R Visualization
Module 1: Getting Started with ggplot2

Learning objectives

1. Produce scatter plots, boxplots, and time series plots using ggplot.
2. Set universal plot settings.
3. Describe what faceting is and apply faceting in ggplot.
4. Modify the aesthetics of an existing ggplot plot (including axis labels and color).
5. Build complex and customized plots from data in a data frame.
6. When to use a stat argument instead of relying on a geom function.
7. Find what stat a geom uses by checking the default value for the stat argument.
8. You can generally use geoms and stats interchangeably.

Introduction

"The simple graph has brought more information to the data analyst's mind than any other device." — **John Tukey**

Visualization is a great place to start with R programming because you get to make elegant and informative plots that help you understand data. In data visualization you'll dive into visualization, learning the basic structure of a ggplot2 plot, and powerful techniques for turning data into plots.

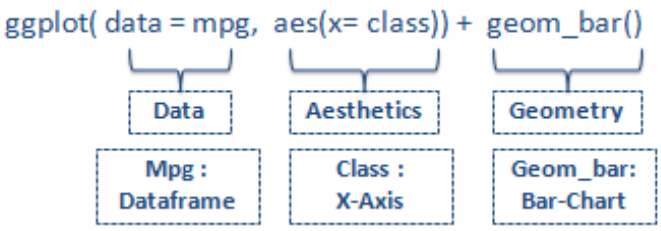
What is ggplot2?

ggplot2 is a robust and versatile R package for producing statistical, or data, graphics is an R package, developed by the most well-known R developer, Hadley Wickham, for generating aesthetic plots and charts.

This grammar, based on the Grammar of Graphics (Wilkinson, 2005), is made up of a set of independent components that can be composed in many different ways which believe in the principle that a plot can be split into the following basic parts -

```
Plot = data + aesthetics + geometry
```

- **data** refers to a data frame (dataset).
- **aesthetics** indicates x and y variables. It is also used to tell R how data are displayed in a plot, e.g. color, size, and shape of points, etc.
- **geometry** refers to the type of graphics (bar charts, histograms, box plot, line plot, density plot, dot plot, etc.)



ggplot2 Standard Syntax

Apart from the above three parts, there are other important parts of the plot -

- **Statistical transformation** allows you to add descriptive statistics on a plot.
- **Scales** are used to control x and y-axis limits.
- **Coordinate system** describes how data coordinates are mapped to the plane of the graphics.
- **Position adjustment** adjusts the position of overlapping objects within a layer.
- **Faceting** implies the same type of graph can be applied to each subset of the data. For example, for variable gender, creating 2 graphs for male and female.

Why ggplot2 is better?

ggplot2 is designed to work in a layered fashion, starting with a layer showing the raw data then adding layers of annotations and statistical summaries. It is especially helpful for students who have not yet developed a structured approach to the analysis used by experts.

Advantages of ggplot2:

- plot specification at a high level of abstraction.
- very flexible.
- theme system for polishing plot appearance.
- the mature and complete graphics system.

The table below shows common charts along with various important functions used in these charts.

Important Plots	**Important Functions**
Scatter Plot	geom_point(), geom_smooth(), stat_smooth()
Bar Chart	geom_bar(), geom_errorbar()
Histogram	geom_histogram(), stat_bin(), position_identity(), position_stack(), position_dodge()
Box Plot	geom_boxplot(), stat_boxplot(), stat_summary()
Line Plot	geom_line(), geom_step(), geom_path(), geom_errorbar()
Pie Chart	coord_polar()

The Setup

Install the ggplot2 packages

To use ggplot2, you must first install it. Make sure you have a recent version of R (at least version 2.8) from `http://r-project.org` and then run the following line of code to download and install the ggplot2 package.

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

And then to load it, use the following:

```
library("ggplot2")
```

You only need to install a package once, but you need to reload it every time you start a new session.

Data

In this module, we'll mostly use one data set that's bundled with `ggplot2:mpg`. It includes information about the fuel economy of popular car models in 1999 and 2008, collected by the US Environmental Protection Agency, `http://fueleconomy.gov`.

You can access the data by loading ggplot2:

```
library(ggplot2)
```

```
mpg
```

```
# A tibble: 234 x 11
   manufacturer model      displ  year   cyl trans      drv     cty   hwy fl    class
   <chr>        <chr>      <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
 1 audi         a4           1.8  1999     4 auto(l5)   f        18    29 p     compact
 2 audi         a4           1.8  1999     4 manual(m5) f        21    29 p     compact
 3 audi         a4           2    2008     4 manual(m6) f        20    31 p     compact
 4 audi         a4           2    2008     4 auto(av)   f        21    30 p     compact
 5 audi         a4           2.8  1999     6 auto(l5)   f        16    26 p     compact
 6 audi         a4           2.8  1999     6 manual(m5) f        18    26 p     compact
 7 audi         a4           3.1  2008     6 auto(av)   f        18    27 p     compact
 8 audi         a4 quattro   1.8  1999     4 manual(m5) 4        18    26 p     compact
 9 audi         a4 quattro   1.8  1999     4 auto(l5)   4        16    25 p     compact
10 audi         a4 quattro   2    2008     4 manual(m6) 4        20    28 p     compact
# ... with 224 more rows
```

The variables are mostly self-explanatory:

- **`cty`** and **`hwy`** record miles per gallon (mpg) for city and highway driving.
- **`displ`** is the engine displacement in liters.
- **`drv`** is the drivetrain: front wheel (f), rear wheel (r) or four-wheel (4).
- **`model`** is the model of car. There are 38 models, selected because they had a new edition every year between 1999 and 2008.
- **`class`** (not shown), is a categorical variable describing the "type" of car: two-seater, SUV, compact, etc.

This dataset suggests many interesting questions. How are engine size and fuel economy related? Do certain manufacturers care more about fuel economy than others? Has fuel economy improved in the last 10 years? We will try to answer some of these questions, and in the process learn how to create some basic plots with ggplot2.

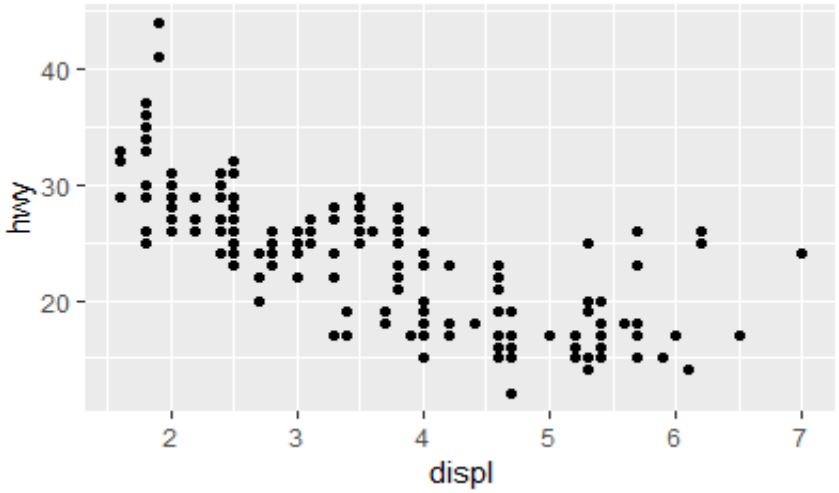
Every ggplot2 plot has three key components:

1. **Data,**
2. A set of **aesthetic mappings** between variables in the data visual properties, and
3. At least one layer which describes how to render each observation. Layers are usually created with a **geom** function.

Here's a simple example:

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

`geom_point()`



This produces a scatterplot defined by

1. Data: `mpg`.
2. Aesthetic mapping: engine size mapped to `X position`, fuel economy to `Y Position`.
3. Layer: points.

The plot shows a strong correlation: as the engine size gets bigger, the fuel economy gets worse. There are also some interesting Outliers: some cars with large engines get higher fuel economy than average. What sort of cars do you think they are?

Aesthetic Mapping

"The greatest value of a picture is when it forces us to notice what we never expected to see."—
John Tukey

In ggplot2, *aesthetic* means "something you can see". This mapping between data and visual aesthetics is the second element of a ggplot2 layer. Aesthetic mappings describe how variables in the data are mapped to visual properties (aesthetics) of geoms. Examples include:

- position (i.e., on the x and y axes)
- color ("outside" color)
- fill ("inside" color)
- shape (of points)
- line type
- size

In summary, you use the **`aes()`** function to define the mapping between your data and your plot. These work in the same way as the x and y aesthetics, and are added into the call to `aes()`:

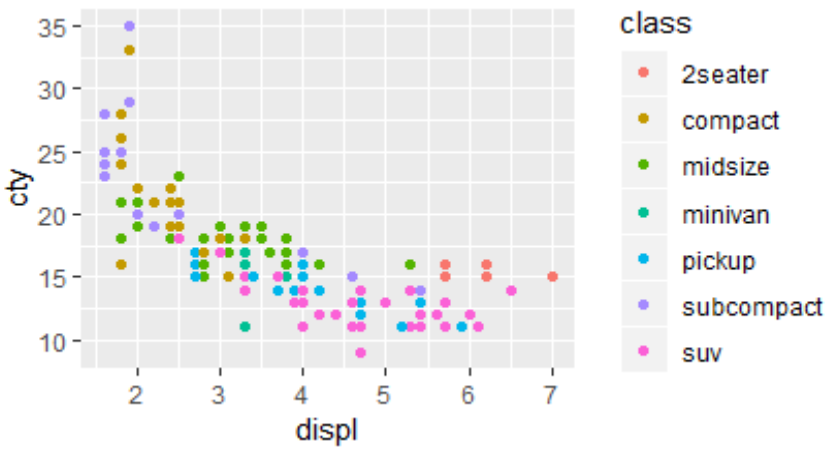
- `aes(displ, hwy, color = class)`
- `aes(displ, hwy, shape = drv)`
- `aes(displ, hwy, size = cyl)`

Aesthetic mappings can be set in `ggplot2()` and in individual layers.

ggplot2 takes care of the details of converting data (e.g., 'f', 'r', '4') into aesthetics (e.g.. `red', 'yellow', 'green') with a **scale**. There is one scale for each aesthetic mapping in a plot. The scale is also responsible for creating a guide, an axis or legend, that allows you to read the plot, converting aesthetic values back into data values. For now, we'll stick with the default scales provided by ggplot2.

To learn more about those outlying variables in the previous scatterplot, we could map the class variable to colour:

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



This gives each point a unique colour corresponding to its class. The legend allows us to read data values from the colour, showing us that the group of cars with unusually high fuel economy for their engine size are two-seaters: cars with big engines, but lightweight bodies.

Different types of aesthetic attributes work better with different types of variables. For example, colour and shape work well with categorical variables, while size works well for continuous variables. The amount of data also make a difference: if there is a lot of data it can be hard to distinguish different groups. An alternative solution is to use facetting, as described next.

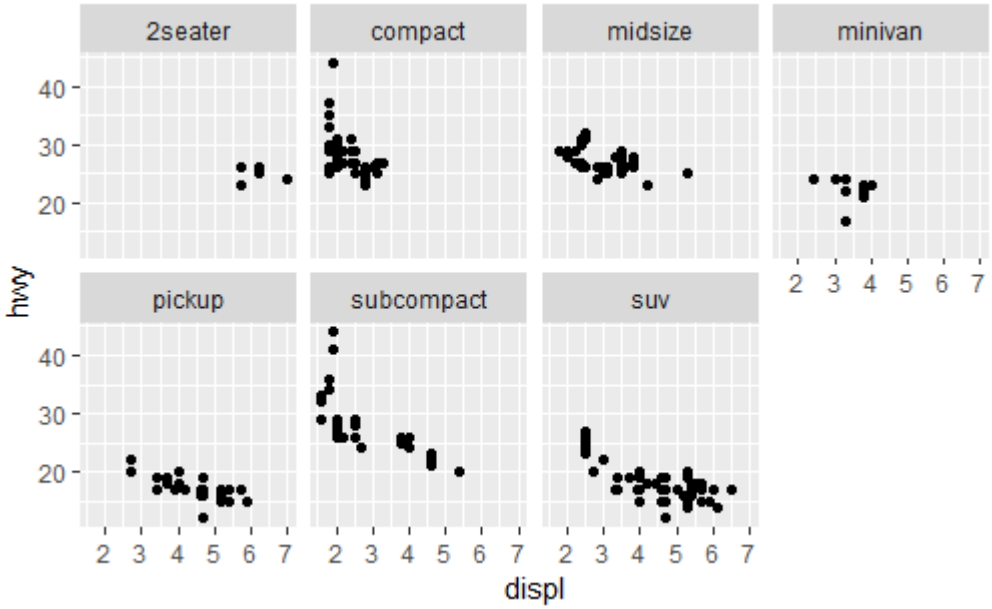
The Facets

One way to add additional variables is with aesthetics. Another way, particularly useful for categorical variables, is to split your plot into facets, subplots that each display one subset of the data.

Facetting creates tables of graphics by splitting the data into subsets and displaying the same graph for each subset. There are two types of facetting: grid and wrapped.

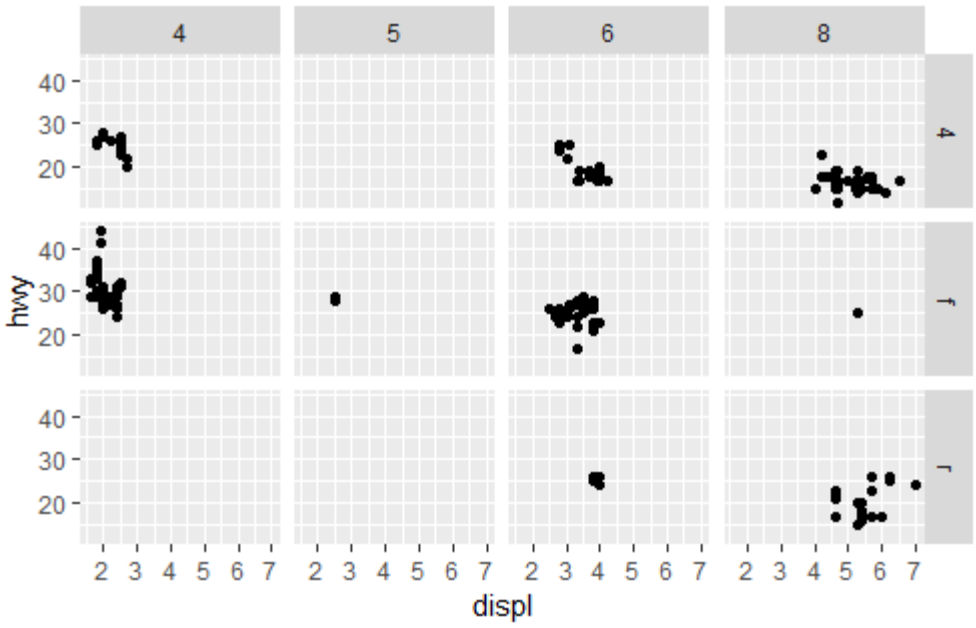
To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` should be a formula, which you create with ~ followed by a variable name. The variable that you pass to `facet_wrap()` should be discrete:

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



To facet your plot on the combination of two variables, add `facet_grid( )` to your plot call. The first argument of `facet_grid()` is also a formula. This time the formula should contain two variable names separated by a **~**:

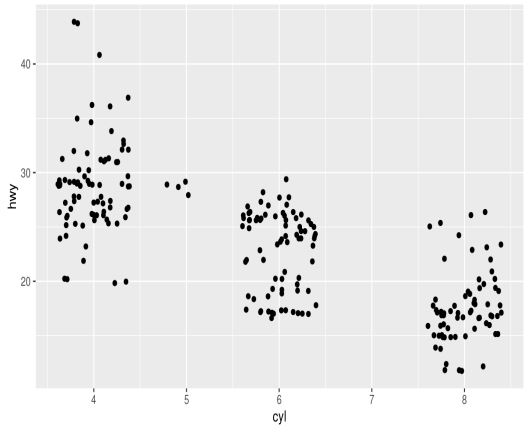
```
ggplot(data = mpg) +
    geom_point(mapping = aes(x = displ, y = hwy)) +
    facet_grid(drv ~ cyl)
```



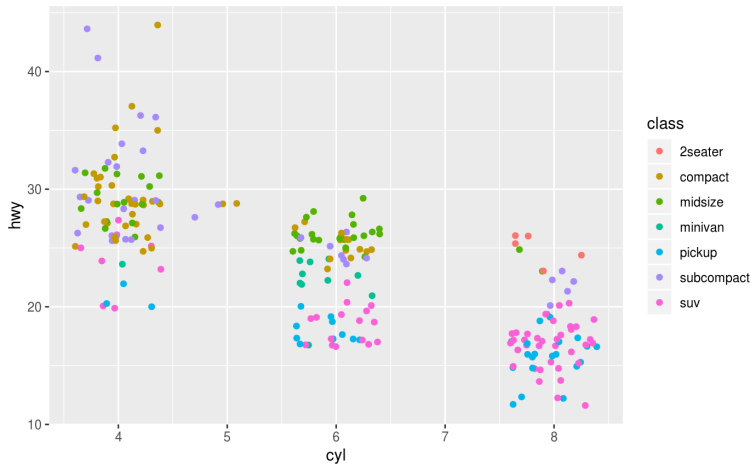
Plot Geoms

A geom is the geometrical object that a plot uses to represent data. A plot's geometry dictates what visual elements will be used.

How are these two plots similar?



Both plots contain the same x variable and the same y variable, and both describe the same data. But the plots are not identical. Each plot uses a different visual object to represent the data. In ggplot2 syntax, we say that they use different geoms.



People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms, and so on. Scatterplots break the trend; they use the point geom. As we see in the preceding plots, you can use different geoms to plot the same data. The plot on the left uses the point geom, and the plot on the right uses the smooth geom, a smooth line fitted to the data.

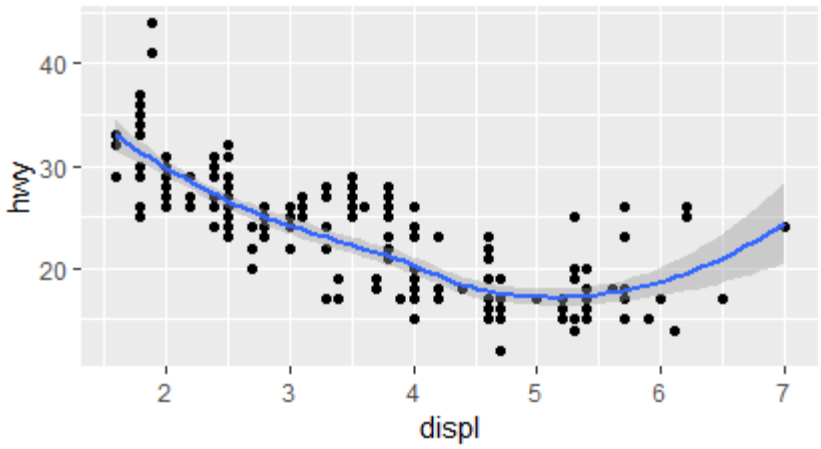
We will look at a variety of ways to construct plots.

- `geom_smooth()` fits a smoother to the data and displays the smooth and its standard error.
- `geom_boxplot()` produces a box-and-whisker plot to summarise the distribution of a set of points.
- `geom_histogram()` and geom_freqpoly() show the distribution of continuous variables.
- `geom_bar()` shows the distribution of categorical variables.
- `geom_path()` and `geom_line()` draw lines between the data points. A line plot is constrained to produce lines that travel from left to right, while paths can go in any direction. Lines are typically used to explore how things change over time.

Adding a Smoother to a Plot

If you have a scatterplot with a lot of noise, it can be hard to see the dominant pattern. In this case, it's useful to add a smoothed line to the plot with `geom_smooth()`:

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

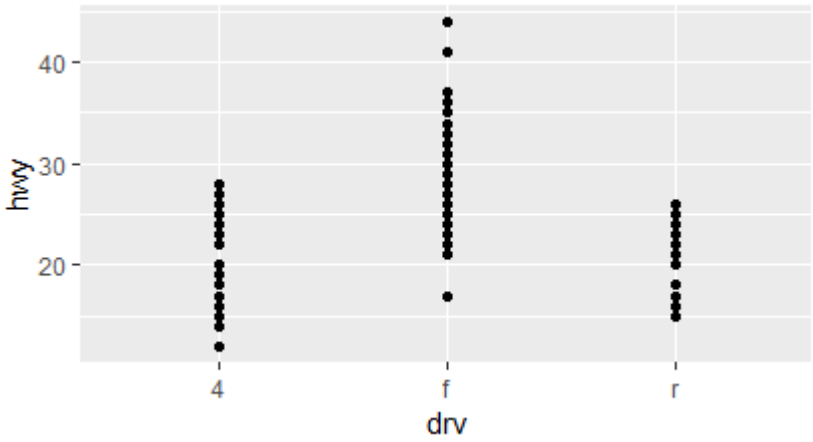


This overlays the scatterplot with a smooth curve, including an assessment of uncertainty in the form of point-wise confidence intervals shown in grey. If you're not interested in the confidence interval, turn it off with `geom_smooth(se = FALSE)`.

Boxplots and tittered Points

When a set of data includes a categorical variable and one or more continuous variables, you will probably be interested to know how the values of the continuous variables vary with the levels of the categorical variable. Say we're interested in seeing how fuel economy varies within car class. We might start with a scatterplot like this:

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



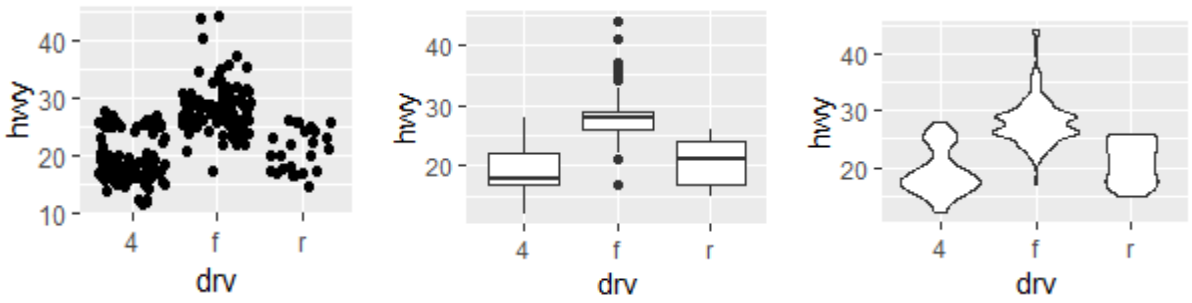
Because there are few unique values of both class and hwy, there is a lot of overplotting. Many points are plotted in the same location, and it's difficult to see the distribution. There are three useful techniques that help alleviate the problem:

- `Jittering, geom_jitter()`, adds a little random noise to the data which can help avoid overplotting.

- `Boxplots, geom_boxplot()`, summarise the shape of the distribution with a handful of summary statistics.
- `Violin plots, geom_violin()`, show a compact representation of the "density" of the distribution, highlighting the areas where more points are found.

These are illustrated below:

```
ggplot(mpg, aes(drv, hwy)) + geom_jitter()
ggplot(mpg, aes(drv, hwy)) + geom_boxplot()
ggplot(mpg, aes(drv, hwy)) + geom_violin()
```



Each method has its strengths and weaknesses. Boxplots summarise the bulk of the distribution with only five numbers, while jittered plots show every point hut only work with relatively small datasets. Violin plots give the richest display, but rely on the calculation of a density estimate, which can be hard to interpret.

