplot(mpg, aes(cty, hwy)) + geom_point() + labs(x = NULL, y = NULL) +
  facet_wrap(~ cut_width(displ, 1), nrow = 2, scales = "free_y")
```



`cut_width(displ, 1)` discretize the continuous variable `displ` by dividing it into bins of width 1. The interval of `displ` is displayed at the top of each panel.

8.8 Coordinates

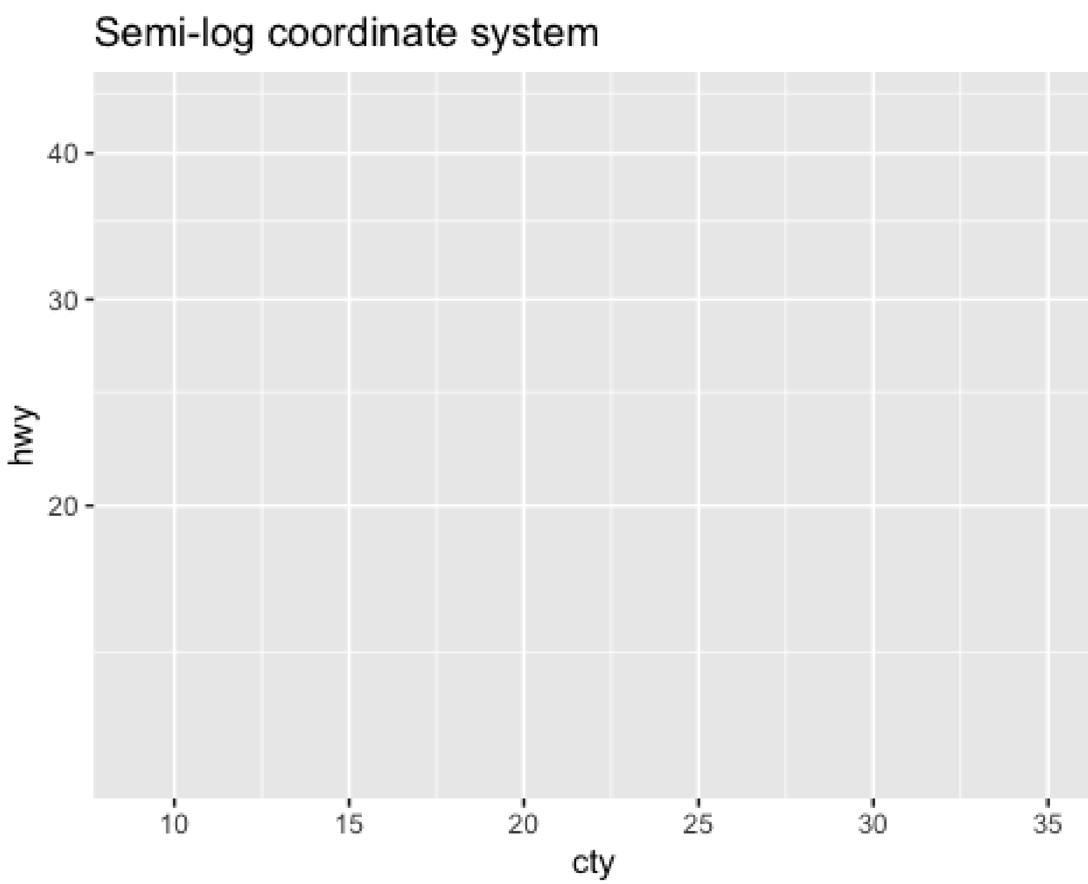
A coordinate system (`coord`) maps the position of objects onto the plane of the plot.

While `scale` controls the values that appear on the axes and how they map from data to position, it is `coordinate` that actually draws the axes and grid lines.

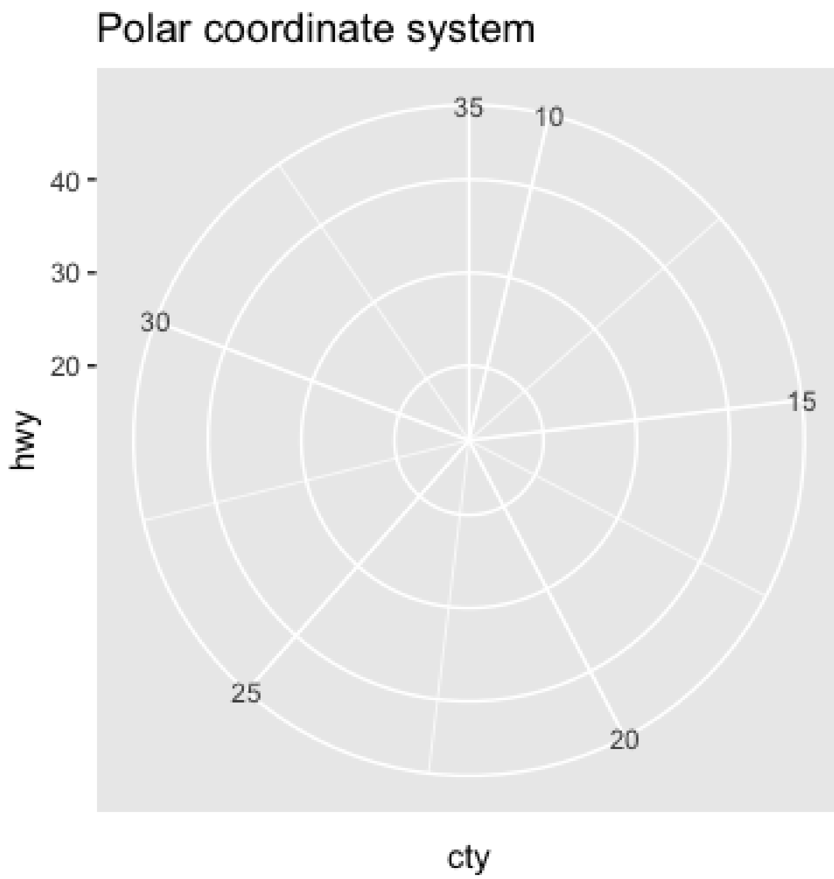
There are 2 types of coordinate system:

- **Linear** coordinate systems preserve the shape of geoms. It requires a fixed and equal spacing between values on the axes.
  - `coord_cartesian()` (default coordinate)
  - `coord_flip()` (with x and y axes flipped)
  - `coord_fixed()` (with a fixed aspect ratio)
- **Non-linear** coordinate systems can change the shape of geoms.
  - `coord_map()`, `coord_quickmap()`, `coord_polar()`, and `coord_trans()`

```
ggplot(mpg, aes(cty, hwy)) + coord_trans(y = "log10") + ggtitle(label =
"Semi-log coordinate system")
```




