Pedro J. Aphalo

Learn R

As a Language

Contents

List of Figures

List of Tables

1

R Extensions: Grammar of Graphics

> The commonality between science and art is in trying to see profoundly—to develop strategies of seeing and showing.
>
> Edward Tufte's answer to Charlotte Thralls
> *An Interview with Edward R. Tufte*, 2004

1.1 Aims of this chapter

Three main data plotting systems are available to R users: base R, package 'lattice' (Sarkar 2008) and package 'ggplot2' (Wickham and Sievert 2016), the last one being the most recent and currently most popular system available in R for plotting data. Even two different sets of graphics primitives (i.e., those used to produce the simplest graphical elements such as lines and symbols) are available in R, those in base R and a newer one in the 'grid' package (Murrell 2011).

In this chapter you will learn the concepts of the layered grammar of graphics, on which package 'ggplot2' is based. You will also learn how to build several types of data plots with package 'ggplot2'. As a consequence of the popularity and flexibility of 'ggplot2', many contributed packages extending its functionality have been developed and deposited in public repositories. However, I will focus mainly on package 'ggplot2' only briefly describing a few of these extensions.

1.2 Packages used in this chapter

If the packages used in this chapter are not yet installed in your computer, you can install them as shown below, as long as package 'learnrbook' is already installed.

```r
install.packages(learnrbook::pkgs_ch_ggplot)
```

To run the examples included in this chapter, you need first to load some packages from the library (see section **??** on page **??** for details on the use of packages).

```r
library(learnrbook)
library(wrapr)
library(scales)
library(ggplot2)
library(ggrepel)
library(gginnards)
library(broom)
library(ggpmisc)
library(ggbeeswarm)
library(ggforce)
library(tikzDevice)
library(lubridate)
library(tidyverse)
library(patchwork)
```

1.3   The components of a plot

I start by briefly presenting concepts central to data visualisation, following the *Data visualization handbook* (Koponen and Hildén 2019). Plots are a medium used to convey information, like text. It is worthwhile keeping this in mind. As with text, the design of plots needs to consider what we want to highlight, what is the take home message we want to convey. The style of the plot should match the expectations and the plot-reading abilities of the expected audience. One needs to be careful to avoid ambiguities and most importantly of all not to miss-inform. Data visualisations like text need to be planned, revised, commented upon, revised again until the best way of expressing our message is found. As we will see through this chapter, the flexibility of the grammar of graphics supports very well this approach to designing and producing high quality data visualizations for different audiences.

Of course, when exploring data we do not need fancy details of graphical design, but we still need the flexibility that allows looking at the same data from many differing angles, highlighting different aspects of them. In the same way as boiler-plate text and text templates have specific but limited uses, all-in-one functions for producing plots do not support well the design of original data visualizations. They tend to get the job done, but lack the flexibility needed to do the best job of communicating with readers. Being this a book about languages, the focus of this chapter is in the layered grammar of graphics.

The plots we will describe in this chapter are classified as *statistical graphics* within the larger field of data visualisation which is broader. Plots such as scatter plots include points (geometric objects) that by their position, shape, colour or some other property directly convey information. If we consider these points their location in the plot "canvas" or "plotting area" is given by the values of their coordinates and any alteration to these coordinates is wrong. Such alterations would break the correspondence between coordinates and observed values thus convey-

ing wrong/false information to the audience. A *data label* is connected to an observation but its position can be displaced as long as its link to the corresponding observation can be inferred, e.g., by the direction of an arrow or even simple proximity. *Annotations*, are additions to a plot that have no connection to individual observations, but rather with all observations taken together, e.g., a text like n = 200 indicating the number of observations included in a corner of a plot. These three elements convey information about observations. The axis labels, legends and keys included in the visualisation make it possible for the reader to retrieve the original values represented in the plot by graphical elements. Other elements in a visualisation even when not carrying additional information still affect the easy with which a plot can be read. These include the size of text and symbols, thickness of lines, font face, the choice of colour palette, etc. In general plots designed to be included in books and journals are unsuitable for oral presentations, and vice versa, mainly because of the different length of time available for the audience to read them. It is important to be aware of the roles played by different plot components when designing a data visualisation and when implementing it using the grammar of graphics.

1.4 The grammar of graphics

What separates 'ggplot2' from base R and trellis/lattice plotting functions is the use of a layered grammar of graphics (the reason behind 'gg' in the name of package 'ggplot2'). What is meant by grammar in this case is that plots are assembled piece by piece using different "nouns" and "verbs" (Cleveland 1985). Instead of using a single function with many arguments, plots are assembled by combining different elements with operators + and %+%. Furthermore, the construction is mostly semantics-based and to a large extent, how plots look when printed, displayed, or exported to a bitmap or vector-graphics file is controlled by themes.

We can think of plotting as translating or mapping the observations or data into a graphical language. We use properties of graphical (or geometrical) objects to represent different aspects of our data. An observation can consist of multiple recorded values. Say an observation of air temperature may be defined by a position in 3-dimensional space and a point in time, in addition to the temperature itself. An observation for the size and shape of a plant can consist of height, stem diameter, number of leaves, size of individual leaves, length of roots, fresh mass, dry mass, etc. If we are interested in the relationship between height and stem diameter, we may want to use cartesian coordinates, *mapping* stem diameter to the x dimension of the plot and the height to the y dimension. The observations could be represented on the plot by points.

The grammar of graphics allows us to design plots by combining various elements in ways that are nearly orthogonal. In other words, the majority of the possible combinations of "words" yield valid plots as long as we assemble them respecting the rules of the grammar. This flexibility makes 'ggplot2' extremely

powerful as we can build plots and even types of plots which were not even considered while designing the 'ggplot2' package.

> ⚠ When a ggplot is built, the whole plot and its components are created as R objects that can be saved in the workspace or written to a file as objects. These objects encode a recipe for constructing the plot, not its final graphical representation. The graphical representation is generated when the object is printed, explicitly or automatically. Thus, the same "gg" plot object can be rendered into different bitmap and vector graphic formats for display or printing.

The transformation of a set of data or observations into a rendered graphic with package 'ggplot2' can be represented as a flow of information, but also as a sequence of actions. However, what avoids that the flexibility becomes a burden is that if we do not explicitly mention all steps in our code, in most cases adequate defaults for them will be used instead. The recipe to build a plot needs to specify a) the data to use, b) which variable to map to which graphical property (or aesthetic), c) which layers to add and which geometric representation to use, d) the scales that establish the link between data values and aesthetic values, e) a coordinate system (affecting only aesthetics x, y and possibly z), f) a theme to use. The result from constructing a plot object using the grammar of graphics is an R object containing a "recipe for a plot", including the data, which behaves similarly to other R objects.

1.4.1   The words of the grammar

Before building a plot step by step, I introduce next the different components of a ggplot recipe, or the words in the grammar.

Data

The data to be plotted must be available as a `data.frame` or `tibble`, with data stored so that each row represents a single observation event, and the columns are different values observed in that single event. In other words, in long form (so-called "tidy data") as described in chapter **??**. The variables to be plotted can be `numeric`, `factor`, `character`, and time or date stored as `POSIXct`. (Some extensions to 'ggplot2' add support for other types of data such as time series).

Mapping

When we design a plot, we need to map data variables to aesthetics (or graphic properties). Most plots will have an x dimension, which is considered an *aesthetic*, and a variable containing numbers (or categories) mapped to it. The position on a 2D plot of, say, a point, will be determined by x and y aesthetics, while in a 3D plot, three aesthetics need to be mapped x, y and z. Many aesthetics are not related to coordinates, they are properties, like color, size, shape, line type, or even rotation angle, which add an additional dimension on which to represent the values of variables and/or constants.

Statistics

Statistics are "words" that represent calculation of summaries or some other operation on the values in the data. When *statistics* are used for a computation, the returned value is passed to a *geometry*, and consequently adding a *statistics* also adds a layer to the plot. For example, `stat_smooth()` fits a smoother, and `stat_summary()` applies a summary function such as `mean(()`. Most statistics are applied by group when data have been grouped by mapping additional aesthetics such as color to a factor.

Geometries

Geometries are "words" that describe the graphics representation of the data: for example, `geom_point()`, plots a point or symbol for each observation or summary value, while `geom_line()`, draws line segments between observations. Some geometries rely by default on statistics, but most "geoms" default to the identity statistics. Each time a *geometry* is used to add a graphical representation of data to a plot, we say that a new *layer* has been added. The grammar of graphics allows plots to contain multiple layers. The name *layer* reflects the fact that each new layer added is plotted on top of the layers already present in the plot, or rather when a plot is printed the layers will be generated in the order they were added to the plot object. For example, one layer in a plot can display the observations, another layer a regression line fitted to them, and a third one may contain annotations such an equation or a text label.

Positions

Positions are "words" that determine the displacement or not of graphical plot elements relative to their original x and y coordinates. They are one of the arguments accepted by *geometries*. Position `position_identity()` introduces no displacement, and for example, `position_stack()` makes it possible to create stacked bar plots and stacked area plots. Positions will be discussed together with geometries as they are always subordinate to them.

Scales

Scales give the "translation" or mapping between data values and the aesthetic values to be actually plotted. Mapping a variable to the "color" aesthetic (also recognized when spelled as "colour") only tells that different values stored in the mapped variable will be represented by different colors. A scale, such as `scale_color_continuous()`, will determine which color in the plot corresponds to which value in the variable. Scales can also define transformations on the data, which are used when mapping data values to aesthetic values. All continuous scales support transformations—e.g., in the case of x and y aesthetics, positions on the plotting region or graphic viewport will be affected by the transformation, while the original values are used for tick labels along the axes or in keys for shapes, colors, etc. Scales are used for all aesthetics, including continuous variables, such as numbers, and categorical ones such as factors. The grammar of graphics allows only one scale per *aesthetic* and plot. This restriction is imposed by design to avoid

ambiguity (e.g., it ensures that the red color will have the same "meaning" in all plot layers where the `color` *aesthetic* is mapped to data). Scales have limits that are set automatically unless supplied explicitly.

Coordinate systems

The most frequently used coordinate system when plotting data, the cartesian system, is the default for most *geometries.* In the cartesian system, x and y are represented as distances on two orthogonal (at $90°$) axes. Additional coordinate systems are available in 'ggplot2' and through extensions. For example, in the polar system of coordinates, the x values are mapped to angles around a central point and y values to the radius. Setting limits to a coordinate system changes the region of the plotting space visible in the plot, but does not discard observations. In other words, when using *statistics*, observations located outside the coordinate limits, i.e., not visible in the rendered plot, will still be included in computations when excluded by coordinate limits but will be ignored when excluded by scale limits.

Themes

How the plots look when displayed or printed can be altered by means of themes. A plot can be saved without adding a theme and then printed or displayed using different themes. Also, individual theme elements can be changed, and whole new themes defined. This adds a lot of flexibility and helps in the separation of the data representation aspects from those related to the graphical design.

Operators

To assemble the elements described above into a ggplot object we normally use operator + and exceptionally `%+%`. This choice makes sense, as we build ggplot objects by sequentially adding members to them.

⚠   The functions corresponding to the different elements of the grammar of graphics have distinctive names with the first few letters hinting at their roles: aesthetics mappings (`aes`), geometric elements (`geom_…`), statistics (`stat_…`), scales (`scale_…`), coordinate systems (`coord_…`), and themes (`theme_…`).

1.4.2   The workings of the grammar

A `"gg"` plot object is an R object of mode `"list"` containing the recipe and data to construct a plot. It is self contained in the sense that the only requirement for rendering it into a graphical representation is the availability of package 'ggplot2'. A `"gg"` object contains the data in one or more data frames and instructions encoded as functions and parameters, but not yet a rendering of the plot into graphical objects. Both data transformations and rendering of the plot into drawing instructions (encoded as graphical objects or *grobs*) take place at the time of printing or exporting the plot, e.g., when saving a bitmap to a file.

To understand ggplots we should first think in terms of the graphical organization of the plot: there is a background layer onto which layers composed by different graphical objects are laid. Each layer contains related graphical objects

| layer data | a | **start** | b | statistic | c | **after stat** | d | geometry + scale | e | **after scale** | f | layer grobs |

FIGURE 1.1
Abstract diagram of data transformations in a ggplot layer showing the stages at which mappings between variables and graphic aesthetics take place.

originating from the same data. The last layer added is the topmost and the first one added the lowermost. Graphical objects in upper layers occlude those in the layers below them if their locations overlap. Although frequently layers in a ggplot share the same data and the same mappings to aesthetics, this is not a requirement. It is possible to build ggplots with independent layers, although always with shared scales and plotting area.

A second perspective on ggplots is that of the process of converting the data into a graphical representation that can be printed on paper or viewed on a computer screen. The transformations applied to the data to achieve this can be thought as a data flow process divided into stages. The diagram in Figure 1.1 represents a single self-contained layer in a plot. The data supplied by the user is transformed in stages into instructions to draw a graphical representation. In 'ggplot2' and its documentation graphical features are called *aesthetics*, with the correspondence between values in the data and values of the aesthetic controlled by *scales*. The values in the data are summarized by *statistics*. However, when no summaries are needed, layers make use of `stat_indentity()`, which copies its input to its output unchanged. *Geometries* provide the "recipe" used to generate graphical objects from the mapped data.

Function `aes()` is used to define mappings to aesthetics. The default for `aes()` is for the mapping to take place at the **start** (leftmost circle in the diagram above), mapping names in the user data to aesthetics such as x, y, colour, shape, etc. The statistic can alter the mapped data, but in most cases not which aesthetics they are mapped to. Statistics can add default mappings for additional aesthetics. In addition the default mappings of the data returned by the statistic can be modified by user code at this later stage, **after stat**. Default mappings can be modified again at the **after scale** stage.

Statistics always return a mapping to the same aesthetics that they require as input. However, the values mapped to these aesthetics at the **after stat** stage are in most cases different from those at **start**. Many statistics return additional variables, which are not mapped by default to any aesthetic. These variables facilitate variations on how results from a given type of data summary are added to plots, including the use of a geometry different from the default set by the statistic. In this case the user has to override default mappings at the **after stat** stage. The additional variables returned by statistcs are listed in their documentation. (See section 1.4.5 on page 18 for details.)

⚠   As mentioned above, all ggplot layers include a statistic and a geometry. From the perspective of the construction of a plot using the grammar, both `stats` and `geoms` are layer constructor functions. While `stats` take a `geom` as one of their arguments, `geoms` take a `stat` as one of their arguments. Thus, in either case a `stat` and a `geom` are added, their positions in the data flow independent of how they were added to a plot, i.e., the diagram in Figure 1.1 applies independently of how we add the layer to the plot. The default statistic of many geometries is `stat_identity()` making their behavior when added to a plot as if the layer they create contained no statistics.

There are some statistics in 'ggplot2' that have companion geometries that can be used (almost) interchangeably. This tends to lead into confusion, and in this book, only geometries that have as default `stat_identity()` are described as geometries in section 1.5. In the case of those that by default use other statistics, like `geom_smooth()` only the companion statistic, `stat_smooth()` for this example, are described in section 1.6.

A ggplot can have a single layer or many layers, but when ggplots have more than one layer, the data flow, computations and generation of graphical objects takes place independently for each layer. As mentioned above, most ggplots do not have fully independent layers, but the layers share the same data and aesthetic mappings at the **start**. Ahead of this point computations in layers are always independent of those in other layers, except that for a given aesthetic only one scale is allowed per plot.

1.4.3   Plot construction

As the use of the grammar is easier to demonstrate by example than to explain with words, I will show how to build plots of increasing complexity, starting from the simplest possible. All elements of a plot have defaults, although in some cases these defaults result in empty plots. Defaults make it possible to create a plot very succinctly. When building a plot step by step, we can consider the different aspects described in the previous section: the structure of the object, the graphic output, and the transformations applied to the data in the route between the recipe stored in an object and graphic output. In this section I emphasize the syntax of the grammar and how it translates into a plot.

Function `ggplot()` by default constructs an empty plot. This similar to how `character()`, `numeric()`, etc. construct empty vectors. This empty skeleton of a plot when printed is displayed as an grey rectangle.

```
ggplot()
```
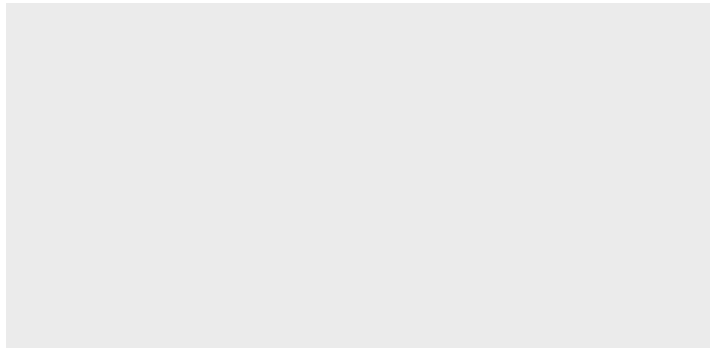
A data frame passed as argument to `data` without adding a mapping results in the same empty grey rectangle (not shown). Data frame `mtcars` is a data set included in R (to read a description, type `help("mtcars")` at the R command prompt).

```
ggplot(data = mtcars)
```

Once the data are available, we need to select a graphical or geometric representation for the quantities we intend to plot. The kind of graphical representation is determined by the geometry, such as `geom_point()` and `geom_line()`, drawing separate points for the observations or connecting them with lines, respectively. A mapping indicates which property of the geometric elements will be used to represent the values stored in a given variable in the user's data. For most geometries we need to provide mappings for both x and y aesthetics, to establish the position of the geometrical shapes like points or lines in the plotting area. Additional aesthetics like colour (applicable to both points and lines) or shape and linetype, applicable to points and lines, respectively, use default scales and palettes. These defaults can be overridden by addition one `scale` or more to the plot (see section 1.9).

Mapping at the **start** stage, `disp` to x and `mpg` to y aesthetics, makes the ranges of the values available. They are used to find default limits for the x and y scales as reflected in the plot axes. The plotting area x and y now match the ranges of the mapped variables, expanded by a small margin. The axis labels also reflect the names of the mapped variables, however, there are no graphical element yet displayed for the individual observations.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg))
```
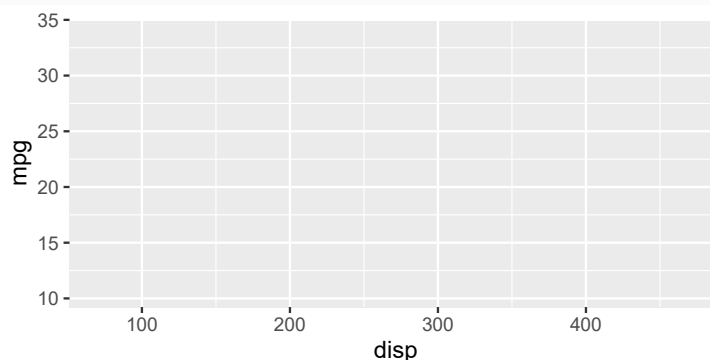
To make observations visible as a plot layer we need to use a suitable *geometry* or `geom` to the plot recipe. Adding `geom_point()` makes the observations visible as points.

```r
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point()
```
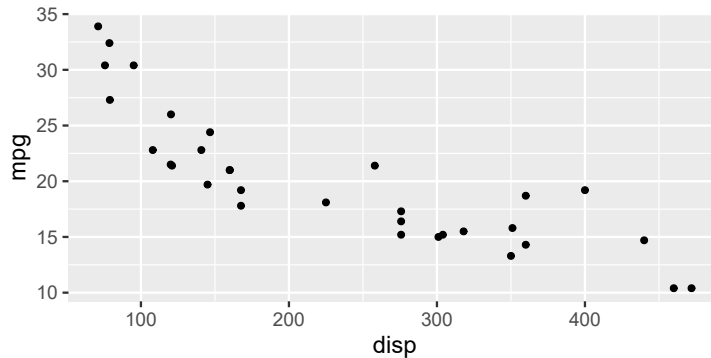

⚠️ In the examples above, the plots were printed automatically, which is the default at the R console. However, as with other R objects, ggplots can be assigned to a variable.

```r
p1 <- ggplot(data = mtcars,
             aes(x = disp, y = mpg)) +
       geom_point()
```

and printed at a later time, and saved to and read from files on disk.

```r
print(p1)
```

Layers and other elements can be also added to a saved ggplot as the saved objects are not the graphical representation of the plots themselves but instead a *recipe* plus data needed to build them.

⌨️🖥️ **1.1** As for any R object `str()` displays the structure of `"gg"` objects. In addition, package 'ggplot2' provides a `summary()` method for `"gg"` plot objects.

As you make progress through the chapter, use these methods to explore the `"gg"` plot objects you construct, paying attention to layers, and global vs. layer-specific data and mappings. You will learn how the plot components are stored as members of `"gg"` plot objects.

Although *aesthetics* are usually mapped to variables in the data, constant aesthetic values can be passed as arguments to layer functions, consistently controlling a property of all elements in a layer. While variables in `data` can be both mapped using `aes()` as whole-plot defaults, as shown above, or within individual layers, constant values for aesthetics have to be set, as shown here, as named arguments passed directly to layer functions, instead of to a call to `aes()`.

```r
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point(color = "blue", shape = "square")
```
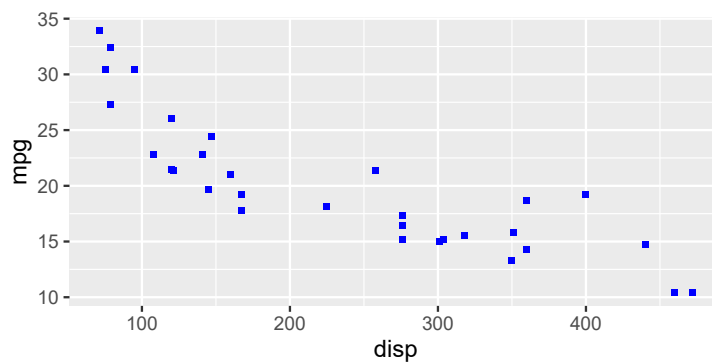
⚠ Mapping an aesthetic to a constant value within a call to `aes()` adds a column containing this value to the data frame received as input by the `stat()`. This value is not interpreted as an aesthetic value but instead as a data value. The plot above, but using a call to `aes()`.

```r
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point(aes(color = "blue", shape = "square"))
```
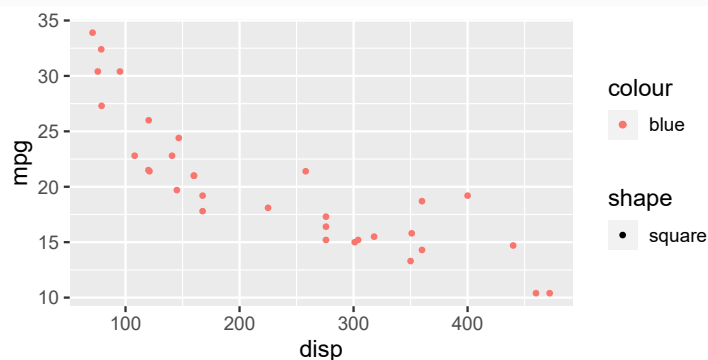

The plot contains red circles instead of blue squares!

In principle, one could correct this plot by adding suitable `scales` but this would be still wasteful by unnecessarily storing many copies of the constant `"blue"` in the `"gg"` plot object.

While a geometry directly constructs during rendering a graphical representation of the observations or summaries in the data it receives as input, a *statistics* or `stat` "sits" in-between the data and a `geom`, applying some computation, usually but not always, to produce a statistical summary of the data. Here `stat_smooth()` fits a linear regression (see section **??** on page **??**) and passes the resulting predicted values to `geom_line()`. Passing `method = "lm"` selects `lm()` as the model fitting function. Passing `formula = y ~ x` sets the model to be fitted. This plot has two layers, one from geometries `geom_point()` and one from `geom_line()`.

```r
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point() +
  stat_smooth(geom = "line", method = "lm", formula = y ~ x)
```
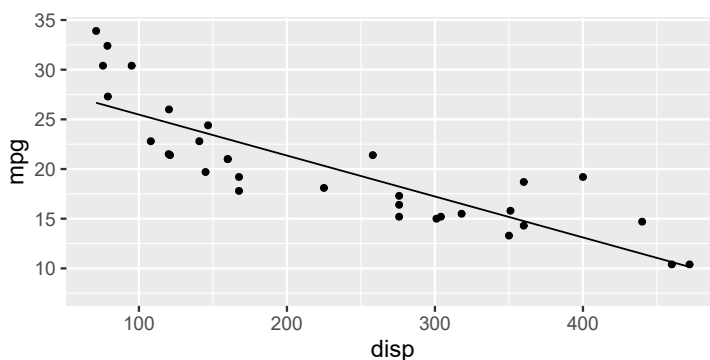
The plots above relied on defaults for *scales*, *coordinates* and *themes*. In the examples below the defaults are overridden by arguments that produce differently rendered plots. Adding `scale_y_log10()` applies a logarithmic transformation to the values mapped to y. This works like plotting using graph paper with rulings spaced according to a logarithmic scale. Tick marks continue to be expressed in the original units, but statistics are applied to the transformed data. In other words, the transformation specified in the scale affects the values in advance of the **start** stage, before they are mapped to aesthetics and passed to *statistics*. Thus, in this example the linear regression is fitted to `log10()` transformed y values and the original x values.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point() +
  stat_smooth(geom = "line", method = "lm", formula = y ~ x) +
  scale_y_log10()
```
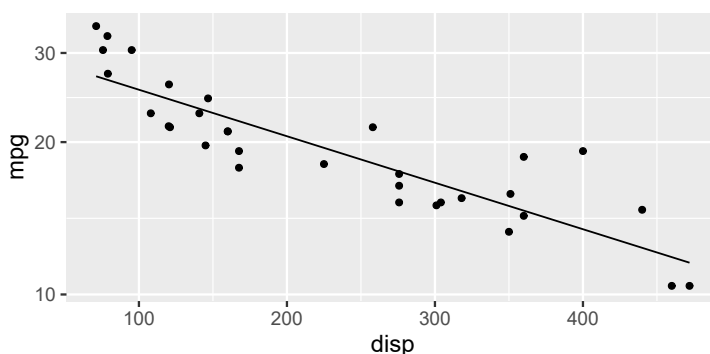

The range limits of a scale can be set manually, instead of automatically as by default. These limits create a virtual *window into the data*: out-of-bounds (oob) observations, those outside the scale limits remain hidden and are not mapped to aesthetics—i.e., these observations are not included in the graphical representation or used in calculations. Crucially, when using *statistics* the computations are only applied to observations that fall within the limits of all scales in use. These limits *indirectly* affect the plotting area when the plotting area is automatically set based on the range of the (within limits) data—even the mapping to values of a different aesthetics may change when a subset of the data are selected by manually setting the limits of a scale.

In contrast to *scale limits*, *coordinates* function as a *zoomed view* into the plotting area, and do not affect which observations are visible to *statistics*. The coordinate system, as expected, is also determined by this grammar element—here we use cartesian coordinates which are the default, but we manually set y limits.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point() +
  stat_smooth(geom = "line", method = "lm", formula = y ~ x) +
  coord_cartesian(ylim = c(15, 25))
```
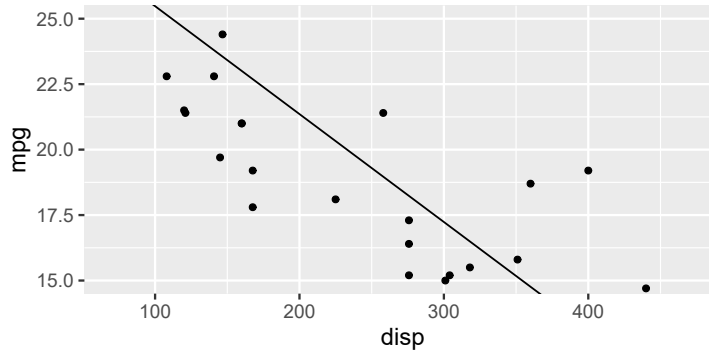

The next example uses a coordinate system transformation. When the transformation is applied to the coordinate system, it affects only the plotting—it sits between the geom and the rendering of the plot. The transformation is applied to the values that were returned by *statistics*. The straight line fitted is plotted on the transformed coordinates as a curve, because the model was fitted to the untransformed data obtaining untransformed predicted values. The coordinate transformation is applied to these predicted values and plotted. (Other coordinate systems are described in sections 1.5.6 and 1.11 on pages 35 and 85, respectively.)

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point() +
  stat_smooth(geom = "line", method = "lm", formula = y ~ x) +
  coord_trans(y = "log10")
```
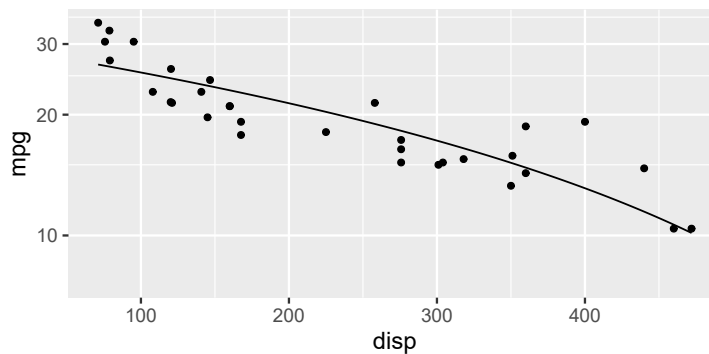

Themes affect the rendering of plots at the time of printing—they can be thought of as style sheets defining the graphic design. A complete theme can override the default gray theme. The plot is the same, the observations are represented in the same way, the limits of the axes are the same and all text is the same. On

the other, hand how these elements are rendered by different themes can be drastically different.

```r
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point() +
  theme_classic()
```
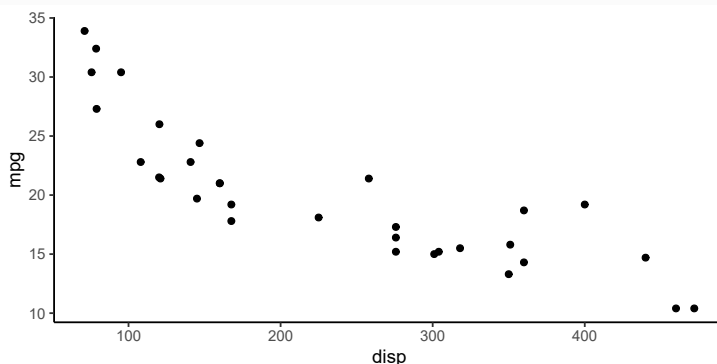

Both the base font size and the base font family can be changed. The base font size controls the size of all text elements, as other sizes are defined relative to the base size. How the plot looks changes when using the same theme as in the previous example, but with a different base point size and font family for text elements. (The use of themes is discussed in section 1.12 on page 88.)

```r
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point() +
  theme_classic(base_size = 20, base_family = "serif")
```
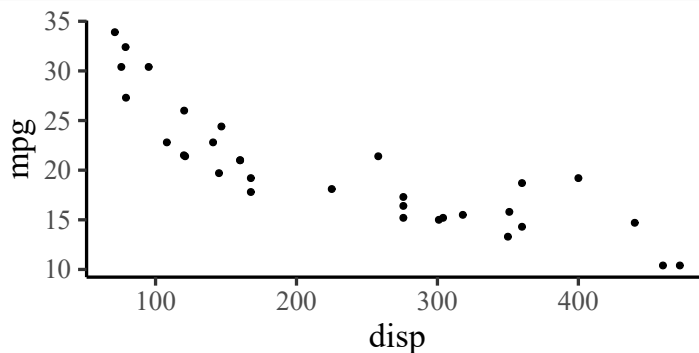

How to set axis labels, tick positions and tick labels will be discussed in depth in section 1.9 on page 67. Function `labs()` is *a convenience function* used to set the title and subtitle of a plot and to replace the default `name` of scales, in this case, those used for axis labels—the default `name` of scales is the name of the mapped variable. In the call to `labs()` the names of aesthetics are used as if they were formal parameters with character strings, or R expressions. as arguments. Below `x` and `y` are the names of the two *aesthetics* to which two variables in `data` were mapped, `disp` and `mpg`. Formal parameters `title` and `subtitle` add these plot elements. (The escaped character \n stands for new line, see section ?? on page ??.)

