R Visualizations - Base and *ggplot2*Statistics 506

Note: Do to the heavy reliance on images in this document, I suggest toggling to light mode via the button in the top-right corner of the document.

R Graphical Systems

In these notes we'll discuss three graphical systems in R.

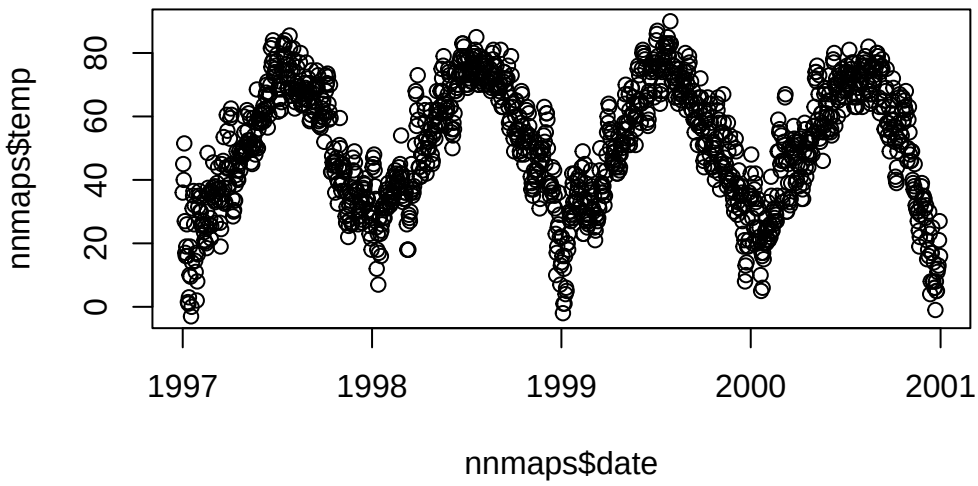
1. Base R plotting
2. **ggplot2**, nominally part of the tidyverse
3. **plotly** for generating interactive plots

There are a number of less commonly used graphical systems, the most important of which is **lattice**. It was a failed attempt to standardize and improve base R's plotting capabilities, and while it still exists in R and remains under development, it is rarely used.

Base R plotting

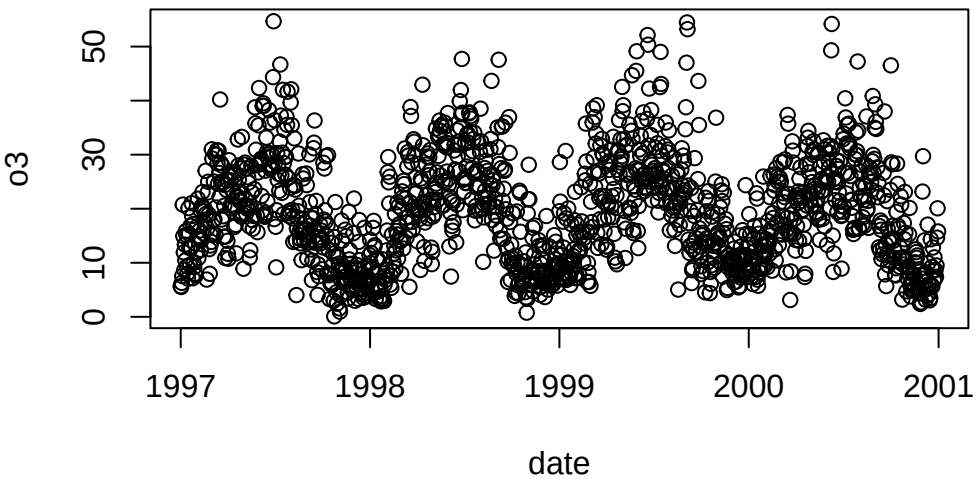
Plotting with base R usually consists of a single high-level plot function, followed optionally by lower-level functions. We'll use the NMMAPS data for our examples. THe National Morbidity and Mortality Air Pollution Study is a time-series collected to look at the relationship between pollutants and mortality. A subset of the data (Chicago for 1997-2000) can be found here. Let's plot the ozone (`o3`) level over time.

```
nnmaps <- read.csv("data/chicago-nmmaps.csv")
nnmaps$date <- as.Date(nnmaps$date)
plot(nnmaps$temp ~ nnmaps$date)
```



The `with` function can make the `plot` call more readable.

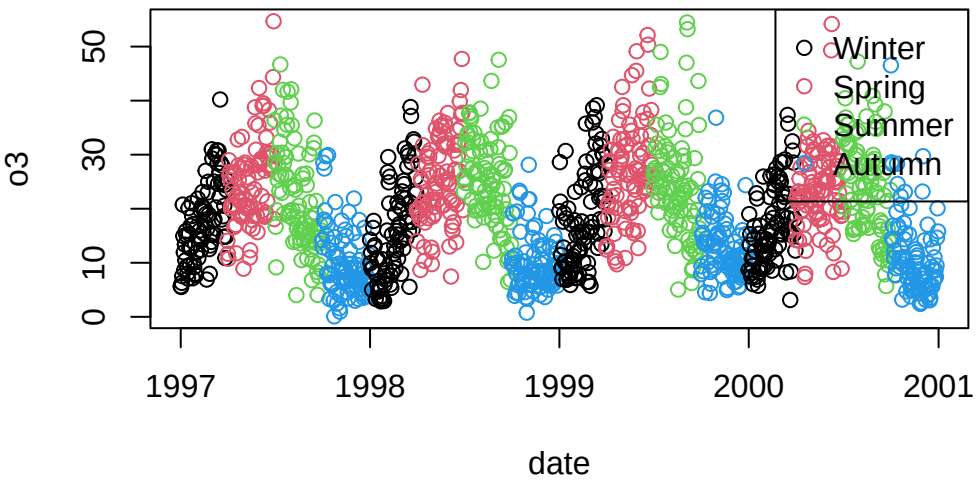
```
with(nnmaps, plot(o3 ~ date))
```



Do **not** use `attach` and `detach`. These can have unintended side-effects and it is easy to lose track of what has been attached.

Let's color the plot by season. `type = "n"` tells R not to draw the plot just the plotting area, and is a quick way to define the ranges of the axes

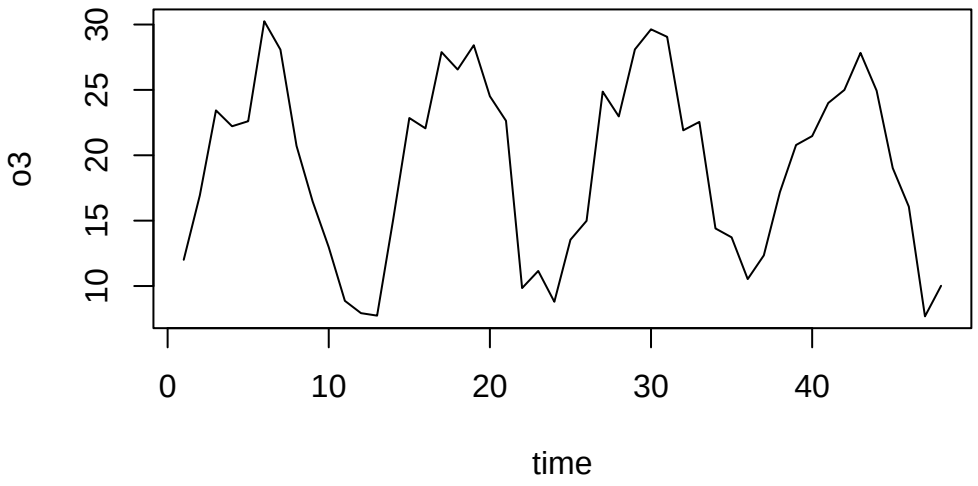
```
nnmaps$season <- factor(nnmaps$season,
                        levels = c("Winter", "Spring",
                                   "Summer", "Autumn"))
with(nnmaps,
     plot(o3 ~ date, col = season))
with(nnmaps,
     legend("topright", levels(season), col = unique(season), pch = 1))
```



`col` can optionally take in any length vector and use recycling.

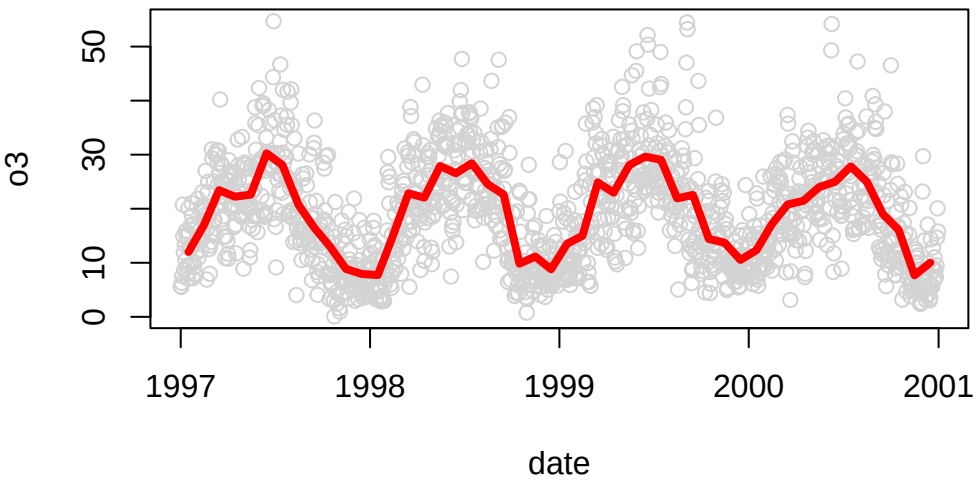
The plot is very messy. We can take the average by month and plot that. The `aggregate` call will produce a lot of warnings due `nnamps` having non-numeric variables; we could subset `nnmaps` before passing it into `aggregate`, but the warnings are ignorable, so we'll do that.

```
suppressWarnings(nnmaps_month <-
                   aggregate(nnmaps, by = list(nnmaps$month_numeric,
                                               nnmaps$year),
                             FUN = mean, na.rm = TRUE))
nnmaps_month <- nnmaps_month[order(nnmaps_month$year,
                                   nnmaps_month$month_numeric), ]
nnmaps_month$time <- seq_len(nrow(nnmaps_month))
with(nnmaps_month,
     plot(o3 ~ time, type = "l"))
```



Let's plot the average over the raw values.

```
with(nnmaps, plot(o3 ~ date, col = "lightgrey"))
with(nnmaps_month, lines(o3 ~ date))
with(nnmaps_month, lines(o3 ~ date, lwd = 4, col = "red"))
```



The `lwd` and `col` arguments don't appear in the help for `plot`, `lines`, etc. Instead, they get passed through `...` into `par()`.

```
head(par())
```

```
$xlog
[1] FALSE

$ylog
[1] FALSE

$adj
[1] 0.5

$ann
[1] TRUE

$ask
[1] FALSE

$bg
[1] "transparent"
```

```
length(par)
```

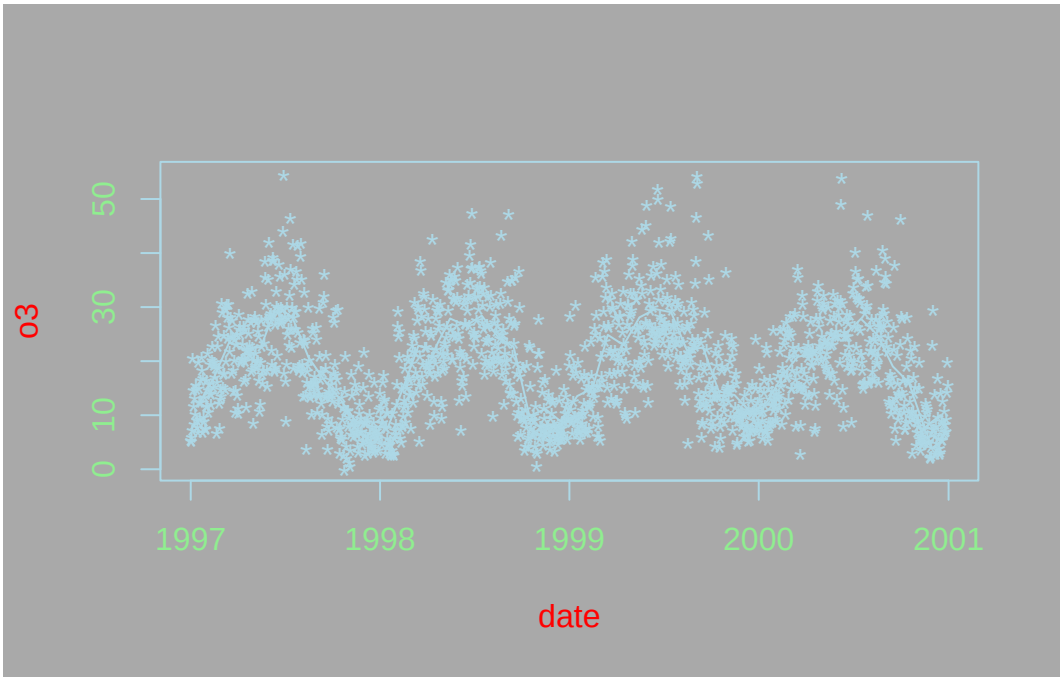
```
[1] 1
```

These graphical parameters are generic and can be passed to (almost) all base R plotting functions. It may be easier to define the parameters prior to plotting. Be sure to save these existing parameters for later restoration.