For jittered points, `geom_jitter()` offers the same control over aesthetics as geom_point(): size, colour, and shape. For `geom_boxplot()` and `geom_violin()`, you can control the outline colour or the internal fill colour.

Histograms and Frequency Polygons

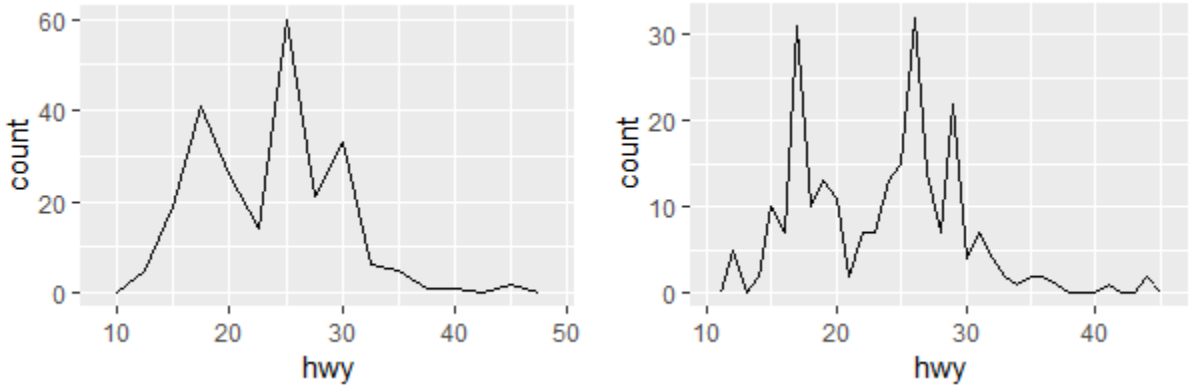
Histograms and frequency polygons show the distribution of a single numeric variable. They provide more information about the distribution of a single group than boxplots do, at the expense of needing more space.

```
ggplot(mpg, aes(hwy)) + geom_histogram()
ggplot(mpg, aes(hwy)) + geom_freqpoly()
```

Both histograms and frequency polygons work in the same way: they bin the data, then count the number of observations in each bin. The only difference is the display: histograms use bars and frequency polygons use lines.

You can control the width of the bins with the binwidth argument (if You don't want evenly spaced bins you can use the breaks argument). It is very important to experiment with the bin width. The default just splits your data into 30 bins, which is unlikely to be the best choice. You should always try many bin widths, and you may find you need multiple bin widths to tell the full story of your data.

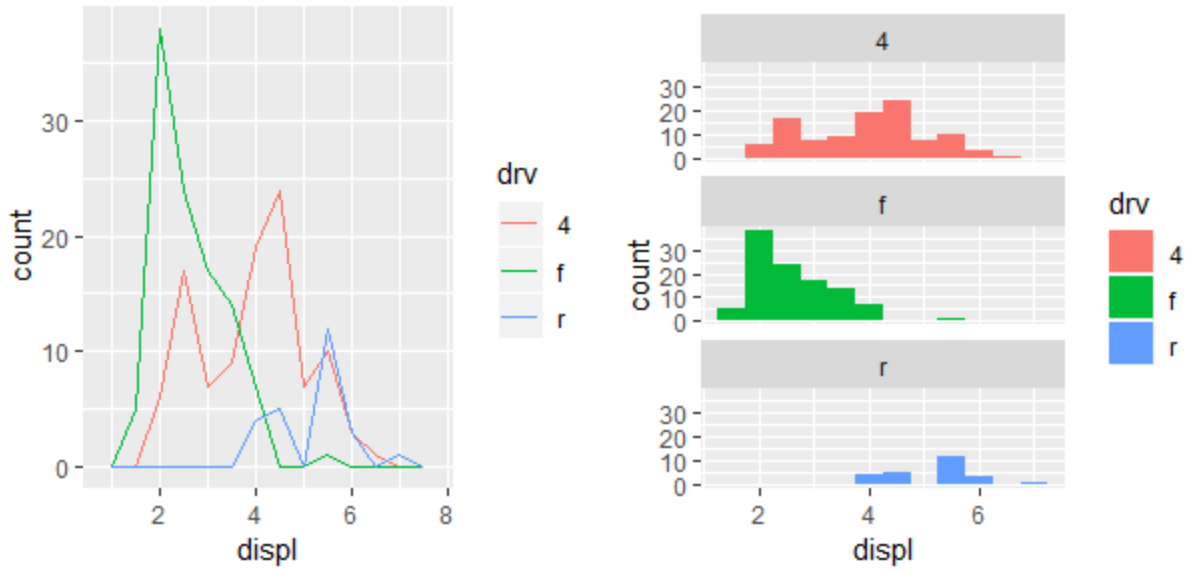
```
ggplot(mpg, aes(hwy)) +
  geom_freqpoly(binwidth = 2.5)
ggplot(mpg, aes(hwy)) +
  geom_freqpoly(binwidth = 1)
```



An alternative to the frequency polygon is the density plot, `geom_density()`. I'm not a fan of density plots because they are harder to interpret since the underlying computations are more complex. They also make assumptions that are not true for all data, namely that the underlying distribution is continuous, unbounded, and smooth.

To compare the distributions of different subgroups, you can map a categorical variable to either `fill` (for `geom_histogram()`) or `colour` (for `geom_freqpoly()`). It's easier to compare distributions using the frequency polygon because the underlying perceptual task is easier. You can also use facetting: this makes comparisons a little harder, but it's easier to see the distribution of each group.

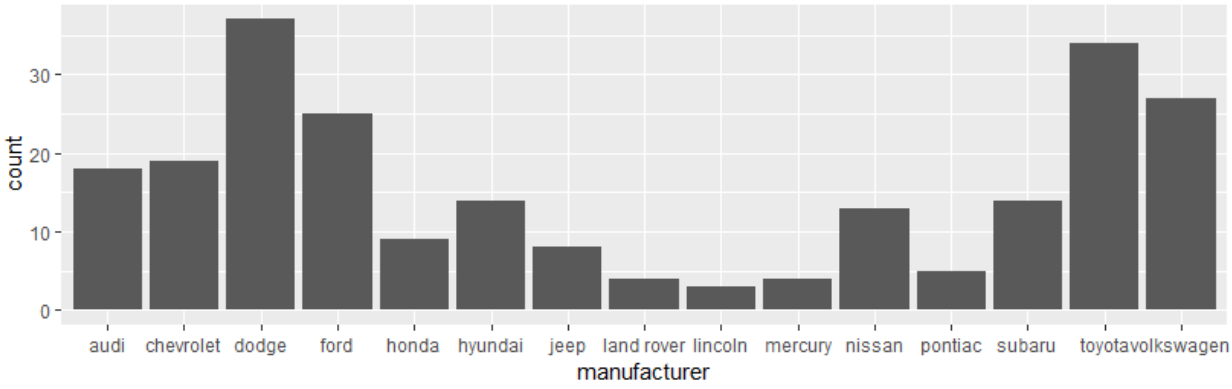
```
ggplot(mpg, aes(displ, colour = drv)) +
  geom_freqpoly(binwidth = 0.5)
ggplot(mpg, aes(displ, fill = drv)) +
  geom_histogram(binwidth = 0.5) +
  facet_wrap(~drv, ncol = 1)
```



Bar Charts

The discrete analogue of the histogram is the `bar chart, geom_bar()`. It's easy to use:

```
ggplot(mpg, aes(manufacturer)) +
  geom_bar()
```

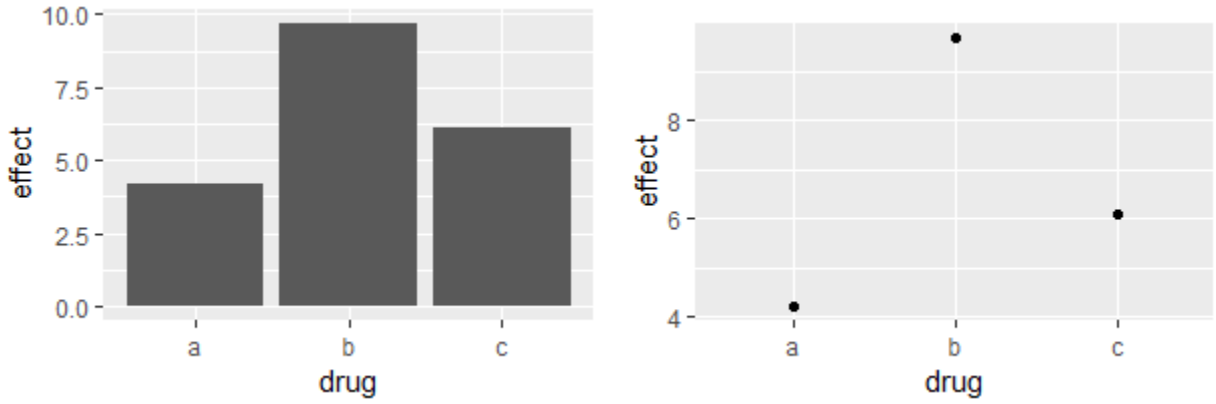


Bar charts can be confusing because there are two rather different plots that are both commonly called bar charts. The above form expects you to have unsummarized data, and each observation contributes one unit to the height of each bar. The other form of the bar chart is used for pre summarised data. For example, you might have three drugs with their average effect:

```
drugs <- data.frame(
  drug = c("a", "b", "c"),
  effect = c(4.2, 9.7, 6.1)
)
```

To display this sort of data, you need to tell geom_bar() to not run the default stat which bins and counts the data. However, I think it's even better to use geom_point() because points take up less space than bars, and don't require that the y axis includes 0.

```
ggplot(drugs, aes(drug, effect)) + geom_bar(stat = "identity")
ggplot(drugs, aes(drug, effect)) + geom_point()
```

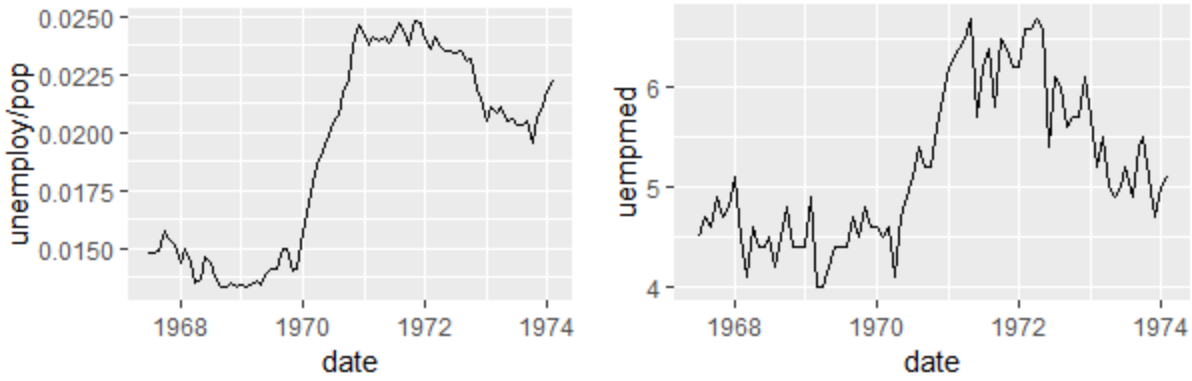


Time Series with Line and Path Plots

Line and path plots are typically used for time series data. Line plots join the points from left to right, while path plots join them in the order that they appear in the dataset (in other words, a line plot is a path plot of the data sorted by x value). Line plots usually have time on the x-axis, showing how a single variable has changed over time. Path plots show how two variables have simultaneously changed over time, with time encoded in the way that observations are connected.

Because the year variable in the mpg dataset only has two values, we'll show some time series plots using the economics dataset, which contains economic data on the US measured over the last 40 years. The figure below shows two plots of unemployment over time, both produced using `geom_line()` The first shows the unemployment rate while the second shows the median number of weeks unemployed. We can already see some differences in these two variables, particularly at the last peak, where the unemployment percentage is lower than it was in the preceding peaks, but the length of unemployment is high.

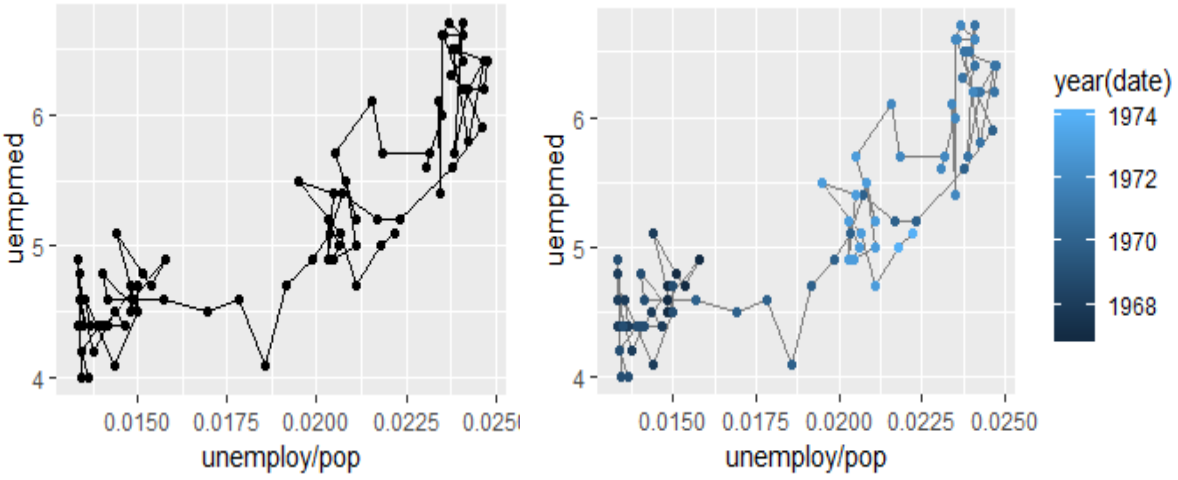
```
ggplot(economics, aes(date, unemploy /pop)) +
  geom_line()
ggplot(economics, aes(date, uempmed)) +
  geom_line()
```



To examine this relationship in greater detail, we would like to draw both time series on the same plot. We could draw a scatterplot of unemployment rate vs. length of unemployment, but then we could no longer see the evolution over time. The solution is to join points adjacent in time with line segments, forming a *path* plot.

Below we plot the unemployment rate vs. length of unemployment and join the individual observations with a path. Because of the many line crossing, the direction in which time flows isn't easy to see in the first plot. In the second plot, we colour the points to make it easier to see the direction of time.

```
ggplot(economics, aes(unemploy / pop, uempmed)) +
  geom_path() +
  geom_point()

year <- function(x) as.POSIXlt(x)$year + 1900
ggplot(economics, aes(unemploy / pop, uempmed)) +
  geom_path(colour = "grey50") +
  geom_point(aes(colour = year(date)))
```



We can see that the unemployment rate and length of unemployment are highly correlated, but in recent years the length of unemployment has been increasing relative to the unemployment rate.

NOTE: If you've mastered the basics and want to learn more, read module 2. It describes the theoretical set of ideas of ggplot2 and shows you how all the pieces fit together. This module will help you create new types of graphics specifically tailored to your needs.

Statistical transformation

The Statistical transformation (stat). It's often useful to transform your data before plotting. A statistical transformation, or **stat**, transforms the data, typically by summarising it in some manner. For example, a useful stat is the smoother, which calculates the smoothed mean of y, conditional on x. You've already many of ggplot2's stats because they're used behind the scenes to generate many important geoms:

- `stat_bin()`: `geom_bar()`, `geom_freqpoly()`, `geom_histogram()`
- `stat_bin2d()`: `geom_bin2d0`
- `stat_bindot()`: `geom_dotplot()`
- `stat_binhex()`: `geom_hex()`
- `stat_boxplot()`: `geom_boxplot()`
- `stat_contour()`: `geom_contour()`
- `stat_quantile()`: `geom_quantile()`
- `stat_smooth()`: `geom_smooth()`
- `stat_sum()`: `geom_count()`

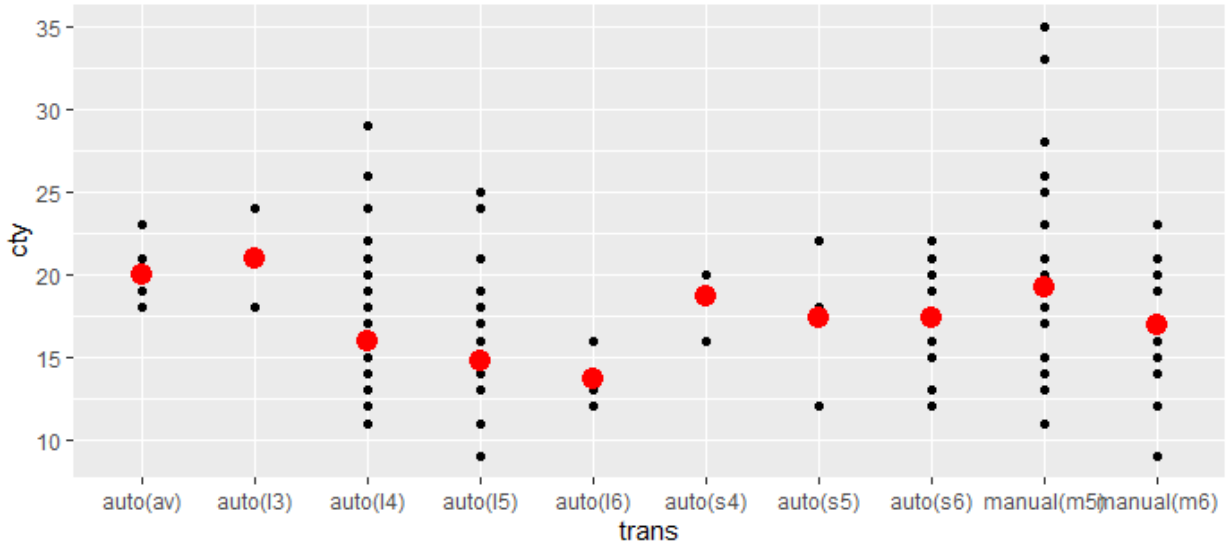
You'll rarely call these functions directly, but they are useful to know about because their documentation often provides more detail about the corresponding statistical transformation.

Other stats can't be created with a `geom_` function:

- `stat_ecdf()`: compute an empirical cumulative distribution plot.
- `stat_function()`: compute y values from a function of x values.
- `stat_summary()`: summarise y values at distinct x values.
- `stat_summary2d()`, `stat_summary_hex()`: summarise binned values.
- `stat_qq()`: perform calculations for a quantile-quantile plot.
- `stat_spoke()`: convert angle and radius to position.
- `stat_unique()`: remove duplicated rows.

There are two ways to use these functions. You can either add a `stat_()` function and override the default geom, or add a `geom_()` function and override the default stat:

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

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



I think it's best to use the first form because it makes it more clear that you're displaying a summary, not the raw data.

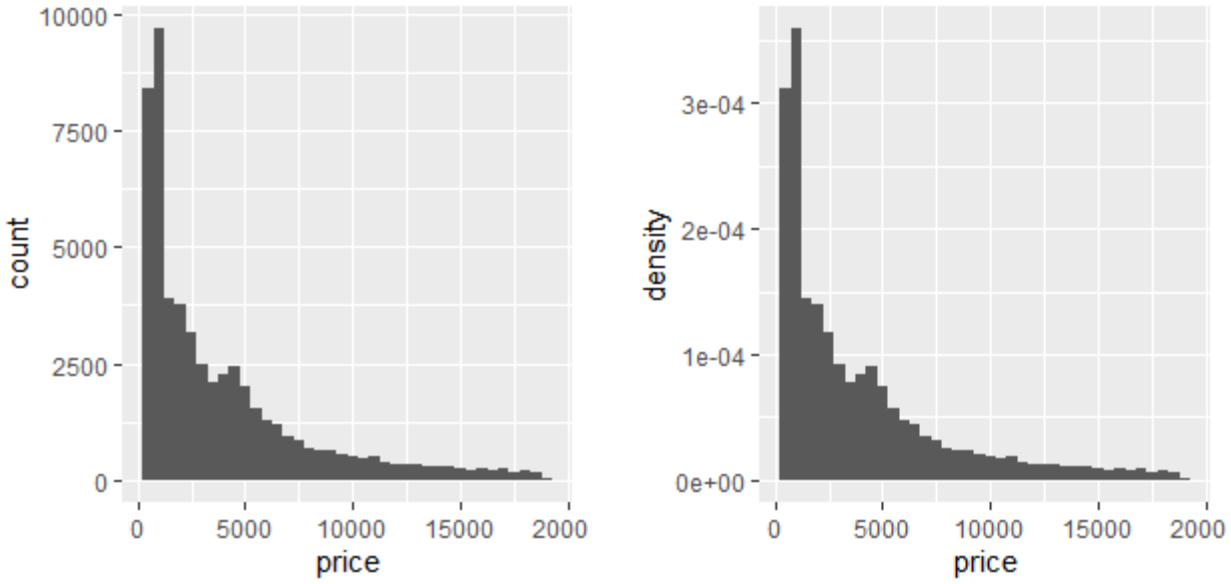
Generated Variables

Internally, a stat takes a data frame as input and returns a data frame as output, and so a stat can add new variables to the original dataset. It is possible to map aesthetics to these new variables. For example, stat_bin, the statistic used to make histograms, produces the following variables:

- count, the number of observations in each bin
- the density of observations in each bin (percentage of total/bar
- x, the center of the bin

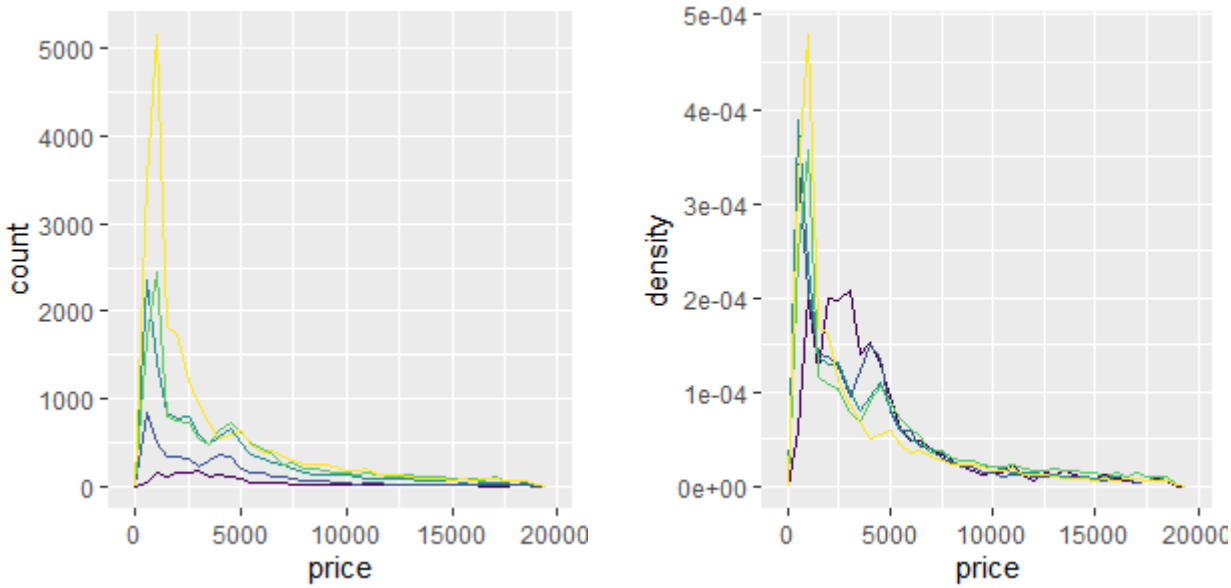
These generated variables can be used instead of the variables present in the original dataset. For example, the default histogram geom assigns the height of the bars to the number of observations (`count`), but if you'd prefer a more traditional histogram, you can use the density (`density`). To refer to a generated variable like density, **“. .”** must surround the name. This prevents confusion in case the original dataset includes a variable with the same name as a generated variable, and it makes it clear to any later reader of the code that this variable was generated by a stat. Each statistic lists the variables that it creates in its documentation. Compare the y-axes on these two plots:

```
ggplot(diamonds, aes(price)) +
  geom_histogram(binwidth = 500)

 ggplot(diamonds, aes(price)) +
      geom_histogram(aes(y    =    ..density..),    binwidth    =    500)
```



This technique is particularly useful when you want to compare the distribution of multiple groups that have very different sizes. For example, it's hard to compare the distribution of price within cut because some groups are quite small. It's easier to compare if we standardize each group to take up the same area:

```
ggplot(diamonds, aes(price, colour = cut)) +
    geom_freqpoly(binwidth = 500) +
    theme(legend.position = "none")

 ggplot(diamonds, aes(price, colour = cut)) +
    geom_freqpoly(aes(y = ..density..), binwidth = 500) +
    theme(legend.position = "none")
```



The result of this plot is rather surprising: low-quality diamonds seem to be more expensive on average.

