Data Visualization with ggplot2 (Part 3)

ggplot2 internals (Chapter 3)

Seun Odeyemi

2019-04-20

Contents

Load Libraries

```
library(readr)
library(dplyr)
library(ggplot2)
# library(ggplot2movies)
library(tidyr)
library(skimr)
library(knitr)
library(kableExtra)
library(RColorBrewer)
library(grid)
library(ggthemes)
library(forcats)
library(GGally)
#> Warning: package 'GGally' was built under R version 3.5.3
library(here)
library(hexbin)
#> Warning: package 'hexbin' was built under R version 3.5.3
```

Grid Graphics

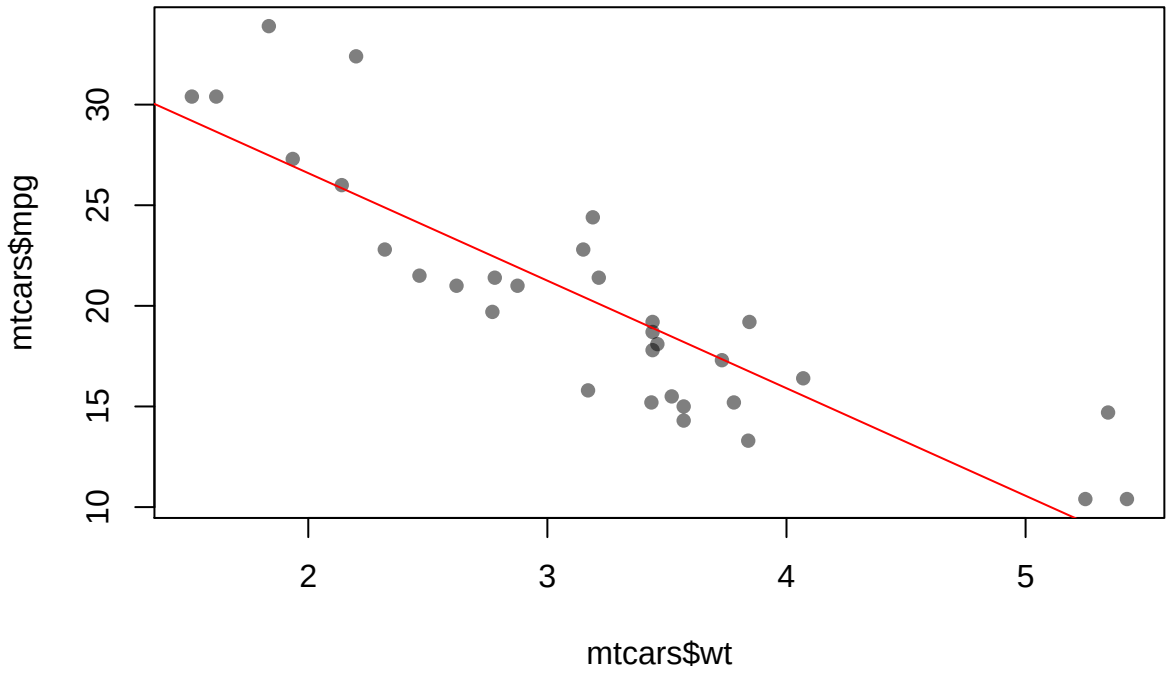
We will delve into the internals of `ggplot2`. To begin we're going to explore the grid graphics system. The functions and concepts of this video will provide an idea of how graphics work in R. There are essentially two plotting systems in R:

1. Base Package
2. Grid graphics

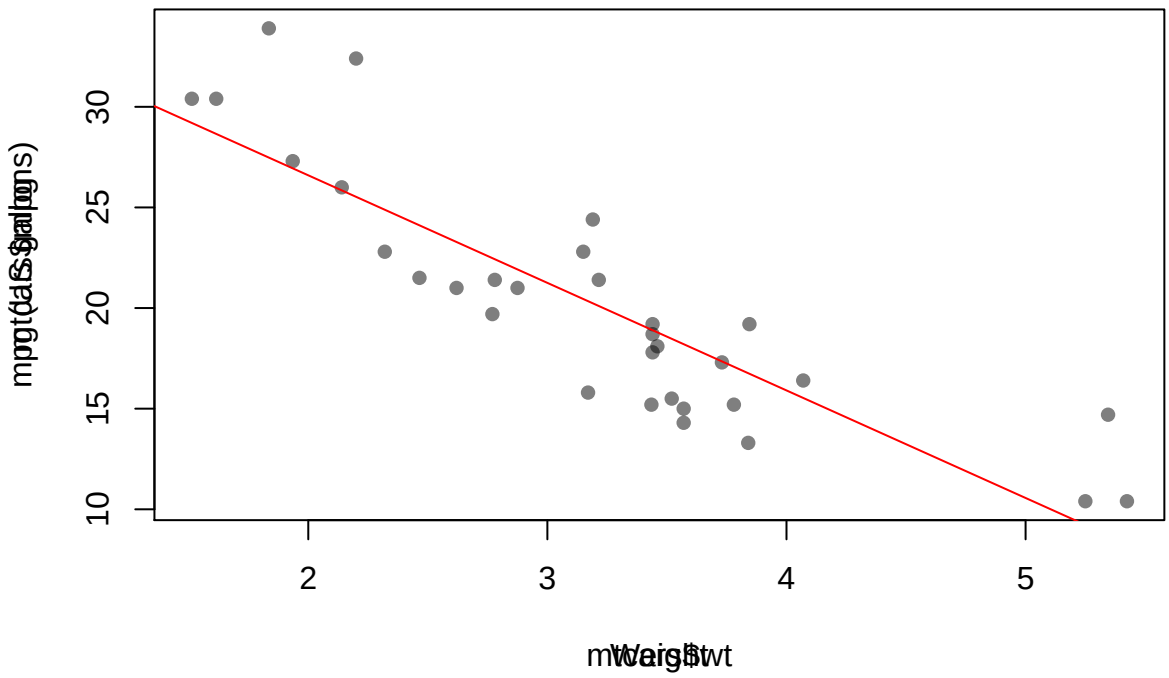
Base Package

We saw base package earlier in the series. The base package treats the graphic device as a static canvas as shown below. We can accessory functions like `abline` to add visual elements to an already existing plot.