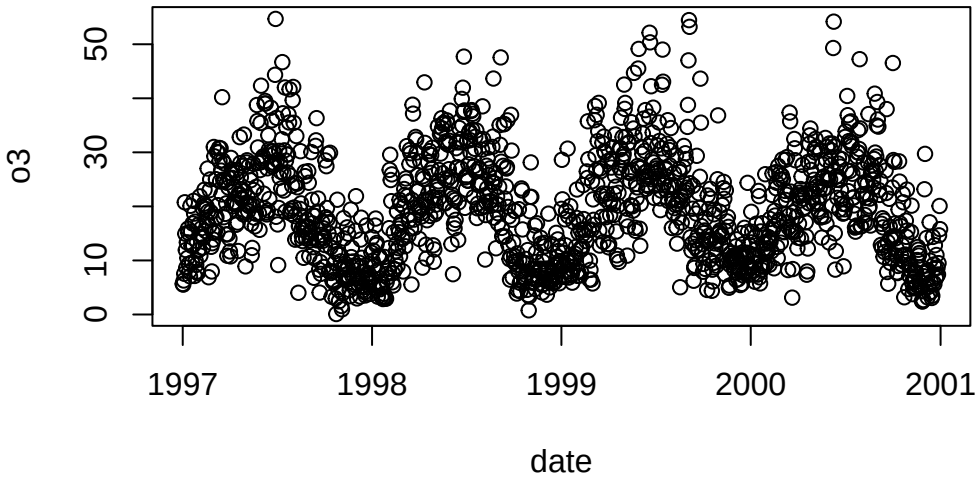
```
par()$bg
```

```
[1] "transparent"
```

```
oldpar <- par(no.readonly = TRUE) # no.readonly avoids warnings later
par(bg = "darkgrey",
    fg = "lightblue",
    col.axis = "lightgreen",
    col.lab = "purple")
with(nnmaps, plot(o3 ~ date, pch = "*", bg = "red", col.lab = "red"))
with(nnmaps_month, lines(o3 ~ date))
```



```
par(oldpar)
with(nnmaps, plot(o3 ~ date))
```



Finally, let's add a smooth curve over these points. It looks sinusoidal, so we can fit a sinusoidal model.

```
# First, make sure *date* has only unique values
length(nnmaps$date) == length(unique(nnmaps$date))
```

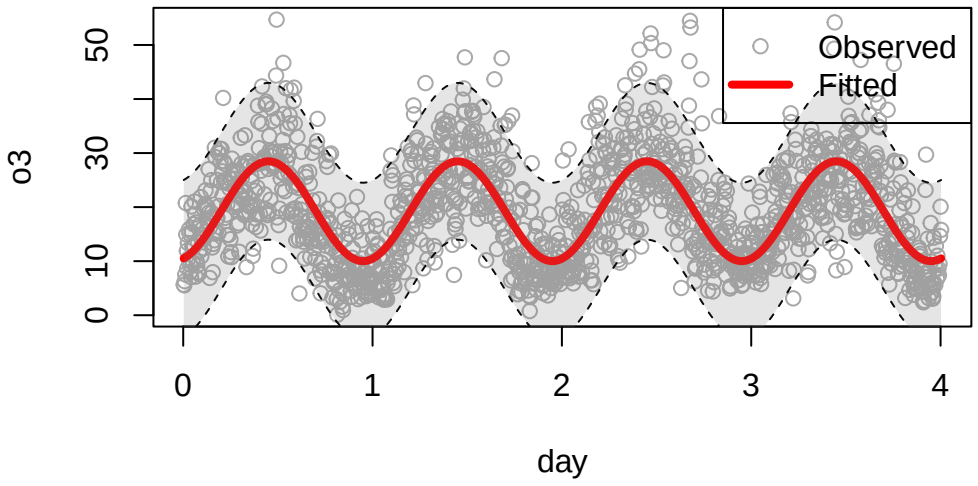
```
[1] TRUE
```

```
# Sort by date
nnmaps <- nnmaps[order(nnmaps$date), ]
# Generate a numeric time variable
nnmaps$day <- seq_len(nrow(nnmaps))/365
nnmaps$daycos <- cos(2 * pi * nnmaps$day)
nnmaps$daysin <- sin(2 * pi * nnmaps$day)
mod <- lm(o3 ~ daycos + daysin, data = nnmaps)
pred <- data.frame(predict(mod, interval = "prediction"))
```

```
Warning in predict.lm(mod, interval = "prediction"): predictions on current data refer to _fu
```

Note that `interval = "confidence"` would be the more common choice here, but the bounds would be very tight and barely visible, so I include the prediction intervals to make the results more visually interesting.

```
with(nnmaps, plot(o3 ~ day, col = "darkgrey"))
lines(pred$fit ~ nnmaps$day, col = "red", lwd = 4)
lines(pred$lwr ~ nnmaps$day, lty = "dashed")
lines(pred$upr ~ nnmaps$day, lty = "dashed")
polygon(c(rev(nnmaps$day), nnmaps$day), c(rev(pred$upr), pred$lwr),
        col = rgb(.5, .5, .5, .2), border = NA)
legend("topright", legend = c("Observed", "Fitted"),
       lty = c(NA, 1), lwd = c(NA, 4),
       pch = c(1, NA),
       col = c("darkgrey", "red"))
```



Other secondary plot functions

We saw above the `lines()` function. There are a number of other functions we can use to add to an existing plot. For example,

- `points()` - add a scatterplot
- `abline()` - draws an infinite-length straight line, either with intercept and slope (`a=` and `b=` argument) or horizontal (`h=`) or vertical (`v=`).
- `segments()` - draws a line segment between two points
- `rect()` and `polygon()` - Draw a rectangle or an arbitrary polygon
- `text()` - add text to a plot (can be used for either a single entry, or to "plot" text, e.g. a scatterplot where the points are labels)
- `title()` or `mtext()` - adding a title or other arbitary text in the margins (outside the plottable area)

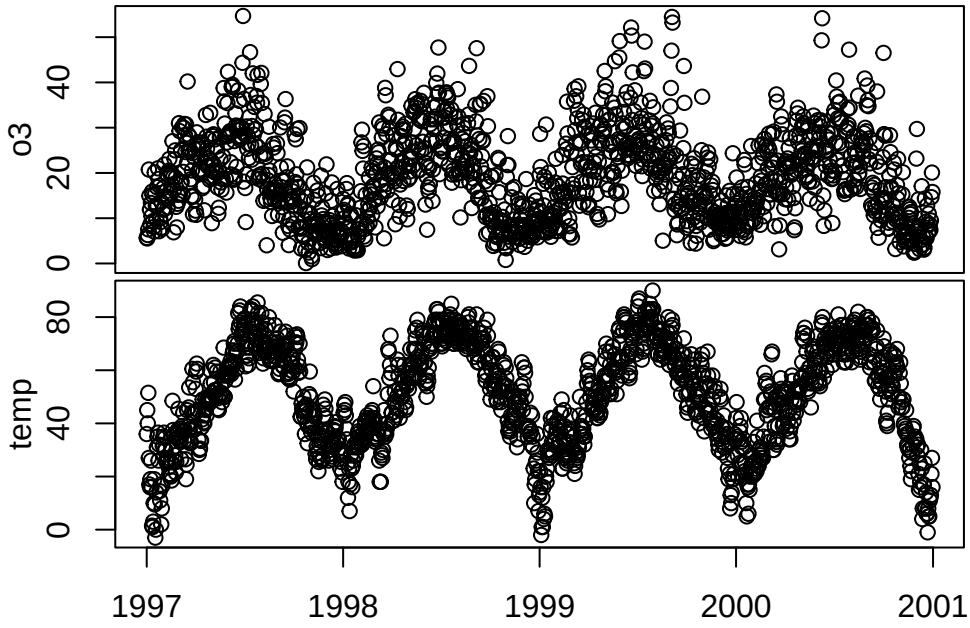
Plot spacing and arrangement

`par` can be used to control the margin spacing of plot, and to arrange multiple plot on one image.

```
#oldpar <- par(no.readonly = TRUE)
#par(mfrow = c(2, 1))
#with(nnmaps, {
#     plot(o3 ~ date)
#     plot(temp ~ date)
#})
```

The margins around the plots are too large. The `oma` (outer margin), `mar` (inner margins) and `mgp` (placement of axis attributes in margins) can be tweaked.

```
par(mfrow = c(2, 1),
    oma = c(1, 1, 0, 0) + .1,
    mar = c(0, 3, 1, 1) + .1,
    mgp = c(2, 1, 0))
with(nnmaps,
     plot(o3 ~ date, xaxt = "n"))
par(mar = c(1, 3, 0, 1) + .1,
    mgp = c(2, 1, 0))
with(nnmaps,
     plot(temp ~ date))
```



The addition of the .1 is by convention; it helps avoids some rough looking results where things are drawn on top of each other. They can be excluded if desired.

There are various references online to help understand this a bit better, and to try and help you avoid randomly choosing values and hoping for the best. For example,

- https://r-graph-gallery.com/74-margin-and-oma-cheatsheet.html
- https://web.archive.org/web/20150623062735/https://research.stowers-institute.org/efg/R/Graphics/Basics/mar-oma/index.htm

Be sure to always reset your `par`.

```
par(oldpar)
```

Graphical devices

Behind the scenes, when producing a plot, R needs a **Device** to draw it to. The Device can be a new plotting window, or it can be an image file. When you first load R and call `plot()`, it automatically creates a new Device as a plotting window for R to draw to.

The exact details of how all this work used to be much more important, but with the advent of RStudio where there is only a single plot open at a time, most of this functionality is only relevant for developers of graphical packages.

A few things that are useful to know. First, in addition to the default Device of drawing a plot to the screen, you can can instead draw to a file on disk.

```
# This code is NOT evaluated
png("myplot.png")
plot(....)
dev.off()
```

By calling `png()`, we are telling R to send all plotting to that Device (specifically that file). So if you were to run that code (with a valid `plot()`), no plot would pop up, but that image would be created.

`dev.off()` "closes" the current Device; when the Device is an output file, it stops writing to it, if the Device is a window inside R, it just removes the window.

Since `par()` settings are specific to a Device, `dev.off()` also has the side-effect of essentially resetting your `par()`.

ggplot2

ggplot2 is a package created by Hadley Wickham which defines a "grammar" for graphics based upon the work of Leland Wilkerson in The Grammer of Graphics. It is now part of the tidyverse, though predates the tidyverse by a large number of years, and does not completely conform to tidyverse standards. The most obvious example of its deviation is that instead of using `%>%` for piping, you "pipe" with addition.

The basic idea is that while base R's plotting mechanics are pretty robust, knowledge in one does not necessarily transfer to another. For example, while `plot` can create scatter plots and line plots easily, the arguments that pass to it don't also pass to `histogram`. Some do (see `?par`), but those aren't always supported.

ggplot2 takes the emphasis away from understanding the specific quirks of each desired plot, and instead attempts to create a unified language of plots, such that the difference between a histogram and a scatter plot call is simply `geom_point` vs `geom_histogram`.

Graphic grammar template

The basic template of any **ggplot2** plot is:

```
ggplot(data = <DATA>,
       mapping = aes(<MAPPING>)) +
  <GEOM_FUNCTION>(
     stat = <STAT>,
```

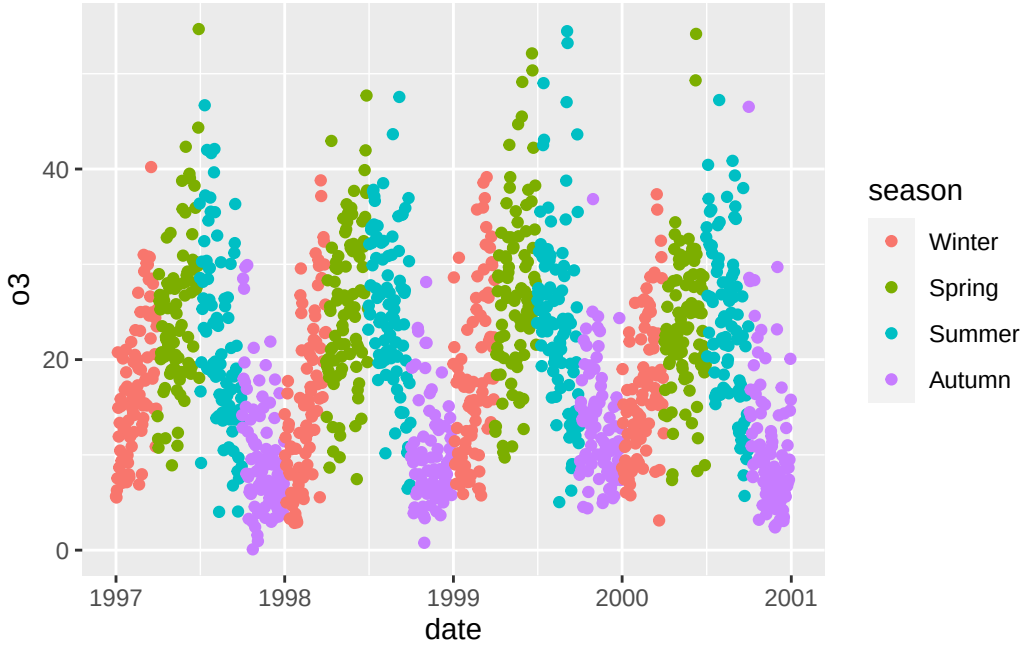
```
    position = <POSITION>
) +
<COORDINATE_FUNCTION> +
<FACET_FUNCTION> +
<SCALE_FUNCTION> +
<THEME_FUNCTION>
```

There are 9 arguments, wrapped in brackets, which can be defined. Rarely will all 9 be defined; sometimes you will define multiple versions of arguments. Only the first three, `<DATA>`, `<MAPPING>` and `<GEOM_FUNCTION>`, are required to produce a meaningful plot.

Example

Let's reproduce the plot from above.

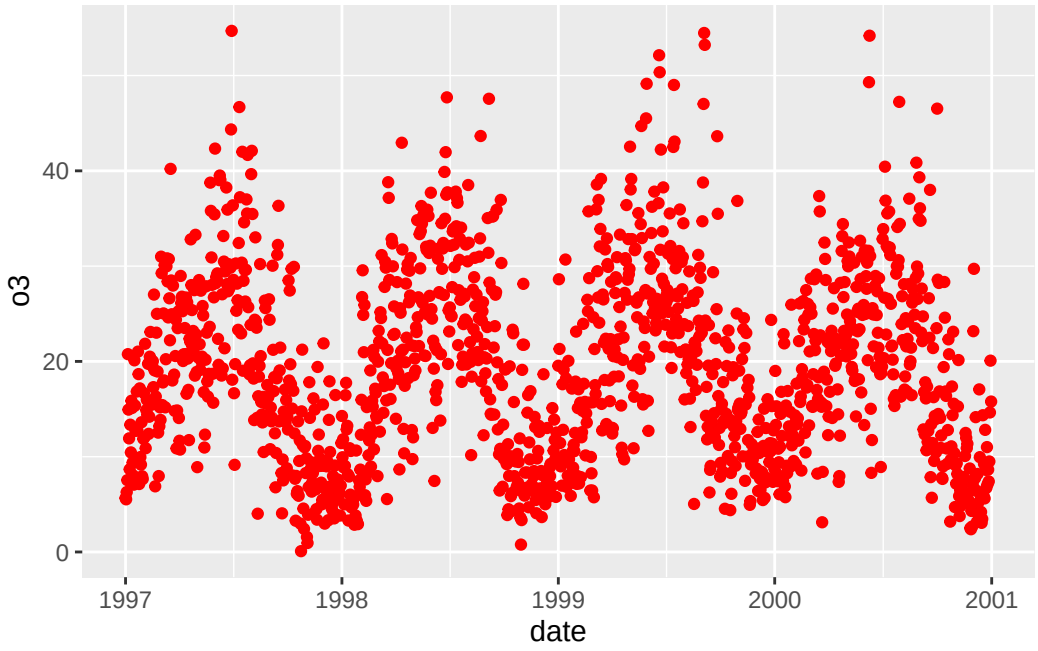
```
library(ggplot2)
ggplot(nnmaps, aes(x = date, y = o3, color = season)) +
  geom_point()
```



The aesthetic mapping defines how variables in the dataset are connected to visual properties or outputs. The terms "aesthetic" and "mapping" are often used interchangeably with the more

formal "aesthetic mapping". Just think of a mapping as defining properties of the output that depend upon variables. For example, coloring the points of a scatter plot based upon a categorical variable is a mapping, whereas coloring all points red is not:

```
ggplot(nnmaps, aes(x = date, y = o3)) +
  geom_point(color = "red")
```