R Visualization
Module 2: Facets and Coordinate Systems

Learning objectives

1. Understanding the underlying grammar and components that control position.
2. Displaying and creating the tables of graphics by splitting the data into subsets and displaying the same graph for each subset.
3. Learning coordinate systems like cartesian can be useful in special cases.

Introduction

This module discusses the position, particularly how facets are laid out on a page, and how coordinate systems within a panel work. There are four components that control position. You have already learned about two of them that work within a facet:

- **Position adjustments** adjust the position of overlapping objects within a layer. These are most useful for bar and other interval geoms but can be useful in other situations.
- **Position scales** control how the values in the data are mapped to positions on the plot.
- **Facetting** is a mechanism for automatically laying out multiple plots on a page. It splits the data into subsets, and then plots each subset in a different panel. Such plots are often called small multiples or trellis graphics.
- **Coordinate systems** control how the two independent position scales are combined to create a 2d coordinate system. The most common coordinate system is Cartesian, but other coordinate systems can be useful in special circumstances.

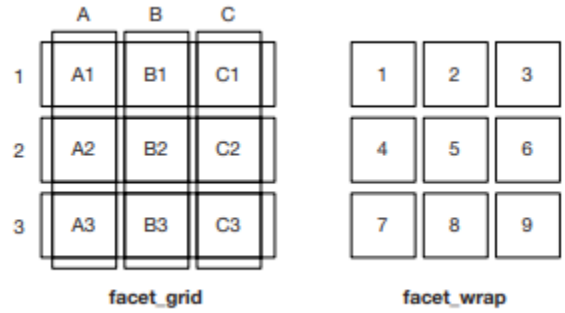
Facetting

Facetting generates small multiples each showing a different subset of the data. Small multiples are a powerful tool for exploratory data analysis: you can rapidly compare patterns in different parts of the data and see whether they are the same or different. This section will discuss how you can fine-tune facets, particularly the way in which they interact with position scales.

There are three types of facetting:

- `facet_null()`: a single plot, the default.
- `facet_wrap()`: "wraps" a gold ribbon of panels into 2d.
- `facet_grid()`: produces a 2d grid of panels defined by variables which form the rows and columns.

The differences between `facet_wrap()` and `facet_grid()` are illustrated below.



This is illustrating the difference between the two facetting system `facet_grid()` (*left*) is fundamentally 2d, being made up of two independent components `facet_wrap()` (*right)* is ld, but wrapped into 2d to save space.

Faceted plots have the capability to fill up a lot of space, so for this module, we will use a subset of the mpg dataset that has a manageable number of levels: three cylinders (4, 6, 8), two types of the drive train (4 and f), and six classes.

```
mpg2 <- subset(mpg, cyl != 5 & drv %in% c("4", "f") & class != "2seater")
```

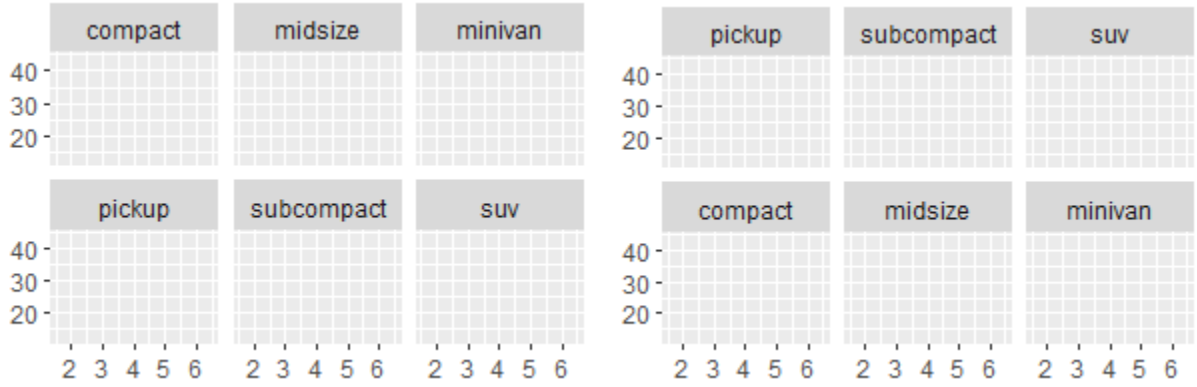
Facet Wrap

`facet_wrap()` makes a long ribbon of panels (generated by any number of variables) and wraps it into 2d. This is useful if you have a single variable with many levels and want to arrange the plots in a more space efficient manner.

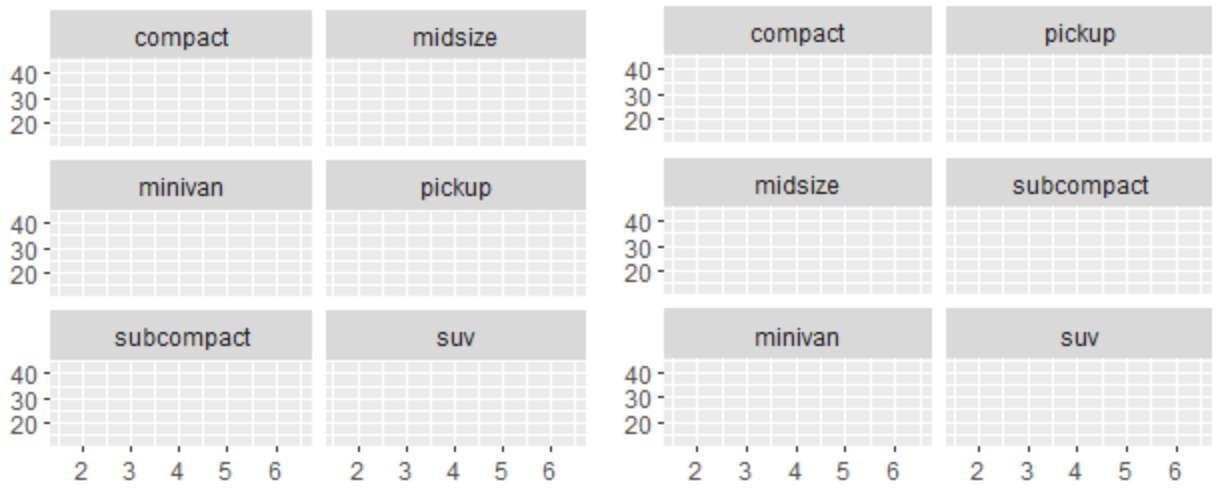
You can control how the ribbon is wrapped into a grid with `ncol, nrow as.table` and `dir.ncol` and now control how many columns and rows (you only need to set one). `as.table` controls whether the facets are laid out like a table (TRUE), with highest values at the bottom-right, or a plot (FALSE), with the highest values at the top-right. dir controls the direction of wrap: horizontal or vertical.

```
base <- ggplot(mpg2, aes(displ, hwy)) +
   geom_blank() +
   xlab(NULL) +
   ylab(NULL)
```

```
base + facet_wrap(~class, ncol = 3)
base + facet_wrap(~class, ncol = 3, as.table = FALSE)
```




```
base + facet_wrap(~class, nrow = 3)
base + facet wrap(~class, nrow = 3, dir = "v")
```

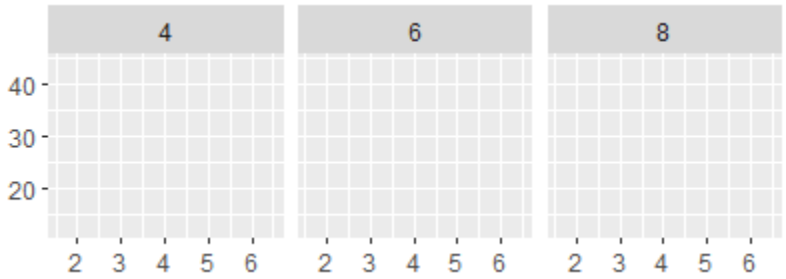


Facet Grid

`facet_grid()` lays out plots in a 2d grid, as defined by a formula:

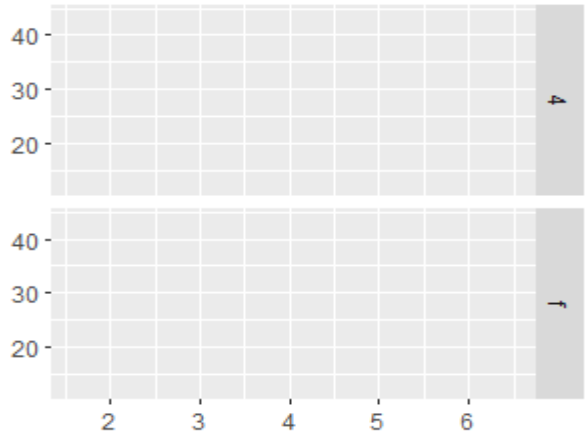
- `. ~ a` spreads the values of `a` across the columns. This direction facilitates comparisons of y position because the vertical scales are aligned.

  ```
  base + facet_grid(. ~ cyl)
  ```



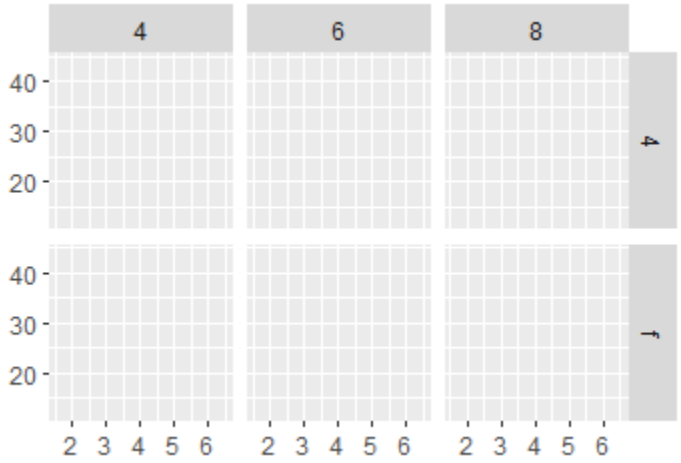
- `b ~ .` spreads the values of `b` down the rows. This direction facilitates comparison of x position because the horizontal scales are aligned. This makes it particularly useful for comparing distributions.

  ```
  base + facet_grid(drv ~ .)
  ```



- `a ~ b` spreads `a` across columns and `b` down rows. You'll usually want to put the variable with the greatest number of levels in the columns, to take advantage of the aspect ratio of your screen.

  ```
  base + facet_grid(drv ~ cyl)
  ```



You can use multiple variables in the rows or columns, by "adding" them together, e.g. a + b ~ c + d. Variables appearing together on the rows or columns are nested in the sense that only combinations that appear in the data will appear in the plot. Variables that are specified on rows and columns will be crossed: all combinations will be shown, including those that didn't appear in the original dataset: this may result in empty panels.

Controlling Scales

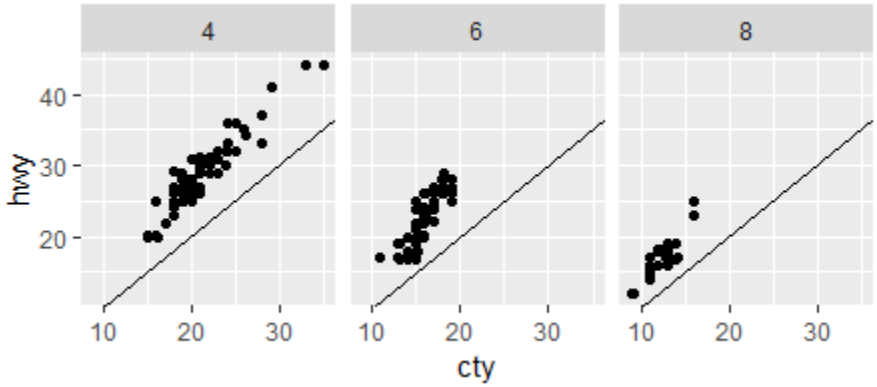
For both facet_wrap (and facet grid) you can control whether the position scales are the same in all panels (fixed) or allowed to vary between panels (free) with the scales parameter:

- scales = "fixed": x and y scales are fixed across all panels.
- scales = "free_x": the x scale is free, and the y scale is fixed.
- scales "free_y": the y scale is free, and the x scale is fixed.
- scales "free": x and y scales vary across panels.

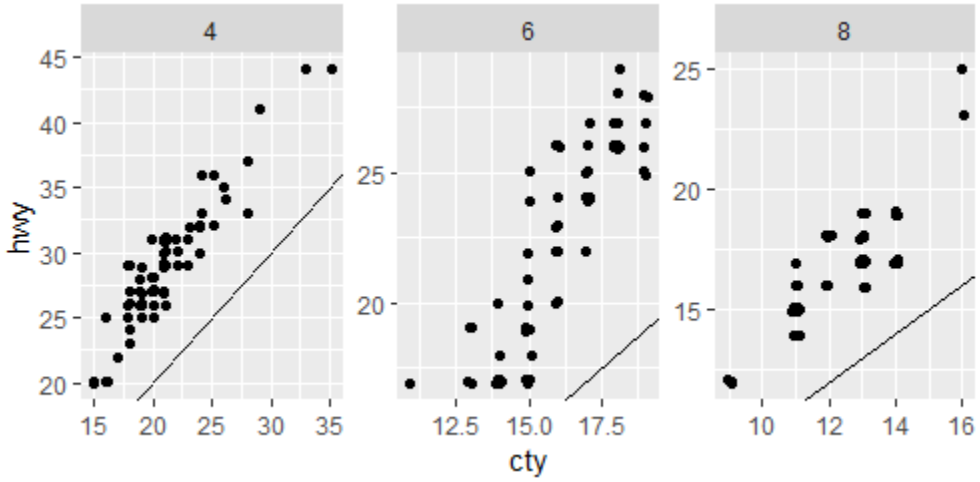
`facet.grid()` imposes an additional constraint on the scales: all panels in a column must have the same x scale, and all panels in a row must have the same y scale. This is because each column shares an x axis, and each row shares a y axis.

Fixed scales make it easier to see patterns across panels; free scales make it easier to see patterns within panels.

```
p <- ggplot(mpg2, aes(cty, hwy)) +
geom_abline() +
geom_jitter(width = 0.1, height = 0.1)
p + facet_wrap(~ cyl)
```




```
p + facet_wrap (~cyl, scales = "free")
```

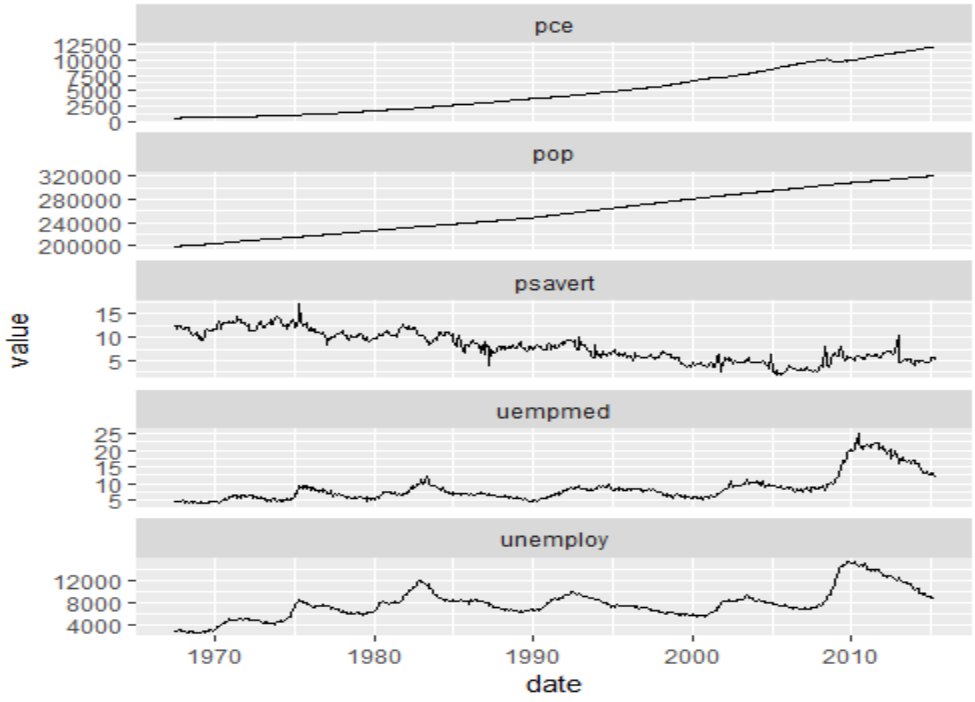


Free scales are also useful when we want to display multiple time series that were measured on different scales. To do this, we first need to change from 'wide to long' data, stacking the separate variables into a single column. An example of this is shown below with the long form of `economic` data.

```
economics_long
# A tibble: 2,870 x 4
```

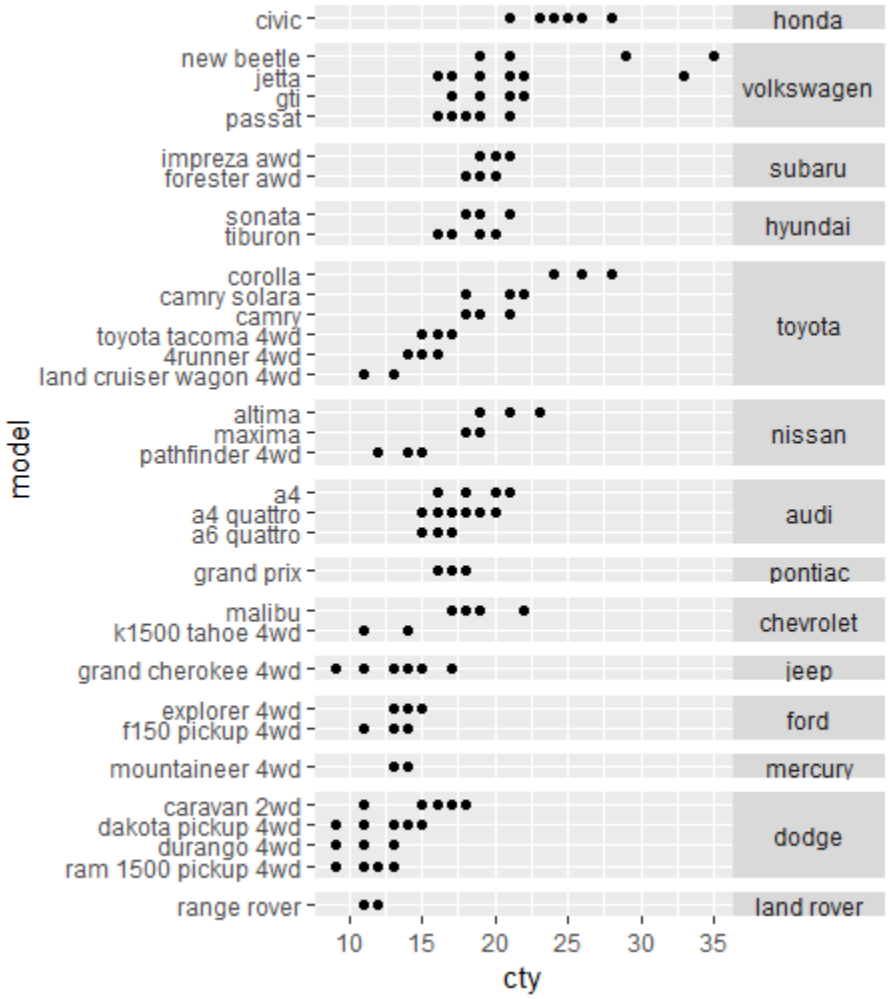
```
# Groups:   variable [5]
   date       variable value  value01
   <date>     <fct>    <dbl>    <dbl>
 1 1967-07-01 pce       507. 0
 2 1967-08-01 pce       510. 0.000266
 3 1967-09-01 pce       516. 0.000764
 4 1967-10-01 pce       513. 0.000472
 5 1967-11-01 pce       518. 0.000918
 6 1967-12-01 pce       526. 0.00158
 7 1968-01-01 pce       532. 0.00207
 8 1968-02-01 pce       534. 0.00230
 9 1968-03-01 pce       545. 0.00322
10 1968-04-01 pce       545. 0.00319
# ... with 2,860 more rows
```

```
ggplot(economics_long, aes(date, value)) +
  geom_line() +
  facet_wrap(~variable, scales = "free_y", ncol = 1)
```



`facet_grid()` has an additional parameter called space, which takes the same values as scales. When space is "free", each column (or row) will have width (or height) proportional to the range of the scale for that column (or row). This makes the scaling equal across the whole plot: 1 cm on each panel maps to the same range of data. (This is somewhat analogous to the sliced axis limits of lattice.) For example, if panel had range 2 and panel b had range 4, one-third of the space would be given to a, and two-thirds to b. This is most useful for categorical scales, where we can assign space proportionally based on the number of levels in each facet, as illustrated below.

```
mpg2$model <- reorder(mpg2$model, mpg2$cty)
mpg2$manufacturer <- reorder(mpg2$manufacturer, -mpg2$cty)

ggplot(mpg2, aes(cty, model)) +
  geom_point() +
  facet_grid(manufacturer ~ ., scales = "free", space = "free") +
  theme(strip.text.y = element_text(angle = 0))
```

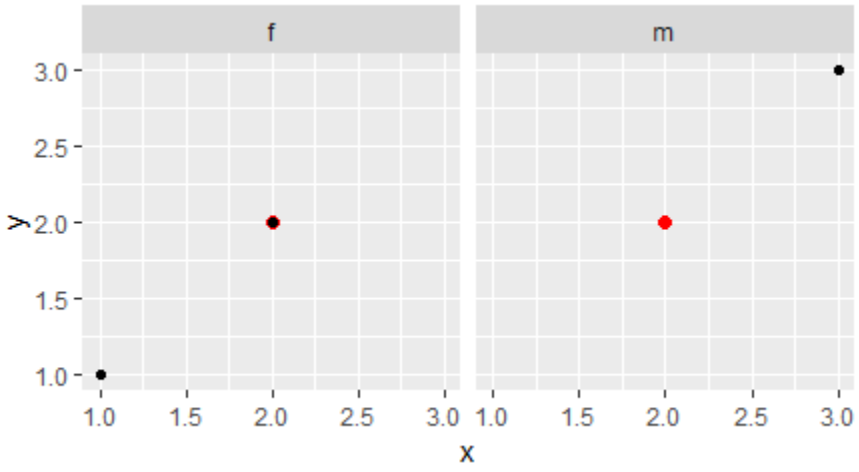


Missing Facetting Variables

If you are using facetting on a plot with multiple datasets, what happens when one of those datasets is missing the facetting variables? This situation commonly arises when you are adding contextual information that should be the same in all panels. For example, imagine you have a spatial display of disease faceted by gender. What happens when you add a map layer that does not contain the gender variable? Here ggplot will do what you expect: it will display the map in every facet: missing facetting variables are treated like they have all values.

