Chapter 12
Programming with ggplot2

12.1 Introduction

A major requirement of a good data analysis is flexibility. If your data changes, or you discover something that makes you rethink your basic assumptions, you need to be able to easily change many plots at once. The main inhibitor of flexibility is code duplication. If you have the same plotting statement repeated over and over again, you'll have to make the same change in many different places. Often just the thought of making all those changes is exhausting! This chapter will help you overcome that problem by showing you how to program with ggplot2.

To make your code more flexible, you need to reduce duplicated code by writing functions. When you notice you're doing the same thing over and over again, think about how you might generalise it and turn it into a function. If you're not that familiar with how functions work in R, you might want to brush up your knowledge at http://adv-r.had.co.nz/Functions.html.

In this chapter I'll show how to write functions that create:

- A single ggplot2 component.
- Multiple ggplot2 components.
- A complete plot.

And then I'll finish off with a brief illustration of how you can apply functional programming techniques to ggplot2 objects.

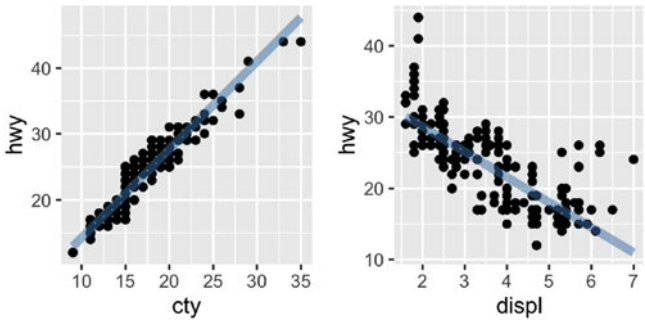
You might also find the cowplot (https://github.com/wilkelab/cowplot) and ggthemes (https://github.com/jrnold/ggthemes) packages helpful. As well as providing reusable components that help you directly, you can also read the source code of the packages to figure out how they work.

H. Wickham, *ggplot2*, Use R!, DOI 10.1007/978-3-319-24277-4_12

12.2 Single Components

Each component of a ggplot plot is an object. Most of the time you create the component and immediately add it to a plot, but you don't have to. Instead, you can save any component to a variable (giving it a name), and then add it to multiple plots:

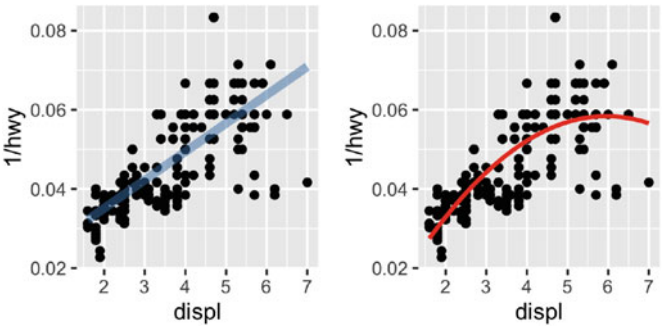
```
bestfit <- geom_smooth(
  method = "lm",
  se = FALSE,
  colour = alpha("steelblue", 0.5),
  size = 2
)
ggplot(mpg, aes(cty, hwy)) +
  geom_point() +
  bestfit
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  bestfit
```



That's a great way to reduce simple types of duplication (it's much better than copying-and-pasting!), but requires that the component be exactly the same each time. If you need more flexibility, you can wrap these reusable snippets in a function. For example, we could extend our `bestfit` object to a more general function for adding lines of best fit to a plot. The following code creates a `geom_lm()` with three parameters: the model `formula`, the line `colour` and the line `size`:

```
geom_lm <- function(formula = y ~ x, colour = alpha("steelblue", 0.5),
                    size = 2, ...)  {
  geom_smooth(formula = formula, se = FALSE, method = "lm", colour = colour,
    size = size, ...)
}
```

```
ggplot(mpg, aes(displ, 1 / hwy)) +
  geom_point() +
  geom_lm()
ggplot(mpg, aes(displ, 1 / hwy)) +
  geom_point() +
  geom_lm(y ~ poly(x, 2), size = 1, colour = "red")
```



Pay close attention to the use of "...". When included in the function definition "..." allows a function to accept arbitrary additional arguments. Inside the function, you can then use "..." to pass those arguments on to another function. Here we pass "..." onto `geom_smooth()` so the user can still modify all the other arguments we haven't explicitly overridden. When you write your own component functions, it's a good idea to always use "..." in this way.