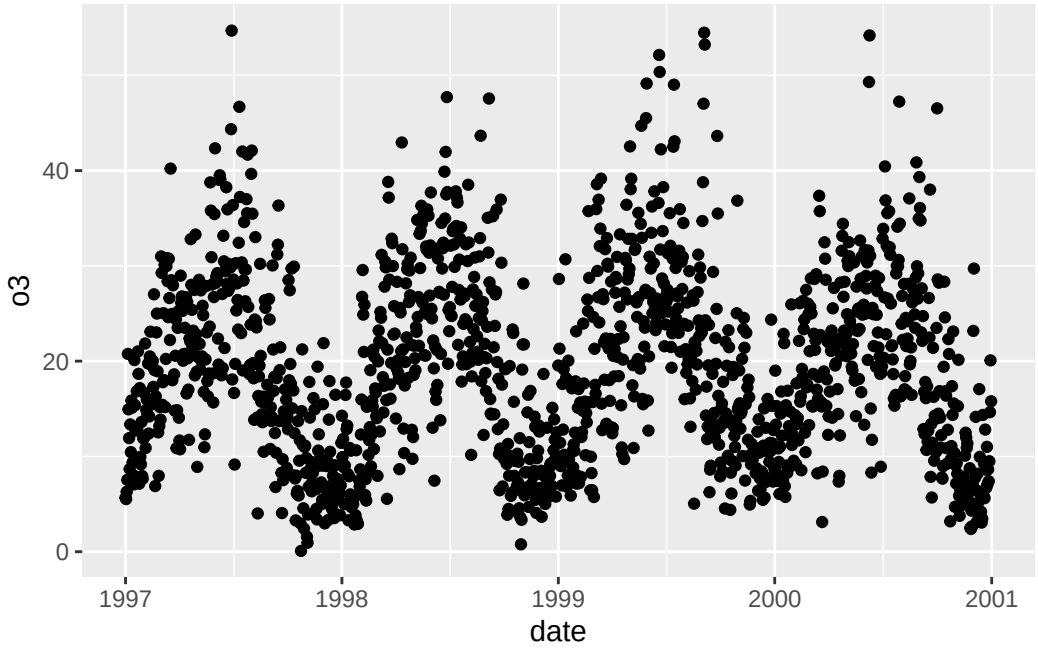
The `geom_point()` function defines the type of plot being generated. In this case, a "point" corresponds to a scatterplot. You can see a full list of geoms on the **ggplot2** website.

The objects created by `ggplot` are `gg`, you can use them to build up plots.

```
myplot <- ggplot(nnmaps, aes(x = date, y = o3))
is(myplot)
```

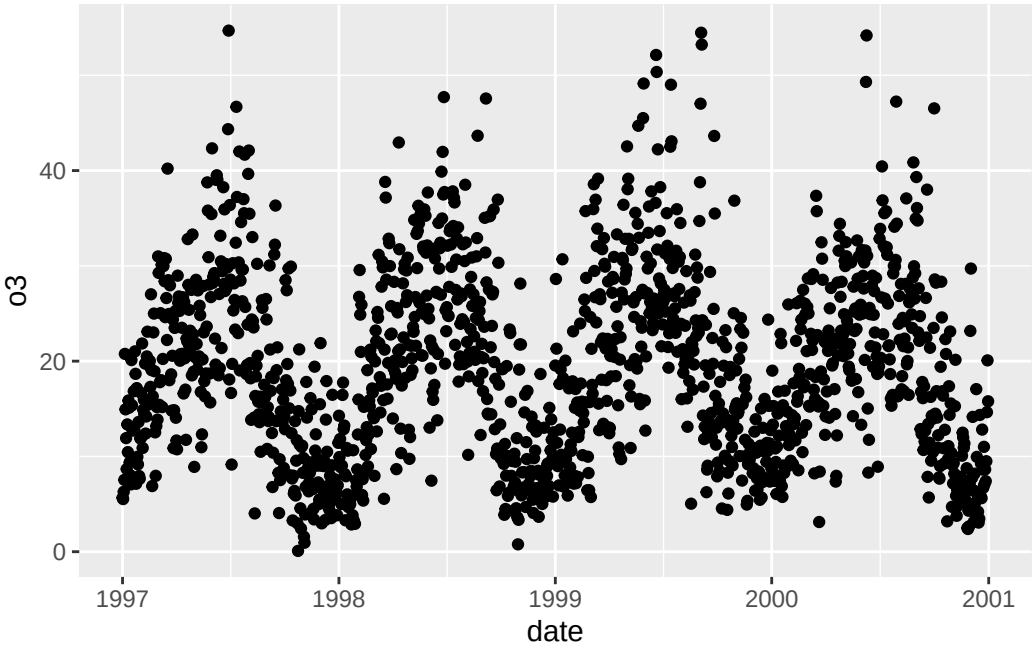
```
[1] "gg"
```

```
myplot + geom_point()
```



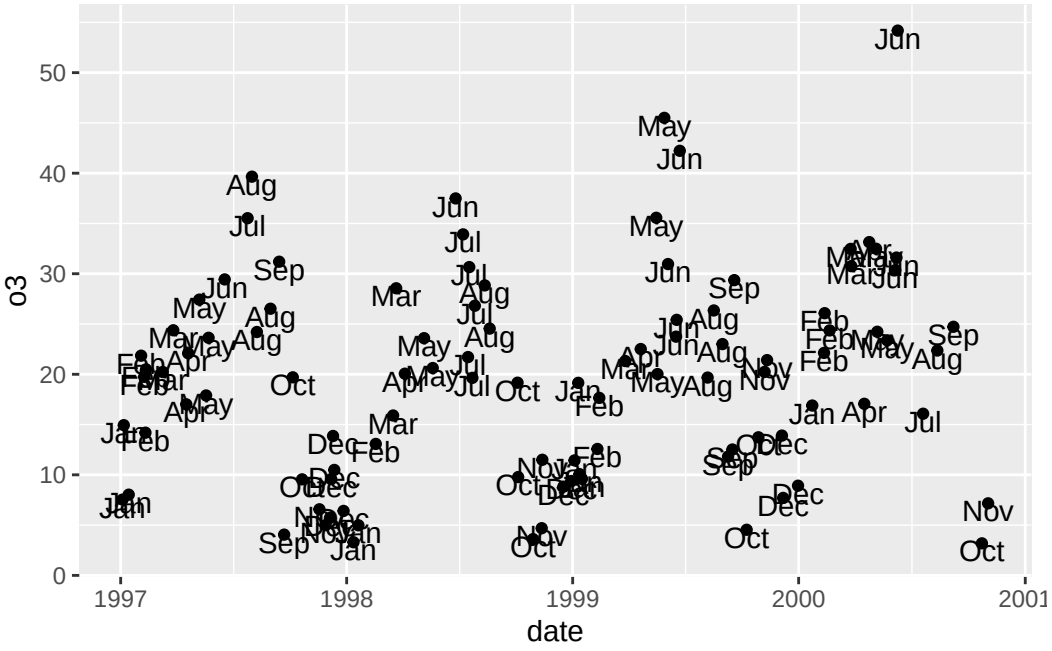
There is inheritance going on here. We could have instead generated the plot with:

```
ggplot() +
  geom_point(nnmaps, mapping = aes(x = date, y = o3))
```



The downside being if we want to stack multiple geoms, they will not inherit from other geoms, only from the main function.

```
nnmaps_small <- nnmaps[sample(1:nrow(nnmaps), 100), ]
ggplot(nnmaps_small, aes(x = date, y = o3)) +
  geom_point() +
  geom_text(aes(label = month), nudge_y = -.75)
```



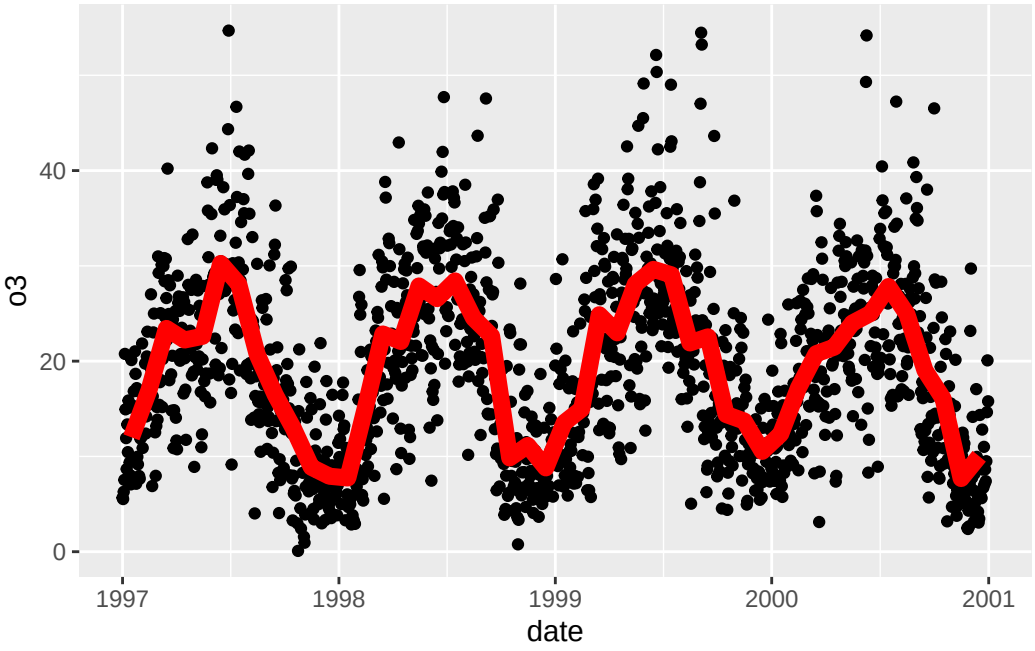
Both geoms inherit the data and the mapping from `ggplot`.

```
ggplot() +
  geom_point(nnmaps_small, aes(x = date, y = o3)) +
  geom_text(aes(label = month), nudge_y = -.5)
```

```
Error in `geom_point()`:
! `mapping` must be created by `aes()`
```

We can replicate the averaged line by breaking inheritance of only the data.

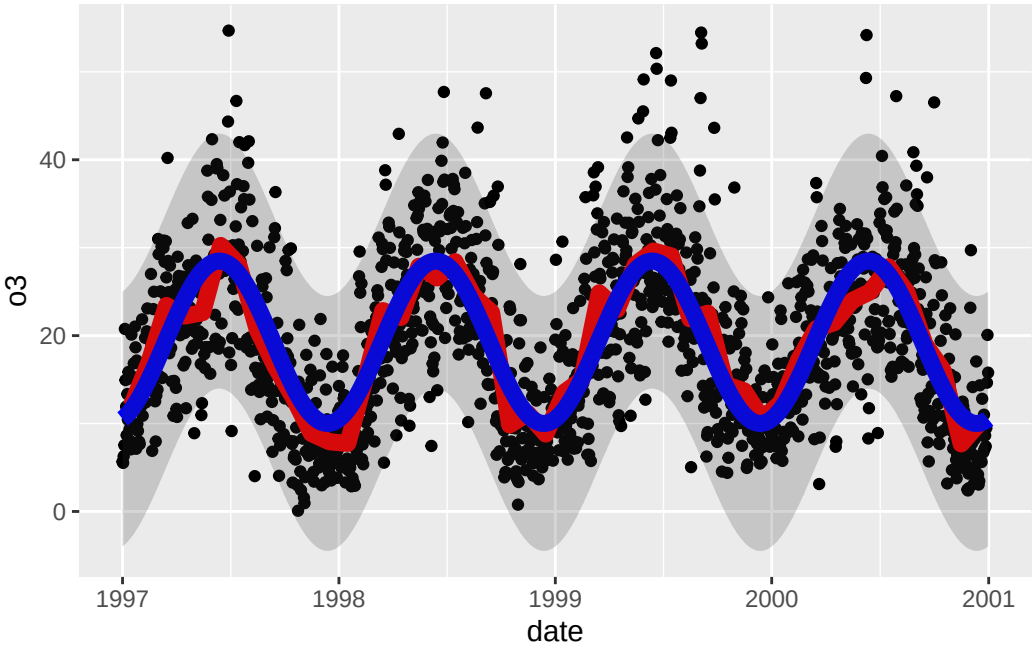
```
ggplot(nnmaps, aes(x = date, y = o3)) +
  geom_point() +
  geom_line(data = nnmaps_month, color = "red", linewidth = 3)
```



1. `geom_point` inherited both data and mapping. `geom_line` inherited only mapping - which worked fine because the same variables are being used in both data. If the names of variables changed, we'd need to pass a new mapping.
2. The order of arguments swaps - `ggplot` has first argument `data=` and second `mapping=`, whereas geoms have first argument `mapping=` and second argument `data=`. What out for bugs related to passing arguments by position.

Finally, let's add the smoothed regression line on top. Here we inherit the original data and some of the original mapping.

```
nnmaps_pred <- dplyr::bind_cols(nnmaps, pred)
ggplot(nnmaps_pred, aes(x = date, y = o3)) +
  geom_point() +
  geom_line(data = nnmaps_month, color = "red", linewidth = 3) +
  geom_line(aes(y = fit), color = "blue", linewidth = 3) +
  geom_ribbon(aes(ymin = lwr, ymax = upr), alpha = .2)
```