Here's a simple example. Note how the single red point from `df2` appears in both panels.

```
df1 <-data.frame(x = 1:3, y = 1:3, gender = c("f", "f", "m"))
df2 <- data.frame(x = 2, y = 2)

ggplot(df1, aes(x, y)) +
  geom_point(data = df2, colour = "red", size = 2) +
  geom_point() +
  facet_wrap(~gender)
```

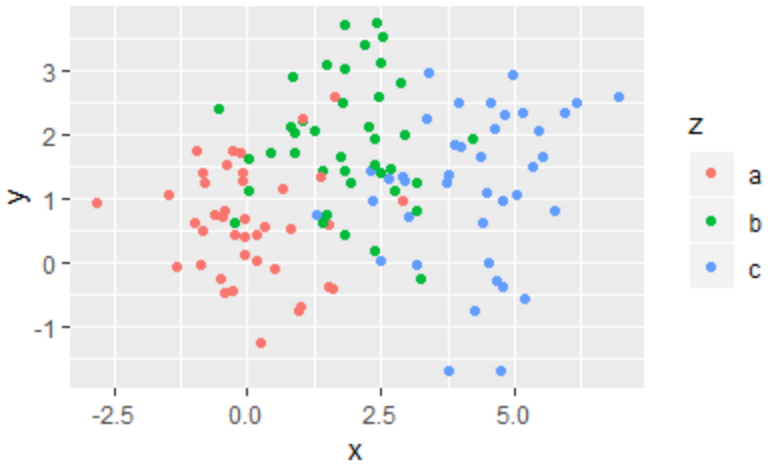


This technique is particularly useful when you add annotations to make it easier to compare between facets.

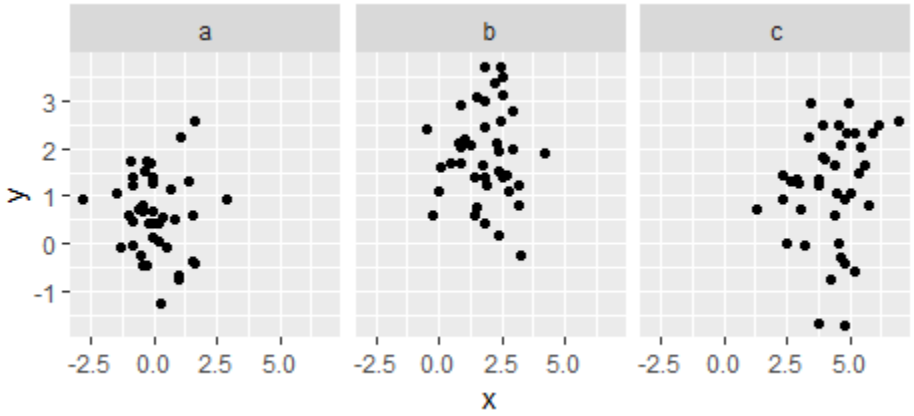
Grouping vs. Facetting

Facetting is an alternative to using aesthetics (like colour, shape or size) to differentiate groups. Both techniques have strengths and weaknesses, based around the relative positions of the subsets. With facetting, each group is quite far apart in its own panel, and there is no overlap between the groups. This is good if the groups overlap a lot, but it does make small differences harder to see. When using aesthetics to differentiate groups, the groups are close together and may overlap, but small differences are easier to see.

```
df <- data.frame(
x = rnorm(120, c(0, 2, 4)),
y = rnorm(120, c(1, 2, 1)),
z = letters[1:3]
)

ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z))
```

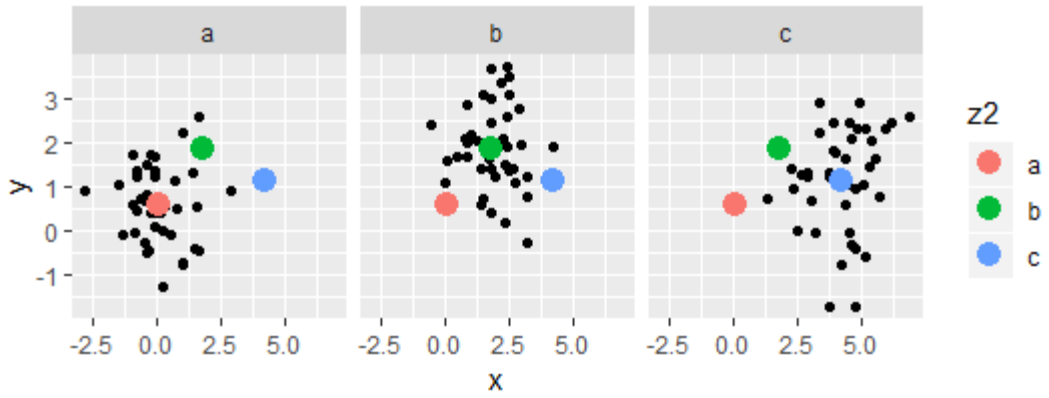



```
ggplot(df, aes(x, y)) +
geom_point() +
facet_wrap(~z)
```



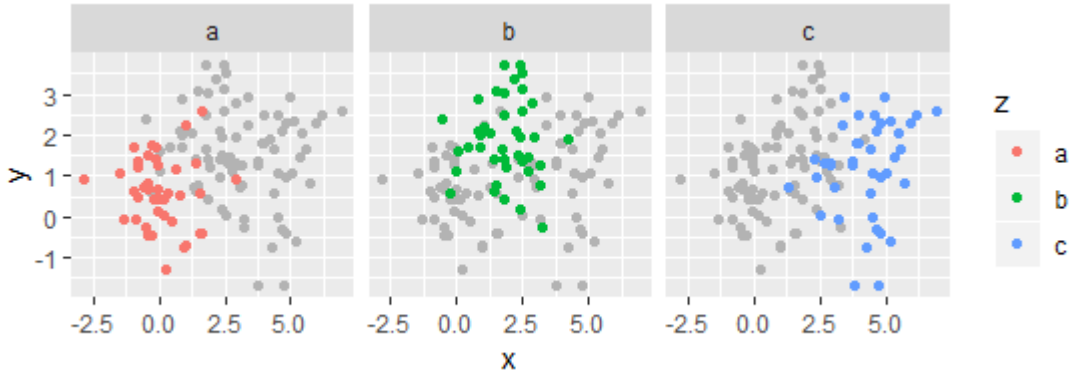
Comparisons between facets often benefit from some thoughtful annotation. For example, in this case, we could show the mean of each group in every panel. Note that we need two "z" variables: one for the facets and one for the colours.

```
df_sum <- df %>%
  group_by(z) %>%
  summarise(x = mean(x), y = mean(y)) %>%
  rename(z2 = z)
ggplot(df, aes(x, y)) +
  geom_point() +
  geom_point(data = df_sum, aes(colour= z2), size = 4) +
  facet_wrap(~z)
```



Another useful technique is to put all the data in the background of each panel:

```
df2 <- dplyr::select (df, -z)

ggplot(df, aes(x, y)) +
  geom_point(data = df2, colour = "grey70") +
  geom_point(aes(colour = z)) +
  facet_wrap(~z)
```



Continuous Variables

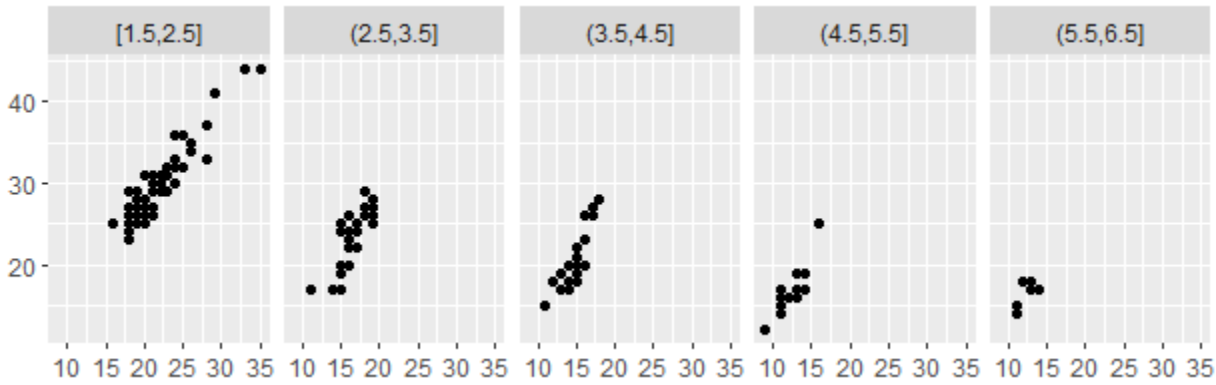
To facet continuous variables, you must first discretize them. ggplot2 provides three helper functions to do so:

- Divide the data into n bins each of the same length: `cut_interval(x, n)`
- Divide the data into bins of width width: `cut_width(x, width)`.
- Divide the data into n bins each containing (approximately) the same number of points: `cut_number(x, n = 10)`.

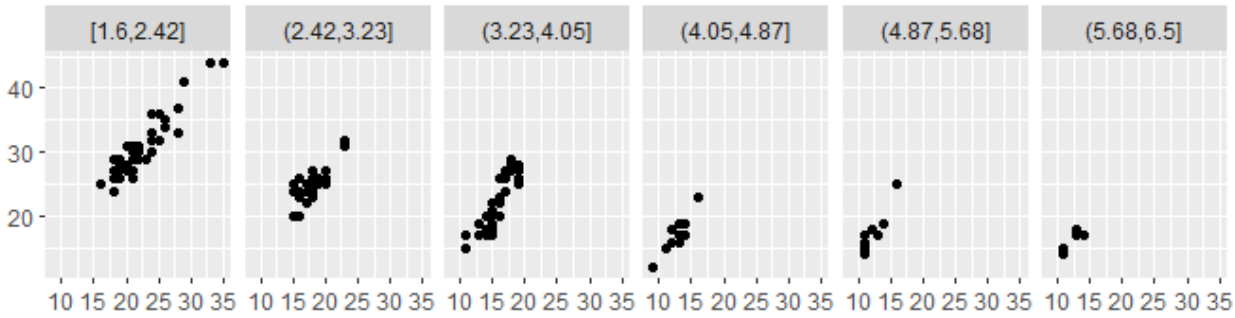
They are illustrated below:

```
#Bins of width 1
mpg2$ disp_w <- cut_width (mpg2$displ,1)
```

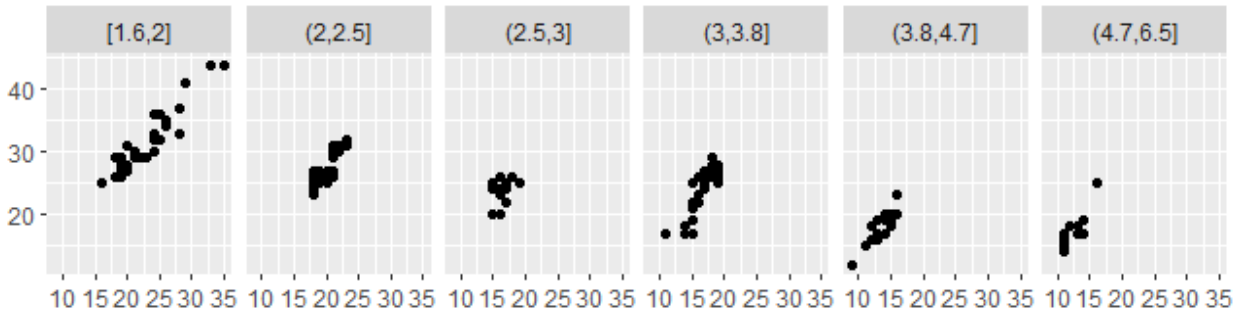
```
#Six bins of equal length
mpg2$disp_i <- cut_interval (mpg2$displ, 6)

# Six bins containing equal numbers of points
mpg2$disp_n <- cut_number(mpg2$displ, 6)

plot <- ggplot (mpg2, aes(cty, hwy)) +
 geom_point() +
 labs(x = NULL, y = NULL)

plot + facet_wrap (~disp_w, nrow = 1)
```




```
plot + facet_wrap (~disp_i, nrow = 1)
```




```
plot + facet_wrap (~disp_n, nrow = 1)
```



Note that the facetting formula does not evaluate functions, so you must first create a new variable containing the discretized data.

Coordinate Systems

Coordinate systems have two main jobs:

- Combine the two position aesthetics to produce a 2d position on the plot. The position aesthetics are called x and y, but they might be better-called position 1 and 2 because their meaning depends on the coordinate system used. For example, with the polar coordinate system, they become angle and radius (or radius and angle), and with maps, they become latitude and longitude.

- In coordination with the faceter, coordinate systems draw axes and panel backgrounds. While the scales control the values that appear on the axes and how they map from data to position, it is the coordinate system which actually draws them. This is because their appearance depends on the coordinate system: an angle axis looks quite different than an x axis.

There are two types of coordinate system. Linear coordinate systems preserve the shape of geoms:

- `coord_cartesian()`: the default Cartesian coordinate system, where the 2d position of an element is given by the combination of the x and y positions.
- `coord_flip()`: Cartesian coordinate system with x and y axes flipped.
- `coord_fixed()`: Cartesian coordinate system with a fixed aspect ratio.

  On the other hand. non-linear coordinate systems can change the shapes: a straight line may no longer be straight. The closest distance between two points may no longer be a straight line.

- `coord_map()/coord.quickmap()`: Map projections.
- `coord_polar()`: Polar coordinates.
- `coord_trans()`: Apply arbitrary transformations to x and y positions, after the data has been processed by the stat.

Linear Coordinate Systems

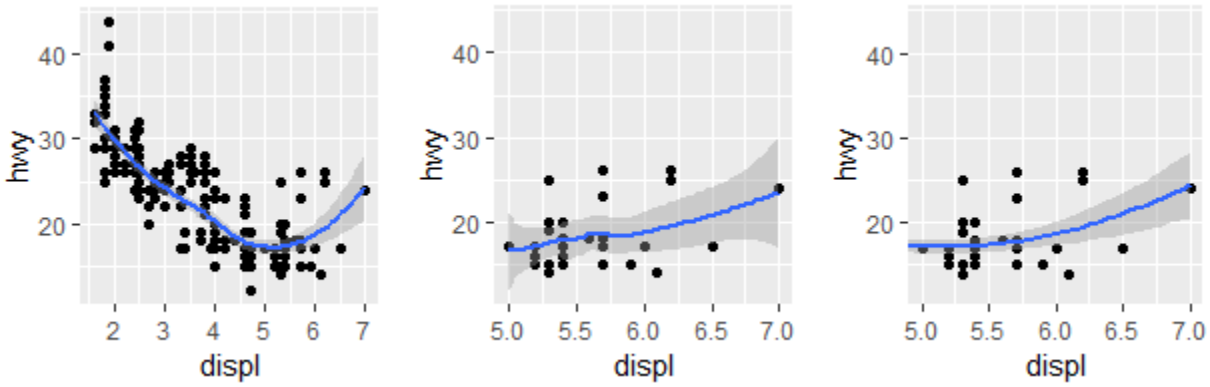
There are three linear coordinate systems: `coord_cartesian()`, `coord_flip()`, `coord_fixed()`.

Zooming into a Plot with `coord.cartesian()`

`coord_cartesian()` has arguments `xlim` and `ylim`. If you think back to the scales, you might wonder why we need these. Doesn't the limits argument of the scales already allow us to control what appears on the plot? The key difference is how the limits work: when setting scale limits, any data outside the limits is thrown away but when setting coordinate system limits we still use all the data, but we only display a small region of the plot. Setting coordinate system limits is like looking at the plot under a magnifying glass.

```
base <- ggplot(mpg, aes(displ, hwy))+
geom_point()+
```

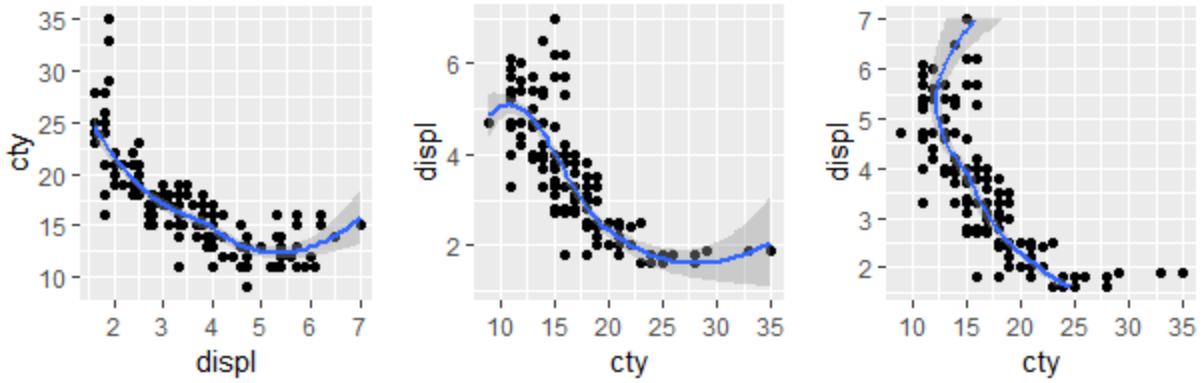
```
geom_smooth()

# Full dataset
base
# Scaling to 5--7 throws away data outside that range
base + scale_x_continuous(limits = c(5, 7))
#> Warning: Removed 196 rows containing non-finite values

#> (stat_smooth)
#> Warning Removed 196 rows containing missing values (geom_point).
# Zooming to 5--7 keeps all the data but only shows some of it
base + coord_cartesian(xlim = c(5, 7))
```



***Flipping the Axis with* `coord_flip()`**

Most statistics and geoms assume you are interested in y values conditional on x values (e.g., smooth, summary, boxplot, line): in most statistical models, the x values are assumed to be measured without error. If you are interested in x conditional on y (or you just want to rotate the plot 90 degrees), you can use `coord_flip()` to exchange the x and y axes. Compare this with just exchanging the variables mapped to x and y:

```
ggplot(mpg, aes (displ, cty))+
  geom_point()+
  geom_smooth()
#  Exchanging cty and displ rotates the plot 98 degrees, but the
smooth is fit to the rotated data.
ggplot(mpg, aes (cty, displ)) +
  geom_point() +
  geom_smooth()
#coord_flip() fits the smooth to the original data, and then rotates
the output
ggplot(mpg, aes(displ, cty)) +
  geom_point()+
  geom_smooth()+
  coord_flip()
```



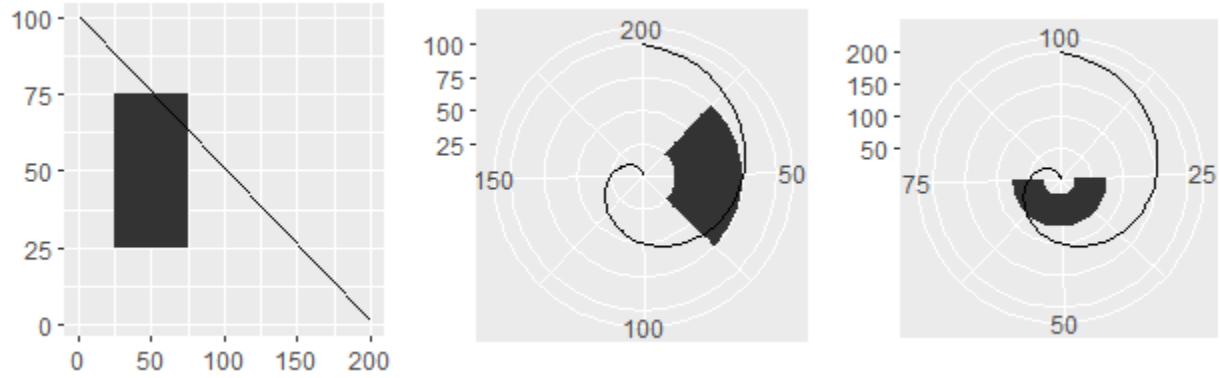
***Equal Scales with* `coord.fixed()`**

`coord_fixed()` fixes the ratio of length on the x and y axes. The default ratio ensures that the x and y axes have equal scales: i.e., 1 cm along the x axis represents the same range of data as 1 cm along the y axis. The aspect ratio will also be set to ensure that the mapping is maintained regardless of the shape of the output device. See the documentation of `coord_fixed()` for more details.

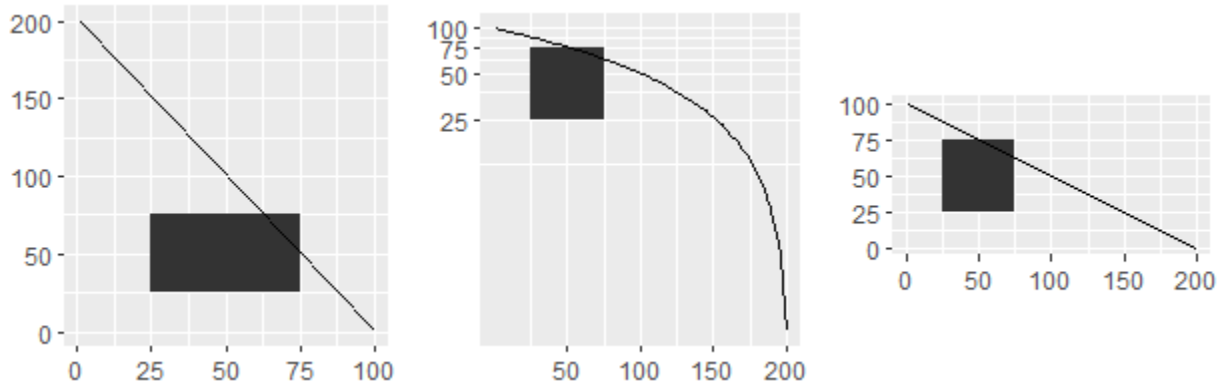
Non-linear Coordinate Systems

Unlike linear coordinates, non-linear coordinates can change the shape of geoms. For example, in polar coordinates a rectangle becomes an arc; in a map projection, the shortest path between two points is not necessarily a straight line. The code below shows how a line and a rectangle are rendered in a few different coordinate systems.

```
rect <- data.frame(x = 50, y = 50)
line <- data.frame(x = c(1, 200), y = c(100, 1))
base <- ggplot(mapping = aes(x, y)) +
  geom_tile(data = rect, aes(width = 50, height = 50)) +
  geom_line(data = line) +
  xlab(NULL) + ylab(NULL)
base
base + coord_polar("x")
base + coord_polar("y")
```



```
base + coord_flip()
base + coord_trans(y = "log10")
base + coord_fixed()
```

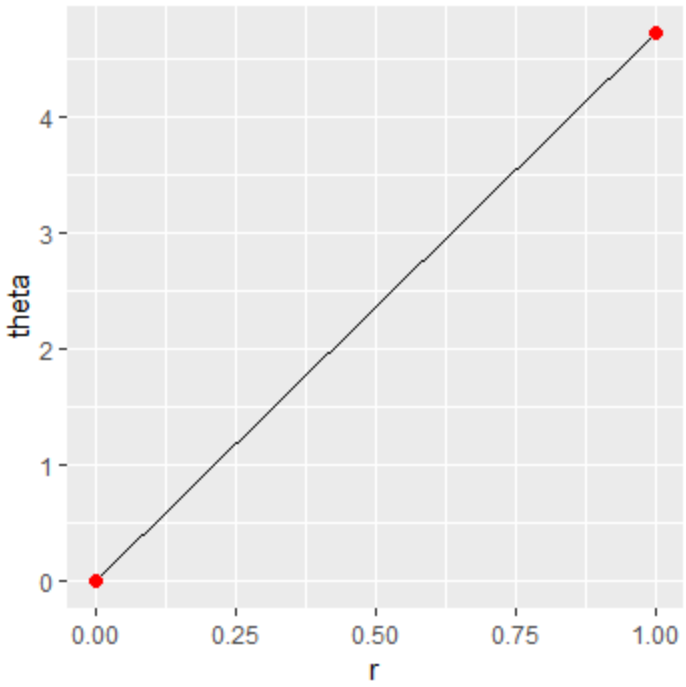


The transformation takes part in two steps. Firstly, the parameterization of each geom is changed to be purely location-based, rather than location and dimension-based. For example, a bar can be represented as an x position (a location), a height and a width (two dimensions). Interpreting height and width in a non-Cartesian coordinate system is hard because a rectangle may no longer have constant height and width, so we convert to a purely location based representation, a polygon defined by the four corners. This effectively converts all geoms to a combination of points, lines and polygons.