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg)) +
  geom_point() +
  labs(x = "Engine displacement (cubic inches)",
       y = "Fuel use efficiency\n(miles per gallon)",
       title = "Motor Trend Car Road Tests",
       subtitle = "Source: 1974 Motor Trend US magazine")
```
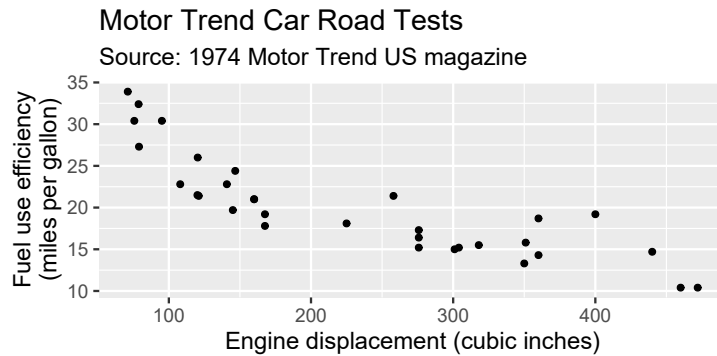

As elsewhere in R, when a value is expected, either a value stored in a variable or a more complex statement returning a suitable value can be passed as an argument to be mapped to an *aesthetic*. In other words, the values to be plotted do not need to be stored as variables (or columns) in the data frame passed as an argument to parameter `data`, they can also be computed from these variables. Here we plot miles-per-gallon, `mpg` on the engine displacement per cylinder by dividing `disp` by `cyl` within the call to `aes()`.

```
ggplot(data = mtcars, aes(x = disp / cyl, y = mpg)) +
  geom_point()
```
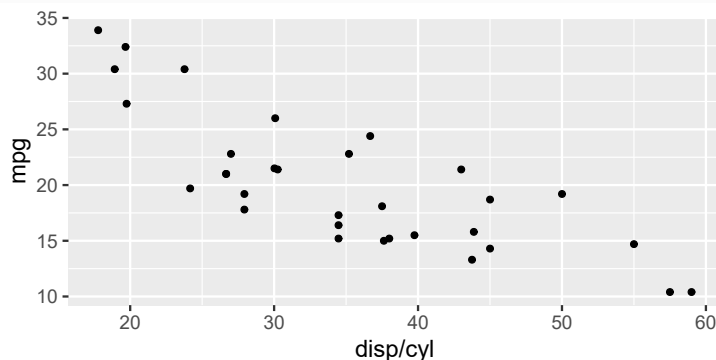

Each of the elements of the grammar exemplified above has several different member functions, and many of the individual *geometries* and *statistics* accept arguments that can be used to modify their behavior. There are also more *aesthetics* than those shown above. Multiple data objects as well as multiple mappings can coexist within a single "gg" plot object. Packages and user code can define new *geometries*, *statistics*, *coordinates* and even implement new *aesthetics*. Being `ggplot()` an S3 method, specializations for objects of classes different from `data.frame` exist. Individual elements in a theme can also be modified and new complete themes created, re-used and shared. We will describe in the remaining sections of this

chapter how to use the grammar of graphics to construct other types of graphical presentations including more complex plots than those in the examples above.

1.4.4   Plots as R objects

We can manipulate `"gg"` plot objects and their components in the same way as other R objects. We can operate on them using the operators and methods defined for the `"gg"` class they belong to. We start by saving, similarly as above, a `"gg"` plot object in variable `p1`.

```r
p1 <- ggplot(data = mtcars,
             aes(x = disp, y = mpg)) +
      geom_point()
```

When we used in the previous section operator + to assemble the plots, we were operating on "anonymous" R objects. In the same way, we can operate on saved or "named" objects.

```r
p1 +
  stat_smooth(geom = "line", method = "lm", formula = y ~ x)
```
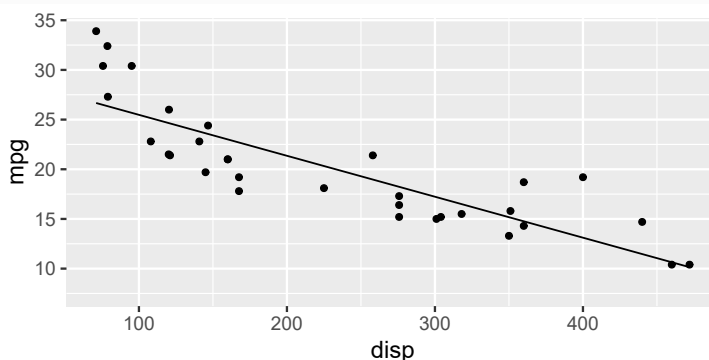

🖮 **1.2** Reproduce the examples in the previous section, using `p` defined above as a basis instead of building each plot from scratch.

🖵   In the examples above we have been adding elements one by one, using the + operator. It is also possible to add multiple components in a single operation using a list. This is useful, when we want to save sets of components in a variable so as to reuse them in multiple plots. This saves typing, ensures consistency and can make alterations to a set of similar plots much easier.

```r
p.ls <- list(
  stat_smooth(geom = "line", method = "lm", formula = y ~ x),
  scale_x_log10())

p1 +
  p.ls
```
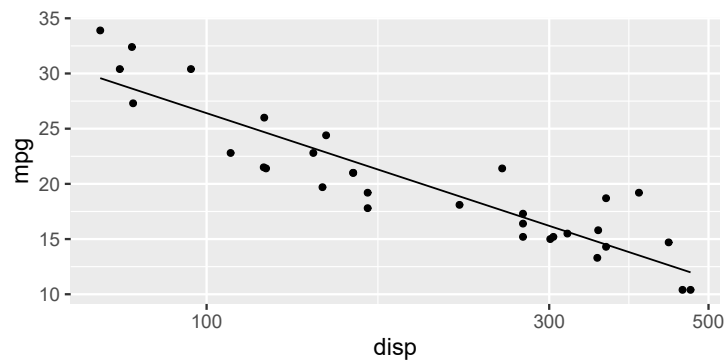
⚠ The separation of plot construction and rendering is possible, because `"gg"` objects are self-contained. A copy of the data object passed as argument is saved within the plot object, similarly as in model-fit objects. In the example above, `p1` by itself could be saved to a file on disk and loaded into a clean R session, even on another computer, and rendered as long as package 'ggplot2' and its dependencies are available. Another consequence of storing a copy of the data in the plot object, is that later changes to the data object used to create a `"gg"` object are *not* reflected in newly rendered plots from this object: we must create a new "gg" object.

⌨ We can look in more detail at how the *recipe* to make a plot is stored in a `"gg"` plot object. Objects of class `"gg"` are of mode `"list"`. In R, lists can contain heterogeneous members. They contain data, function definitions, and unevaluated expressions. In other words the data plus instructions to transform the data, to map them into graphic objects, and various aspects of the rendering from scale limits to type faces to use. (R lists are described in section **??** on page **??**.)

As an example we show the top level members of the `"gg"` plot object `p1`, a simple plot. Method `summary()` shows the components without making explicit the structure of the object.

```
summary(p1)
## data: mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb [32x11]
## mapping:   x = ~disp, y = ~mpg
## faceting: <ggproto object: Class FacetNull, Facet, gg>
##     compute_layout: function
##     draw_back: function
##     draw_front: function
##     draw_labels: function
##     draw_panels: function
##     finish_data: function
##     init_scales: function
##     map_data: function
##     params: list
##     setup_data: function
##     setup_params: function
##     shrink: TRUE
##     train_scales: function
##     vars: function
##     super:  <ggproto object: Class FacetNull, Facet, gg>
## --------------------------------
## geom_point: na.rm = FALSE
## stat_identity: na.rm = FALSE
## position_identity
```

Method `str()` shows the structure of objects and can be also used to advantage with ggplots (not shown). Alternatively, `names()` extracts the names of the top-level members of `p1`.

```
names(p1)
##  [1] "data"        "layers"      "scales"    "guides"    "mapping"
##  [6] "theme"       "coordinates" "facet"     "plot_env"  "labels"
```

⌨ 🖥 **1.3** Explore in more detail the different members of object `p1`. For example, the code statement below extracts member `"layers"` from object `p1` and display its structure.

```
str(p1$layers, max.level = 1)
```

How many layers are present in this case?

1.4.5   Mappings in detail

In the case of simple plots, based on data contained in a single data frame, the usual style is to code a plot as described above, passing an argument, `mtcars` in these examples, to the `data` parameter of `ggplot()`. Data passed in this way becomes the default for all layers in the plot. The same applies to the argument passed to `mapping`.

```
ggplot(data = mtcars,
       mapping = aes(x = disp, y = mpg)) +
  geom_point()
```

However, the grammar of graphics contemplates the possibility of data and mappings restricted to individual layers, passed to statistics or geometries through their `mapping` formal parameter. In this case, those mappings set in the call to

`ggplot()`, if present, are overridden by arguments passed to individual layers, making it possible to code the same plot as follows.

```
ggplot() +
  geom_point(data = mtcars,
             mapping = aes(x = disp, y = mpg))
```

The last two examples above show two different approaches. When plots have a single layer, they are equally convenient. If a plot has multiple layer based on the same data, the first of this approaches makes this clear. If each layer uses different data, the second approach is better than using a combination of the two approaches. As discussed earlier, using a consistent and clear style when coding is important.

In some cases, when flexibility is needed while constructing complex plots with multiple layers other idioms can be preferable, e.g., when assembling a plot from "pieces´´ stored in variables or built programmatically. Some examples are given below.

The default mapping can also be added directly with the + operator, instead of being passed as an argument to `ggplot()`.

```
ggplot(data = mtcars) +
  aes(x = disp, y = mpg) +
  geom_point()
```

It is also possible to have a default mapping for the whole plot, but no default data.

```
ggplot() +
  aes(x = disp, y = mpg) +
  geom_point(data = mtcars)
```

We can save the mapping to a variable and add the variable instead of the call to `aes()` in each of the examples above, of which we show only the first one.

```
my.mapping <- aes(x = disp, y = mpg)
ggplot(data = mtcars,
       mapping = my.mapping) +
  geom_point()
```

In all these examples, the plot remains unchanged (not shown). However, the flexibility of the grammar allows the assembly of plots from separately constructed pieces and reusing these pieces by storing them in variables.

The mapping of aesthetics does not necessarily have to be to variable in `data`, it can be a code statement that returns a value computed from one of more variables in `data` and constant values. In this simple example, `mpg` in miles per gallon is first converted into km per litre.

```
my.mapping <- aes(x = disp, y = mpg * 0.43)
ggplot(data = mtcars,
       mapping = my.mapping) +
  geom_point()
```
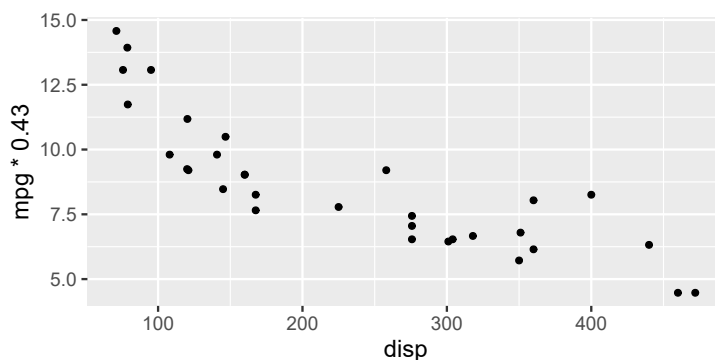
Subsetting of data is in principle not possible within a call to `aes()`, but see below. When a plot has default data, the argument passed to parameter `data` of a layer function, can be a function returning a data frame. In this case, the function is applied to the default data and the data frame it returns used in the layer. In the example below an anonymous function defined in-line, extracts a subset of rows in data and highlights it by overplotting with a different color. The argument passed to data is a function definition, not a call, thus, if passed by name, no parentheses are used.

```
ggplot(data = mtcars,
       mapping = aes(x = disp, y = mpg)) +
  geom_point(size = 4) +
  geom_point(data = function(x){subset(x = x, cyl == 4)},
             color = "yellow", size = 1.5)
```
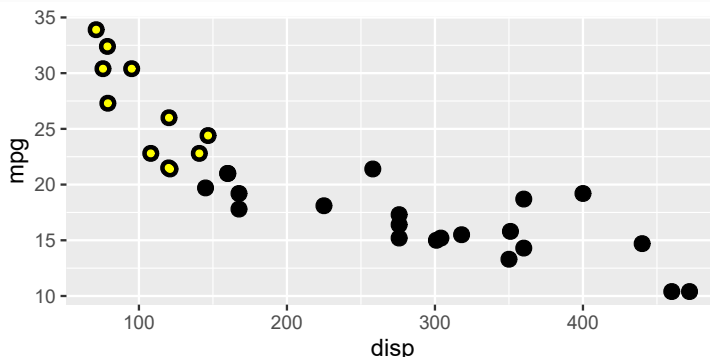

The plot's default data can also be operated upon using the 'magrittr' pipe operator, but not the pipe operator native to R (|>) or the dot-pipe operator from 'wrapr' (see section ?? on page ??). In my opining, using a function as above is simpler and clearer.

```
ggplot(data = mtcars,
       mapping = aes(x = disp, y = mpg)) +
  geom_point(size = 4) +
  geom_point(data = . %>% subset(x = ., cyl == 4), color = "yellow",
             size = 1.5)
```

For this example, an alternative approach is to put the test condition within the call to `aes()`. Setting `size = 0` draws a very small point, but visible. Using NA instead, skips the rows, and `na.rm = TRUE` silences the warning. This is a roundabout approach to subsetting.

```
ggplot(data = mtcars,
       mapping = aes(x = disp, y = mpg)) +
  geom_point(size = 4) +
  geom_point(color = "yellow",
             aes(size = ifelse(cyl == 4, 1.5, NA)),
             na.rm = TRUE) +
  scale_size_identity()
```
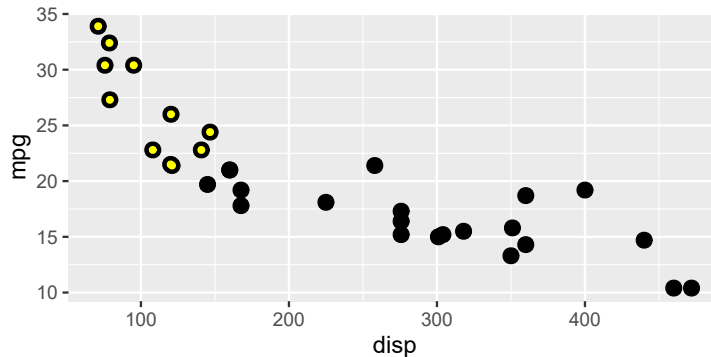

▣ *Late mapping* of variables to aesthetics has been possible in 'ggplot2' for a long time using as notation enclosure of the name of a variable returned by a statistic between `..`, but this notation has been deprecated some time ago and replaced by `stat()`. In both cases, this imposed a limitation: it was impossible to map a computed variable to the same aesthetic as input to the statistic and to the geometry in the same layer. There were also some other quirks that prevented passing some arguments to the geometry through the dots `...` parameter of a statistic.

Since version 3.3.0 of 'ggplot2' the syntax used for mapping variables to aesthetics is based on functions `stage()`, `after_stat()` and `after_scale()`. Function `after_stat()` replaces both `stat()` and the `..` notation.

The documentation of 'ggplot2' gives several good examples of cases when the new syntax is useful. I give here a different example. A polynomial fitted to data using `rlm()`. RLM is a procedure that automatically assigns before computing the residual sums of squares, weights to the individual residuals in an attempt to protect the estimated fit from the influence of extreme observations or outliers. When using this and similar methods it is of interest to plot the residuals together with the weights. A frequent approach is to map weights to a gradient between two colours. The code below constructs a data frame containing artificial data that includes an extreme value or outlier.

```
set.seed(4321)
X <- 0:10
Y <- (X + X^2 + X^3) + rnorm(length(X), mean = 0, sd = mean(X^3) / 4)
df1 <- data.frame(X, Y)
df2 <- df1
df2[6, "Y"] <-df1[6, "Y"] * 10
```

In the first plot `after_stat()` is used to map variable `weights` computed by the statistic to the `colour` aesthetic. In `stat_fit_residuals()`, `geom_point()` is used by default. This figure shows the raw residuals with no weights applied (mapped to

y by default), and the computed weights (with range 0 to 1) encoded by colours ranging between red and blue.

```
ggplot(df2, aes(x = X, y = Y)) +
  stat_fit_residuals(formula = y ~ poly(x, 3, raw = TRUE), method = "rlm",
                     mapping = aes(colour = after_stat(weights)),
                     show.legend = TRUE) +
  scale_color_gradient(low = "red", high = "blue", limits = c(0, 1),
                       guide = "colourbar")
```
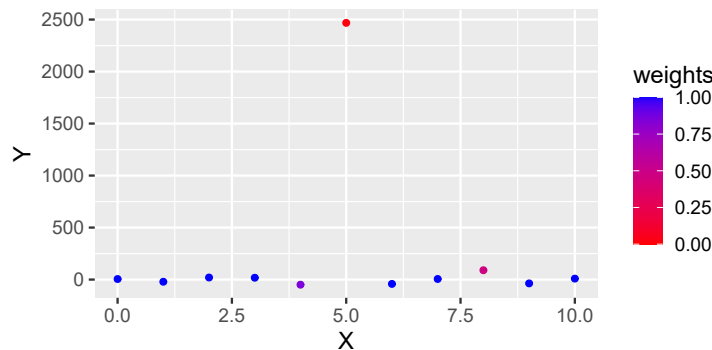

In the second plot weighted residuals are mapped to the y aesthetic, and weights, as above, to the color aesthetic. A call to `stage()` can distinguish the mapping ahead of the statistic (`start`) from that after the statistic, i.e., ahead of the geometry. As above, the default geometry, `geom_point()` is used. The mapping in this example can be read as: the variable x from the data frame `df2` is mapped to the x aesthetic at all stages. Variable Y from the data frame `df2` is mapped to the y aesthetic ahead of the computations in `stat_fit_residuals()`. After the computations, variables y and `weights` in the data frame returned by `stat_fit_residuals()` are multiplied and mapped to the y ahead of `geom_point()`.

```
ggplot(df2) +
  stat_fit_residuals(formula = y ~ poly(x, 3, raw = TRUE),
                     method = "rlm",
                     mapping = aes(x = X,
                                   y = stage(start = Y,
                                             after_stat = y * weights),
                                   colour = after_stat(weights)),
                     show.legend = TRUE) +
  scale_color_gradient(low = "red", high = "blue", limits = c(0, 1),
                       guide = "colourbar")
```
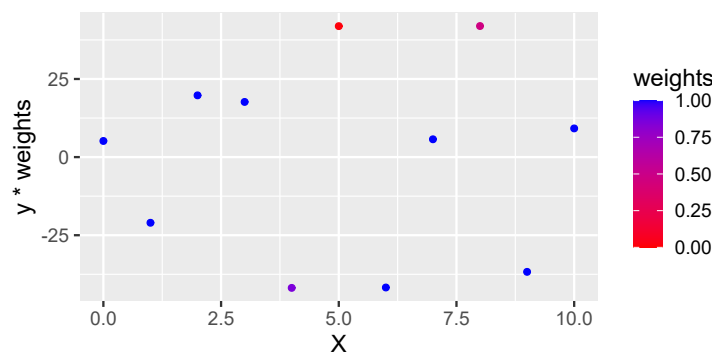
⊡ When fitting models to observations with `lm()`, the un-weighted residuals are used compute the sum of squares. Function `lm()` supports the use of weights supplied by the user. In `rlm()` the weights are computed automatically.

1.5 Geometries

Different geometries support different *aesthetics*. While `geom_point()` supports `shape`, and `geom_line()` supports `linetype`, both support `x`, `y`, `color` and `size`. In this section I describe frequently used `geometries` from package 'ggplot2' and from a few packages that extend 'ggplot2'. The graphic output from some code examples will not be shown, with the expectation that readers will run them to see the plots.

Mainly for historical reasons, *geometries* accept a *statistic* as an argument, in the same way as *statistics* accept a *geometry* as an argument. In this section I only describe *geometries* which have as a default *statistic* `stat_identity`. *Geometries* that have other *statistics* as default are described in section 1.6.2 together with the corresponding *statistics*.

1.5.1 Point

As shown earlier in this chapter, `geom_point()`, can be used to add a layer with observations represented by "points" or symbols. Variable `cyl` describes the numbers of cylinders in the engines of the cars. It is a numeric variable, and when mapped to color, a continuous color scale is used to represent this variable.

The first examples build scatter plots, because numeric variables are mapped to both `x` and `y`. Some scales, like those for `color`, exist in two "flavors," one suitable for numeric variables (continuous) and another for factors (discrete).

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, color = cyl)) +
  geom_point()
```
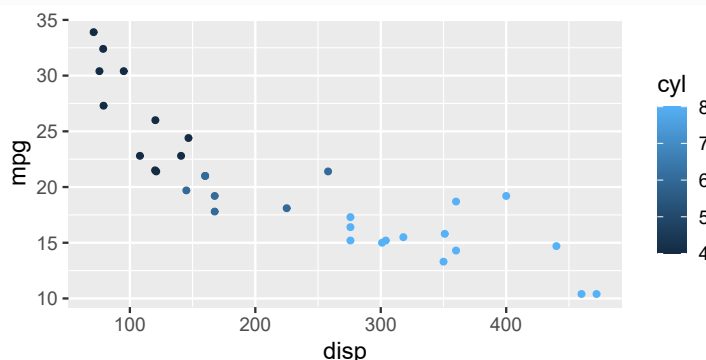

If we convert `cyl` into a factor, a discrete color scale is used instead of a continuous one.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, color = factor(cyl))) +
  geom_point()
```

If we convert `cyl` into an ordered factor, a different discrete color scale is used by default.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, color = ordered(cyl))) +
  geom_point()
```

⌨ **1.4** Try a different mapping: `disp` → `color`, `cyl` → `x`. Continue by using `help(mtcars)` and/or `names(mtcars)` to see what variables are available, and then try the combinations that trigger your curiosity—i.e., explore the data.

The mapping between data values and aesthetic values is controlled by scales. Different color scales, and even palettes within a given scale, provide different mappings between data values and rendered colours.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, color = factor(cyl))) +
  geom_point() +
  scale_color_brewer(type = "qual", palette = 2)
```

The data, aesthetics mappings, and geometries are the same as in earlier code; to alter how the plot looks, we have changed only the scale and palette used for the color aesthetic. Conceptually it is still exactly the same plot we created earlier, except for the colours used. This is a very important point to understand, because it allows us to separate two different concerns: the semantic structure and the graphic design.

⌨ **1.5** Try the different palettes available through the brewer scale. You can play directly with the palettes using function `brewer_pal()` from package 'scales' together with `show_col()`).

```
show_col(brewer_pal()(3))
show_col(brewer_pal(type = "qual", palette = 2, direction = 1)(3))
```

 Once you have found a suitable palette for these data, redo the plot above with the chosen palette.

When not relying on colors, the most common way of distinguishing groups of observations in scatter plots is to use the `shape` of the points as an *aesthetic*. We need to change a single "word" in the code statement to achieve this different mapping.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, shape = factor(cyl))) +
  geom_point()
```

We can use `scale_shape_manual` to choose each shape to be used. We set three "open" shapes that we will see later are very useful as they obey both `color` and `fill` *aesthetics*.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, shape = factor(cyl))) +
  geom_point() +
  scale_shape_manual(values = c(21, 22, 23))
```

It is also possible to use characters as shapes. The character is centered on the

position of the observation. As the numbers used as symbols are self-explanatory, we suppress the default guide or key.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, shape = factor(cyl))) +
  geom_point(size = 2.5) +
  scale_shape_manual(values = c("4", "6", "8"), guide = "none")
```
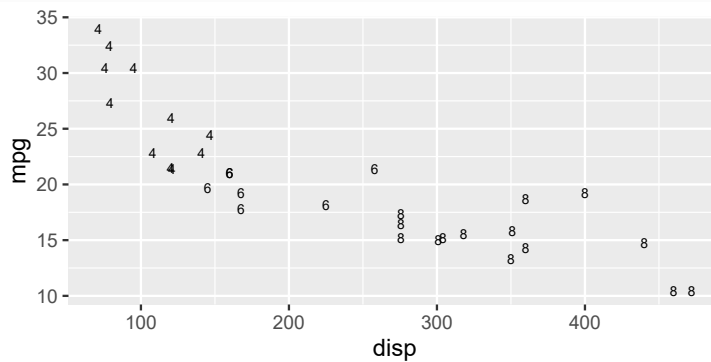

One variable in the data can be mapped to more than one aesthetic, allowing redundant aesthetics. This may seem wasteful, but it is extremely useful as it allows one to produce figures that, even when produced in color, can still be read if reproduced as black-and-white images.

```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          shape = factor(cyl),
                          color = factor(cyl))) +
  geom_point()
```

Dot plots are similar to scatter plots but a factor is mapped to either the x or y *aesthetic*. Dot plots are prone to have overlapping observations, and one way of making these points visible is to make them partly transparent by setting a constant value smaller than one for the `alpha` *aesthetic*.

```
ggplot(data = mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_point(alpha = 1/3)
```
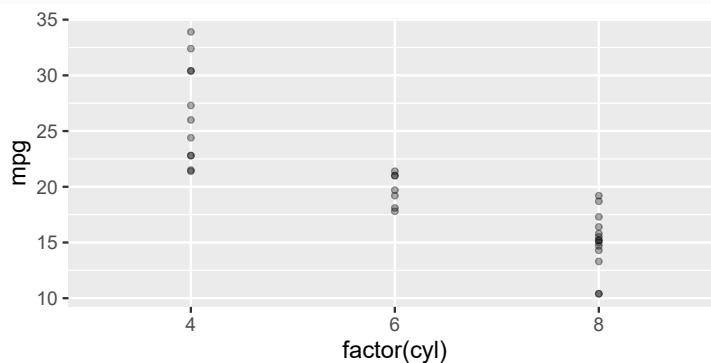

Function `position_identity()`, which is the default for `geom_point()`, does not alter the coordinates or position of observations, as shown in all examples above. To make overlapping observations visible, instead of making the points semitransparent as above, we can randomly displace them. This is called *jitter*, and can be added using `position_jitter()` as argument to formal parameter `position`. The

amount of jitter is set by nemeric arguments passed to `width` and/or `height`, given as a fraction of the distance between adjacent factor levels in the plot.

```
ggplot(data = mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_point(position = position_jitter(width = 0.25, heigh = 0.5))
```
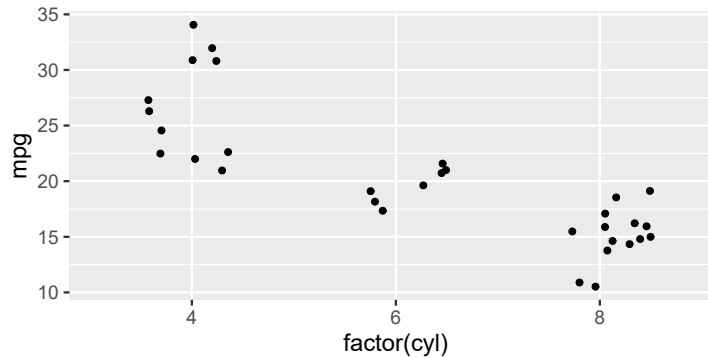

⚠ The name as a character string can be also used when no arguments need to be passed to the *position* function, and for some positions by passing numerical arguments to specific parameters of geometries. However, the default width of ±0.5 tends to be rarely optimal.

```
ggplot(data = mtcars, aes(x = factor(cyl), y = mpg), colour = factor(cyl)) +
  geom_point(position = "jitter")
```
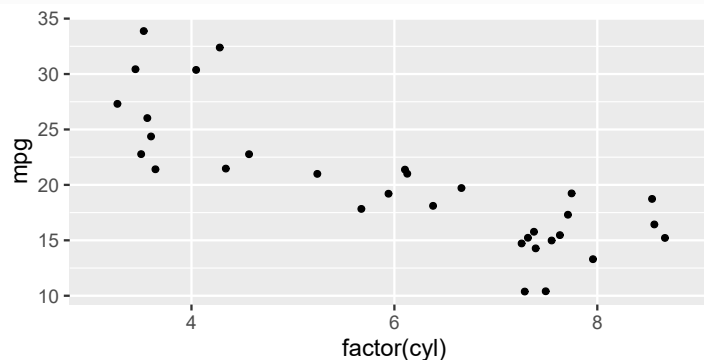

🖵 The displacement introduced by jitter and nudge differ in that jitter is random, and nudge deterministic. In each case the displacement can be separately adjusted vertically and horizontally. Jitter, as shown above is useful when we desire to make visible overlapping points. Nudge is most frequently used with data labels to avoid occluding points or other graphical features.

Layer function `geom_point_s()` from package 'ggpp' is used below to make the displacement visible by drawing an arrow connecting original and displaced positions for each observation. We need to use the `_keep` flavor of the position functions for arrows to be drawn.

```
ggplot(data = mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_point(colour = "blue") +
  geom_point_s(position = position_jitter_keep(width = 0.25, heigh = 0.5),
               colour = "red")
```
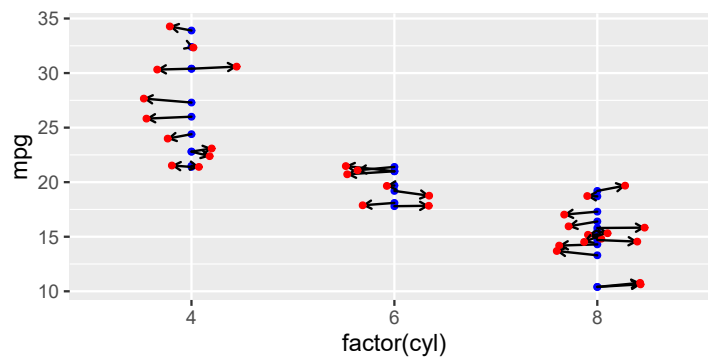
The amount of nudging is set by a distance expressed in data units through parameters x and y. (Factors have mode `numeric` and each level is represented by an integer, thus distance between levels of a factor is 1.)

```
ggplot(data = mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_point(colour = "blue") +
  geom_point_s(position = position_nudge_keep(x = 0.2, y = 1),
               colour = "red")
```
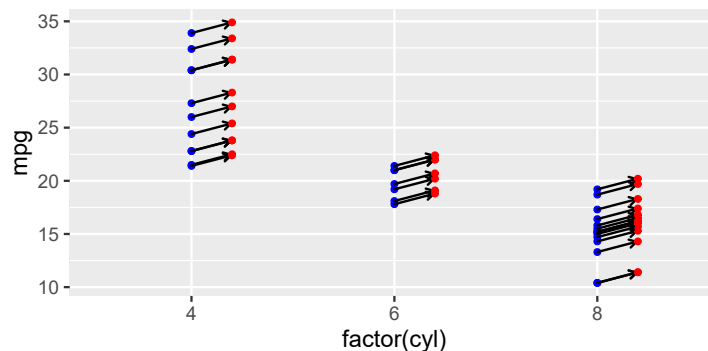

We can create a "bubble" plot by mapping the `size` *aesthetic* to a continuous variable. In this case, one has to think what is visually more meaningful. Although the radius of the shape is frequently mapped, due to how human perception works, mapping a variable to the area of the shape is more useful by being perceptually closer to a linear mapping. For this example we add a new variable to the plot. The weight of the car in tons and map it to the area of the points.

```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          color = factor(cyl),
                          size = wt)) +
  scale_size_area() +
  geom_point()
```
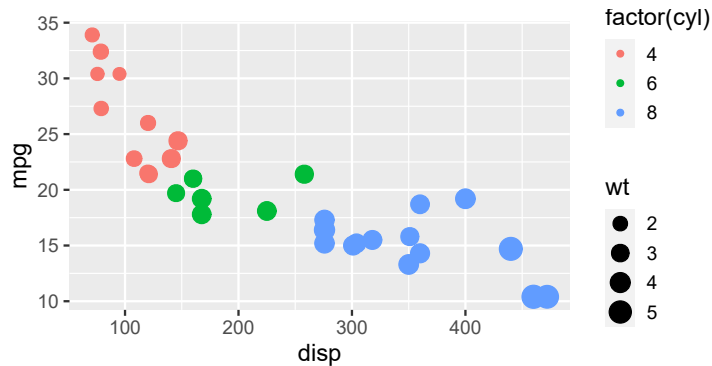
⌨ **1.6** If we use a radius-based scale the "impression" is different.

```r
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          color = factor(cyl),
                          size = wt)) +
  scale_size() +
  geom_point()
```

Make the plot, look at it carefully. Check the numerical values of some of the weights, and assess if your perception of the plot matches the numbers behind it.