Finally, note that you can only *add* components to a plot; you can't modify or remove existing objects.

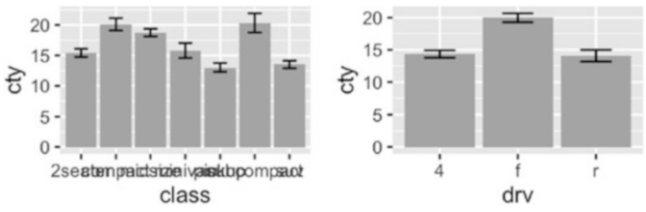
12.2.1 Exercises

1. Create an object that represents a pink histogram with 100 bins.
2. Create an object that represents a fill scale with the Blues ColorBrewer palette.
3. Read the source code for `theme_grey()`. What are its arguments? How does it work?
4. Create `scale_colour_wesanderson()`. It should have a parameter to pick the palette from the wesanderson package, and create either a continuous or discrete scale.

12.3 Multiple Components

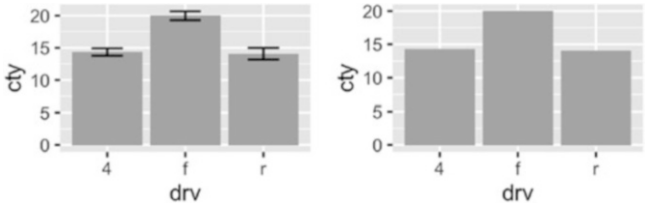
It's not always possible to achieve your goals with a single component. Fortunately, ggplot2 has a convenient way of adding multiple components to a plot in one step with a list. The following function adds two layers: one to show the mean, and one to show its standard error:

```
geom_mean <- function() {
  list(
    stat_summary(fun.y = "mean", geom = "bar", fill = "grey70"),
    stat_summary(fun.data = "mean_cl_normal", geom = "errorbar", width = 0.4)
  )
}
ggplot(mpg, aes(class, cty)) + geom_mean()
ggplot(mpg, aes(drv, cty)) + geom_mean()
```



If the list contains any `NULL` elements, they're ignored. This makes it easy to conditionally add components:

```
geom_mean <- function(se = TRUE) {
  list(
    stat_summary(fun.y = "mean", geom = "bar", fill = "grey70"),
    if (se)
      stat_summary(fun.data = "mean_cl_normal", geom = "errorbar", width = 0.4)
  )
}
ggplot(mpg, aes(drv, cty)) + geom_mean()
ggplot(mpg, aes(drv, cty)) + geom_mean(se = FALSE)
```



12.3.1 Plot Components

You're not just limited to adding layers in this way. You can also include any of the following object types in the list:

- A data.frame, which will override the default dataset associated with the plot. (If you add a data frame by itself, you'll need to use `%+%`, but this is not necessary if the data frame is in a list.)
- An `aes()` object, which will be combined with the existing default aesthetic mapping.
- Scales, which override existing scales, with a warning if they've already been set by the user.
- Coordinate systems and facetting specification, which override the existing settings.
- Theme components, which override the specified components.

12.3.2 Annotation

It's often useful to add standard annotations to a plot. In this case, your function will also set the data in the layer function, rather than inheriting it from the plot. There are two other options that you should set when you do this. These ensure that the layer is self-contained:

- `inherit.aes = FALSE` prevents the layer from inheriting aesthetics from the parent plot. This ensures your annotation works regardless of what else is on the plot.
- `show.legend = FALSE` ensures that your annotation won't appear in the legend.

One example of this technique is the `borders()` function built into ggplot2. It's designed to add map borders from one of the datasets in the maps package:

```
borders <- function(database = "world", regions = ".", fill = NA,
                    colour = "grey50", ...) {
  df <- map_data(database, regions)
  geom_polygon(
    aes_(~lat, ~long, group = ~group),
    data = df, fill = fill, colour = colour, ...,
    inherit.aes = FALSE, show.legend = FALSE
  )
}
```

12.3.3 Additional Arguments

If you want to pass additional arguments to the components in your function, ... is no good: there's no way to direct different arguments to different components. Instead, you'll need to think about how you want your function to work, balancing the benefits of having one function that does it all vs. the cost of having a complex function that's harder to understand.