Once all geoms have a location-based representation, the next step is to transform each location into the new coordinate system. It is easy to transform points, because a point is still a point no matter what coordinate system you are in. Lines and polygons are harder, because a straight line may longer be straight in the new coordinate system. To make the problem tractable, we assume that all coordinate transformations are smooth, in the sense that all very short lines will still be very short straight lines in the new coordinate system. With this assumption in hand, we can transform lines and polygons by breaking them up into many small line segments and transforming each segment. This process is called munching and is illustrated below:

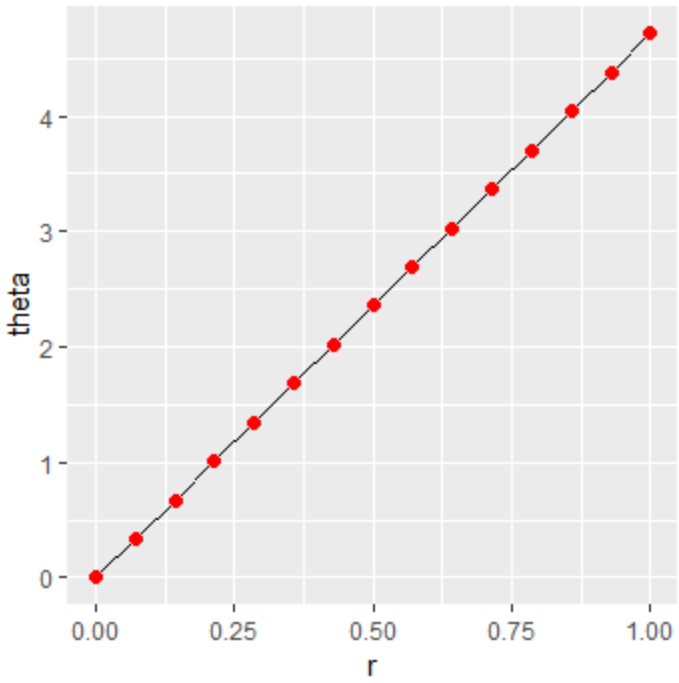
1. We start with a line parameterized by its two endpoints:

```
df <- data.frame(r = c(0, 1), theta = c(0, 3/2 * pi))
ggplot(df, aes(r, theta)) +
   geom_line () +
   geom_point (size = 2, colour = "red")
```



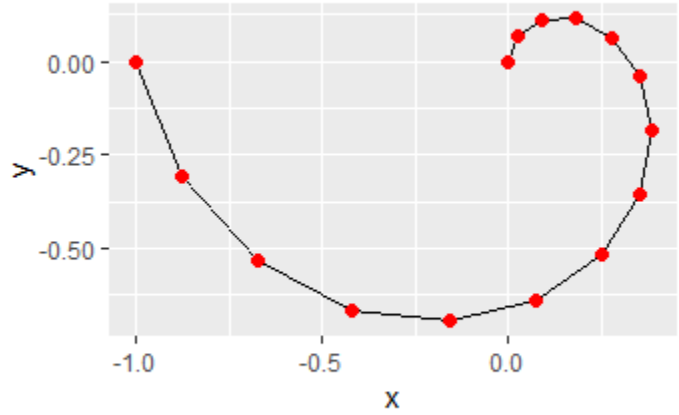
2. We break it into multiple line segments, each with two endpoints.

```
interp <- function(rng, n) {
      seq(rng[1], rng[2], length = n)
}
munched <- data.frame(
      r =  interp(df$r, 15),
      theta = interp(df$theta, 15)
)

ggplot(munched, aes(r, theta)) +
      geom_line() +
      geom_point(size = 2, colour ="red" )
```



3. We transform the locations of each piece:

```
transformed <- transform(munched,
      x = r * sin(theta),
      y = r * cos(theta)
)

ggplot(transformed, aes(x, y)) +
      geom_path() +
      geom_point(size = 2, colour = "red") +
      coord_fixed()
```

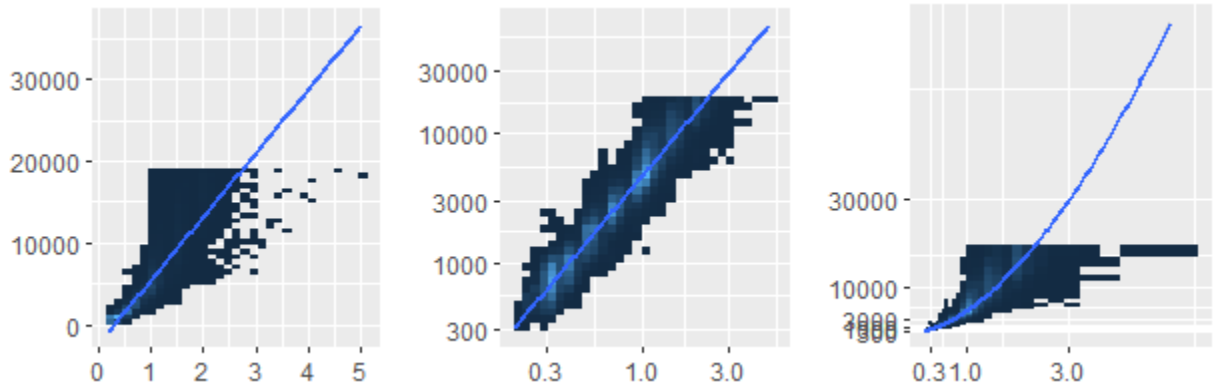


Internally ggplot2 uses many more segments so that the result looks smooth.

***Transformations with* `coord_trans()`**

Like limits, we can also transform the data in two places: at the scale level or at the coordinate system level. `coord_trans()` has arguments x and y which should be strings naming the transformer or transformer objects. Transforming at the scale level occurs before statistics are computed and does not change the shape of the geom. Transforming at the coordinate system level occurs after the statistics have been computed, and does affect the shape of the geom. Using both together allows us to model the data on a transform scale and then back-transform it for interpretation: a common pattern in the analysis.

```
#Linear model on original scale is poor fit
base <- ggplot(diamonds, aes(carat, price))+
  stat_bin2d() +
  geom_smooth(method = "lm”)+
  xlab(NULL) +
  ylab(NULL) +
  theme (legend.position = "none")

base

# Better fit on log scale, but harder to interpret
base +
   scale_x_log10() +
   scale_y_log10()

# Fit on log scale, then backtransform to original.
# Highlights lack of expensive diamonds with large carats
pow10 <- scales::exp_trans(10)
base +
   scale_x_log10() +
   scale_y_log10() +
   coord_trans(x = pow10, y = pow10)
```

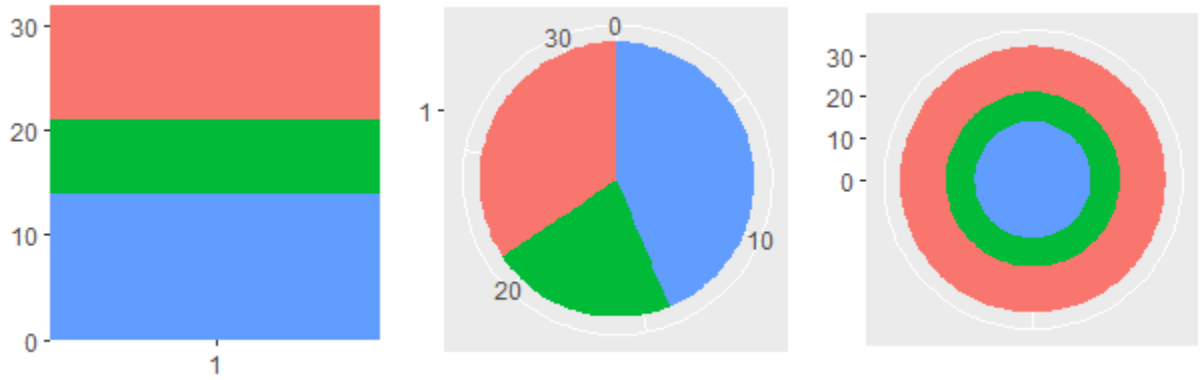


***Polar Coordinates with* `coord_polar()`**

Using polar coordinates gives rise to pie charts and wind roses (from bar geoms), and radar charts (from line geoms). Polar coordinates are often used for circular data, particularly time or direction, but the perceptual properties are not good because the angle is harder to perceive for small radii than it is for large radii. The theta argument determines which position variable is mapped to angle (by default, x) and which to radius.

The code below shows how we can turn a bar into a pie chart or a bullseye chart by changing the coordinate system.

```
base <- ggplot(mtcars, aes(factor(1), fill = factor (cyl))) +
  geom_bar(width = 1) +
  theme (legend.position = "none") +
  scale_x_discrete(NULL, expand =c(0, 0)) +
  scale_y_continuous(NULL, expand = c(0, 0)

# Stacked barchart
base

# Pie chart
base + coord_polar(theta = "y")

# The bullseye chart
base + coord_polar()
```



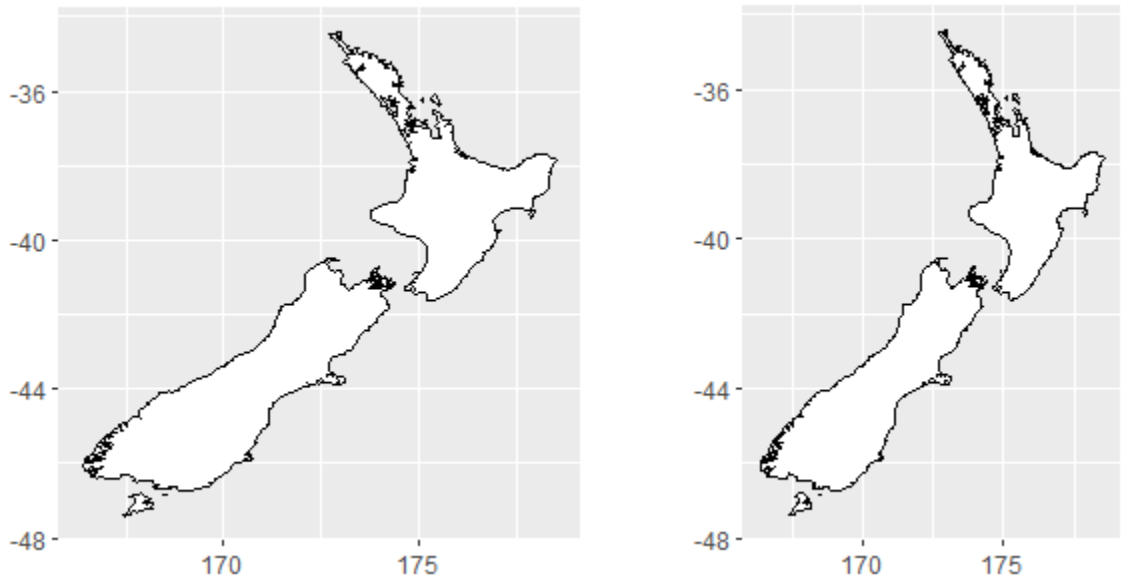
***Map Projections with* `coord_map()`**

Maps are intrinsically displays of spherical data. Simply plotting raw longitudes and latitudes is misleading, so we must project the data. There are two ways to do this with ggplot2:

- `coord_quickmap()` is a quick and dirty approximation that sets the aspect ratio to ensure than 1m of latitude and 1m of longitude are the same distance in the middle of the plot. There are a reasonable place to start for smaller regions and is very fast.

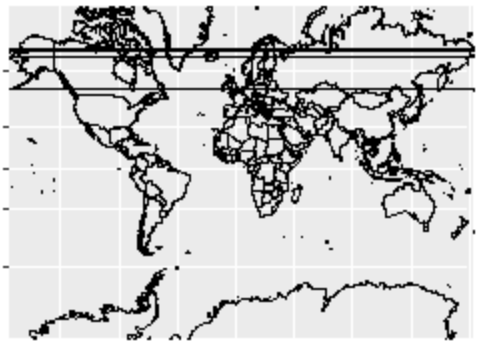
  ```
  # Prepare a map of NZ
  nzmap <- ggplot(map_data("nz"), aes(long, lat, group = group))
  +
  ```

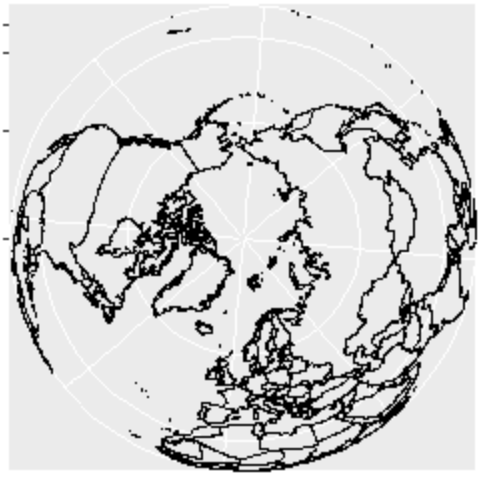
```
geom_polygon(fill = "white", colour = "black") +
xlab(NULL) + ylab(NULL)

# Plot it in cartesian coordinates
nzmap
# With the aspect ratio approximation
nzmap + coord_quickmap()
```



- `coord_map()` uses the **mapproj** package, https://cran.r-project.org/ `package = mapproj` to do a formal map projection. It takes the same arguments as `mapproj::mapproject()` for controlling the projection. It is much slower than `coord_quickmap()` because it must munch the data and transform each piece.

```
world <- map_data("world")
worldmap <- ggplot(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 + coord_map()
# Some crazier projections
worldmap + coord_map("ortho")
```





R Visualization
Module 3: Scales, Axes, and Legends

Learning objectives

1. You will learn about size, colour, position or shape to enhance the visual properties of your plot.
2. Common scale arguments like axes and legends.
3. Specifying limits that are derived from the range of the data.

Introduction

Scales control the mapping from data to aesthetics. They take your data and turn it into something that you can see, like size, colour, position or shape. Scales also provide the tools that let you read the plot: axes and legends. Formally, each scale is a function from a region in data space (the domain of the scale) to a region in aesthetic space (the range of the scale). The axis or legend is the inverse function it allows you to convert visual properties back to data.

You can generate many plots without knowing how scales work but understanding scales and learning how to manipulate them will give you much more control.

Modifying Scales

A scale is required for every aesthetic used on the plot. When you write:

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

What actually happens is this:

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

Default scales are named according to the aesthetic and the variable type: `scale_y_continuous()`, `scale_colour_discrete()`, etc.

It would be tedious to manually add a scale every time you used a new aesthetic, so ggplot2 does it for you But if you want to override the defaults, you'll need to add the scale yourself like this:

```
  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")
```

The use of + to "add" scales to a plot is a little misleading. When you + a scale, you're not actually adding it to the plot, but overriding the existing scale. This means that the following two specifications are equivalent:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_x_continuous("Label 1") +
  scale_x_continuous("Label 2")
#> Scale for is already present. Adding another scale for 'x',
#> Which will replace the existing scale.
```

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

```
scale_x_continuous("Label 2")
```

Note the message: if you see this in your own code, you need to reorganize your code specification to only add a single scale. You can also use a different scale altogether:

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

You've probably already figured out the naming scheme for scales for scales, but to be concrete, it's made up of three pieces separated by "_":

1. scale
2. The name of the aesthetic (e.g., colour, shape or x)
3. The name of the scale (e.g., continuous, discrete, brewer).

Guides: Legends and Axes

The component of a scale that you're most likely to want to modify is the **guide**, the axis or legend associated with the scale. Guides allow you to read observations from the plot and map them back to their original values. In ggplot2, guides are produced automatically based on the layers in your plot. This is very different to base R graphics, where you are responsible for drawing the legends by hand. In ggplot2, you don't directly control the legend; instead, you set up the data so that there's a clear mapping between data and aesthetics, and a legend is generated for you automatically.

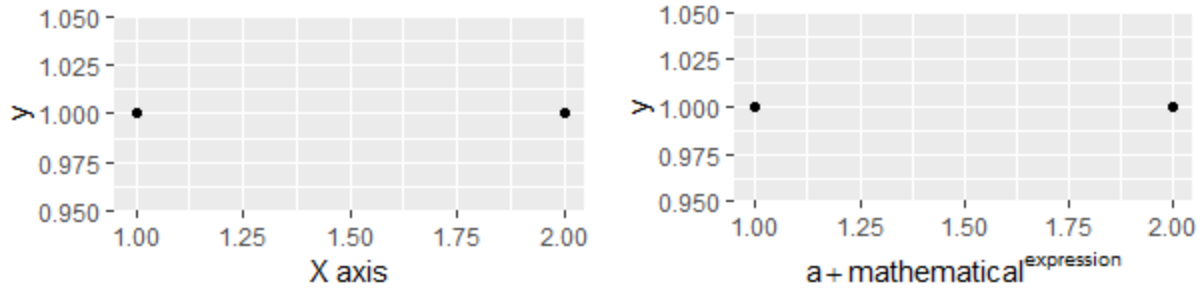
You might find it surprising that axes and legends are the same type of thing, but while they look very different there are many natural correspondences between the two, as shown in the table below:

Axis	Legend	Argument name
Label	Title	name
Ticks & grid line	Key	breaks
Tick label	Key label	labels

Scale Title

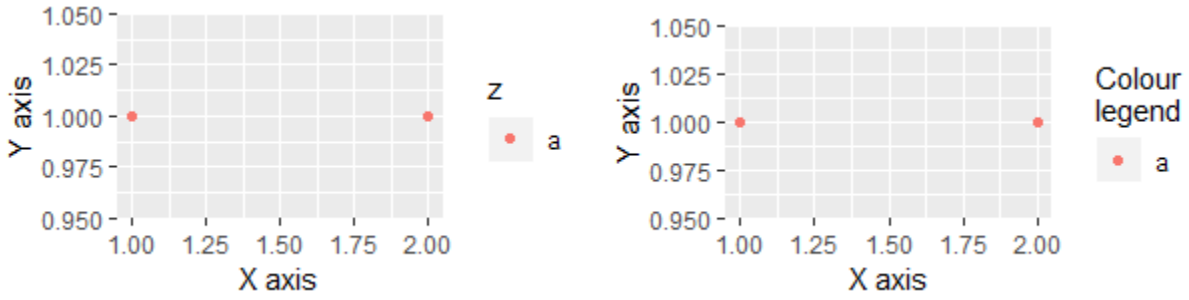
The first argument to the scale function, name, the axes/legend title, You can supply text strings (using \n for the breaks) or mathematical expressions in `quote()` (as described in `?plotmath`):

```
df <- data.frame (x = 1:2, y = 1, z = "a")
p <- ggplot(df, aes(x, y)) + geom_point()
p + scale_x_continuous("X axis")
p + scale_x_continuous(quote(a + mathematical ^ expression))
```



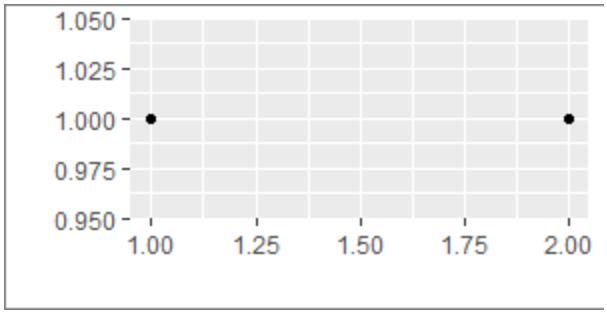
Because tweaking these labels is such a common task, there are three helpers that save you some typing: `xlab()`, `ylab()` and `labs()`:

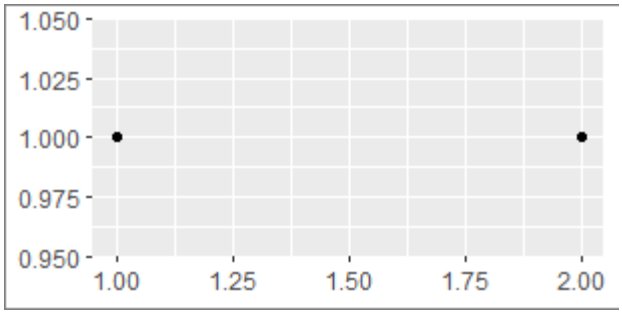
```
p <- ggplot(df, aes(x, y)) + geom_point(aes (colour = z))
p +
  xlab("X axis") +
  ylab("Y axis")
p + labs(x="X axis", y = "Y axis", colour = "Colour\nlegend")
```



There are two ways to remove the axis label. Setting it to " " omits the label, but still allocates space; NULL removes the label and its space. Look closely at the left and bottom borders of the following two plots. I've drawn a grey rectangle around the plot to make it easier to see the difference.