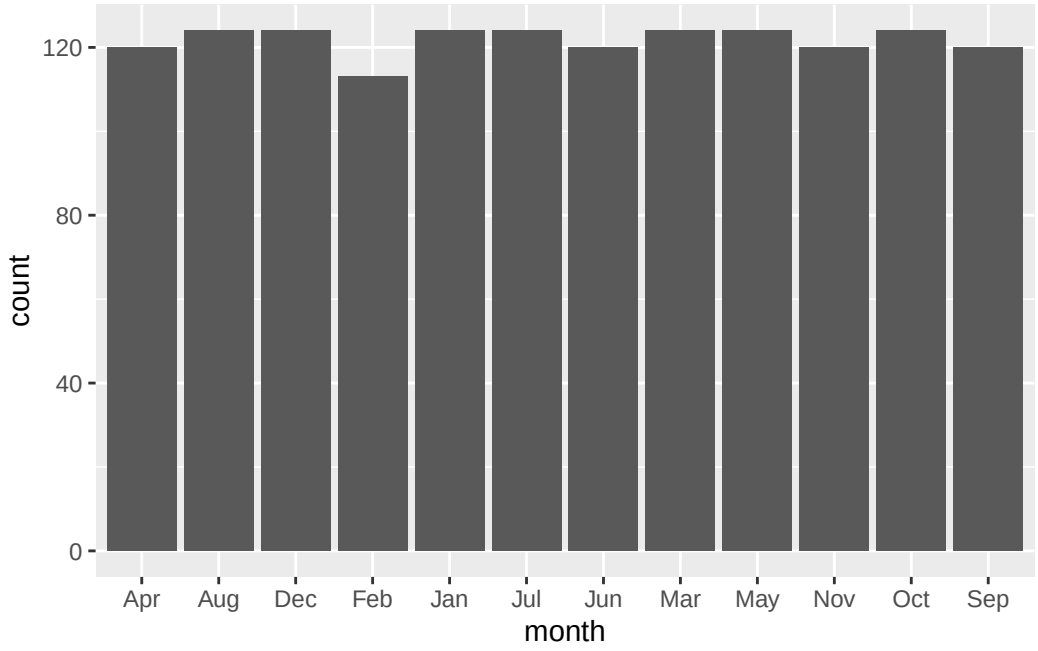


The other arguments

Stats

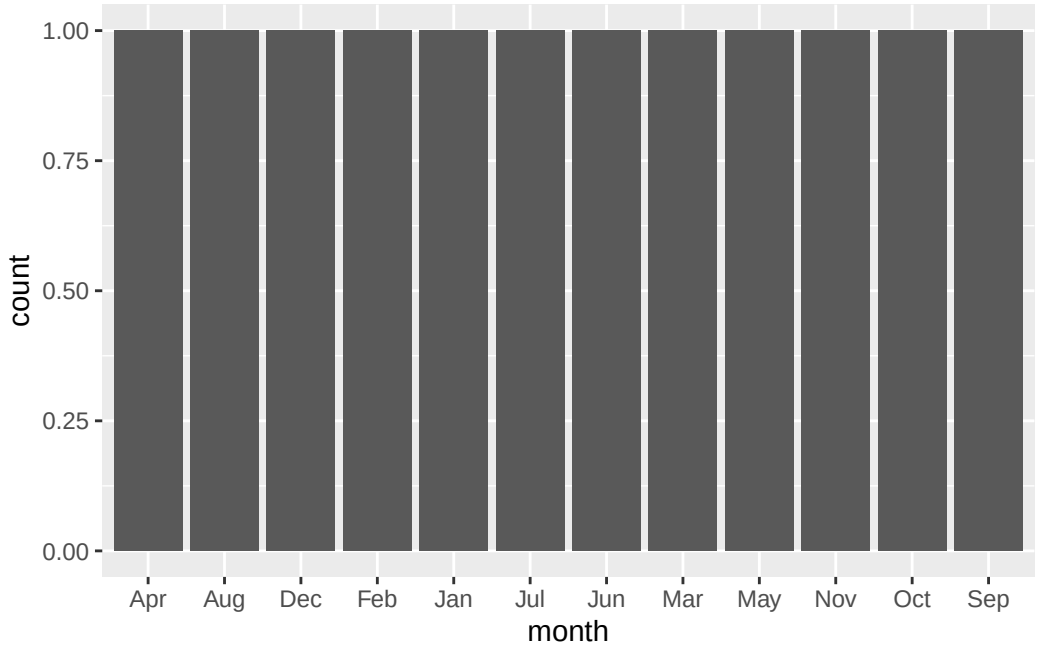
Behind the scenes, certain geoms are transforming the data before plotting. For example,

```
ggplot(nnmaps, aes(x = month)) +
  geom_bar()
```

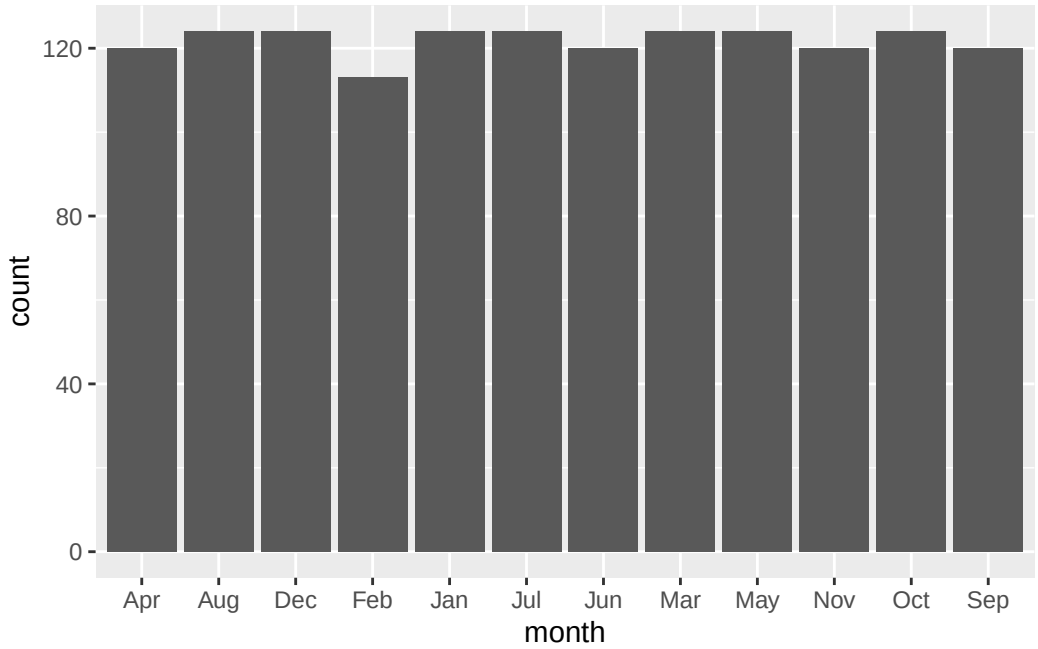


`geom_bar` is performing a count on the number of month. If you already have summarized data (e.g. you have just the counts already), you may want to tell `geom_bar` to use the existing count variable instead of computing a new count. Passing `stat = "identity"` and `aes(x = class, y = count)` will accomplish this.

```
tmonths <- data.frame(table(nnmaps$month))
names(tmonths) <- c("month", "count")
ggplot(tmonths, aes(x = month)) +
  geom_bar()
```




```
ggplot(tmonths, aes(x = month, y = count)) +
  geom_bar(stat = "identity")
```

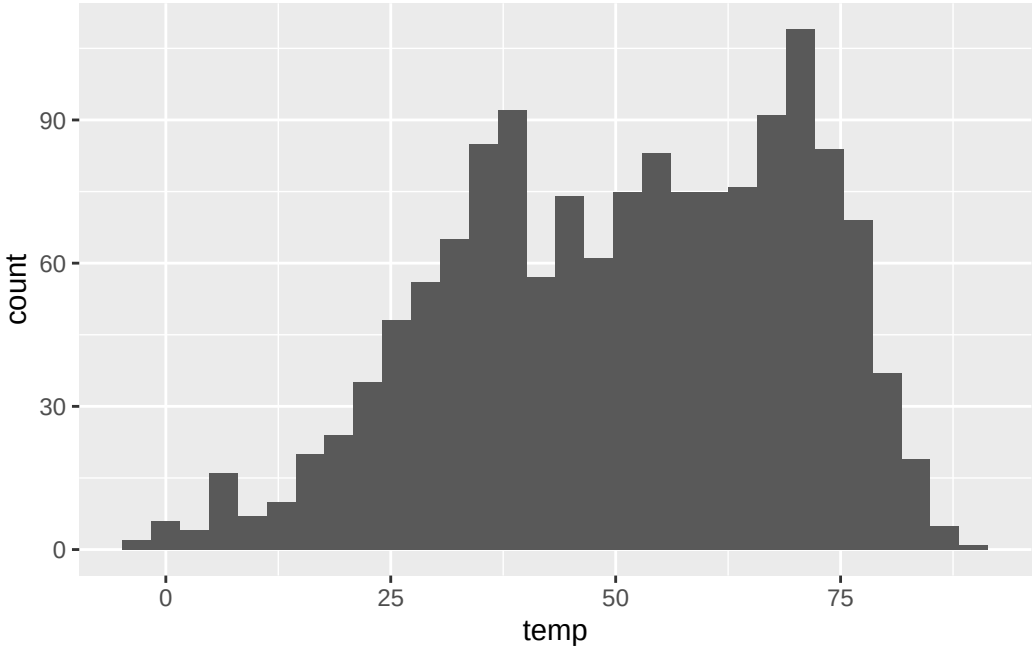


Position

The position argument is used to tweak how certain aspects of the plots are displayed. Its use depends heavily on the type of plot. For each geom, some positions will work, some will do nothing, and some will produce nonsense. They are most commonly used when trying to create grouped plots.

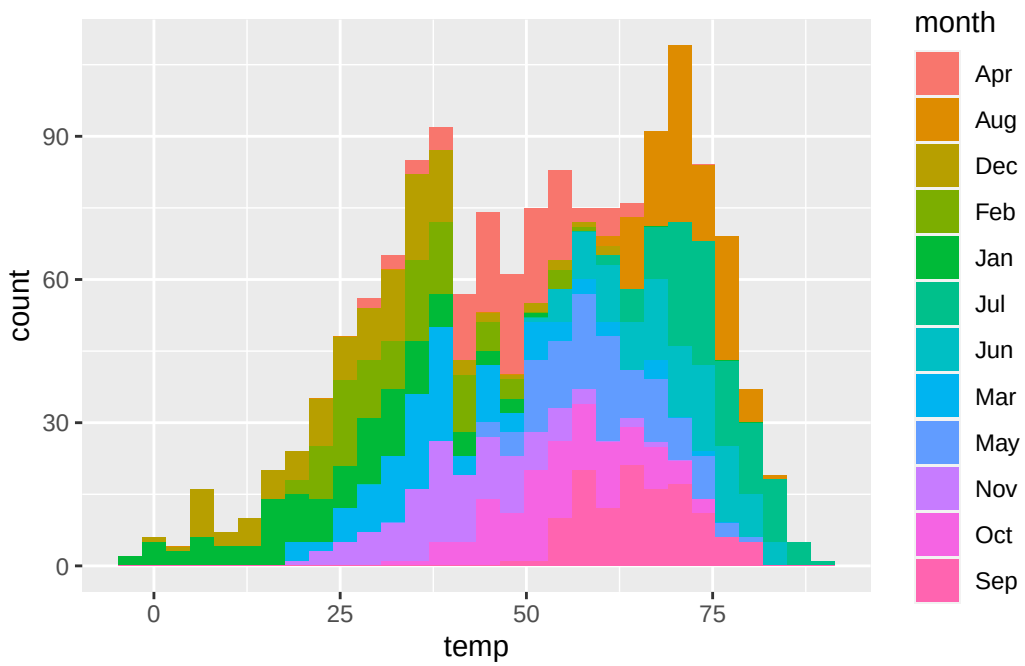
For example, we can look at a histogram of `temp`.

```
ggplot(nnmaps, aes(x = temp)) + geom_histogram()
```

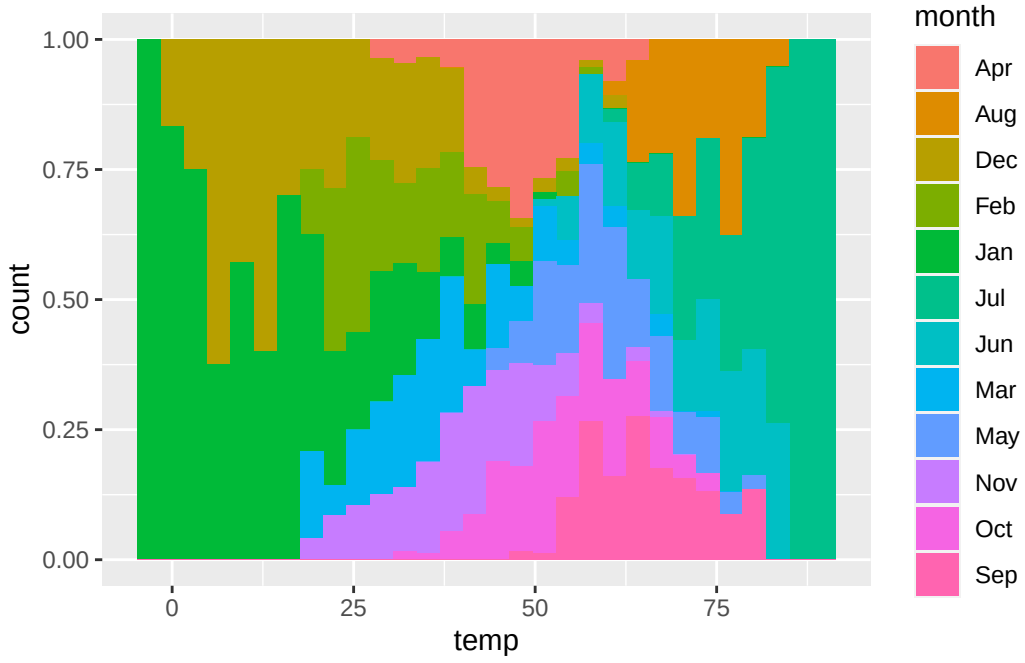


If we add `fill = month`, it will group by class. The default is `position = "stack"`; another nice option is `fill`

```
nnmaps2 <- nnmaps[!is.na(nnmaps$month), ]
ggplot(nnmaps2, aes(x = temp, fill = month)) +
  geom_histogram()
```




```
ggplot(nnmaps2, aes(x = temp, fill = month)) +
  geom_histogram(position = "fill")
```



In general:

- `identity` is useful when you want to plot things exactly as they are.
- `stack` is useful when you want to look both at overall values and per-group values.
- `dodge` is useful for group comparisons.
- `fill` is useful for considering percentages instead of counts.
- `jitter` is useful for scatter plots (and similar) when multiple values may be placed at the same point.

Positions are generally a trial-and-error procedure for me. If the default isn't sufficient, see if the others look/work better.

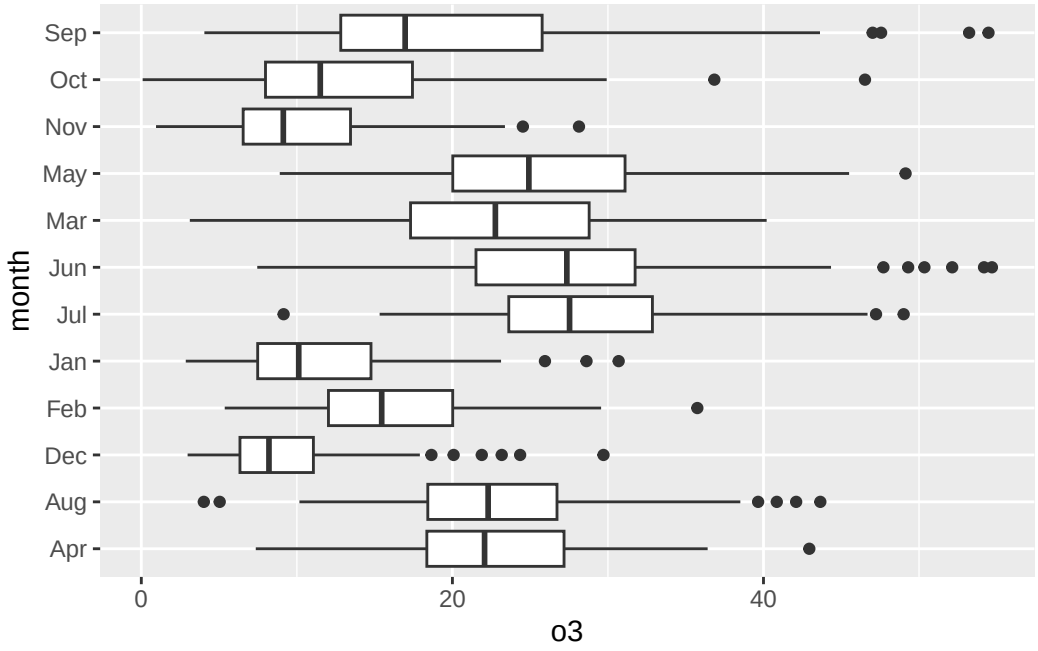
Coordinate

The default coordinate system is `coord_cartesian()`. There are two useful and two niche different coordinate systems.