As a final example summarizing the use of `geom_point()`, we combine different *aesthetics* and *scales* in the same scatter plot.

```r
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          shape = factor(cyl),
                          fill = factor(cyl),
                          size = wt)) +
  geom_point(alpha = 0.33, color = "black") +
  scale_size_area() +
  scale_shape_manual(values = c(21, 22, 23))
```
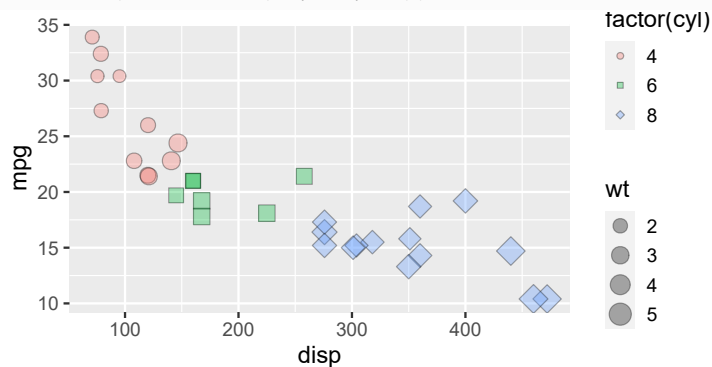

⌨ **1.7** Play with the code in the chunk above. Remove or change each of the mappings and the scale, display the new plot, and compare it to the one above. Continue playing with the code until you are sure you understand what graphical element in the plot is added or modified by each individual argument or "word" in the code statement.

It is common to draw error bars together with points representing means or medians of observations and `geom_pointrange()` achieves this task based on the values mapped to the `x`, `y`, `ymin` and `ymax`, using `y` for the position of the point

and `ymin` and `ymax` for the positions of the ends of the line segment representing a range. Two other *geometries*, `geom_range()` and `geom_errorbar()` draw only a segment or a segment with capped ends. They are frequently used together with *statistics* when summaries are calculated on the fly, but can also be used directly when the data summaries are stored in a data frame passed as an argument to `data`.

1.5.2   Rug

Rarely, rug plots are used by themselves. Instead they are usually an addition to scatter plots. An example of the use of `geom_rug()` follows. They make it easier to see the distribution of observations along the x- and y-axes.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, color = factor(cyl))) +
  geom_point() +
  geom_rug()
```
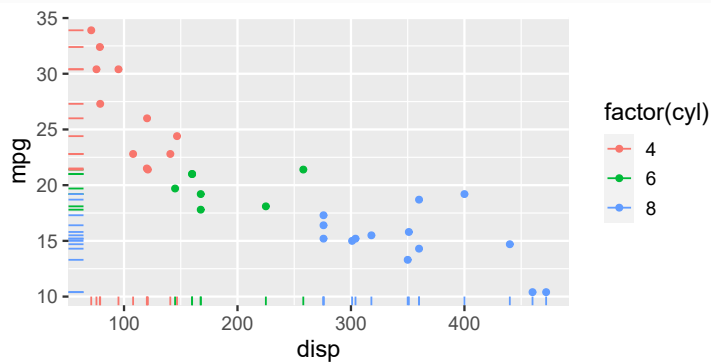

⚠ Rug plots are most useful when the local density of observations is not too high, otherwise rugs become too cluttered and the "rug threads" may overlap. When overlap is moderate, making the segments semitransparent by setting the `alpha` aesthetic to a constant value smaller than one, can make the variation in density easier to appreciate. When the number of observations is large, marginal density plots should be preferred.

1.5.3   Line and area

For line plots we use `geom_line()`. The `size` of a line is its thickness, and as we had `shape` for points, we have `linetype` for lines. In a line plot, observations in successive rows of the data frame, or the subset corresponding to a group, are joined by straight lines. We use a different data set included in R, `Orange`, with data on the growth of five orange trees. See the help page for `Orange` for details.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, linetype = Tree)) +
  geom_line()
```
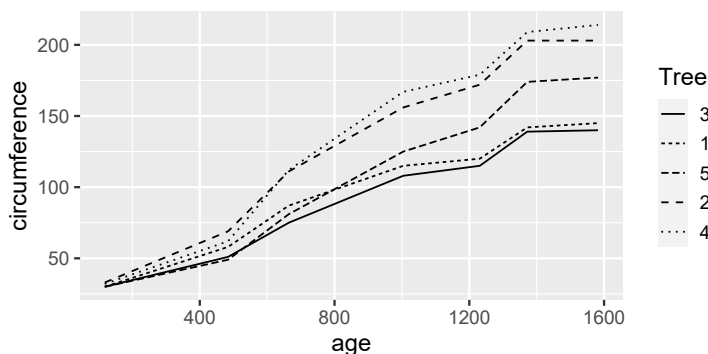
Instead of drawing a line joining the successive observations, we may want to draw a disconnected straight-line segment for each observation or row in the data. In this case, we use `geom_segment()` which accepts x, xend, y and yend as mapped aesthetics. `geom_curve()` draws curved lines, and the curvature, control points, and angles can be controlled through additional *aesthetics*. These two *geometries* support arrow heads at their ends. Other *geometries* useful for drawing lines or segments are `geom_path()`, which is similar to `geom_line()`, but instead of joining observations according to the values mapped to x, it joins them according to their row-order in `data`, and `geom_spoke()`, which is similar to `geom_segment()` but using a polar parametrization, based on x, y for origin, and `angle` and `radius` for the segment. Finally, `geom_step()` plots only vertical and horizontal lines to join the observations, creating a stepped line.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, linetype = Tree)) +
  geom_step()
```
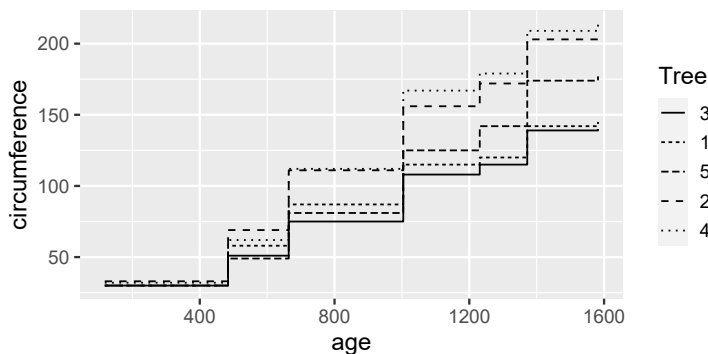

⌨ **1.8** Using the following toy data, make three plots using `geom_line()`, `geom_path()`, and `geom_step` to add a layer.

```
toy.df <- data.frame(x = c(1,3,2,4), y = c(0,1,0,1))
```

While `geom_line()` draws a line joining observations, `geom_area()` supports filling the area below the line according to the `fill` *aesthetic*. In contrast `geom_ribbon()` draws two lines based on the x, ymin and ymax *aesthetics*, with the space between the lines filled according to the `fill` *aesthetic*. Finally, `geom_polygon()` is similar to `geom_path()` but connects the extreme observations forming a closed polygon that supports `fill`.

Much of what was described above for `geom_point()` can be adapted to `geom_line()`, `geom_ribbon()`, `geom_area()` and other *geometries* described in this section. In some cases, it is useful to stack the areas—e.g., when the values represent parts of a bigger whole. In the next, contrived, example, we stack the growth of the different trees by using `position = "stack"` instead of the default `position = "identity"`. (Compare the y axis of the figure below to that drawn using `geom_line()` on page 29.)

```
ggplot(data = Orange,
       aes(x = age, y = circumference, fill = Tree)) +
  geom_area(position = "stack")
```
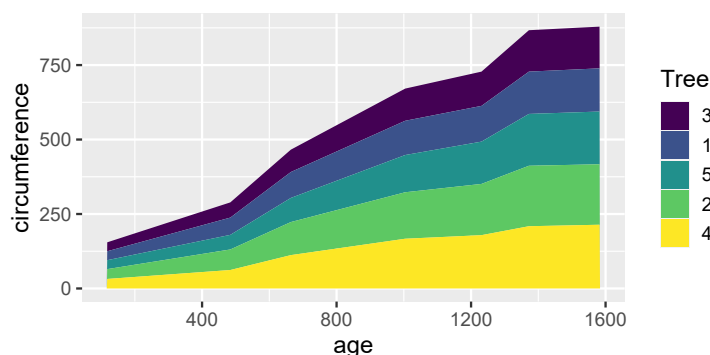

Finally, three *geometries* for drawing lines across the whole plotting area: `geom_hline()`, `geom_vline()` and `geom_abline()`. The first two draw horizontal and vertical lines, respectively, while the third one draws straight lines according to the *aesthetics* `slope` and `intercept` determining the position. The lines drawn with these three geoms extend to the edge of the plotting area.

`geom_hline()` and `geom_vline()` require a single aesthetic, `yintercept` and `xintercept`, respectively. Different from other geoms, the data for these aesthetics can also be passed as constant numeric vectors. The reason for this is that these geoms are most frequently used to annotate plots rather than plotting observations. Let's assume that we want to highlight an event at the age of 1000 days.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, fill = Tree)) +
  geom_area(position = "stack") +
  geom_vline(xintercept = 1000, color = "gray75") +
  geom_vline(xintercept = 1000, linetype = "dotted")
```
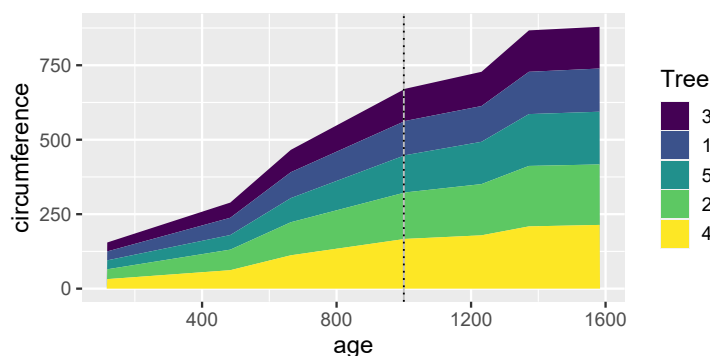
⌨ **1.9** Change the order of the three layers in the example above. How did the figure change? What order is best? Would the same order be the best for a scatter plot? And would it be necessary to add two `geom_vline()` layers?

1.5.4 Column

The *geometry* `geom_col()` can be used to create *column plots* where each bar represents an observation or case in the data.

⚠ R users not familiar yet with 'ggplot2' are frequently surprised by the default behavior of `geom_bar()` as it uses `stat_count()` to produce a histogram, rather than plotting values as is (see section 1.6.4 on page 51). `geom_col()` is identical to `geom_bar()` but with `"identity"` as the default statistic.

We create artificial data that we will reuse in multiple variations of the next figure.

```
set.seed(654321)
my.col.data <- data.frame(treatment = factor(rep(c("A", "B", "C"), 2)),
                          group = factor(rep(c("male", "female"), c(3, 3))),
                          measurement = rnorm(6) + c(5.5, 5, 7))
```

First we plot data for females only, using defaults for all *aesthetics* except x and y which we explicitly map to variables.

```
ggplot(subset(my.col.data, group == "female"),
       aes(x = treatment, y = measurement)) +
  geom_col()
```
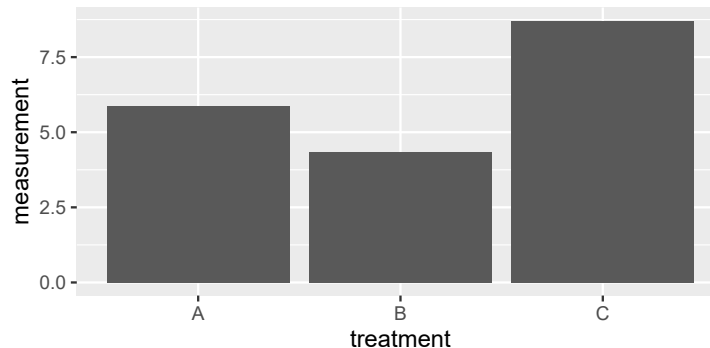

We play with *aesthetics* to produce a plot with a semi-formal style—e.g., suitable for a science popularization article or book. See section 1.9 and section 1.12 for information on scales and themes, respectively. We set `width = 0.5` to make the bars narrower. Setting `color = "white"` overrides the default color of the lines bordering the bars.

```
ggplot(my.col.data, aes(x = treatment, y = measurement, fill = group)) +
  geom_col(color = "white", width = 0.5) +
  scale_fill_grey() + theme_dark()
```
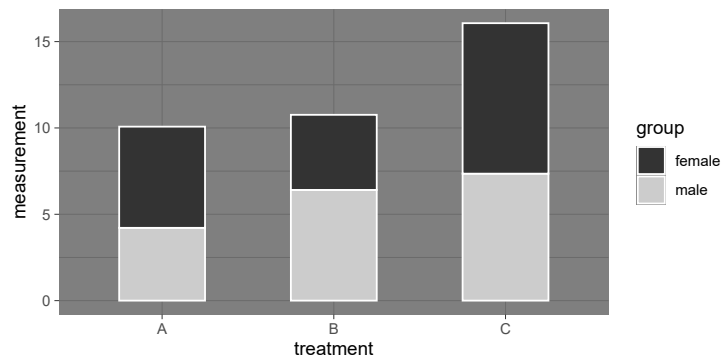
We next use a formal style, and in addition, put the bars side by side by setting `position = "dodge"` to override the default `position = "stack"`. Setting `color = NA` removes the lines bordering the bars.

```
ggplot(my.col.data, aes(x = treatment, y = measurement, fill = group)) +
    geom_col(color = NA, position = "dodge") +
    scale_fill_grey() + theme_classic()
```
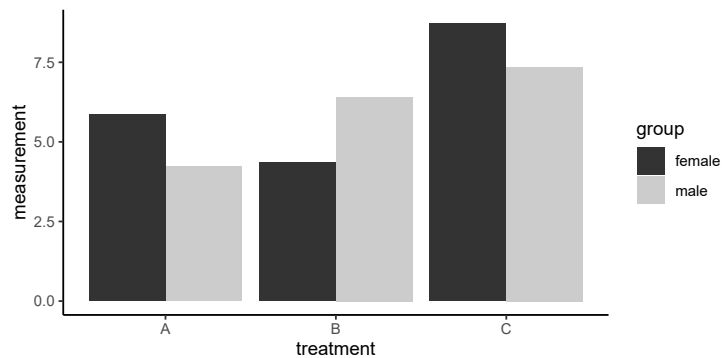

⌨ **1.10** Change the argument to `position`, or let the default be active, until you understand its effect on the figure. What is the difference between *positions* `"identity"`, `"dodge"` and `"stack"`?

⌨ **1.11** Use constants as arguments for *aesthetics* or map variable `treatment` to one or more of the *aesthetics* used by `geom_col()`, such as `color`, `fill`, `linetype`, `size`, `alpha` and `width`.

1.5.5 Tiles

We can draw square or rectangular tiles with `geom_tile()` producing tile plots or simple heat maps.

We here generate 100 random draws from the *F* distribution with degrees of freedom $v_1 = 5, v_2 = 20$.

```
set.seed(1234)
randomf.df <- data.frame(F.value = rf(100, df1 = 5, df2 = 20),
                         x = rep(letters[1:10], 10),
                         y = LETTERS[rep(1:10, rep(10, 10))])
```

`geom_tile()` requires aesthetics x and y, with no defaults, and `width` and `height` with defaults that make all tiles of equal size filling the plotting area.

```
ggplot(randomf.df, aes(x, y, fill = F.value)) +
  geom_tile()
```
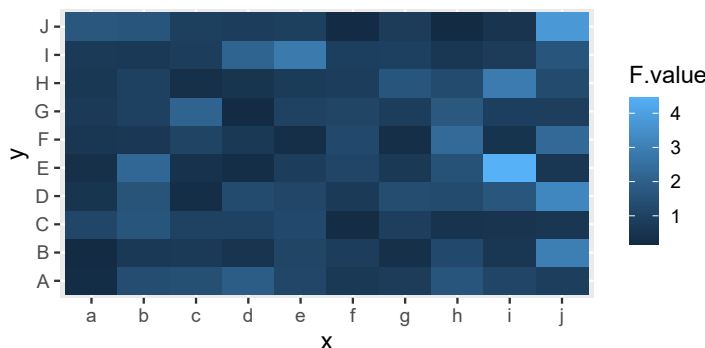

We can set `color = "gray75"` and `linewidth = 1.33` to make the tile borders more visible as in the example below, or use a contrasting color, to better delineate the borders of the tiles. What to use will depend on whether the individual tiles add meaningful information. In cases like when rows of tiles correspond to individual genes and columns to discrete treatments, the use of contrasting tile borders is preferable. In contrast, in the case when the tiles are an approximation to a continuous surface such as measurements on a regular spatial grid, it is best to suppress the tile borders.

```
ggplot(randomf.df, aes(x, y, fill = F.value)) +
  geom_tile(color = "gray75", linewidth = 1.33)
```
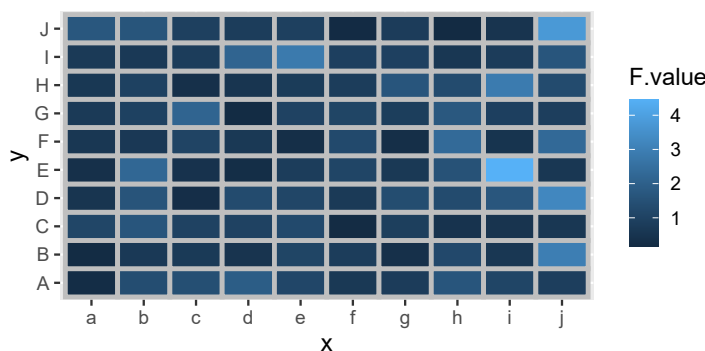

⌨ **1.12** Play with the arguments passed to parameters `color` and `size` in the example above, considering what features of the data are most clearly perceived in each of the plots you create.

Any continuous fill scale can be used to control the appearance. Here we show a tile plot using a gray gradient, with missing values in red.

```
ggplot(randomf.df, aes(x, y, fill = F.value) +
  geom_tile(color = "white") +
  scale_fill_gradient(low = "gray15", high = "gray85", na.value = "red")
```

In contrast to `geom_tile()`, `geom_rect()` draws rectangular tiles based on the position of the corners, mapped to aesthetics `xmin`, `xmax`, `ymin` and `ymax`.

1.5.6 Simple features (sf)

'ggplot2' version 3.0.0 or later supports the plotting of shape data similar to the plotting in geographic information systems (GIS) through geom_sf() and its companions, geom_sf_text(), geom_sf_label(), and stat_sf(). This makes it possible to display data on maps, for example, using different fill values for different regions. Special *coordinate* coord_sf() can be used to select different projections for maps. The *aesthetic* used is called geometry and contrary to all the other aesthetics we have seen until now, the values to be mapped are of class sfc containing *simple features* data with multiple components. Manipulation of simple features data is supported by package 'sf'. This subject exceeds the scope of this book, so a single and very simple example follows.

```
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
ggplot(nc) +
  geom_sf(aes(fill = AREA), color = "gray90")
```
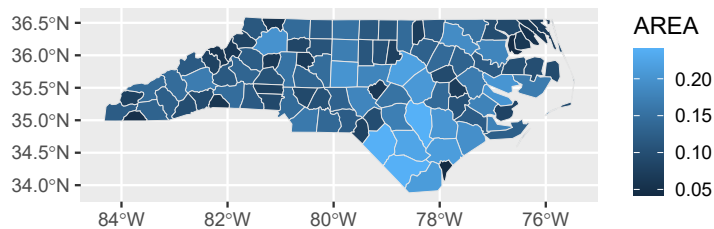

1.5.7 Text

We can use geom_text() or geom_label() to add text labels to observations. For geom_text() and geom_label(), the aesthetic label provides the text to be plotted and the usual aesthetics x and y, the location of the labels. As one would expect, the color and size aesthetics can also be used for the text.

```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          color = factor(cyl),
                          size = wt,
                          label = cyl)) +
  scale_size() +
  geom_point() +
  geom_text(color = "darkblue", size = 3)
```
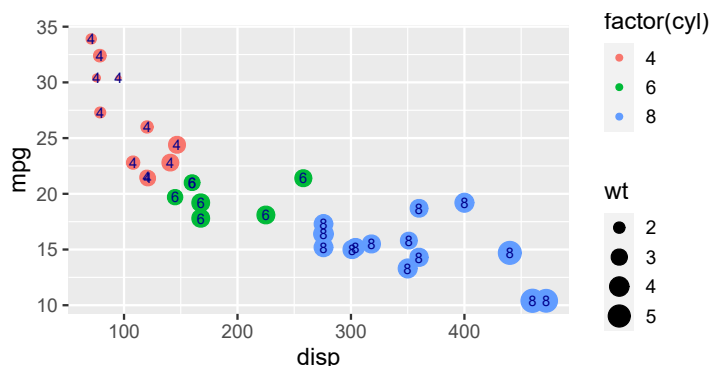
In addition, `angle` and `vjust` and `hjust` can be used to rotate the text and adjust its position. The default value of 0.5 for both `hjust` and `vjust` sets the center of the text at the supplied x and y coordinates. "Vertical" and "horizontal" for justification refer to the text, not the plot. This is important when `angle` is different from zero. Values larger than 0.5 shift the label left or down, and values smaller than 0.5, right or up with respect to its x and y coordinates. A value of 1 or 0 sets the text so that its edge is at the supplied coordinate. Values outside the range $0 \ldots 1$ shift the text even farther away, however, still using units based on the length or height of the text label. Recent versions of 'ggplot2' make possible justification using character constants for alignment: `"left"`, `"middle"`, `"right"`, `"bottom"`, `"center"` and `"top"`, and two special alignments, `"inward"` and `"outward"`, that automatically vary based on the position in the plotting area.

In the case of `geom_label()` the text is enclosed in a box, which obeys the `fill` *aesthetic* and takes additional parameters (described starting at page 38) allowing control of the shape and size of the box. However, `geom_label()` does not support rotation with the `angle` aesthetic.

⚠️ You should be aware that R and 'ggplot2' support the use of UNICODE, such as UTF8 character encodings in strings. If your editor or IDE supports their use, then you can type Greek letters and simple maths symbols directly, and they *may* show correctly in labels if a suitable font is loaded and an extended encoding like UTF8 is in use by the operating system. Even if UTF8 is in use, text is not fully portable unless the same font is available, as even if the character positions are standardized for many languages, most UNICODE fonts support at most a small number of languages. In principle one can use this mechanism to have labels both using other alphabets and languages like Chinese with their numerous symbols mixed in the same figure. Furthermore, the support for fonts and consequently character sets in R is output-device dependent. The font encoding used by R by default depends on the default locale settings of the operating system, which can also lead to garbage printed to the console or wrong characters being plotted running the same code on a different computer from the one where a script was created. Not all is lost, though, as R can be coerced to use system fonts and Google fonts with functions provided by packages 'showtext' and 'extrafont'. Encoding-related problems, especially in MS-Windows, are common.

In the remaining examples, with output not shown, we use `geom_text()` or

`geom_label()` together with `geom_point()` as this is how they may be used to label observations.

```
my.data <-
  data.frame(x = 1:5,
              y = rep(2, 5),
              label = c("a", "b", "c", "d", "e"))

ggplot(my.data, aes(x, y, label = label)) +
  geom_text(angle = 45, hjust = 1.5, size = 8) +
  geom_point()
```

⌨ **1.13** Modify the example above to use `geom_label()` instead of `geom_text()` using, in addition, the `fill` aesthetic.

In the next example we select a different font family, using the same characters in the Roman alphabet. The names `"sans"` (the default), `"serif"` and `"mono"` are recognized by all graphics devices on all operating systems. Additional fonts are available for specific graphic devices, such as the 35 "PDF" fonts by the `pdf()` device. In this case, their names can be queried with `names(pdfFonts())`.

```
ggplot(my.data, aes(x, y, label = label)) +
  geom_text(angle = 45, hjust = 1.5, size = 8, family = "serif") +
  geom_point()
```

⌨ **1.14** In the examples above the character strings were all of the same length, containing a single character. Redo the plots above with longer character strings of various lengths mapped to the `label` *aesthetic*. Do also play with justification of these labels.

Plotting (mathematical) expressions involves mapping to the `label` aesthetic character strings that can be parsed as expressions, and setting `parse = TRUE` (see section 1.14 on page 94). Here, we build the character strings using `paste()` but, of course, they could also have been entered one by one. This use of `paste()` provides an example of recycling of shorter vectors (see section ?? on page ??).

```
my.data <-
  data.frame(x = 1:5, y = rep(2, 5), label = paste("alpha[", 1:5, "]", sep = ""))
my.data$label
## [1] "alpha[1]" "alpha[2]" "alpha[3]" "alpha[4]" "alpha[5]"
```

Text and labels do not automatically expand the plotting area past their anchoring coordinates. In the example above, we need to use `expand_limits()` to ensure that the text is not clipped at the edge of the plotting area.

```
ggplot(my.data, aes(x, y, label = label)) +
  geom_text(hjust = -0.2, parse = TRUE, size = 6) +
  geom_point() +
  expand_limits(x = 5.2)
```
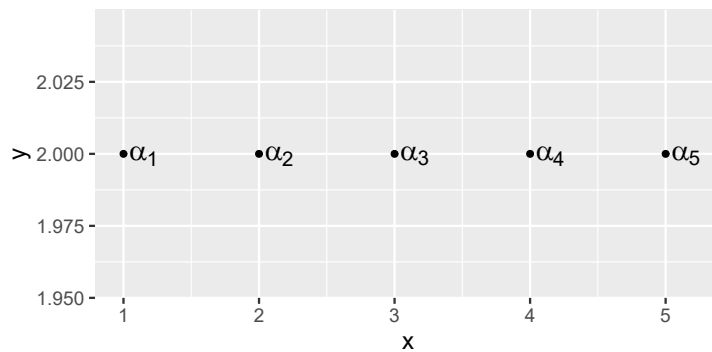
In the example above, we mapped to label the text to be parsed. It is also possible, and usually preferable, to build suitable labels on the fly within `aes()` when setting the mapping for `label`. Here we use `geom_text()` with strings to be parsed into expressions created on the fly within the call to `aes()`. The same approach can be used for regular character strings not requiring parsing.

```r
ggplot(my.data, aes(x, y, label = paste("alpha[", x, "]", sep = ""))) +
  geom_text(hjust = -0.2, parse = TRUE, size = 6) +
  geom_point()
```

As `geom_label()` obeys the same parameters as `geom_text()` except for `angle`, we briefly describe below only the additional parameters compared to `geom_text()`. We may want to alter the default width of the border line or the color used to `fill` the rectangle, or to change the "roundness" of the corners. To suppress the border line, use `label.size = 0`. Corner roundness is controlled by parameter `label.r` and the size of the margin around the text by `label.padding`.

```r
my.data <-
  data.frame(x = 1:5, y = rep(2, 5),
             label = c("one", "two", "three", "four", "five"))

ggplot(my.data, aes(x, y, label = label)) +
  geom_label(hjust = -0.2, size = 6,
             label.size = 0L,
             label.r = unit(0, "lines"),
             label.padding = unit(0.15, "lines"),
             fill = "yellow", alpha = 0.5) +
  geom_point() +
  expand_limits(x = 5.6)
```
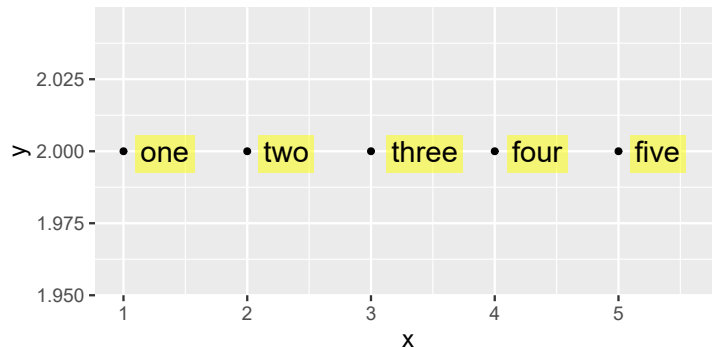
⌨ **1.15** Play with the arguments to the different parameters and with the *aesthetics* to get an idea of what can be done with them. For example, use thicker border lines and increase the padding so that a visually well-balanced margin is retained. You may also try mapping the `fill` and `color` *aesthetics* to factors in the data.

If the parameter `check_overlap` of `geom_text()` is set to TRUE, text overlap will be avoided by suppressing the text that would otherwise overlap other text. *Repulsive* versions of `geom_text()` and `geom_label()`, `geom_text_repel()` and `geom_label_repel()`, are available in package 'ggrepel'. These *geometries* avoid overlaps by automatically repositioning the text or labels. Please read the package documentation for details of how to control the repulsion strength and direction, and the properties of the segments linking the labels to the position of their data coordinates. Nearly all aesthetics supported by `geom_text()` and `geom_label()` are supported by the repulsive versions. However, given that a segment connects the label or text to its anchor point, several properties of these segments can also be controlled with aesthetics or arguments.

```
ggplot(data = mtcars,
       aes(x = disp, y = mpg, color = factor(cyl), size = wt, label = cyl)) +
  scale_size() +
  geom_point(alpha = 1/3) +
  geom_text_repel(color = "black", size = 3,
                  min.segment.length = 0.2, point.padding = 0.1)
```
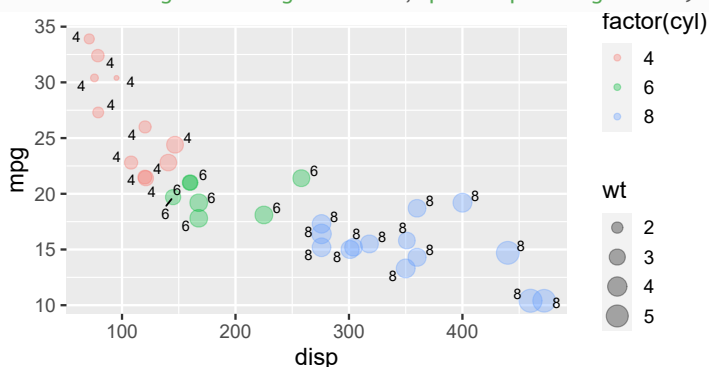

1.5.8 Plot insets

The support for insets in 'ggplot2' is confined to `annotation_custom()` which was designed to be used for static annotations expected to be the same in each panel of a plot (the use of annotations is described in section 1.10). Package 'ggpmisc' provides geoms that mimic `geom_text()` in relation to the *aesthetics* used, but that similarly to `geom_sf()`, expect that the column in `data` mapped to the `label` aesthetics are lists of objects containing multiple pieces of information, rather than atomic vectors. Three geometries are currently available: `geom_table()`, `geom_plot()` and `geom_grob()`.