```
p <- ggplot(df, aes(x, y)) +
     geom_point() +
     theme(plot.background = element_rect(colour = "grey50"))
p + labs(x = "", y = "")
p + labs(x = NULL, y = NULL)
```



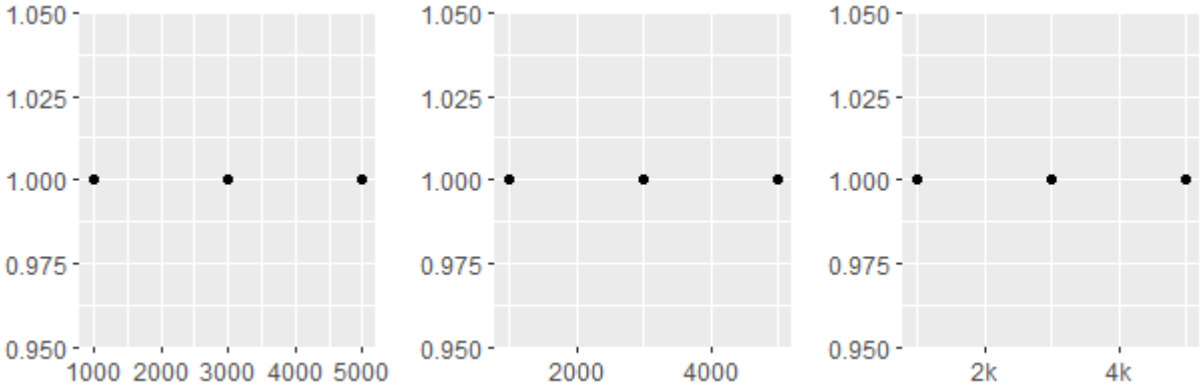


Breaks and Labels

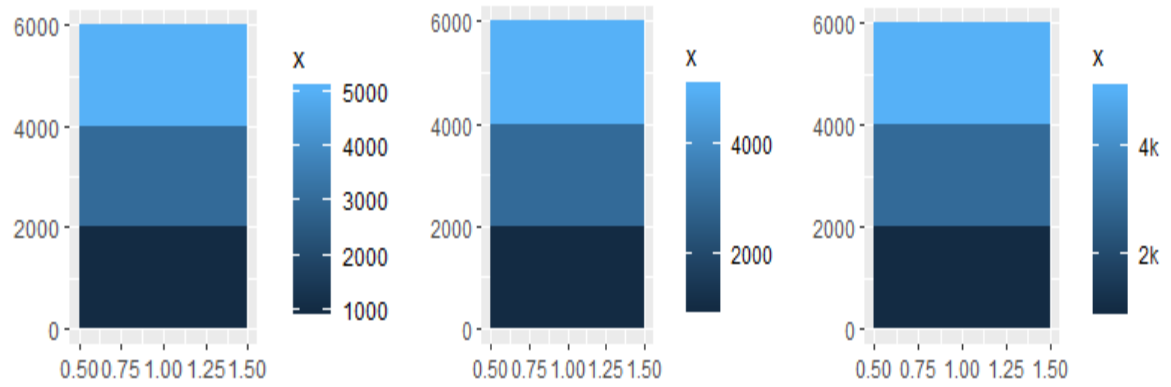
The breaks argument controls which values appear as tick marks on axes and keys on legends. Each break has an associated label, controlled by the labels argument. If you set labels, you must also set breaks; otherwise, if data changes, the breaks will no longer align with the labels.

The following code shows some basic examples for both axes and legends.

```
df  <- data.frame(x = c(1, 3, 5) * 1000, y = 1)
axs <- ggplot(df, aes(x, y)) +
  geom_point() +
  labs(x = NULL, y = NULL)
axs
axs + scale_x_continuous(breaks = c(2000, 4000))
axs  +  scale_x_continuous(breaks  =  c(2000,4000),    labels  =
c("2k", "4k"))
```

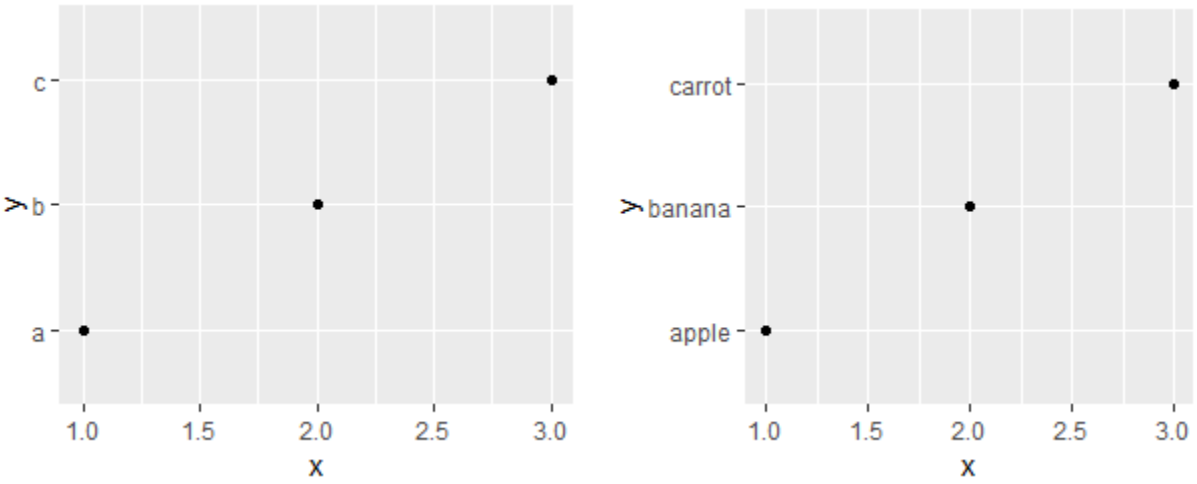



```
leg <- ggplot(df, aes(y, x, fill = x)) +
  geom_tile() +
  labs(x = NULL, y = NULL)
leg
leg + scale_fill_continuous(breaks = c(2000, 4000))
leg  +  scale_fill_continuous(breaks  =  c(2000,  4000),  labels  =
c("2k", "4k"))
```



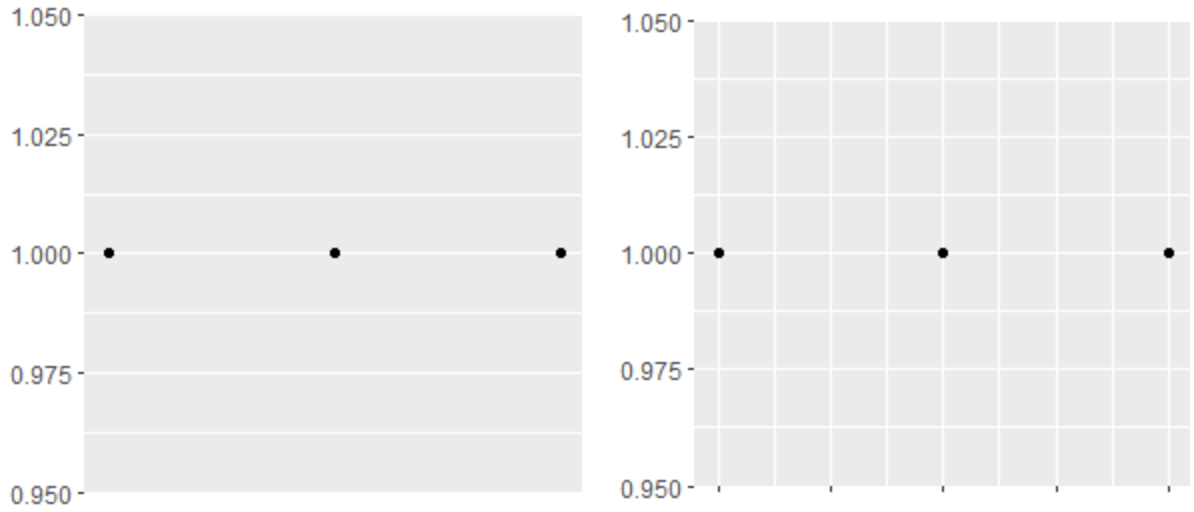
If you want to relabel the breaks in a categorical scale, you can use a name labels vector:

```
df2 <- data.frame(x = 1:3, y = c("a", "b", "c"))
ggplot(df2, aes(x, y)) +
  geom_point()
ggplot(df2, aes(x, y)) +
  geom_point() +
  scale_y_discrete(labels = c(a = "apple", b = "banana", c =
"carrot"))
```

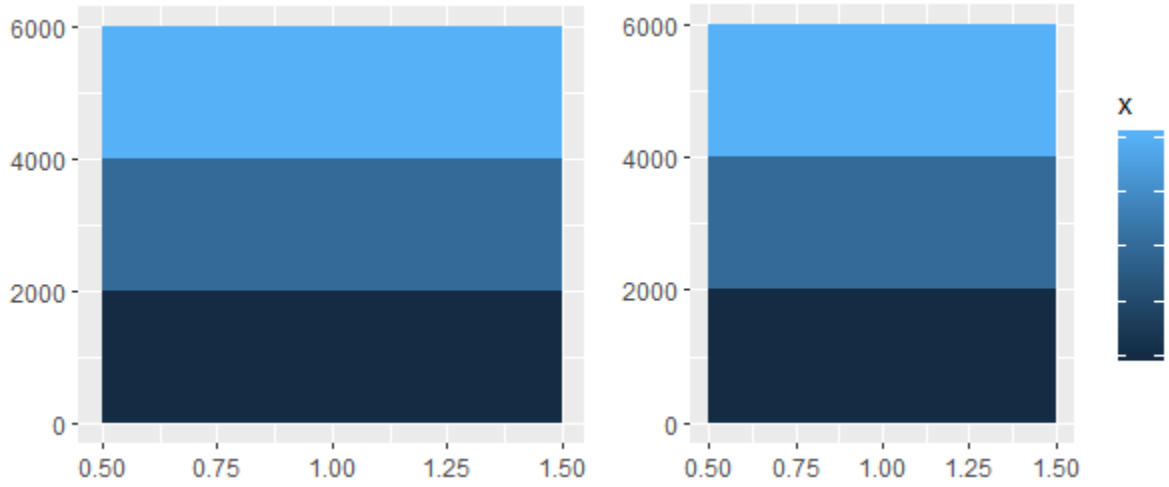


To suppress breaks (and for axes, grid lines) or labels, set them to NULL:

```
axs + scale_x_continuous(breaks = NULL)
axs + scale_x_continuous(labels = NULL)
```

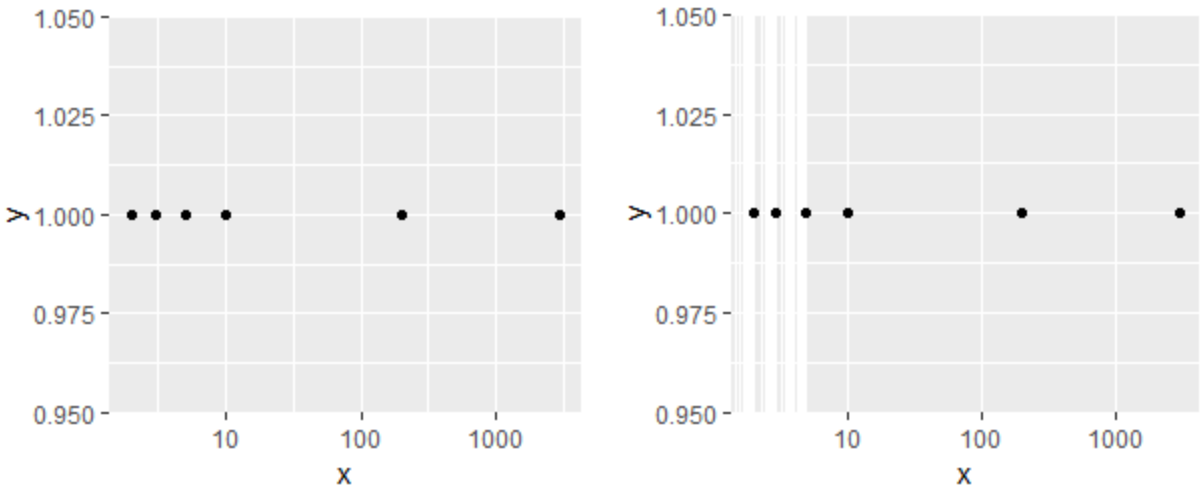



```
leg + scale_fill_continuous(breaks = NULL)
leg + scale_fill_continuous(labels = NULL)
```



You can adjust the minor breaks (the faint grid lines that appear between the major grid lines) by supplying a numeric vector of positions to the `minor_breaks` argument. This is particularly useful for log scales:

```
df <- data.frame(x = c(2, 3, 5, 10, 200,3000), y = 1)
ggplot(df, aes(x, y)) +
  geom_point() +
scale_x_log10()
mb <- as.numeric(1:10 %o% 10 ^ (0:4))
ggplot(df, aes(x, y)) +
  geom_point() +
  scale_x_log10(minor_breaks = log10(mb))
```



Note the use of `%0%` to quickly generate the multiplication table, and that the minor breaks must be supplied on the transformed scale.

Legends

While the most important parameters are shared between axes and legends, there are some extra options that only apply to legends. Legends are more complicated than axes because:

1. A legend can display multiple aesthetics (e.g. colour and shape), from multiple layers, and the symbol displayed in a legend varies based on the geom used in the laver.

2. Axes always appear in the same place. Legends can appear in different places, so you need some global way of controlling them.

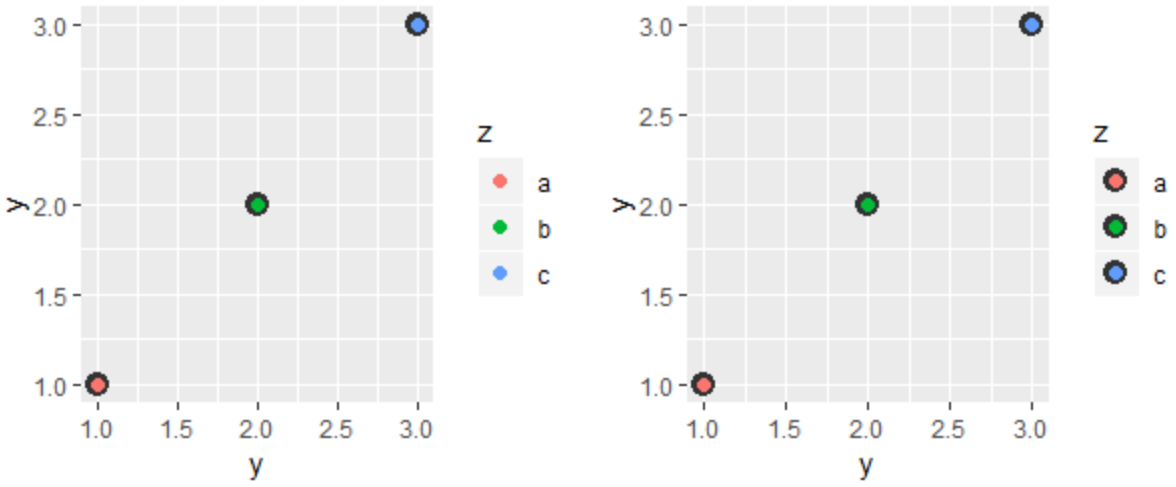
3. Legends have considerably more details that can be tweaked: should they be displayed vertically or horizontally? How many columns? How big should the keys be?

Layers and Legends

A legend may need to draw symbols from multiple layers. For example, if you've mapped colour to both points and lines, the keys will show both points and lines. If you've mapped fill colour, you get a rectangle.

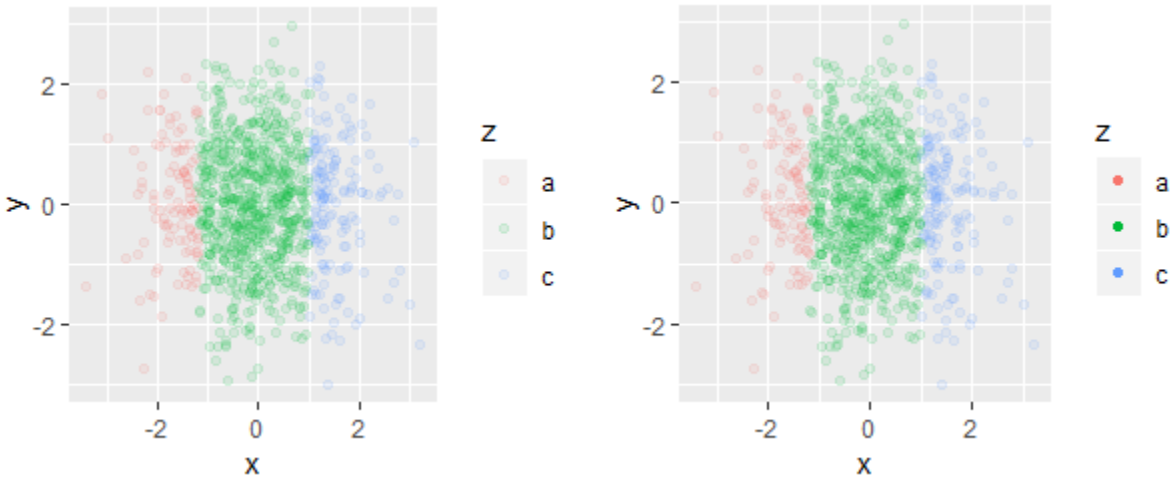
By default, a layer will only appear if the corresponding aesthetic is mapped to a variable with `aes()`. You can override whether or not a layer appears in the legend with show.legend: FALSE to prevent a layer from ever appearing in the legend; TRUE forces it to appear when it otherwise wouldn't. Using TRUE can be useful in conjunction with the following trick to make points stand out:

```
df <- data.frame(y = c(1,2,3), z = c("a","b","c"))
ggplot(df, aes(y, y)) +
  geom_point(size = 4, colour = "grey20") +
  geom_point(aes(colour = z), size = 2)
ggplot(df, aes(y, y)) +
  geom_point(size = 4, colour = "grey20", show.legend = TRUE) +
  geom_point(aes(colour = z), size = 2)
```



Sometimes you want the geoms in the legend to display differently to the geoms in the plot. This is particularly useful when you've used transparency or size to deal with moderate overplotting and also used colour in the plot. You can do this using the override.aes parameter of `guide_legend()`, which you'll learn more about shortly.

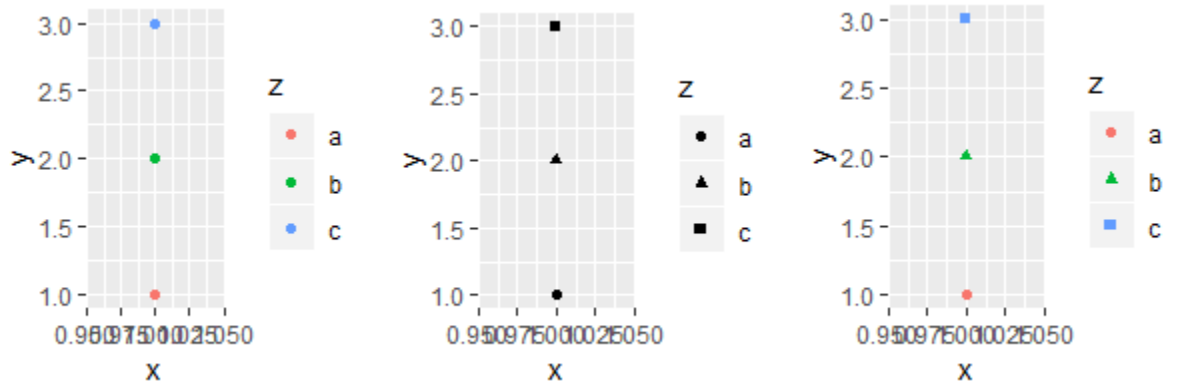
```
norm <- data.frame(x = rnorm(1000), y = rnorm(1000))
norm$z <- cut(norm$x, 3, labels = c("a", "b", "c"))
ggplot(norm, aes(x, y)) +
  geom_point(aes(colour = z), alpha = 0.1)
ggplot(norm, aes(x, y)) +
  geom_point(aes(colour = z), alpha = 0.1) +
  guides(colour = guide_legend(override.aes = list(alpha =1)))
```



ggplot2 tries to use the fewest number of legends to accurately convey the aesthetics used in the plot. It does this by combining legends where the same variable is mapped to different aesthetics. The figure below shows how this works for points: if both colour and shape are mapped to the same variable, then only a single legend is necessary.

```
df <- data.frame(x = 1, y = c(1,2,3), z = c("a","b","c"))
ggplot(df, aes(x, y)) + geom_point(aes(colour = z))
ggplot(df, aes(x, y)) + geom_point(aes(shape = z))
```

```
ggplot(df, aes(x, y)) + geom_point(aes(shape = z, colour = z))
```



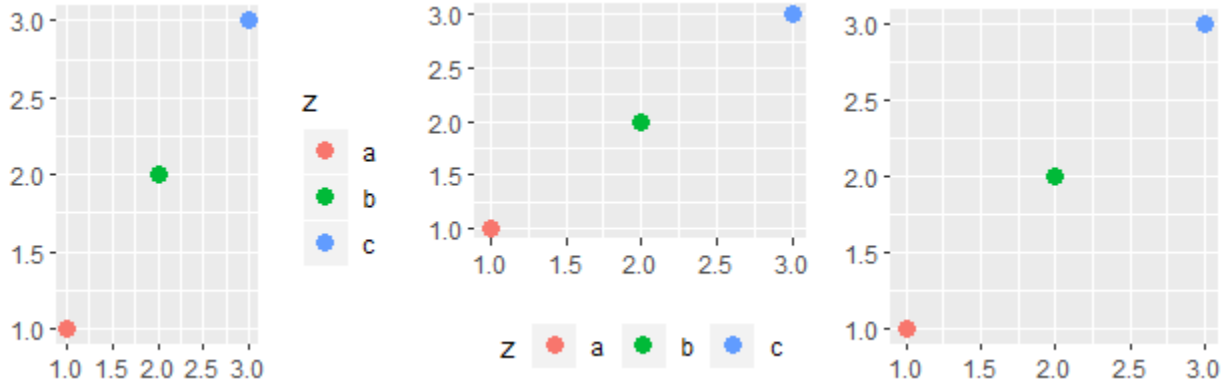
In order for legends to be merged, they must have the same name. So if you change the name of one of the scales, you'll need to change it for all of them.

Legend Layout

A number of settings that affect the overall display of the legends are controlled through the theme system, all you need to know is that you modify theme settings with the `theme()` function.

The position and justification of legends are controlled by the theme setting legend.position, which takes values "right", "left", "top", "bottom" or"none (no legend).

```
df <- data.frame(x = 1:3, y = 1:3, z = c("a", "b", "c"))
base  <- ggplot (df, aes(x, y)) +
  geom_point(aes(colour = z), size = 3) +
  xlab(NULL) +
  ylab(NULL)
base + theme (legend.position = "right") #the default
base + theme (legend.position = "bottom")
base + theme (legend.position = "none")
```



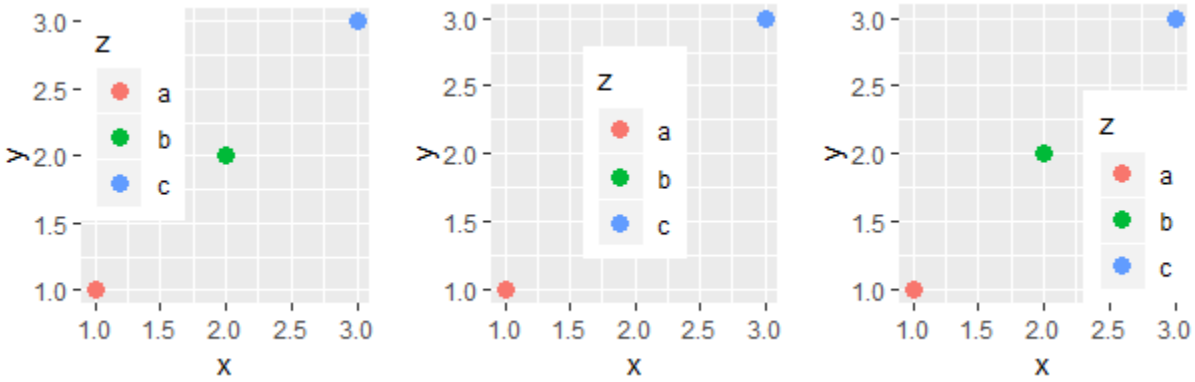
Switching between left/right and top/bottom modifies how the keys in each legend are laid out (horizontal or vertically), and how multiple legends are stacked (horizontal or vertically). If needed, you can adjust those options independently:

- `legend.direction`: layout of items in legends ("horizontal or "vertical").

- `legend.box`: arrangement of multiple legends ("horizontal" or "vertical").
- `legend.box.just`: justification of each legend within the overall bounding box, when there are multiple legends ("top", "bottom". "left", or "right").

Alternatively, if there's a lot of blank space in your plot you might want to place the legend inside the plot. You can do this by setting legend.position to a numeric vector of length two. The numbers represent a relative location in the panel area: `c(0, 1)` is the top-left corner and `c(1, 0)` is the bottom-right-corner. You control which corner of the legend the `legend.position` refers to with legend.justification, which is specified in a similar way. Unfortunately positioning the legend exactly where you want it requires a lot of trial and error.