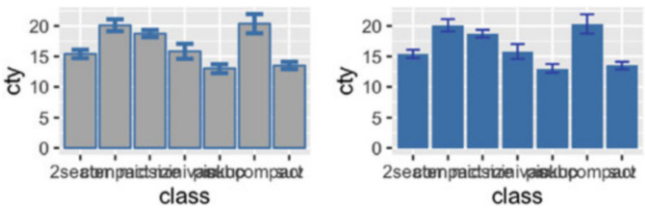
To get you started, here's one approach using `modifyList()` and `do.call()`:

```
geom_mean <- function(..., bar.params = list(), errorbar.params = list()) {
  params <- list(...)
  bar.params <- modifyList(params, bar.params)
  errorbar.params  <- modifyList(params, errorbar.params)

  bar <- do.call("stat_summary", modifyList(
    list(fun.y = "mean", geom = "bar", fill = "grey70"),
    bar.params)
  )
  errorbar <- do.call("stat_summary", modifyList(
    list(fun.data = "mean_cl_normal", geom = "errorbar", width = 0.4),
    errorbar.params)
  )

  list(bar, errorbar)
}

ggplot(mpg, aes(class, cty)) +
  geom_mean(
    colour = "steelblue",
    errorbar.params = list(width = 0.5, size = 1)
  )
ggplot(mpg, aes(class, cty)) +
  geom_mean(
    bar.params = list(fill = "steelblue"),
    errorbar.params = list(colour = "blue")
  )
```



If you need more complex behaviour, it might be easier to create a custom geom or stat. You can learn about that in the extending ggplot2 vignette included with the package. Read it by running `vignette("extending-ggplot2")`.

12.3.4 Exercises

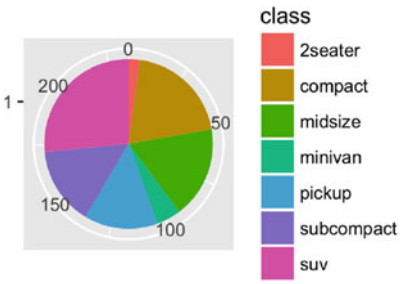
1. To make the best use of space, many examples in this book hide the axes labels and legend. I've just copied-and-pasted the same code into multiple places, but it would make more sense to create a reusable function. What would that function look like?
2. Extend the `borders()` function to also add `coord_quickmap()` to the plot.
3. Look through your own code. What combinations of geoms or scales do you use all the time? How could you extract the pattern into a reusable function?

12.4 Plot Functions

Creating small reusable components is most in line with the ggplot2 spirit: you can recombine them flexibly to create whatever plot you want. But sometimes you're creating the same plot over and over again, and you don't need that flexibility. Instead of creating components, you might want to write a function that takes data and parameters and returns a complete plot.

For example, you could wrap up the complete code needed to make a piechart:

```
piechart <- function(data, mapping) {
  ggplot(data, mapping) +
    geom_bar(width = 1) +
    coord_polar(theta = "y") +
    xlab(NULL) +
    ylab(NULL)
}
piechart(mpg, aes(factor(1), fill = class))
```



This is much less flexible than the component based approach, but equally, it's much more concise. Note that I was careful to return the plot object, rather than printing it. That makes it possible add on other ggplot2 components.