```
ggplot(mpg, aes(cty, hwy)) + coord_polar() + ggtitle(label = "Polar
coordinate system")
```



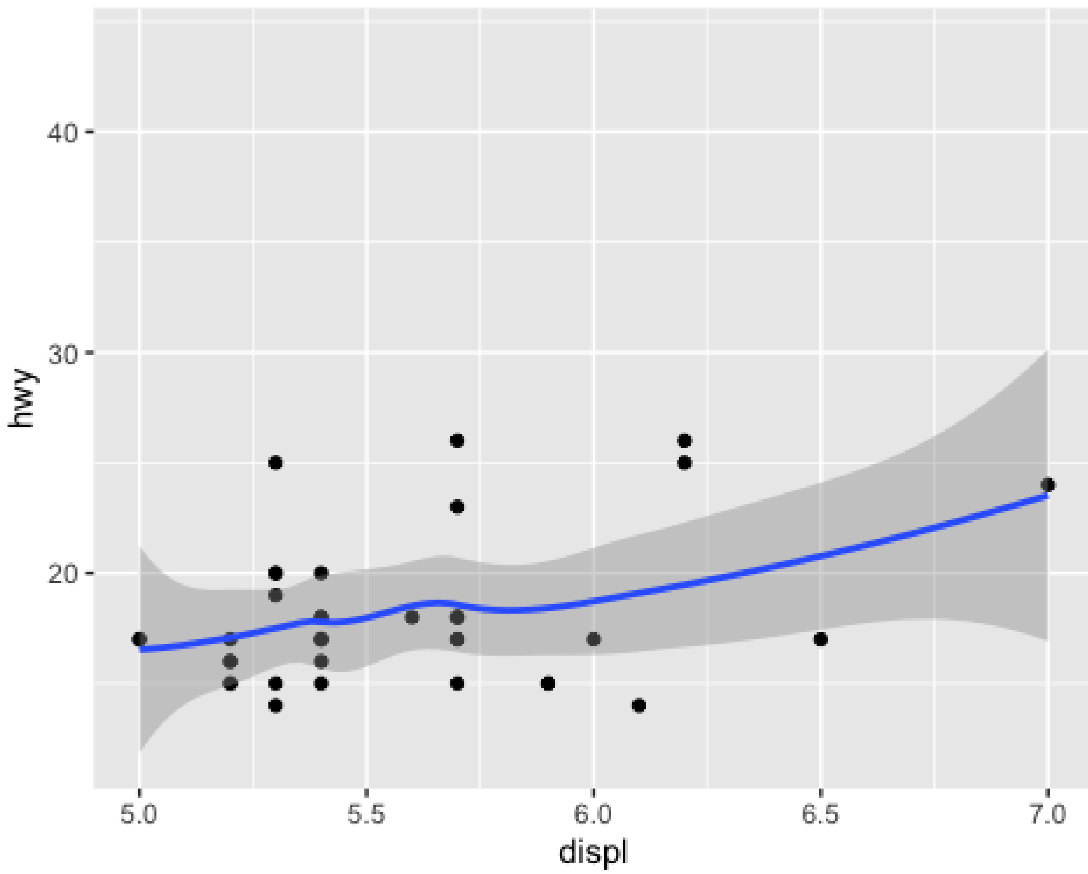
Zooming in with `xlim` and `ylim`

Linear coordinate systems (`coord_cartesian()`, `coord_flip()`, and `coord_fixed()`) have arguments `xlim` and `ylim`.

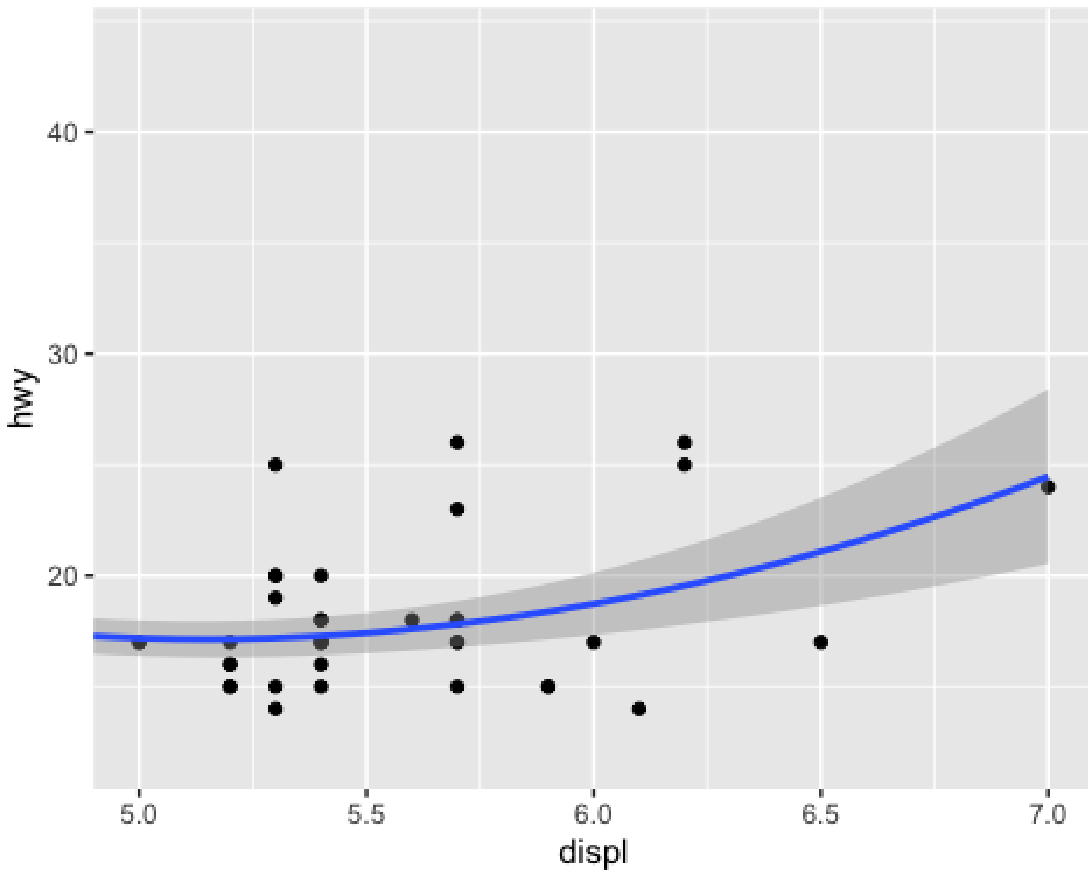
Setting *scale limits* throws any data outside the `limits` away, while setting *coordinate limits* keeps all the data but only displays the specified region of the plot.

In other words, setting coordinate `limits` performs purely **visual zooming** and does not affect the underlying data.

```
# setting scale limits:
ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth() +
scale_x_continuous(limits = c(5, 7))

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'

## Warning: Removed 196 rows containing non-finite values (stat_smooth).

## Warning: Removed 196 rows containing missing values (geom_point).
```




```
# setting coordinate limits:
ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth() +
coord_cartesian(xlim = c(5, 7))

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Map Projections with coord_map()

We use the "world" map provided by the `maps` package. The function `map_data()` in `ggplot2` turns data from the `maps` package in to a data frame suitable for plotting.

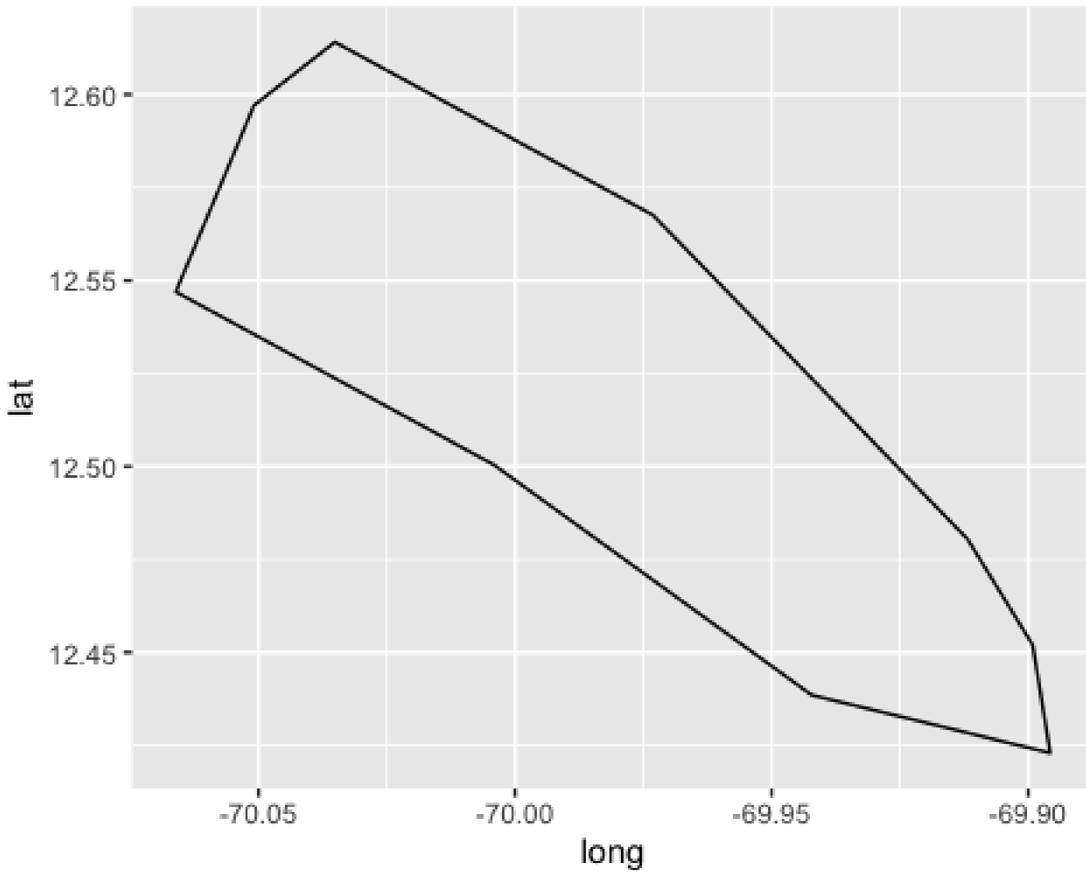
```
# install.packages("maps")
library(maps)

##
## Attaching package: 'maps'

## The following object is masked from 'package:purrr':
##
##     map

head(map_data("world"))

##        long      lat group order region subregion
## 1 -69.89912 12.45200     1     1  Aruba      <NA>
## 2 -69.89571 12.42300     1     2  Aruba      <NA>
## 3 -69.94219 12.43853     1     3  Aruba      <NA>
## 4 -70.00415 12.50049     1     4  Aruba      <NA>
## 5 -70.06612 12.54697     1     5  Aruba      <NA>
## 6 -70.05088 12.59707     1     6  Aruba      <NA>
```

We can think of the world map as consisting of many polygons. The data frame `map_data("world")` contains the location (`long` and `lat`) of the polygons' vertices.