⚠ Given that `geom_table()`, `geom_plot()` and `geom_grob()` will rarely use a mapping inherited from the whole plot, by default they do not inherit it. Either the

mapping should be supplied as an argument to these functions or their parameter `inherit.aes` explicitly set to TRUE.

The plotting of tables by mapping a list of data frames to the `label` *aesthetic* is done with `geom_table()`. Positioning, justification, and angle work as for `geom_text()` and are applied to the whole table. Only `tibble` objects (see documentation of package 'tibble') can contain, as variables, lists of data frames, so this *geometry* requires the use of `tibble` objects to store the data. The table(s) are created as 'grid' `grob` objects, collected in a tree and added to the `ggplot` object as a new layer.

We first generate a `tibble` containing summaries from the data, formatted as character strings, wrap this tibble in a list, and store this list as a column in another `tibble`. To accomplish this, we use functions from the 'tidyverse' described in chapter **??**.

```r
mtcars %.>%
  group_by(., cyl) %.>%
  summarize(.,
            "mean wt" = format(mean(wt), digits = 3),
            "mean disp" = format(mean(disp), digits = 2),
            "mean mpg" = format(mean(mpg), digits = 2)) -> my.table
table.tb <- tibble(x = 500, y = 35, table.inset = list(my.table))
```

```r
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          color = factor(cyl),
                          size = wt,
                          label = cyl)) +
  scale_size() +
  geom_point() +
  geom_table(data = table.tb,
             aes(x = x, y = y, label = table.inset),
             color = "black", size = 3)
```
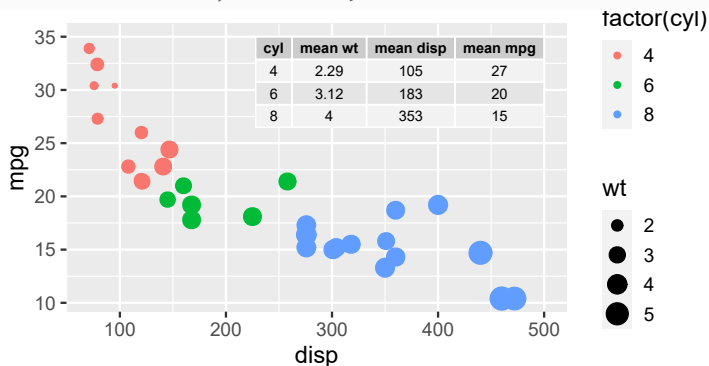

The `color` and `size` aesthetics control the text in the table(s) as a whole. It is also possible to rotate the table(s) using `angle`. As with text labels, justification is interpreted in relation to table-text orientation. We set the `y = 0` in `data.tb` and then use `vjust = 1` to position the top of the table at this coordinate value.

```r
ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(cyl))) +
  geom_point() +
  geom_table(data = table.tb,
             aes(x = x, y = y, label = table.inset),
             color = "blue", size = 3,
```

```
                          hjust = 1, vjust = 0, angle = 90)
```

Parsed text, using R's *plotmath* syntax is supported in the table, with fallback to plain text in case of parsing errors, on a cell-by-cell basis. We end this section with a simple example, which even if not very useful, demonstrates that `geom_table()` behaves like a "normal" ggplot *geometry* and that a table can be the only layer in a ggplot if desired. The addition of multiple tables with a single call to `geom_table()` by passing a `tibble` with multiple rows as an argument for `data` is also possible.

```
tb.pm <- tibble('x^0' = 1,
                'x^1' = 1:5,
                'x^2' = (1:5)^2,
                'x^3' = (1:5)^3)
data.tb <- tibble(x = 1, y = 1, table.inset = list(tb.pm))
ggplot(data.tb, mapping = aes(x, y, label = table.inset)) +
  geom_table(inherit.aes = TRUE, size = 7, parse = TRUE) +
  theme_void()
```

⌨ The *geometry* `geom_table()` uses functions from package 'gridExtra' to build a graphical object for the table. The use of table themes was not yet supported by this geometry at the time of writing.

Geometry `geom_plot()` works much like `geom_table()`, but instead of expecting a list of data frames or tibbles to be mapped to the `label` aesthetics, it expects a list of ggplots (objects of class `gg`). This allows adding as an inset to a ggplot, another ggplot. In the times when plots were hand drafted with India ink on paper, the use of inset plots was more frequent than nowadays. Inset plots can be very useful for zooming-in on parts of a main plot where observations are crowded and for displaying summaries based on the observations shown in the main plot. The inset plots are nested in viewports which control the dimensions of the inset plot, and aesthetics `vp.height` and `vp.width` control their sizes—with defaults of 1/3 of the height and width of the plotting area of the main plot. Themes can be applied separately to the main and inset plots.

In the first example of inset plots, we include one of the summaries shown above as an inset table. We first create a tibble containing the plot to be inset.

```
mtcars %.>%
  group_by(., cyl) %.>%
  summarize(., mean.mpg = mean(mpg)) %.>%
  ggplot(data = .,
         aes(factor(cyl), mean.mpg, fill = factor(cyl))) +
  scale_fill_discrete(guide = "none") +
  scale_y_continuous(name = NULL) +
    geom_col() +
    theme_bw(8) -> my.plot
plot.tb <- tibble(x = 500, y = 35, plot.inset = list(my.plot))
```

```
ggplot(data = mtcars, aes(x = disp, y = mpg,
                          color = factor(cyl))) +
  geom_point() +
  geom_plot(data = plot.tb,
            aes(x = x, y = y, label = plot.inset),
            vp.width = 1/2,
            hjust = "inward", vjust = "inward")
```
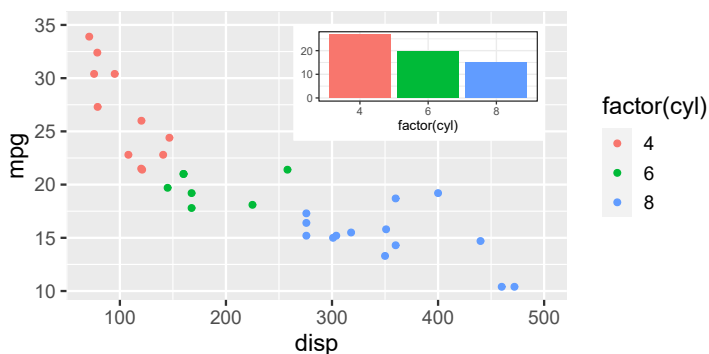
In the second example we add the zoomed version of the same plot as an inset. 1) Manually set limits to the coordinates to zoom into a region of the main plot, 2) set the *theme* of the inset, 3) remove axis labels as they are the same as in the main plot, 4) and 5) highlight the zoomed-in region in the main plot. This fairly complex example shows how a new extension to 'ggplot2' can integrate well into the grammar of graphics paradigm. In this example, to show an alternative approach, instead of collecting all the data into a data frame, we map constant values directly to the various aesthetics within `annotate()` (see section 1.10 on page 82).

```r
p.main <- ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(cyl))) +
  geom_point()
p.inset <- p.main +
  coord_cartesian(xlim = c(270, 330), ylim = c(14, 19)) +
  labs(x = NULL, y = NULL) +
  scale_color_discrete(guide = "none") +
  theme_bw(8) + theme(aspect.ratio = 1)
p.main +
  geom_plot(x = 480, y = 34, label = list(p.inset), vp.height = 1/2,
            hjust = "inward", vjust = "inward") +
  annotate(geom = "rect", fill = NA, color = "black",
           xmin = 270, xmax = 330, ymin = 14, ymax = 19,
           linetype = "dotted")
```
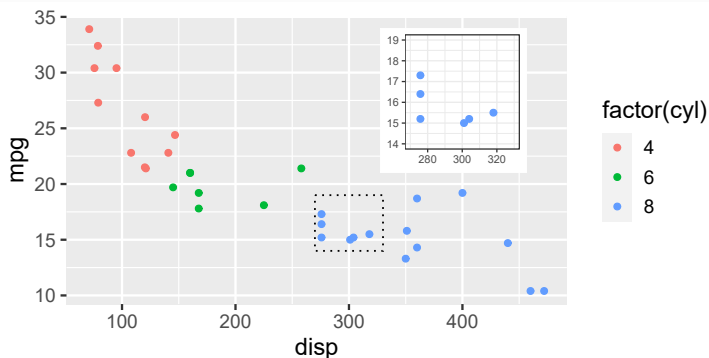

Geometry `geom_grob()` works much like `geom_table()` and `geom_plot()` but expects a list of 'grid' graphical objects, called `grob` for short. This adds generality at the expense of having to separately create the grobs either using 'grid' or by converting other objects into grobs. This geometry is as flexible as `annotation_custom()` with respect to the grobs, but behaves as a *geometry*. We

show an example that adds two bitmaps to the plot. The bitmaps are read from PNG files, converted into grobs, and added to the plot as a new layer. The PNG bitmaps used have a transparent background.

```r
file1.name <-
    system.file("extdata", "Isoquercitin.png", package = "ggpmisc", must-
Work = TRUE)
Isoquercitin <- magick::image_read(file1.name)
file2.name <-
  system.file("extdata", "Robinin.png", package = "ggpmisc", mustWork = TRUE)
Robinin <- magick::image_read(file2.name)
grob.tb <- tibble(x = c(0, 100), y = c(10, 20), height = 1/3, width = c(1/2),
                  grobs = list(grid::rasterGrob(image = Isoquercitin),
                               grid::rasterGrob(image = Robinin)))

ggplot() +
  geom_grob(data = grob.tb,
          aes(x = x, y = y, label = grobs, vp.height = height, vp.width = width),
              hjust = "inward", vjust = "inward")
```
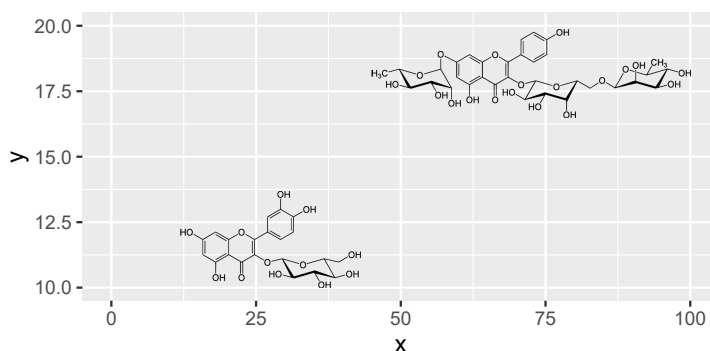

Grid graphics provide the low-level functions that both 'ggplot2' and 'lattice' use under the hood. Grid supports different types of units for expressing the co-ordinates of positions within the plotting area. All examples outside this text box use "native" data coordinates, however, coordinates can be also given in physical units like "mm". More useful when working with scalable plots is to use "npc" *normalized parent coordinates*, which are expressed as numbers in the range 0 to 1, relative to the dimensions of the sides of the current *viewport*, with origin at the lower left corner.

Package 'ggplot2' interprets x and y coordinates in "native" data coordinates, and trickery seems to be needed to get around this limitation. A rather general solution is provided by package 'ggpmisc' through *aesthetics* npcx and npcy and *geometries* that support them. At the time of writing, geom_text_npc(), geom_label_npc(), geom_table_npc(), geom_plot_npc() and geom_grob_npc(). These *geometries* are useful for annotating plots and adding insets at positions relative to the plotting area that remain always consistent across different plots, or across panels when using facets with free axis limits. Being geometries they provide freedom in the elements added to different panels and their positions.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(cyl))) +
  geom_point() +
  geom_label_npc(npcx = 0.5, npcy = 0.9, label = "a label", color = "black")
```
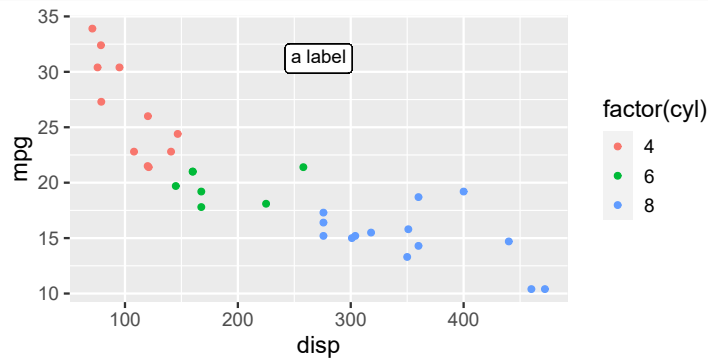

1.6   Statistics

Before learning about 'ggplot2' *statistics*, it is important to have clear how the mapping of factors to *aesthetics* works. When a factor, for example, is mapped to color, it creates a new grouping, with the observations matching a given level of the factor, corresponding to a group. Most *statistics* operate on the data for each of these groups separately, returning a summary for each group, for example, the mean of the observations in a group.

1.6.1   Functions

In addition to plotting data from a data frame with variables to map to x and y *aesthetics*, it is possible to have only a variable mapped to x and use stat_function() to compute the values to be mapped to y using an R function. This avoids the need to generate data beforehand as even the number of data points to be generated can be set in geom_function(). Any R function, user defined or not, can be used as long as it is vectorized, with the length of the returned vector equal to the length of the vector passed as first argument to it. The variable mapped to x determines the range, and the argument to parameter n of geom_function() the length of the generated vector that is passed as first argument to fun when it is called to generate the values to napped to y. These are the x and y values passed to the *geometry*.

We start with the Normal distribution function. We rely on the defaults n = 101 and geom = "path".

```
ggplot(data.frame(x = -3:3), aes(x = x)) +
  stat_function(fun = dnorm)
```
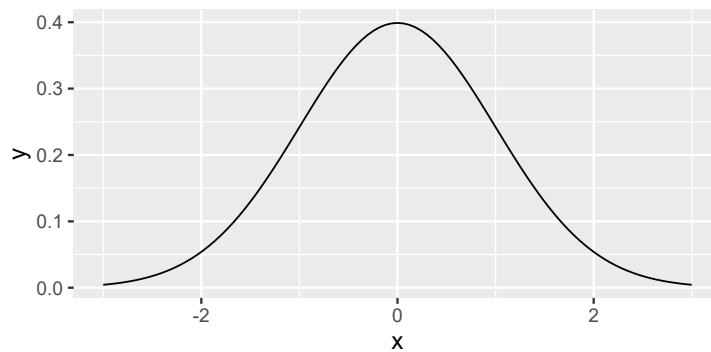
Using a list we can even pass by name additional arguments to use when the function is called.

```
ggplot(data.frame(x = -3:3), aes(x = x)) +
  stat_function(fun = dnorm, args = list(mean = 1, sd = .5))
```

⌨ **1.16** Edit the code above so as to plot in the same figure three curves, either for three different values for `mean` or for three different values for `sd`.

Named user-defined functions (not shown), and anonymous functions (below) can also be used.

```
ggplot(data.frame(x = 0:1), aes(x = x)) +
  stat_function(fun = function(x, a, b){a + b * x^2},
                args = list(a = 1, b = 1.4))
```

⌨ **1.17** Edit the code above to use a different function, such as e^{x+k}, adjusting the argument(s) passed through `args` accordingly. Do this by means of an anonymous function, and by means of an equivalent named function defined by your code.

1.6.2 Summaries

The summaries discussed in this section can be superimposed on raw data plots, or plotted on their own. Beware, that if scale limits are manually set, the summaries will be calculated from the subset of observations within these limits. Scale limits can be altered when explicitly defining a scale or by means of functions `xlim()` and `ylim()`. See section 1.11 on page 85 for an explanation of how coordinate limits can be used to zoom into a plot without excluding of x and y values from the data.

It is possible to summarize data on the fly when plotting. We describe in the same section the calculation of measures of central tendency and of variation, as `stat_summary()` allows them to be calculated simultaneously and added together with a single layer.

For use in the examples, we generate some normally distributed artificial data.

```
fake.data <- data.frame(
  y = c(rnorm(10, mean = 2, sd = 0.5),
        rnorm(10, mean = 4, sd = 0.7)),
  group = factor(c(rep("A", 10), rep("B", 10)))
  )
```

We will reuse a "base" scatter plot in a series of examples, so that the differences are easier to appreciate. We first add just the mean. In this case, we need to pass as an argument to `stat_summary()`, the geom to use, as the default one, `geom_pointrange()`, expects data for plotting error bars in addition to the mean. This example uses a hyphen character as the constant value of `shape` (see the example for `geom_point()` on page 25 on the use of digits as `shape`). Instead of passing `"mean"` as an argument to parameter `fun` (earlier called `fun.y`), we can pass, if desired, other summary functions like `"median"`. In the case of these functions that return a single computed value, we pass them, or character strings with their names, as an argument to parameter `fun`.

```
ggplot(data = fake.data, aes(y = y, x = group)) +
  geom_point(shape = 21) +
  stat_summary(fun = "mean", geom = "point",
               color = "red", shape = "-", size = 10)
```
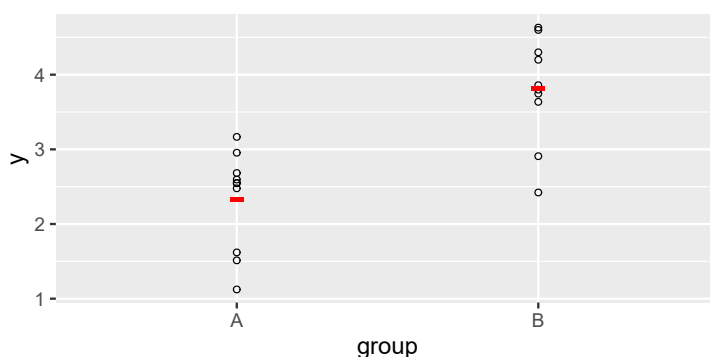

To pass as an argument a function that returns a central value like the mean plus confidence or other limits, we use parameter `fun.data` instead of `fun`. In the next example we add means and confidence intervals for $p = 0.95$ (the default) assuming normality.

```
stat_summary(fun.data = "mean_cl_normal", color = "red", size = 1, alpha = 0.7)
```

We can override the default of $p = 0.95$ for confidence intervals by setting, for example, `conf.int = 0.90` in the list of arguments passed to the function. The intervals can also be computed without assuming normality, using the empirical distribution estimated from the data by bootstrap. To achieve this we pass to `fun.data` the argument `"mean_cl_boot"` instead of `"mean_cl_normal"`.

```
stat_summary(fun.data = "mean_cl_boot",
             fun.args = list(conf.int = 0.90),
             color = "red", size = 1, alpha = 0.7)
```

For $\bar{x} \pm$ s.e. we should pass `"mean_se"` and for $\bar{x} \pm$ s.d. `"mean_sdl"`.

```
stat_summary(fun.data = "mean_se",
             color = "red", size = 1, alpha = 0.7)
```

We do not give an example here, but it is possible to use user-defined functions instead of the functions exported by package 'ggplot2' (based on those in package 'Hmisc'). Because arguments to the function used, except for the first one containing the variable in `data` mapped to the y aesthetic, are supplied as a named

list through parameter `fun.args`, the names used for parameters in the function definition need only match the names in this list.

Finally, we plot the means in a scatter plot, with the observations superimposed on the error bars as a result of the order in which the layers are added to the plot. In this case, we set `fill`, `color` and `alpha` (transparency) to constants, but in more complex data sets, mapping them to factors in `data` can be used for grouping of observations. Here, adding two plot layers with `stat_summary()` allows us to plot the mean and the error bars using different colors.

```
ggplot(data = fake.data, aes(y = y, x = group)) +
  stat_summary(fun = "mean", geom = "point",
               fill = "white", color = "black") +
  stat_summary(fun.data = "mean_cl_boot",
               geom = "errorbar",
               width = 0.1, size = 1, color = "red") +
  geom_point(size = 3, alpha = 0.3)
```

We can plot means, or other summaries, by group mapped to `x` (`class` in this example) as columns by passing `"col"` as an argument to `geom`. In this way we avoid the need to compute the summaries in advance.

```
ggplot(mpg, aes(class, hwy)) +
  stat_summary(geom = "col", fun = mean)
```
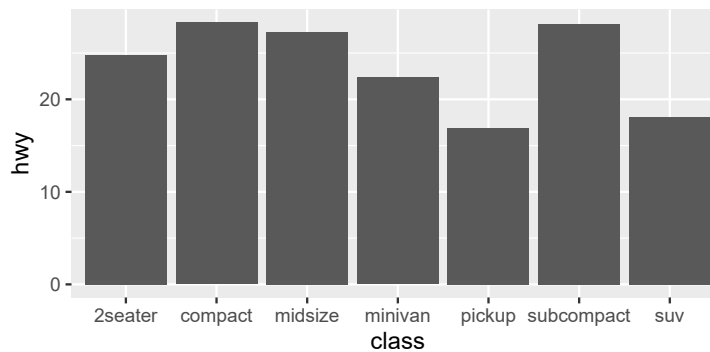

We can easily add error bars to the column plot. We use `size` to make the lines of the error bars thicker. The default *geometry* in `stat_summary()` is `geom_pointrange()`, so we can pass `"linerange"` as an argument for `geom` to eliminate the point.

```
  stat_summary(geom = "linerange", fun.data = "mean_se",
               size = 1, color = "red")
```

Passing `"errorbar"` instead of `"linerange"` to `geom` results in traditional "capped" error bars. However, this type of error bar has been criticized as adding unnecessary clutter to plots (Tufte 1983). We can use `width` to reduce the width of the caps at the ends of the error bars.

If we have already calculated values for the summaries, we can still obtain the same plots by mapping variables to the *aesthetics* required by `geom_errorbar()` and `geom_linerange()`: `x`, `y`, `ymax` and `ymin`.

⊟   The "reverse" syntax is also valid, as we can add the *geometry* to the plot object and pass the *statistics* as an argument to it. In general in this book we avoid this alternative syntax for the sake of consistency.

```
ggplot(mpg, aes(class, hwy)) +
  geom_col(stat = "summary", fun = mean)
```

1.6.3 Smoothers and models

For describing or highlighting relationships between pairs of continuous variables, using a line, straight or curved, in a plot is very effective. To draw lines that provide a meaningful and accurate description of the relationship, we fit models to observations and base the plotted line on model predictions. Being this a statistical procedure and observations subject to uncontrolled variation, we can also assess the reliability of the estimation. See section **??** on page **??** for a description of the model fitting procedures underlying the plotting described in the current section.

The *statistic* `stat_smooth()` fits a smooth curve to observations in the case when the scales for x and y are continuous—the corresponding *geometry* `geom_smooth()` uses this *statistic*, and differs only in how arguments are passed to formal parameters. For the first example, we use `stat_smooth()` with the default smoother, a spline. The type of smoother is automatically chosen based on the number of observations and informed by a message. The `formula` must be stated using the names of the x and y aesthetics, rather than the original names of the mapped variables in the `mtcars` data frame. Splines are described in section **??** on page **??**.

```
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
      stat_smooth(formula = y ~ x)
```

In most cases we will want to plot the observations as points together with the smoother. We can plot the observation on top of the smoother, as done here, or the smoother on top of the observations.

```
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  stat_smooth(formula = y ~ x) +
  geom_point()
```

```
## `geom_smooth()` using method = 'loess'
```
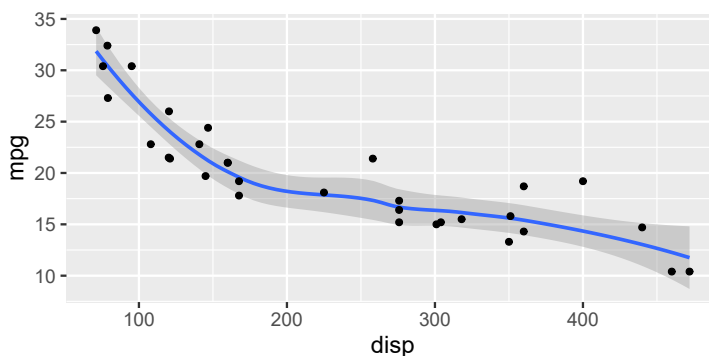

Instead of using the default spline, we can fit a different model. In this example we use a linear model as smoother, fitted by `lm()`. Model fitting is explained in section **??** on page **??**.

```
  stat_smooth(method = "lm", formula = y ~ x) +
```

These data are really grouped, so we map variable `cyl` to the `color` *aesthetic*.

Now we get three groups of points with different colours but also three separate smooth lines.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(cyl))) +
  stat_smooth(method = "lm", formula = y ~ x) +
  geom_point()
```
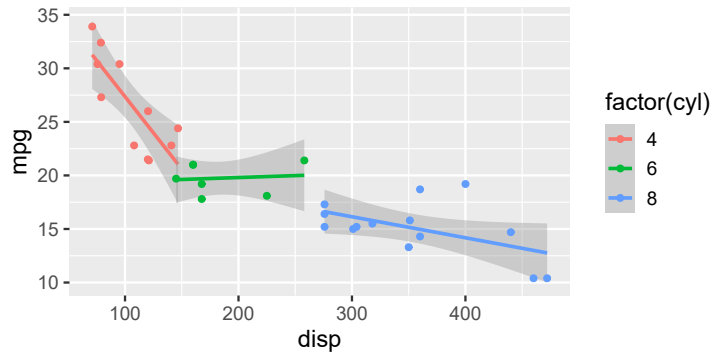

To obtain a single smoother for the three groups, we need to set the mapping of the `color` *aesthetic* to a constant within `stat_smooth()`. This local value overrides the default `color` mapping set in `ggplot()` just for this plot layer. We use `"black"` but this could be replaced by any other color definition known to R.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(cyl))) +
  stat_smooth(method = "lm", formula = y ~ x, color = "black") +
  geom_point()
```

Instead of using the `formula` for a linear regression as smoother, we pass a different `formula` as an argument. In this example we use a polynomial of order 2.

```
ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(cyl))) +
  stat_smooth(method = "lm", formula = y ~ poly(x, 2), color = "black") +
  geom_point()
```
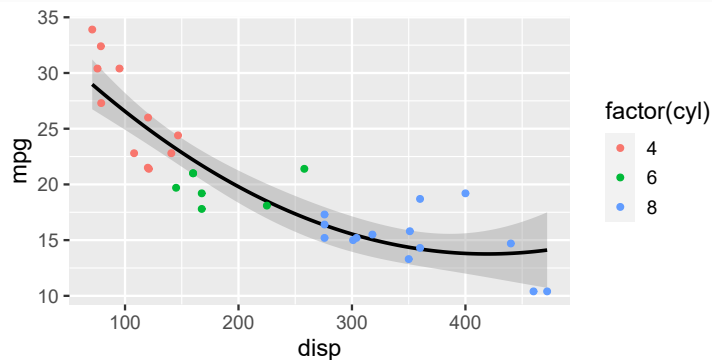

It is possible to use other types of models, including GAM and GLM, as smoothers, but we will give only two simple examples of the use of `nls()` to fit a model non-linear in its parameters (see section ?? on page ?? for details about fitting this same model with `nls()`). In the first one we fit a Michaelis-Menten equation to reaction rate (`rate`) versus reactant concentration (`conc`). `Puromycin` is a data set included in the R distribution. Function `SSmicmen()` is also from R, and is a *self-starting* implementation of the Michaelis-Menten equation. Thanks to this, even though the fit is done with an iterative algorithm, we do not need to explicitly

provide starting values for the parameters to be fitted. We need to set `se = FALSE` because standard errors are not supported by the `predict()` method for `nls` fitted models.

```r
ggplot(Puromycin, aes(conc, rate, color = state)) +
  geom_point() +
  geom_smooth(method = "nls",
              formula =  y ~ SSmicmen(x, Vm, K),
              se = FALSE)
```

In the second example we define the same model directly in the model formula, and provide the starting values explicitly. The names used for the parameters to be fitted can be chosen at will, within the restrictions of the R language, but of course the names used in `formula` and `start` must match each other.

```r
ggplot(Puromycin, aes(conc, rate, color = state)) +
  geom_point() +
  geom_smooth(method = "nls",
              method.args = list(formula =  y ~ (Vmax * x) / (k + x),
                                 start = list(Vmax = 200, k = 0.05)),
              se = FALSE)
```

In some cases it is desirable to annotate plots with fitted model equations or fitted parameters. One way of achieving this is by fitting the model and then extracting the parameters to manually construct text strings to use for text or label annotations. However, package 'ggpmisc' makes it possible to automate such annotations in many cases. This package also provides `stat_poly_line()` which is similar to `stat_smooth()` but with `method = "lm"` consistently as its default irrespective of the number of observations.

```r
my.formula <- y ~ poly(x, 2)
ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(cyl))) +
  stat_poly_line(formula = my.formula, color = "black") +
  stat_poly_eq(formula = my.formula, mapping = use_label(c("eq", "F")),
               color = "black", parse = TRUE, label.x.npc = 0.3) +
  geom_point()

## Warning: The `scale_name` argument of `continuous_scale()` is deprecated as
## of ggplot2
## 3.5.0.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```
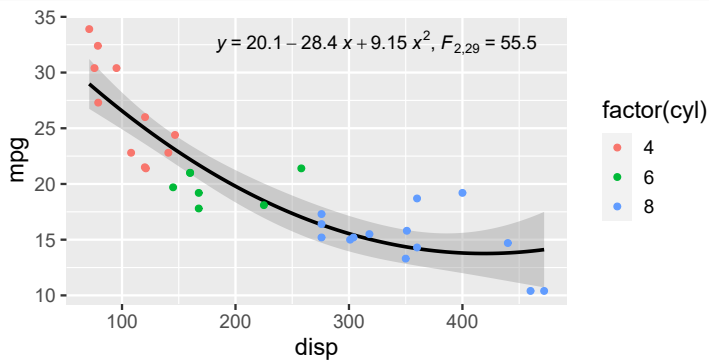
This same package makes it possible to annotate plots with summary tables from a model fit.

```r
my.formula <- y ~ poly(x, 2)
ggplot(data = mtcars, aes(x = disp, y = mpg, color = factor(cyl))) +
  stat_poly_line(formula = my.formula, color = "black") +
  stat_fit_tb(method.args = list(formula = my.formula),
              color = "black",
              tb.vars = c(Parameter = "term",
                          Estimate = "estimate",
                          "s.e." = "std.error",
                          "italic(t)" = "statistic",
                          "italic(P)" = "p.value"),
              label.y.npc = "top", label.x.npc = "right",
              parse = TRUE) +
  geom_point()
```
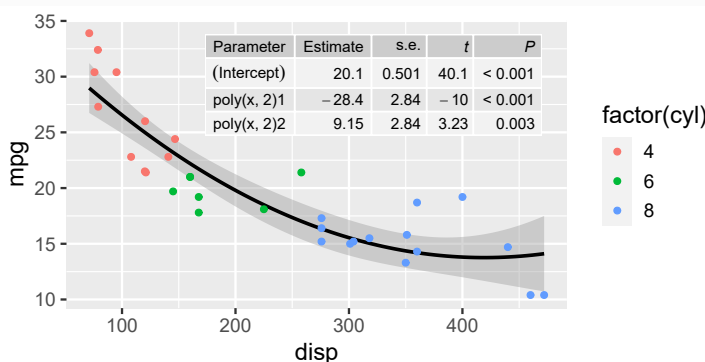

Package 'ggpmisc' provides additional *statistics* for the annotation of plots based on fitted models supported by package 'broom' and its extensions. It also supports lines and equations for quantile regression and major axis regression. Please see the package documentation for details.