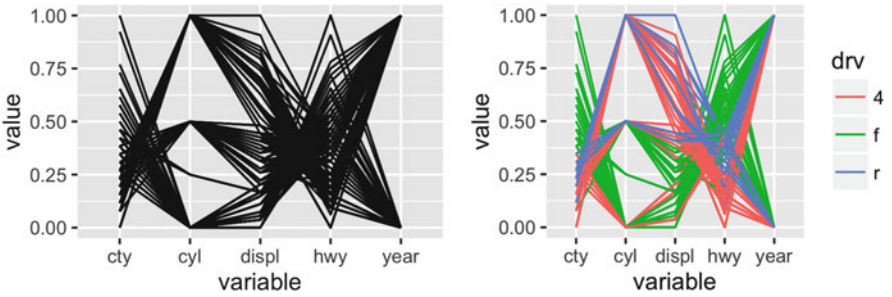
You can take a similar approach to drawing parallel coordinates plots (PCPs). PCPs require a transformation of the data, so I recommend writing two functions: one that does the transformation and one that generates the plot. Keeping these two pieces separate makes life much easier if you later want to reuse the same transformation for a different visualisation.

```
pcp_data <- function(df) {
  is_numeric <- vapply(df, is.numeric, logical(1))

  # Rescale numeric columns
  rescale01 <- function(x) {
    rng <- range(x, na.rm = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
  }
  df[is_numeric] <- lapply(df[is_numeric], rescale01)

  # Add row identifier
  df$.row <- rownames(df)

  # Treat numerics as value (aka measure) variables
  # gather_ is the standard-evaluation version of gather, and
  # is usually easier to program with.
  tidyr::gather_(df, "variable", "value", names(df)[is_numeric])
}
pcp <- function(df, ...) {
  df <- pcp_data(df)
  ggplot(df, aes(variable, value, group = .row)) + geom_line(...)
}
pcp(mpg)
pcp(mpg, aes(colour = drv))
```



A complete exploration of this idea is `qplot()`, which provides a fairly deep wrapper around the most common `ggplot()` options. I recommend studying the source code if you want to see how far these basic techniques can take you.

12.4.1 Indirectly Referring to Variables

The `piechart()` function above is a little unappealing because it requires the user to know the exact `aes()` specification that generates a pie chart. It would be more convenient if the user could simply specify the name of the variable to plot. To do that you'll need to learn a bit more about how `aes()` works.

`aes()` uses non-standard evaluation: rather than looking at the values of its arguments, it looks at their expressions. This makes it difficult to work with programmatically as there's no way to store the name of a variable in an object and then refer to it later:

```
x_var <- "displ"
aes(x_var)
#> * x -> x_var
```

Instead we need to use `aes_()`, which uses regular evaluation. There are two basic ways to create a mapping with `aes_()`:

- Using a *quoted call*, created by `quote()`, `substitute()`, `as.name()`, or `parse()`.

```
aes_(quote(displ))
#> * x -> displ
aes_(as.name(x_var))
#> * x -> displ
aes_(parse(text = x_var)[[1]])
#> * x -> displ
f <- function(x_var) {
```

```
  aes_(substitute(x_var))
}
f(displ)
#> * x -> displ
```

The difference between `as.name()` and `parse()` is subtle. If `x_var` is "a + b", `as.name()` will turn it into a variable called `‘a + b‘`, `parse()` will turn it into the function call `a + b`. (If this is confusing, http://adv-r.had.co.nz/Expressions.html might help).

- Using a formula, created with `~`.

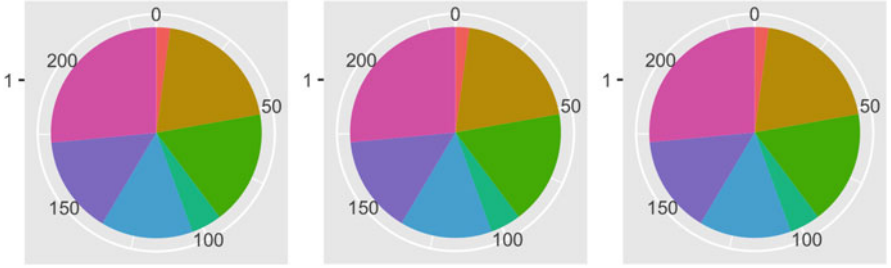
```
aes_(~displ)
#> * x -> displ
```

`aes_()` gives us three options for how a user can supply variables: as a string, as a formula, or as a bare expression. These three options are illustrated below

```
piechart1 <- function(data, var, ...) {
  piechart(data, aes_(~factor(1), fill = as.name(var)))
}
piechart1(mpg, "class") + theme(legend.position = "none")

piechart2 <- function(data, var, ...) {
  piechart(data, aes_(~factor(1), fill = var))
}
piechart2(mpg, ~class) + theme(legend.position = "none")

piechart3 <- function(data, var, ...) {
  piechart(data, aes_(~factor(1), fill = substitute(var)))
}
piechart3(mpg, class) + theme(legend.position = "none")
```



There's another advantage to `aes_()` over `aes()` if you're writing ggplot2 plots inside a package: using `aes_(~x, ~y)` instead of `aes(x, y)` avoids the global variables NOTE in `R CMD check`.

