MTH 3270 Notes 2

2 Some R Programming Features (B.4)

2.1 The Logical Operations "And", "Or", and "Not"

2.1.1 Logical Operations and Compound Logical Expressions

- ***Logical operators*** (or ***Boolean operators***) correspond to "and", "or", and "not", and are written in R as:

```
!               # "Not"
&               # "And"
|               # "Or"
```

- These operate on `"logical"` (`TRUE` or `FALSE`) expressions and return `TRUE` or `FALSE` values. They're listed above in order of operator precedence (highest to lowest).

2.1.2 Logical Operations on Scalar Logical Expressions

- `&` returns `TRUE` if both expressions are `TRUE`, and it returns `FALSE` if at least one expression is `FALSE`:

```
TRUE & TRUE
```

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

```
TRUE & FALSE
```

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

- `|` returns `TRUE` if at least one of the expressions is `TRUE`, and it returns `FALSE` if both expressions are `FALSE`:

```
FALSE | TRUE
```

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

```
FALSE | FALSE

## [1] FALSE
```

- The negation operator, `!`, returns the "opposite" of a logical expression:

```
!TRUE

## [1] FALSE

!FALSE

## [1] TRUE
```

- As an example, to test whether a variable `x` lies *between* two numbers (60 and 70), we type:

```
x <- 63
x > 60 & x < 70

## [1] TRUE
```

  and to test whether it lies *outside* the range (60 to 70), we can type:

```
x < 60 | x > 70

## [1] FALSE
```

- The logical operators `&`, `|`, and `!` operate *elementwise* on `"logical"` vectors. For example:

```
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, FALSE)

## [1]  TRUE FALSE FALSE
```

- As another example, here are two vectors, `Syst` and `Diast`, containing systolic and diastolic blood pressures for five people:

```
Syst

## [1] 110 119 111 113 128

Diast

## [1] 70 74 88 74 83
```

  A blood pressure considered normal if the systolic level is less than 120 *and* the diastolic level is less than 80. To identify the people with normal blood pressures, we can type:

```
Syst < 120 & Diast < 80

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

or we can use `which()`:

```
which(Syst < 120 & Diast < 80)

## [1] 1 2 4
```

- Pay attention to the operator precedence for `&`, `|`, and `!`. More information can be found by typing:

```
? Syntax
```

Parentheses can be used to control the order of operations.

Section 2.1 Exercises

Exercise 1 Here are two variables `x` and `y`:

```
x <- 4
y <- 7
```

Guess what the result of each of the following will be, then check your answers.

a) `x > 2 & y == 7`

b) `x < 0 | y == 7`

c) `!(x < 0)`

Exercise 2 Recall that the operator precedence of the logical operators, from highest to lowest, is `!`, `&`, and `|`. The order of operations can be controlled using parentheses. Guess what the result of each of the following commands will be, then check your answers.

a) `10 < 20 | 15 < 16 & 9 == 10`

b) `(10 < 20 | 15 < 16) & 9 == 10`

c) `4 < 3 & (5 < 6 | 8 < 9)`

d) `(4 < 3 & 5 < 6) | 8 < 9`

Exercise 3 Recall that the logical operators operate *elementwise* on vectors. Guess what the result of each of the following will be, then check your answers.

a) `c(FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE)`

b) `c(FALSE, TRUE, FALSE) | c(TRUE, TRUE, FALSE)`

c) `!c(FALSE, TRUE, FALSE)`

Exercise 4 Recall that `is.na()` will identify missing values (`NA`s) in a vector. Here's a vector `x`:

```
x <- c(1, 2, NA, 6, 3, NA, 5)
```

(See what happens when you type `is.na(x)`).

Guess what the following command will return, then check your answer:

```
!is.na(x)
```

Exercise 5 Here's a vector `x`:

```
x <- c(1, 2, 6, 3, 5)
```

Guess what the result of each of the following commands will be, then check your answers.

a) `!(x > 4)`

b) `x > 4 & x < 6`

c) `!(x > 4 & x < 6)`

d) `x > 4 | x < 6`

Exercise 6 Here are two vectors, `Gender` and `Age`:

```
Gender <- c("m", "f", "m", "f", "f")
Age <- c(27, 34, 55, 21, 43)
```

Guess what each of the following commands will return, then check your answers.

a) `Gender == "m" & Age > 40`

b) `Gender == "m" | Age > 40`

2.2 User-Defined Functions

- We can create a ***user-defined function*** using:

```
function()          # Used to create user-defined functions.
return()            # Used within a function definition to terminate
                    # the function call and return a value.
```

- Here's a simple (mathematical) function $f(x)$:

$$f(x) = x^2$$

In R, we can represent this as user-defined function that takes an argument `x` and returns its square:

```
my.f <- function(x) {
  return(x^2)
}
```

We use user-defined functions just as we would built-in functions:

```
my.f(x = 2)

## [1] 4
```

We can look at the function definition by typing its name on the command line:

```
my.f

## function(x) {
##   return(x^2)
## }
```

- The general format for a *user-defined function* is:

```
my.fun <- function(arg1, arg2, ..., argk) {
     statement1
     statement2
          .
          .
          .
     statementq
     return(value)              # The value to be returned.  We could
   }                            # also just write value (without return()).
```

  Above,

  - `arg1`, `arg2`, ..., `argk` are argument names (that we're free to choose) for k arguments.
  - `statement1`, `statement2`, ..., `statementq` are a set of q statements (which may involve `arg1`, `arg2`, ..., `argk`).
  - `value` is a value (or expression that gives a value) to be returned by the function.

- If the body of the function consists of just a `value` (or expression), we can omit the curly brackets `{ }` as long as we write the entire function definition on a single line:

```
my.fun <- function(arg1, arg2, ..., argk) value
```

- As another example, here's a (mathematical) function that computes the average absolute value of two numbers x and y:

$$f(x, y) = \frac{|x| + |y|}{2},$$

  We can write this function in R as:

```
AvgAbsVal <- function(x, y) {
  avg <- (abs(x) + abs(y)) / 2
  return(avg)
}
```

  We can now call `AvgAbsVal()`, passing it values via the arguments `x` and `y`:

```
AvgAbsVal(x = -4, y = 2)

## [1] 3
```

- In fact, we don't need to use `return()`. In the absence of a `return()` statement, R returns the value of any expression that appears by itself as the last line of the function definition:

```
AvgAbsVal <- function(x, y) {
  avg <- (abs(x) + abs(y)) / 2
  avg
}
```

2.2.1 Formal Arguments and Actual Arguments

- A function's arguments are sometimes referred to as ***formal*** arguments. Values passed to the function are referred to as ***actual*** arguments.
- For example, below, `x` is a *formal* argument, and `z` is an *actual* argument:

```
g <- function(x) {
  x + 1
}
```

```
z <- 5
g(x = z)              # x is a formal argument, z is an actual argument

## [1] 6
```

2.2.2 Specifying Default Values for a Function's Arguments

- We can specify default values for one or more of a function's arguments by specifying `arg = expr` in the function definition:

```
my.fun <- function(arg1, arg2 = expr2, ..., argk = exprk) {
     statement1
     statement2
          .
          .
          .
     statementq
     value
  }
```

- For example, below we define `AvgAbsVal()` so that the default value for `y` is `NULL`.

```
AvgAbsVal <- function(x, y = NULL) {
  if (is.null(y)) {
    return(abs(x))
  } else {
    avg <- (abs(x) + abs(y)) / 2
    return(avg)
  }
}
```

Note the use of the `if()` statement. Now, if a value for `y` isn't passed to `AvgAbsVal()`, it uses `NULL` and returns `abs(x)`:

```
AvgAbsVal(x = -120)

## [1] 120
```

2.2.3 Variable Number of Arguments Using "..."

- Functions can be written to take a *variable number* of arguments. The argument name `...` in the function definition will match any number of arguments.

  Within the body of the function, we can refer to `...` as if it was the name of a variable.

- For example, here's a function that returns the mean of all the values in an arbitrary number of vectors:

```
mean.of.all <- function(...) {
    mean(c(...))
}
```

If `us.sales`, `europe.sales`, and `other.sales` were vectors, the command

```
mean.of.all(us.sales, europe.sales, other.sales)
```

would combine them and take the mean of the combined data. The effect of `c(...)` is as if `c()` were called with the same three arguments, `us.sales`, `europe.sales`, and `other.sales`, that were passed to `mean.of.all()`.

- Many of R's built-in functions take a variable number of arguments. For example look at the help pages for `list()` and `c()` by typing:

```
? list
? c
```

2.2.4 Printing Warning or Error Messages Using `warning()` or `stop()`

- The following functions are useful for terminating a function call or just printing a warning message:

```
stop()          # Terminate a function call and print an error message.
warning()       # Print a warning message (without terminating the
                # function call).
```

- `stop()` and `warning()` are usually used in `if()` statements within functions.
- `stop()` terminates a function call (without returning a value) and prints an error message. Here's an example:

```
my.ratio <- function(x, y) {
    if (y == 0) stop("Cannot divide by 0")
    return(x/y)
}
```

An attempt to pass the value `0` for `y` now results in the following:

```
my.ratio(x = 3, y = 0)

## Error in my.ratio(x = 3, y = 0):  Cannot divide by 0
```

(Note that the last line, `return(x/y)`, was never encountered during the call to `my.ratio()`.)

- `warning()` just prints a warning message to the screen without terminating the function call. Here's an example:

```
my.ratio <- function(x, y) {
    if (y == 0) warning("Attempt made to divide by 0")
    return(x/y)
}
```

Now when we pass the value `0` for `y`, the function call isn't terminated (`Inf` is returned), but we get a warning message:

```
my.ratio(x = 3, y = 0)

## Warning in my.ratio(x = 3, y = 0):  Attempt made to divide by 0

## [1] Inf
```

Section 2.2 Exercises

Exercise 7 Here's a function:

```
f1 <- function(x) {
    y <- x + 1
    return(y)
  }
```

If we replace `return(y)` by just `y`:

```
f2 <- function(x) {
    y <- x + 1
    y
  }
```

does the function do the same thing?

Exercise 8 In the code below, which argument (`x` or `z`) is the *formal* argument and which is the *actual* argument?