We can plot a polygon by connecting its vertices using `geom_path()`. `geom_path()` connects the observations in the order in which they appear in the data.

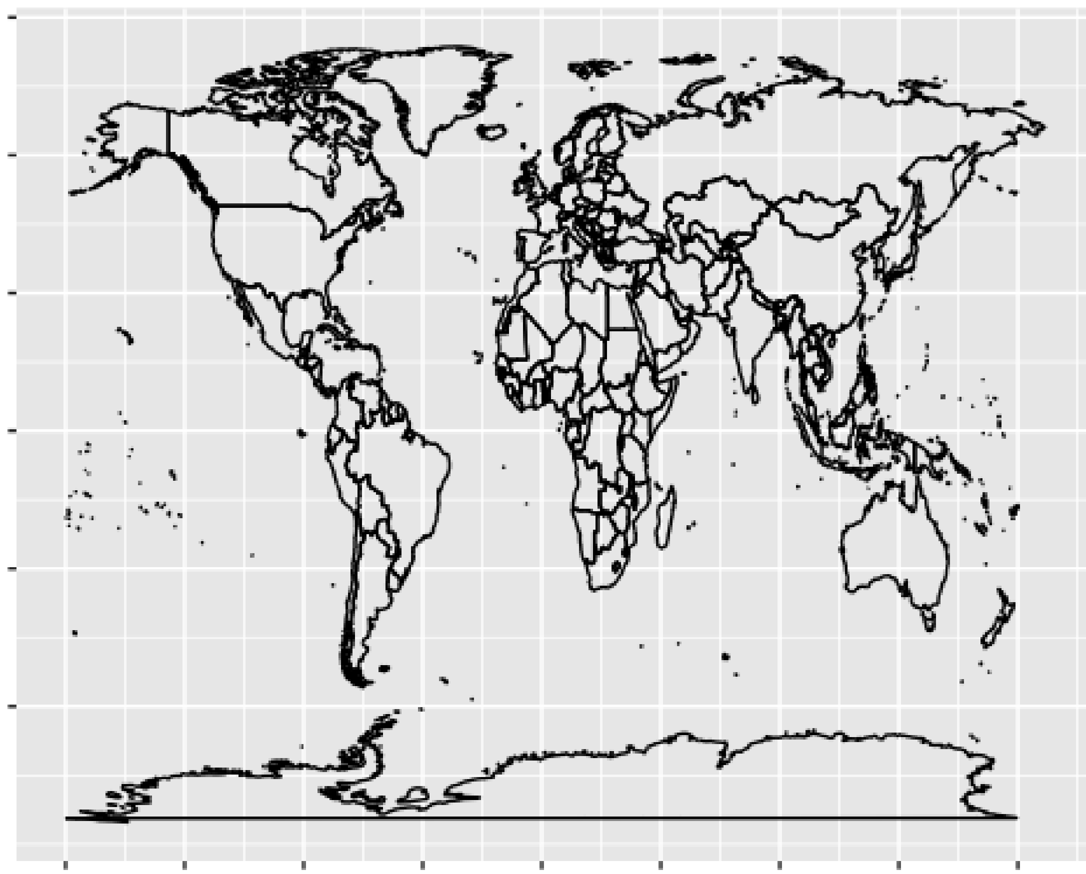
The following code plots the 1st polygon (indicated by `group`):

```
test <- map_data("world") %>% filter(group == 1)
ggplot(test, aes(long, lat)) + geom_path()
```



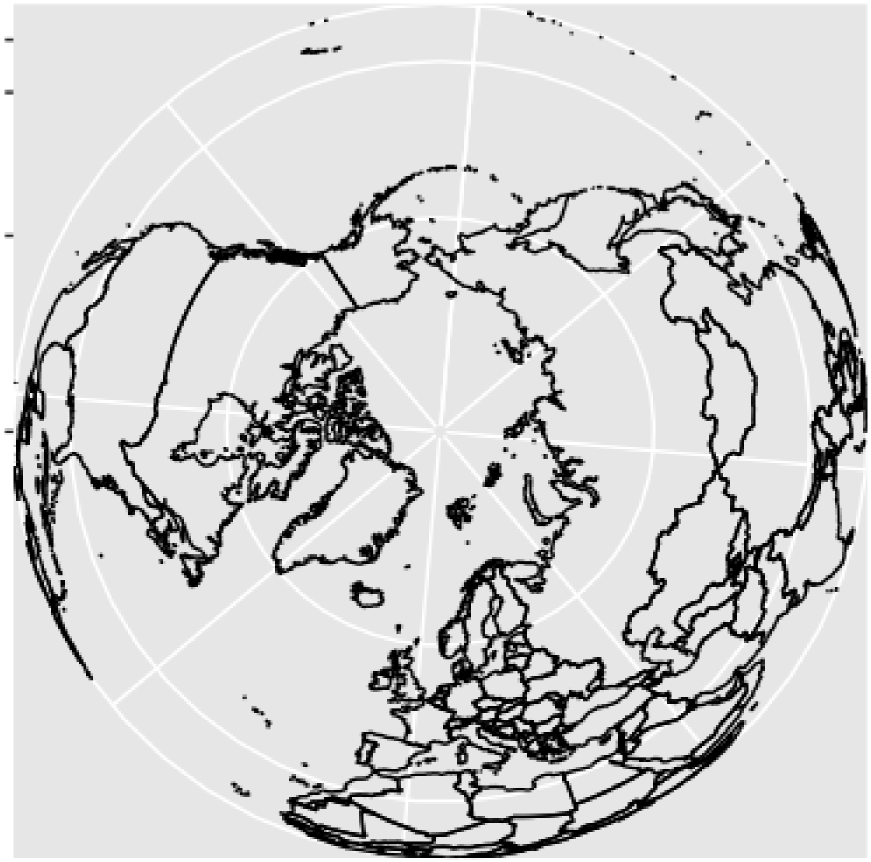
Now, let's plot the whole world map:

```
worldmap <- ggplot(map_data("world"), aes(long, lat, group = group)) +
geom_path() + scale_y_continuous(NULL, breaks = (-2:3) * 30, labels = NULL) +
scale_x_continuous(NULL, breaks = (-4:4) * 45, labels = NULL)
worldmap
```



We can use the function `coord_map()` in `ggplot2` to project a portion of the earth, which is approximately spherical, onto a flat 2D plane.

```
# install.packages("mapproj")
library(mapproj)
worldmap + coord_map("ortho")
```



[Task 4: Population Density Heatmap of Hong Kong]

We are going to reproduce the heatmap that represents the population density of Hong Kong's 18 districts.

Download `hk_mapdata.csv` and `hk_districts.csv` from Canvas.

