Chapter 4

Build a plot layer by layer

4.1 Introduction

Layering is the mechanism by which additional data elements are added to a plot. Each layer can come from a different dataset and have a different aesthetic mapping, allowing us to create plots that could not be generated using `qplot()`, which permits only a single dataset and a single set of aesthetic mappings.

This chapter is mainly a technical description of how layers, geoms, statistics and position adjustments work: how you call and customise them. The next chapter, the "toolbox", describes how you can use different geoms and stats to solve particular visualisation problems. These two chapters are companions, with this chapter explaining the theory and the next chapter explaining the practical aspects of using layers to achieve your graphical goals.

Section 4.2 will teach you how to initialise a plot object by hand, a task that `qplot()` performs for us. The plot is not ready to be displayed until at least one layer is added, as described in Section 4.3. This section first describes the complete layer specification, which helps you see exactly how the components of the grammar are realised in R code, and then shows you the shortcuts that will save you a lot of time. As you have learned in the previous chapter, there are five components of a layer:

- The data, § 4.4, which must be an R data frame, and can be changed after the plot is created.
- A set of aesthetic mappings, § 4.5, which describe how variables in the data are mapped to aesthetic properties of the layer. This section includes a description of how layer settings override the plot defaults, the difference between setting and mapping, and the important group aesthetic.
- The geom, § 4.6, which describes the geometric used to draw the layer. The geom defines the set of available aesthetic properties.
- The stat, § 4.7, which takes the raw data and transforms it in some useful way. The stat returns a data frame with new variables that can also be mapped to aesthetics with a special syntax.

H. Wickham, *ggplot2*, Use R, DOI 10.1007/978-0-387-98141-3_4,

- The position adjustment, § 4.8, which adjusts elements to avoid overplotting.

To conclude, Section 4.9 shows you some plotting techniques that pull together everything you have learned in this chapter to create novel visualisations and to visualise model information along with your data.

4.2 Creating a plot

When we used `qplot()`, it did a lot of things for us: it created a plot object, added layers, and displayed the result, using many default values along the way. To create the plot object ourselves, we use `ggplot()`. This has two arguments: **data** and aesthetic **mapping**. These arguments set up defaults for the plot and can be omitted if you specify data and aesthetics when adding each layer. The data argument needs little explanation: It's the data frame that you want to visualise. You are already familiar with aesthetic mappings from `qplot()`, and the syntax here is quite similar, although you need to wrap the pairs of aesthetic attribute and variable name in the `aes()` function. `aes()` is described more fully in Section 4.5, but it's not very tricky. The following example specifies a default mapping of x to carat, y to price and colour to cut.

```
p <- ggplot(diamonds, aes(carat, price, colour = cut))
```

This plot object cannot be displayed until we add a layer: there is nothing to see!

4.3 Layers

A minimal layer may do nothing more than specify a **geom**, a way of visually representing the data. If we add a point geom to the plot we just created, we create a scatterplot, which can then be rendered.

```
p <- p + layer(geom = "point")
```

Note how we use `+` to **add** the layer to the plot. This layer uses the plot defaults for data and aesthetic mapping and it uses default values for two optional arguments: the statistical transformation (the stat) and the position adjustment. A more fully specified layer can take any or all of these arguments:

```
layer(geom, geom_params, stat, stat_params, data, mapping,
  position)
```

Here is what a more complicated call looks like. It produces a histogram (a combination of bars and binning) coloured "steelblue" with a bin width of 2:

```
p <- ggplot(diamonds, aes(x = carat))
p <- p + layer(
  geom = "bar",
  geom_params = list(fill = "steelblue"),
  stat = "bin",
  stat_params = list(binwidth = 2)
)
p
```

This layer specification is precise but verbose. We can simplify it by using shortcuts that rely on the fact that every geom is associated with a default statistic and position, and every statistic with a default geom. This means that you only need to specify one of `stat` or `geom` to get a completely specified layer, with parameters passed on to the geom or stat as appropriate. This expression generates the same layer as the full layer command above:

```
geom_histogram(binwidth = 2, fill = "steelblue")
```

All the shortcut functions have the same basic form, beginning with `geom_` or `stat_`:

```
geom_XXX(mapping, data, ..., geom, position)
stat_XXX(mapping, data, ..., stat, position)
```

Their common parameters define the components of the layer:

- **mapping** (optional): A set of aesthetic mappings, specified using the `aes()` function and combined with the plot defaults as described in Section 4.5.
- **data** (optional): A dataset which overrides the default plot dataset. It is most commonly omitted, in which case the layer will use the default plot data. See Section 4.4.
- **...**: Parameters for the geom or stat, such as bin width in the histogram or bandwidth for a loess smoother. You can also use aesthetic properties as parameters. When you do this you **set** the property to a fixed value, not **map** it to a variable in the dataset. The example above showed setting the fill colour of the histogram to "steelblue". See Section 4.5.2 for more examples.
- **geom** or **stat** (optional): You can override the default `stat` for a `geom`, or the default `geom` for a `stat`. This is a text string containing the name of the geom to use. Using the default will give you a standard plot; overriding the default allows you to achieve something more exotic, as shown in Section 4.9.1.
- **position** (optional): Choose a method for adjusting overlapping objects, as described in Section 4.8.

Note that the order of `data` and `mapping` arguments is switched between `ggplot()` and the layer functions. This is because you almost always specify

data for the plot, and almost always specify aesthetics—but *not* data—for the layers. We suggest explicitly naming all other arguments rather than relying on positional matching. This makes the code more readable and is the style followed in this book.