```
g <- function(x) {
  x^2 - 1
}
```

```
z <- 2
g(x = z)
```

```
## [1] 3
```

Exercise 9

a) Write a function that takes two arguments, `x` and `y`, and returns their *relative difference*, defined as

$$f(x, y) = \left| \frac{x - y}{y} \right|,$$

where $|\cdot|$ is the absolute value. Test your functions by passing it a few different values for `x` and `y`.

b) What happens when you pass it the value `y = 0`? What about when you pass it `x = 0` and `y = 0`.

c) Rewrite your function so that it specifies a default value of 1 for `y`.

Exercise 10 Write a function that takes a vector argument `x` and returns a *list* containing the mean, median, standard deviation, and range of `x`. Use `mean()`, `median()`, `sd()`, and `range()`.

Exercise 11 Look at the help page for `list()` by typing

```
? list
```

What do the `...`'s mean when it says `list(...)` under Usage?

Exercise 12

a) Write a function that takes two vectors `x` and `y` and returns the maximum value in the two vectors combined. **Hint**: Use `max()` with `c(x, y)`.

b) Modify your function so that it uses `...` to take a *variable number* of vectors as arguments, and returns the maximum value in all the vectors combined.

3 Data Visualization (Graphics) (2)

3.1 Introduction

- Two important R graphics packages:
  - The `"graphics"` package in base R (discussed in Class Notes 1).
  - The `"ggplot2"` package (discussed below).
- A set of variables can be displayed many ways. The goal of visualization is to **convey information clearly** (E. Tufte).

Data Set: `diamonds`

The `diamonds` data set (in `"ggplot2"`) contains the prices and other attributes of almost 54,000 round cut diamonds. The ten variables are:

`price`	price in US dollars ($326–$18,823).
`carat`	weight of the diamond (0.2–5.01)
`cut`	quality of the cut (Fair, Good, Very Good, Premium, Ideal).
`color`	diamond color, from J (worst) to D (best).
`clarity`	a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best)).
`x`	length in mm (0–10.74).
`y`	width in mm (0–58.9).
`z`	depth in mm (0–31.8).
`depth`	total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43–79).
`table`	width of top of diamond relative to widest point (43–95).

- **Example**: In the `diamonds` data set, how prevalent are the different `cut` classes? Here are three ways of visualizing that information:

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

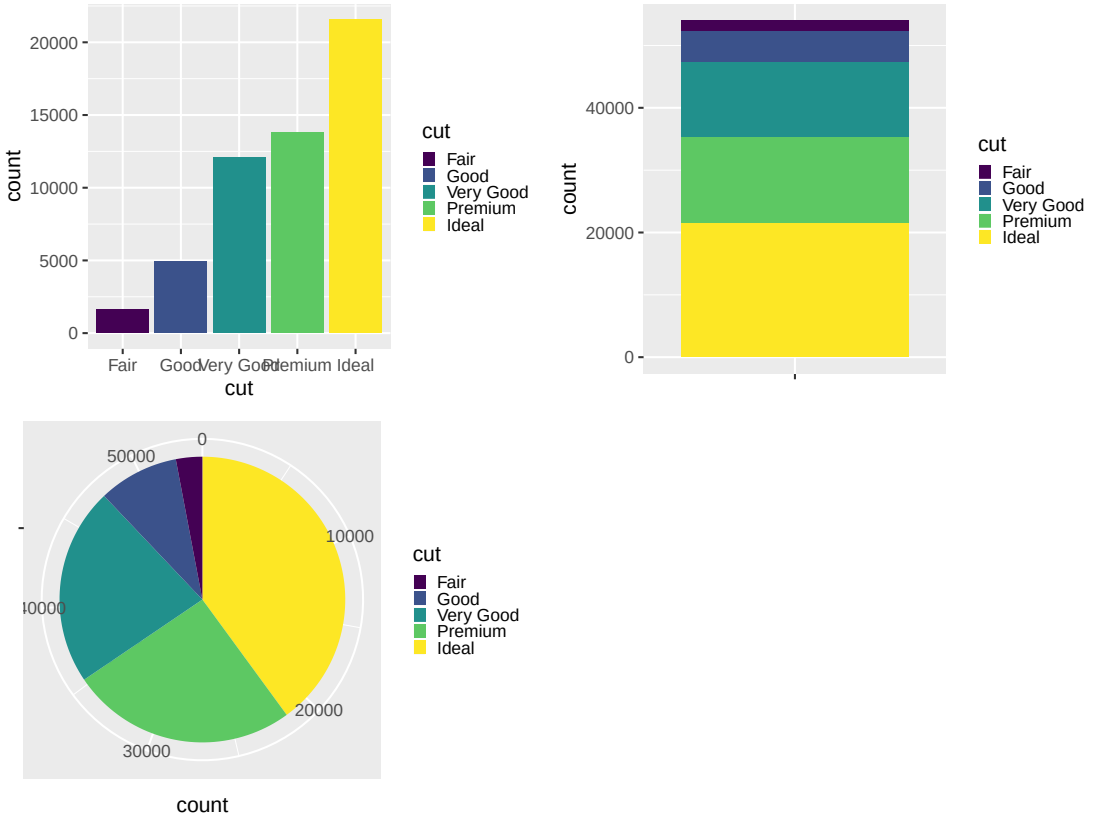


Figure 1

```
## Stacked bar plot
ggplot(data = diamonds)  +
  geom_bar(mapping = aes(x = "", fill = cut))  +
  xlab(label = NULL)
```

```
## Pie chart
ggplot(data = diamonds)  +
  geom_bar(mapping = aes(x = "", fill = cut))  +
  xlab(label = NULL) +
  coord_polar(theta = "y")
```

We can see from the graphs that `Ideal` is most prevalent class and `Fair` the least.

- A challenge is displaying **multiple variables** at the same time. One way to do it is to map values of variables to **aesthetic properties** such as color, shape, and size.

Data Set: `mpg`

The `mpg` data set (bundled with `"ggplot2"`) includes information about the fuel economy of popular car models from 1999 to 2008.

`manufacturer`	Categorical variable (`"audi"`, `"chevrolet"`, `"dodge"`, etc.).
`model`	Categorical variable indicating the model of car. The 38 models had a new edition every year between 1999 and 2008.
`displ`	is the engine displacement in liters.
`year`	The year of manufacture.
`cyl`	The number of cylinders.
`trans`	The type of transmission.
`drv`	The drivetrain: front wheel (`"f"`), rear wheel (`"r"`), or four wheel (`"4"`).
`cty`	Miles per gallon (mpg) for city driving.
`hwy`	Miles per gallon (mpg) for highway driving.
`fl`	The fuel type.
`class`	Categorical variable describing the "type" of car: `two-seater`, `SUV`, `compact`, etc.

- **Example**: Fig. 2 shows three ways of displaying **three** variables, `displ`, `hwy`, and `drv`:

```
## scatterplot with drv as different colors
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = drv))
```

```
## scatterplot with drv as different shapes
ggplot(data = mpg) +
  geom_point(aes(x = displ, y = hwy, shape = drv))
```

```
## smooth lines with drv as different colors
ggplot(data = mpg) +
  geom_smooth(aes(x = displ, y = hwy, color = drv), se = FALSE)
```

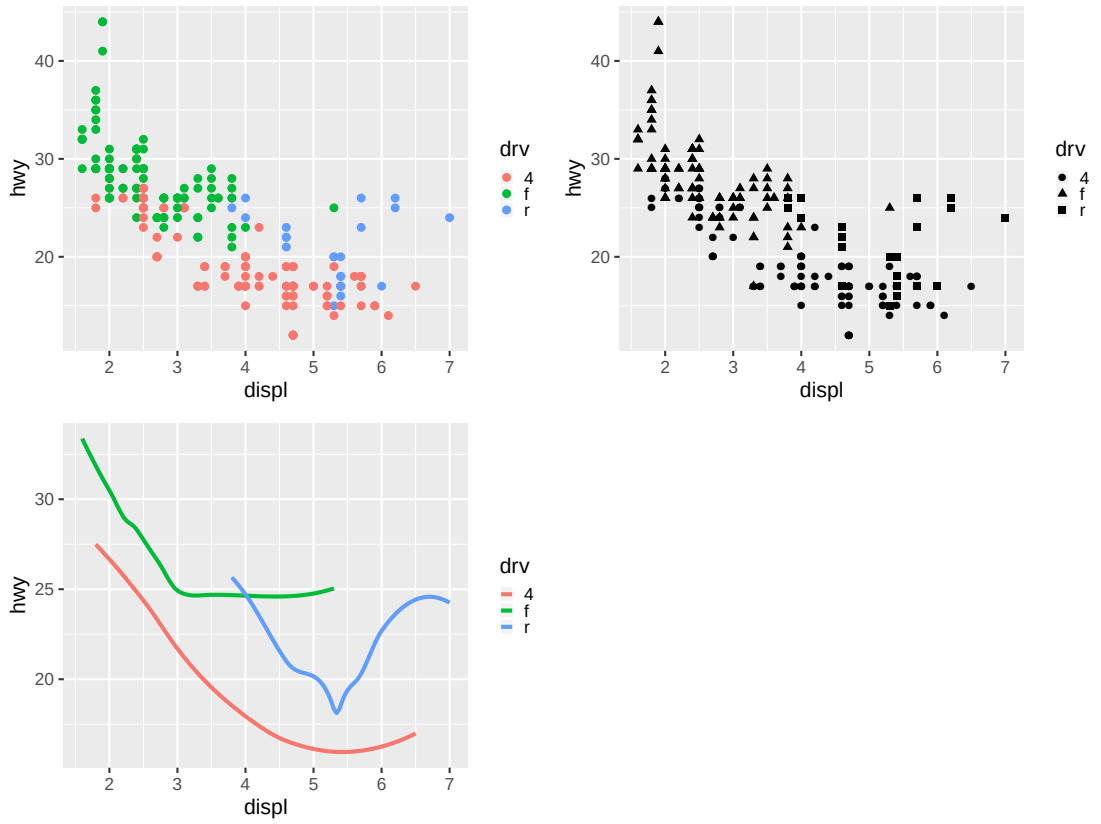


Figure 2

- Another way to display three variables (when one of them is categorical) is via **faceting**.
- **Example**: In Fig. 3, the **three** variables, `displ`, `hwy`, and `drv`, are displayed using **faceting**:

```
## Scatterplots with faceting
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(facets = ~ drv, nrow = 1, ncol = 3)
```

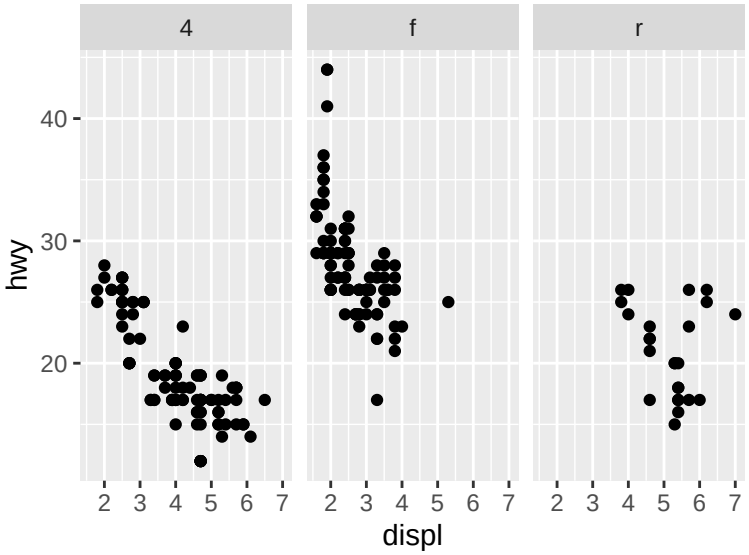


Figure 3

3.2 A Taxonomy for Data Graphics

- Here's a **taxonomy** for characterizing statistical graphs (N. Yau). Graphs are characterized by:
  - ***Visual cues***: For portraying **values** of variables (both numerical and categorical).

Human Ability to **Discern Differences**	**Visual Cue (and Type of Variable)**
Easiest to Discern	Position (numerical)
↓	Length (numerical)
	Angle (numerical)
	Direction (numerical)
↓	Shape (categorical)
	Area (numerical)
	Volume (numerical)
↓	Shade (either)
Hardest to Discern	Color (either)

  - ***Coordinate system***: For determining the positions of points or other geometric elements in the graph.
    - Cartesian (x and y axes)
    - Polar (angle θ and radius r)
    - Geographic (latitude and longitude)
  - ***Scale***: For determining how distances in the graph translate to differences in the value of the variable.

    - Numerical (linear or logarithmic)
    - Categorical (nominal or ordinal)
    - Time (linear or cyclical)
  - ***Context***: Title, axis labels, legend, etc. which make the graph *meaningful*.

- **Example**:
  - In Fig. 1, upper left (bar plot):
    - **Visual cues**: *position* (along the y axis) and *color*.
    - **Coordinate system**: *Cartesian* (but the x coordinate is meaningless).
    - **Scale**: *numerical* (on the y-axis) and *categorical* (on the x-axis and as colors).
    - **Context**: *legend* and *axis labels*.
  - In Fig. 1, upper right (stacked bar plot):
    - **Visual cues**: *length* (along the y axis) and *color*.
    - **Coordinate system**: *Cartesian* (but the x coordinate is meaningless).
    - **Scale**: *numerical* (along the y axis) and *categorical* (as colors).
    - **Context**: *legend* and *y-axis label*.
  - Fig. 1, bottom left (pie chart):
    - **Visual cues**: arc *length* (or *angle*) and *color*.
    - **Coordinate system**: *polar* (but the r coordinate is meaningless).
    - **Scale**: *numerical* (for the angle θ) and *categorical* (as colors).
    - **Context**: *legend* and *horizontal axis label*.

- **Example**:
  - In Fig. 2, upper left (scatterplot):
    - **Visual cues**: *position* (along the x and y axes) and *color*.
    - **Coordinate system**: *Cartesian*.
    - **Scale**: *numerical* (on the two axes) and *categorical* (as colors).
    - **Context**: *legend* and *axis labels*.
  - In Fig. 2, upper right (scatterplot):
    - **Visual cues**: *position* (along the x and y axes) and *shape*.
    - **Coordinate system**: *Cartesian*.
    - **Scale**: *numerical* (on the two axes) and *categorical* (as shapes).
    - **Context**: *legend* and *axis labels*.
  - In Fig. 2, bottom left (smooth line plot):
    - **Visual cues**: *direction* (of the lines) and *color*.
    - **Coordinate system**: *Cartesian*.
    - **Scale**: *numerical* (on the two axes) and *categorical* (as colors).
    - **Context**: *legend* and *axis labels*.

Section 3.2 Exercises

Exercise 13 Load the `"ggplot2"` package (which contains the `diamonds` data set):

```
library(ggplot2)
```

Run each code chunk below, and for each graph indicate:

- The **visual cues** that are used.
- The **coordinate system** that's used.
- The **scales** that are used.
- How **context** is provided.

a)
```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price, color = cut)) +
  ggtitle("Diamond Price vs Weight")
```

b)
```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price)) +
  geom_smooth(mapping = aes(x = carat, y = price), se = FALSE) +
  ggtitle("Diamond Price vs Weight")
```

c)
```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = price)) +
  ggtitle("Histogram of Diamond Prices")
```

Exercise 14 This exercise uses the `mpg` data set. Run each code chunk below, and for each graph indicate:

- The **visual cues** that are used.
- The **coordinate system** that's used.
- The **scales** that are used.
- How **context** is provided.

a)
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, size = cyl)) +
  ggtitle("Highway MPG vs Displacement")
```

b)
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = cty, size = hwy,
                           color = cyl, shape = drv)) +
  ggtitle("City MPG vs Displacement")
```

3.3 A Grammar for Graphics with "ggplot2" (3)

3.3.1 Introduction

- We'll see now how to make graphs using the "ggplot2" package. Typing:

```
help(package = "ggplot2")
```

  shows a list of the functions (and data sets) in "ggplot2".

- "ggplot2" (H. Wickham) is based on the so-called ***grammar of graphics*** (L. Wilkinson).

  The **grammar of graphics** is slightly different from the **taxonomy** described above. It uses the following components shared by statistical graphs:

  - A ***data set*** containing the *variables* to be plotted.
  - ***Aesthetic mappings*** that associate *variables* with *aesthetic properties* (aka **visual cues**) such as position, color, size, and shape.
  - At least one ***layer*** of ***geometric objects*** such as points, bars, or lines.
  - A ***coordinate system*** and, for each aesthetic mapping, a ***scale*** that associates *values of the variable* to *values of the aesthetic property*. The scale is given in legends, x and y axis annotations, etc.
  - Optionally, a ***statistical transformation*** to be displayed (instead of the raw data), for example category *counts* for a bar plot, bin *counts* for a histogram, etc.
  - Optionally, a ***facet*** specification (usually no faceting).

- **Every "ggplot2" plot command requires**:

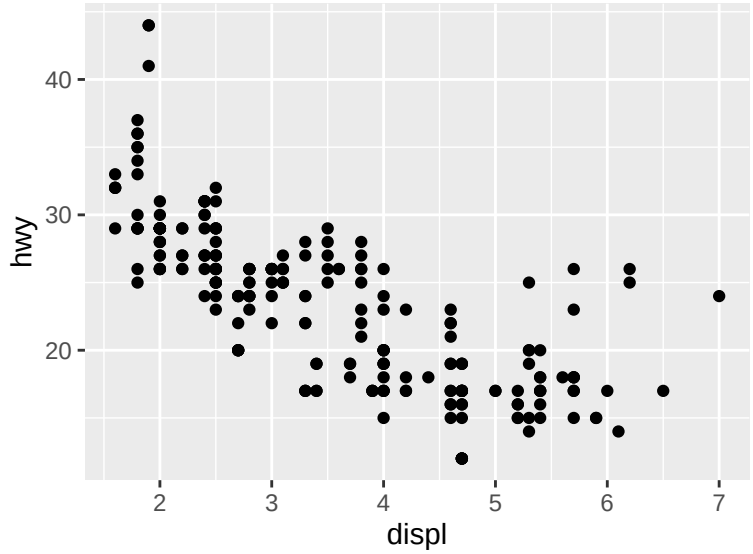
  1. A **data set**, usually specified in `ggplot()`, containing the variables to be plotted.
  2. One or more **aesthetic mappings** associating variables with aesthetic properties (visual cues), specified in `aes()`.
  3. At least one **layer** of **geometric objects**, added via a `geom()` function.

```
ggplot()      Specify a data set and initialize a plot.
aes()         Specify aesthetic mappings.  Used within ggplot() or a
              geom() function.
geom_*()      Add a layer of geometric objects (e.g. geom_bar(),
              geom_point(), etc.).
```

- The graph is initiated using `ggplot()`, and **layers** of **geometric objects** are added using one ore more of the `geom()` functions.

- **Example**:

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

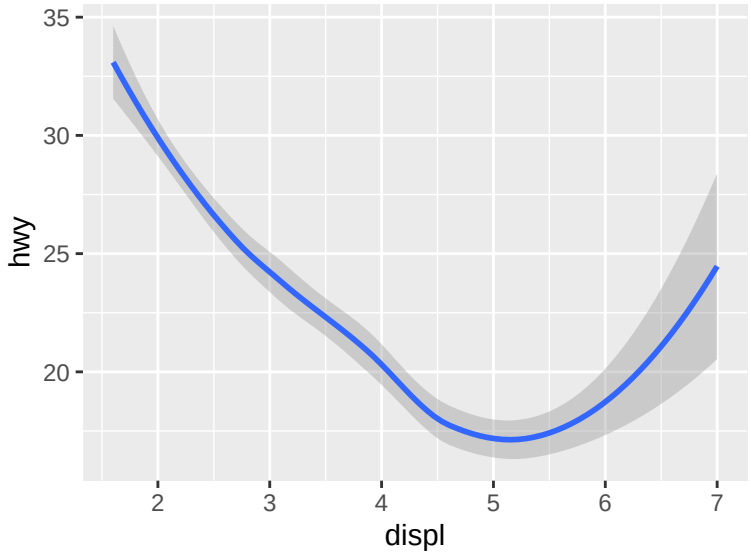


In the command above:

  - The **data set** is `mpg`.
  - The two **aesthetic mappings** map `displ` to the x-axis and `hwy` to the y-axis.
  - The **layer** consists of points added via `geom_point()`.

- **Example**:

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

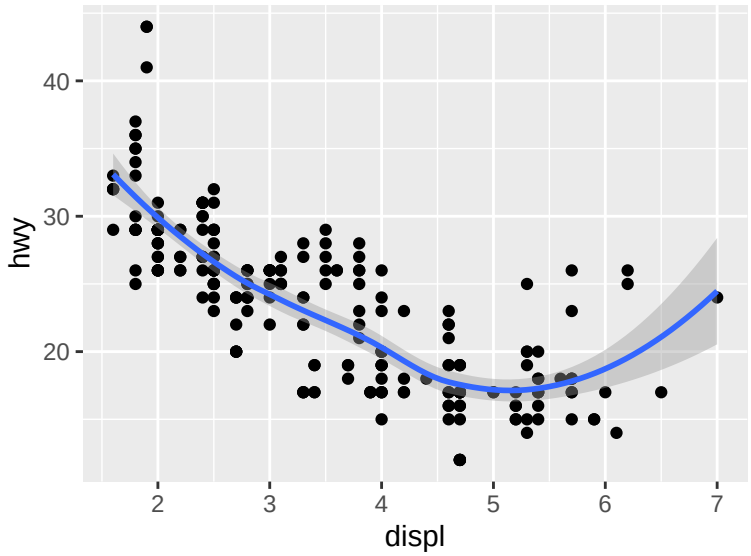


In the command above:

- The **data set** is `mpg`.
- The two **aesthetic mappings** map `displ` to the x-axis and `hwy` to the y-axis.
- The **layer** consists of a smooth line added via `geom_smooth()`.

- We can add *more than one* **layer** to a single graph using different `geom()` functions.
- **Example**:

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



- Here's a basic **graphing template**:

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

(Replace the bracketed sections with a **data set**, `geom()` **function**, and **aesthetic mappings**. Add layers using additional `geom()` functions.)

- Threes comments:

1. `ggplot()` merely initializes the graph by setting up a **coordinate system**.
2. The **data set** is usually specified in `ggplot()`, but it could also be specified in the `geom()` function.
   - If it's specified in `ggplot()`, it's used as the default for *all* **layers**.
   - If it's specified in the `geom()` function, it only applies to that *particular* **layer**.
3. The **aesthetic mapping** is usually specified in the `geom()` function, but it could also be specified in `ggplot()`.
   - If it's specified in `ggplot()`, it's used as the default for *all* **layers**.
   - If it's specified in the `geom()` function, it only applies to that *particular* **layer**.

- Other components of the *grammar* (**coordinate system** and **scale**, **statistic**, and **faceting**) aren't required in the `"ggplot2"` plot command – they have reasonable default settings. But they can be altered with the following functions.