1.6.4 Frequencies and counts

When the number of observations is rather small, we can rely on the density of graphical elements to convey the density of the observations. For example, scatter plots using well-chosen values for alpha can give a satisfactory impression of the density. Rug plots, described in section 1.5.2 on page 29, can also satisfactorily convey the density of observations along x and/or y axes. Such approaches do not involve computations, while the *statistics* described in this section do. Frequencies by value-range (or bins) and empirical density functions are summaries especially useful when the number of observations is large. These summaries can be computed in one or more dimensions.

Histograms are defined by how the plotted values are calculated. Although histograms are most frequently plotted as bar plots, many bar or "column" plots are not histograms. Although rarely done in practice, a histogram could be plotted using a different *geometry* using stat_bin(), the *statistic* used by default by geom_histogram(). This *statistic* does binning of observations before computing frequencies, and is suitable for continuous x scales. When a factor is mapped to x, stat_count() should be used, which is the default stat for geom_bar(). These two

geometries are described in this section about statistics, because they default to using statistics different from `stat_identity()` and consequently summarize the data.

As before, we generate suitable artificial data.

```r
set.seed(12345)
my.data <-
  data.frame(x = rnorm(200),
             y = c(rnorm(100, -1, 1), rnorm(100, 1, 1)),
             group = factor(rep(c("A", "B"), c(100, 100))) )
```

We could have relied on the default number of bins automatically computed by the `stat_bin()` statistic, however, we here set it to 15 with `bins = 15`. It is important to remember that in this case no variable in `data` is mapped onto the `y` *aesthetic*.

```r
ggplot(my.data, aes(x)) +
  geom_histogram(bins = 15)
```
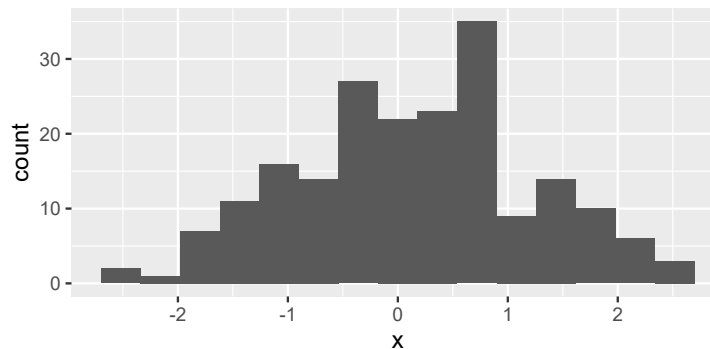

If we create a grouping by mapping a factor to an additional *aesthetic* how the bars created are positioned with respect to each other becomes relevant. We can then plot side by side with `position = "dodge"`, stacked one above the other with `position = "stack"` and overlapping with `position = "identity"` in which case we need to make them semi-transparent with `alpha = 0.5` so that they all remain visible.

```r
ggplot(my.data, aes(y, fill = group)) +
  geom_histogram(bins = 15, position = "dodge")
```

The computed values are contained in the `data` that the *geometry* "receives" from the *statistic*. Many statistics compute additional values that are not mapped by default. These can be mapped with `aes()` by enclosing them in a call to `stat()`. From the help page we can learn that in addition to counts in variable `count`, density is returned in variable `density` by this statistic. Consequently, we can create a histogram with the counts per bin expressed as densities whose integral is one (rather than their sum, as the width of the bins is in this case different from one), as follows.

```r
ggplot(my.data, aes(y, fill = group)) +
  geom_histogram(mapping = aes(y = after_stat(density)), bins = 15, posi-
tion = "dodge")
```
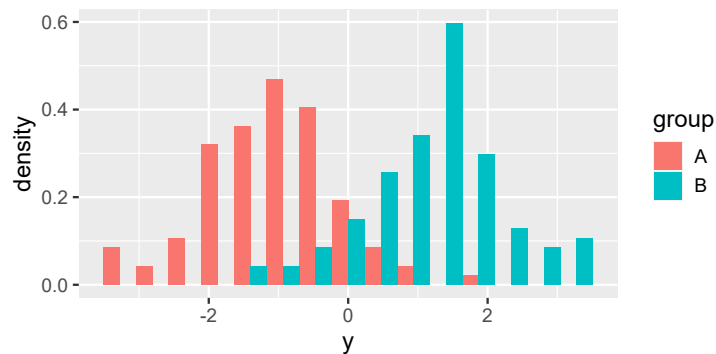
If it were not for the easier to remember name of `geom_histogram()`, adding the layers with `stat_bin()` or `stat_count()` would be preferable as it makes clear that computations on the data are involved.

```
ggplot(my.data, aes(y, fill = group)) +
  stat_bin(bins = 15, position = "dodge")
```

The *statistic* `stat_bin2d()`, and its matching *geometry* `geom_bin2d()`, by default compute a frequency histogram in two dimensions, along the x and y *aesthetics*. The frequency for each rectangular tile is mapped onto a `fill` scale. As for `stat_bin()`, `density` is also computed and available to be mapped as shown above for `geom_histogram`. In this example, to compare dispersion in two dimensions, equal x and y scales are most suitable, which we achieve by adding `coord_fixed()`, which is a variation of the default `coord_cartesian()` (see section 1.11 on page 85 for details on other systems of coordinates).

```
ggplot(my.data, aes(x, y)) +
  stat_bin2d(bins = 8) +
  coord_fixed(ratio = 1)
```
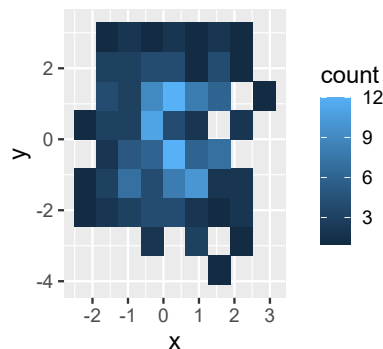

The *statistic* `stat_bin_hex()`, and its matching *geometry* `geom_hex()`, differ from `stat_bin2d()` in their use of hexagonal instead of square tiles. By default the frequency or `count` for each hexagon is mapped to the `fill` aesthetic, but counts expressed as `density` are also computed and can be mapped with `aes(fill = after_stat(density))`.

```
ggplot(my.data, aes(x, y)) +
  stat_bin_hex(bins = 8) +
  coord_fixed(ratio = 1)
```
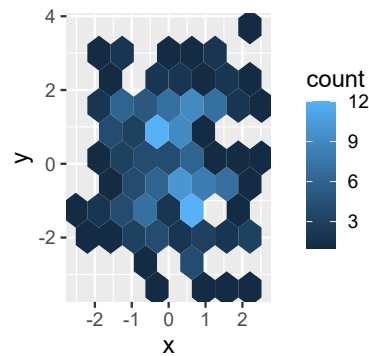
1.6.5 Density functions

Empirical density functions are the equivalent of a histogram, but are continuous and not calculated using bins. They can be estimated in 1 or 2 dimensions (1D or 2D), for x or x and y, respectively. As with histograms it is possible to use different *geometries* to visualize them. Examples of the use of `geom_density()` to create 1D density plots follow.

```
ggplot(my.data, aes(y, color = group)) +
  geom_density()
```
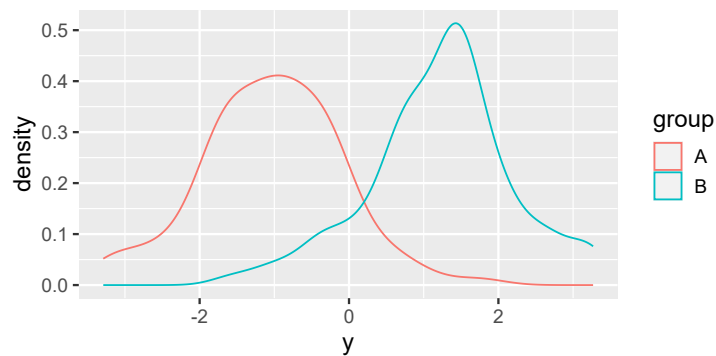

A semitransparent fill can be used instead of coloured lines.

```
ggplot(my.data, aes(y, fill = group)) +
  geom_density(alpha = 0.5)
```

Examples of 2D density plots follow. In the first example we use two *geometries* which were earlier described, `geom_point()` and `geom_rug()`, to plot the observations in the background. With `stat_density_2d()` we add a two-dimensional density "map" represented using isolines. We map `group` to the `color` *aesthetic*.

```
ggplot(my.data, aes(x, y, color = group)) +
  geom_point() +
  geom_rug() +
  stat_density_2d()
```
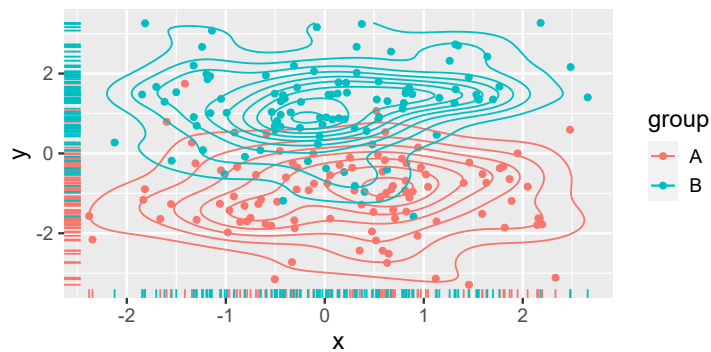
In this case, `geom_density_2d()` is equivalent, and we can replace it in the last line in the chunk above.

```
geom_density_2d()
```

In the next example we plot the groups in separate panels, and use a *geometry* supporting the `fill` *aesthetic* and we map to it the variable `level`, computed by `stat_density_2d()`

```
ggplot(my.data, aes(x, y)) +
stat_density_2d(aes(fill = after_stat(level)), geom = "polygon") +
  facet_wrap(~group)
```
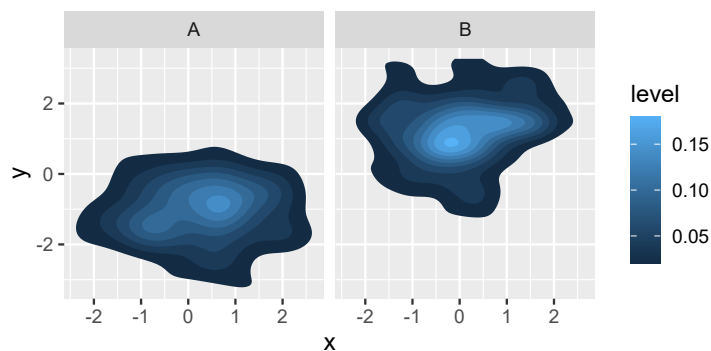

1.6.6 Box and whiskers plots

Box and whiskers plots, also very frequently called just box plots, are also summaries that convey some of the properties of a distribution. They are calculated and plotted by means of `stat_boxplot()` or its matching `geom_boxplot()`. Although they can be calculated and plotted based on just a few observations, they are not useful unless each box plot is based on more than 10 to 15 observations.

```
ggplot(my.data, aes(group, y)) +
  stat_boxplot()
```
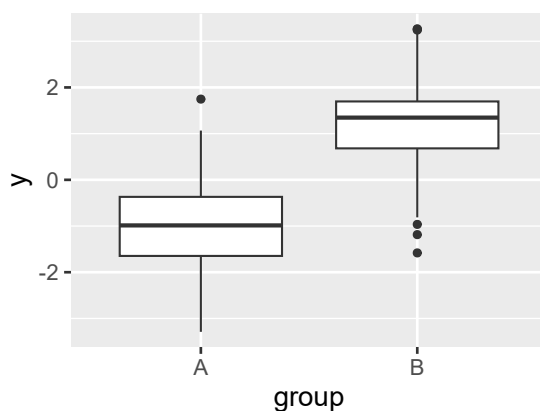
As with other *statistics*, their appearance obeys both the usual *aesthetics* such as `color`, and parameters specific to this type of visual representation: `outlier.color`, `outlier.fill`, `outlier.shape`, `outlier.size`, `outlier.stroke` and `outlier.alpha`, which affect the outliers in a way similar to the equivalent `aethetics` in `geom_point()`. The shape and width of the "box" can be adjusted with `notch`, `notchwidth` and `varwidth`. Notches in a boxplot serve a similar role for comparing medians as confidence limits serve when comparing means.

```
ggplot(my.data, aes(group, y)) +
  stat_boxplot(notch = TRUE, width = 0.4,
               outlier.color = "red", outlier.shape = "*", outlier.size = 5)
```
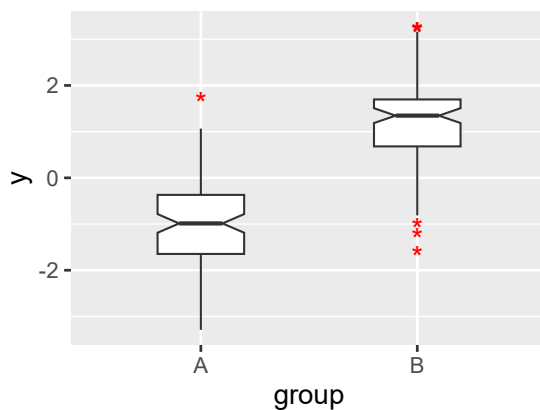

1.6.7   Violin plots

Violin plots are a more recent development than box plots, and usable with relatively large numbers of observations. They could be thought of as being a sort of hybrid between an empirical density function (see section 1.6.5 on page 54) and a box plot (see section 1.6.6 on page 55). As is the case with box plots, they are particularly useful when comparing distributions of related data, side by side. They can be created with `geom_violin()` as shown in the examples below.

```
ggplot(my.data, aes(group, y)) +
  geom_violin()
```

```
ggplot(my.data, aes(group, y, fill = group)) +
  geom_violin(alpha = 0.16) +
  geom_point(alpha = 0.33, size = 1.5,
             color = "black", shape = 21)
```
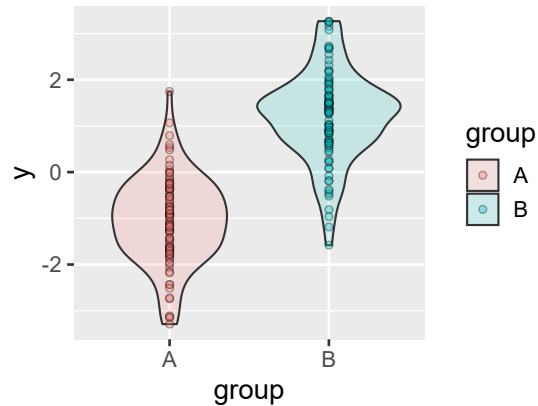

As with other *geometries*, their appearance obeys both the usual *aesthetics* such as color, and others specific to these types of visual representation.

Other types of displays related to violin plots are *beeswarm* plots and *sina* plots, and can be produced with *geometries* defined in packages 'ggbeeswarm' and 'ggforce', respectively. A minimal example of a beeswarm plot is shown below. See the documentation of the packages for details about the many options in their use.

```
ggplot(my.data, aes(group, y)) +
  geom_quasirandom()
```
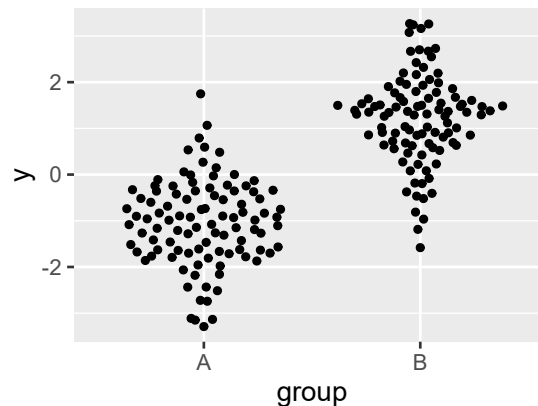

1.7   Flipped plot layers

Although it is the norm to design plots so that the independent variable is on the x axis, i.e., mapped to the x aesthetic, there are situations where swapping the roles of x and y is useful. In 'ggplot2' this is described as *flipping the orientation* of a plot. In the present section I exemplify both cases where the flipping is automatic

and where it requires user intervention. Some geometries like `geom_point()` are symmetric on the *x* and *y* aesthetics, but others like `geom_line()` operate differently on *x* and *y*. This is also the cases for almost all *statistics*.

'`ggplot2`' version 3.3.5, supports flipping in most geometries and statistics where it is meaningful, using a new syntax. This new approach is different to the flip of the coordinate system, and similar to that implemented by package '`ggstance`'. However, instead of defining new horizontal layer functions as in '`ggstance`', now the orientation of many layer functions from '`ggplot2`' can be changed by the user. This has made '`ggstance`' nearly redundant and the coding of flipped plots easier and more intuitive. Although '`ggplot2`' has offered `coord_flip()` for a long time, this affects the whole plot rather than individual layers.

When a factor is mapped to *x* or *y* flipping is automatic. A factor creates groups and summaries are computed per group, i.e., per level of the factor irrespective of the factor being mapped to the *x* or *y* aesthetic. Dodging and jitter do not need any special syntax as it was the case with package '`ggstance`'.

There are also cases that require user intervention. For example, flipping must be requested manually if both *x* and *y* are mapped to continuous variables. This is, for example, the case with `stat_smooth()` and a fit of *x* on *y*.

📭 In ggplot statistics, passing orientation = "y" results in flipping, that is applying the calculations after swapping the mappings of the *x* and *y* aesthetics. After applying the calculations the mappings of the *x* and *y* aesthetics are swapped again (diagram below).
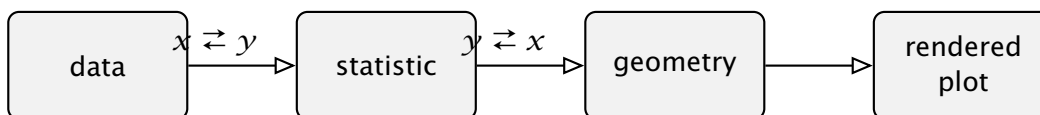
In geometries, passing orientation = "y" results in flipping of the aesthetics but with a twist. For example, in `geom_line()`, flipping changes the drawing of the lines. Normally observations are sorted along the *x* axis for drawing the segments connecting them. If we flip this layer, observations are sorted along the *y* axis before drawing the connecting segments, which can make a major difference. The variables shown on each axis remain the same, as does the position of points drawn with `geom_point()`. In this example only two segments are the same in the flipped plot and the not-flipped one.

```r
ggplot(mtcars[1:8, ], aes(x = hp, y = mpg)) +
    geom_point() +
    geom_line(alpha = 0.5) +
    geom_line(alpha = 0.5, orientation = "y", colour = "red")
```
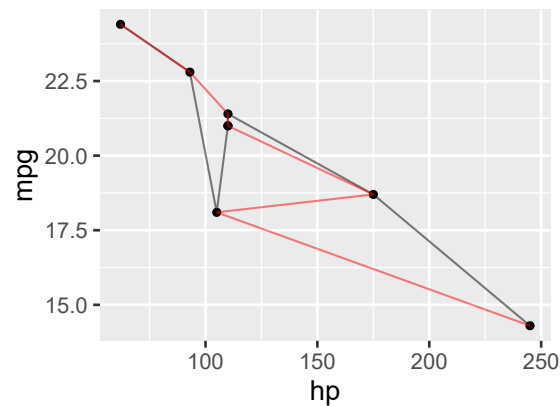
The next pair of examples exemplify automatic flipping using `stat_boxplot()`. Here we map the factor `Species` first to x and then to y. In both cases boxplots have been computed and plotted for each level of the factor. Statistics `stat_boxplot()`, `stat_summary()`, `stat_histogram()` and `stat_density()` behave similarly with respect to flipping.

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) +
    stat_boxplot()
```
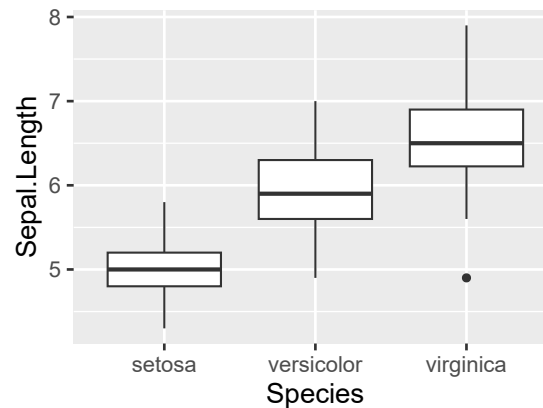


```
ggplot(iris, aes(x = Sepal.Length, y = Species)) +
    stat_boxplot()
```
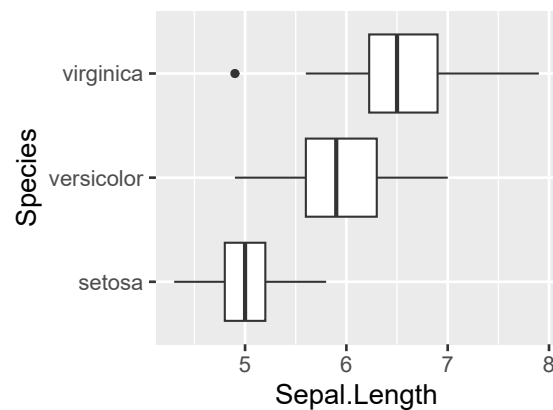
When we map a variable to only one of x or y the flip is also automatic.

```r
ggplot(iris, aes(x = Sepal.Length, color = Species)) +
    stat_density(fill = NA)
```
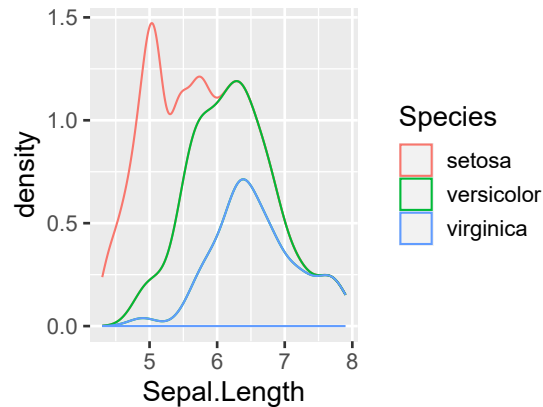


```r
ggplot(iris, aes(y = Sepal.Length, color = Species)) +
    stat_density(fill = NA)
```
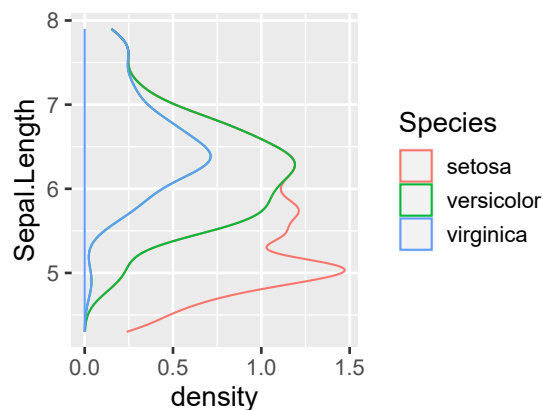

> 🖳 In the case of ordinary least squares (OLS), regressions of y on x and of x on y in most cases yield different fitted lines, even if R^2 is consistent. This is due to the assumption that x values are known, either set or measured without error, i.e., not subject to random variation. All unexplained variation in the data is assumed to be in y. See Chapter **??** on page **??** or consult a Statistics book such as *Modern Statistics for Modern Biology* (Holmes and Huber 2019, pp. 168–170) for additional information.

With two continuous variables mapped, the default is to take x as independent and y as dependent. This matters, of course, when computations as in model fitting treat x and y differently. In this case parameter `orientation` can be used to indicate which of x or y is the independent or explanatory variable.

```r
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
  stat_smooth(method = "lm", formula = y ~ x) +
  geom_point() +
  facet_wrap(~Species, scales = "free")
```
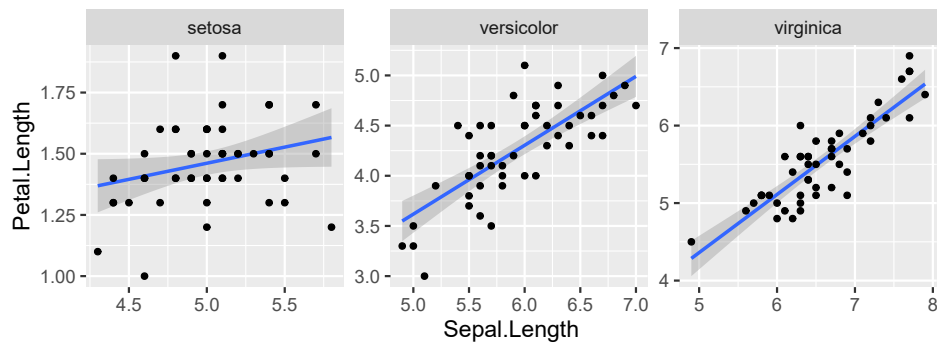
With `orientation = "y"` we tell that y is the independent variable. In the case of `geom_smooth()` this means implicitly swapping x and y in `formula`.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    stat_smooth(method = "lm", formula = y ~ x, orientation = "y") +
    geom_point() +
    facet_wrap(~Species, scales = "free")
```
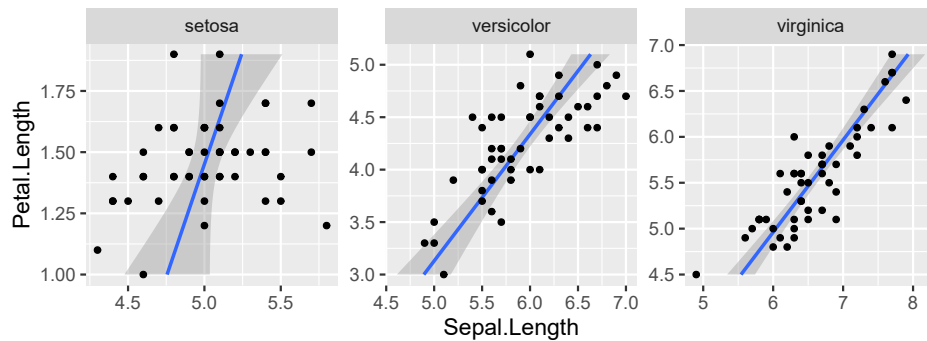

Flipping the orientation of plot layers with `orientation = "y"` is not equivalent to flipping the whole plot with `coord_flip()`. In the first case which axis is considered independent for computation changes but not the positions of the axes in the plot, while in the second case the position of the x and y axes in the plot is swapped. So, when coordinates are flipped the x aesthetic is plotted on the vertical axis and the y aesthetic on the horizontal axis, but the role of the variable mapped to the x aesthetic remains as explanatory variable.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    stat_smooth(method = "lm", formula = y ~ x) +
    geom_point() +
    coord_flip() +
    facet_wrap(~Species, scales = "free")
```
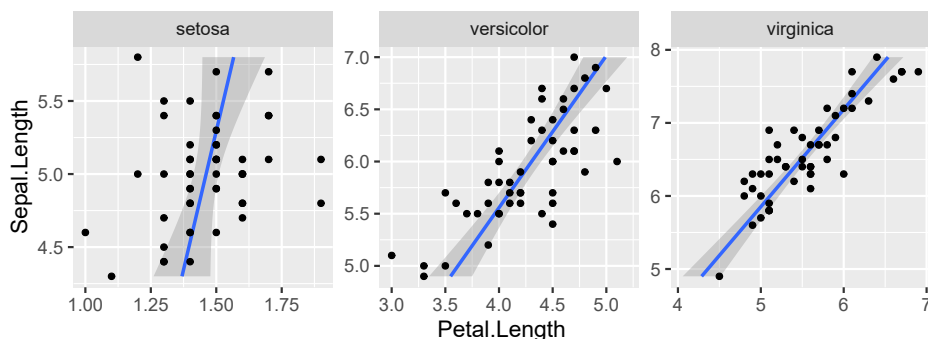
In package 'ggpmisc' (version $\geq 0.4.1$) statistics related to model fitting have an `orientation` parameter as those from package 'ggplot2' do, but in addition they accept formulas where x is on the lhs and y on the rhs, such as `formula = x ~ y` providing a syntax consistent with R's model fitting functions. In the next pair of examples we use `stat_poly_line()`. In the first example in this pair, the default `formula = y ~ x` is used, while in the second example we pass explicitly `formula = x ~ y` to force the flipping of the fitted model. To make the difference clear, we plot both linear regressions on the same plots.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    stat_poly_line() +
    stat_poly_line(formula = x ~ y, color = "red", fill = "yellow") +
    geom_point() +
    facet_wrap(~Species, scales = "free")
```
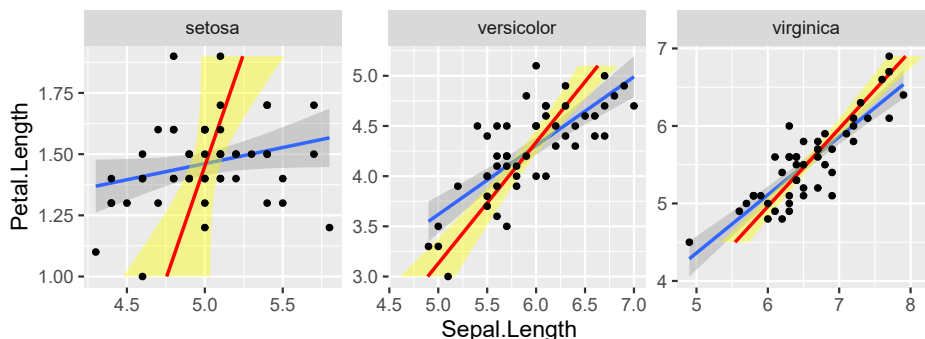

In the case of the `iris` data used for these examples, both approaches used above to linear regression are wrong. The variables mapped to x and y are correlated but both are measured with error and subject to biological variation. In this case the correct approach is to not assume that there is a variable that can be considered independent, and instead use a method like major axis (MA) regression, as can be seen below.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    stat_ma_line() +
    geom_point() +
    facet_wrap(~Species, scales = "free")
```
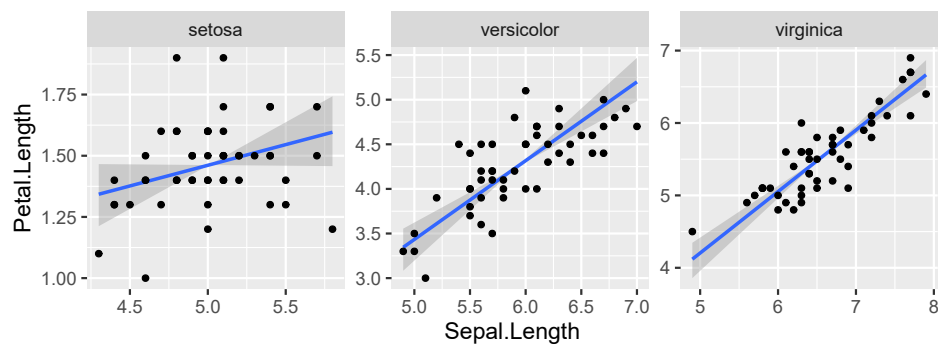
A related problem is when we need to summarize in the same plot layer x and y values. A simple example is adding a point with coordinates given by the means along the x and y axes as we need to pass these computed means simultaneously to `geom_point()`. Package 'ggplot2' provides `stat_density_2d()` and `stat_summary_2d()`. However, `stat_summary_2d()` uses bins, and is similar to `stat_density_2d()` in how the computed values are returned. Package 'ggpmisc' provides two dimensional equivalents of `stat_summary()`: `stat_centroid()`, which applies the same summary function along x and y, and `stat_summary_xy()`, which accepts one function for x and one for y.