Layers can be added to plots created with `ggplot()` or `qplot()`. Remember, behind the scenes, `qplot()` is doing exactly the same thing: it creates a plot object and then adds layers. The following example shows the equivalence between these two ways of making plots.

```
ggplot(msleep, aes(sleep_rem / sleep_total, awake)) +
  geom_point()
# which is equivalent to
qplot(sleep_rem / sleep_total, awake, data = msleep)

# You can add layers to qplot too:
qplot(sleep_rem / sleep_total, awake, data = msleep) +
  geom_smooth()
# This is equivalent to
qplot(sleep_rem / sleep_total, awake, data = msleep,
  geom = c("point", "smooth"))
# or
ggplot(msleep, aes(sleep_rem / sleep_total, awake)) +
  geom_point() + geom_smooth()
```

You've seen that plot objects can be stored as variables. The summary function can be helpful for inspecting the structure of a plot without plotting it, as seen in the following example. The summary shows information about the plot defaults, and then each layer. You will learn about scales and faceting in Chapters 6 and 7.

```
> p <- ggplot(msleep, aes(sleep_rem / sleep_total, awake))
> summary(p)
data: name, genus, vore, order, conservation,
  sleep_total, sleep_rem, sleep_cycle, awake,
  brainwt, bodywt [83x11]
mapping:  x = sleep_rem/sleep_total, y = awake
scales:   x, y
faceting: facet_grid(. ~ ., FALSE)
>
> p <- p + geom_point()
> summary(p)
data: name, genus, vore, order, conservation,
  sleep_total, sleep_rem, sleep_cycle, awake,
  brainwt, bodywt [83x11]
mapping:  x = sleep_rem/sleep_total, y = awake
scales:   x, y
```

```
faceting: facet_grid(. ~ ., FALSE)
-----------------------------------
geom_point: na.rm = FALSE
stat_identity:
position_identity: (width = NULL, height = NULL)
```

Layers are regular R objects and so can be stored as variables, making it easy to write clean code that reduces duplication. For example, a set of plots can be initialised using different data then enhanced with the same layer. If you later decide to change that layer, you only need to do so in one place. The following shows a simple example, where we create a layer that displays a translucent thick blue line of best fit.

```
bestfit <- geom_smooth(method = "lm", se = F,
  colour = alpha("steelblue", 0.5), size = 2)
qplot(sleep_rem, sleep_total, data = msleep) + bestfit
qplot(awake, brainwt, data = msleep, log = "y") + bestfit
qplot(bodywt, brainwt, data = msleep, log = "xy") + bestfit
```

The following sections describe data and mappings in more detail, then go on to describe the available geoms, stats and position adjustments.

4.4 Data

The restriction on the data is simple: it must be a data frame. This is restrictive, and unlike other graphics packages in R. Lattice functions can take an optional data frame or use vectors directly from the global environment. Base methods often work with vectors, data frames or other R objects. However, there are good reasons for this restriction. Your data is very important, and it's better to be explicit about exactly what is done with it. It also allows a cleaner separation of concerns so that `ggplot2` deals only with plotting data, not wrangling it into different forms, for which you might find the `plyr` or `reshape` packages helpful. A single data frame is also easier to save than a multitude of vectors, which means it's easier to reproduce your results or send your data to someone else.

This restriction also makes it very easy to produce the same plot for different data: you just change the data frame. You can replace the old dataset with `%+%`, as shown in the following example. (You might expect that this would use `+` like all the other components, but unfortunately due to a restriction in R this is not possible.) Swapping out the data makes it easy to experiment with imputation schemes or model fits, as shown in Section 4.9.3.

```
p <- ggplot(mtcars, aes(mpg, wt, colour = cyl)) + geom_point()
p
```

```
mtcars <- transform(mtcars, mpg = mpg ^ 2)
p %+% mtcars
```

Any change of values or dimensions is legitimate. However, if a variable changes from discrete to continuous (or vice versa), you will need to change the default scales, as described in Section 6.3.

It is not necessary to specify a default dataset except when using faceting; faceting is a global operation (i.e., it works on all layers) and it needs to have a base dataset which defines the set of facets for all datasets. See Section 7.2.4 for more details. If the default dataset is omitted, every layer must supply its own data.

The data is stored in the plot object as a copy, not a reference. This has two important consequences: if your data changes, the plot will not; and `ggplot2` objects are entirely self-contained so that they can be `save()`d to disk and later `load()`ed and plotted without needing anything else from that session.

4.5 Aesthetic mappings

To describe the way that variables in the data are mapped to things that we can perceive on the plot (the "aesthetics"), we use the `aes` function. The `aes` function takes a list of aesthetic-variable pairs like these:

```
aes(x = weight, y = height, colour = age)
```

Here we are mapping x-position to weight, y-position to height and colour to age. The first two arguments can be left without names, in which case they correspond to the x and y variables. This matches the way that `qplot()` is normally used. You should never refer to variables outside of the dataset (e.g., with `diamonds$carat`), as this makes it impossible to encapsulate all of the data needed for plotting in a single object.

```
aes(weight, height, colour = sqrt(age))
```

Note that functions of variables can be used.

Any variable in an `aes()` specification must be contained inside the plot or layer data. This is one of the ways in which `ggplot2` objects are guaranteed to be entirely self-contained, so that they can be stored and re-used.

4.5.1 Plots and layers

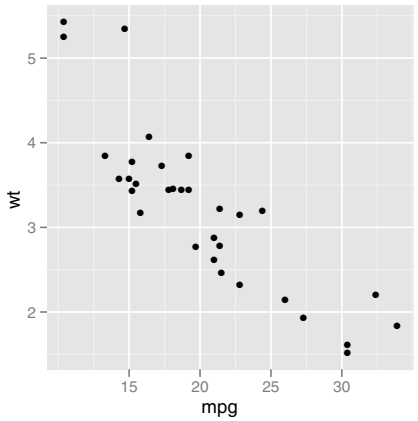
The default aesthetic mappings can be set when the plot is initialised or modified later using `+`, as in this example:

```
> p <- ggplot(mtcars)
> summary(p)
data: mpg, cyl, disp, hp, drat, wt, qsec, vs, am,
```

```
  gear, carb [32x11]
faceting: facet_grid(. ~ ., FALSE)
>
> p <- p + aes(wt, hp)
> summary(p)
data: mpg, cyl, disp, hp, drat, wt, qsec, vs, am,
  gear, carb [32x11]
mapping:  x = wt, y = hp
scales:   list(), list()
faceting: facet_grid(. ~ ., FALSE)
```

One reason you might want to do this is shown in Section 4.9.3. We have seen several examples of using the default mapping when adding a layer to a plot:

```
> p <- ggplot(mtcars, aes(x = mpg, y = wt))
> p + geom_point()
```



The default mappings in the plot `p` can be extended or overridden in the layers, as with the following code. The results are shown in Figure 4.1.

```
p + geom_point(aes(colour = factor(cyl)))
p + geom_point(aes(y = disp))
```

The rules are summarised in Table 4.1. Aesthetic mappings specified in a layer affect only that layer. For that reason, unless you modify the default scales, axis labels and legend titles will be based on the plot defaults. The way to change these is described in Section 6.5.