```
coord_*()     Set the coordinate system (e.g. coord_polar()
              for polar coordinates, i.e. pie charts, and coord_flip()
              to flip the horizontal and vertical axes).
scale_*()     Define the scale (e.g. scale_y_log10() for a log
              scale on the y-axis), and control its features.
stat_*()      Specify a statistic to be plotted (e.g. stat_count() for
              a bar plot and stat_function() to graph a function).
facet_wrap()  Specify faceting configuration.
```

- **Titles** and **axis labels** can be added using these functions.

```
ggtitle()         Add a main title.
xlab(), ylab()    Add an x-axis or y-axis label.
labs()            Add a title to the legend.
```

Section 3.3 Exercises

Exercise 15 `ggplot()` merely sets up a coordinate system. Type the following command, and describe what you see:

```
ggplot(data = mpg)
```

Exercise 16 The data set is usually specified in `ggplot()`, but it could also be specified in the `geom()` function.

If it's specified in `ggplot()`, it's used for all layers. If it's specified in `geom()`, it only applies to that layer.

Verify that the following commands both make the same scatterplot:

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

```
## Specify data in geom() function:
ggplot() +
  geom_point(data = mpg, mapping = aes(x = displ, y = hwy))
```

Exercise 17 The aesthetic mapping is usually specified in the `geom()` function, but it could also be specified in `ggplot()`.

If it's specified in `geom()`, it only applies to that layer. If it's specified in `ggplot()`, it's used for all layers.

Verify that the following commands both make the same scatterplot:

```
## Specify aesthetics in geom() function:
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```

```
## Specify aesthetics in ggplot():
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point()
```

Exercise 18 This exercise uses the `mpg` data set.

a) Guess what the `ggtitle()`, `xlab()`, and `ylab()` commands do to the scatterplot below. Then check your answers.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  ggtitle(label = "Highway MPG vs Displacement") +
  xlab(label = "Displacement") +
  ylab(label = "Highway MPG")
```

b) Guess what the `labs()` command does to the scatterplot below. Then check your answer.

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

Exercise 19 This exercise uses the `mpg` data set again.

a) Make a scatterplot of `hwy` (on the y-axis) versus `cyl` (x-axis). Report your R command(s).

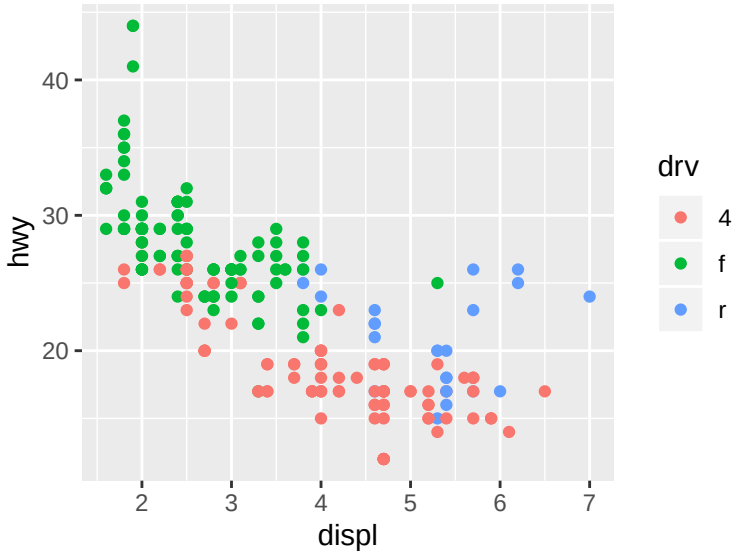
b) Reproduce the scatterplot of Part *a*, but now add a second *layer* to the plot using `geom_smooth()`. Report your R command(s).

c) Make a scatterplot of `class` (y-axis) versus `drv` (x-axis)? What happens? Why is the plot not useful?

3.3.2 More on Aesthetic Mappings

- Here's a graph with *three* **aesthetic properties** (`x` position, `y` position, and `color`) associated with the *variables* `displ`, `hwy`, and `drv`:

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



The **scale** associating a specific color to each `drv` class is created automatically (as is the legend).

- The set of **aesthetic mappings** that are available will depend on what type of **geometric object** is being plotted. For example, the `linetype` aesthetic would be available for a variable in `geom_line()` but not in `geom_point()`.

  To see which **aesthetic mappings** are available for a given **geometric object**, look at the help page for the `geom()` function, e.g.

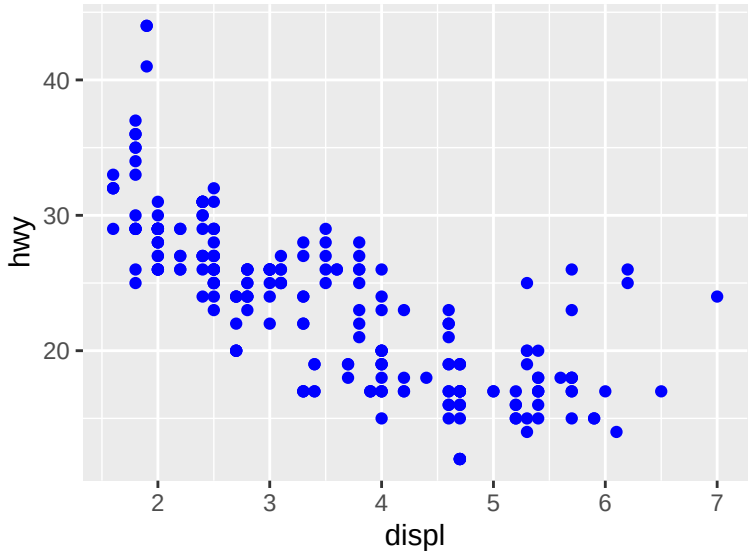
```
? geom_point
```

  From the help page for `geom_point()`, we can see that the `x`, `y`, `color`, `shape`, and `size` **aesthetics** are available (among others).

- Instead of mapping **aesthetic properties** to values of a *variable*, we can set them *manually* by passing a value for the **aesthetic** via a named argument in the `geom()` function, *outside* `aes()`.

  For example, to make all of the points in the plot blue, we type:

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



We could also *manually* set points to a specific `size` (in mm) or a `shape` (as numeric identifier from 0 to 24).

For more information, look at the so-called **vignette** for `"ggplot2"`s aesthetic specifications by typing:

```
vignette("ggplot2-specs")
```

Section 3.3 Exercises

Exercise 20 This exercise concerns changing the **aesthetic mappings** in a plot. First produce the following plot:

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

a) Reproduce the plot, but with `cyl` mapped to the `size` aesthetic. How does the plot differ from the one above?

b) What happens when you try to map `cyl` to the `shape` aesthetic? Try it, and report what happens.

c) What happens when you map the `"logical"` values in `displ<5` to an aesthetic property? Try it, and describe the plot in a sentence or two.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = displ < 5))
```

d) What happens when you map the same variable to multiple aesthetics? Try the both of the following.

- Alter the code below so that `hwy` gets mapped to *both* `y` *and* `color`. Describe the plot in a sentence or two.

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

- Now alter the code so that `hwy` gets mapped to `y`, `color`, *and* `size`. Describe the plot in a sentence or two.

Exercise 21 To make all of the points in a plot blue, we can set the `color` aesthetic of the points *manually* in the `geom()` function, *outside* `aes()`, by typing:

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

A common mistake is doing it *inside* `aes()`. What happens when you type the following?

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

3.3.3 More on Layers

- Here are some of the most important `geom()` functions for adding **layers** of **geometric objects** to a plot:

```
geom_point()      Add scatterplot points
geom_smooth()     Add smoothed scatterplot line
geom_histogram()  Add a histogram
geom_density()    Add a density curve
geom_bar()        Add bars, with heights corresponding to the number
                  of cases in each group.
geom_col()        Add bars, with heights corresponding to values in
                  the data.
geom_boxplot()    Add boxplot
geom_line()       Add line through points in the order in which they
                  appear on the x-axis.
geom_path()       Add line through points in the order in which they
                  appear in the data set.
geom_text()       Add text directly to the plot.  The position
                  and label of the text are specified via aes(x, y,
                  label).
geom_label()      Add text with a rectangle behind it, making it
                  easier to read.  The position
                  and label of the text are specified via aes(x, y,
                  label).
```

For a full list, type:

```
help(package = "ggplot2")
```

- Recall that typing:

```
ggplot(data = mpg)
```

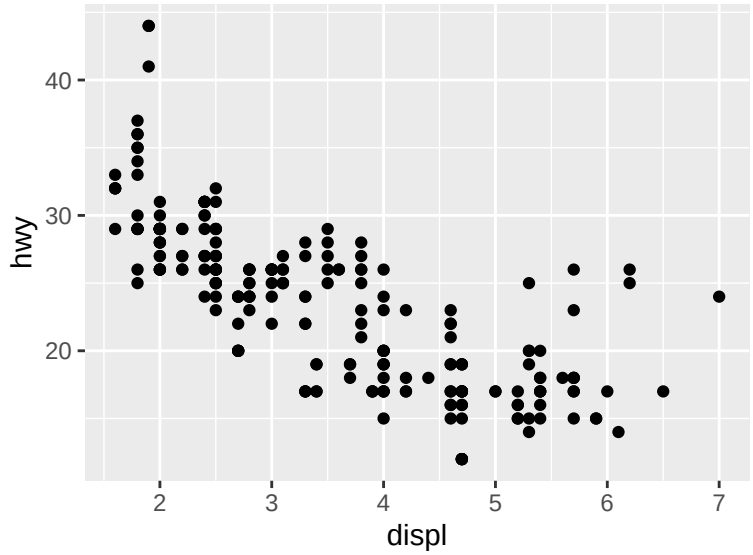
just creates a blank graph. **Layers** of **geometric objects** are added using one or more `geom()` functions.