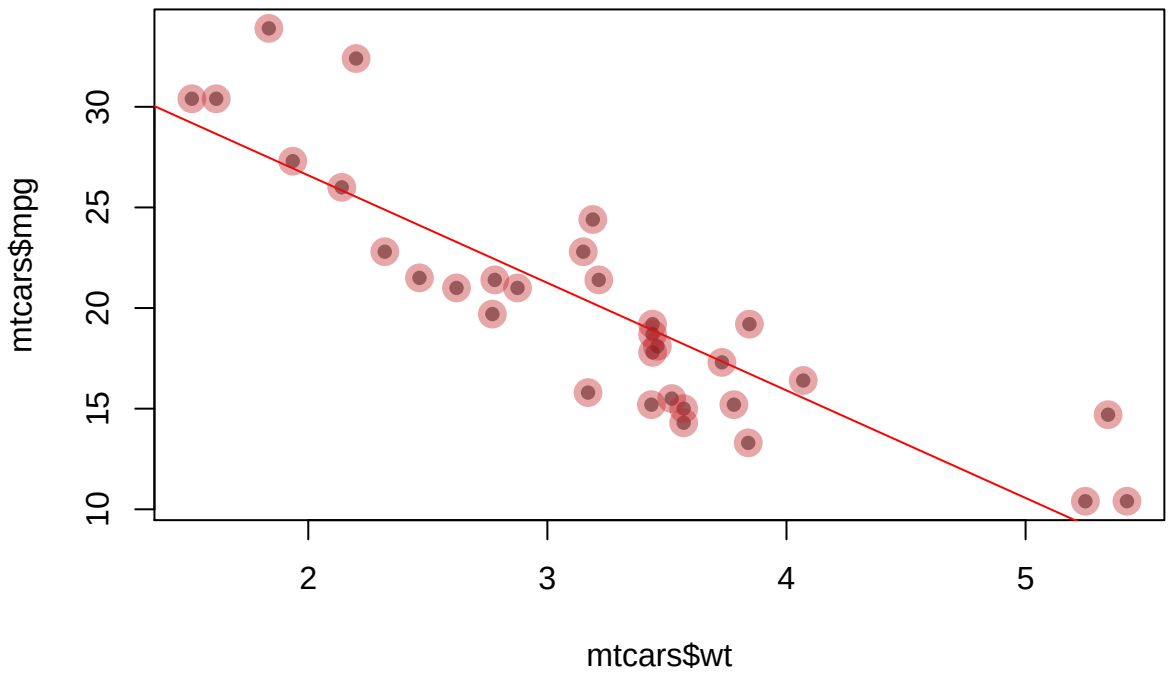
```
# a scatter plot of mtcars variable wt and mpg

plot(mtcars$wt, mtcars$mpg, pch = 16, col = "#00000080")
abline(lm(mpg ~ wt, data = mtcars), col = "red")
```



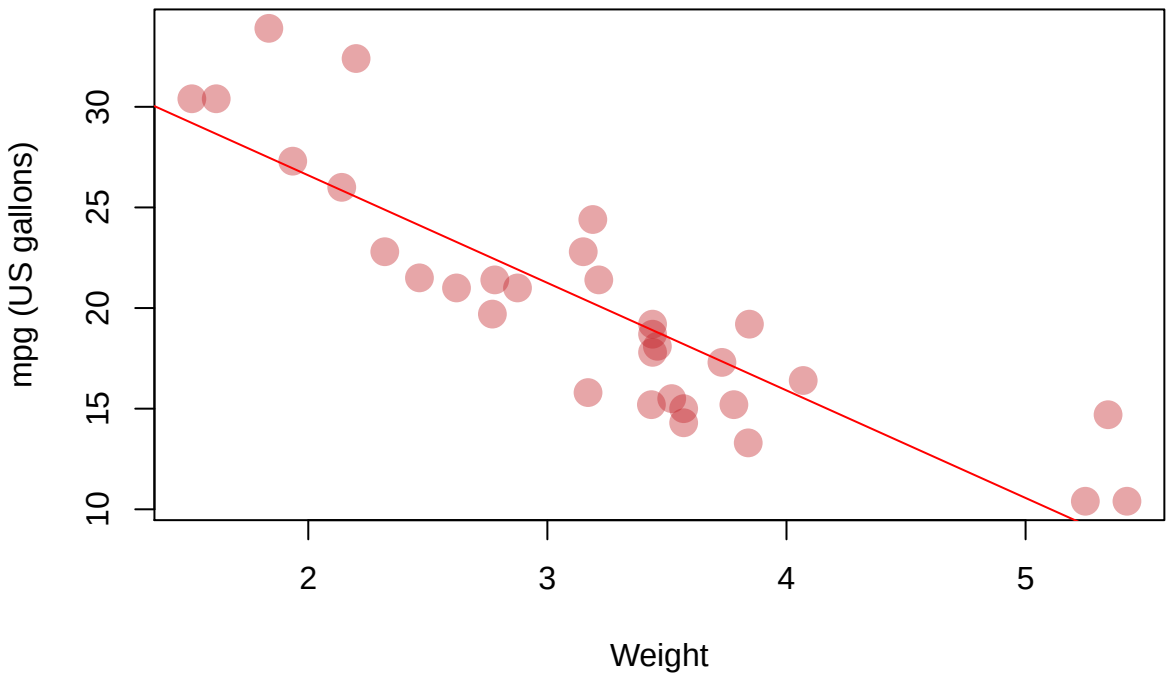

```
# base package - change labels
plot(mtcars$wt, mtcars$mpg, pch = 16, col = "#00000080")
abline(lm(mpg ~ wt, data = mtcars), col = "red")
mtext("Weight", 1, 3)
mtext("mpg (US gallons)", 2, 3)
```




```
# base package - change dots
plot(mtcars$wt, mtcars$mpg, pch = 16, col = "#00000080")
abline(lm(mpg ~ wt, data = mtcars), col = "red")
points(mtcars$wt, mtcars$mpg, pch = 16,
col = "#C3212766", cex = 2)
```




```
# base package - restart
plot(mtcars$wt, mtcars$mpg, pch = 16, col = "#C3212766",
     cex = 2, xlab = "Weight", ylab = "mpg (US gallons)")
abline(lm(mpg ~ wt, data = mtcars), col = "red")
```



The `grid` package was developed by Paul Murell to overcome the deficiencies in the base package. It doesn't actually make plots by itself, it provides a set of low level functions that are used to construct complex plots.