4.5.2 Setting vs. mapping

Instead of mapping an aesthetic property to a variable, you can set it to a single value by specifying it in the layer parameters. Aesthetics can vary

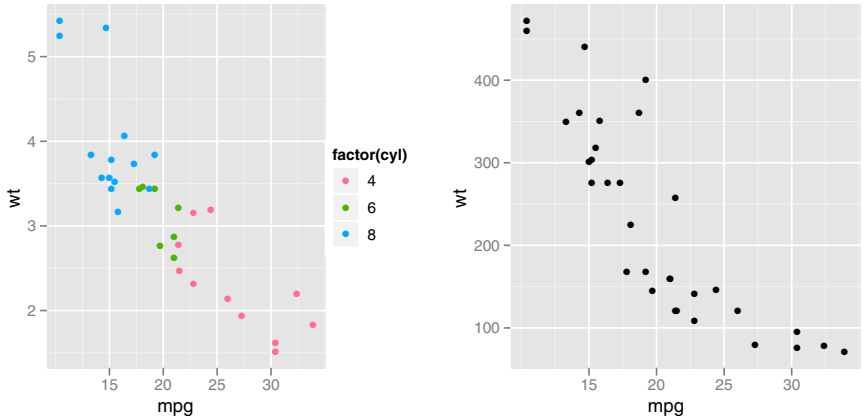


Fig. 4.1: Overriding aesthetics. (Left) Overriding colour with `factor(cyl)` and (right) overriding y-position with `disp`

Operation	Layer aesthetics	Result
Add	`aes(colour = cyl)`	`aes(mpg, wt, colour = cyl)`
Override	`aes(y = disp)`	`aes(mpg, disp)`
Remove	`aes(y = NULL)`	`aes(mpg)`

Table 4.1: Rules for combining layer mappings with the default mapping of `aes(mpg, wt)`. Layer aesthetics can add to, override, and remove the default mappings.

for each observation being plotted, while parameters do not. We **map** an aesthetic to a variable (e.g., (aes(colour = cut))) or **set** it to a constant (e.g., `colour = "red"`). For example, the following layer sets the colour of the points, using the colour parameter of the layer:

```
p <- ggplot(mtcars, aes(mpg, wt))
p + geom_point(colour = "darkblue")
```

This sets the point colour to be dark blue instead of black. This is quite different than

```
p + geom_point(aes(colour = "darkblue"))
```

This **maps** (not sets) the colour to the value "darkblue". This effectively creates a new variable containing only the value "darkblue" and then maps colour to that new variable. Because this value is discrete, the default colour scale uses evenly spaced colours on the colour wheel, and since there is only one value this colour is pinkish. The difference between setting and mapping is illustrated in Figure 4.2.

With `qplot()`, you can do the same thing by putting the value inside of `I()`, e.g., `colour = I("darkblue")`. Chapter B describes how values should be specified for the various aesthetics.

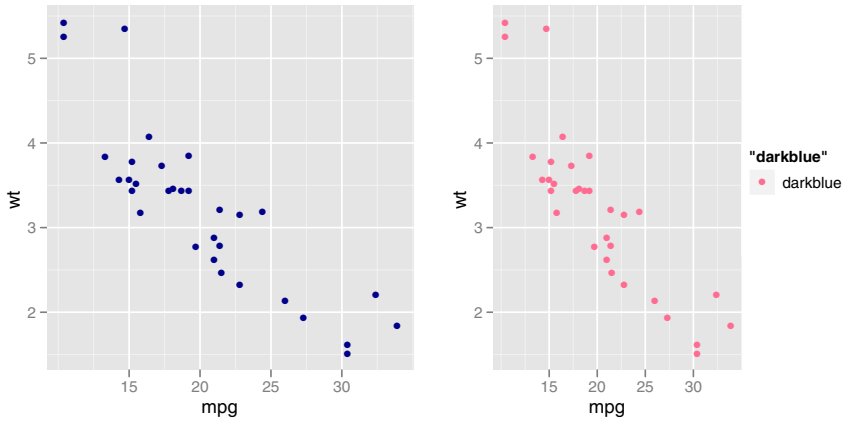


Fig. 4.2: The difference between (left) setting colour to `"darkblue"` and (right) mapping colour to `"darkblue"`. When `"darkblue"` is mapped to colour, it is treated as a regular value and scaled with the default colour scale. This results in pinkish points and a legend.

4.5.3 Grouping

In `ggplot2`, geoms can be roughly divided into individual and collective geoms. An individual geom has a distinctive graphical object for each row in the data frame. For example, the point geom has a single point for each observation. On the other hand, collective geoms represent multiple observations. This may be a result of a statistical summary, or may be fundamental to the display of the geom, as with polygons. Lines and paths fall somewhere in between: each overall line is composed of a set of straight segments, but each segment represents two points. How do we control which observations go in which individual graphical element? This is the job of the `group` aesthetic.

By default, the `group` is set to the interaction of all discrete variables in the plot. This often partitions the data correctly, but when it does not, or when no discrete variable is used in the plot, you will need to explicitly define the grouping structure, by mapping group to a variable that has a different value for each group. The `interaction()` function is useful if a single pre-existing variable doesn't cleanly separate groups, but a combination does.

There are three common cases where the default is not enough, and we will consider each one below. In the following examples, we will use a simple

longitudinal dataset, `Oxboys`, from the `nlme` package. It records the heights (height) and centered ages (age) of 26 boys (Subject), measured on nine occasions (Occasion).

Multiple groups, one aesthetic.

In many situations, you want to separate your data into groups, but render them in the same way. When looking at the data in aggregate you want to be able to distinguish individual subjects, but not identify them. This is common in longitudinal studies with many subjects, where the plots are often descriptively called spaghetti plots.

The first plot in Figure 4.3 shows a set of time series plots, one for each boy. You can see the separate growth trajectories for each boy, but there is no way to see which boy belongs to which trajectory. This plot was generated with:

```
p <- ggplot(Oxboys, aes(age, height, group = Subject)) +
  geom_line()
```

We specified the Subject as the grouping variable to get a line for each boy. The second plot in the figure shows the result of leaving this out: we get a single line which passes through every point. This is not very useful! Line plots with an incorrect grouping specification typically have this characteristic appearance.