- We can *change* the type of **layer** by using a different `geom()` function. For example, below, first we save the graph (without *any* **layers**) as an object `g`:

```
g <- ggplot(data = mpg)
```

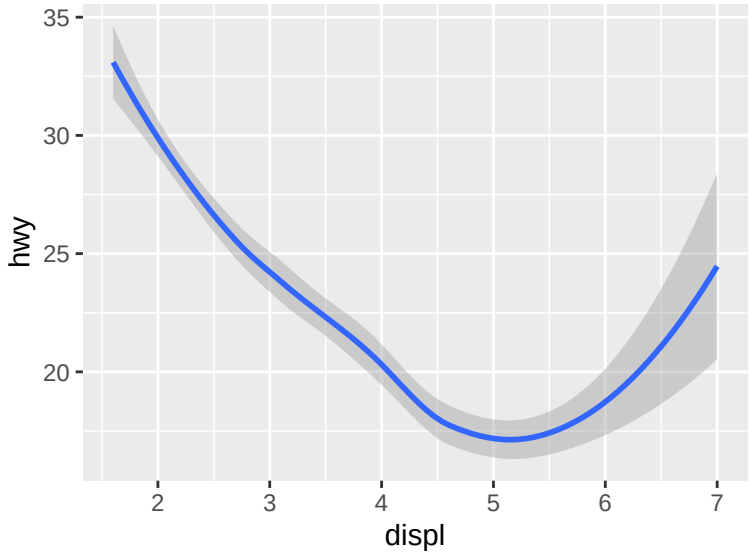
Now we *add* a **layer** of *points*:

```
g + geom_point(mapping = aes(x = displ, y = hwy))
```



and here we use a different `geom()` function to *change* the **layer** to *a smooth line*:

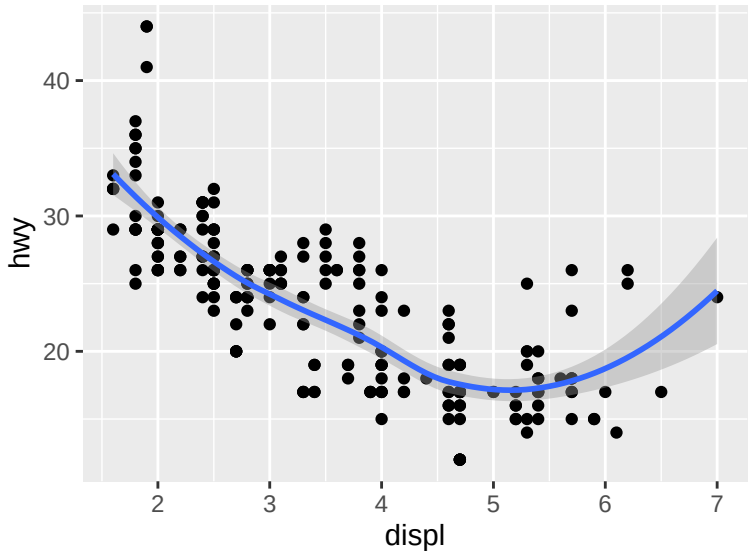
```
g + geom_smooth(mapping = aes(x = displ, y = hwy))
```



- Below we add *more than one* **layer** to a single graph.

  Notice we set the **aesthetic mapping** in `ggplot()` (instead of in the `geom()` functions) so that it applies to *both* **layers**:

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

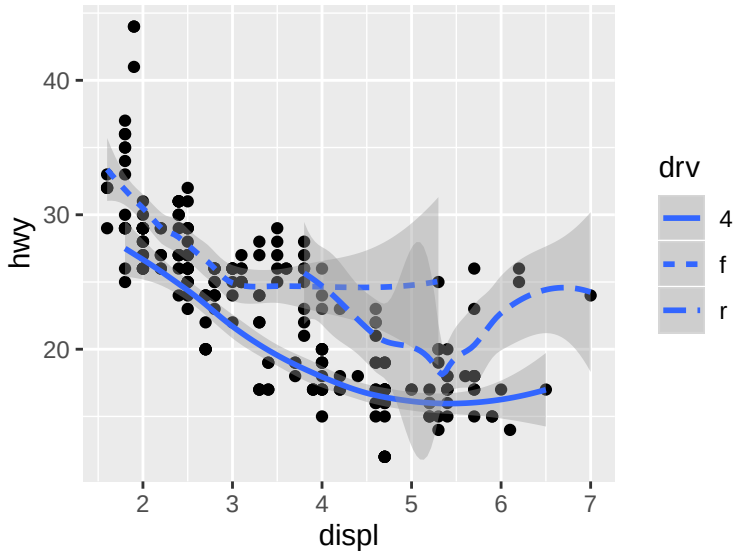


- Recall that the set of **aesthetic mappings** available depends on the type of **geometric object** being plotted.

For example, `linetype` is available for `geom_smooth()` but not for for `geom_point()`.

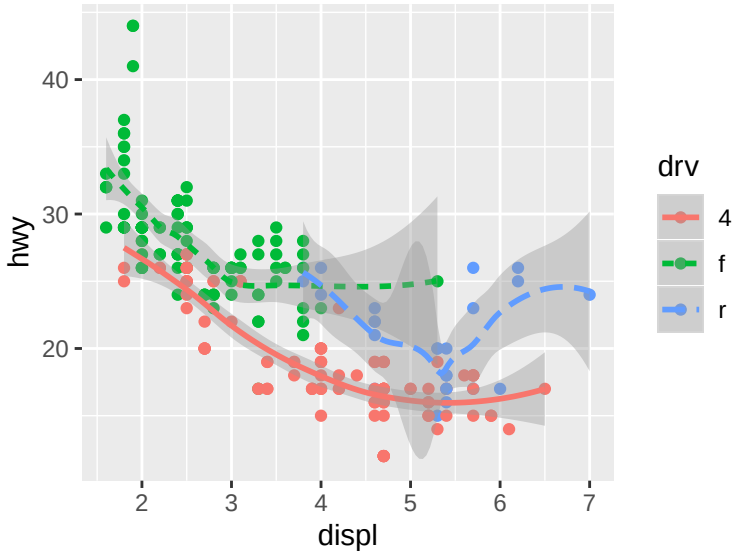
Below, we display a third variable, `drv`, via the `linetype` **aesthetic**. This plots a separate line for each `drv` class of cars:

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



- To see the different `drv` groups better, we can use the `color` **aesthetic** to distinguish them:

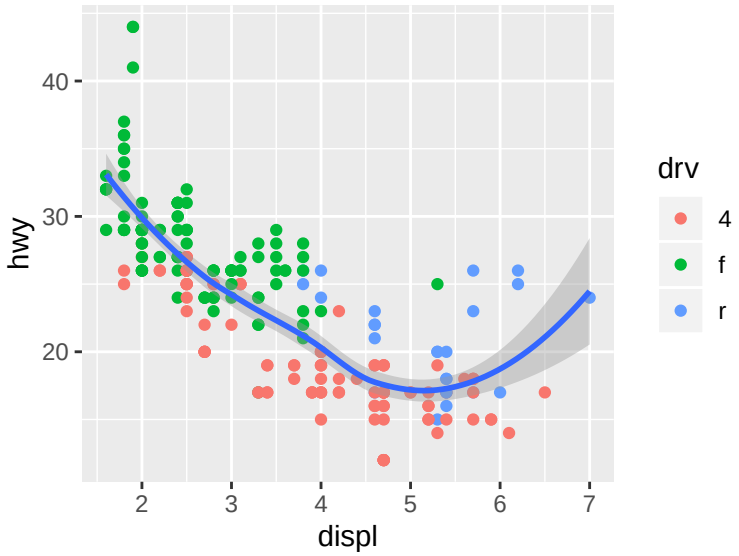
```
ggplot(data = mpg, aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  geom_smooth(mapping = aes(linetype = drv))
```



- Recall that **aesthetic mappings** specified in a `geom()` function apply to *that layer only*.

  Below, the `color` **aesthetic** applies to the *points* but not the smooth *line*:

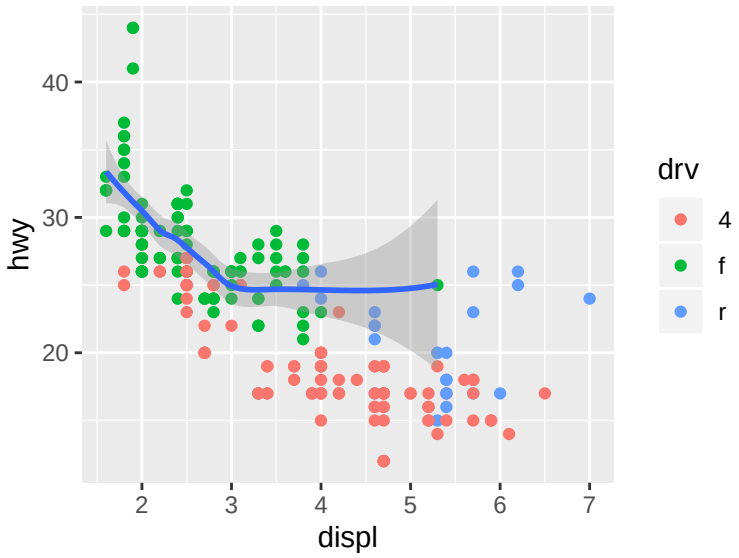
```
ggplot(data = mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = drv)) +
  geom_smooth()
```



- We can even use a *different* **data set** for each **layer** of **geometric objects**. To do so, specify the **data sets** via the `geom()` functions (instead of in `ggplot()`).

Below, we use only a subset of the `mpg` data set for the smooth line. (The `filter()` function from the `"dplyr"` package extracts just the `"f"` `drv` class of cars from `mpg`):

```
library(dplyr)
ggplot() +
  geom_point(data = mpg, aes(x = displ, y = hwy, color = drv)) +
  geom_smooth(data = filter(mpg, drv == "f"), aes(x = displ, y = hwy))
```



Section 3.3 Exercises

Exercise 22 Recall that if an aesthetic mapping specified in `ggplot()`, it's used for all layers, but if it's specified in `geom()`, it only applies to that layer.

Look at the graph that the following commands produce.

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

(Note that `drv` is mapped to `colors` for both the points *and* the lines.)

a) Try to predict what the following graph will look like, then check your answer. Report your findings.

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

b) Now try to predict what the following graph will look like, then check your answer. Report your findings.

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

Exercise 23 This exercise concerns adding and altering **layers** of a plot. Recall that `ggplot()` merely sets up a coordinate system:

```
ggplot(data = mpg)
```

a) Modify the command above so that the following **layer** is added to the plot. Report what you see.

```
geom_boxplot(aes(x = drv, y = displ))
```

b) What happens if you use `geom_col()` in place of `geom_boxplot()` in Part *b*? Report what you see.

(Note that `geom_col()` makes a bar plot of the *sums* of the `displ` values for each `drv` group. To plot the *means*, see the examples in the help page for `geom_col()`.)