`hk_mapdata.csv` contains the data on the latitute and longitude coordinates of the boundaries of Hong Kong's 18 districts. Load the data to create a tibble named `hk_mapdata`:

```
hk_mapdata <- read_csv("hk_mapdata.csv")

## Parsed with column specification:
## cols(
##   X = col_double(),
##   Y = col_double(),
##   District = col_character(),
##   Long = col_double(),
##   Lat = col_double(),
##   Polygon = col_double(),
##   Dist_Index = col_double()
## )
```

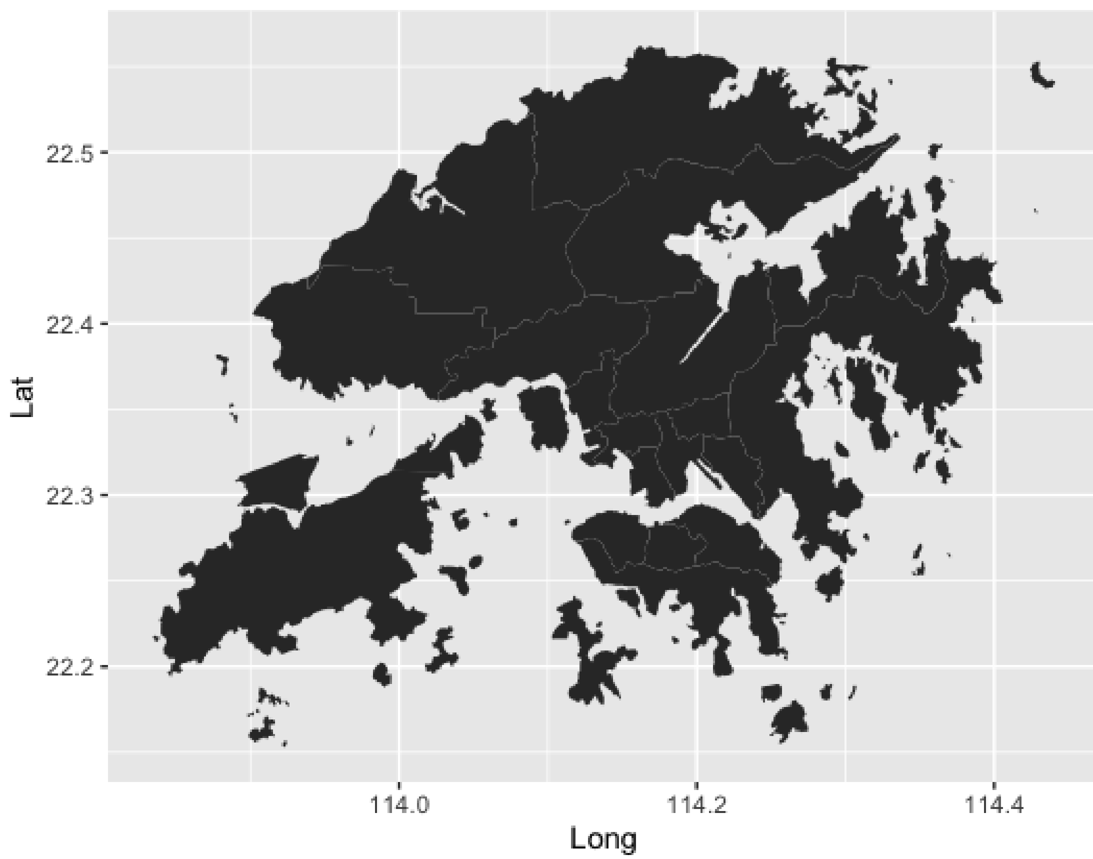
```
hk_mapdata

## # A tibble: 25,509 x 7
##        X     Y District              Long   Lat Polygon Dist_Index
##    <dbl> <dbl> <chr>                <dbl> <dbl>   <dbl>      <dbl>
##  1  114.  22.3 Central and Western  114.  22.3       1          1
##  2  114.  22.3 Central and Western  114.  22.3       1          1
##  3  114.  22.3 Central and Western  114.  22.3       1          1
##  4  114.  22.3 Central and Western  114.  22.3       1          1
##  5  114.  22.3 Central and Western  114.  22.3       1          1
##  6  114.  22.3 Central and Western  114.  22.3       1          1
##  7  114.  22.3 Central and Western  114.  22.3       1          1
##  8  114.  22.3 Central and Western  114.  22.3       1          1
##  9  114.  22.3 Central and Western  114.  22.3       1          1
## 10  114.  22.3 Central and Western  114.  22.3       1          1
## # … with 25,499 more rows
```

(a) Find out how many polygons the "Islands" district contains, and how many polygons the "Sha Tin" district contains.

(b) Use `geom_polygon()` to plot the boundary of the "Sha Tin" district (`?geom_polygon`).

(c) Use `geom_polygon()` to plot the boundary of the "Islands" district.

(d) Use `geom_polygon()` to plot the boundary of all 18 districts. The expected plot is as follows:



Tips:

1. A district may be associated with multiple polygons, and a polygon may be associated with multiple districts. Use `table(hk_mapdata$Polygon, hk_mapdata$District)` to check.

2. If a group isn't defined by a single variable, but instead by a combination of multiple variables, use `interaction()` to combine them.

(e) `hk_districts.csv` contains the total population (`Population`) and population density (`Density`) of the 18 districts in Hong Kong. Load the data and create a tibble named `hk_districts`.

```
hk_districts <- read_csv("hk_districts.csv")

## Parsed with column specification:
## cols(
##   District = col_character(),
##   Population = col_double(),
##   Area = col_double(),
##   Density = col_character(),
##   Region = col_character(),
##   Code = col_character()
## )

hk_districts
```

```
## # A tibble: 18 x 6
##    District            Population   Area Density  Region Code 
##    <chr>                    <dbl>  <dbl> <chr>    <chr>  <chr>
##  1 Central and Western     244600  12.4  < 20000  HK     CW   
##  2 Eastern                 574500  18.6  < 40000  HK     EA   
##  3 Southern                269200  38.8  < 10000  HK     SO   
##  4 Wan Chai                150900   9.83 < 20000  HK     WC   
##  5 Sham Shui Po            390600   9.35 < 50000  KL     SS   
##  6 Kowloon City            405400  10.0  < 50000  KL     KC   
##  7 Kwun Tong               641100  11.3  >= 50000 KL     KU   
##  8 Wong Tai Sin            426200   9.3  < 50000  KL     WT   
##  9 Yau Tsim Mong           318100   6.99 < 50000  KL     YT   
## 10 Islands                 146900 175.   < 10000  NT     IS   
## 11 Kwai Tsing              507100  23.3  < 30000  NT     KI   
## 12 North                   310800 137.   < 10000  NT     NO   
## 13 Sai Kung                448600 130.   < 10000  NT     SK   
## 14 Sha Tin                 648200  68.7  < 10000  NT     ST   
## 15 Tai Po                  307100 136.   < 10000  NT     TP   
## 16 Tsuen Wan               303600  61.7  < 10000  NT     TW   
## 17 Tuen Mun                495900  82.9  < 10000  NT     TM   
## 18 Yuen Long               607200 138.   < 10000  NT     YL   
```

Merge the tibbles `hk_mapdata` and `hk_districts` in order to add district information to `hk_mapdata`.

Tips: Use `inner_join()` in `dplyr` (`?inner_join`).

(f) Create the population density heatmap using the augmented tibble produced in (e).

Tips: Use the `fill` aesthetic.