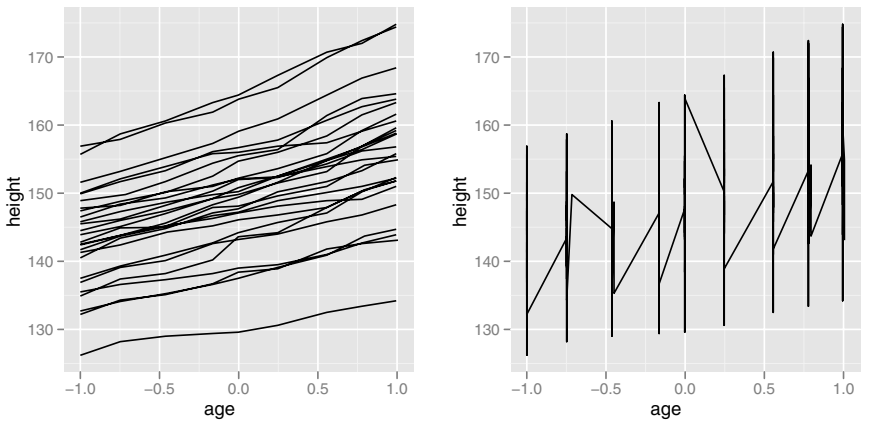


Fig. 4.3: (Left) Correctly specifying `group = Subject` produces one line per subject. (Right) A single line connects all observations. This pattern is characteristic of an incorrect grouping aesthetic, and is what we see if the group aesthetic is omitted, which in this case is equivalent to `group = 1`.

Different groups on different layers.

Sometimes we want to plot summaries based on different levels of aggregation. Different layers might have different group aesthetics, so that some display individual level data while others display summaries of larger groups.

Building on the previous example, suppose we want to add a single smooth line to the plot just created, based on the ages and heights of *all* the boys. If we use the same grouping for the smooth that we used for the line, we get the first plot in Figure 4.4.

```
p + geom_smooth(aes(group = Subject), method="lm", se = F)
```

This is not what we wanted; we have inadvertently added a smoothed line for each boy. This new layer needs a different group aesthetic, `group = 1`, so that the new line will be based on all the data, as shown in the second plot in the figure. The modified layer looks like this:

```
p + geom_smooth(aes(group = 1), method="lm", size = 2, se = F)
```

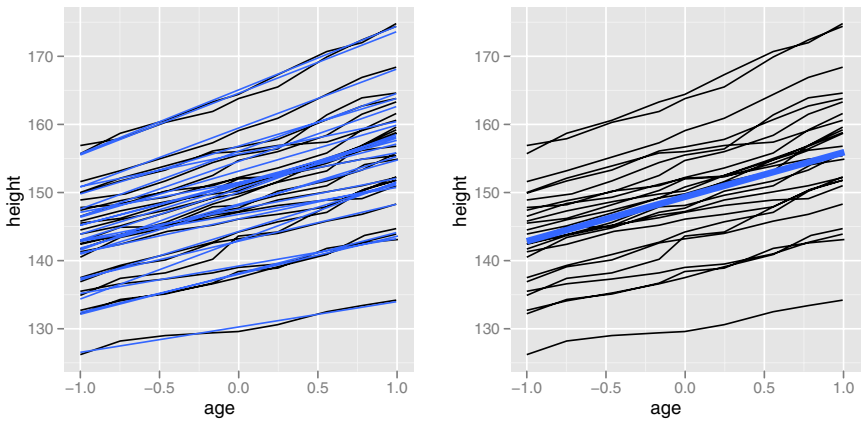


Fig. 4.4: Adding smooths to the Oxboys data. (Left) Using the same grouping as the lines results in a line of best fit for each boy. (Right) Using `aes(group = 1)` in the smooth layer fits a single line of best fit across all boys.

Note how we stored the first plot in the variable `p`, so we could experiment with the code to generate the second layer without having to re-enter any of the code for the first layer. This is a useful time-saving technique, and is expanded upon in Chapter 10.

Overriding the default grouping.

The plot has a discrete scale but you want to draw lines that connect *across* groups. This is the strategy used in interaction plots, profile plots, and parallel coordinate plots, among others. For example, we draw boxplots of height at each measurement occasion, as shown in the first figure in Figure 4.5:

```
boysbox <- ggplot(Oxboys, aes(Occasion, height)) + geom_boxplot()
```

There is no need to specify the group aesthetic here; the default grouping works because occasion is a discrete variable. To overlay individual trajectories we again need to override the default grouping for that layer with `aes(group = Subject)`, as shown in the second plot in the figure.

```
boysbox + geom_line(aes(group = Subject), colour = "#3366FF")
```

We change the line colour in the second layer to make them distinct from the boxes. This is another example of setting an aesthetic to a fixed value. The colour is a rendering attribute, which has no corresponding variable in the data.

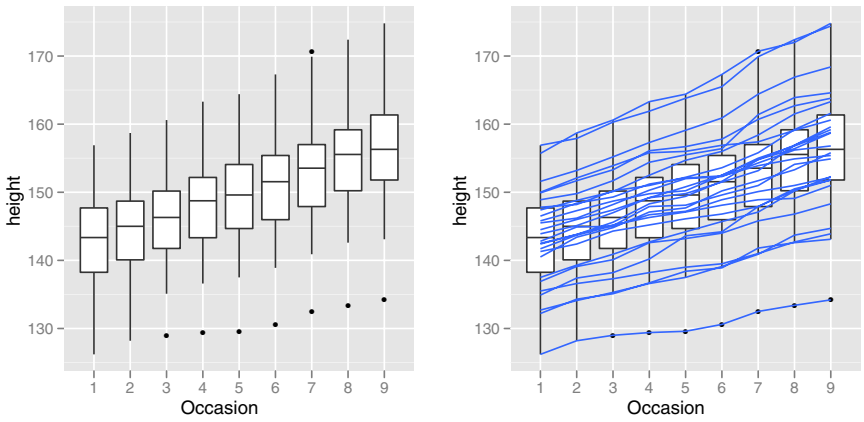


Fig. 4.5: (Left) If boxplots are used to look at the distribution of heights at each occasion (a discrete variable), the default grouping works correctly. (Right) If trajectories of individual boys are overlaid with `geom_line()`, then `aes(group = Subject)` is needed for the new layer.