Exercise 24 Univariate numerical data are usually graphed in a histogram or density plot. In either case, the variable is mapped to `x` via `aes()` (and there's no `y` variable).

a) Use `ggplot()` and `geom_histogram()` to make a histogram of `hwy` (from the `mpg` data set). Report your R command(s).

b) Replace `geom_histogram()` in your command from Part *a* by `geom_density()` to make a density plot `hwy`. Report your R command(s).

c) To make a plot with *two* **layers**, the histogram from Part *a* and the density plot from Part *b*, we need rescale that histogram so that the *y*-axis scale is the same as the density curve. Make the following graph and describe the result:

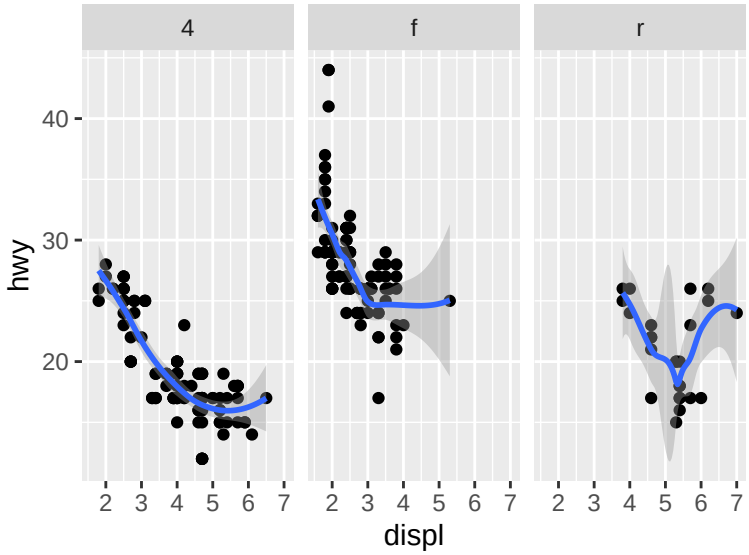
```
ggplot(mpg, mapping = aes(x = hwy)) +
  geom_histogram(mapping = aes(y = stat(density))) +
  geom_density()
```

3.3.4 More on Faceting

- Another way to display a third *categorical* variable (such as `drv`) is via **faceting**, which results in a separate plot for each subset of the data (e.g. for each `drv` class):

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

```
  geom_smooth() +
  facet_wrap(facets = ~ drv, nrow = 1, ncol = 3)
```

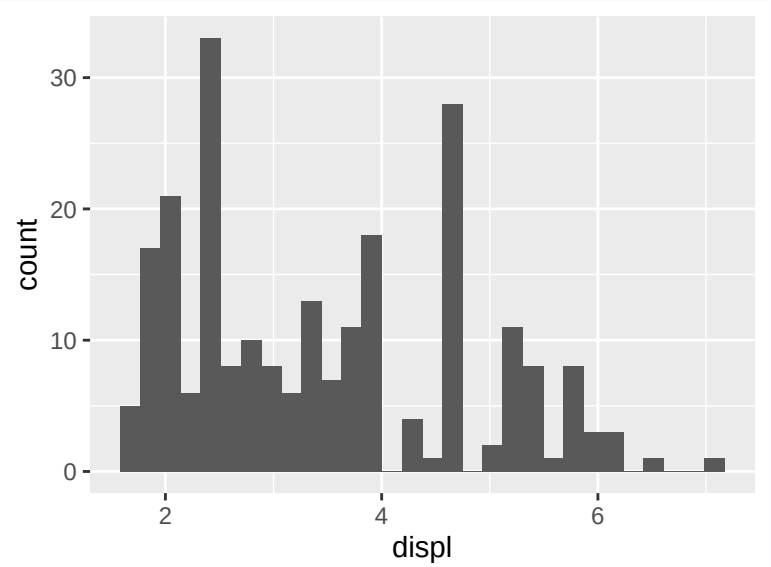


Above, we pass a so-called *formula*, `~ drv`, to `facet_wrap()` via the `facets` argument.

Section 3.3 Exercises

Exercise 25 This exercise concerns **faceting** for displaying a third *categorical* variable. Here's a histogram of the `displ` variable from the `mpg` data set:

```
ggplot(data = mpg) +
  geom_histogram(aes(x = displ))
```



a) One way to display the third (*categorical*) variable `drv` is via either the `color` or `fill` *aesthetic*. Look at the following two graphs.

```
ggplot(data = mpg) +
  geom_histogram(aes(x = displ, color = drv))
```

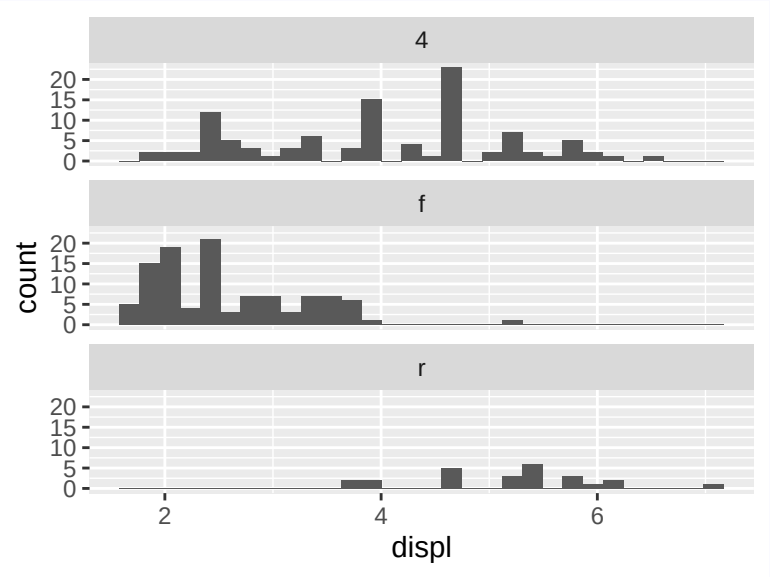
```
ggplot(data = mpg) +
  geom_histogram(aes(x = displ, fill = drv))
```

Which graph do you prefer?

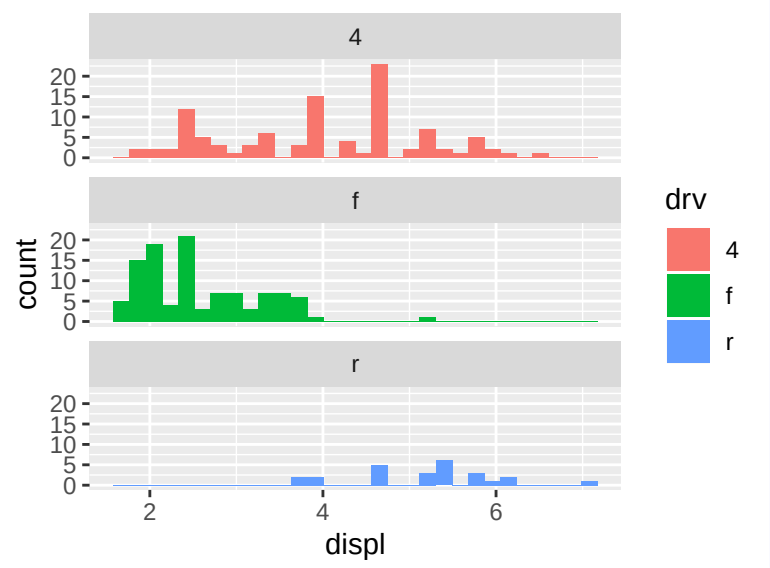
b) Alter the following code chunk,

```
ggplot(data = mpg) +
  geom_histogram(aes(x = displ))
```

using `facet_wrap()`, with `facets = ~ drv`, to duplicate the following graph. Report your R command(s):



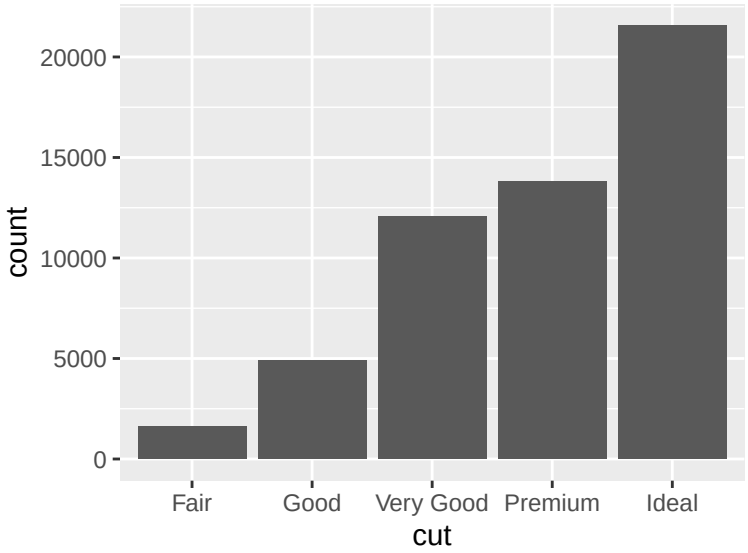
c) Now alter the code chunk by adding another aesthetic mapping, `fill = drv`, to duplicate the following graph:



3.3.5 Statistical Transformations

- Consider the bar plot below:

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



On the x-axis is `cut`, a variable in the `diamonds` data set. On the y-axis is `count`, which is *not* a variable in the data set.

- Many graphs (e.g. *scatterplots*) plot the raw data. Others plot ***statistics*** computed from the data:
  - *Bar plots* plot **counts** for each category.
  - *Histograms* plot **counts** for each bin (class interval).
  - *Scatterplot smoothers* plot y predictions based on a model fitted to the data.
  - *Boxplots* plot a set of five summary statistics of the data.
- Each `geom()` function has a default ***statistical transformation*** of the data that it plots.

  You can see the default **statistical transformation** by looking at the `geom()` function's help page. For example, typing:

```
? geom_bar
```

  shows that the default **statistic** for `geom_bar()` is "count".

- Each `stat()` function has a default set of **geometric objects** that it plots.

  We can usually use a `stat()` function in place of its `geom()` function. For example, the following do the same thing:

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

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

- Occasionally, we need to override the default **statistical transformation** of a `geom()` function. We can do this using the `stat` argument in the `geom()` function.