> `ggplot2` is built on top of grid graphics

Two important components to grid graphics:

1. The ability to create a variety of graphic outputs
2. The ability to layer and position outputs with **viewports**

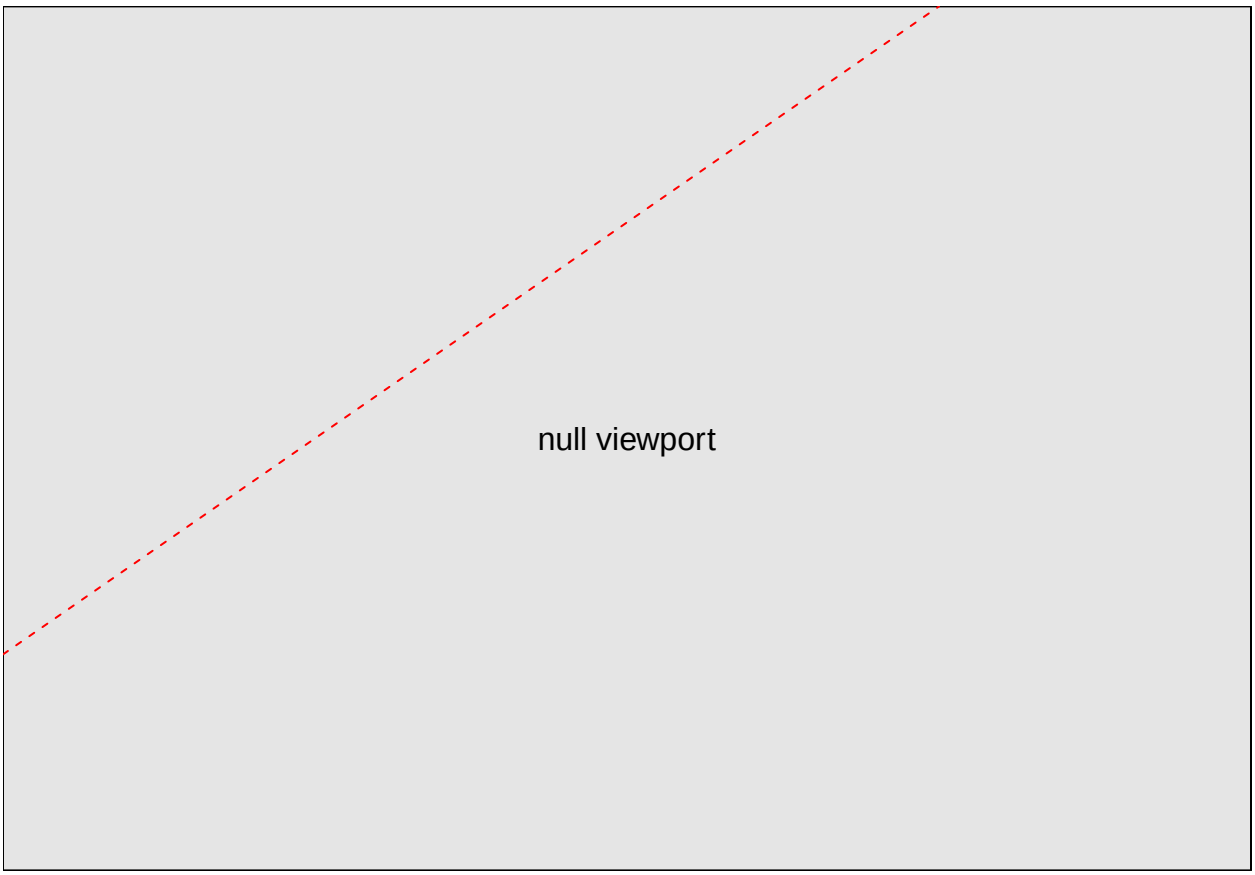
Viewport Basics (1)

To get familiar with grid graphics, you'll begin with using some `grid` functions. The `grid` package is already loaded into your R session, so you can get started straight away!

Note: In DataCamp's learning interface, each change you make to the plot will appear as a new plot, so you can see the effect of each command.

```
# Draw rectangle in null viewport
grid.rect(gp = gpar(fill = "grey90"))
# vp <- viewport(x = 0.5, y = 0.5, w = 0.5, h = 0.5,
#                just = "center")
# pushViewport(vp)

# Write text in null viewport
grid.text("null viewport")

# Draw a line
```