4.5.4 Matching aesthetics to graphic objects

Another important issue with collective geom is how the aesthetics of the individual observations are mapped to the aesthetics of the complete entity. For

individual geoms, this isn't a problem, because each observation is represented by a single graphical element. However, high data densities can make it difficult (or impossible) to distinguish between individual points and in some sense the point geom becomes a collective geom, a single blob of points.

Lines and paths operate on an off-by-one principle: there is one more observation than line segment, and so the aesthetic for the first observation is used for the first segment, the second observation for the second segment and so on. This means that the aesthetic for the last observation is not used, as shown in Figure 4.6. An additional limitation for paths and lines is that that line type must be constant over each individual line, in R there is no way to draw a joined up line which has varying line type.

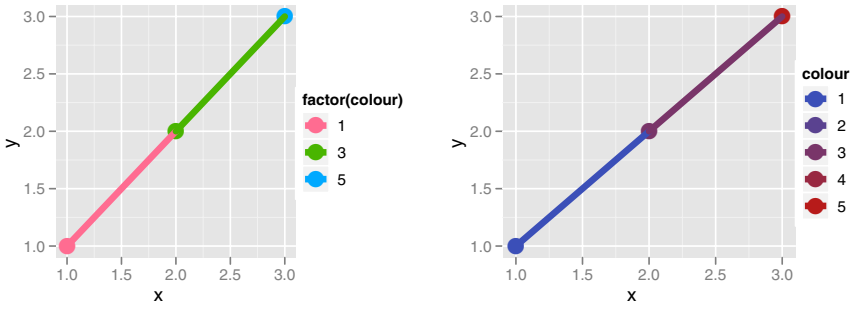
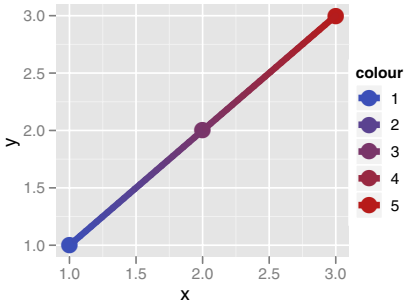


Fig. 4.6: For lines and paths, the aesthetics of the line segment are determined by the aesthetic of the beginning observation. If colour is categorical (left) there is no meaningful way to interpolate between adjacent colours. If colour is continuous (right), there is, but this is not done by default.

You could imagine a more complicated system where segments smoothly blend from one aesthetic to another. This would work for continuous variables like size or colour, but not for line type, and is not used in `ggplot2`. If this is the behaviour you want, you can perform the linear interpolation yourself, as shown below.

```
> xgrid <- with(df, seq(min(x), max(x), length = 50))
> interp <- data.frame(
+   x = xgrid,
+   y = approx(df$x, df$y, xout = xgrid)$y,
+   colour = approx(df$x, df$colour, xout = xgrid)$y
+ )
> qplot(x, y, data = df, colour = colour, size = I(5)) +
+   geom_line(data = interp, size = 2)
```



For all other collective geoms, like polygons, the aesthetics from the individual components are only used if they are all the same, otherwise the default value is used. This makes sense for fill as it is a property of the entire object: it doesn't make sense to think about having a different fill colour for each point on the border of the polygon.

These issues are most relevant when mapping aesthetics to continuous variable, because, as described above, when you introduce a mapping to a discrete variable, it will by default split apart collective geoms into smaller pieces. This works particularly well for bar and area plots, because stacking the individual pieces produces the same shape as the original ungrouped data. This is illustrated in Figure 4.7.

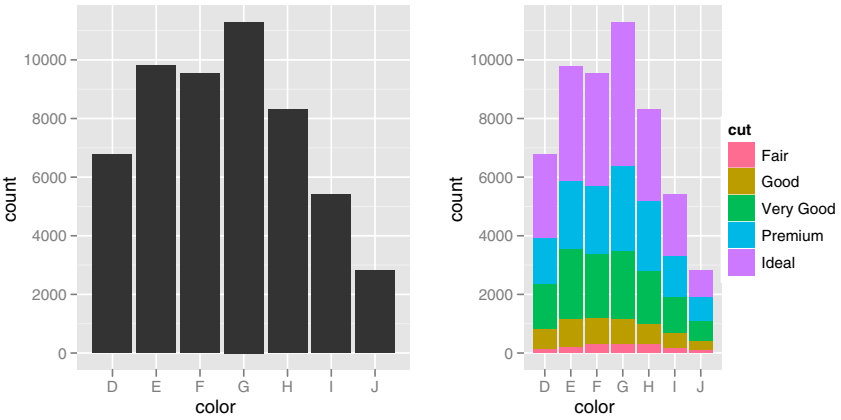


Fig. 4.7: Splitting apart a bar chart (left) produces a plot (right) that has the same outline as the original.

4.6 Geoms

Geometric objects, or **geom**s for short, perform the actual rendering of the layer, control the type of plot that you create. For example, using a point geom will create a scatterplot, while using a line geom will create a line plot. Table 4.2 lists all of the geoms available in `ggplot2`.

Each geom has a set of aesthetics that it understands, and a set that are required for drawing. For example, a point requires x and y position, and understands colour, size and shape aesthetics. A bar requires height (`ymax`), and understands width, border colour and fill colour. These are listed for all geoms in Table 4.3.

Some geoms differ primarily in the way that they are parameterised. For example, the tile geom is specified in terms of the location of its centre and its height and width, while the rect geom is parameterised in terms of its top (`ymax`), bottom (`ymin`), left (`xmin`) and right (`right`) positions. Internally, the rect geom is described as a polygon, and it is parameters are the locations of the four corners. This is useful for non-Cartesian coordinate systems, as you will learn in Chapter 7.

Every geom has a default statistic, and every statistic a default geom. For example, the bin statistic defaults to using the bar geom to produce a histogram. These defaults are listed in Table 4.3. Overriding these defaults will still produce valid plots, but they may violate graphical conventions. See examples in Section 4.9.1.