```
base <- ggplot(df, aes(x, y)) +
          geom_point(aes(colour = z), size = 3)

base + theme(legend.position = c(0, 1), legend.justification =
c(0, 1))
base     +     theme(legend.position     =     c(0.5,     0.5),
legend.justification = c(0.5, 0.5))
base + theme(legend.position = c(1, 0), legend.justification =
c(1, 0))
```



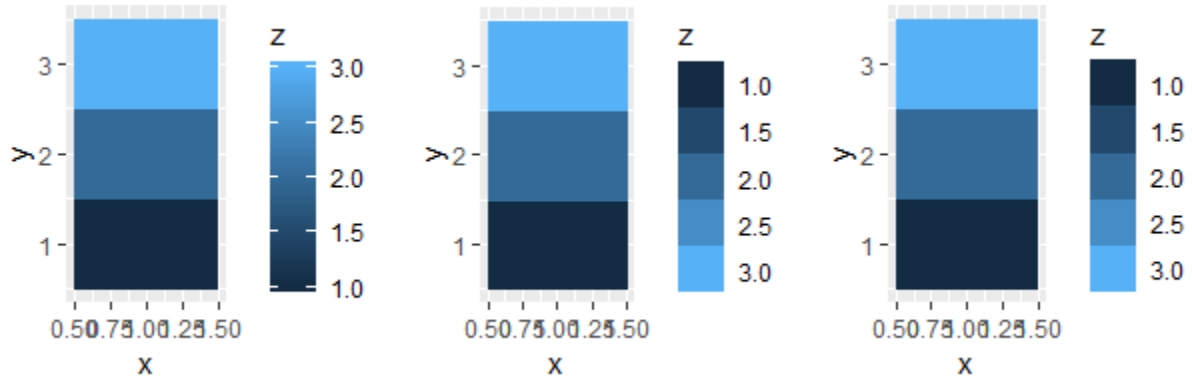
There's also a margin around the legends, which you can suppress with `legend.margin = unit(0, "mm")`

Guide Functions

The guide functions, `guide_colorbar()` and `guide_legend()`, offer additional control over the fine details of the legend. Legend guides can be used for any aesthetic (discrete or continuous) while the colour bar guide can only be used with continuous colour scales.

You can override the default guide using the guide argument of the corresponding scale function, or more conveniently, the `guides()` helper function. `guides()` works like `labs()`: you can override the default guide associated with each aesthetic.

```
df <- data.frame (x = 1, y = 1:3, z = 1:3)
base <- ggplot(df, aes(x, y)) + geom_raster(aes(fill = z))
base
base + scale_fill_continuous(guide = guide_legend())
base + guides(fill = guide_legend())
```



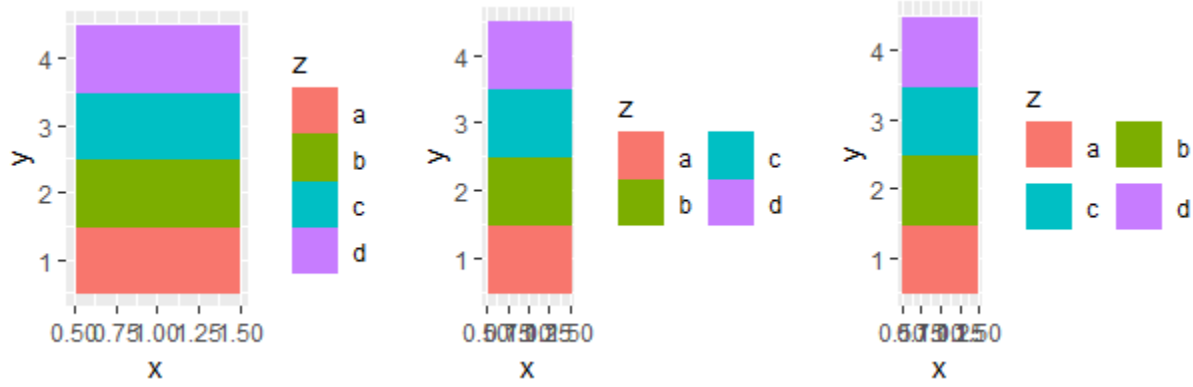
Both functions have numerous examples in their documentation help pages illustrate all of their arguments. Most of the arguments to the guide function control the fine level details of the text colour, size, font, etc.

guide_legend()

The legend guide displays individual keys in a table. The most useful options are:

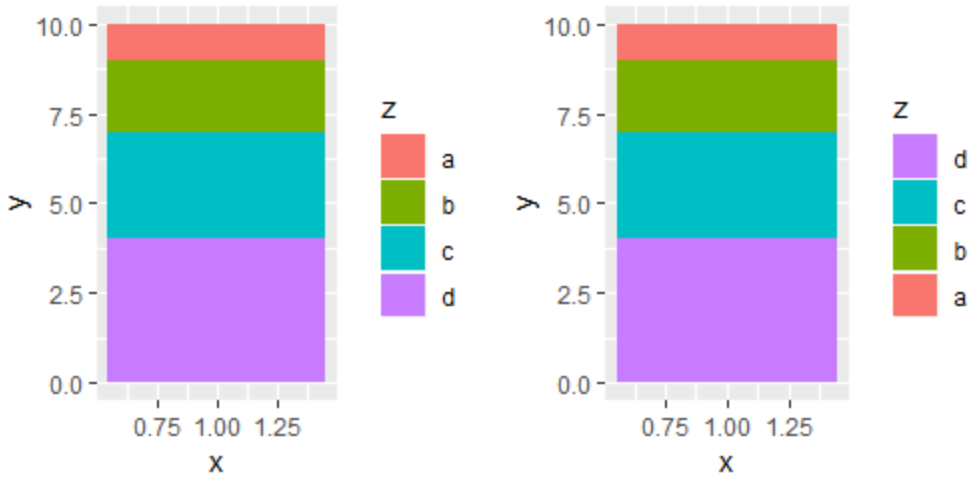
- `nrow` or `ncol` which specify the dimensions of the table. byrow controls how the table is filled: `FALSE` fills it by column (the default), TRUE fills it by row.

```
df <- data.frame(x = 1, y = 1:4, z = letters[1:4])
# Base plot
p <- ggplot(df, aes(x, y)) + geom_raster(aes(fill = z))
p
p + guides(fill = guide_legend (ncol = 2))
p + guides(fill = guide_legend(ncol = 2, byrow = TRUE))
```



- `reverse` reverses the order of the keys. This is particularly useful when you have stacked bars because the default stacking and legend orders are different:

```
p <- ggplot(df, aes(1, y)) + geom_bar(stat = "identity",
aes(fill = z))
p
p + guides(fill = guide_legend(reverse = TRUE))
```



- `override.aes`: override some of the aesthetic settings derived from eachlaver. This is useful if you want to make the elements in the legend more visually prominent.
- `keywidth` and `keyheight` (along with `default.unit`) allow you to specify the size of the keys. These are grid units, e.g. `unit(1, "cm")`.

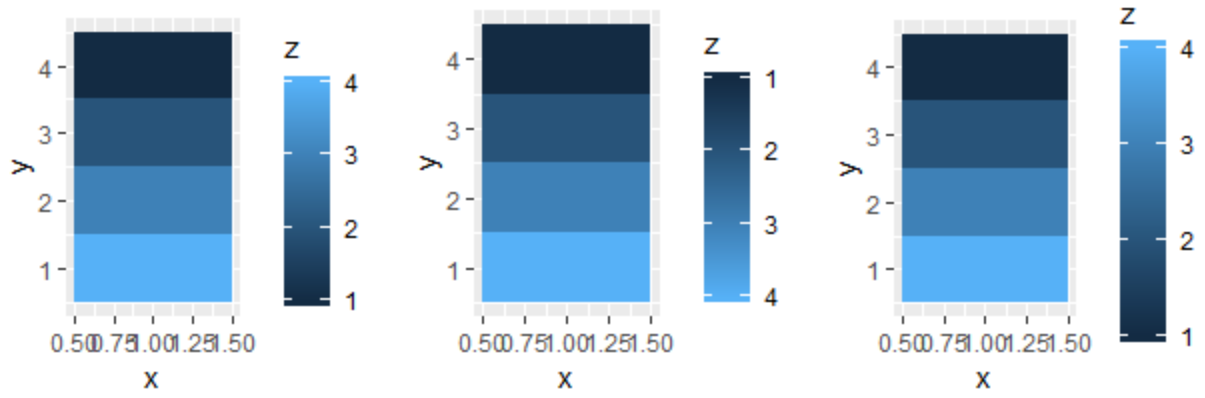
guide_colourbar

The colour bar guide is designed for continuous ranges of colors -- as its name implies, it outputs a rectangle over which the color gradient varies. The most important arguments are:

- `barwidth` and `barheight` (along with default.unit) allow you to specify the size of the bar. These are grid units, e.g. unit(1, "cm").
- nbin controls the number of slices. You may want to increase this from the default value of 20 if you draw a very long bar.
- reverse flips the colour bar to put the lowest values at the top.

These options are illustrated below:

```
df <- data.frame(x = 1, y = 1:4, z = 4:1)
p <- ggplot (df, aes(x, y))  + geom_tile(aes(fill =  z))

p
p + guides(fill = guide_colorbar(reverse = TRUE))
p  +  guides(fill  =  guide_colorbar(barheight  =  unit(4,
"cm")))
```



Limits

The limits, or domain, of a scale are usually derived from the range of the data. There are two reasons you might want to specify limits rather than relying on the data:

1. You want to make limits smaller than the range of the data to focus on an interesting area of the plot.

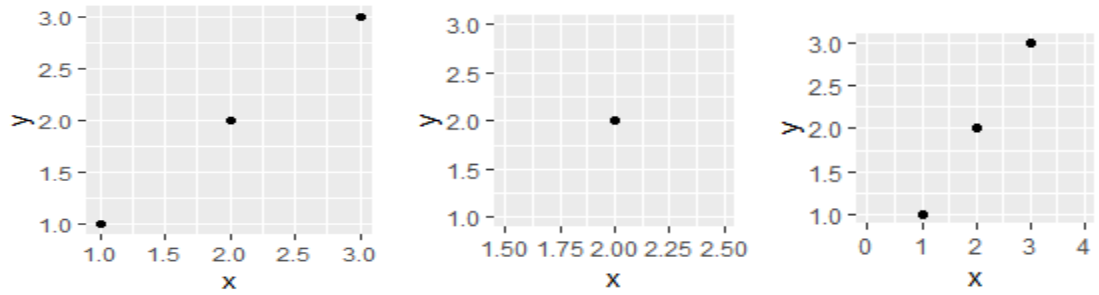
2. You want to make the limits larger than the range of the data because you want multiple plots to match up.

It's most natural to think about the limits of position scales: they map directly to the ranges of the axes. But limits also apply to scales that have legends, like colour, size, and shape. This is particularly important to realise if you want your colours to match up across multiple plots in your paper.

You can modify the limits using the `limits` parameter of the scale:

- For continuous scales, this should be a numeric vector of length two. If you only want to set the upper or lower limit, you can set the other value to `NA`.
- For discrete scales, this is a character vector which enumerates all possible values.

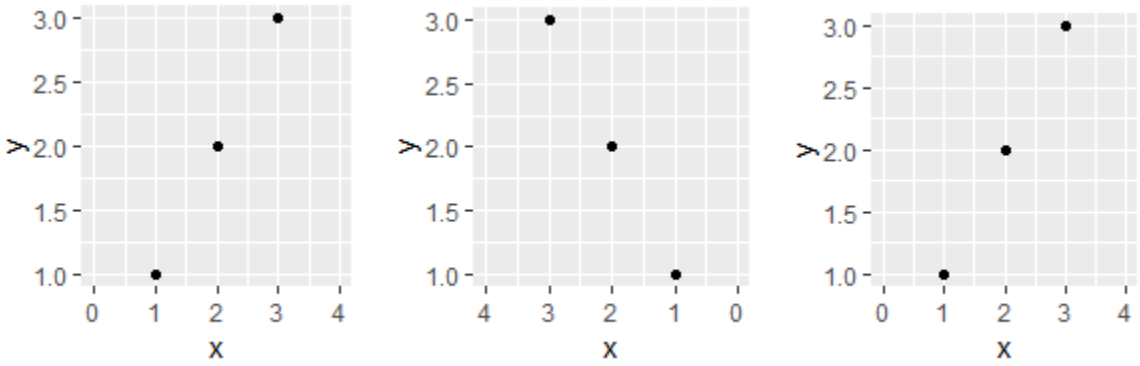
```
df <- data.frame (x = 1:3, y = 1:3)
base <- ggplot(df, aes(x, y)) + geom_point()

base
base + scale_x_continuous (limits = c(1.5, 2.5))
#> Warning Removed 2 rows containing missing values (geom_point).
base + scale_x_continuous(limits = c(0, 4))
```



Because modifying the limits is such a common task, ggplot2 provides some helper to make this even easier: `xlim()`, `ylim()` and `lims()`. These functions inspect their input and then create the appropriate scale, as follows:

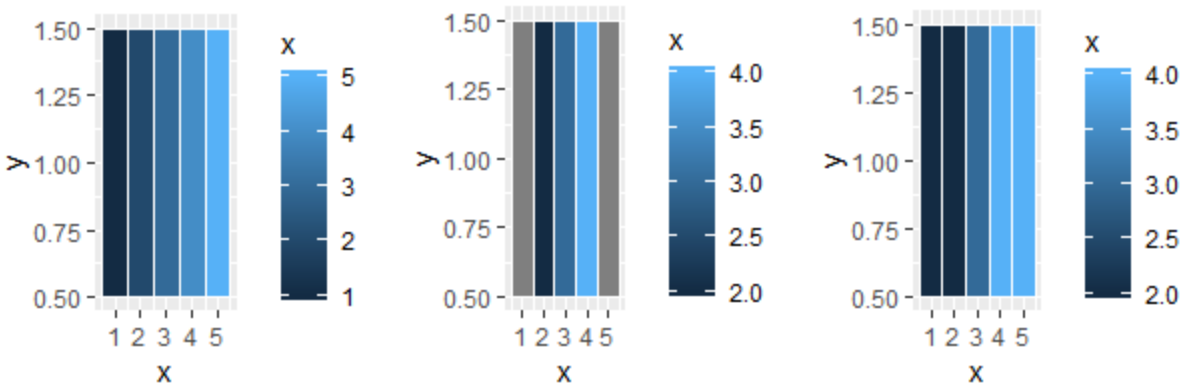
- `xlim(10, 20)`: a continuous scale from 10 to 20
- `ylim(20, 10)`: a reversed continuous scale from 20 to 10
- `xlim("a", "b", "c")`: a discrete scale
- `xlim(as.Date(c("2008-05-01 ", "2008-08-01")))`: a date scale from May 1 to August 1 2008.

```
base + xlim(0, 4)
base + xlim(4,0)
base + lims(x = c(0, 4))
```



By default, any data outside the limits is converted to NA. This means that setting the limits is not the same as visually zooming in to a region of the plot. To do that, you need to use the xlim and ylim arguments to `coord_cartesian()`. This performs purely visual zooming and does not affect the underlying data. You can override this with the `oob` (out of bounds) argument to the scale. The default is `scales:: censor()` which replaces any value outside the limits with NA. Another option is `scales:: squish()` which squishes all values into the range:

```
df <- data.frame(x = 1:5)
p <- ggplot(df, aes(x, 1)) + geom_tile(aes(fill = x), colour =
"white")
p
p + scale_fill_gradient(limits = c(2, 4))
p + scale_fill_gradient(limits = c(2, 4), oob = scales:: squish)
```



Scales Toolbox

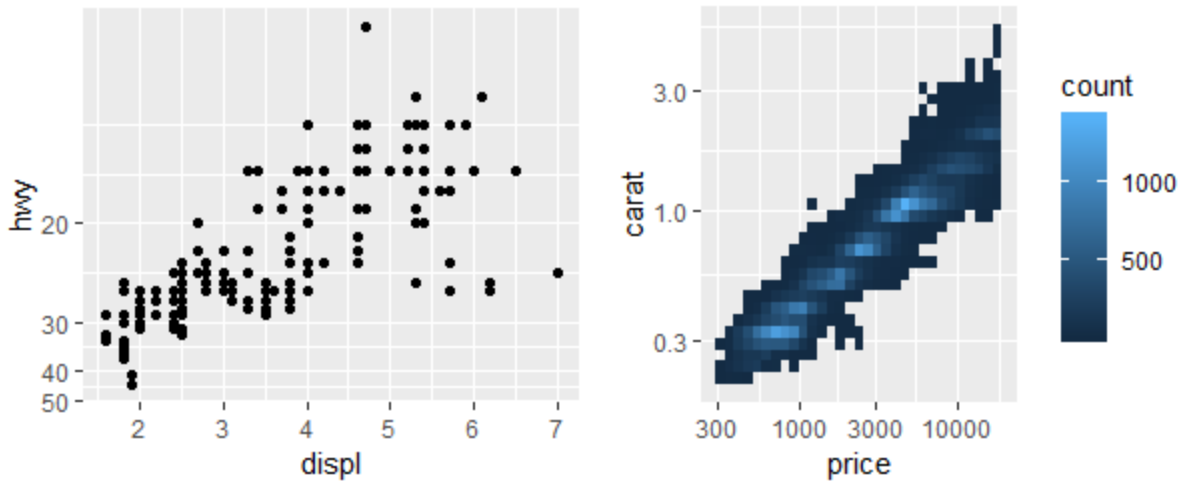
As well as tweaking the options of the default scales, you can also override them completely with new scales. Scales can be divided roughly into four families:

- Continuous position scales used to map integer, numeric, and date/time data to x and y position.
- Colour scales. used to map continuous and discrete data to colours.
- Manual scales, used to map discrete variables to your choice of size, line type, shape or colour.
- The identity scale, paradoxically used to plot variables *without* scaling them. This is useful if your data is already a vector of colour names.

Continuous Position Scales

Every plot has two position scales. x and y. The most common continuous position scales are `scale_x_continuous()` and `scale_y_continuous()`, which linearly map data to the x and y axis. The most interesting variations are produced using transformations. Every continuous scale takes a trans argument, allowing the use of a variety of transformations:

```
#Convert from economy to fuel consumption
ggplot(mpg, aes (displ, hwy)) +
  geom_point() +
  scale_y_continuous(trans = "reciprocal")

#log transform x and y axes
ggplot(diamonds, aes(price, carat)) +
  geom_bin2d() +
  scale_x_continuous (trans = "log10") +
  scale_y_continuous (trans = "log10")
```



Colour

After position, the most commonly used aesthetic is colour. There are quite a few different ways of mapping values to colours in `ggplot2`: four different gradient-based methods for continuous values, and two methods for mapping discrete values.

Continuous

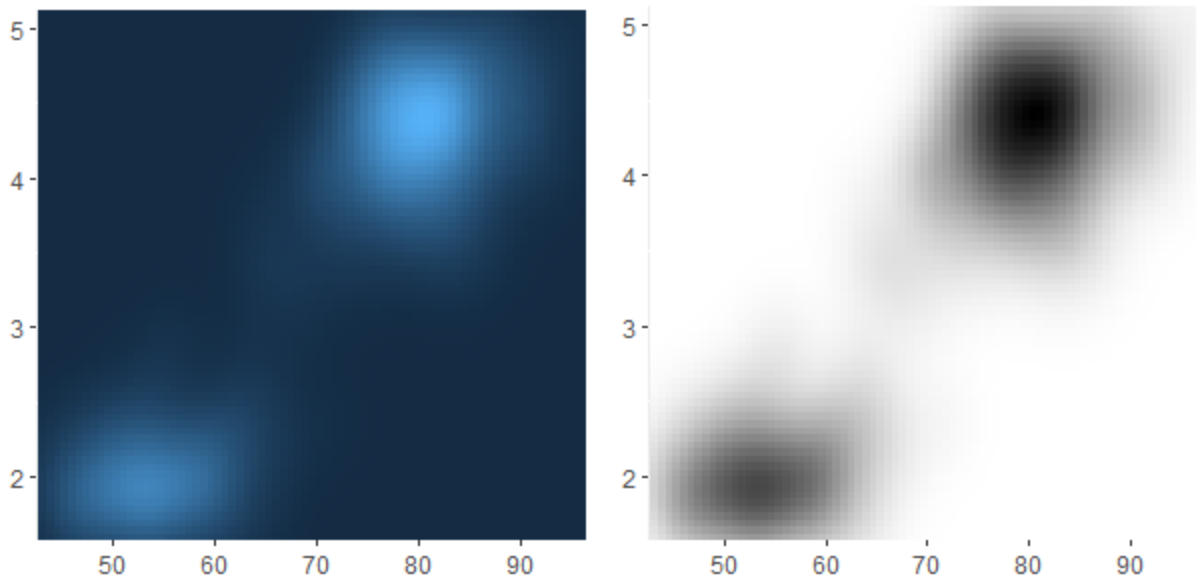
Colour gradients are often used to show the height of a 2d surface. In the following example, we'll use the surface of a 2d density estimate of the faithful dataset (Azzalini and Bowman, 1990), which records the waiting time between eruptions and during each eruption for the Old Faithful geyser in Yellowstone Park. I hide the legends and set expand to 0, to focus on the appearance of the data. Remember: I'm illustrating these scales with filled tiles, but you can also use them with coloured lines and points.

```
erupt <- ggplot(faithfuld,aes(waiting,eruptions,fill= density))
+
  geom_raster () +
  scale_x_continuous (NULL, expand = c(0, 0)) +
  scale_y_continuous (NULL, expand = c(0, 0)) +
  theme(legend.position = "none")
```

There are four continuous colour scales:

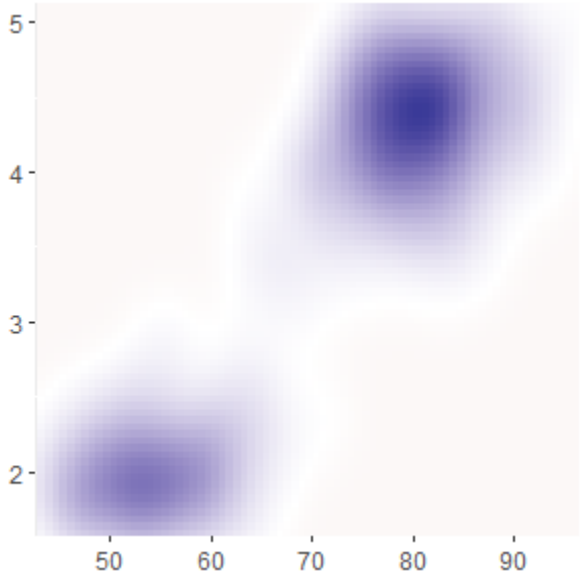
1. `scale_colour_gradient()` and `scale_fill_gradient()`: a two-colour gradient, low-high (light blue-dark blue). This is the default scale for continuous colour, and is the same as `scale_colour_continuous()`. Arguments low and high control the colours at either end of the gradient.