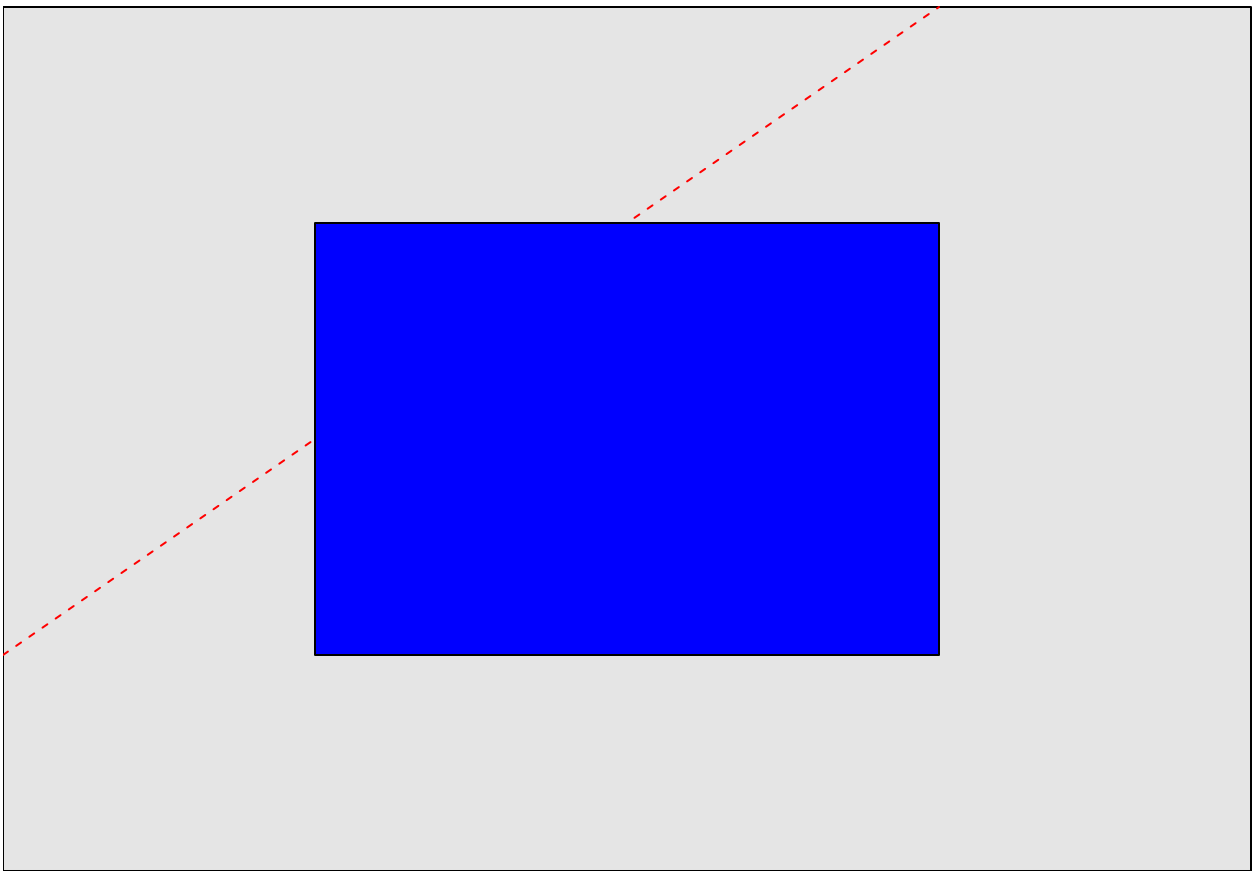
```
grid.lines(x = c(0, 0.75), y = c(0.25, 1),
           gp = gpar(lty = 2, col = "red"))
```



Viewport Basics (2)

The code from the previous exercise that populates the null viewport with some basic shapes is already available. Let's take the next step and start manipulating the stack of viewports.

```
# Populate null viewport
grid.rect(gp = gpar(fill = "grey90"))
grid.text("null viewport")
grid.lines(x = c(0,0.75), y = c(0.25, 1),
           gp = gpar(lty = 2, col = "red"))

# Create new viewport: vp
vp <- viewport(x = 0.5, y = 0.5, w = 0.5, h = 0.5, just = "center")

# Push vp
pushViewport(vp)

# Populate new viewport with rectangle
grid.rect(gp = gpar(fill = "blue"))
```

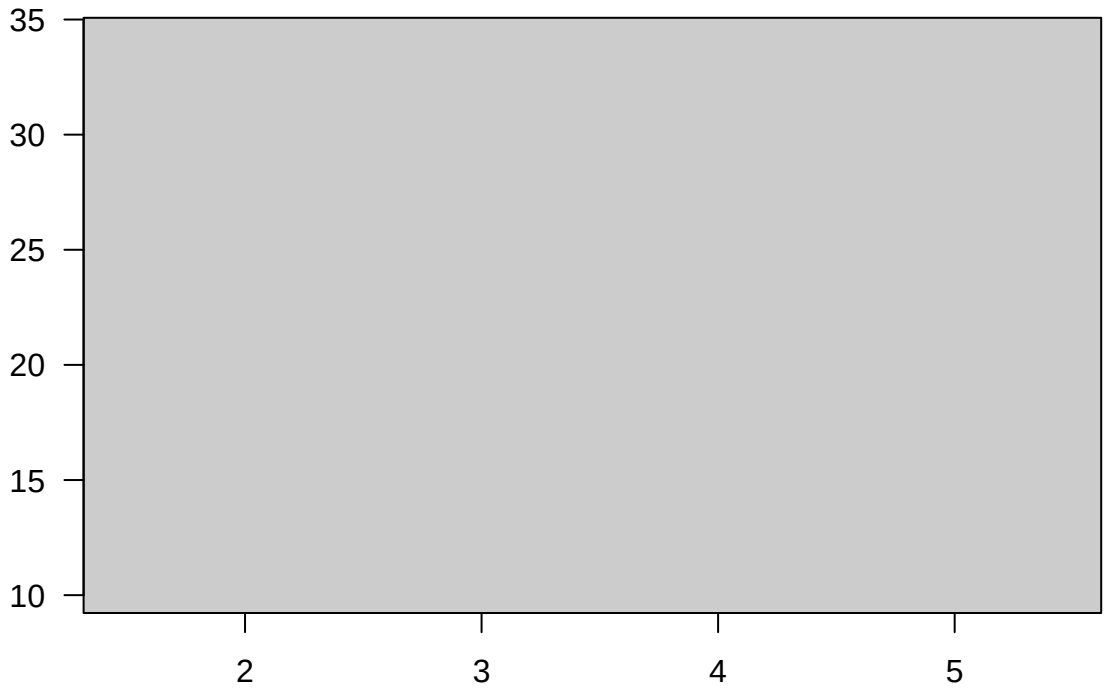


Build a Plot from Scratch (1)

Using the viewports, you can create plots, manipulating the space as needed.

In this exercise you'll establish your grid viewport and in the following exercise you'll populate it with values.

```
# 1 - Create plot viewport: pvp
mar <- c(5, 4, 2, 2)
pvp <- plotViewport(mar)

# 2 - Push pvp
pushViewport(pvp)

# 3 - Add rectangle
grid.rect(gp = gpar(fill = "grey80"))

# Create data viewport: dvp
dvp <- dataViewport(xData = mtcars$wt, yData = mtcars$mpg)

# 4 - Push dvp
pushViewport(dvp)

# Add two axes
grid.xaxis()
grid.yaxis()
```