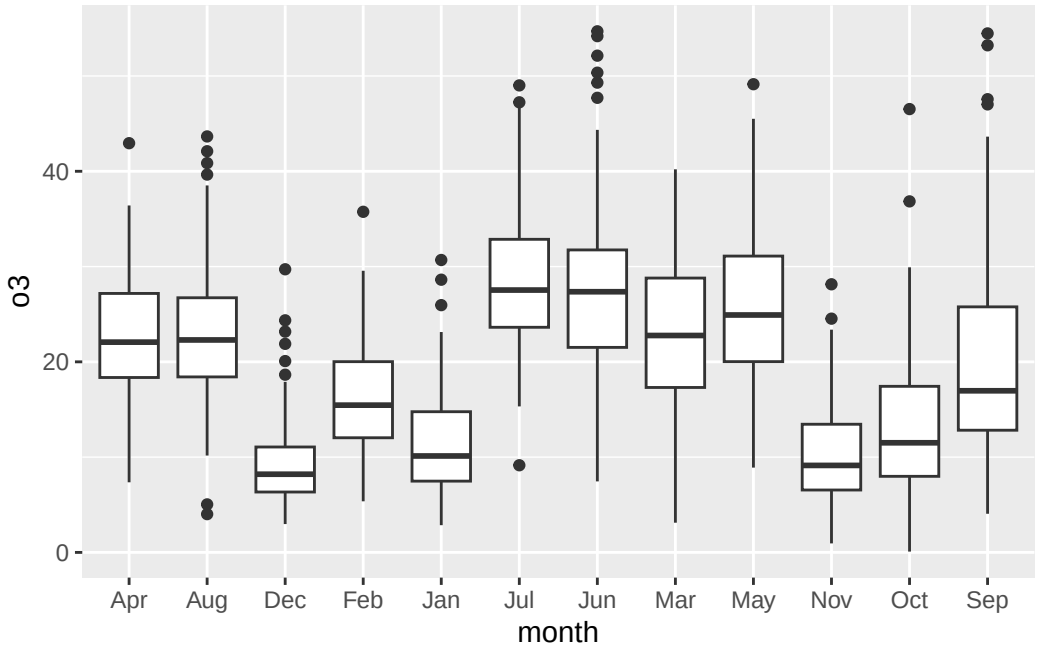
- `coord_fixed` forces the x and y axes to have a fixed ratio between units on each. (Default ratio is 1:1, it takes an argument to define the ratio)
- `coord_flip` flips x and y. Most useful to get e.g. horizontal instead of vertical bar charts.
- `coord_map` plots map data
- `coord_polar` plots on the polar coordinate system.

Demonstrations of `coord_fixed` and `coord_flip`:

```
g <- ggplot(nnmaps2, aes(x = o3, y = month)) +
  geom_boxplot()
g
```

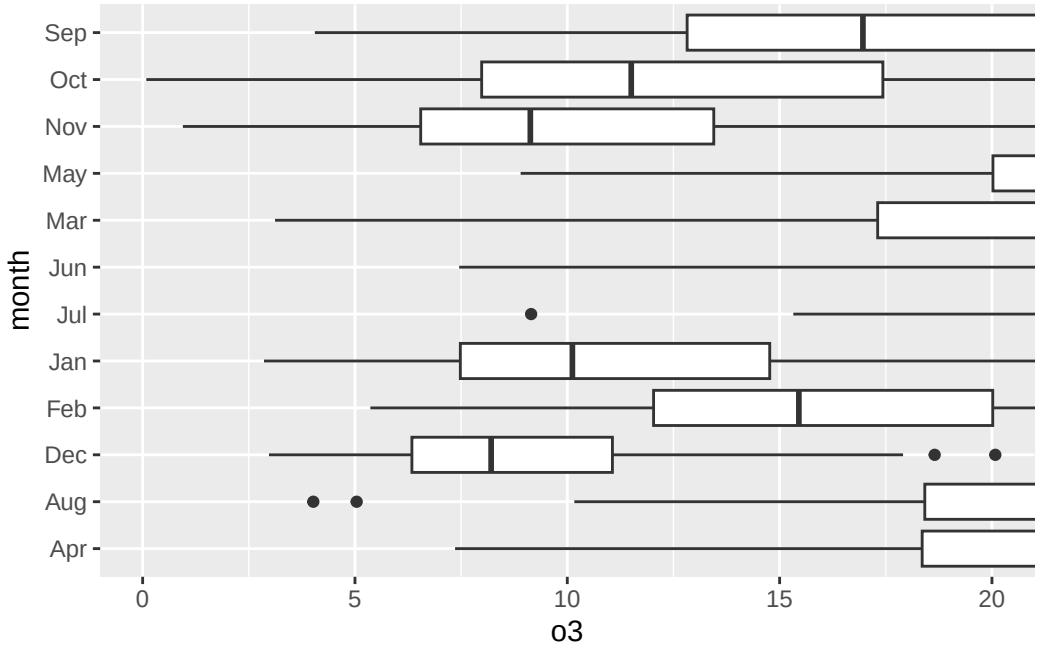

Sep
Oct
Nov
May
Mar
Jun
Jul
Jan
Feb
Dec
Aug
Apr
month
0
20
40
o3

```
g + coord_flip()
```


40
20
0
o3
Apr
Aug
Dec
Feb
Jan
Jul
Jun
Mar
May
Nov
Oct
Sep
month

`coord_cartesian` (as well as `_fixed` and `_flip`) take `xlim` and `ylim` arguments.

```
g + coord_cartesian(xlim = c(0, 20))
```

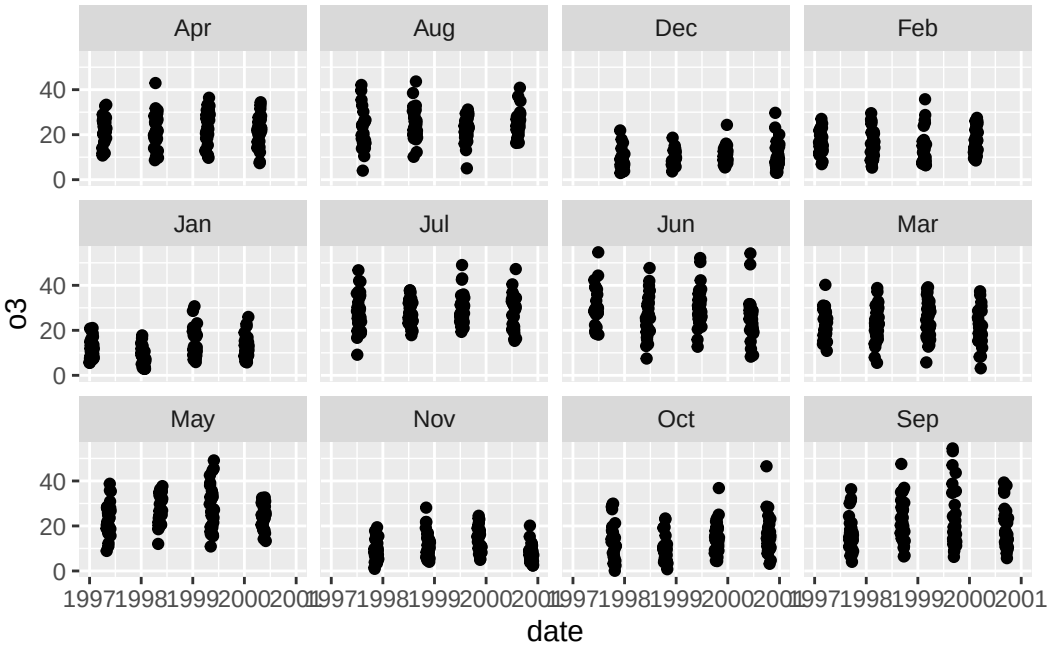


Facet

While `gg` objects support using `par(mfrow...` to include multiple plots on one Device, the built-in faceting system can also be very useful to produce a grid of plots.

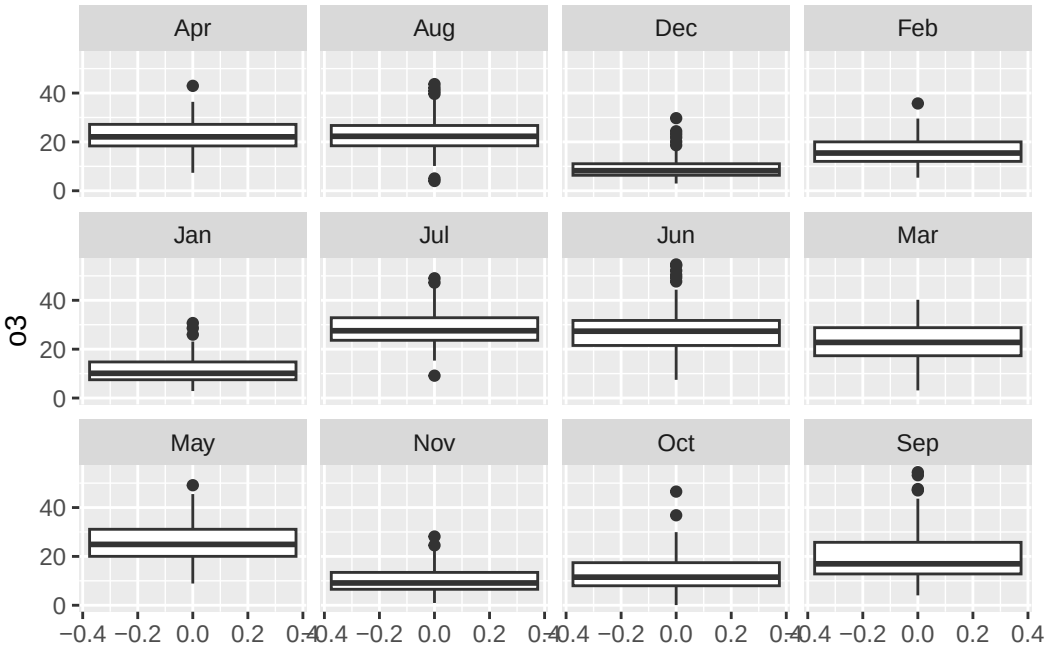
In most cases, use `facet_wrap`, which will automatically pick the right number of rows and columns.

```
ggplot(nnmaps2, aes(x = date, y = o3)) +
  geom_point() +
  facet_wrap(vars(month))
```



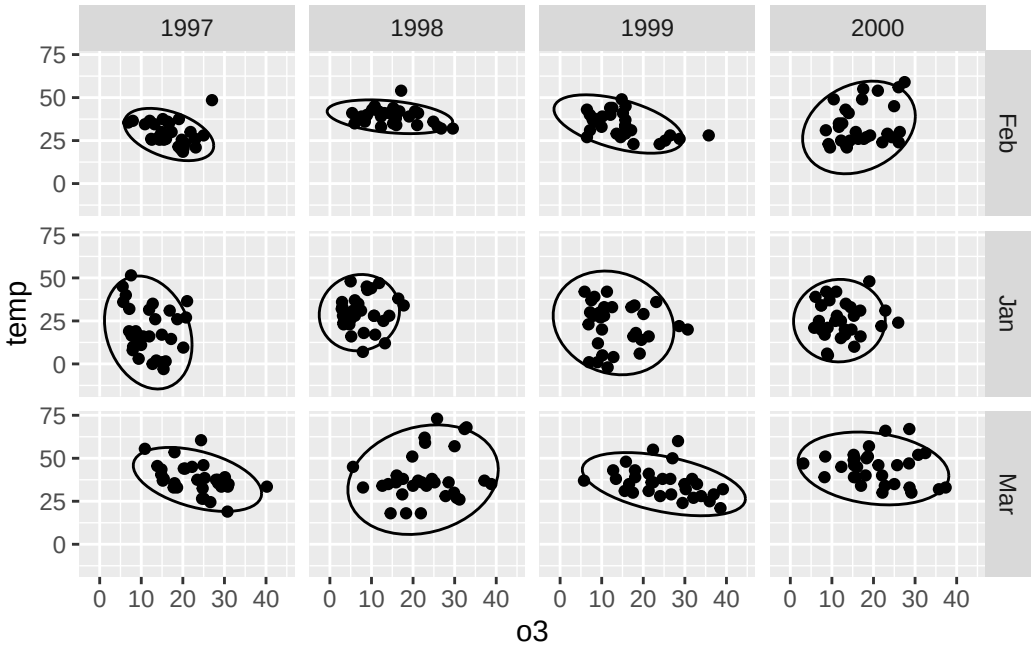
Note the use of `vars(month)` due to tidyverse's non-standard evalution. Passing `"month"` works just as well.

```
ggplot(nnmaps2, aes(x = o3)) +
  geom_boxplot() +
  coord_flip() +
  facet_wrap("month")
```



`facet_grid` is more precise; by default it does not "wrap", producing a single row of plots. You can pass in two variables to facet by each variable in one direction.