12.4.2 The Plot Environment

As you create more sophisticated plotting functions, you'll need to understand a bit more about ggplot2's scoping rules. ggplot2 was written well before I understood the full intricacies of non-standard evaluation, so it has a rather simple scoping system. If a variable is not found in the `data`, it is looked for in *the* plot environment. There is only one environment for a plot (not one for each layer), and it is the environment in which `ggplot()` is called from (i.e. the `parent.frame()`).

This means that the following function won't work because `n` is not stored in an environment accessible when the expressions in `aes()` are evaluated.

```
f <- function() {
  n <- 10
  geom_line(aes(x / n))
}
df <- data.frame(x = 1:3, y = 1:3)
ggplot(df, aes(x, y)) + f()
#> Error in x/n: non-numeric argument to binary operator
```

Note that this is only a problem with the `mapping` argument. All other arguments are evaluated immediately so their values (not a reference to a name) are stored in the plot object. This means the following function will work:

```
f <- function() {
  colour <- "blue"
  geom_line(colour = colour)
}
ggplot(df, aes(x, y)) + f()
```

If you need to use a different environment for the plot, you can specify it with the `environment` argument to `ggplot()`. You'll need to do this if you're creating a plot function that takes user provided data. See `qplot()` for an example.

12.4.3 Exercises

1. Create a `distribution()` function specially designed for visualising continuous distributions. Allow the user to supply a dataset and the name of a variable to visualise. Let them choose between histograms, frequency polygons, and density plots. What other arguments might you want to include?
2. What additional arguments should `pcp()` take? What are the downsides of how ... is used in the current code?

3. Advanced: why doesn't this code work? How can you fix it?

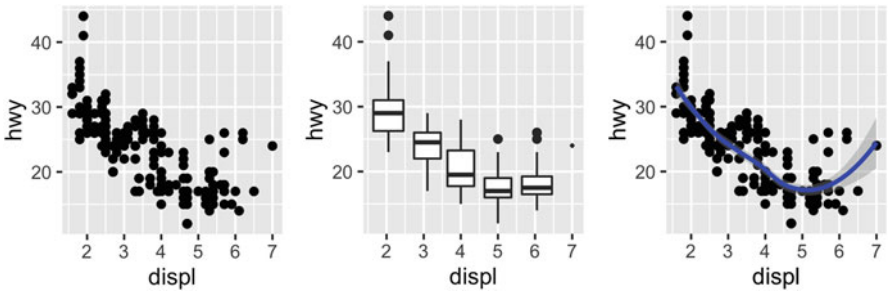
```
f <- function() {
  levs <- c("2seater", "compact", "midsize", "minivan", "pickup",
    "subcompact", "suv")
  piechart3(mpg, factor(class, levels = levs))
}
f()
#> Error in factor(class, levels = levs): object 'levs' not found
```

12.5 Functional Programming

Since ggplot2 objects are just regular R objects, you can put them in a list. This means you can apply all of R's great functional programming tools. For example, if you wanted to add different geoms to the same base plot, you could put them in a list and use `lapply()`.

```
geoms <- list(
  geom_point(),
  geom_boxplot(aes(group = cut_width(displ, 1))),
  list(geom_point(), geom_smooth())
)

p <- ggplot(mpg, aes(displ, hwy))
lapply(geoms, function(g) p + g)
#> [[1]]
#>
#> [[2]]
#>
#> [[3]]
```



If you're not familiar with functional programming, read through http://adv-r.had.co.nz/Functional-programming.html and think about how you might apply the techniques to your duplicated plotting code.

12.5.1 Exercises

1. How could you add a `geom_point()` layer to each element of the following list?

```
plots <- list(
  ggplot(mpg, aes(displ, hwy)),
  ggplot(diamonds, aes(carat, price)),
  ggplot(faithfuld, aes(waiting, eruptions, size = density))
)
```

2. What does the following function do? What's a better name for it?

```
mystery <- function(...) {
  Reduce(`+`, list(...), accumulate = TRUE)
}

mystery(
  ggplot(mpg, aes(displ, hwy)) + geom_point(),
  geom_smooth(),
  xlab(NULL),
  ylab(NULL)
)
```