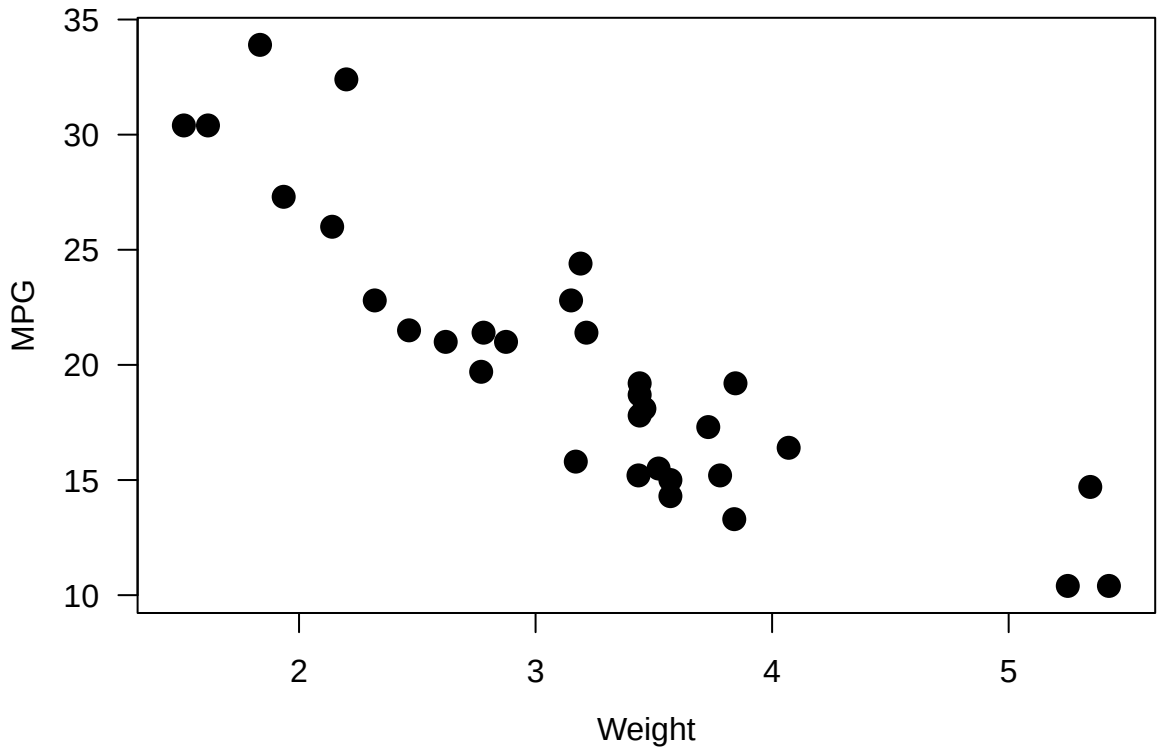


Great job! You're on your way to building a plot from scratch.

Build a Plot from Scratch (2)

The work you did before to build a plot from scratch is already included. Now you're ready to add the points and the appropriate labels.

```
# Work from before
pushViewport(plotViewport(c(5, 4, 2, 2)))
grid.rect(gp = gpar())
pushViewport(dataViewport(xData = mtcars$wt, yData = mtcars$mpg))
grid.xaxis()
grid.yaxis()

# 1 - Add text to x axis
grid.text("Weight", y = unit(-3, "lines"))

# 2 - Add text to y axis
grid.text("MPG", x = unit(-3, "lines"), rot = 90)

# 3 - Add points
grid.points(x = mtcars$wt, y = mtcars$mpg, pch = 16)
```

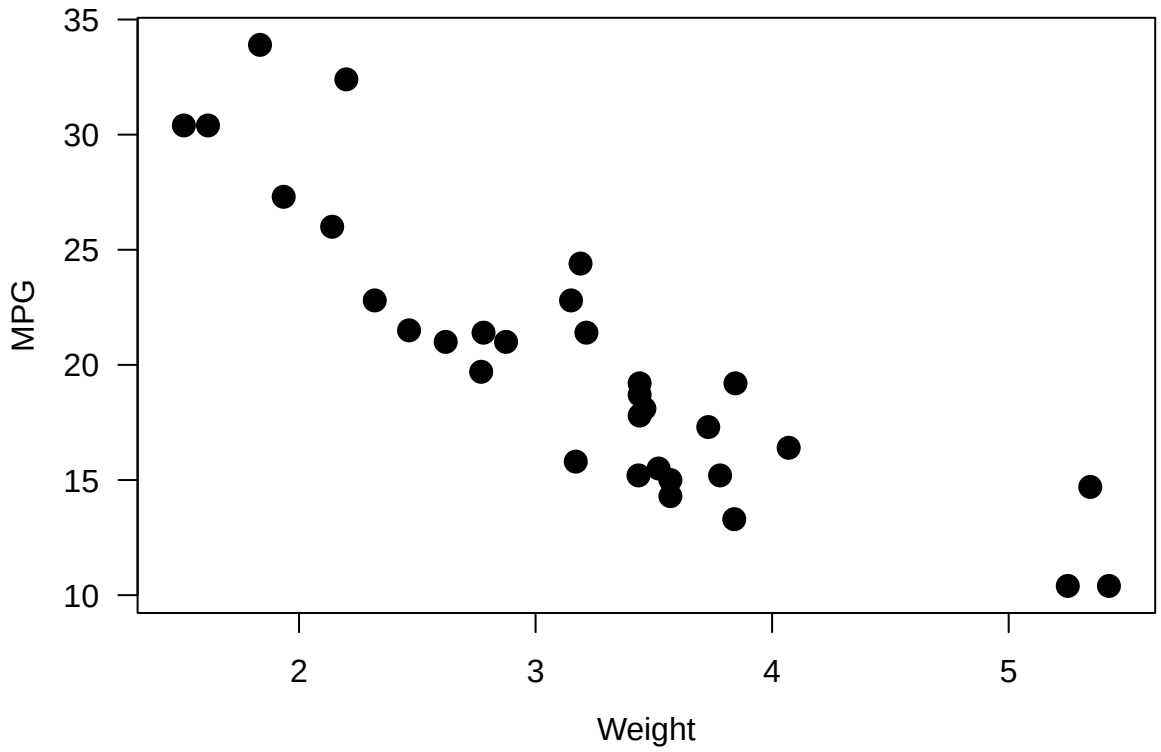


Excellent! The basics of `ggplot2` are present here.