```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    geom_point() +
    stat_centroid(color = "red") +
    facet_wrap(~Species, scales = "free")
```
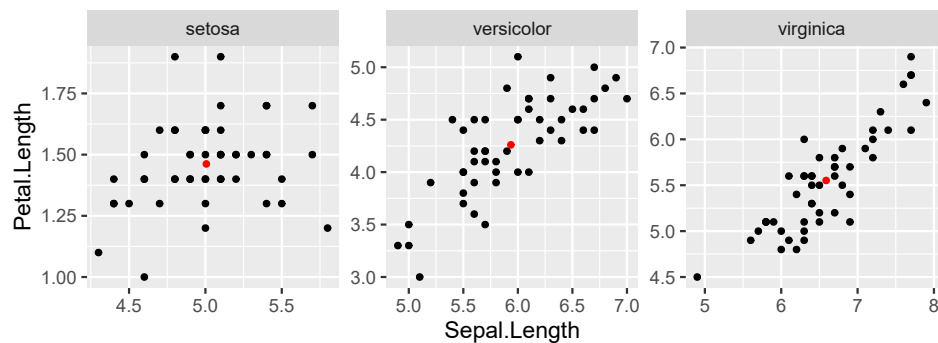


```
ggplot(iris, aes(Sepal.Length, Petal.Length)) +
    geom_point() +
    stat_centroid(geom = "rug", sides = "trbl",
                  color = "red", linewidth = 1.5) +
    facet_wrap(~Species, scales = "free")
```
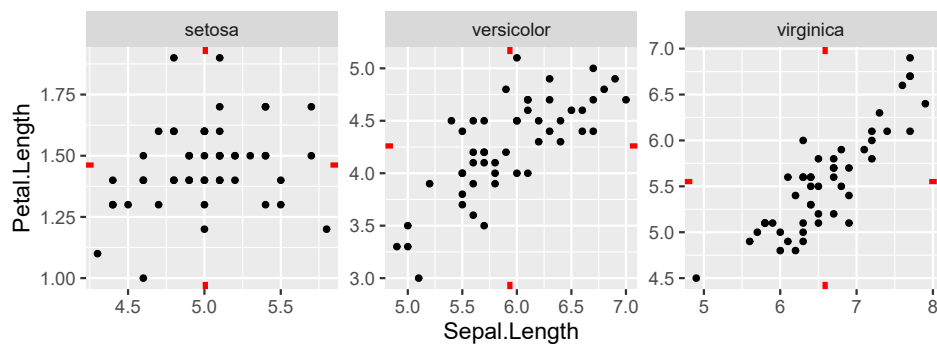
☐ **1.18** Which of the plots in the last two chunks above can be created by adding two layers with `stat_summary()`? Recreate this plot using `stat_summary()`.

1.8   Facets

Facets are used in a special kind of plots containing multiple panels in which the panels share some properties. These sets of coordinated panels are a useful tool for visualizing complex data. These plots became popular through the `trellis` graphs in S, and the 'lattice' package in R. The basic idea is to have rows and/or columns of plots with common scales, all plots showing values for the same response variable. This is useful when there are multiple classification factors in a data set. Similar-looking plots, but with free scales or with the same scale but a 'floating' intercept, are sometimes also useful. In 'ggplot2' there are two possible types of facets: facets organized in a grid, and facets along a single 'axis' of variation but, possibly, wrapped into two or more rows. These are produced by adding `facet_grid()` or `facet_wrap()`, respectively. In the examples below we use `geom_point()` but faceting can be used with `ggplot` objects containing diverse kinds of layers, displaying either observations or summaries from `data`.

We start by creating and saving a single-panel plot that we will use through this section to demonstrate how the same plot changes when we add facets.

```
p <- ggplot(data = mtcars, aes(wt, mpg)) +
  geom_point()
p
```
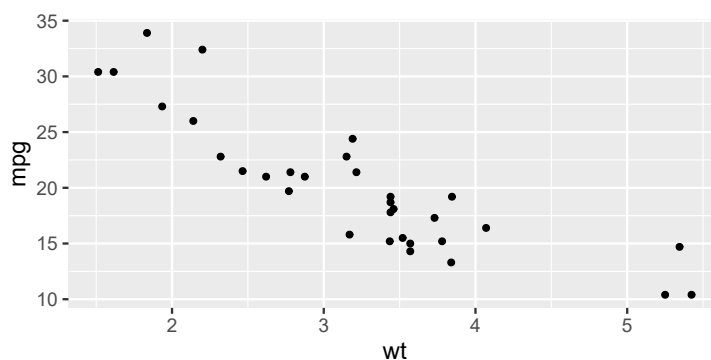
A grid of panels has two dimensions, rows and cols. These dimensions in the grid of plot panels can be "mapped" to factors. Until recently a formula syntax was the only available one. Although this notation has been retained, the preferred syntax is currently to use the parameters rows and cols. We use cols in this example. Note that we need to use vars() to enclose the names of the variables in the data. The "headings" of the panels or *strip labels* are by default the levels of the factors.

```
p + facet_grid(cols = vars(cyl))
```
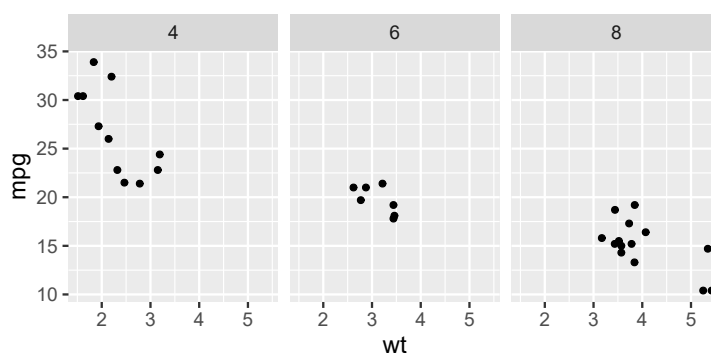

In the "historical notation" the same plot would have been coded as follows.

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

By default, all panels share the same scale limits and share the plotting space evenly, but these defaults can be overridden.

```
p + facet_grid(cols = vars(cyl), scales = "free")
p + facet_grid(cols = vars(cyl), scales = "free", space = "free")
```

To obtain a 2D grid we need to specify both rows and cols.

```
p + facet_grid(rows = vars(vs), cols = vars(am))
```

Margins display an additional column or row of panels with the combined data.

```
p + facet_grid(cols = vars(cyl), margins = TRUE)
```
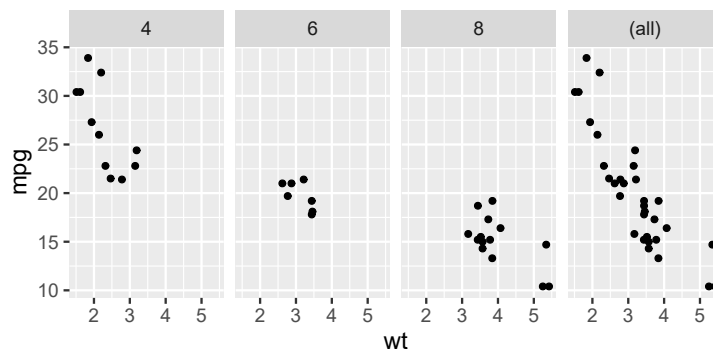
We can represent more than one variable per dimension of the grid of plot panels. For this example, we also override the default `labeller` used for the panels with one that includes the name of the variable in addition to factor levels in the *strip labels*.

```
p + facet_grid(cols = vars(vs, am), labeller = label_both)
```
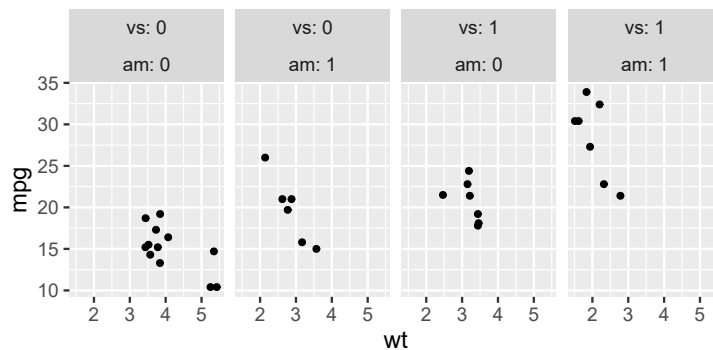

☕ Sometimes we may want to have mathematical expressions or Greek letters in the panel headings. The next example shows a way of achieving this. The key is to use as `labeller` a function that parses character strings into R expressions.

```
mtcars$cyl12 <- factor(mtcars$cyl,
                       labels = c("alpha", "beta", "sqrt(x, y)"))
p1 <- ggplot(data = mtcars, aes(mpg, wt)) +
      geom_point() +
      facet_grid(cols = vars(cyl12), labeller = label_parsed)
```

More frequently we may need to include the levels of the factor used in the faceting as part of the labels. Here we use as `labeller`, function `label_bquote()` with a special syntax that allows us to use an expression where replacement based on the facet (panel) data takes place. See section 1.14 for an example of the use of `bquote()`, the R function on which `label_bquote()`, is built.

```
p +
  facet_grid(cols = vars(cyl),
             labeller = label_bquote(cols = .(cyl)~"cylinders"))
```

In the next example we create a plot with wrapped facets. In this case the number of levels is small, and no wrapping takes place by default. In cases when more panels are present, wrapping into two or more continuation rows is the default.

Here, we force wrapping with `nrow = 2`. When using `facet_wrap()` there is only one dimension, and the parameter is called `facets`, instead of `rows` or `cols`.

```
p + facet_wrap(facets = vars(cyl), nrow = 2)
```
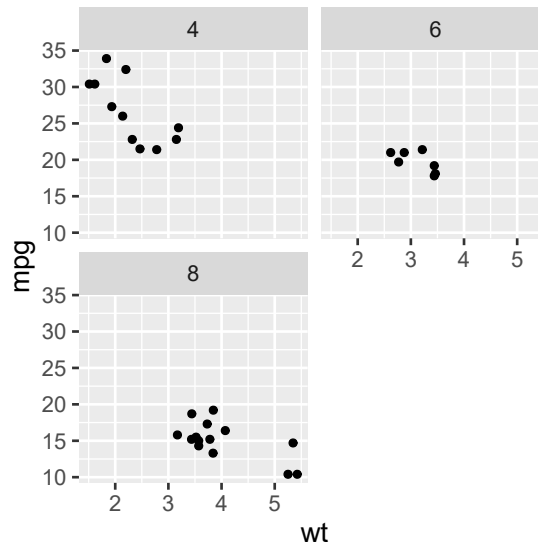

The example below (plot not shown), is similar to the earlier one for `facet_grid`, but faceting according to two factors with `facet_wrap()` along a single wrapped row of panels.

```
p + facet_wrap(facets = vars(vs, am), nrow = 2, labeller = label_both)
```

1.9 Scales

In earlier sections of this chapter, examples have used the default *scales* or we have set them with convenience functions. In the present section we describe in more detail the use of *scales*. There are *scales* available for different *aesthetics* ($\approx$ attributes) of the plotted geometrical objects, such as position (`x`, `y`, `z`), `size`, `shape`, `linetype`, `color`, `fill`, `alpha` or transparency, `angle`. Scales determine how values in `data` are mapped to values of an *aesthetics*, and how these values are labeled.

Depending on the characteristics of the data being mapped, *scales* can be continuous or discrete, for `numeric` or `factor` variables in `data`, respectively. On the other hand, some *aesthetics*, like `size`, can vary continuously but others like `linetype` are inherently discrete. In addition to discrete scales for inherently discrete *aesthetics*, discrete scales are available for those *aesthetics* that are inherently continuous, like `x`, `y`, `size`, `color`, etc.

The scales used by default set the mapping automatically (e.g., which color value corresponds to $x = 0$ and which one to $x = 1$). However, for each *aesthetic* such

as `color`, there are multiple scales to choose from when creating a plot, both continuous and discrete (e.g., 20 different color scales in 'ggplot2' 3.2.0).

> 🖥   *Aesthetics* in a plot layer, in addition to being determined by mappings, can also be set to constant values (e.g., plotting all points in a layer in red instead of the default black). *Aesthetics* set to constant values, are not mapped to data, and are consequently independent of scales. In other words, properties of plot elements can be either set to a single constant value of an *aesthetic* affecting all observations present in the layer `data`, or mapped to a variable in `data` in which case the value of the *aesthetic*, such as `color`, will depend on the values of the mapped variable.

The most direct mapping to data is `identity`, which means that the data is taken at its face value. In a color scale, say `scale_color_identity()`, the variable in the data would be encoded with values such as `"red"`, `"blue"`—i.e., valid R colours. In a simple mapping using `scale_color_discrete()` levels of a factor, such as `"treatment"` and `"control"` would be represented as distinct colours with the correspondence of individual factor levels to individual colours selected automatically by default. In contrast with `scale_color_manual()` the user needs to explicitly provide the mapping between factor levels and colours by passing arguments to the scale functions' parameters `breaks` and `values`.

A continuous data variable needs to be mapped to an *aesthetic* through a continuous scale such as `scale_color_continuous()` or one its various variants. Values in a `numeric` variable will be mapped into a continuous range of colours, determined either automatically through a palette or manually by giving the colours at the extremes, and optionally at multiple intermediate values, within the range of variation of the mapped variable (e.g., scale settings so that the color varies gradually between `"red"` and `"gray50"`). Handling of missing values is such that mapping a value in a variable to an `NA` value for an aesthetic such as color makes the mapped values invisible. The reverse, mapping `NA` values in the data to a specific value of an aesthetic is also possible (e.g., displaying `NA` values in the mapped variable in red, while other values are mapped to shades of blue).

1.9.1   Axis and key labels

First we describe a feature common to all scales, their `name`. The default `name` of all scales is the name of the variable or the expression mapped to it. In the case of the `x`, `y` and `z` *aesthetics* the `name` given to the scale is used for the axis labels. For other *aesthetics* the name of the scale becomes the "heading" or *key title* of the guide or key. All scales have a `name` parameter to which a character string or R expression (see section 1.14) can be passed as an argument to override the default.

Whole-plot title, subtitle and caption are not connected to *scales* or `data`. A title (`label`) and `subtitle` can be added least confusingly with function `ggtitle()` by passing either character strings or R expressions as arguments.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line() +
  geom_point() +
```

```
expand_limits(y = 0) +
scale_x_continuous(name = "Time (d)") +
scale_y_continuous(name = "Circumference (mm)") +
ggtitle(label = "Growth of orange trees",
        subtitle = "Starting from 1968-12-31")
```
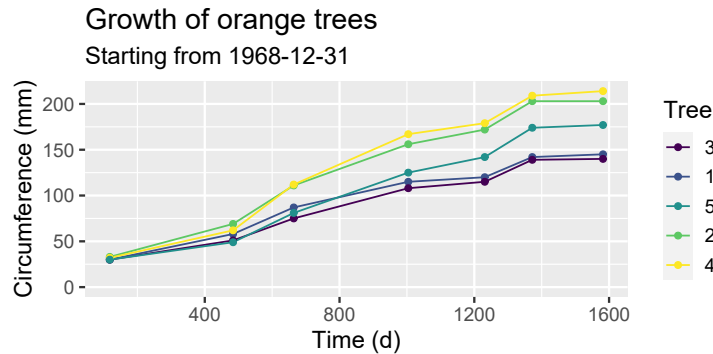

Convenience functions `xlab()` and `ylab()` can be used to set the axis labels to match those in the previous chunk.

```
xlab("Time (d)") +
ylab("Circumference (mm)") +
```

Convenience function `labs()` is useful when we use default scales for all the *aesthetics* in a plot but want to manually set axis labels and/or key titles—i.e., the `name` of these scales. `labs()` accepts arguments for these names using, as parameter names, the names of the *aesthetics*. It also allows us to set `title`, `subtitle`, `caption` and `tag`, of which the first two can also be set with `ggtitle()`.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line() +
  geom_point() +
  expand_limits(y = 0) +
  labs(title = "Growth of orange trees",
       subtitle = "Starting from 1968-12-31",
       caption = "see Draper, N. R. and Smith, H. (1998)",
       tag = "A",
       x = "Time (d)",
       y = "Circumference (mm)",
       color = "Tree\nnumber")
```
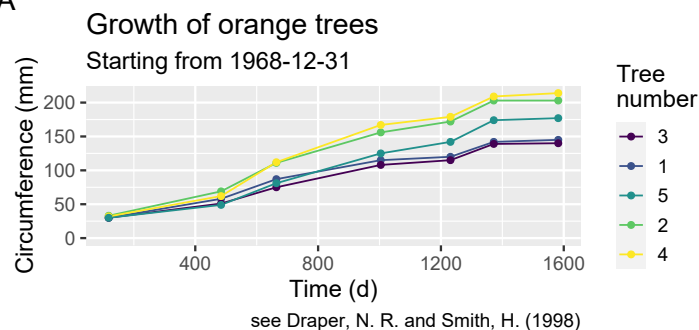
⌨ **1.19** Make an empty plot (`ggplot()`) and add to it as title an R expression producing $y = b_0 + b_1 x + b_2 x^2$. (Hint: have a look at the examples for the use of expressions in the `plotmath` demo in R by typing `demo(plotmath)` at the R console.

1.9.2 Continuous scales

We start by listing the most frequently used arguments to the continuous scale functions: `name`, `breaks`, `minor_breaks`, `labels`, `limits`, `expand`, `na.value`, `trans`, `guide`, and `position`. The value of `name` is used for axis labels or the key title (see previous section). The arguments to `breaks` and `minor_breaks` override the default locations of major and minor ticks and grid lines. Setting them to `NULL` suppresses the ticks. By default the tick labels are generated from the value of `breaks` but an argument to `labels` of the same length as `breaks` will replace these defaults. The values of `limits` determine both the range of values in the data included and the plotting area as described above—by default the out-of-bounds (`oob`) observations are replaced by `NA` but it is possible to instead "squish" these observations towards the edge of the plotting area. The argument to `expand` determines the size of the margins or padding added to the area delimited by `lims` when setting the "visual" plotting area. The value passed to `na.value` is used as a replacement for `NA` valued observations—most useful for `color` and `fill` aesthetics. The transformation object passed as an argument to `trans` determines the transformation used—the transformation affects the rendering, but breaks and tick labels remain expressed in the original data units. The argument to `guide` determines the type of key or removes the default key. Depending on the scale in question not all these parameters are available.

We generate new fake data.

```
fake2.data <-
  data.frame(y = c(rnorm(20, mean = 20, sd = 5),
                   rnorm(20, mean = 40, sd = 10)),
             group = factor(c(rep("A", 20), rep("B", 20))),
             z = rnorm(40, mean = 12, sd = 6))
```

Limits

Limits are relevant to all kinds of *scales*. Limits are set through parameter `limits` of the different scale functions. They can also be set with convenience functions `xlim()` and `ylim()` in the case of the x and y *aesthetics*, and more generally with function `lims()` which like `labs()`, takes arguments named according to the name of the *aesthetics*. The `limits` argument of scales accepts vectors, factors or a function computing them from `data`. In contrast, the convenience functions do not accept functions as their arguments.

In the next example we set "hard" limits, which will exclude some observations from the plot and from any computation of summaries or fitting of smoothers. More exactly, the off-limits observations are converted to `NA` values before they are passed as `data` to *geometries*.

```
ggplot(fake2.data, aes(z, y)) + geom_point() +
  scale_y_continuous(limits = c(0, 100))
```

To set only one limit leaving the other free, we can use NA as a boundary.

```
scale_y_continuous(limits = c(50, NA))
```

Convenience functions ylim() and xlim() can be used to set the limits to the default x and y scales in use. We here use ylim(), but xlim() is identical except for the *scale* it affects.

```
ylim(50, NA)
```

In general, setting hard limits should be avoided, even though a warning is issued about NA values being omitted, as it is easy to unwillingly subset the data being plotted. It is preferable to use function expand_limits() as it safely *expands* the dynamically computed default limits of a scale—the scale limits will grow past the requested expanded limits when needed to accommodate all observations. The arguments to x and y are numeric vectors of length one or two each, matching how the limits of the x and y continuous scales are defined. Here we expand the limits to include the origin.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  expand_limits(y = 0, x = 0)
```
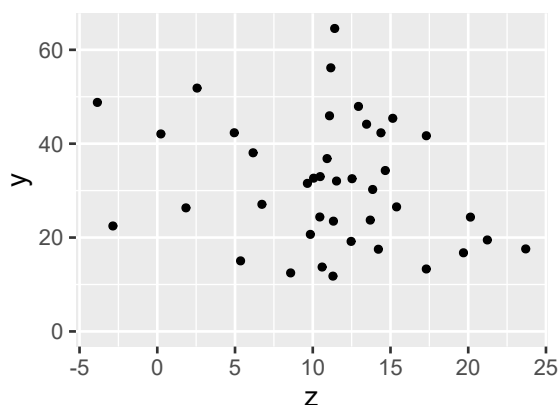

The expand parameter of the scales plays a different role than expand_limits(). It controls how much larger the "visual" plotting area is compared to the limits of the actual plotting area. In other words, it adds a "margin" or padding to the plotting area outside the limits set either dynamically or manually. Very rarely plots are drawn so that observations are plotted on top of the axes, avoiding this is a key role of expand. Rug plots and marginal annotations will also require the plotting area to be expanded. In 'ggplot2' the default is to always apply some expansion.

We here set the upper limit of the plotting area to be expanded by adding padding to the top and remove the default padding from the bottom of the plotting area.

```
ggplot(fake2.data,
  aes(fill = group, color = group, x = y)) +
  stat_density(alpha = 0.3) +
  scale_y_continuous(expand = expand_scale(add = c(0, 0.02)))
```

Here we instead use a multiplier to a similar effect as above; we add 10% compared to the range of the limits.

```
  scale_y_continuous(expand = expand_scale(mult = c(0, 0.1)))
```

In the case of scales, we cannot reverse their direction through the setting of limits. We need instead to use a transformation as described in section 1.9.2 on page 74. But, inconsistently, `xlim()` and `ylim()` do implicitly allow this transformation through the numeric values passed as limits.

⌨ **1.20** Test what the result is when the first limit is larger than the second one. Is it the same as when setting these same values as limits with `ylim()`?

```
ggplot(fake2.data, aes(z, y)) + geom_point() +
  scale_y_continuous(limits = c(100, 0))
```

Ticks and their labels

Parameter `breaks` is used to set the location of ticks along the axis. Parameter `labels` is used to set the tick labels. Both parameters can be passed either a vector or a function as an argument. The default is to compute "good" breaks based on the limits and format the numbers as strings.

When manually setting breaks, we can keep the default computed labels for the breaks.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  scale_y_continuous(breaks = c(20, pi * 10, 40, 60))
```

The default breaks are computed by function `pretty_breaks()` from 'scales'. The argument passed to its parameter `n` determines the target number ticks to be generated automatically, but the actual number of ticks computed may be slightly different depending on the range of the data.

```
  scale_y_continuous(breaks = pretty_breaks(n = 7))
```

We can set tick labels manually, in parallel to the setting of `breaks` by passing as arguments two vectors of equal length. In the next example we use an expression to obtain a Greek letter.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  scale_y_continuous(breaks = c(20, pi * 10, 40, 60),
                     labels = c("20", expression(10*pi), "40", "60"))
```
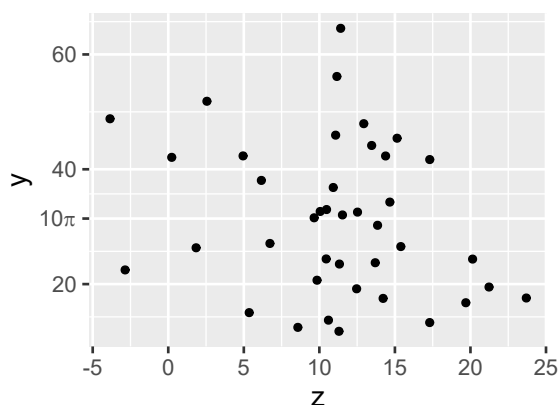
Package 'scales' provides several functions for the automatic generation of tick labels. For example, to display tick labels as percentages for data available as decimal fractions, we can use function `percent()`.

```r
ggplot(fake2.data, aes(z, y / max(y))) +
  geom_point() +
  scale_y_continuous(labels = percent)
```
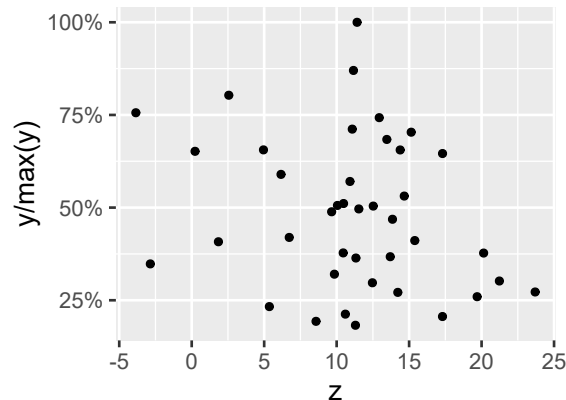

For currency, we can use `dollar()`, to include commas separating thousands, millions, so on, we can use `comma()`, and for numbers formatted using exponents of 10—useful for logarithmic-transformed scales—we can use `scientific_format()`, `label_number(scale_cut = cut_short_scale())`, `label_log()`, or `label_number(scale_cut = cut_si("g"))`. As shown below, some of these functions can be useful with untransformed continuous scales.

```r
ggplot(fake2.data, aes(z, y * 1000)) +
  geom_point() +
        scale_y_continuous(name     =     "Mass",     labels     =     la-
bel_number(scale_cut = cut_si("g")))
```
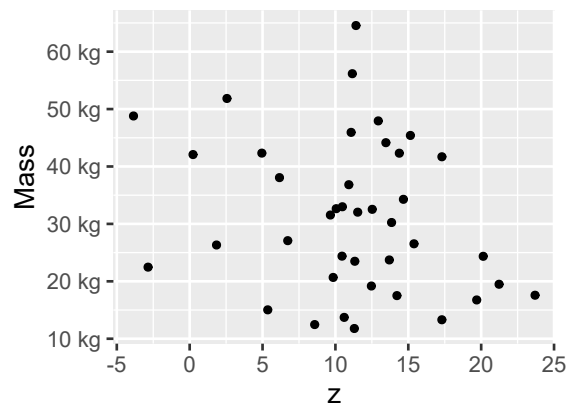

With date values mapped to x or y, tick labels are created with functions `label_date()` or `label_date_short()`. In the case of time, tick labels are created with function `label_time()`.

```
## ADD EXAMPLES USING FORMATS for dates and times
```

It is also possible to use user-defined functions both for breaks and labels.

Transformed scales

The default scales used by the `x` and `y` aesthetics, `scale_x_continuous()` and `scale_y_continuous()`, accept a user-supplied transformation function as an argument to `trans` with default codetrans = "identity" (no transformation). In addition, there are predefined convenience scale functions for `log10`, `sqrt` and `reverse`.

⚠️   Similar to the maths functions of R, the name of the scales are `scale_x_log10()` and `scale_y_log10()` rather than `scale_y_log()` because in R, the function `log` returns the natural logarithm.

We can use `scale_x_reverse()` to reverse the direction of a continuous scale,

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  scale_x_reverse()
```
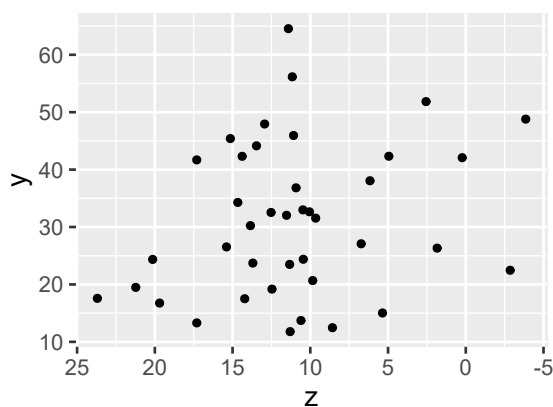

Axis tick-labels display the original values before applying the transformation. The `"breaks"` need to be given in the original scale as well. We use `scale_y_log10()` to apply a $\log_{10}$ transformation to the y values.

```
scale_y_log10(breaks=c(10,20,50,100))
```

Using a transformation in a scale is not equivalent to applying the same transformation on the fly when mapping a variable to the x (or y) *aesthetic* as this results in tick-labels expressed in transformed values.

```
ggplot(fake2.data, aes(z, log10(y))) +
  geom_point()
```

We show next how to specify a transformation to a continuous scale, using a predefined "transformation" object.

```
scale_y_continuous(trans = "reciprocal")
```

Natural logarithms are important in growth analysis as the slope against time gives the relative growth rate. We show this with the `Orange` data set.

```
ggplot(data = Orange,
       aes(x = age, y = circumference, color = Tree)) +
  geom_line() +
  geom_point() +
  scale_y_continuous(trans = "log", breaks = c(20, 50, 100, 200))
```

Position of x and y axes

The default position of axes can be changed through parameter `position`, using character constants `"bottom"`, `"top"`, `"left"` and `"right"`.

```
ggplot(data = mtcars, aes(wt, mpg)) +
  geom_point() +
  scale_x_continuous(position = "top") +
  scale_y_continuous(position = "right")
```


Secondary axes

It is also possible to add secondary axes with ticks displayed in a transformed scale.

```
ggplot(data = mtcars, aes(wt, mpg)) +
  geom_point() +
  scale_y_continuous(sec.axis = sec_axis(~ . ^-1, name = "1/y") )
```


It is also possible to use different `breaks` and `labels` than for the main axes, and to provide a different `name` to be used as a secondary axis label.

```
  scale_y_continuous(sec.axis = sec_axis(~ . / 2.3521458, name = expres-
sion(km / l),
                                          breaks = c(5, 7.5, 10, 12.5)))
```

1.9.3   Time and date scales for x and y

In R and many other computing languages, time values are stored as integer or numeric values subject to special interpretation. Times stored as objects of class POSIXct can be mapped to continuous *aesthetics* such as x and y. Special scales are available for these quantities.

We can set limits and breaks using constants as time or dates. These are most easily input with the functions in packages 'lubridate' or 'anytime'.

⚠   Warnings are issued in the next two chunks as we are using scale limits to subset a part of the observations present in data.