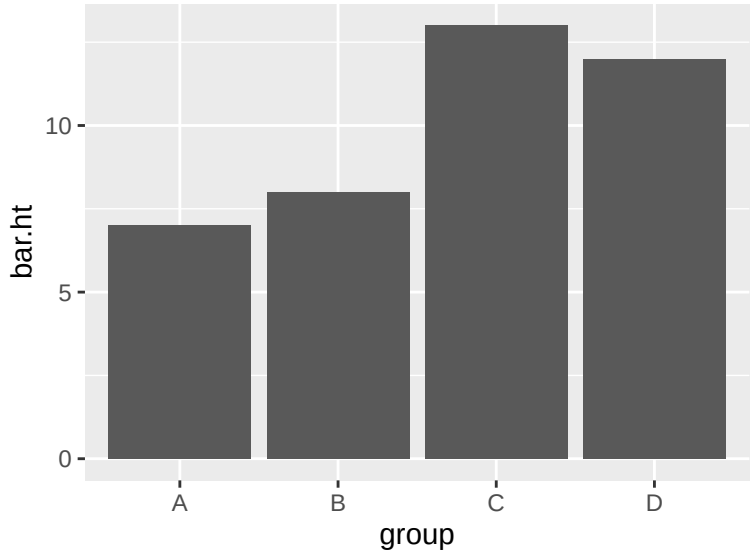
  For example, the following makes a bar plot whose bar heights are *values* in a data set, *not* category *counts*:

```
my.data <- data.frame(group = c("A", "B", "C", "D"), bar.ht = c(7, 8, 13, 12))

my.data

##   group bar.ht
## 1     A      7
## 2     B      8
## 3     C     13
## 4     D     12
```

```
ggplot(data = my.data) +
  geom_bar(mapping = aes(x = group, y = bar.ht), stat = "identity")
```



- Sometimes it's desirable to use a `stat()` function directly (instead of a `geom()` function), for example to make your code more readable:

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

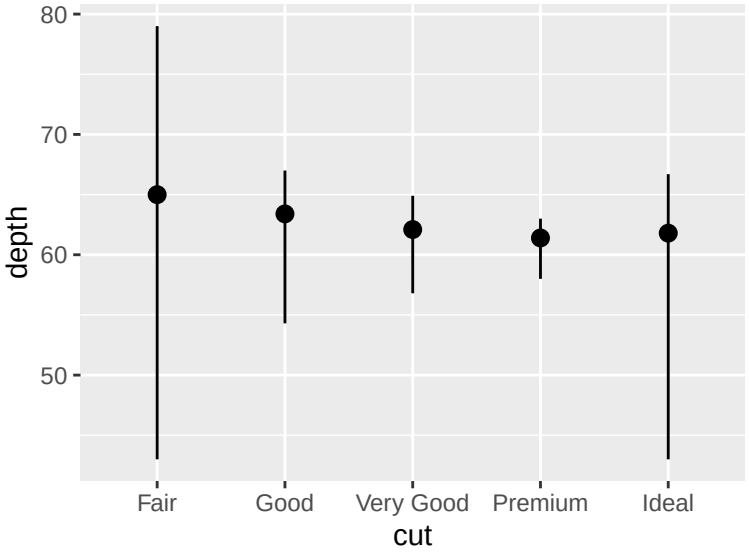


Figure 4

- There are more than 20 `stat()` functions, each of which has its own help page. For a list, see the `"ggplot2"` help page:

```
help(package = ggplot2)
```

Section 3.3 Exercises

Exercise 26 Here's the code for Fig. 4 again.

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

a) Each `stat()` function has a default type of **geometric object** that it plots. Look at the help page for `stat_summary()`. What's its default type of **geom**?

b) Verify that `geom_pointrange()` (the default default **geom** for `stat_summary()`)

can be used to duplicate Fig. 4 by typing:

```
grouped_by_cut <- data.frame(
                     cut = c("Fair",  "Good", "Very Good",
                             "Premium", "Ideal"),
                     lower = c(43.0, 54.3, 56.8, 58.0, 43.0),
                     upper = c(79.0, 67.0, 64.9, 63.0, 66.7),
                     median = c(65.0, 63.4, 62.1, 61.4, 61.8))
```

```
ggplot(data = grouped_by_cut) +
  geom_pointrange(mapping = aes(x = cut,
                                y = median,
                                ymin = lower,
                                ymax = upper))
```

Exercise 27 Look under "Computed Variables" in the help page for `stat_smooth()`. What statistical values does it compute?

Exercise 28 Look at the help page for `geom_col()`. What does `geom_col()` do? How does it differ from `geom_bar()`?

3.3.6 Position Adjustments

- There are two **aesthetics** for coloring bars of a bar plot, `fill` and `color`.

  Look at the difference between the following graphs.

```
## Color the bars using the color aesthetic
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, color = cut))
```

```
## Color the bars using the fill aesthetic
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = cut))
```

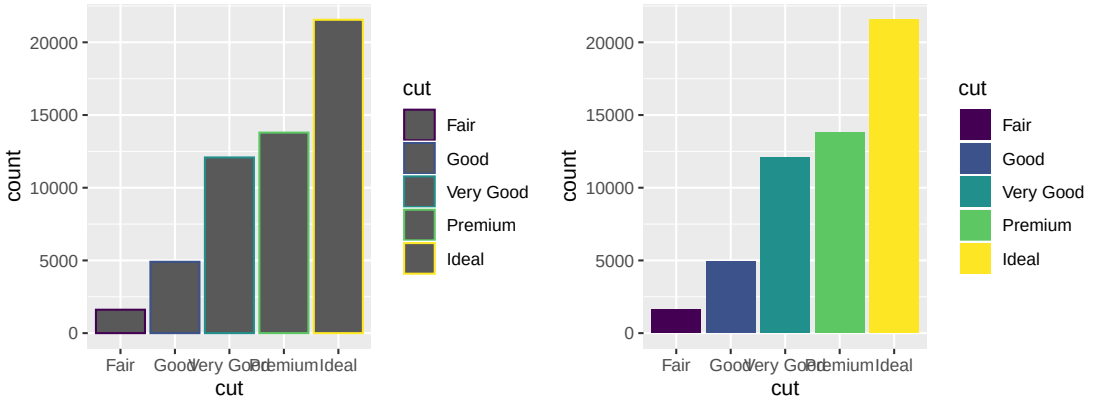
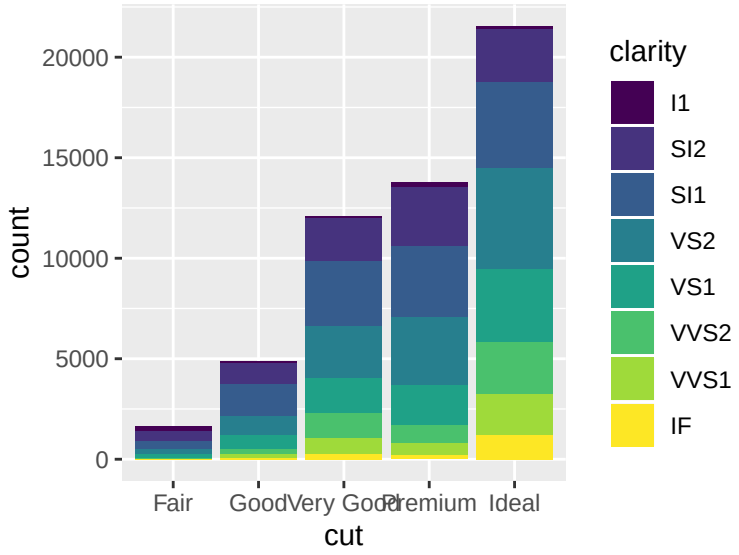


Figure 5

- Now watch what happens when we map `fill` to another variable, like `clarity`:

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



The bars are automatically "stacked" – each colored rectangle is a combination of a `cut` and a `clarity`.

The stacking is done automatically by **position adjustment** of the `position` argument.

- If we don't want the bars "stacked", we can specify one of the other three options for `position`: `"identity"`, `"fill"`, and `"dodge"`.
  - `position = "identity"` will place the object exactly where it falls in the graph, which is not very useful for bar plots because the bars will overlap. To see the bars,

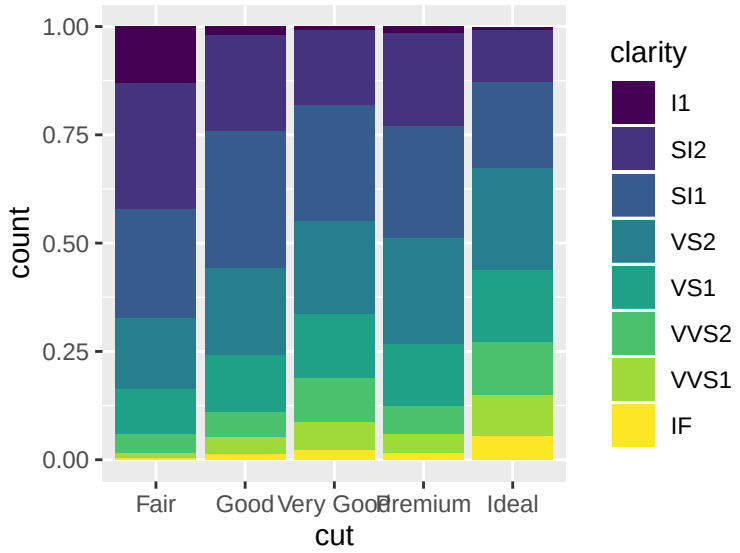
we'd need to ether set the **alpha transparency** value (e.g. `alpha = 1/5`) or make the bars completely transparent (`fill = NA`). For example, try the following:

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

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

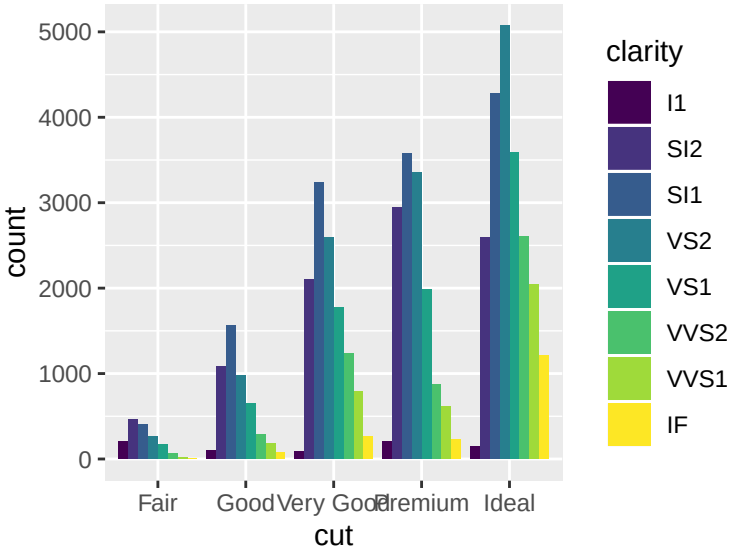
- `position = "fill"` stacks the bars but makes them all the same height. This makes it easier to compare *proportions* across groups.

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



- `position = "dodge"` places overlapping objects directly *beside* each other. This makes it easier to compare the counts.

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



- There's one other `position` adjustment that's useful for *scatterplots* (when x is a discrete variable).

  In the `mpg` data set, there are 236 observations. With no `position` adjustment, only 126 points are visible (Fig. 6, left).

  The problem is that many points overlap each other, which is called ***overplotting***.