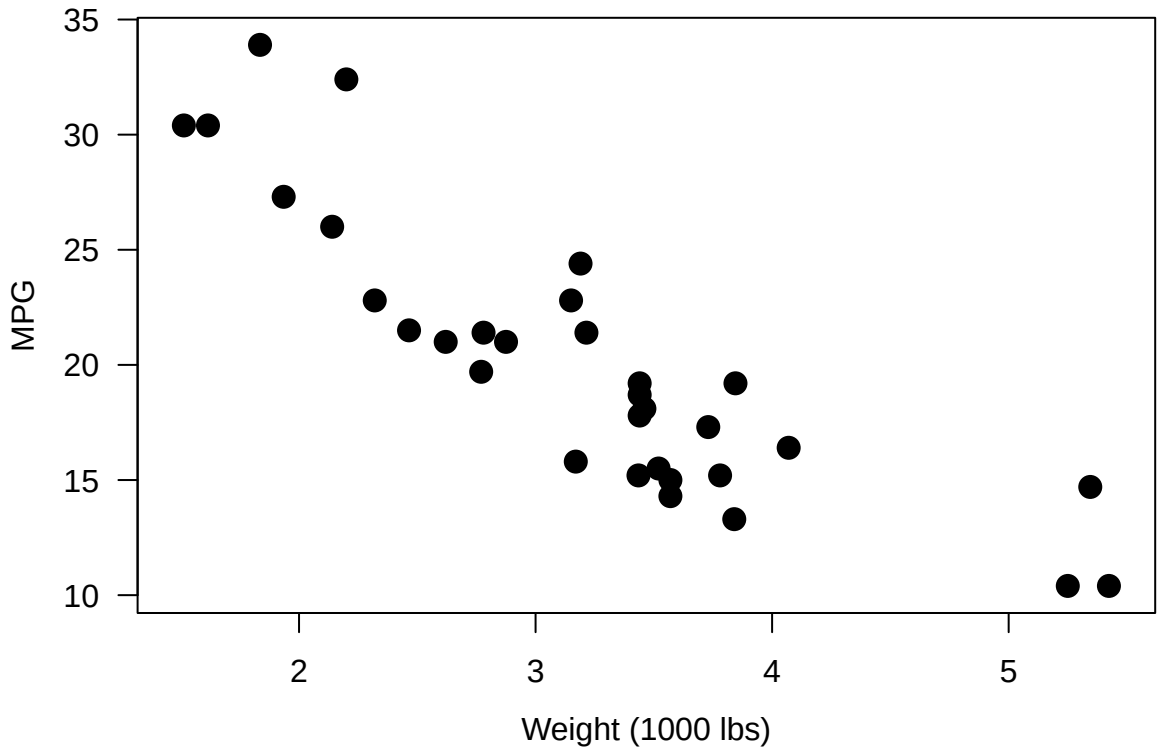
Modifying a Plot with grid.edit

The commands you've coded up to now to create the plot are available in the editor. The great thing about `grid`, in comparison to `base`, is that you can name the different plot elements, so that you can access them and change them later on. You can do this with the `grid.edit()` function. Give it a try!

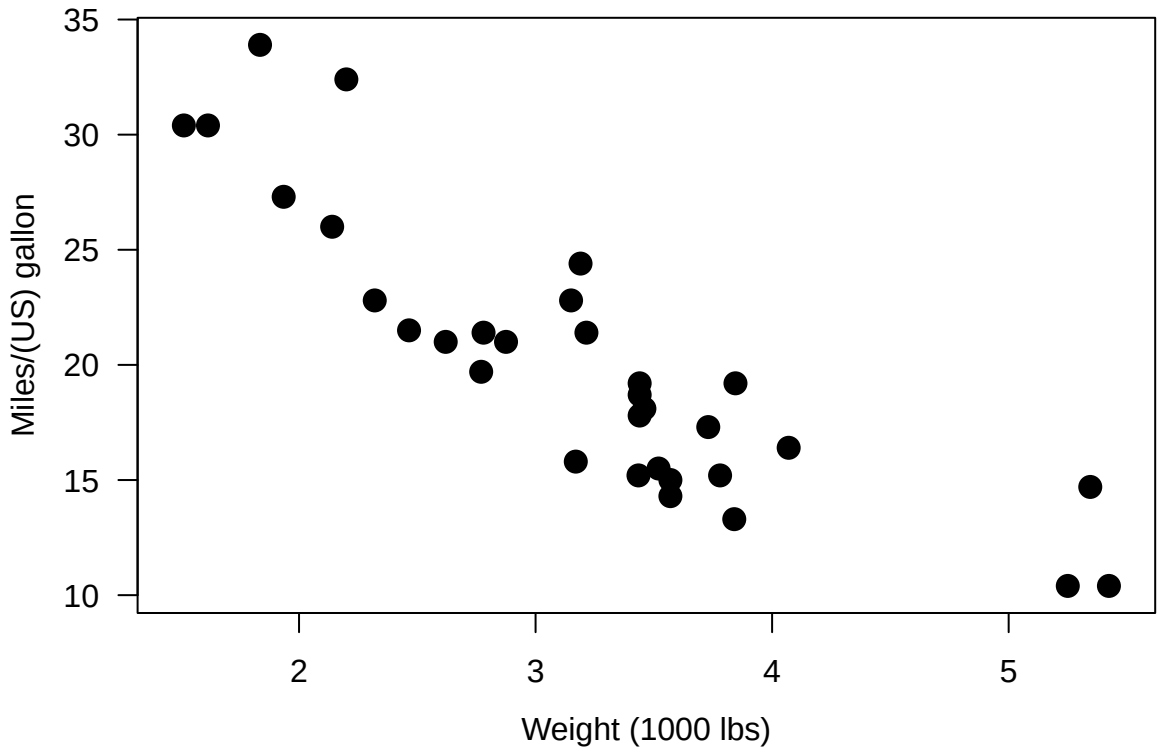
```
# Work from before
pushViewport(plotViewport(c(5, 4, 2, 2)))
grid.rect(gp = gpar())
pushViewport(dataViewport(xData = mtcars$wt, yData = mtcars$mpg))
grid.xaxis()
grid.yaxis()

# Work from before - add names
grid.text("Weight", y = unit(-3, "lines"), name = "xaxis")
grid.text("MPG", x = unit(-3, "lines"), rot = 90, name = "yaxis")
grid.points(x = mtcars$wt, y = mtcars$mpg, pch = 16, name = "datapoints")
```