The expected plot is as follows:

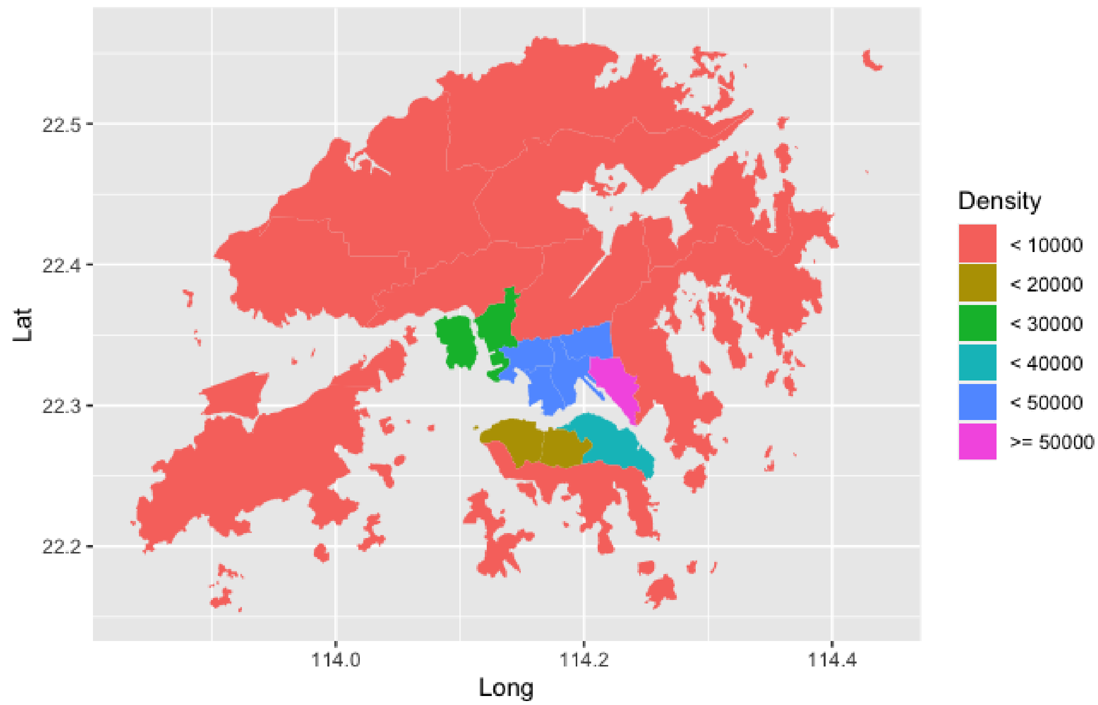


Fig. 7 Task 4 (f)

(g) Create a heatmap for comparing total populations of different districts. The expected plot is as follows:

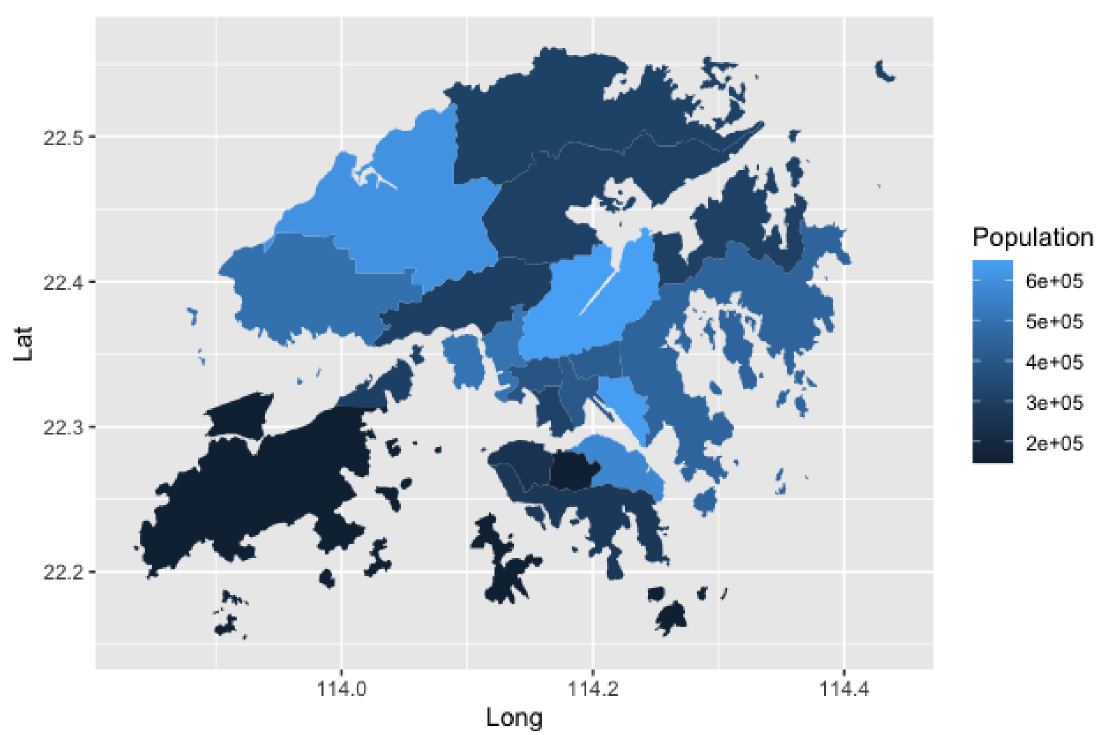


Fig. 7 Task 4 (g)

How the color scale used in this plot is different from that used in the plot in (f)? Why?

[End of Task 4]

8.9 Themes

`ggplot2` separates the control over elements of a plot into *data* and *non-data* parts.

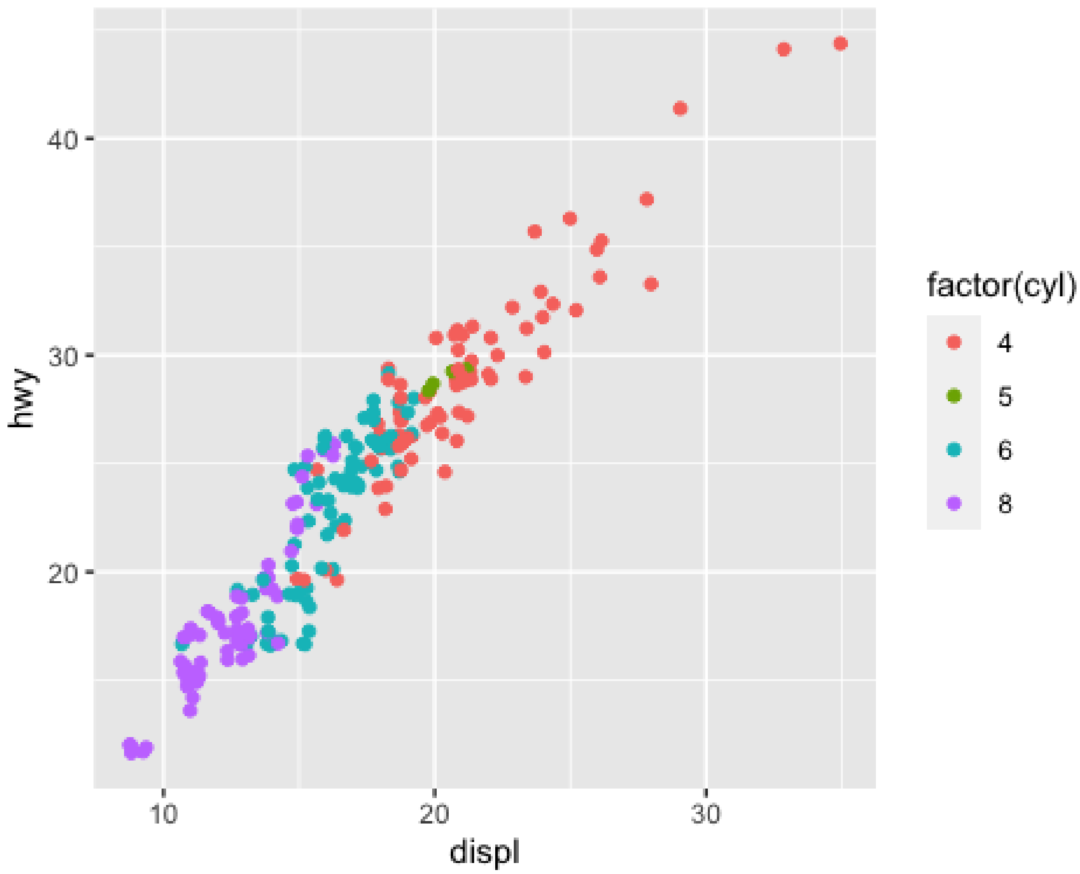
After the plot has been created, every detail of the rendering can be edited using the theming system (theme).

- **Theme Elements**:
  - Theme elements are the non-data elements that we can control.
  - These elements can be roughly grouped into five categories: plot, axis, legend, panel and facet. E.g., `plot.title`, `axis.ticks.x`, `legend.key.height`, etc.
- **Element Functions**:
  - Each element is associated with an element function, describeing the visual properties of the element.
  - There are 4 basic types of built-in element functions: `element_text()`, `element_line()`, `element_rect()`, and `element_blank()`.