  A solution is to specify `position = "jitter"`, which adds a small amount of random noise to each observation before plotting (Fig. 6, right).

```
## No position adjustment -- leads to overplotting
g1 <- ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ , y = hwy))
```

```
## Add jitter to the positions -- eliminates overplotting
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ , y = hwy), position = "jitter")
```

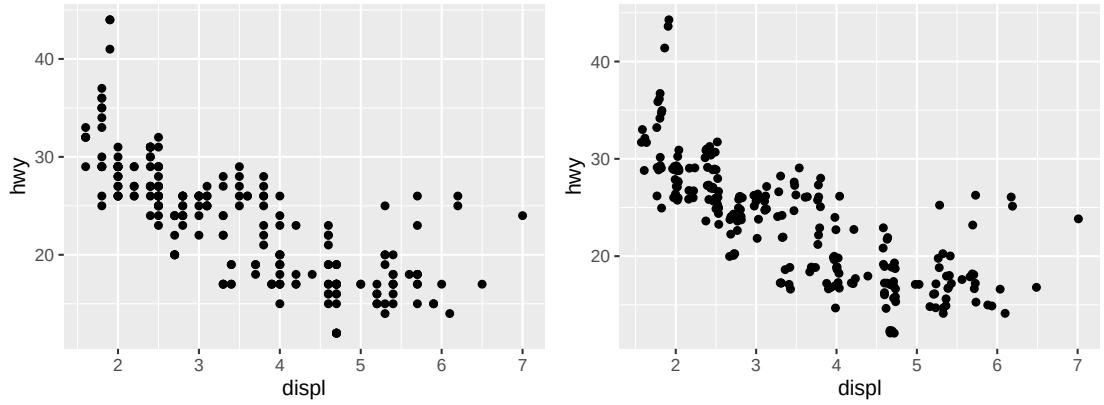


Figure 6

- Because "jittering" points is so useful, `"ggplot2"` has a shorthand for `geom_point(position = "jitter")`: `geom_jitter()`.
- To learn more about **position adjustments**, look at the help pages: `?position_dodge`, `?position_fill`, `?position_identity`, `?position_jitter`, `?position_stack`.

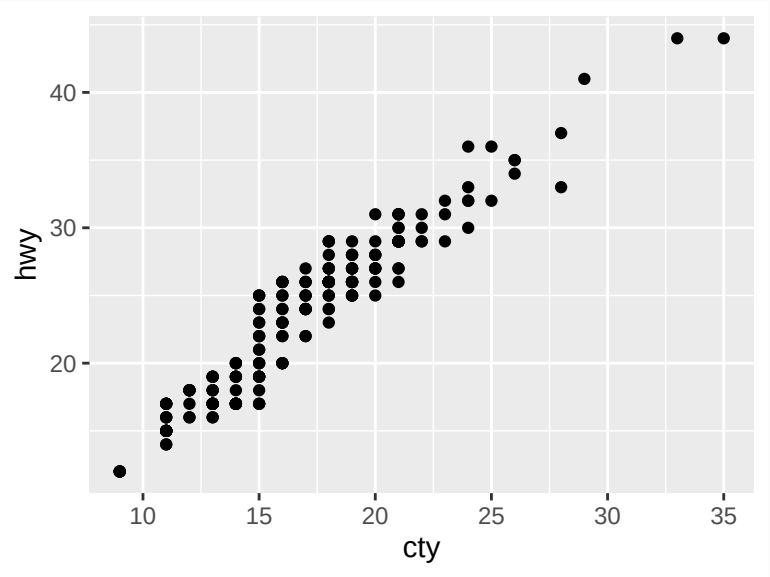
Section 3.3 Exercises

Exercise 29 Look at the help page for the `geom_bar()` function. What's the default value for the `position` argument? What's the default `position` for `geom_point()`?

Exercise 30 This problem concern **position adjustments**.

a) What's the problem with the following plot? **Hint**: The `mpg` data set contains 234 observations.

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



b) Re-run the command above using `position = "jitter"` and describe the improvement in the plot.

c) Because "jittering" points is so useful, `"ggplot2"` has a shorthand for `geom_point(position = "jitter")`: `geom_jitter()`.

Verify that the following command reproduces the "jittered" plot of Part *b*:

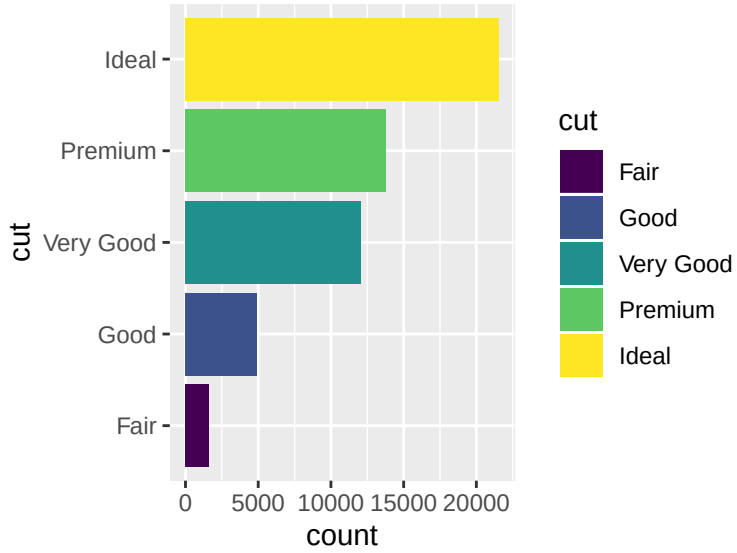
```
ggplot(data = mpg) +
  geom_jitter(mapping = aes(x = displ , y = hwy))
```

3.3.7 Coordinate Systems

- The default **coordinate system** in `ggplot()` is *Cartesian*.

Occasionally, other coordinate systems are useful. In particular, the `coord_flip()` function is useful for switching the x- and y-axes:

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

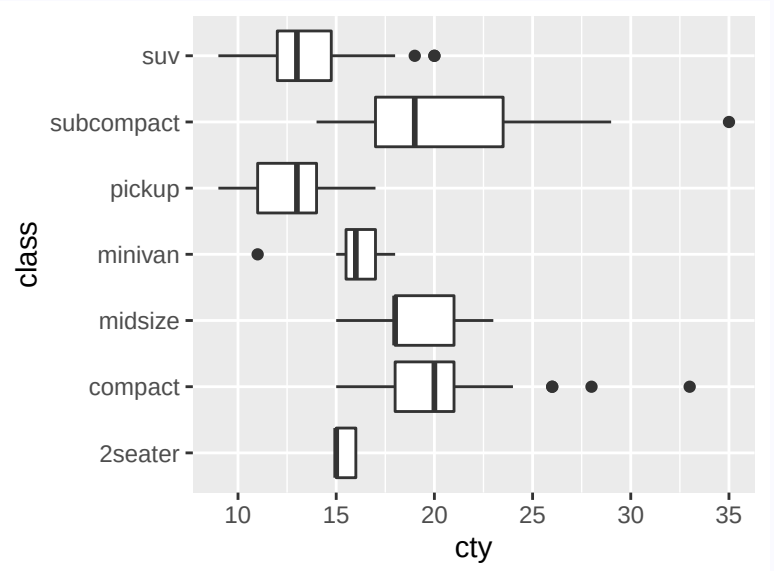


Section 3.3 Exercises

Exercise 31 Consider the following plot:

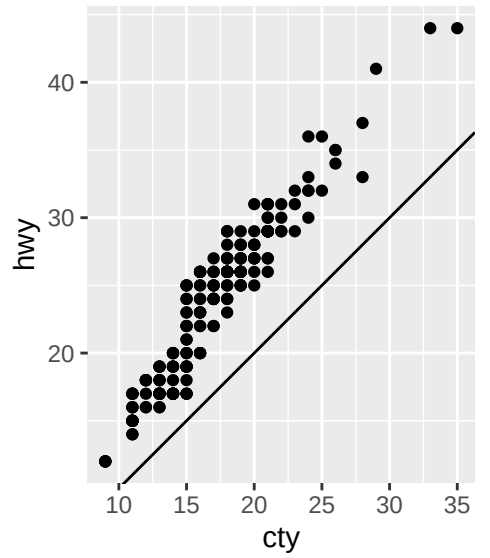
```
ggplot(data = mpg) +
  geom_boxplot(mapping = aes(x = class, y = cty))
```

Alter the code given above using `coord_flip()` to produce the following plot. Report your R command(s).



Exercise 32 What does the following plot tell you about city and highway mpg? Why is `coord_fixed()` important? What does `geom_abline()` do?

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point() +
  coord_fixed() +
  geom_abline()
```



3.3.8 The Layered Grammar of Graphics

- You now have the foundation to make *any* type of graph using the `"ggplot2"` package.
- Here's a **general template** (that incorporates **position adjustments**, **stats**, **coordinate systems**, and **faceting** into the **basic template** given earlier):

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
    stat = <STAT>,
    position = <POSITION> )  +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

3.4 Acknowledgment

- The above notes (and several examples) on the `"ggplot2"` package borrow heavily from the book:

  R for Data Science, by Wickham, H., Grolemund, G., O'Reilly, 2017.

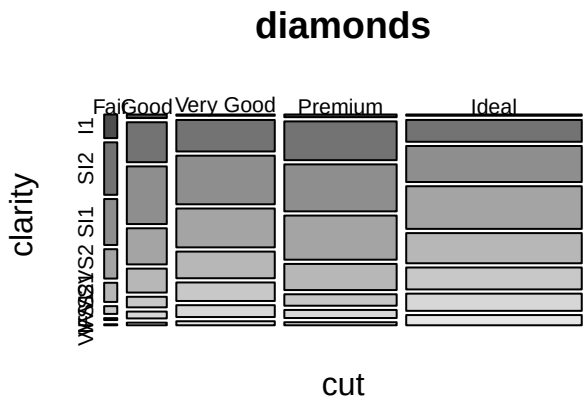
3.4.1 Mosaic Plots

- We saw one way to plot *two* categorical variables (`cut` and `clarity`) using bar plots.
- Another way is using a ***mosaic plot***, which can be created using the `mosaicplot()` function (from the base R `"graphics"` package).

mosaicplot()	*Make a mosaic plot of two categorical variables.*

- The main argument passed to `mosaicplot()` can be either of two types of objects:
  - A *contingency table* of **counts**, as produced by the `table()` function, or
  - An R *formula* of the form `y ~ x`, where `y` and `x` are both categorical variables, and (optionally) a data frame containing the variables.
- For example, here's how to make a *mosaic plot* of the `cut` and `clarity` variables from `diamonds` using a *formula*:

```
mosaicplot(cut ~ clarity, data = diamonds, color = TRUE)
```



The *area* of each box in the *mosaic plot* is proportional to the prevalence of `diamonds` in that combination of `cut` and `clarity`.

The width of each box is proportional to the percentage of `diamonds` with the given `cut`. The height is proportional to the percentage of the `diamonds` within the `cut` that have the given `clarity`.

(Compare with the bar plots of these same two variables created earlier.)

Section 3.4 Exercises

Exercise 33 This exercise uses the `diamonds` data set.

a) Make a *mosaic plot* of the `cut` and `color` variables. Report your R command(s).

b) Based on the plot, which combination of `cut` and `color` do you think is most prevalent? Which is least prevalent?

c) Type the following command:

```
table(diamonds$color, diamonds$cut)
```

Is your answer to Part *b* consistent with the values in the *contingency table*?