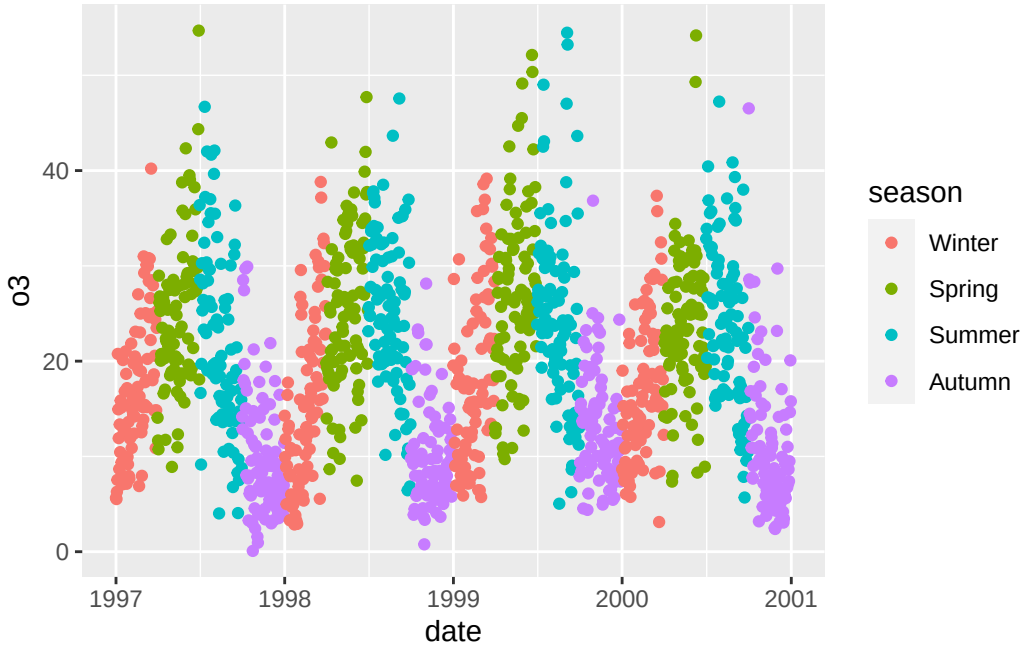
```
nnmaps3 <- nnmaps2[nnmaps2$month %in% c("Jan", "Feb", "Mar"), ]
ggplot(nnmaps3, aes(x = o3, y = temp)) +
  geom_point() +
  stat_ellipse() +
  facet_grid(vars(month), vars(year))
```



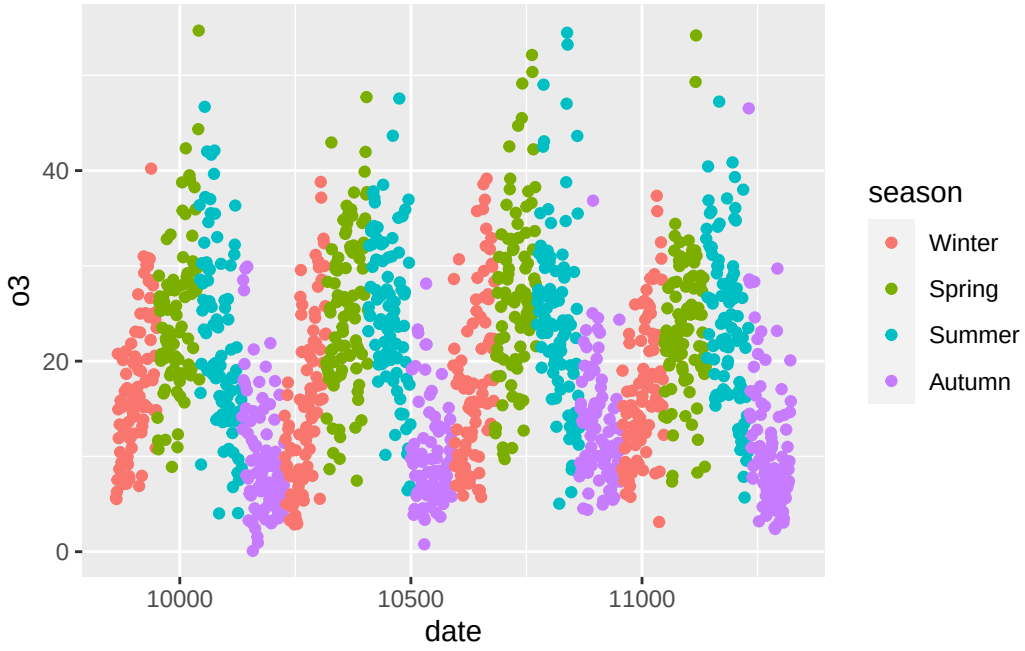
Scale

Scales are used to map data onto the plot. When generating a new plot, certain default scales are used behind the scenes. For example:

```
ggplot(nnmaps, aes(x = date, y = o3, color = season)) +
  geom_point()
```

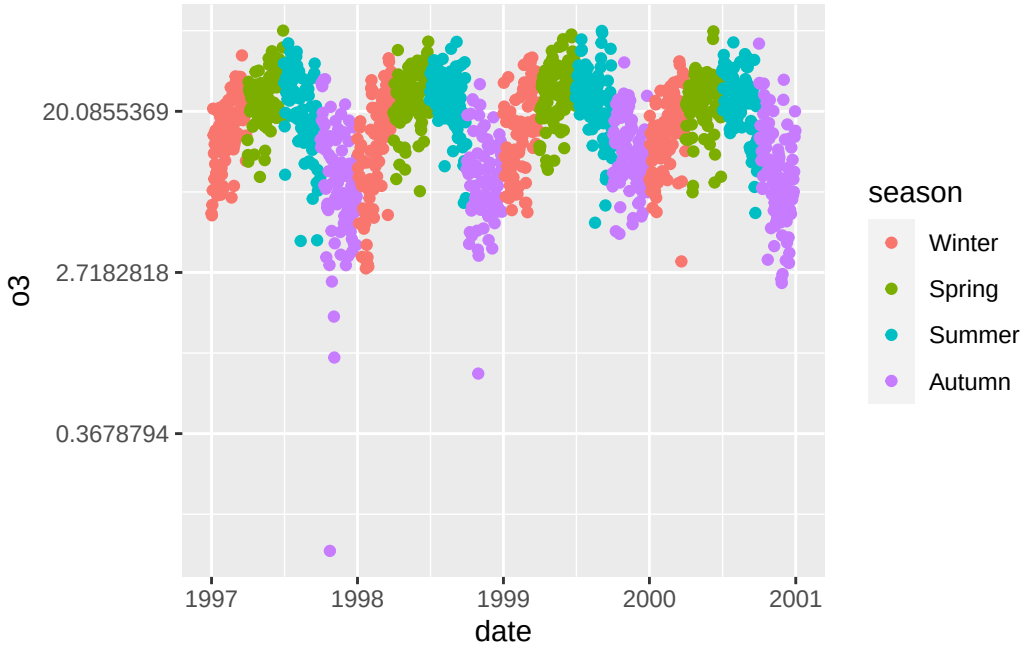



```
ggplot(nnmaps, aes(x = date, y = o3, color = season)) +
  geom_point() +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_color_discrete()
```



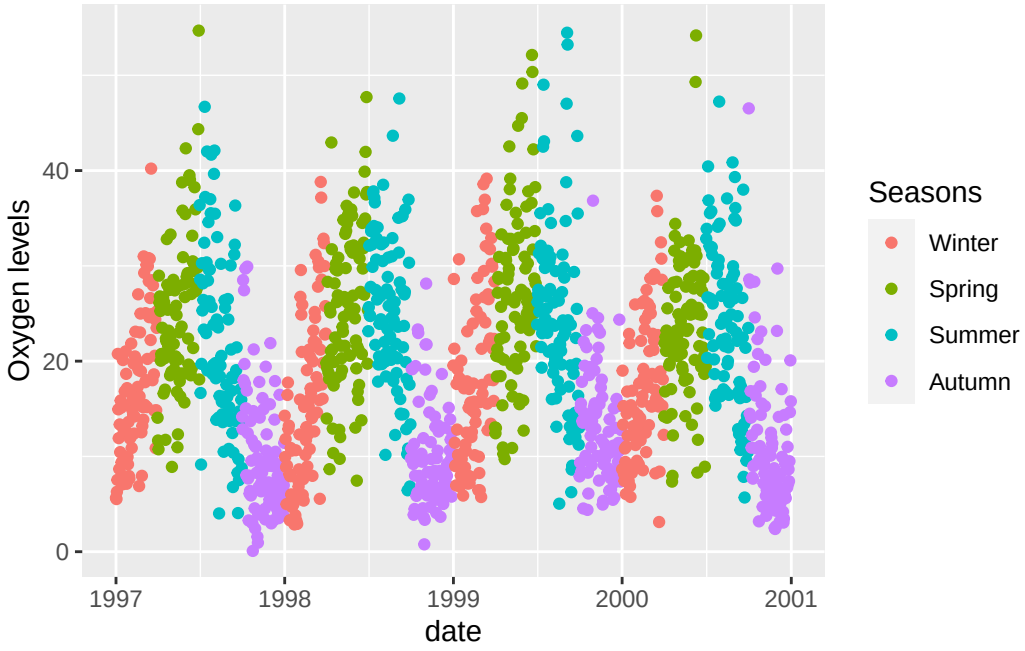
Scales can change many aspects of the mapping from the data to the visualization. For example, to plot the o3 value on the log scale:

```
ggplot(nnmaps, aes(x = date, y = o3, color = season)) +
  geom_point() +
  scale_y_continuous(trans = "log")
```



They can also be used just to manipulate labels and such.

```
ggplot(nnmaps, aes(x = date, y = o3, color = season)) +
  geom_point() +
  scale_y_continuous(name = "Oxygen levels") +
  scale_color_discrete(name = "Seasons")
```

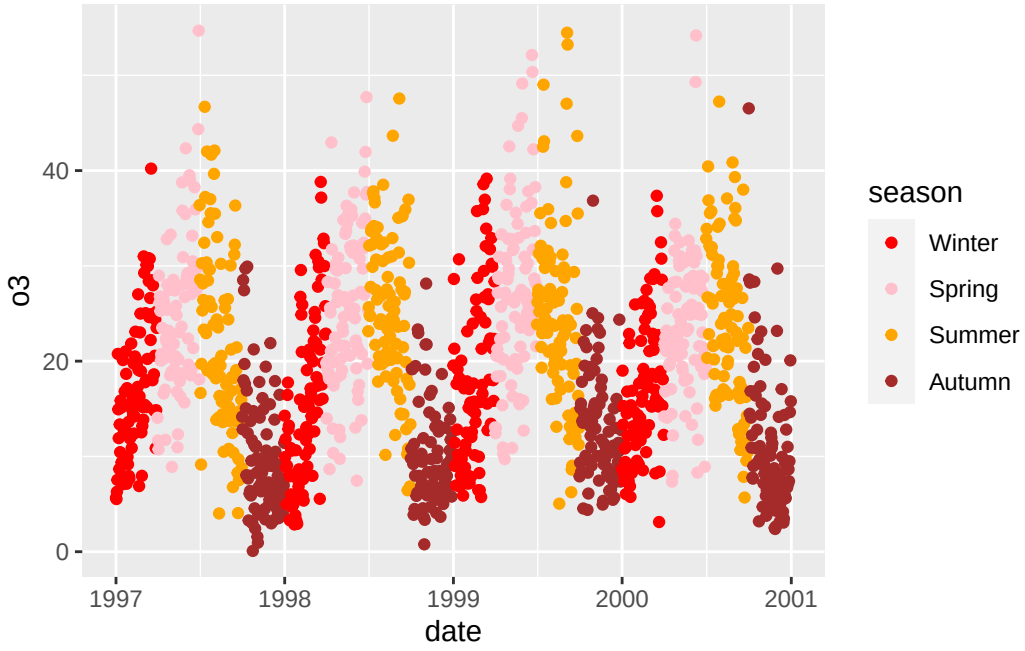


We can also use scales to completely change how a variable is treated.

```
ggplot(nnmaps, aes(x = date, y = o3, color = season)) +
  geom_point() +
  scale_y_binned()
```

Finally, we can use scales to change colors

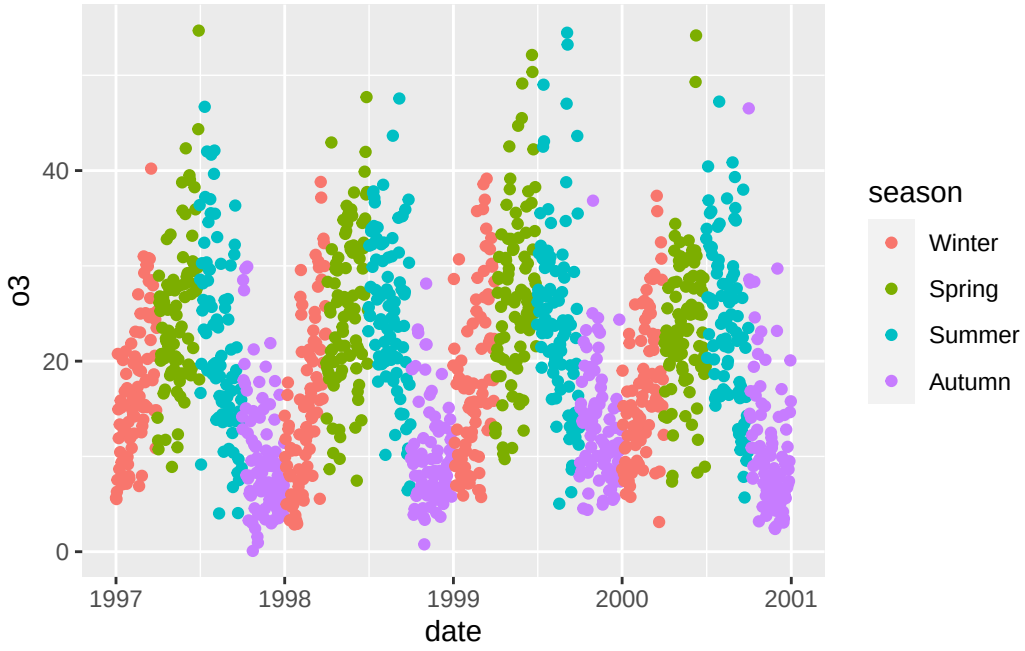
```
ggplot(nnmaps, aes(x = date, y = o3, color = season)) +
  geom_point() +
  scale_color_manual(values = c("red", "pink", "orange", "brown"))
```



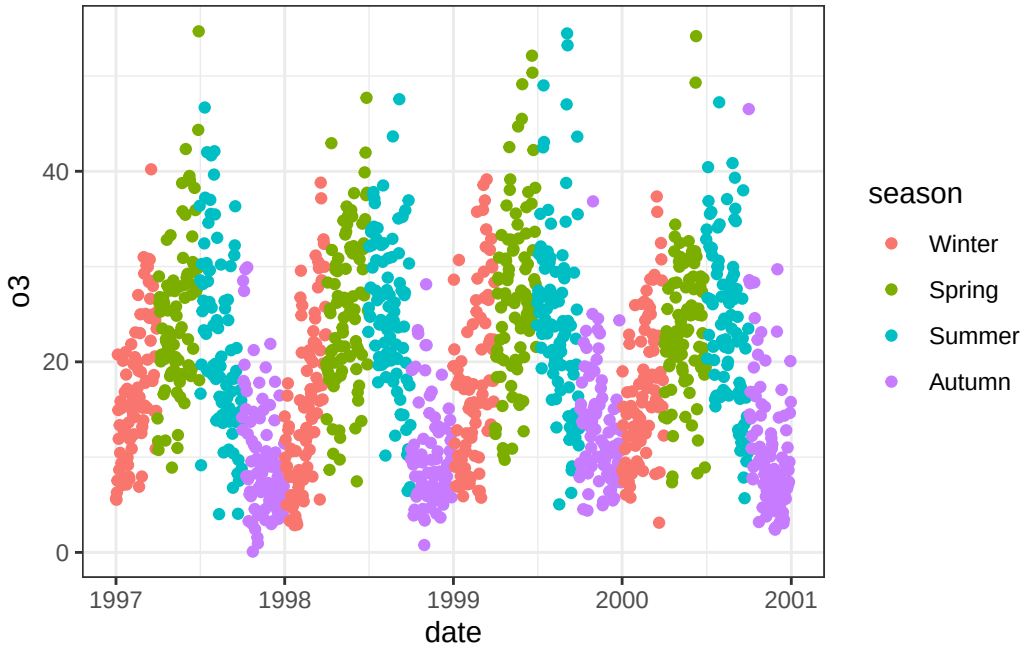
Theme

There are default themes, and you can customize themes.

```
g <- ggplot(nnmaps, aes(x = date, y = o3, color = season)) +
  geom_point()
g
```