```r
ggplot(data = weather_wk_25_2019.tb,
       aes(with_tz(time, tzone = "EET"), air_temp_C)) +
  geom_line() +
  scale_x_datetime(name = NULL,
                   breaks = ymd_hm("2019-06-11 12:00", tz = "EET") + days(0:1),
                 limits = ymd_hm("2019-06-11 00:00", tz = "EET") + days(c(0, 2))) +
  scale_y_continuous(name = "Air temperature (C)") +
  expand_limits(y = 0)

## Warning: Removed 7199 rows containing missing values or values outside the
scale range
## (`geom_line()`).
```
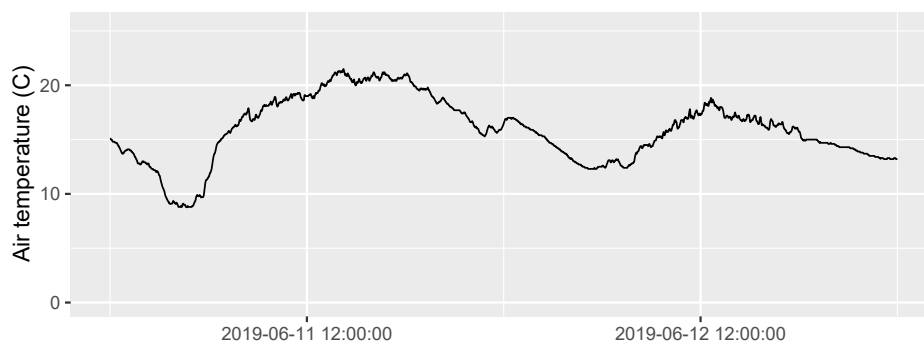

By default the tick labels produced and their formatting are automatically selected based on the extent of the time data. For example, if we have all data collected within a single day, then the tick labels will show hours and minutes. If we plot data for several years, the labels will show the date portion of the time instant. The default is frequently good enough, but it is possible, as for numbers, to use different formatter functions to generate the tick labels.

```r
ggplot(data = weather_wk_25_2019.tb,
       aes(with_tz(time, tzone = "EET"), air_temp_C)) +
  geom_line() +
  scale_x_datetime(name = NULL,
                   date_breaks = "1 hour",
                 limits = ymd_hm("2019-06-16 00:00", tz = "EET") + hours(c(6, 18)),
                   date_labels = "%H:%M") +
  scale_y_continuous(name = "Air temperature (C)") +
  expand_limits(y = 0)

## Warning: Removed 9359 rows containing missing values or values outside the
```

```
scale range
## (`geom_line()`).
```
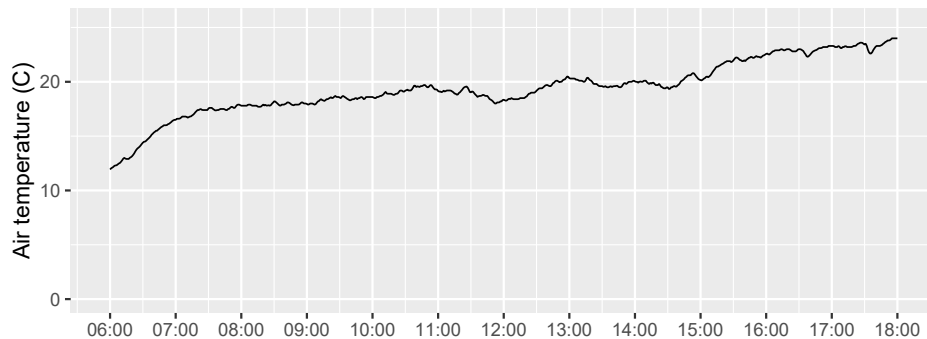

📧 **1.21** The formatting strings used are those supported by `strptime()` and `help(strptime)` lists them. Change, in the two examples above, the y-axis labels used and the limits—e.g., include a single hour or a whole week of data, check which tick labels are produced by default and then pass as an argument to `date_labels` different format strings, taking into account that in addition to the *conversion specification* codes, format strings can include additional text.

1.9.4   Discrete scales for x and y

In the case of ordered or unordered factors, the tick labels are by default the names of the factor levels. Consequently, one roundabout way of obtaining the desired tick labels is to set them as factor levels. This approach is not recommended as in many cases the text of the desired tick labels may not be recognized as a valid name making the code using them more difficult to type in scripts or at the command prompt. It is best to use simple mnemonic short names for factor levels and variables, and to set suitable labels through *scales* when plotting, as we will show here.

We can use `scale_x_discrete()` to reorder and select the columns without altering the data. If we use this approach to subset the data, then to avoid warnings we need to add `na.rm = TRUE`. We additionally use `scale_x_discrete` to convert level names to uppercase.

```
ggplot(mpg, aes(class, hwy)) +
  stat_summary(geom = "col", fun = mean, na.rm = TRUE) +
  scale_x_discrete(limits = c("compact", "subcompact", "midsize"),
                   labels = c("COMPACT", "SUBCOMPACT", "MIDSIZE"))
```
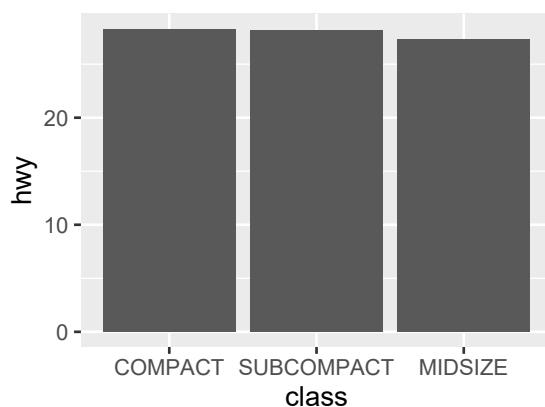
If, as in the previous example, only the case of character strings needs to be changed, passing function `toupper()` or `tolower()` allows a more general and less error-prone approach. In fact any function, user defined or not, which converts the values of `limits` into the desired values can be passed as an argument to `labels`.

```
scale_x_discrete(limits = c("compact", "subcompact", "midsize"),
                 labels = toupper)
```

Alternatively, we can change the order of the columns in the plot by reordering the levels of factor `mpg$class`. This approach makes sense if the ordering needs to be done programmatically based on values in `data`. See section **??** on page **??** for details. The example below shows how to reorder the columns, corresponding to the levels of `class` based on the `mean()` of `hwy`.

```
ggplot(mpg, aes(reorder(x = factor(class), X = hwy, FUN = mean), hwy)) +
  stat_summary(geom = "col", fun = mean)
```

1.9.5   Size

For the `size` *aesthetic*, several scales are available, both discrete and continuous. They do not differ much from those already described above. *Geometries* `geom_point()`, `geom_line()`, `geom_hline()`, `geom_vline()`, `geom_text()`, `geom_label()` obey `size` as expected. In the case of `geom_bar()`, `geom_col()`, `geom_area()` and all other geometric elements bordered by lines, `size` is obeyed by these border lines. In fact, other aesthetics natural for lines such as `linetype` also apply to these borders.

When using `size` scales, `breaks` and `labels` affect the key or `guide`. In scales that produce a key passing `guide = "none"` removes the key corresponding to the scale.

1.9.6   Color and fill

color and fill scales are similar, but they affect different elements of the plot. All visual elements in a plot obey the `color` *aesthetic*, but only elements that have an inner region and a boundary, obey both `color` and `fill` *aesthetics*. There are separate but equivalent sets of scales available for these two *aesthetics*. We will

describe in more detail the `color` *aesthetic* and give only some examples for `fill`. We will, however, start by reviewing how colors are defined and used in R.

Color definitions in R

Colors can be specified in R not only through character strings with the names of previously defined colors, but also directly as strings describing the RGB (red, green and blue) components as hexadecimal numbers (on base 16 expressed using 0, 1, 2, 3, 4, 6, 7, 8, 9, A, B, C, D, E, and F as "digits") such as `"#FFFFFF"` for white or `"#000000"` for black, or `"#FF0000"` for the brightest available pure red.

The list of color names known to R can be obtained be typing `colors()` at the R console. Given the number of colors available, we may want to subset them based on their names. Function `colors()` returns a character vector. We can use `grep()` to find the names containing a given character substring, in this example `"dark"`.

```
length(colors())
## [1] 657
grep("dark",colors(), value = TRUE)
##  [1] "darkblue"       "darkcyan"       "darkgoldenrod"  "darkgoldenrod1"
##  [5] "darkgoldenrod2" "darkgoldenrod3" "darkgoldenrod4" "darkgray"
##  [9] "darkgreen"      "darkgrey"       "darkkhaki"      "darkmagenta"
## [13] "darkolivegreen" "darkolivegreen1" "darkolivegreen2" "darkolivegreen3"
## [17] "darkolivegreen4" "darkorange"     "darkorange1"    "darkorange2"
## [21] "darkorange3"    "darkorange4"    "darkorchid"     "darkorchid1"
## [25] "darkorchid2"    "darkorchid3"    "darkorchid4"    "darkred"
## [29] "darksalmon"     "darkseagreen"   "darkseagreen1"  "darkseagreen2"
## [33] "darkseagreen3"  "darkseagreen4"  "darkslateblue"  "darkslategray"
## [37] "darkslategray1" "darkslategray2" "darkslategray3" "darkslategray4"
## [41] "darkslategrey"  "darkturquoise"  "darkviolet"
```

To retrieve the RGB values for a color definition we use:

```
col2rgb("purple")
##       [,1]
## red    160
## green   32
## blue   240
col2rgb("#FF0000")
##       [,1]
## red    255
## green    0
## blue     0
```

Color definitions in R can contain a *transparency* described by an `alpha` value, which by default is not returned.

```
col2rgb("purple", alpha = TRUE)
##       [,1]
## red    160
## green   32
## blue   240
## alpha  255
```

With function `rgb()` we can define new colors. Enter `help(rgb)` for more details.

```
rgb(1, 1, 0)
## [1] "#FFFF00"
```

```
rgb(1, 1, 0, names = "my.color")
##  my.color
## "#FFFF00"
rgb(255, 255, 0, names = "my.color", maxColorValue = 255)
##  my.color
## "#FFFF00"
```

As described above, colors can be defined in the RGB *color space*, however, other color models such as HSV (hue, saturation, value) can be also used to define colours.

```
hsv(c(0,0.25,0.5,0.75,1), 0.5, 0.5)
## [1] "#804040" "#608040" "#408080" "#604080" "#804040"
```

Probably a more useful flavor of HSV colors for use in scales are those returned by function `hcl()` for hue, chroma and luminance. While the "value" and "saturation" in HSV are based on physical values, the "chroma" and "luminance" values in HCL are based on human visual perception. Colours with equal luminance will be seen as equally bright by an "average" human. In a scale based on different hues but equal chroma and luminance values, as used by package 'ggplot2', all colours are perceived as equally bright. The hues need to be expressed as angles in degrees, with values between zero and 360.

```
hcl(c(0,0.25,0.5,0.75,1) * 360)
## [1] "#FFC5D0" "#D4D8A7" "#99E2D8" "#D5D0FC" "#FFC5D0"
```

It is also important to remember that humans can only distinguish a limited set of colours, and even smaller color gamuts can be reproduced by screens and printers. Furthermore, variation from individual to individual exists in color perception, including different types of color blindness. It is important to take this into account when choosing the colors used in illustrations.

1.9.7 Continuous color-related scales

Continuous color scales `scale_color_continuous()`, `scale_color_gradient()`, `scale_color_gradient2()`, `scale_color_gradientn()`, `scale_color_date()` and `scale_color_datetime()`, give a smooth continuous gradient between two or more colours. They are used with `numeric`, `date` and `datetime` data. A corresponding set of `fill` scales is also available. Other scales like `scale_color_viridis_c()` and `scale_color_distiller()` are based on the use of ready-made palettes of sets of color gradients chosen to work well together under multiple conditions or for human vision including different types of color blindness.

1.9.8 Discrete color-related scales

Color scales `scale_color_discrete()`, `scale_color_hue()`, `scale_color_gray()` are used with categorical data stored as factors. Other scales like `scale_color_viridis_d()` and `scale_color_brewer()` provide discrete sets of colours based on palettes.

1.9.9 Binned scales

Before version 3.3.0 of 'ggplot2' only two types of scales were available, continuous and discrete. A third type of scales (implemented for all the aesthetics where relevant) was added in version 3.3.0 called *binned*. They are to be used with continuous variables, but they discretize the continuous values into bins or classes, each for a range of values, and then represent them in the plot using a discrete set of values. We re-do the figure shown on page 22 but replacing `scale_color_gradient()` by `scale_color_binned()`.

```
set.seed(4321)
X <- 0:10
Y <- (X + X^2 + X^3) + rnorm(length(X), mean = 0, sd = mean(X^3) / 4)
df1 <- data.frame(X, Y)
df2 <- df1
df2[6, "Y"] <-df1[6, "Y"] * 10
```

```
ggplot(df2) +
  stat_fit_residuals(formula = y ~ poly(x, 3, raw = TRUE),
                     method = "rlm",
                     mapping = aes(x = X,
                                   y = stage(start = Y,
                                             after_stat = y * weights),
                                   colour = after_stat(weights)),
                     show.legend = TRUE) +
  scale_color_binned(low = "red", high = "blue", limits = c(0, 1),
                     guide = "colourbar", n.breaks = 5)
```
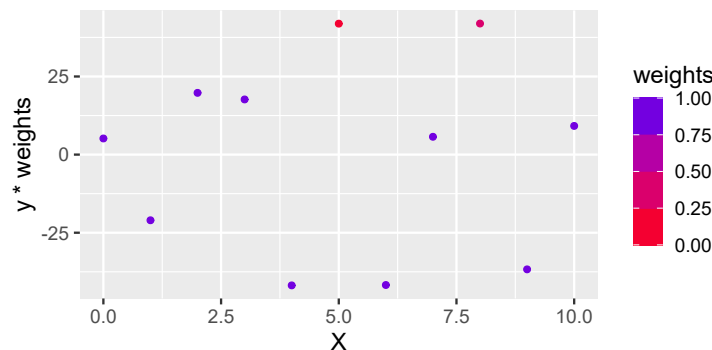

The advantage of binned scales is that they facilitate the fast reading of the plot while their disadvantage is the decreased resolution of the scale. The choice of a binned vs. continuous scale, and the number and boundaries of bins, set by the argument passed to parameter `n.breaks` or to `breaks` need to be chosen carefully, taking into account the audience, the length of time available to the viewer to peruse the plot vs. the density of observations. Transformations are also allowed in these scales as in others.

1.9.10 Identity scales

In the case of identity scales, the mapping is one to-one to the data. For example, if we map the `color` or `fill` *aesthetic* to a variable using `scale_color_identity()` or `scale_fill_identity()`, the mapped variable must already contain valid color

definitions. In the case of mapping `alpha`, the variable must contain numeric values in the range 0 to 1.

We create a data frame containing a variable `colors` containing character strings interpretable as the names of color definitions known to R. We then use them directly in the plot.

```
df3 <- data.frame(x = 1:10, y = dnorm(10), colors = rep(c("red", "blue"), 5))

ggplot(df3, aes(x, y, color = colors)) +
  geom_point() +
  scale_color_identity()
```
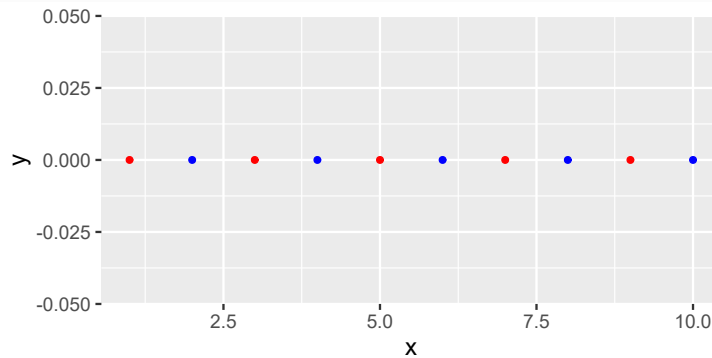

⌨ **1.22** How does the plot look, if the identity scale is deleted from the example above? Edit and re-run the example code.

While using the identity scale, how would you need to change the code example above, to produce a plot with green and purple points?

1.10 Adding annotations

The idea of annotations is that they add plot elements that are not directly connected with `data`, which we could call "decorations" such as arrows used to highlight some feature of the data, specific points along an axis, etc. They are referenced to the "natural" coordinates used to plot the observations, but are elements that do not represent observations or summaries computed from the observations. Annotations are added to a ggplot with `annotate()` as plot layers (each call to `annotate()` creates a new layer). To achieve the behavior expected of annotations, `annotate()` does not inherit the default `data` or `mapping` of variables to *aesthetics*. Annotations frequently make use of `"text"` or `"label"` *geometries* with character strings as data, possibly to be parsed as expressions. However, for example, the `"segment"` geometry can be used to add arrows.

⚠ While layers added to a plot directly using *geometries* and *statistics* respect faceting, annotation layers added with `annotate()` are replicated unchanged in every panel of a faceted plot. The reason is that annotation layers accept *aesthetics*

only as constant values which are the same for every panel as no grouping is possible without a `mapping` to `data`.

We show a simple example using `"text"` as *geometry*.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  annotate(geom = "text",
           label = "origin",
           x = 0, y = 0,
           color = "blue",
           size=4)
```
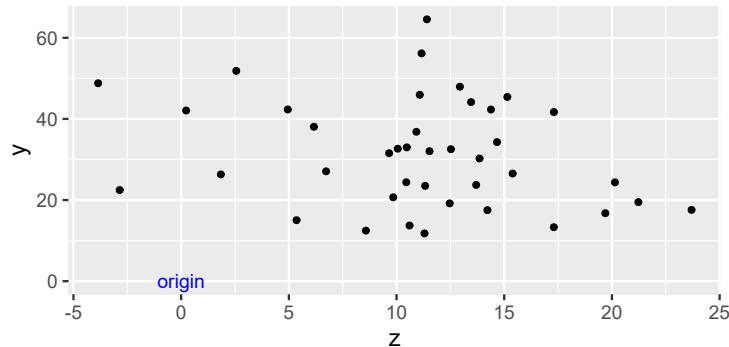

⌨ 1.23 Play with the values of the arguments to `annotate()` to vary the position, size, color, font family, font face, rotation angle and justification of the annotation.

It is relatively common to use inset tables, plots, bitmaps or vector plots as annotations. With `annotation_custom()`, grobs ('grid' graphical object) can be added to a ggplot. To add another or the same plot as an inset, we first need to convert it into a grob. In the case of a ggplot we use `ggplotGrob()`. In this example the inset is a zoomed-in window into the main plot. In addition to the grob, we need to provide the coordinates expressed in "natural" data units of the main plot for the location of the grob.

```
p <- ggplot(fake2.data, aes(z, y)) +
  geom_point()
p + expand_limits(x = 40) +
  annotation_custom(ggplotGrob(p + coord_cartesian(xlim = c(5, 10), ylim = c(20, 40)) +
                                 theme_bw(10)),
                    xmin = 21, xmax = 40, ymin = 30, ymax = 60)
```
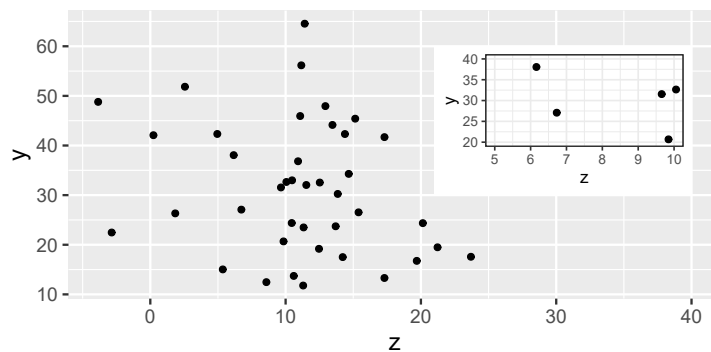

This approach has the limitation that if used together with faceting, the inset

will be the same for each plot panel. See section 1.5.8 on page 39 for *geometries* that can be used to add insets.

In the next example, in addition to adding expressions as annotations, we also pass expressions as tick labels through the scale. Do notice that we use recycling for setting the breaks, as `c(0, 0.5, 1, 1.5, 2) * pi` is equivalent to `c(0, 0.5 * pi, pi, 1.5 * pi, 2 * pi`. Annotations are plotted at their own position, unrelated to any observation in the data, but using the same coordinates and units as for plotting the data.

```r
ggplot(data.frame(x = c(0, 2 * pi)), aes(x = x)) +
  stat_function(fun = sin) +
  scale_x_continuous(
    breaks = c(0, 0.5, 1, 1.5, 2) * pi,
    labels = c("0", expression(0.5~pi), expression(pi),
               expression(1.5~pi), expression(2~pi))) +
  labs(y = "sin(x)") +
  annotate(geom = "text",
           label = c("+", "-"),
           x = c(0.5, 1.5) * pi, y = c(0.5, -0.5),
           size = 20) +
  annotate(geom = "point",
           color = "red",
           shape = 21,
           fill = "white",
           x = c(0, 1, 2) * pi, y = 0,
           size = 6)
```
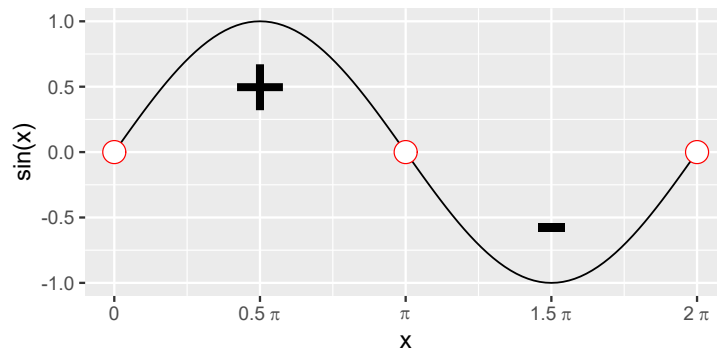

1.24 Modify the plot above to show the cosine instead of the sine function, replacing `sin` with `cos`. This is easy, but the catch is that you will need to relocate the annotations.

⚠️   We cannot use `annotate()` with `geom = "vline"` or `geom = "hline"` as we can use `geom = "line"` or `geom = "segment"`. Instead, `geom_vline()` and/or `geom_hline()` can be used directly passing constant arguments to them. See section 1.5.3 on page 31.

1.11 Coordinates and circular plots

Circular plots can be thought of as plots equivalent to those described earlier in this chapter but drawn using a different system of coordinates. This is a key insight, that the grammar of graphics as implemented in 'ggplot2' makes use of. To obtain circular plots we use the same *geometries*, *statistics* and *scales* we have been using with the default system of cartesian coordinates. The only thing that we need to do is to add `coord_polar()` to override the default. Of course only some observed quantities can be better perceived in circular plots than in cartesian plots. Here we add a new "word" to the grammar of graphics, *coordinates*, such as `coord_polar()`. When using polar coordinates, the x and y *aesthetics* correspond to the angle and radial distance, respectively.

1.11.1 Wind-rose plots

Some types of data are more naturally expressed on polar coordinates than on cartesian coordinates. The clearest example is wind direction, from which the name derives. In some cases of time series data with a strong periodic variation, polar coordinates can be used to highlight any phase shifts or changes in frequency. A more mundane application is to plot variation in a response variable through the day with a clock-face-like representation of time of day.

Wind rose plots are frequently histograms or density plots drawn on a polar system of coordinates (see sections 1.6.4 and 1.6.5 on pages 51 and 54, respectively for a description of the use of these *statistics* and *geometries*). We will use them for examples where we plot wind speed and direction data, measured once per minute during 24 h (from package 'learnrbook').

Here we plot a circular histogram of wind directions with 30-degree-wide bins. We use `stat_bin()`. The counts represent the number of minutes during 24 h when the wind direction was within each bin.

```r
p <- ggplot(viikki_d29.dat, aes(WindDir_D1_WVT))  +
  coord_polar() +
  scale_x_continuous(breaks = c(0, 90, 180, 270),
                     labels = c("N", "E", "S", "W"),
                     limits = c(0, 360),
                     expand = c(0, 0),
                     name = "Wind direction")
p + stat_bin(color = "black", fill = "gray50", geom = "bar",
             binwidth = 30, na.rm = TRUE) + labs(y = "Frequency")
```
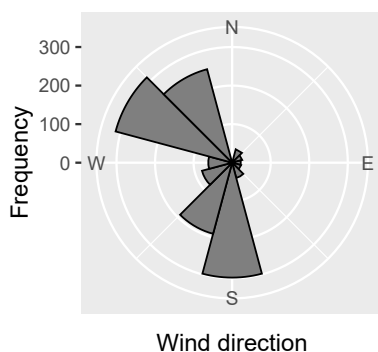
For an equivalent plot, using an empirical density, we have to use `stat_density()` instead of `stat_bin()`, `geom_polygon()` instead of `geom_bar()` and change the `name` of the y scale.

```
p + stat_density(color = "black", fill = "gray50",
                 geom = "polygon", size = 1) + labs(y = "Density")

## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```
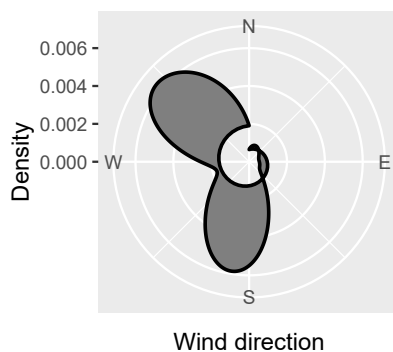

As the final wind-rose plot example, we do 2D density plot with facets added with `facet_wrap()` to have separate panels for AM and PM. This plot uses fill to describe the density of observations for different combinations wind directions and speeds, the radius (*y aesthetic*) to represent wind speeds and the angle (*x aesthetic*) to represent wind direction.

```
ggplot(viikki_d29.dat, aes(WindDir_D1_WVT, WindSpd_S_WVT)) +
  coord_polar() +
  stat_density_2d(aes(fill = after_stat(level)), geom = "polygon") +
  scale_x_continuous(breaks = c(0, 90, 180, 270),
                     labels = c("N", "E", "S", "W"),
                     limits = c(0, 360),
                     expand = c(0, 0),
                     name = "Wind direction") +
  scale_y_continuous(name = "Wind speed (m/s)") +
  facet_wrap(~factor(ifelse(hour(solar_time) < 12, "AM", "PM")))
```
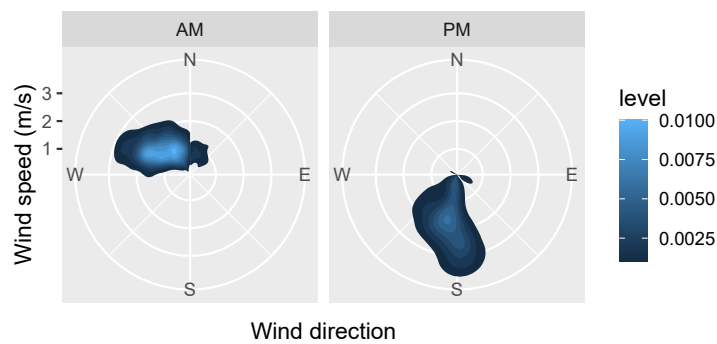
1.11.2 Pie charts

⚠ Pie charts are more difficult to read than bar charts because our brain is better at comparing lengths than angles. If used, pie charts should only be used to show composition, or fractional components that add up to a total. In this case, used only if the number of "pie slices" is small (rule of thumb: seven at most), however in general, they are best avoided.

As we use `geom_bar()` which defaults to use `stat_count()`. We use the brewer scale for nice colors.

```r
ggplot(data = mpg, aes(x = factor(1), fill = factor(class))) +
  geom_bar(width = 1, color = "black") +
  coord_polar(theta = "y") +
  scale_fill_brewer() +
  scale_x_discrete(breaks = NULL) +
  labs(x = NULL, fill = "Vehicle class")
```
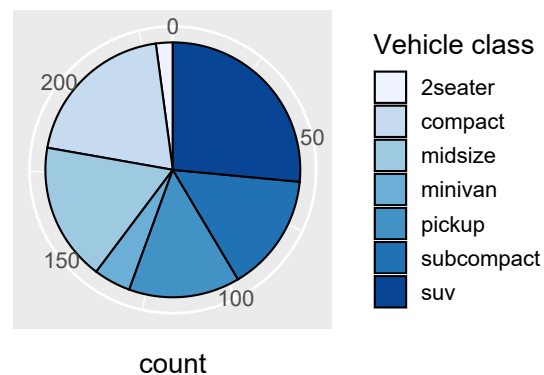

⌨ **1.25** Edit the code for the pie chart above to obtain a bar chart. Which one of the two plots is easier to read?

1.12   Themes

In 'ggplot2', *themes* are the equivalent of style sheets. They determine how the different elements of a plot are rendered when displayed, printed or saved to a file. *Themes* do not alter what aesthetics or scales are used to plot the observations or summaries, but instead how text-labels, titles, axes, grids, plotting-area background and grid, etc., are formatted and if displayed or not. Package 'ggplot2' includes several predefined *theme constructors* (usually described as *themes*), and independently developed extension packages define additional ones. These constructors return complete themes, which when added to a plot, replace any theme already present in whole. In addition to choosing among these already available *complete themes*, users can modify the ones already present by adding *incomplete themes* to a plot. When used in this way, *incomplete themes* usually are created on the fly. It is also possible to create new theme constructors that return complete themes, similar to `theme_gray()` from 'ggplot2'.

1.12.1   Complete themes

The theme used by default is `theme_gray()` with default arguments. In 'ggplot2', predefined themes are defined as constructor functions, with parameters. These parameters allow changing some "base" properties. The `base_size` for text elements controlled is given in points, and affects all text elements in the returned theme object as the size of these elements is by default defined relative to the base size. Another parameter, `base_family`, allows the font family to be set. These functions return complete themes.

⚠ *Themes* have no effect on layers produced by *geometries* as themes have no effect on *mappings*, *scales* or *aesthetics*. In the name `theme_bw()` black-and- white refers to the color of the background of the plotting area and labels. If the *color* or fill *aesthetics* are mapped or set to a constant in the figure, these will be respected irrespective of the theme. We cannot convert a color figure into a black-and-white one by adding a *theme*, we need to change the *aesthetics* used, for example, use `shape` instead of `color` for a layer added with `geom_point()`.

Even the default `theme_gray()` can be added to a plot, to modify it, if arguments different to the defaults are passed when called. In this example we override the default base size with a larger one and the default sans-serif font with one with serifs.

```
ggplot(fake2.data, aes(z, y)) +
  geom_point() +
  theme_gray(base_size = 15,
             base_family = "serif")
```
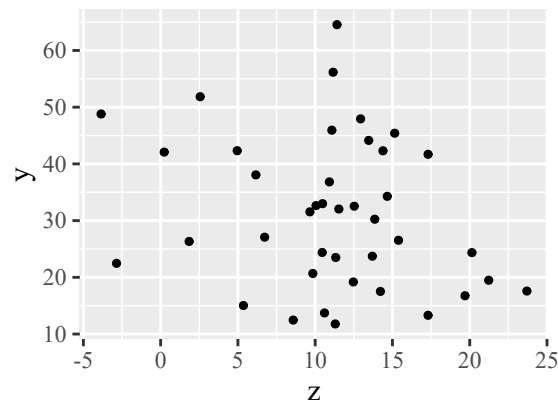
⌨ **1.26** Change the code in the previous chunk to use, one at a time, each of the predefined themes from 'ggplot2': `theme_bw()`, `theme_classic()`, `theme_minimal()`, `theme_linedraw()`, `theme_light()`, `theme_dark()` and `theme_void()`.

▣ Predefined "themes" like `theme_gray()` are, in reality, not themes but instead are constructors of theme objects. The *themes* they return when called depend on the arguments passed to their parameters. In other words, `theme_gray(base_size = 15)`, creates a different theme than `theme_gray(base_size = 11)`. In this case, as sizes of different text elements are defined relative to the base size, the size of all text elements changes in coordination. Font size changes by *themes* do not affect the size of text or labels in plot layers created with geometries, as their size is controlled by the `size` *aesthetic*.

A frequent idiom is to create a plot without specifying a theme, and then adding the theme when printing or saving it. This can save work, for example, when producing different versions of the same plot for a publication and a talk.