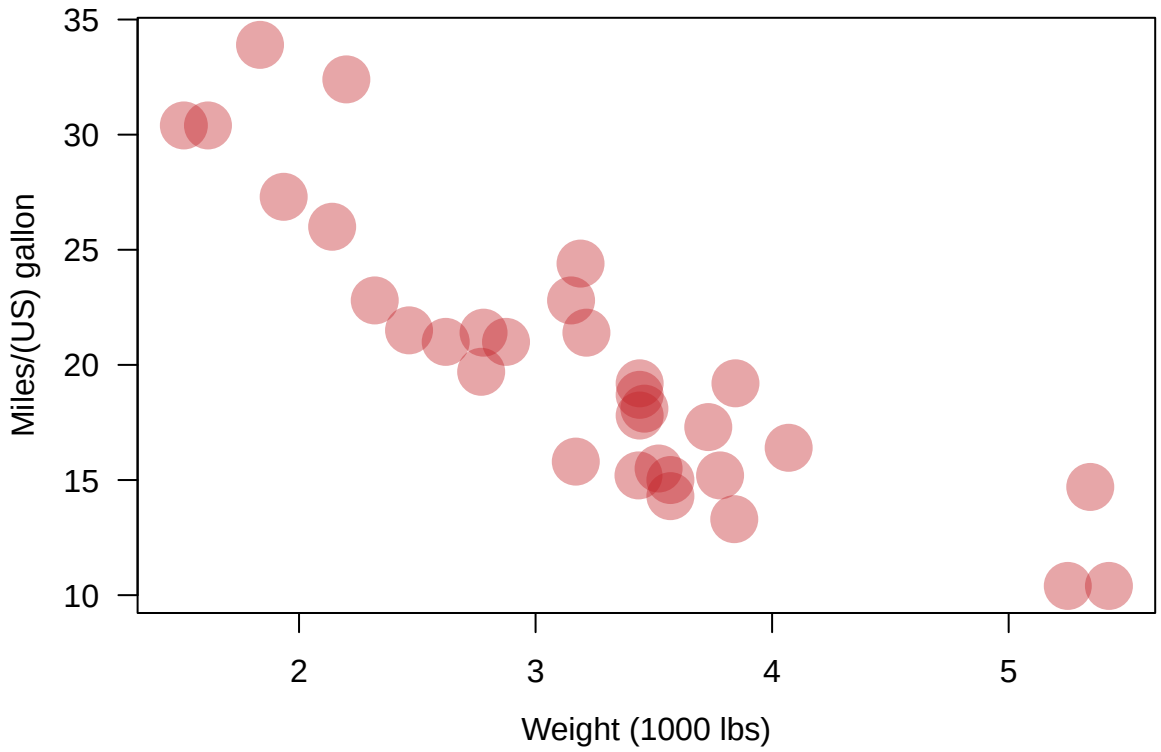

```
# Edit "xaxis"
grid.edit("xaxis", label = "Weight (1000 lbs)")
```




```
# Edit "yaxis"
grid.edit("yaxis", label = "Miles/(US) gallon")
```




```
# Edit "datapoints"
grid.edit("datapoints",
          gp = gpar(col = "#C3212766", cex = 2))
```




```
# cex = character expansion
```

Wonderful! This is a great looking-scatter plot.

Grid Graphics in ggplot2

We can produce graphical objects typically called **grobs**.

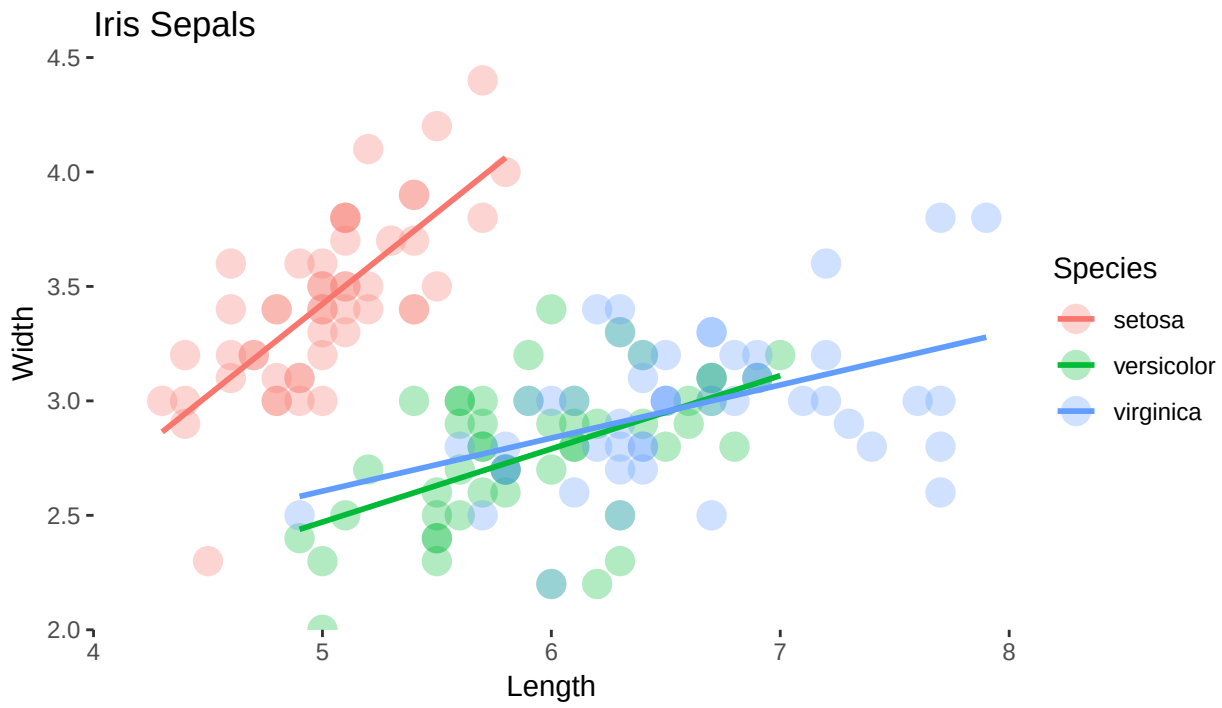
Graphic Output	Graphics Object
grid.rect()	rectGrob()
grid.lines()	linesGrob()
grid.circle()	circleGrob()
grid.polygon()	polygonGrob()
grid.text()	textGrob()

Table 1: Graphics output and their respective objects.