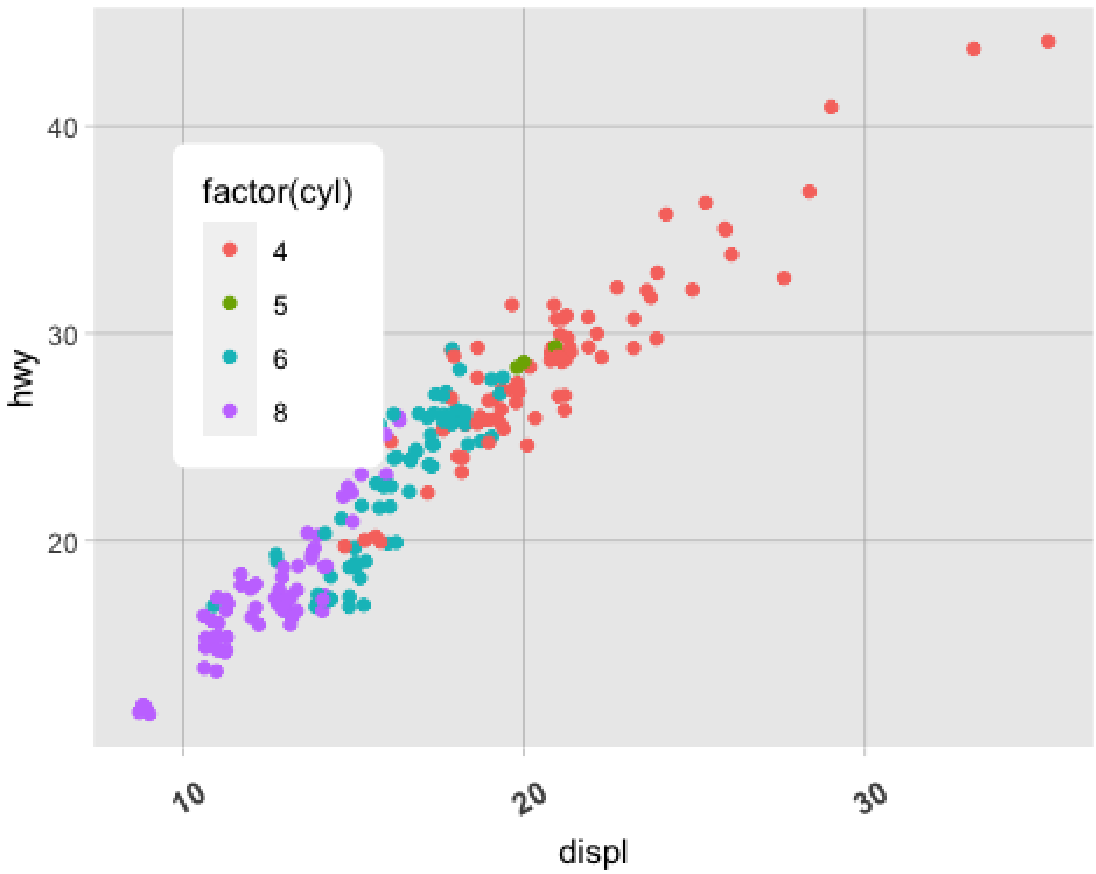
Modifying Theme Elements

We set an `element.name` to a value, or use code of the form `plot + theme(element.name = element_function())` to modify a theme element.

```
ggplot(mpg, aes(displ, hwy)) + geom_jitter(aes(cty, hwy, colour =
factor(cyl)))
```



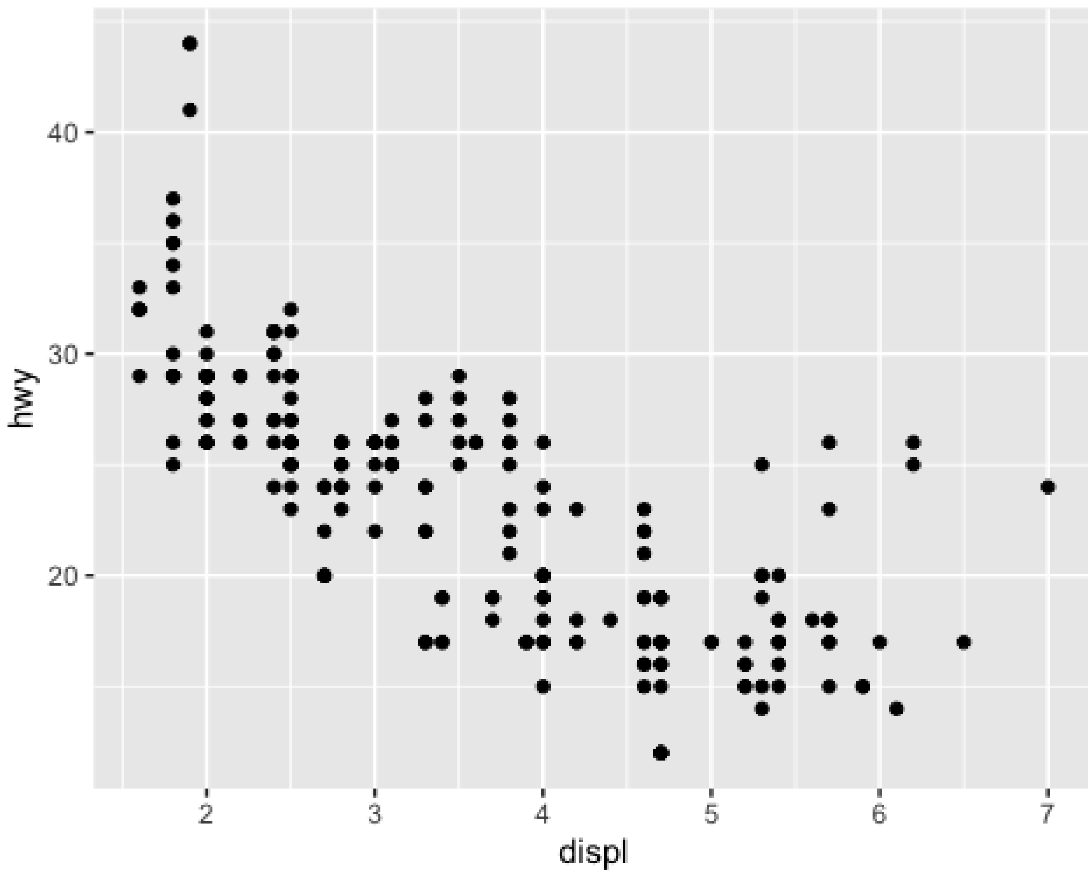

```
# modify the theme:
ggplot(mpg, aes(displ, hwy)) + geom_jitter(aes(cty, hwy, colour =
factor(cyl))) +
  theme(legend.background = element_rect(fill = "white", size = 4, colour =
"white"),
        legend.justification = c(-0.5, 1.5),
        legend.position = c(0, 1),
        axis.ticks = element_line(colour = "grey70", size = 0.2),
        axis.text.x = element_text(margin = margin(t = 10), face = "bold",
size = 10, angle = 30),
        panel.grid.major = element_line(colour = "grey70", size = 0.2),
        panel.grid.minor = element_blank())
```