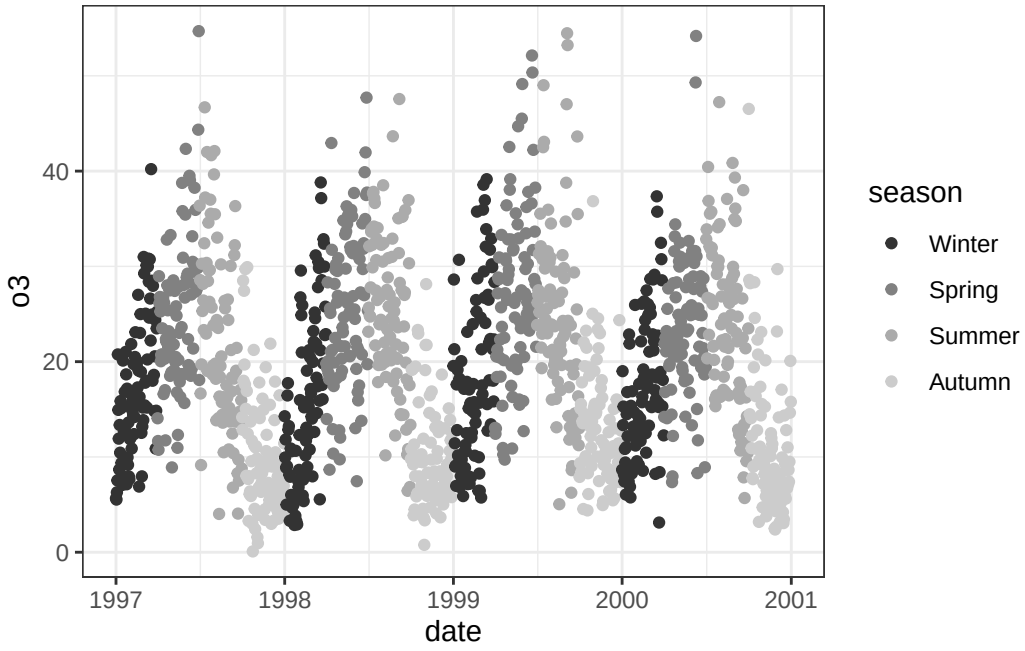

40
20
0
o3
1997
1998
1999
2000
2001
date
season
Winter
Spring
Summer
Autumn

```
g + theme_bw()
```


40
20
0
o3
1997
1998
1999
2000
2001
date
season
Winter
Spring
Summer
Autumn

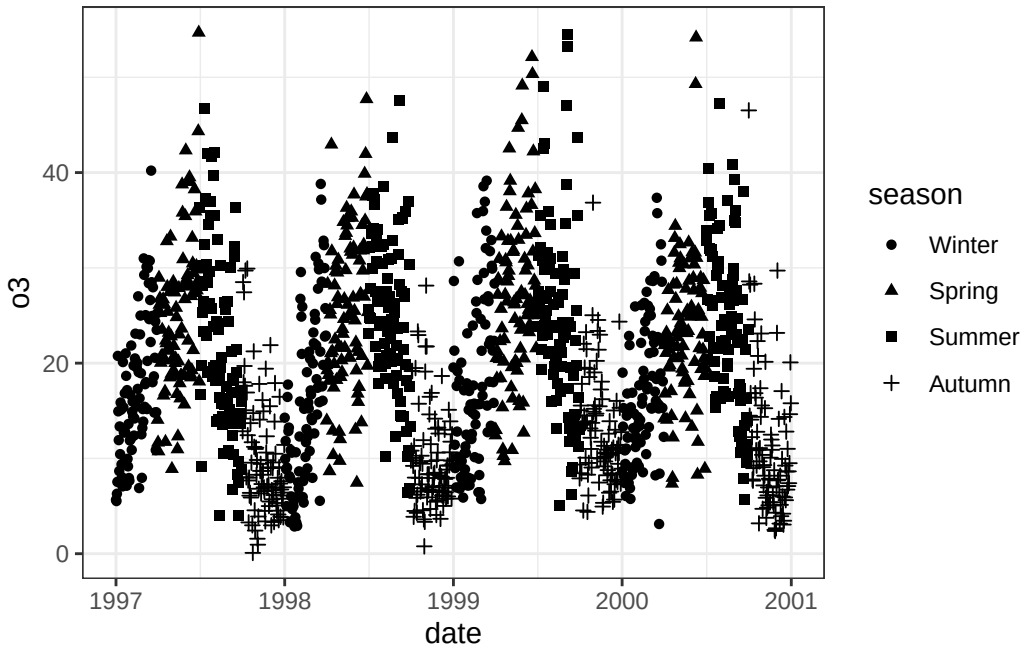
Themes refer to only the ploptting area and around; you'd need to modify the color scales to go fully black and white for example.

```
g +
  theme_bw() +
  scale_colour_grey()
```



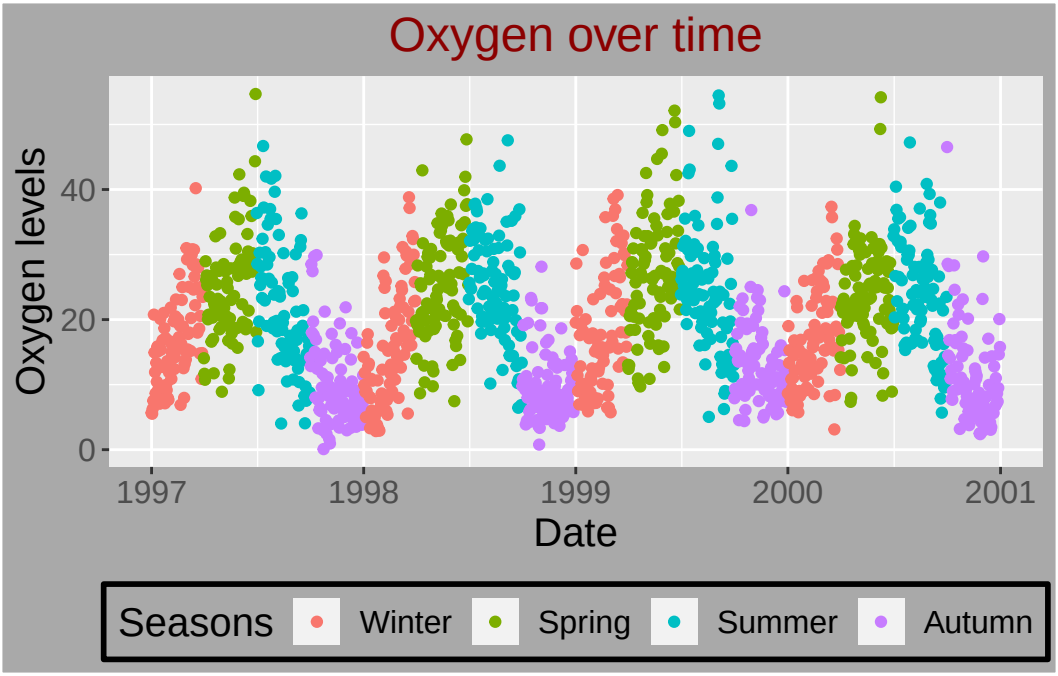
Perhaps even better would be to change the plotting characters.

```
ggplot(nnmaps, aes(x = date, y = o3, shape = season)) +
  geom_point(color = "black") +
  theme_bw()
```



The generic `theme` function allows more fine-grained control.

```
g +
  ggtitle("Oxygen over time") +
  scale_y_continuous(name = "Oxygen levels") +
  scale_x_date(name = "Date") +
  scale_color_discrete(name = "Seasons") +
  theme(plot.title = element_text(color = "darkred", hjust = .5),
        legend.position = "bottom",
        legend.background = element_rect(linewidth = 1,
                                         color = "black",
                                         fill = "darkgrey"),
        plot.background = element_rect(fill = "darkgrey"),
        text = element_text(size = 15))
```



Representing higher dimensions in scatter plots

Although scatter plots are inherently a two (or three) dimensional visualization, clever use of plot characteristics can represent higher dimensions. There is always a trade-off - the more information you represent in a single plot, the more likely you are to confuse the reader.

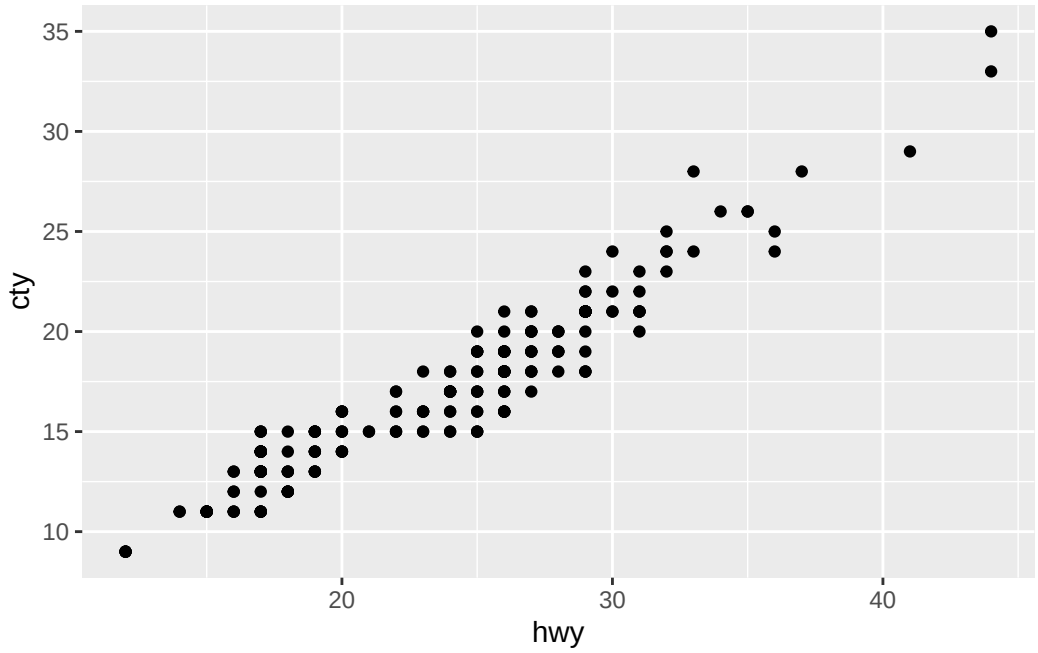
We'll be using the `mpg` dataset from ggplot2. Our primary variables of interest will be city mileage (`cty`) versus highway mileage (`hwy`), and we'll be adding other variables as we go.

```
library(ggplot2)
str(mpg)
```

```
tibble [234 x 11] (S3: tbl_df/tbl/data.frame)
 $ manufacturer: chr [1:234] "audi" "audi" "audi" "audi" ...
 $ model       : chr [1:234] "a4" "a4" "a4" "a4" ...
 $ displ       : num [1:234] 1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
 $ year        : int [1:234] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
 $ cyl         : int [1:234] 4 4 4 4 6 6 6 4 4 4 ...
 $ trans       : chr [1:234] "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
 $ drv         : chr [1:234] "f" "f" "f" "f" ...
 $ cty         : int [1:234] 18 21 20 21 16 18 18 18 16 20 ...
```

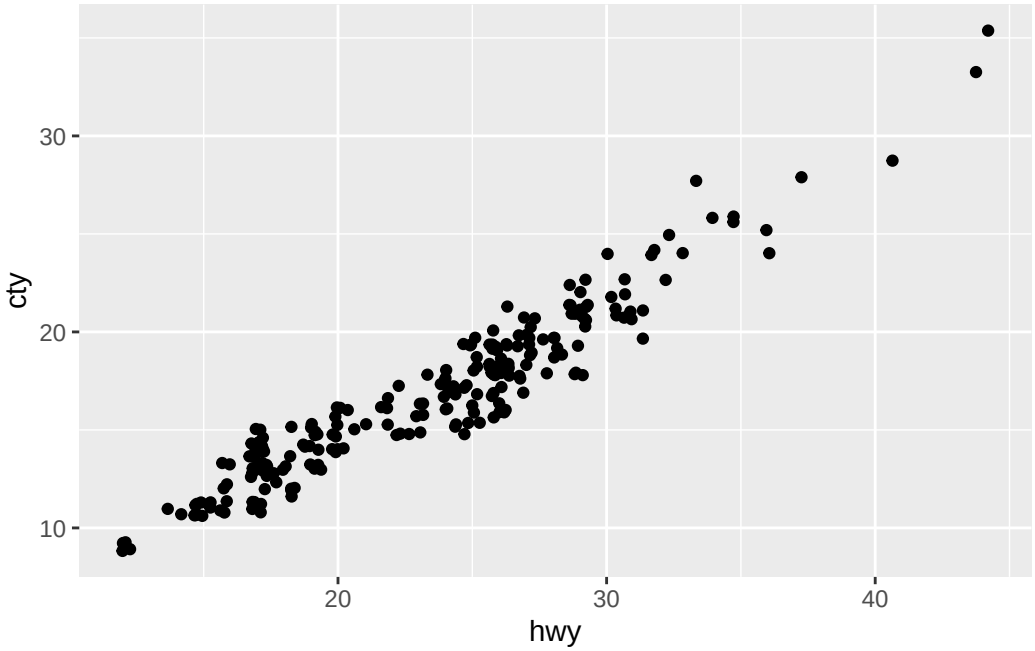
```
$ hwy          : int [1:234] 29 29 31 30 26 26 27 26 25 28 ...
$ fl           : chr [1:234] "p" "p" "p" "p" ...
$ class        : chr [1:234] "compact" "compact" "compact" "compact" ...
```

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



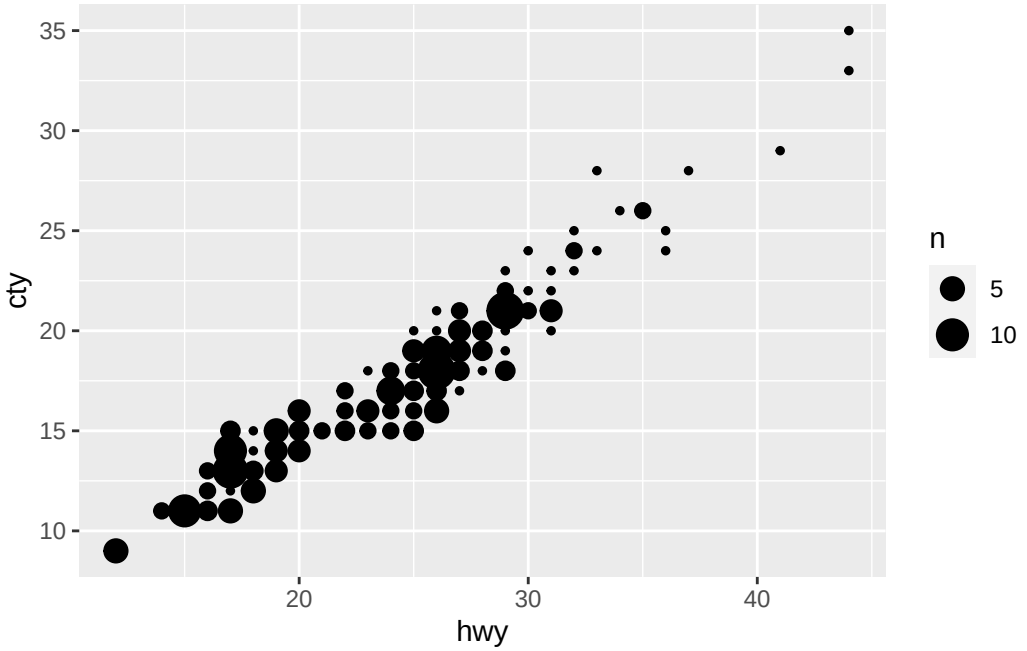
Before we add any higher dimensions, note that each point in that plot may represent multiple cars, e.g. there are a Jeep Grand Cherokee and a Toyota Tacoma that each have 17 `cty` and 22 `hwy`. To make that clear, we can either use jitter:

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



or make the size of each point relative to the count:

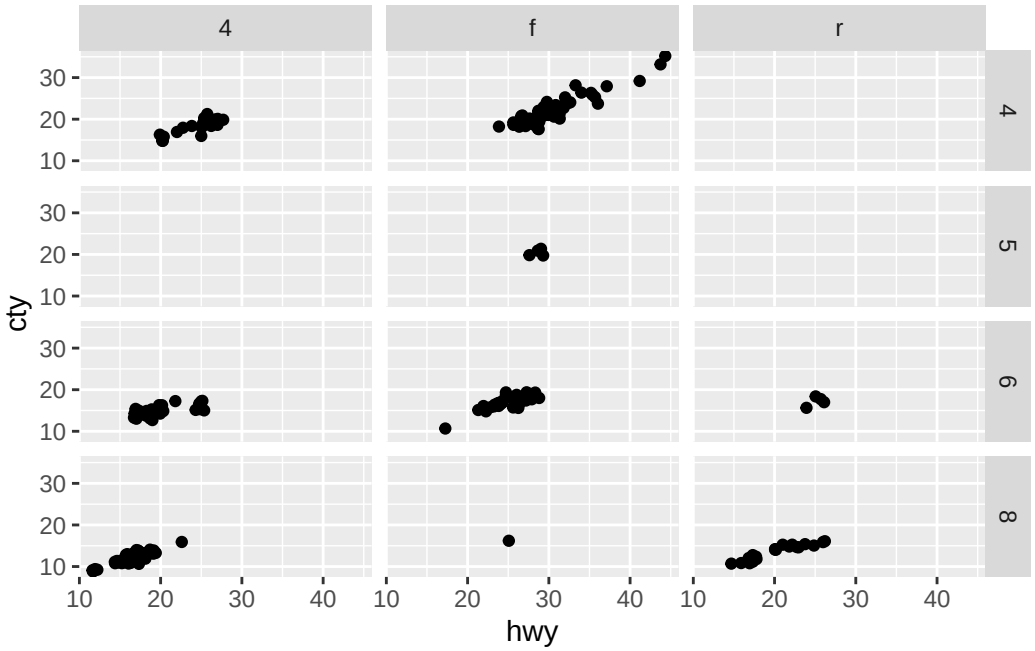
```
ggplot(mpg, aes(x = hwy, y = cty)) +
  geom_count()
```



The `geom_count` version looks nicer, but changing the size of the points is a technique we'll use to represent a 3rd dimension, so we'll stick with the `geom_jitter` version.

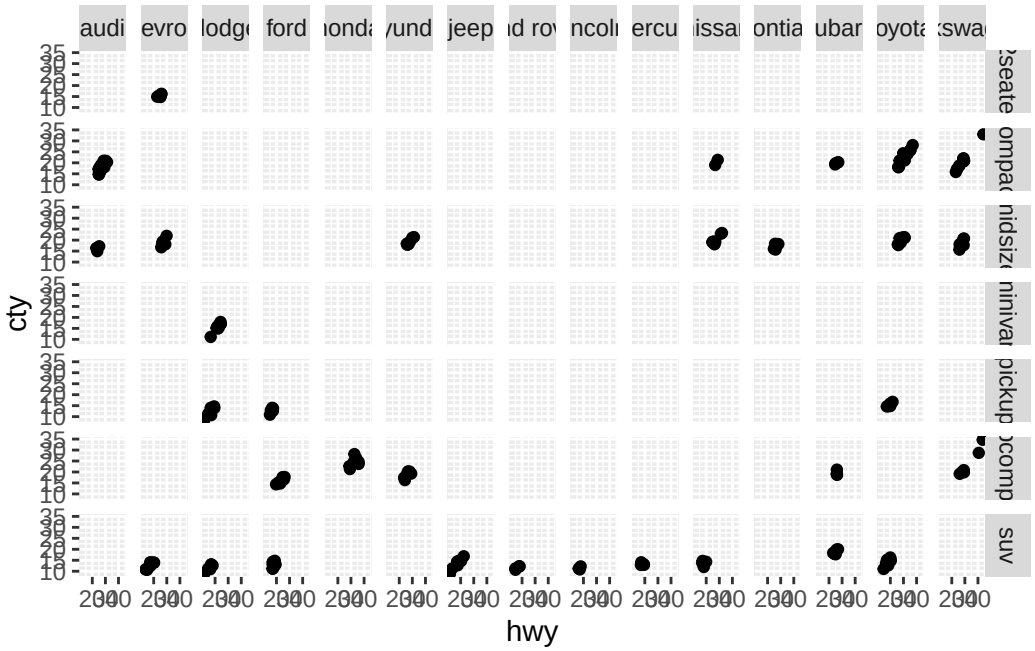
Displaying a 3rd (or higher) dimension of categorical variables is straightforward and you may have already used some or all of these methods. The most straightforward is faceting which we saw earlier:

```
ggplot(mpg, aes(x = hwy, y = cty)) +
  geom_jitter() +
  facet_grid(vars(cyl), vars(drv))
```



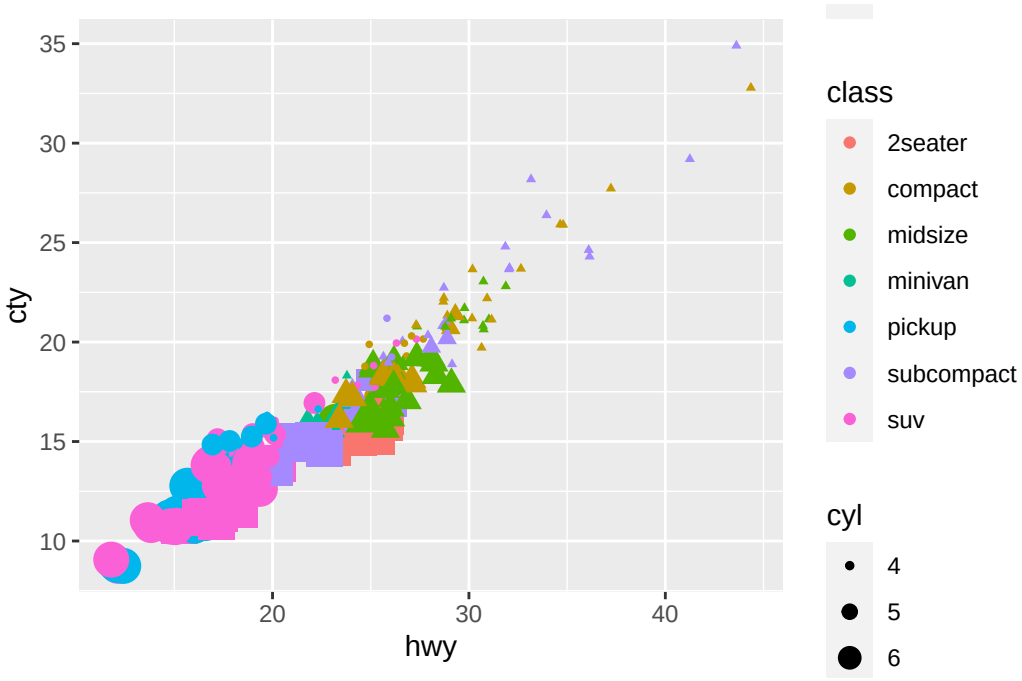
This allows us to represent four dimensions painlessly, as long as the two categorical dimensions have few unique values:

```
ggplot(mpg, aes(x = hwy, y = cty)) +
  geom_jitter() +
  facet_grid(vars(class), vars(manufacturer))
```



If we wanted a single plot, we can manipulate the points drawn to be distinct per group.

```
ggplot(mpg, aes(x = hwy, y = cty, color = class, shape = drv, size = cyl)) +
  geom_jitter() +
  scale_radius()
```



(Note the `+ scale_radius()`. When using the mapping `size`, I sometimes see issues with the smallest point being out of proportion with the rest. [Try it - run the above without the last line.] Adding `scale_radius()` [as opposed to `scale_size()`] can sometimes fix it - try both options and choose whichever one you like best.)

We've gotten messy here (probably too confusing for publication) but this is plotting 5 dimensions! From this plot we can see:

- Mileage in city or highway is strongly correlated.
- More cylinders = worse mileage.
- 4 wheel drive is less efficient than rear-wheel which is less efficient than front-wheel.
- SUV's and pickups have the worse mileage; 2-seaters, minivans and midsize have moderate mileage; subcompacts and compacts have the best mileage.

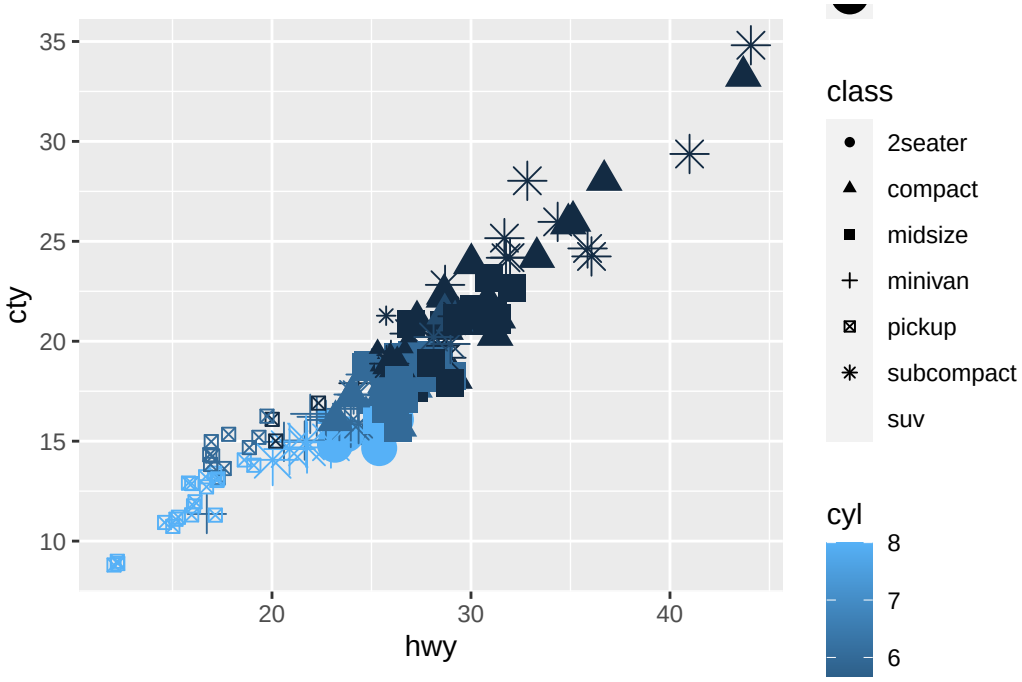
Note the choice of which variable gets which modification. `cyl`, which is ordinal, is used for the size. There are more `class` levels than `drv`, so I feel `class` is a better choice for colors. Look how bad this could look with different choices:

```
ggplot(mpg) +
  geom_jitter(aes(x = hwy, y = cty, color = cyl, shape = class, size = drv))
```

```
Warning: Using size for a discrete variable is not advised.
```

```
Warning: The shape palette can deal with a maximum of 6 discrete values because
more than 6 becomes difficult to discriminate; you have 7. Consider
specifying shapes manually if you must have them.
```

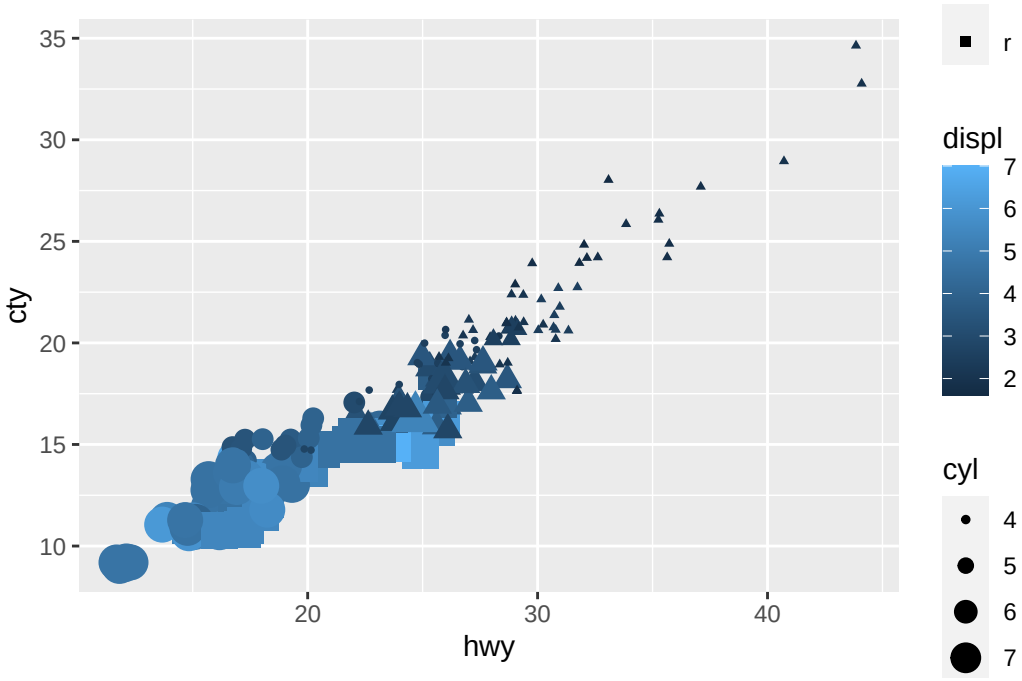
```
Warning: Removed 62 rows containing missing values (`geom_point()`).
```



ggplot even complains… a lot.

Note that color functions differently for continuous data:

```
ggplot(mpg) +
  geom_jitter(aes(x = hwy, y = cty, color = displ, shape = drv, size = cyl)) +
  scale_radius()
```



We see that larger `displ` (displacement, a.k.a size) corresponds to worse mileage.