```
erupt
erupt + scale_fill_gradient(low = "white", high = "black")
```



2. `scale_colour_gradient2()` and `scale_fill_gradient2()`: a three-colour gradient, low-med-high (red-white-blue). As well as low and high colours, these scales also have a mid colour for the colour of the midpoint. The midpoint defaults to 0, but can be set to any value with the midpoint argument. It's artificial to use this colour scale with this dataset, but we can force it by using the median of the density as the midpoint. Note that the blues are much more intense than the reds (which you only see as a very pale pink)

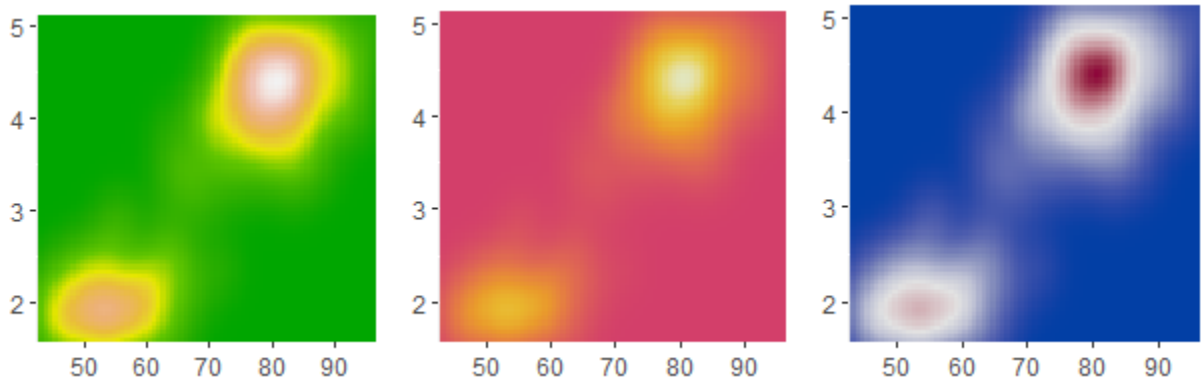
```
mid <- median(faithful$density)
erupt + scale_fill_gradient2(midpoint = mid)
```



3. `scale_colour_gradient()` and `scale_fill_gradient()`: a custom n-colour gradient. This is useful if you have colours that are meaningful for your data (e.g.. black body colours or standard terrain colours), or you'd like to use a palette produced by another package.

The following code includes palettes generated from routines in the **colorspace** package. (Zeileis et al.,2008) describes the philosophy behind these palettes and provides a good introduction to some of the complexities of creating good colour scales.

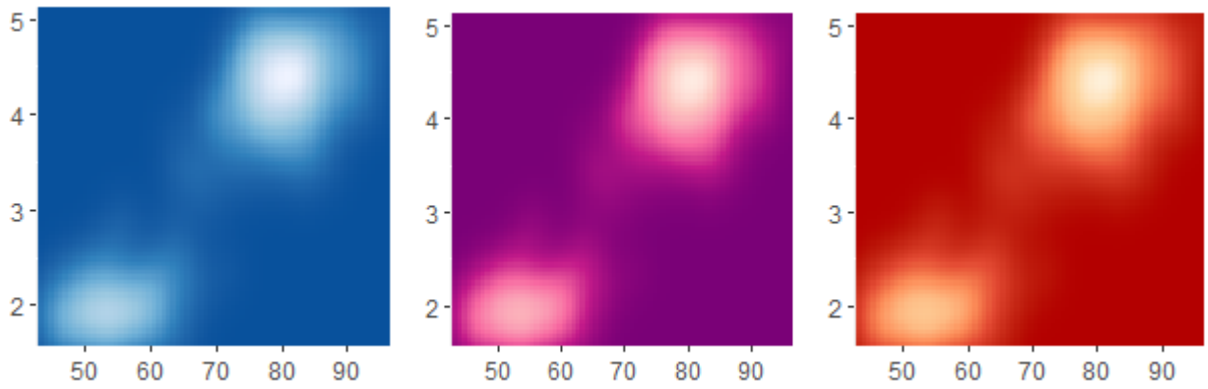
```
erupt + scale_fill_gradientn(colours = terrain.colors(7))
erupt + scale_fill_gradientn(colours = colorspace::heat_hcl(7))
erupt              +scale_fill_gradientn(colours              =
colorspace::diverge_hcl(7))
```



By default, colours will be evenly spaced along the range of the data. To make them unevenly spaced, use the values argument, which should be a vector of values between 0 and 1.

4. `scale_color_distiller()` and `scale_fill_gradient()` apply the ColorBrewer colour scales to continuous data. You use it the same way as `scale_fill_brewer()`, described below:

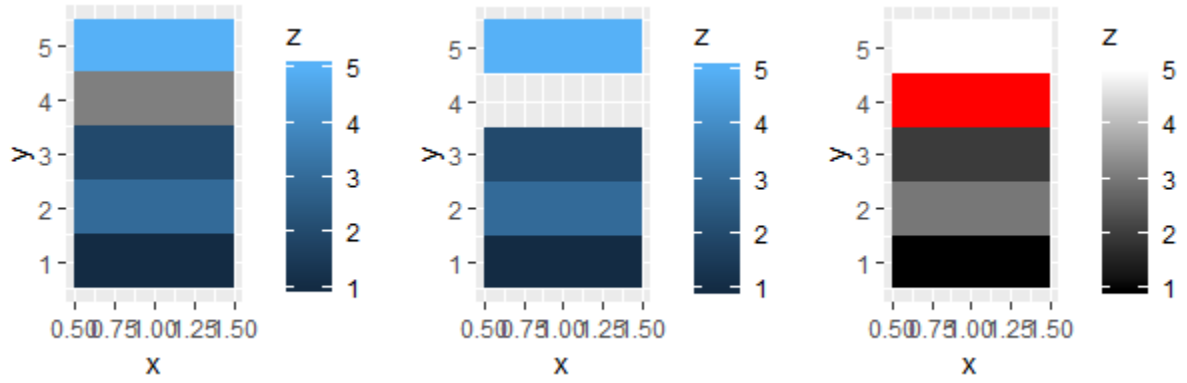
```
erupt + scale_fill_distiller()
erupt + scale_fill_distiller(palette = "RdPu" )
erupt + scale_fill_distiller(palette = "OrRd")
```



All continuous colour scales have an na.value parameter that controls what colour is used for missing values (including values outside the range of the scale limits). By default it is set to grey, which will stand out when you use a colourful scale. If you use a black and white scale, you might want to set it to something else to make it more obvious.

```
df <- data.frame(x= 1, y = 1:5, z = c(1, 3, 2, NA, 5))
```

```
p <- ggplot(df, aes(x, y)) + geom_tile(aes(fill = z), size = 5)
p
#Make missing colours invisible
p + scale_fill_gradient(na.value = NA)
#Customise on a black and white scale
p + scale_fill_gradient(low = "black", high = "white", na.value
= "red")
```



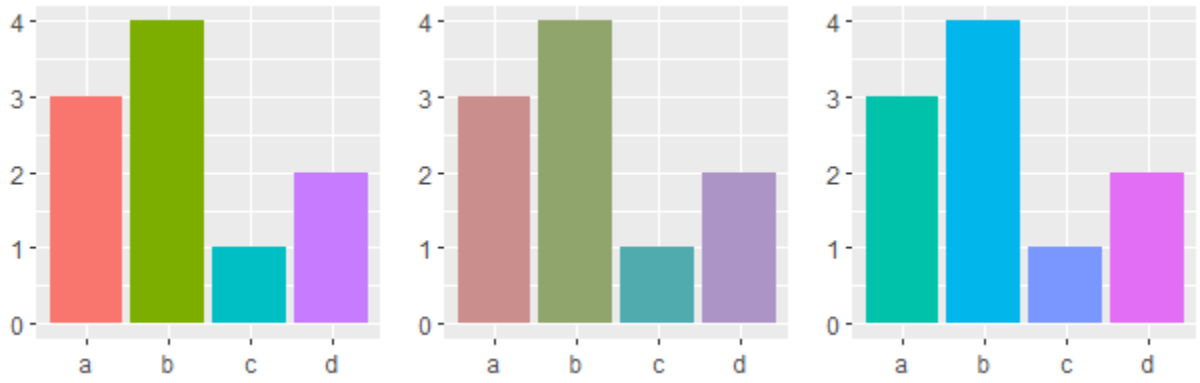
Discrete

There are four colour scales for discrete data. We illustrate them with a bar chart that encodes both position and fill to the same variable:

```
df <- data.frame(x = c("a", "b", "c", "d" ), y = c(3, 4, 1, 2))
bars <- ggplot(df, aes(x, y, fill = x)) +
  geom_bar(stat = "identity") +
  labs(x = NULL, y = NULL) +
  theme (legend.position = "none")
```

- The default colour scheme, scale.color hue), picks evenly spaced hues around the HCL colour wheel. This works well for up to about eight colours, but after that it becomes hard to tell the different colours apart. You can control the default chroma and luminance, and the range of hues,with the h, c and 1 arguments:

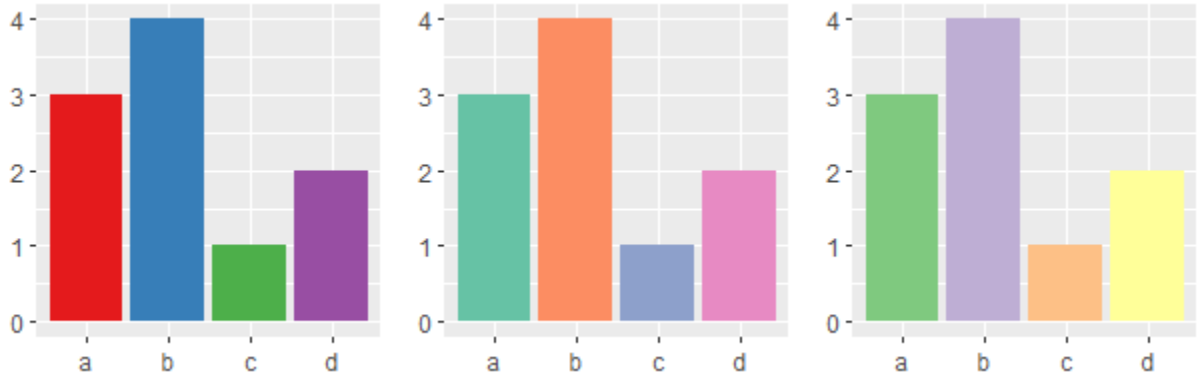
```
bars
bars + scale_fill_hue(c = 40)
bars + scale_fill_hue(h = c(180, 300))
```



One disadvantage of the default colour scheme is that because the colours all have the same luminance and chroma, when you print them in black and white, they all appear as an identical shade of grey.

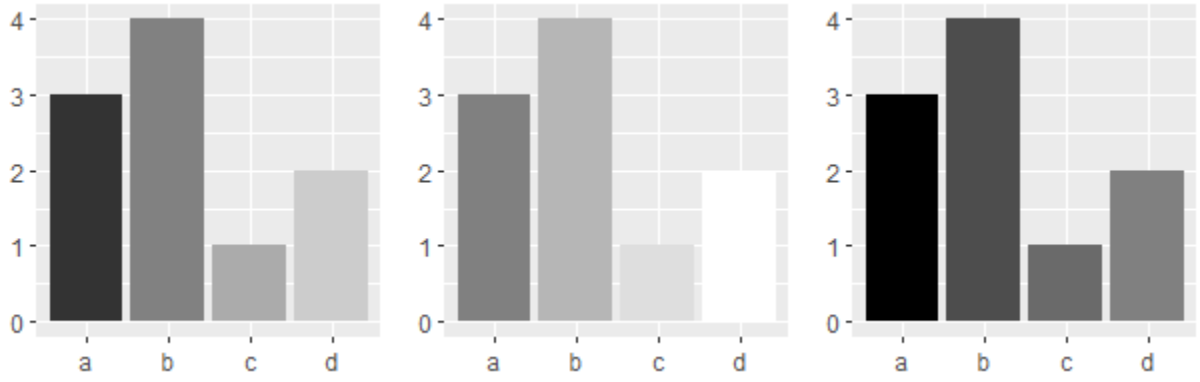
- `scale_colour_brewer()` uses handpicked "ColorBrewer" colours, http://colorbrewer2.org/. These colours have been designed to work well in a wide variety of situations, although the focus is on maps and so the colours tend to work better when displayed in large areas. For categorical data, the palettes most of interest are 'Set1' and 'Dark2' for points and 'Set2', 'Pastel1', Pastel2' and Accent' for areas. Use `RColorBrewer::display.brewer.all()` to list all palettes.

```
bars + scale_fill_brewer(palette = "Set1")
bars + scale_fill_brewer(palette = "Set2")
bars + scale_fill_brewer(palette = "Accent")
```



- `scale_colour_grey()` maps discrete data to grays, from light to dark.

```
bars + scale_fill_grey()
bars + scale_fill_grey(start = 0.5, end = 1)
bars + scale_fill_grey(start = 0, end = 0.5)
```



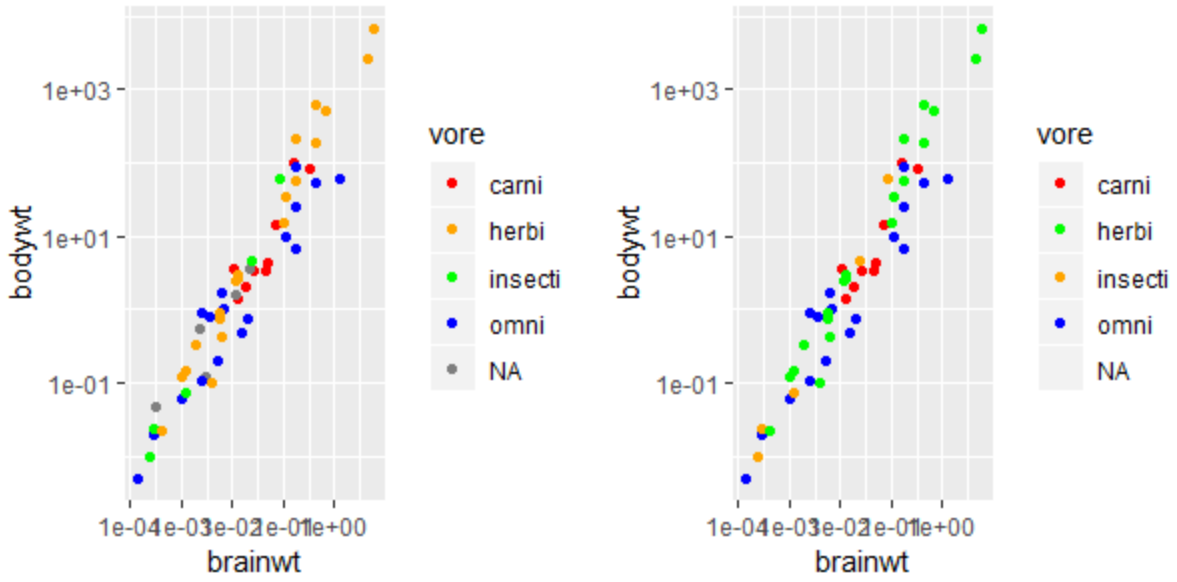
The Manual Discrete Scale

The discrete scales, `scale_linetype()`, `scale_shape()`, and `scale_size_discrete()` basically have no options. These scales are just a list of valid values that are mapped to the unique discrete values.

If you want to customize these scales, you need to create your own new scale with the manual scale: `scale_shape_manual()`, `scale_linetype_manual()`, `scale_colour_manual()`. The manual scale has one important argument, values, where you specify the values that the scale should produce. If this vector is named, it will match the values of the output to the values of the input: otherwise, it will match in order of the levels of the discrete variable. You will need some knowledge of the valid aesthetic values, which are described in `vignette("ggplot2-specs")`.

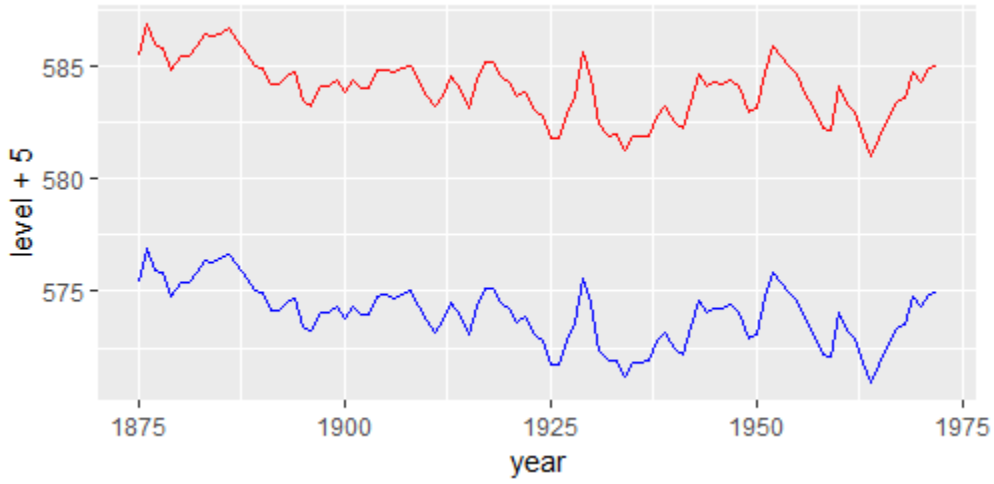
The following code demonstrates the use of `scale_colour_manual()`:

```
plot <- ggplot(msleep, aes(brainwt, bodywt)) +
scale_x_log10() +
  scale_y_log10()
  plot +
  geom_point(aes(colour = vore)) +
  scale_colour_manual(
    values = c("red", "orange", "green", "blue"),
    na.value = "grey50"
   )
  # Warning: Removed 27 rows containing missing values (geom
point).

  colours <- c(
    carni = "red",
    insecti = "orange",
    herbi = "green",
    omni = "blue"
  )

  plot +
    geom_point(aes(colour = vore)) +
    scale_colour_manual(values = colours)
  #> Warning: Removed 27 Containing missing values(geom_point)
```



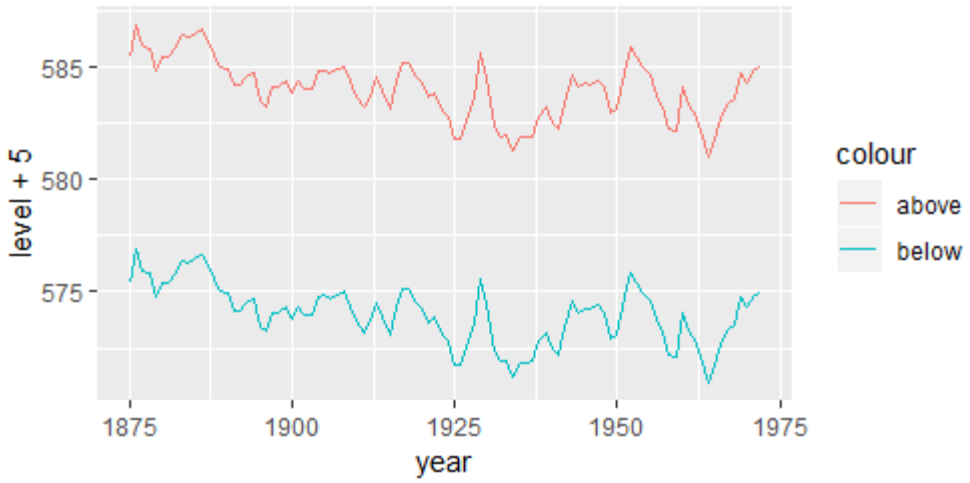
The following example shows a creative use of `scale_colour_manual()` to display multiple variables on the same plot and show a useful legend. In most other plotting systems, you'd colour the lines and then add a legend:

```
huron <- data.frame(year = 1875:1972, level =
as.numeric(LakeHuron))
ggplot(huron, aes(year)) +
    geom_line(aes(y = level + 5), colour = "red") +
    geom_line(aes(y = level - 5), colour = "blue")
```



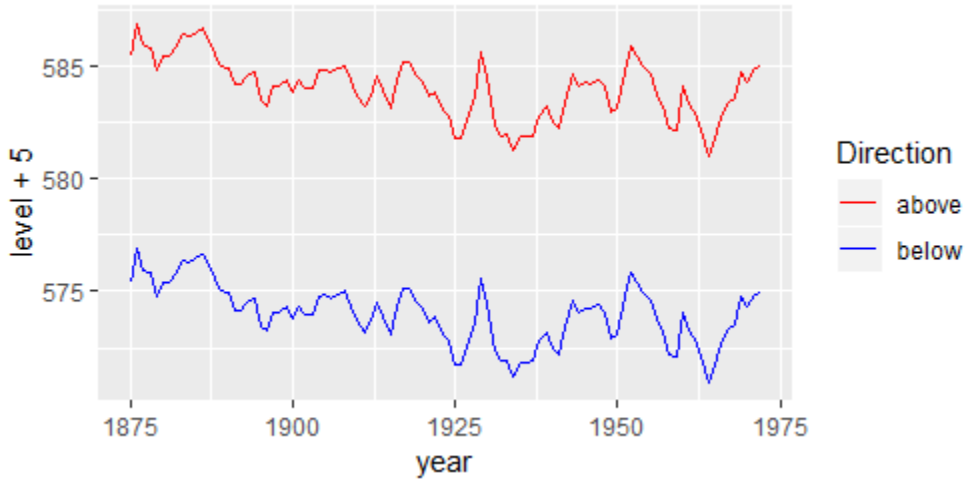
That doesn't work in ggplot because there's no way to add a legend manually. Instead, give the lines informative labels:

```
ggplot(huron, aes(year)) +
geom_line(aes(y = level 5, colour = "above")) +
geom_line(aes(y = level 5, colour = "below"))
```



And then tell the scale how to map labels to colours:

```
ggplot(huron, aes(year)) +
  geom_line (aes(y = level + 5, colour = "above")) +
  geom_line (aes(y = level - 5, colour = "below")) +
  scale_colour_manual ("Direction",
    values = c("above" = "red", "below" = "blue")
)
```



The Identity Scale

The identity scale is used when your data is already scaled, when the data and aesthetic spaces are the same. The code below shows an example where the identity scale is useful. `luv.colours` contains the locations of all R's built-in colours in the LUV colour space (the space that HCL is based on). A legend is unnecessary because the point colour represents itself: the data and aesthetic spaces are the same.

```
head(luv_colours)

          L             u         v           col

1 9341.570 -3.370649e-12    0.0000         white
```

```
2 9100.962 -4.749170e+02 -635.3502     aliceblue
3 8809.518  1.008865e+03 1668.0042  antiquewhite
4 8935.225  1.065698e+03 1674.5948 antiquewhite1
5 8452.499  1.014911e+03 1609.5923 antiquewhite2
6 7498.378  9.029892e+02 1401.7026 antiquewhite3
```

```
ggplot(luv_colours, aes(u, v)) +
    geom_point(aes(colour = col), size = 3) +
    scale_color_identity() +
    coord_equal()
```