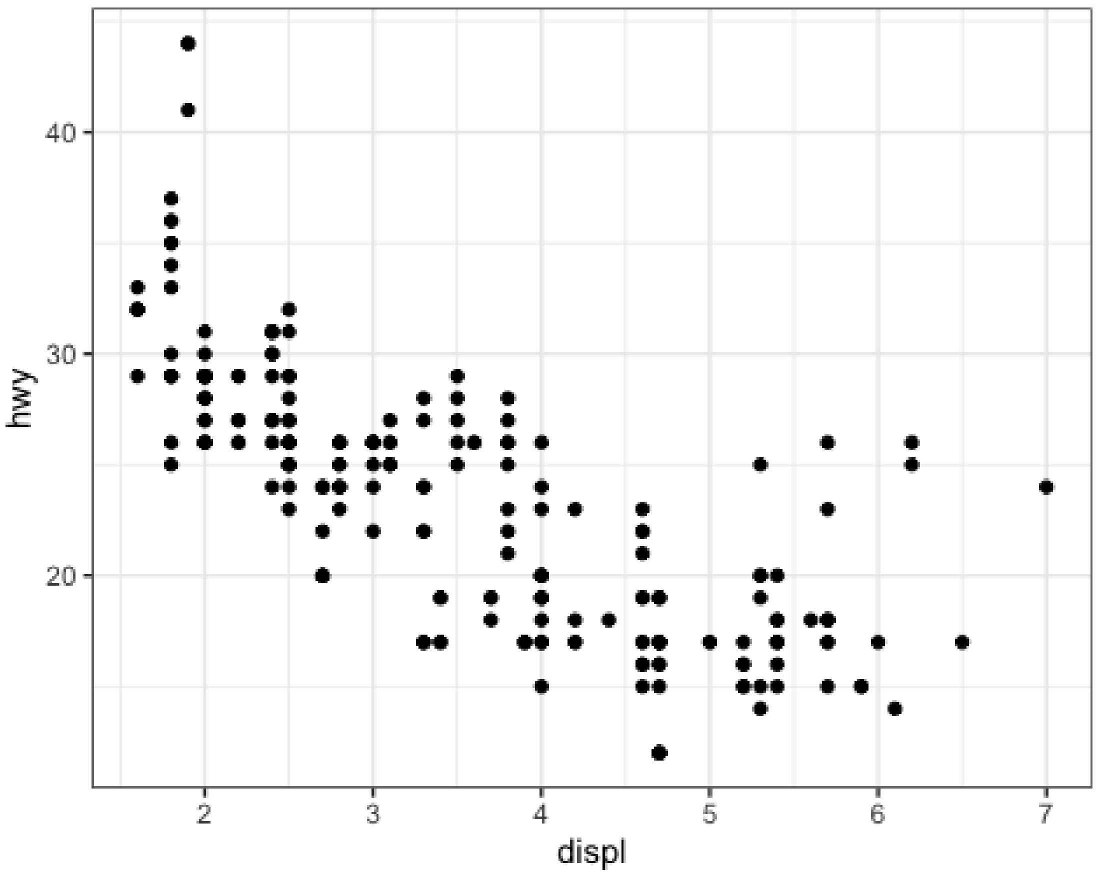
Complete Themes

There are several **complete themes** built in to `ggplot2`, setting all of the theme elements to values designed to work together harmoniously.

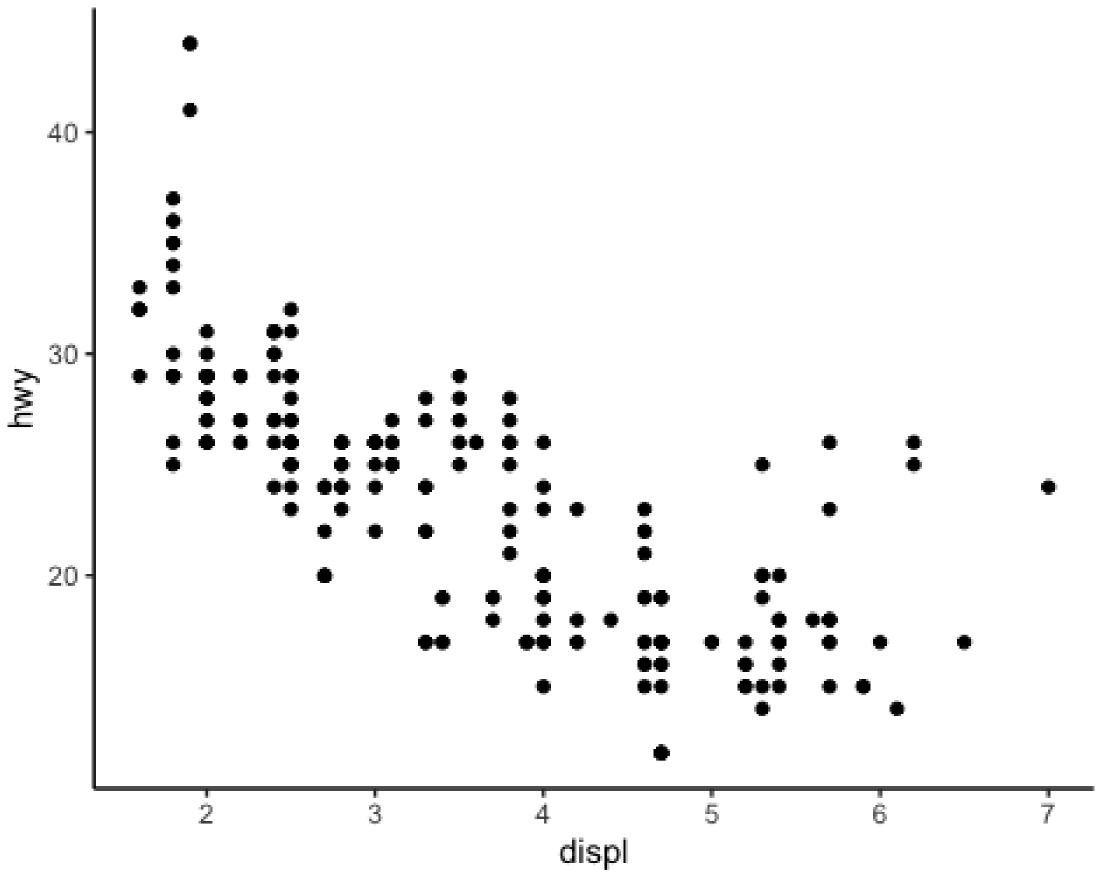
```
ggplot(mpg, aes(displ, hwy)) + geom_point()   # default complete theme is
theme_gray()
```




```
ggplot(mpg, aes(displ, hwy)) + geom_point() + theme_bw()
```

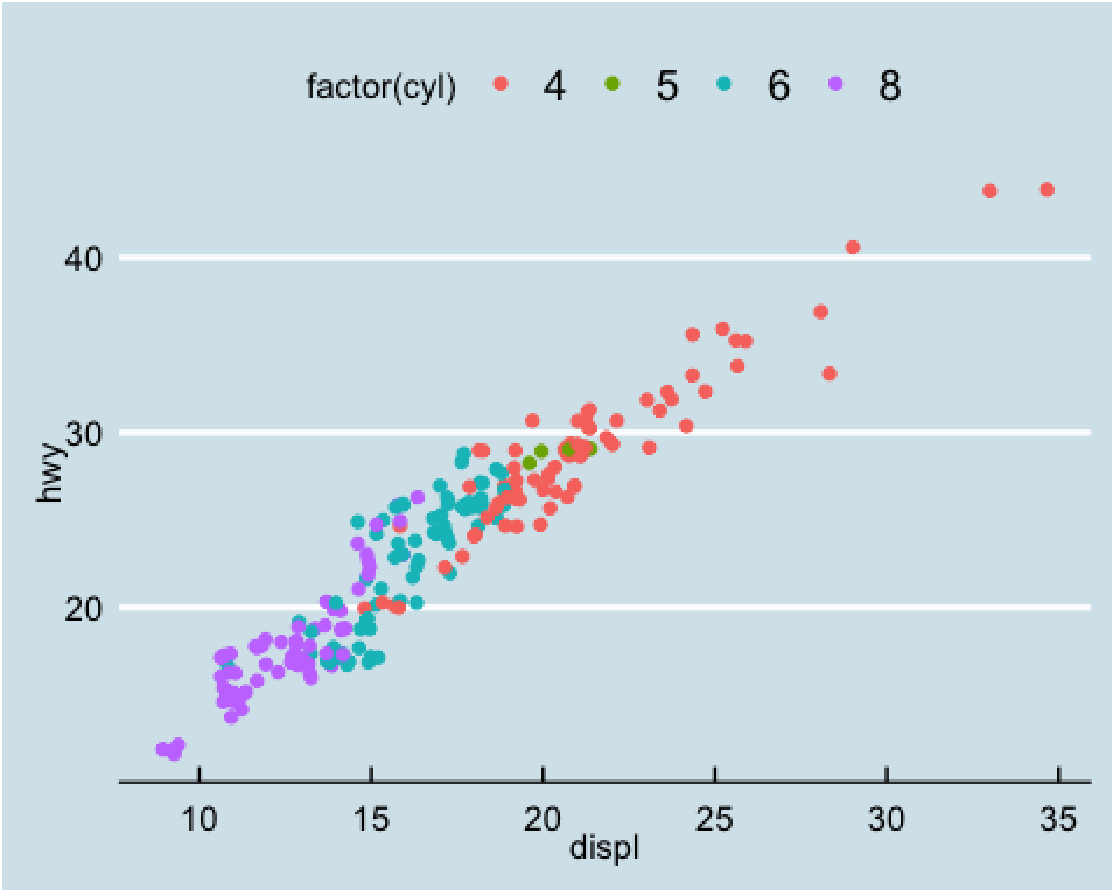



```
ggplot(mpg, aes(displ, hwy)) + geom_point() + theme_classic()
```