4.7 Stat

A statistical transformation, or **stat**, transforms the data, typically by summarising it in some manner. For example, a useful stat is the smoother, which calculates the mean of y, conditional on x, subject to some restriction that ensures smoothness. All currently available stats are listed in Table 4.4. To make sense in a graphic context a stat must be location-scale invariant: $\mathrm{f}(x + a) = \mathrm{f}(x) + a$ and $\mathrm{f}(b \cdot x) = b \cdot \mathrm{f}(x)$. This ensures that the transformation stays the same when you change the scales of the plot.

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

- `count`, the number of observations in each bin
- `density`, the density of observations in each bin (percentage of total / bar width)
- `x`, the centre of the bin

These generated variables can be used instead of the variables present in the original dataset. For example, the default histogram geom assigns the

Name	Description
abline	Line, specified by slope and intercept
area	Area plots
bar	Bars, rectangles with bases on y-axis
blank	Blank, draws nothing
boxplot	Box-and-whisker plot
contour	Display contours of a 3d surface in 2d
crossbar	Hollow bar with middle indicated by horizontal line
density	Display a smooth density estimate
density_2d	Contours from a 2d density estimate
errorbar	Error bars
histogram	Histogram
hline	Line, horizontal
interval	Base for all interval (range) geoms
jitter	Points, jittered to reduce overplotting
line	Connect observations, in order of x value
linerange	An interval represented by a vertical line
path	Connect observations, in original order
point	Points, as for a scatterplot
pointrange	An interval represented by a vertical line, with a point in the middle
polygon	Polygon, a filled path
quantile	Add quantile lines from a quantile regression
ribbon	Ribbons, y range with continuous x values
rug	Marginal rug plots
segment	Single line segments
smooth	Add a smoothed condition mean
step	Connect observations by stairs
text	Textual annotations
tile	Tile plot as densely as possible, assuming that every tile is the same size
vline	Line, vertical

Table 4.2: Geoms in `ggplot2`

Name	Default stat	Aesthetics
abline	abline	colour, linetype, size
area	identity	colour, fill, linetype, size, **x**, **y**
bar	bin	colour, fill, linetype, size, weight, **x**
bin2d	bin2d	colour, fill, linetype, size, weight, **xmax**, **xmin**, **ymax**, **ymin**
blank	identity	
boxplot	boxplot	colour, fill, **lower**, **middle**, size, **upper**, weight, **x**, **ymax**, **ymin**
contour	contour	colour, linetype, size, weight, **x**, **y**
crossbar	identity	colour, fill, linetype, size, **x**, **y**, **ymax**, **ymin**
density	density	colour, fill, linetype, size, weight, **x**, **y**
density2d	density2d	colour, linetype, size, weight, **x**, **y**
errorbar	identity	colour, linetype, size, width, **x**, **ymax**, **ymin**
freqpoly	bin	colour, linetype, size
hex	binhex	colour, fill, size, **x**, **y**
histogram	bin	colour, fill, linetype, size, weight, **x**
hline	hline	colour, linetype, size
jitter	identity	colour, fill, shape, size, **x**, **y**
line	identity	colour, linetype, size, **x**, **y**
linerange	identity	colour, linetype, size, **x**, **ymax**, **ymin**
path	identity	colour, linetype, size, **x**, **y**
point	identity	colour, fill, shape, size, **x**, **y**
pointrange	identity	colour, fill, linetype, shape, size, **x**, **y**, **ymax**, **ymin**
polygon	identity	colour, fill, linetype, size, **x**, **y**
quantile	quantile	colour, linetype, size, weight, **x**, **y**
rect	identity	colour, fill, linetype, size, **xmax**, **xmin**, **ymax**, **ymin**
ribbon	identity	colour, fill, linetype, size, **x**, **ymax**, **ymin**
rug	identity	colour, linetype, size
segment	identity	colour, linetype, size, **x**, **xend**, **y**, **yend**
smooth	smooth	alpha, colour, fill, linetype, size, weight, **x**, **y**
step	identity	colour, linetype, size, **x**, **y**
text	identity	angle, colour, hjust, **label**, size, vjust, **x**, **y**
tile	identity	colour, fill, linetype, size, **x**, **y**
vline	vline	colour, linetype, size

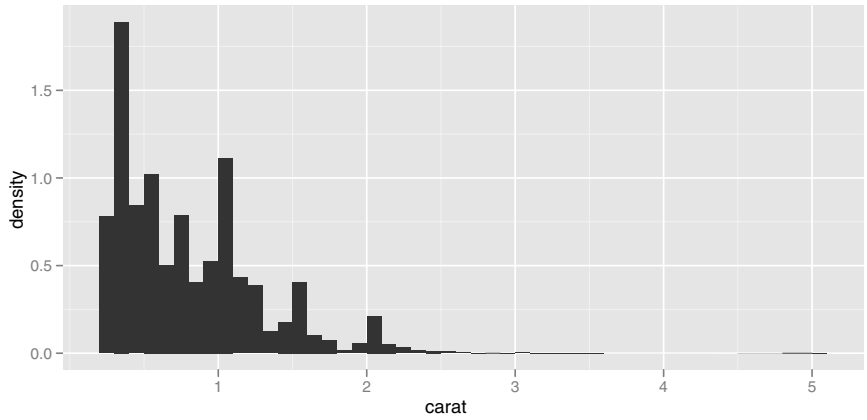
Table 4.3: Default statistics and aesthetics. Emboldened aesthetics are required.

Name	Description
bin	Bin data
boxplot	Calculate components of box-and-whisker plot
contour	Contours of 3d data
density	Density estimation, 1d
density_2d	Density estimation, 2d
function	Superimpose a function
identity	Don't transform data
qq	Calculation for quantile-quantile plot
quantile	Continuous quantiles
smooth	Add a smoother
spoke	Convert angle and radius to xend and yend
step	Create stair steps
sum	Sum unique values. Useful for overplotting on scatterplots
summary	Summarise y values at every unique x
unique	Remove duplicates

Table 4.4: Stats in `ggplot2`

height of the bars to the number of observations (`count`), but if you'd prefer a more traditional histogram, you can use the density (`density`). The following example shows a density histogram of `carat` from the diamonds dataset.

```
> ggplot(diamonds, aes(carat)) +
+   geom_histogram(aes(y = ..density..), binwidth = 0.1)
```



The names of generated variables must be surrounded with `..` when used. This prevents confusion in case the original dataset includes a variable with the same name as a generated variable, and it makes it clear to any later

reader of the code that this variable was generated by a stat. Each statistic lists the variables that it creates in its documentation.