```
p <- ggplot(fake2.data, aes(z, y)) +
      geom_point()
print(p + theme_bw())
```

It is also possible to change the theme used by default in the current R session with `theme_set()`.

```
old_theme <- theme_set(theme_bw(15))
```

Similar to other functions used to change options in R, `theme_set()` returns the previous setting. By saving this value to a variable, here `old_theme`, we are able to restore the previous default, or undo the change.

```
theme_set(old_theme)
p
```

▣ The use of a grey background as default for plots is unusual. This graphic design decision originates in the typesetters desire to maintain a uniform luminosity throughout the text and plots in a page. Many scientific journals require or at least prefer a more traditional graphic design. Theme `theme_bw()` is the most versatile of the traditional designs supported as it works well both for individual plots as for plots with facets as it includes a box. Theme `theme_classic()` lacking

a box and grid works well for individual plots as is, but needs changes to the facet bars when used with facets.

1.12.2   Incomplete themes

If we want to extensively modify a theme, and/or reuse it in multiple plots, it is best to create a new constructor, or a modified complete theme as described in the next section. In other cases we may need to tweak some theme settings for a single figure, in which case we can most effectively do this when creating a plot. We exemplify this approach by solving the problem of overlapping *x*-axis tick labels. In practice this problem is most frequent when factor levels have long names or the labels are dates. Rotating the tick labels is the most elegant solution from the graphics design point of view.

```
ggplot(fake2.data, aes(z + 1000, y)) +
  geom_point() +
  scale_x_continuous(breaks = scales::pretty_breaks(n = 8)) +
  theme(axis.text.x = element_text(angle = 90, hjust = 1, vjust = 0.5))
```
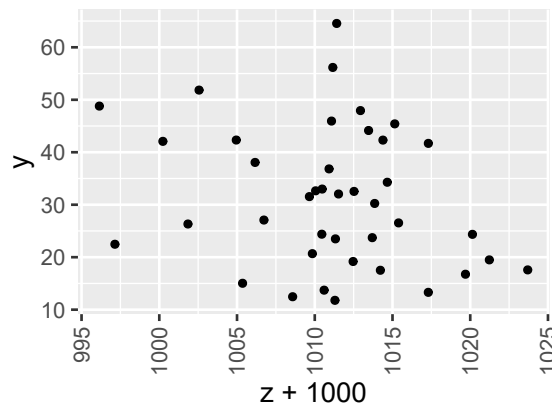

⚠️   When tick labels are rotated, one usually needs to set both the horizontal and vertical justification, `hjust` and `vjust`, as the default values stop being suitable. This is due to the fact that justification settings are referenced to the text itself rather than to the plot, i.e., **vertical** justification of *x*-axis tick labels rotated 90 degrees shifts their alignment with respect to tick marks along the (**horizontal**) *x* axis.

⌨ **1.27** Play with the code in the last chunk above, modifying the values used for `angle`, `hjust` and `vjust`. (Angles are expressed in degrees, and justification with values between 0 and 1).

A less elegant approach is to use a smaller font size. Within `theme()`, function `rel()` can be used to set size relative to the base size. In this example, we use `axis.text.x` so as to change the size of tick labels only for the *x* axis.

```
  theme(axis.text.x = element_text(size = rel(0.6)))
```

Theme definitions follow a hierarchy, allowing us to modify the formatting of groups of similar elements, as well as of individual elements. In the chunk above,

had we used `axis.text` instead of `axis.text.x`, the change would have affected the tick labels in both x and y axes.

⌨ **1.28** Modify the example above, so that the tick labels on the x-axis are blue and those on the y-axis red, and the font size is the same for both axes, but changed from the default. Consult the documentation for `theme()` to find out the names of the elements that need to be given new values. For examples, see *ggplot2: Elegant Graphics for Data Analysis* (Wickham and Sievert 2016) and *R Graphics Cookbook* (Chang 2018).

Formatting of other text elements can be adjusted in a similar way, as well as thickness of axes, length of tick marks, grid lines, etc. However, in most cases these are graphic design elements that are best kept consistent throughout sets of plots and best handled by creating a new *theme* that can be easily reused.

⚠ If you both add a *complete theme* and want to modify some of its elements, you should add the whole theme before modifying it with + `theme(...)`. This may seem obvious once one has a good grasp of the grammar of graphics, but can be at first disconcerting.

It is also possible to modify the default theme used for rendering all subsequent plots.

```
old_theme <- theme_update(text = element_text(color = "darkred"))
```

1.12.3   Defining a new theme

Themes can be defined both from scratch, or by modifying existing saved themes, and saving the modified version. As discussed above, it is also possible to define a new, parameterized theme constructor function.

Unless we plan to widely reuse the new theme, there is usually no need to define a new function. We can simply save the modified theme to a variable and add it to different plots as needed. As we will be adding a "ready-build" theme object rather than a function, we do not use parentheses.

```
my_theme <- theme_bw() + theme(text = element_text(color = "darkred"))
p + my_theme
```
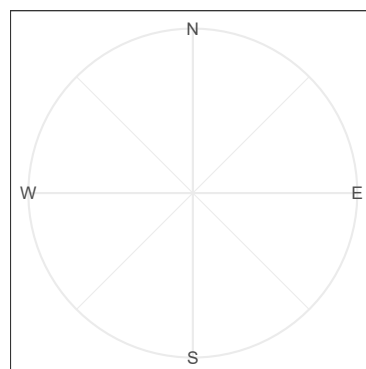

Wind direction

⌨ **1.29** It is always good to learn to recognize error messages. One way of doing this is by generating errors on purpose. So do add parentheses to the statement in the code chunk above and study the error message.

🖥 How to create a new theme constructor similar to those in package 'ggplot2' can be fairly simple if the changes are few. As the implementation details of theme objects may change in future versions of 'ggplot2', the safest approach is to rely only on the public interface of the package. We can "wrap" the functions exported by package 'ggplot2' inside a new function. For this we need to find out what are the parameters and their order and duplicate these in our wrapper. Looking at the "usage" section of the help page for `theme_gray()` is enough. In this case, we retain compatibility, but add a new base parameter, `base_color`, and set a different default for `base_family`. The key detail is passing `complete = TRUE` to `theme()`, as this tags the returned theme as being usable by itself, resulting in replacement of any theme already in a plot when it is added.

```r
my_theme_gray <-
  function (base_size = 11,
            base_family = "serif",
            base_line_size = base_size/22,
            base_rect_size = base_size/22,
            base_color = "darkblue") {
    theme_gray(base_size = base_size,
               base_family = base_family,
               base_line_size = base_line_size,
               base_rect_size = base_rect_size) +
    theme(line = element_line(color = base_color),
          rect = element_rect(color = base_color),
          text = element_text(color = base_color),
          title = element_text(color = base_color),
          axis.text = element_text(color = base_color), complete = TRUE)
  }
```

In the chunk above we have created our own theme constructor, without too much effort, and using an approach that is very likely to continue working with future versions of 'ggplot2'. The saved theme is a function with parameters and defaults for them. In this example we have kept the function parameters the same as those used in 'ggplot2', only adding an additional parameter after the existing ones to maximize compatibility and avoid surprising users. To avoid surprising users, we may want additionally to make `my_theme_grey()` a synonym of `my_theme_gray()` following 'ggplot2' practice.

```r
my_theme_grey <- my_theme_gray
```

Finally, we use the new theme constructor in the same way as those defined in 'ggplot2'.

```r
p + my_theme_gray(15, base_color = "darkred")
```
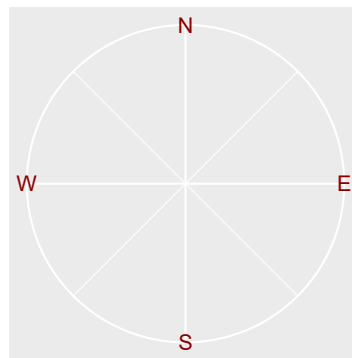
Wind direction

1.13 Composing plots

In section 1.8 on page 64, we described how facets can be used to create coordinated sets of panels, based on a single data set. Rather frequently, we need to assemble a composite plot from individually created plots. If one wishes to have correctly aligned axis labels and plotting areas, similar to when using facets, then the task is not easy to achieve without the help of especial tools.

Package 'patchwork' defines a simple grammar for composing plots created with 'ggplot2'. We briefly describe here the use of operators +, | and /, although 'patchwork' provides additional tools for defining complex layouts of panels. While + allows different layouts, | composes panels side by side, and / composes panels on top of each other. The plots to be used as panels can be grouped using parentheses.

We start by creating and saving three plots.

```
p1 <- ggplot(mpg, aes(displ, cty, color = factor(cyl))) +
        geom_point() +
        theme(legend.position = "top")
p2 <- ggplot(mpg, aes(displ, cty, color = factor(year))) +
        geom_point() +
        theme(legend.position = "top")
p3 <- ggplot(mpg, aes(factor(model), cty)) +
        geom_point() +
        theme(axis.text.x =
                element_text(angle = 90, hjust = 1, vjust = 0.5))
```

Next, we compose a plot using as panels the three plots created above (plot not shown).

```
(p1 | p2) / p3
```

We add a title and tag the panels with a letter. In this, and similar cases, parentheses may be needed to alter the default precedence of the R operators.

```
((p1 | p2) / p3) +
  plot_annotation(title = "Fuel use in city traffic:", tag_levels = 'a')
```

Fuel use in city traffic:

a   b

c
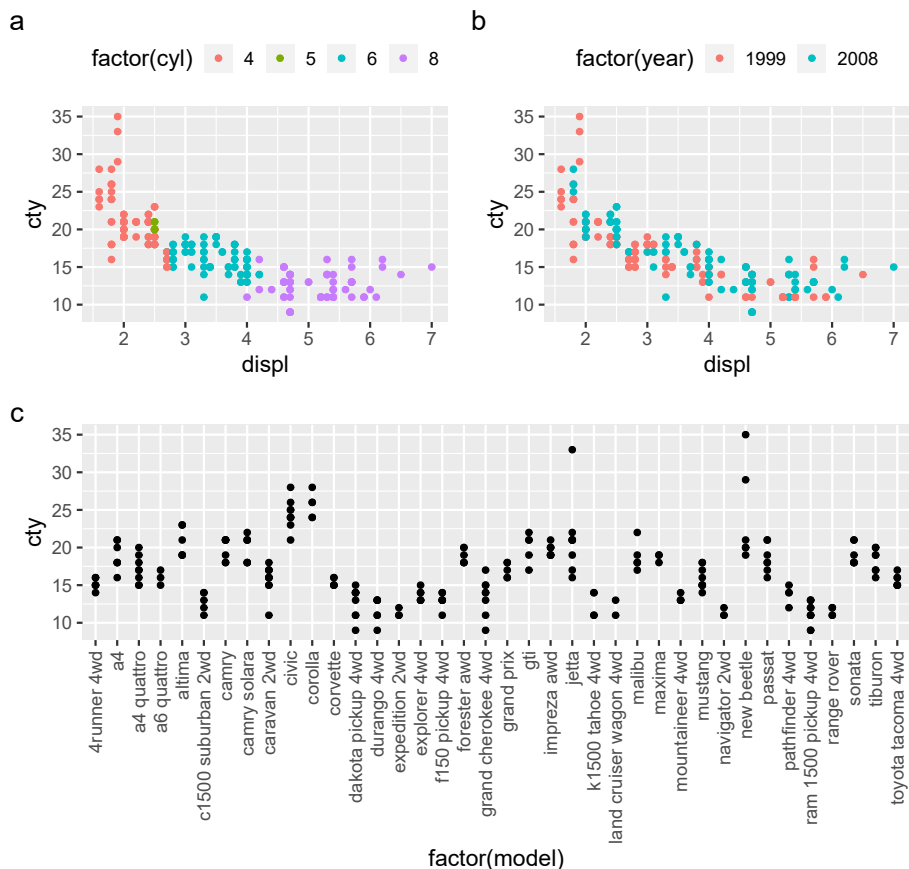
Package 'patchwork' has in recent versions tools for the creation of complex layouts, addition of insets and combining in the same layout plots and other graphic objects such as bitmaps such as photographs and even tables.

1.14   Using `plotmath` expressions

In sections 1.6.1 and 1.5.7 we gave some simple examples of the use of R expressions in plots. The `plotmath` demo and help in R provide enough information to start using expressions in plots. However, composing syntactically correct expressions can be challenging because their syntax is rather unusual. Although expressions are shown here in the context of plotting, they are also used in other contexts in R code.

In general it is possible to create *expressions* explicitly with function `expression()`, or by parsing a character string. In the case of 'ggplot2' for some plot elements, layers created with `geom_text()` and `geom_label()`, and the strip labels of facets the parsing is delayed and applied to mapped character variables

in `data`. In contrast, for titles, subtitles, captions, axis-labels, etc. (anything that is defined within `labs()`) the expressions have to be entered explicitly, or saved as such into a variable, and the variable passed as an argument.

When plotting expressions using `geom_text()`, that character strings are to be parsed is signaled with `parse = TRUE`. In the case of facets' strip labels, parsing or not depends on the *labeller* function used. An additional twist is in this case the possibility of combining static character strings with values taken from `data`.

The most difficult thing to remember when writing expressions is how to connect the different parts. A tilde (~) adds space in between symbols. Asterisk (*) can be also used as a connector, and is needed usually when dealing with numbers. Using space is allowed in some situations, but not in others. To include bits of text within an expression we need to use quotation marks. For a long list of examples have a look at the output and code displayed by `demo(plotmath)` at the R command prompt.

We will use a couple of complex examples to show how to use expressions for different elements of a plot. We first create a data frame, using `paste()` to assemble a vector of subscripted α values as character strings suitable for parsing into expressions.

```r
set.seed(54321) # make sure we always generate the same data
my.data <-
  data.frame(x = 1:5,
             y = rnorm(5),
             greek.label = paste("alpha[", 1:5, "]", sep = ""))
```

We use as x-axis label, a Greek α character with i as subscript, and in the y-axis label, we have a superscript in the units. For the title we use a character string but for the subtitle a rather complex expression. We create these expressions with function `expression()`.

We label each observation with a subscripted *alpha*. We cannot pass expressions to *geometries* by simply mapping them to the label aesthetic. Instead, we pass character strings that can be parsed into expressions. In other words, character strings, that are written using the syntax of expressions. We need to set `parse = TRUE` in the call to the *geometry* so that the strings, instead of being plotted as is, are parsed into expressions before the plot is rendered.

```r
ggplot(my.data, aes(x, y, label = greek.label)) +
  geom_point() +
  geom_text(angle = 45, hjust = 1.2, parse = TRUE) +
  labs(x = expression(alpha[i]),
       y = expression(Speed~~(m~s^{-1})),
       title = "Using expressions",
       subtitle = expression(sqrt(alpha[1] + frac(beta, gamma))))
```

Using expressions

$$\sqrt{\alpha_1 + \frac{\beta}{\gamma}}$$

We can also use a character string stored in a variable, and use function `parse()` to parse it in cases where an expression is required as we do here for `subtitle`. In this example we also set tick labels to expressions, taking advantage that `expression()` accepts multiple arguments separated by commas returning a vector of expressions.

```
my_eq.char <- "alpha[i]"
ggplot(my.data, aes(x, y)) +
   geom_point() +
   labs(title = parse(text = my_eq.char)) +
   scale_x_continuous(name = expression(alpha[i]),
                      breaks = c(1,3,5),
                      labels = expression(alpha[1], alpha[3], alpha[5]))
```
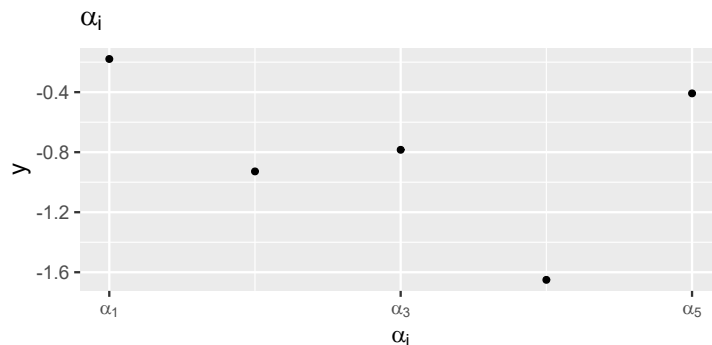

A different approach (no example shown) would be to use `parse()` explicitly for each individual label, something that might be needed if the tick labels need to be "assembled" programmatically instead of set as constants.

⬚ **Differences between `parse()` and `expression()`.** Function `parse()` takes as an argument a character string. This is very useful as the character string can be created programmatically. When using `expression()` this is not possible, except for substitution at execution time of the value of variables into the expression. See the help pages for both functions.

Function `expression()` accepts its arguments without any delimiters. Function `parse()` takes a single character string as an argument to be parsed, in which case quotation marks within the string need to be *escaped* (using \" where a literal " is desired). We can, also in both cases, embed a character string by means of one of the functions `plain()`, `italic()`, `bold()` or `bolditalic()` which also affect the font

used. The argument to these functions needs to be a character string delimited by quotation marks if it is not to be parsed.

When using `expression()`, bare quotation marks can be embedded,

```
ggplot(cars, aes(speed, dist)) +
  geom_point() +
  xlab(expression(x[1]*"  test"))
```

while in the case of `parse()` they need to be *escaped*,

```
ggplot(cars, aes(speed, dist)) +
  geom_point() +
  xlab(parse(text = "x[1]*\"  test\""))
```

and in some cases will be enclosed within a format function.

```
ggplot(cars, aes(speed, dist)) +
  geom_point() +
  xlab(parse(text = "x[1]*italic(\"  test\")"))
```

Some additional remarks. If `expression()` is passed multiple arguments, it returns a vector of expressions. Where `ggplot()` expects a single value as an argument, as in the case of axis labels, only the first member of the vector will be used.

```
ggplot(cars, aes(speed, dist)) +
  geom_point() +
  xlab(expression(x[1], "  test"))
```

Depending on the location within a expression, spaces maybe ignored, or illegal. To juxtapose elements without adding space use `*`, to explicitly insert white space, use `~`. As shown above, spaces are accepted within quoted text. Consequently, the following alternatives can also be used.

```
  xlab(parse(text = "x[1]~~~\"test\""))
```

```
  xlab(parse(text = "x[1]~~~plain(test)"))
```

However, unquoted white space is discarded.

```
  xlab(parse(text = "x[1]*plain(   test)"))
```

Finally, it can be surprising that trailing zeros in numeric values appearing within an expression or text to be parsed are dropped. To force the trailing zeros to be retained we need to enclose the number in quotation marks so that it is interpreted as a character string.

```
ggplot(cars, aes(speed, dist)) +
  geom_point() +
  annotate(geom = "text",
           x = rep(6, 3), y = c(90, 100, 110),
           label = c("'1.00'*x^2", "1.00*x^2", "1.01*x^2"), parse = TRUE)
```

Above we used `paste()` to insert values stored in a variable; functions `format()`, `sprintf()`, and `strftime()` allow the conversion into character strings of other values. These functions can be used when creating plots to generate suitable character strings for the `label` *aesthetic* out of numeric, logical, date, time, and even character values. They can be, for example, used to create labels within a call to `aes()`.

```
sprintf("log(%.3f) = %.3f", 5, log(5))
## [1] "log(5.000) = 1.609"
sprintf("log(%.3g) = %.3g", 5, log(5))
```

```
## [1] "log(5) = 1.61"
```

⌨ **1.30** Study the chunk above. If you are familiar with `C` or `C++` function `sprintf()` will already be familiar to you, otherwise study its help page.

Play with functions `format()`, `sprintf()`, and `strftime()`, formatting different types of data, into character strings of different widths, with different numbers of digits, etc.

It is also possible to substitute the value of variables or, in fact, the result of evaluation, into a new expression, allowing on the fly construction of expressions. Such expressions are frequently used as labels in plots. This is achieved through use of *quoting* and *substitution*.

We use `bquote()` to substitute variables or expressions enclosed in `.( )` by their value. Be aware that the argument to `bquote()` needs to be written as an expression; in this example we need to use a tilde, `~`, to insert a space between words. Furthermore, if the expressions include variables, these will be searched for in the environment rather than in `data`, except within a call to `aes()`.

```
ggplot(cars, aes(speed, dist)) +
  geom_point() +
  labs(title = bquote(Time~zone: .(Sys.timezone())),
       subtitle = bquote(Date: .(as.character(today()))))
       )
```
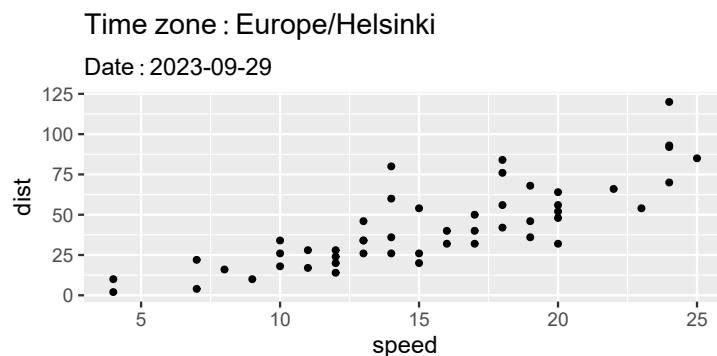

In the case of `substitute()` we supply what is to be used for substitution through a named list.

```
ggplot(cars, aes(speed, dist)) +
  geom_point() +
  labs(title = substitute(Time~zone: tz, list(tz = Sys.timezone())),
       subtitle = substitute(Date: date, list(date = as.character(today()))))
       )
```

Time zone : Europe/Helsinki

Date : 2023-09-29
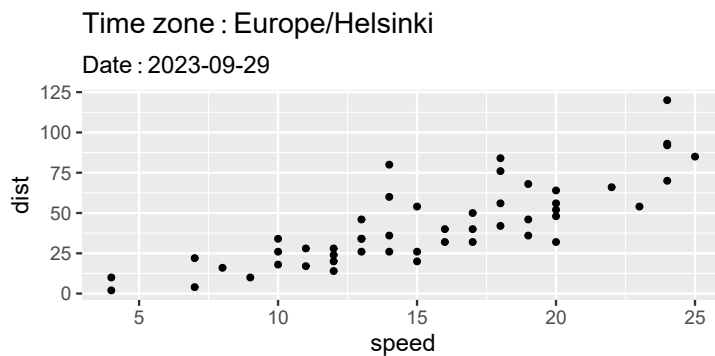
For example, substitution can be used to assemble an expression within a function based on the arguments passed. One case of interest is to retrieve the name of the object passed as an argument, from within a function.

```r
deparse_test <- function(x) {
  print(deparse(substitute(x)))
}

a <- "saved in variable"

deparse_test("constant")
## [1] "\"constant\""
deparse_test(1 + 2)
## [1] "1 + 2"
deparse_test(a)
## [1] "a"
```

Package 'ggtext' provides rich-text (basic HTML and Markdown) support for 'ggplot2', both for annotations and for data visualization. This is an alternative to the use of R expressions.

1.15 Creating complex data displays

The grammar of graphics allows one to build and test plots incrementally. In daily use, when creating a completely new plot, it is best to start with a simple design for a plot, print() this plot, checking that the output is as expected and the code error-free. Afterwards, one can map additional *aesthetics* and add *geometries* and *statistics* gradually. The final steps are then to add *annotations* and the text or expressions used for titles, and axis and key labels. Another approach is to start with an existing plot and modify it, e.g., by using the same plotting code with different data or mapping different variables. When reusing code for a different data set, scale limits and names are likely to need to be edited.

⌨ **1.31** Build a graphically complex data plot of your interest, step by step. By step by step, I do not refer to using the grammar in the construction of the plot as earlier, but of taking advantage of this modularity to test intermediate versions in an iterative design process, first by building up the complex plot in stages as

a tool in debugging, and later using iteration in the processes of improving the graphic design of the plot and improving its readability and effectiveness.

1.16 Creating sets of plots

Plots to be presented at a given occasion or published as part of the same work need to be consistent in various respects: themes, scales and palettes, annotations, titles and captions. To guarantee this consistency we need to build plots modularly and avoid repetition by assigning names to the "modules" that need to be used multiple times.

1.16.1 Saving plot layers and scales in variables

When creating plots with 'ggplot2', objects are composed using operator + to assemble together the individual components. The functions that create plot layers, scales, etc. are constructors of objects and the objects they return can be stored in variables, and once saved, added to multiple plots at a later time.

We create a plot and save it to variable `myplot` and we separately save the values returned by a call to function `labs()`.

```r
myplot <- ggplot(data = mtcars,
                 aes(x = disp, y = mpg,
                 color = factor(cyl))) +
          geom_point()

mylabs <- labs(x = "Engine displacement)",
               y = "Gross horsepower",
               color = "Number of\ncylinders",
               shape = "Number of\ncylinders")
```

We assemble the final plot from the two parts we saved into variables. This is useful when we need to create several plots ensuring that scale `name` arguments are used consistently. In the example above, we saved these names, but the approach can be used for other plot components or lists of components.

⚠ When composing plots with the + operator, the left-hand-side operand must be a `"gg"` object. The left operand is added to the `"gg"` object and the result returned.

```r
myplot
myplot + mylabs + theme_bw(16)
myplot + mylabs + theme_bw(16) + ylim(0, NA)
```

We can also save intermediate results.

```r
mylogplot <- myplot + scale_y_log10(limits=c(8,55))
mylogplot + mylabs + theme_bw(16)
```

1.16.2 Saving plot layers and scales in lists

If the pieces to be put together do not include a `"gg"` object, we can group them into an R list and save it. When we later add the saved list to a `"gg"` object, the members of the list are added one by one to the plot respecting their order.

```r
myparts <- list(mylabs, theme_bw(16))
mylogplot + myparts
```

⌨ **1.32** Revise the code you wrote for the "playground" exercise in section 1.15, but this time, pre-building and saving groups of elements that you expect to be useful unchanged when composing a different plot of the same type, or a plot of a different type from the same data.

1.16.3 Using functions as building blocks

When the blocks we assemble need to accept arguments when used, we have to define functions instead of saving plot components to variables. The functions we define, have to return a `"gg"` object, a list of plot components, or a single plot component. The simplest use is to alter some defaults in existing constructor functions returning `"gg"` objects or layers. The ellipsis (`...`) allows passing named arguments to a nested function. In this case, every single argument passed by name to `bw_ggplot()` will be copied as argument to the nested call to `ggplot()`. Be aware, that supplying arguments by position, is possible only for parameters explicitly included in the definition of the wrapper function,

```r
bw_ggplot <- function(...) {
  ggplot(...) +
  theme_bw()
}
```

which could be used as follows.

```r
bw_ggplot(data = mtcars,
          aes(x = disp, y = mpg,
          color = factor(cyl))) +
          geom_point()
```

1.17 Generating output files

It is possible, when using RStudio, to directly export the displayed plot to a file using a menu. However, if the file will have to be generated again at a later time, or a series of plots need to be produced with consistent format, it is best to include the commands to export the plot in the script.

In R, files are created by printing to different devices. Printing is directed to a currently open device such a window in RStudio. Some devices produce screen output, others files. Devices depend on drivers. There are both devices that are part of R and additional ones defined in contributed packages.

Creating a file involves opening a device, printing and closing the device in sequence. In most cases the file remains locked until the device is close.

For example when rendering a plot to PDF, Encapsulated Postcript, SVG or other vector graphics formats, arguments passed to `width` and `height` are expressed in inches.

```r
fig1 <- ggplot(data.frame(x = -3:3), aes(x = x)) +
  stat_function(fun = dnorm)
pdf(file = "fig1.pdf", width = 8, height = 6)
print(fig1)
dev.off()
```

For Encapsulated Postscript and SVG output, we only need to substitute `pdf()` with `postscript()` or `svg()`, respectively.

```r
postscript(file = "fig1.eps", width = 8, height = 6)
print(fig1)
dev.off()
```

In the case of graphics devices for file output in BMP, JPEG, PNG and TIFF bitmap formats, arguments passed to `width` and `height` are expressed in pixels.

```r
tiff(file = "fig1.tiff", width = 1000, height = 800)
print(fig1)
dev.off()
```

⚠ Some graphics devices are part of base-R, and others are implemented in contributed packages. In some cases, there are multiple graphic device available for rendering graphics in a given file format. These devices usually use different libraries, or have been designed with different aims. These alternative graphic devices can also differ in their function signature, i.e., have differences in the parameters and their names. In cases when rendering fails inexplicably, it can be worthwhile to switch to an alternative graphics device to find out if the problem is in the plot or in the rendering engine.

1.18   Debugging ggplots

Package 'gginnards' provides an enhanced `str()` methods, `num_layers()`, `top_layer()`, `bottom_layer()`, and `mapped_vars()`.

Package 'ggtrace' xxxx

1.19   Further reading

An in-depth discussion of the many extensions to package 'ggplot2' is outside the scope of this book. Several books describe in detail the use of 'ggplot2', being *ggplot2: Elegant Graphics for Data Analysis* (Wickham and Sievert 2016) the one written by the main author of the package. For inspiration or worked out examples,

the book *R Graphics Cookbook* (Chang 2018) is an excellent reference. In depth explanations of the technical aspects of R graphics are available in the book *R Graphics* (Murrell 2019).

```
## Error : package 'ggplot2' is required by 'ggpp' so will not be detached
```

Bibliography

Chang, W. (2018). *R Graphics Cookbook*. 2nd ed. O'Reilly UK Ltd. ISBN: 1491978600 (cit. on pp. 91, 103).

Cleveland, W. S. (1985). *The Elements of Graphing Data*. Wadsworth, Inc. ISBN: 978-0534037291 (cit. on p. 3).

Holmes, S. and W. Huber (2019). *Modern Statistics for Modern Biology*. Cambridge University Press. 382 pp. ISBN: 1108705294 (cit. on p. 60).

Koponen, J. and J. Hildén (2019). *Data visualization handbook*. Espoo, Finland: Aalto University. ISBN: 9789526074498 (cit. on p. 2).

Murrell, P. (2011). *R Graphics*. 2nd ed. Chapman and Hall/CRC, p. 546. ISBN: 1439831769 (cit. on p. 1).

— (2019). *R Graphics*. 3rd ed. Portland: Chapman and Hall/CRC. 423 pp. ISBN: 1498789056 (cit. on p. 103).

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. 1st ed. Springer, p. 268. ISBN: 0387759689 (cit. on p. 1).

Tufte, E. R. (1983). *The Visual Display of Quantitative Information*. Cheshire, CT: Graphics Press. 197 pp. ISBN: 0-9613921-0-X (cit. on p. 47).

Wickham, H. and C. Sievert (2016). *ggplot2: Elegant Graphics for Data Analysis*. 2nd ed. Springer. XVI + 260. ISBN: 978-3-319-24277-4 (cit. on pp. 1, 91, 102).

Zachry, M. and C. Thralls (Oct. 2004). "An Interview with Edward R. Tufte". In: *Technical Communication Quarterly* 13.4, pp. 447–462. DOI: 10.1207/s15427625tcq1304_5.

General Index

Alphabetic Index of R Names

Index of R Names by Category

R names and symbols grouped into the categories 'classes and modes', 'constant and special values', 'control of execution', 'data objects', 'functions and methods', 'names and their scope', and 'operators'.