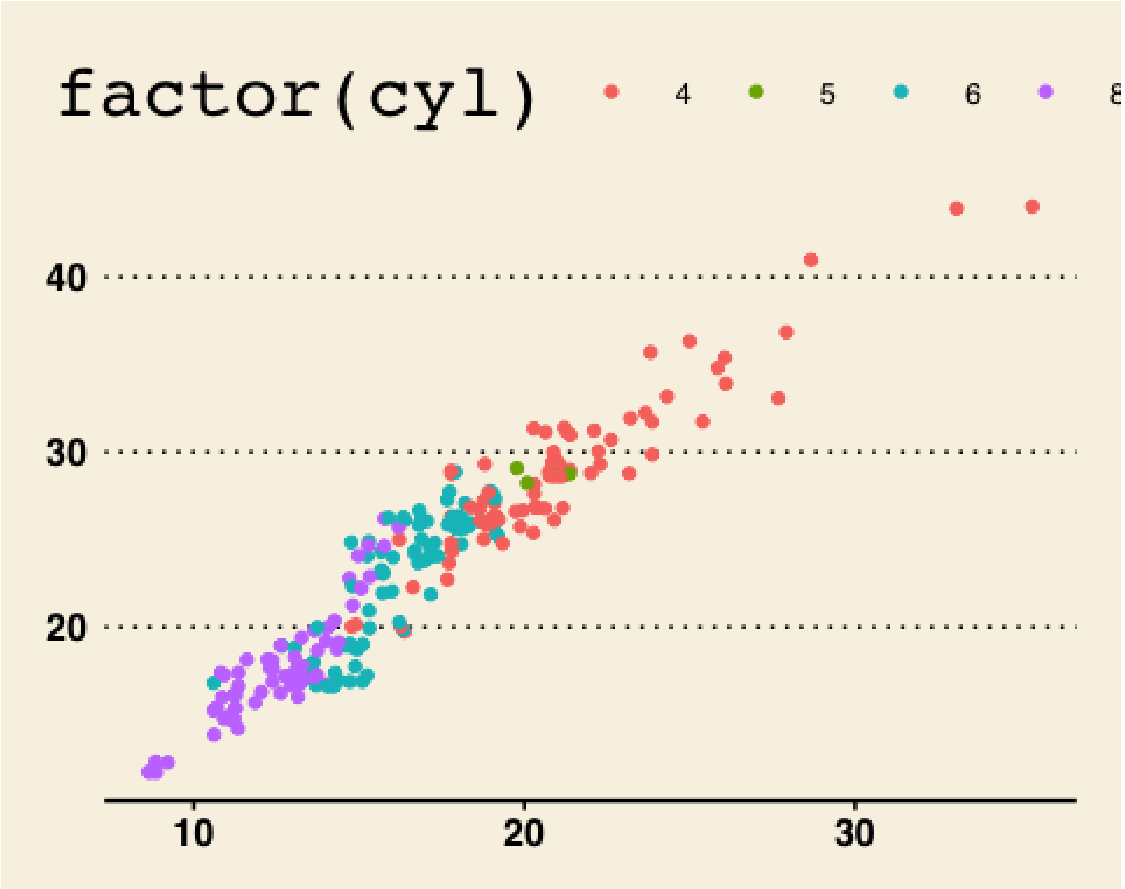


Packages like `ggthemes` provide more complete themes to use.

```
# install.packages("ggthemes")
library(ggthemes)
ggplot(mpg, aes(displ, hwy)) + geom_jitter(aes(cty, hwy, colour =
factor(cyl))) + theme_economist()
```




```
ggplot(mpg, aes(displ, hwy)) + geom_jitter(aes(cty, hwy, colour =
factor(cyl))) + theme_wsj()
```



8.10 Summary

All together, the **layered grammar of graphics** defines a plot as the combination of grammatical elements:

- Essential grammatical elements: data, geoms, aesthetics
- Optional grammatical elements: stats, positions, scales, facets, coords, themes

By thinking "verb", "noun", "adjective", etc. for graphics, `ggplot2` provides a "theory" of graphics on which to build new graphics and graphical objects and shortens the distance from mind to page.

For more information, see https://ggplot2.tidyverse.org/reference/.

[Task 5: Population Density Heatmap of Hong Kong, Continued]

We are gonna improve the plot created in Task 4 (g).

A major problem with this plot is that the light blue ("#56B1F7") indicates districts with a large population, while the dark blue ("#132B43") indicates districts with a large population, which is quite counterintuitive.

- Try to reverse the colour gradient using `scale_fill_gradient()`.

Other improvements you can make include:

- Remove axis labels
- Add a title to the plot, put it in the middle (instead of on the left)
- Change the position of legend to the top
- Change the legend label and title
- Change the background color to white using `theme_bw()`

The expected plot is as follows.

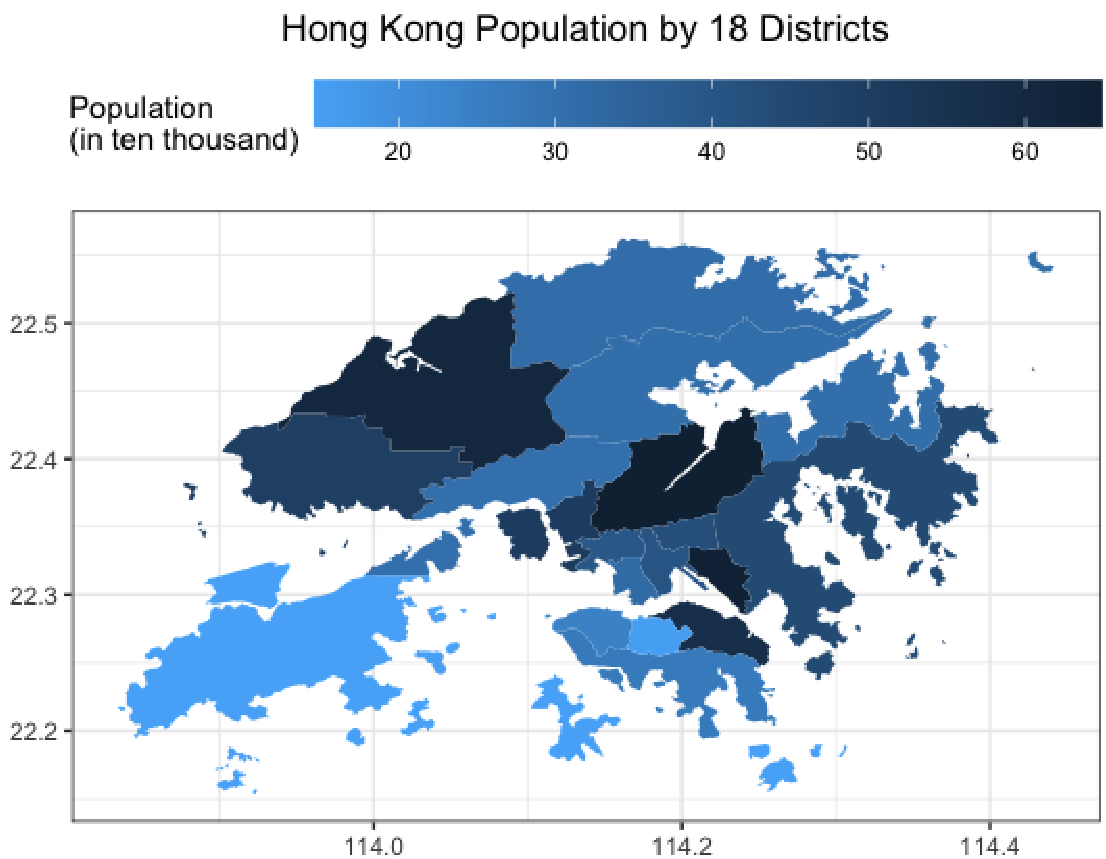


Fig.8 Task 5

[End of Task 5]