The syntax to produce this plot with `qplot()` is very similar:

```
qplot(carat, ..density.., data = diamonds, geom="histogram",
  binwidth = 0.1)
```

4.8 Position adjustments

Position adjustments apply minor tweaks to the position of elements within a layer. Table 4.5 lists all of the position adjustments available within `ggplot2`. Position adjustments are normally used with discrete data. Continuous data typically doesn't overlap exactly, and when it does (because of high data density) minor adjustments, like jittering, are usually insufficient to fix the problem.

Adjustment	Description
dodge	Adjust position by dodging overlaps to the side
fill	Stack overlapping objects and standardise have equal height
identity	Don't adjust position
jitter	Jitter points to avoid overplotting
stack	Stack overlapping objects on top of one another

Table 4.5: The five position adjustments.

The different types of adjustment are best illustrated with a bar chart. Figure 4.8 shows stacking, filling and dodging. Stacking puts bars on the same x on top of one another; filling does the same, but normalises height to 1; and dodging places the bars side-by-side. Dodging is rather similar to faceting, and the advantages and disadvantages of each method are described in Section 7.2.6. For these operations to work, each bar must have the same width and not overlap with any others. The identity adjustment (i.e., do nothing) doesn't make much sense for bars, but is shown in Figure 4.9 along with a line plot of the same data for reference.

4.9 Pulling it all together

Once you have become comfortable with combining layers, you will be able to create graphics that are both intricate and useful. The following examples demonstrate some of the ways to use the capabilities of layers that have been introduced in this chapter. These are just to get you started. You are limited only by your imagination!

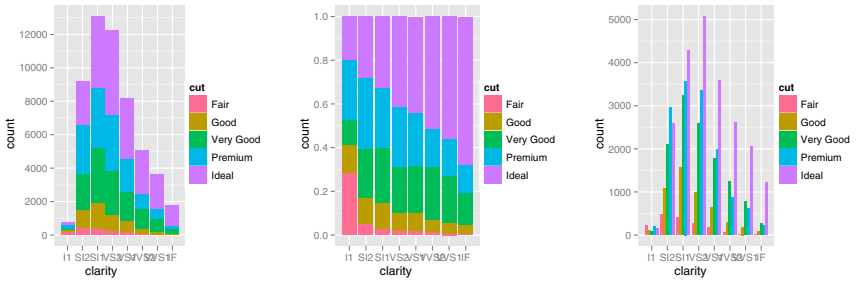


Fig. 4.8: Three position adjustments applied to a bar chart. From left to right, stacking, filling and dodging.

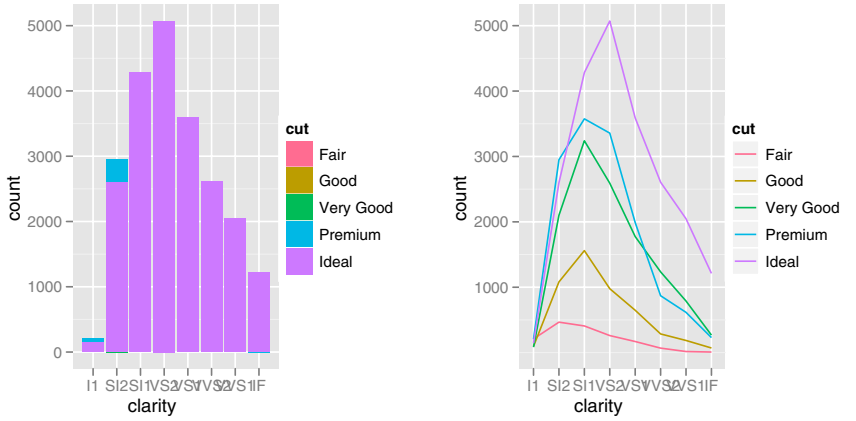


Fig. 4.9: The identity positon adjustment is not useful for bars, (left) because each bar obscures the bars behind. (Right) It is useful for lines, however, because lines do not have the same problem.

4.9.1 Combining geoms and stats

By connecting geoms with different statistics, you can easily create new graphics. Figure 4.10 shows three variations on a histogram. They all use the same statistical transformation underlying a histogram (the bin stat), but use different geoms to display the results: the area geom, the point geom and the tile geom.

```
d <- ggplot(diamonds, aes(carat)) + xlim(0, 3)
d + stat_bin(aes(ymax = ..count..), binwidth = 0.1, geom = "area")
d + stat_bin(
  aes(size = ..density..), binwidth = 0.1,
  geom = "point", position="identity"
```

```
)
d + stat_bin(
  aes(y = 1, fill = ..count..), binwidth = 0.1,
  geom = "tile", position="identity"
)
```

(The use of `xlim()` will be discussed in Section 6.4.2, in the presentation of the use of scales and axes, but you can already guess that it is used here to set the limits of the horizontal axis.)

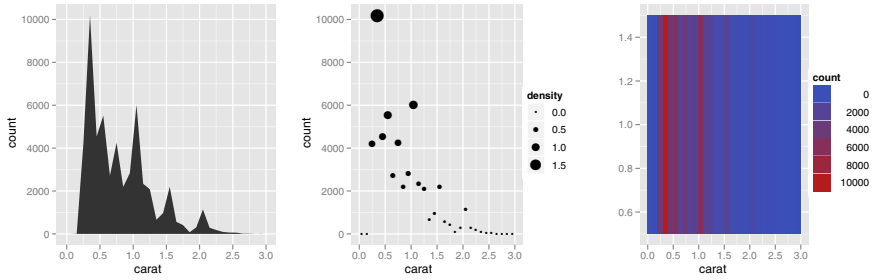


Fig. 4.10: Three variations on the histogram. (Left) A frequency polygon; (middle) a scatterplot with both size and height mapped to frequency; (right) a heatmap representing frequency with colour.

A number of the geoms available in `ggplot2` were derived from other geoms in a process like the one just described, starting with an existing geom and making a few changes in the default aesthetics or stat. For example, the jitter geom is simply the point geom with the default position adjustment set to jitter. Once it becomes clear that a particular variant is going to be used a lot or used in a very different context, it makes sense to create a new geom. Table 4.6 lists these "aliased" geoms.

Aliased geom	Base geom	Changes in default
area	ribbon	`aes(min = 0, max = y), position = "stack"`
density	area	`stat = "density"`
freqpoly	line	`stat = "bin"`
histogram	bar	`stat = "bin"`
jitter	point	`position = "jitter"`
quantile	line	`stat = "quantile"`
smooth	ribbon	`stat = "smooth"`

Table 4.6: Geoms that were created by modifying the defaults of another geom.

4.9.2 Displaying precomputed statistics

If you have data which has already been summarised, and you just want to use it, you'll need to use `stat_identity()`, which leaves the data unchanged, and then map the appropriate variables to the appropriate aesthetics.

4.9.3 Varying aesthetics and data

One of the more powerful capabilities of `ggplot2` is the ability to plot different datasets on different layers. This may seem strange: Why would you want to plot different data on the same plot? In practice, you often have related datasets that should be shown together. A very common example is supplementing the data with predictions from a model. While the smooth geom can add a wide range of different smooths to your plot, it is no substitute for an external quantitative model that summarises your understanding of the data.

Let's look again at the `Oxboys` dataset which was used in Section 4.5.3. In Figure 4.4, we showed linear fits for individual boys (left) and for the whole group (right). Neither model is particularly appropriate: The group model ignores the within-subject correlation and the individual model doesn't use information about the typical growth pattern to more accurately predict individuals. In practice we might use a mixed model to do better. This section explores how we can combine the output from this more sophisticated model with the original data to gain more insight into both the data and the model.

First we'll load the `nlme` package, and fit a model with varying intercepts and slopes. (Exploring the fit of individual models shows that this is a reasonable first pass.) We'll also create a plot to use as a template. This regenerates the first plot in Figure 4.3, but we're not going to render it until we've added data from the model.

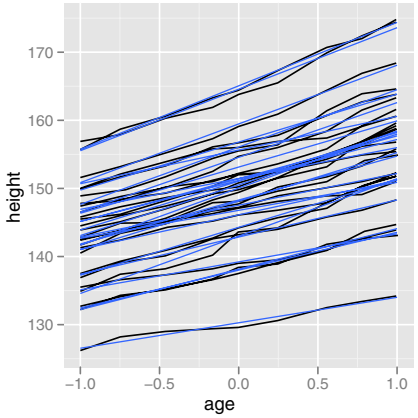
```
> require(nlme, quiet = TRUE, warn.conflicts = FALSE)
> model <- lme(height ~ age, data = Oxboys,
+  random = ~ 1 + age | Subject)
> oplot <- ggplot(Oxboys, aes(age, height, group = Subject)) +
+   geom_line()
```

Next we'll compare the predicted trajectories to the actual trajectories. We do this by building up a grid that contains all combinations of ages and subjects. This is overkill for this simple linear case, where we only need two values of age to draw the predicted straight line, but we show it here because it is necessary when the model is more complex. Next we add the predictions from the model back into this dataset, as a variable called height.

```
> age_grid <- seq(-1, 1, length = 10)
> subjects <- unique(Oxboys$Subject)
>
> preds <- expand.grid(age = age_grid, Subject = subjects)
> preds$height <- predict(model, preds)
```

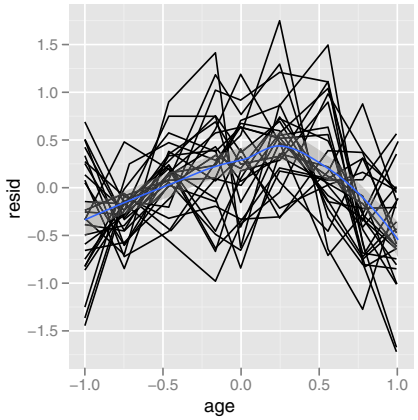
Once we have the predictions we can display them along with the original data. Because we have used the same variable names as the original `Oxboys` dataset, and we want the same group aesthetic, we don't need to specify any aesthetics; we only need to override the default dataset. We also set two aesthetic parameters to make it a bit easier to compare the predictions to the actual values.

```
> oplot + geom_line(data = preds, colour = "#3366FF", size= 0.4)
```



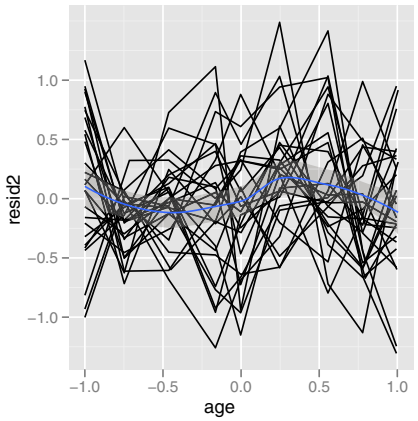
It seems that the model does a good job of capturing the high-level structure of the data, but it's hard to see the details: plots of longitudinal data are often called spaghetti plots, and with good reason. Another way to compare the model to the data is to look at residuals, so let's do that. We add the predictions from the model to the original data (fitted), calculate residuals (resid), and add the residuals as well. The next plot is a little more complicated: We update the plot dataset (recall the use of %+% to update the default data), change the default y aesthetic to resid, and add a smooth line for all observations.

```
> Oxboys$fitted <- predict(model)
> Oxboys$resid <- with(Oxboys, fitted - height)
>
> oplot %+% Oxboys + aes(y = resid) + geom_smooth(aes(group=1))
```



The smooth line makes it evident that the residuals are not random, showing a deficiency in the model. We add a quadratic term, refit the model, recalculate predictions and residuals, and replot. There is now less evidence of model inadequacy.

```
> model2 <- update(model, height ~ age + I(age ^ 2))
> Oxboys$fitted2 <- predict(model2)
> Oxboys$resid2 <- with(Oxboys, fitted2 - height)
>
> oplot %+% Oxboys + aes(y = resid2) + geom_smooth(aes(group=1))
```



Notice how easily we were able to modify the plot object. We updated the data and replotted twice without needing to reinitialise `oplot`. Layering in `ggplot2` is designed to work well with the iterative process of fitting and evaluating models.