Underlying every `ggplot2` object is a collection of grobs. Let's take a look at the plot below.

```
p <- ggplot(iris, aes(x = Sepal.Length,
                      y = Sepal.Width,
                      col = Species)) +
  geom_point(alpha = 0.3, size = 5, shape = 16) +
  # changed geom_smooth to stat_smooth
  # same outcome
```

```
  stat_smooth(method = "lm", se = FALSE) +
  scale_y_continuous("Width", limits = c(2, 4.5), expand = c(0,0)) +
  scale_x_continuous("Length", limits = c(4, 8), expand = c(0,0)) +
  coord_equal() +
  ggtitle("Iris Sepals") +
  theme(rect = element_blank())
p
```




```
# generate grob table for plot p
g <- ggplotGrob(p)
g
#> TableGrob (12 x 11) "layout": 19 grobs
#>     z         cells       name
#> 1   0 ( 1-12, 1-11) background
#> 2   5 ( 6- 6, 4- 4)     spacer
#> 3   7 ( 7- 7, 4- 4)     axis-l
#> 4   3 ( 8- 8, 4- 4)     spacer
#> 5   6 ( 6- 6, 5- 5)     axis-t
#> 6   1 ( 7- 7, 5- 5)      panel
#> 7   9 ( 8- 8, 5- 5)     axis-b
#> 8   4 ( 6- 6, 6- 6)     spacer
#> 9   8 ( 7- 7, 6- 6)     axis-r
#> 10  2 ( 8- 8, 6- 6)     spacer
#> 11 10 ( 5- 5, 5- 5)     xlab-t
#> 12 11 ( 9- 9, 5- 5)     xlab-b
```

```
#> 13 12 ( 7- 7, 3- 3)     ylab-l
#> 14 13 ( 7- 7, 7- 7)     ylab-r
#> 15 14 ( 7- 7, 9- 9)  guide-box
#> 16 15 ( 4- 4, 5- 5)   subtitle
#> 17 16 ( 3- 3, 5- 5)      title
#> 18 17 (10-10, 5- 5)    caption
#> 19 18 ( 2- 2, 2- 2)        tag
#>                                                 grob
#> 1        zeroGrob[plot.background..zeroGrob.183]
#> 2                                  zeroGrob[NULL]
#> 3            absoluteGrob[GRID.absoluteGrob.141]
#> 4                                  zeroGrob[NULL]
#> 5                                  zeroGrob[NULL]
#> 6                       gTree[panel-1.gTree.127]
#> 7            absoluteGrob[GRID.absoluteGrob.134]
#> 8                                  zeroGrob[NULL]
#> 9                                  zeroGrob[NULL]
#> 10                                 zeroGrob[NULL]
#> 11                                 zeroGrob[NULL]
#> 12 titleGrob[axis.title.x.bottom..titleGrob.144]
#> 13   titleGrob[axis.title.y.left..titleGrob.147]
#> 14                                 zeroGrob[NULL]
#> 15                              gtable[guide-box]
#> 16         zeroGrob[plot.subtitle..zeroGrob.180]
#> 17          titleGrob[plot.title..titleGrob.179]
#> 18          zeroGrob[plot.caption..zeroGrob.182]
#> 19              zeroGrob[plot.tag..zeroGrob.181]
```

Exploring the gTable

In the previous chapter you saw *graphical outputs* using a variety of `grid` functions. *Graphical Objects*, aka *Grobs*, are the object form of these items and can be found in your `ggplot2` plots. Let's take a look at how these grobs are stored in `ggplot` objects.

```
# A simple plot p
p <- ggplot(mtcars, aes(x = wt, y = mpg, col = factor(cyl))) + geom_point()

# Create gtab with ggplotGrob()
gtab <- ggplotGrob(p)

# Print out gtab
gtab
#> TableGrob (12 x 11) "layout": 19 grobs
#>     z         cells       name
#> 1   0 ( 1-12, 1-11) background
#> 2   5 ( 6- 6, 4- 4)     spacer
#> 3   7 ( 7- 7, 4- 4)     axis-l
#> 4   3 ( 8- 8, 4- 4)     spacer
#> 5   6 ( 6- 6, 5- 5)     axis-t
#> 6   1 ( 7- 7, 5- 5)      panel
#> 7   9 ( 8- 8, 5- 5)     axis-b
#> 8   4 ( 6- 6, 6- 6)     spacer
#> 9   8 ( 7- 7, 6- 6)     axis-r
```

```
#> 10  2 ( 8- 8, 6- 6)     spacer
#> 11 10 ( 5- 5, 5- 5)     xlab-t
#> 12 11 ( 9- 9, 5- 5)     xlab-b
#> 13 12 ( 7- 7, 3- 3)     ylab-l
#> 14 13 ( 7- 7, 7- 7)     ylab-r
#> 15 14 ( 7- 7, 9- 9)  guide-box
#> 16 15 ( 4- 4, 5- 5)   subtitle
#> 17 16 ( 3- 3, 5- 5)      title
#> 18 17 (10-10, 5- 5)    caption
#> 19 18 ( 2- 2, 2- 2)        tag
#>                                           grob
#> 1               rect[plot.background..rect.253]
#> 2                                 zeroGrob[NULL]
#> 3           absoluteGrob[GRID.absoluteGrob.214]
#> 4                                 zeroGrob[NULL]
#> 5                                 zeroGrob[NULL]
#> 6                      gTree[panel-1.gTree.200]
#> 7           absoluteGrob[GRID.absoluteGrob.207]
#> 8                                 zeroGrob[NULL]
#> 9                                 zeroGrob[NULL]
#> 10                                zeroGrob[NULL]
#> 11                                zeroGrob[NULL]
#> 12 titleGrob[axis.title.x.bottom..titleGrob.217]
#> 13   titleGrob[axis.title.y.left..titleGrob.220]
#> 14                                zeroGrob[NULL]
#> 15                             gtable[guide-box]
#> 16         zeroGrob[plot.subtitle..zeroGrob.249]
#> 17            zeroGrob[plot.title..zeroGrob.248]
#> 18          zeroGrob[plot.caption..zeroGrob.251]
#> 19              zeroGrob[plot.tag..zeroGrob.250]

# Extract the grobs from gtab
g <- gtab$grobs

# Draw only the legend
legend_index <- which(vapply(g, inherits, what = "gtable", logical(1)))
grid.draw(g[[legend_index]])
```

Nice work! We can grab any grob of interest from the gTable.

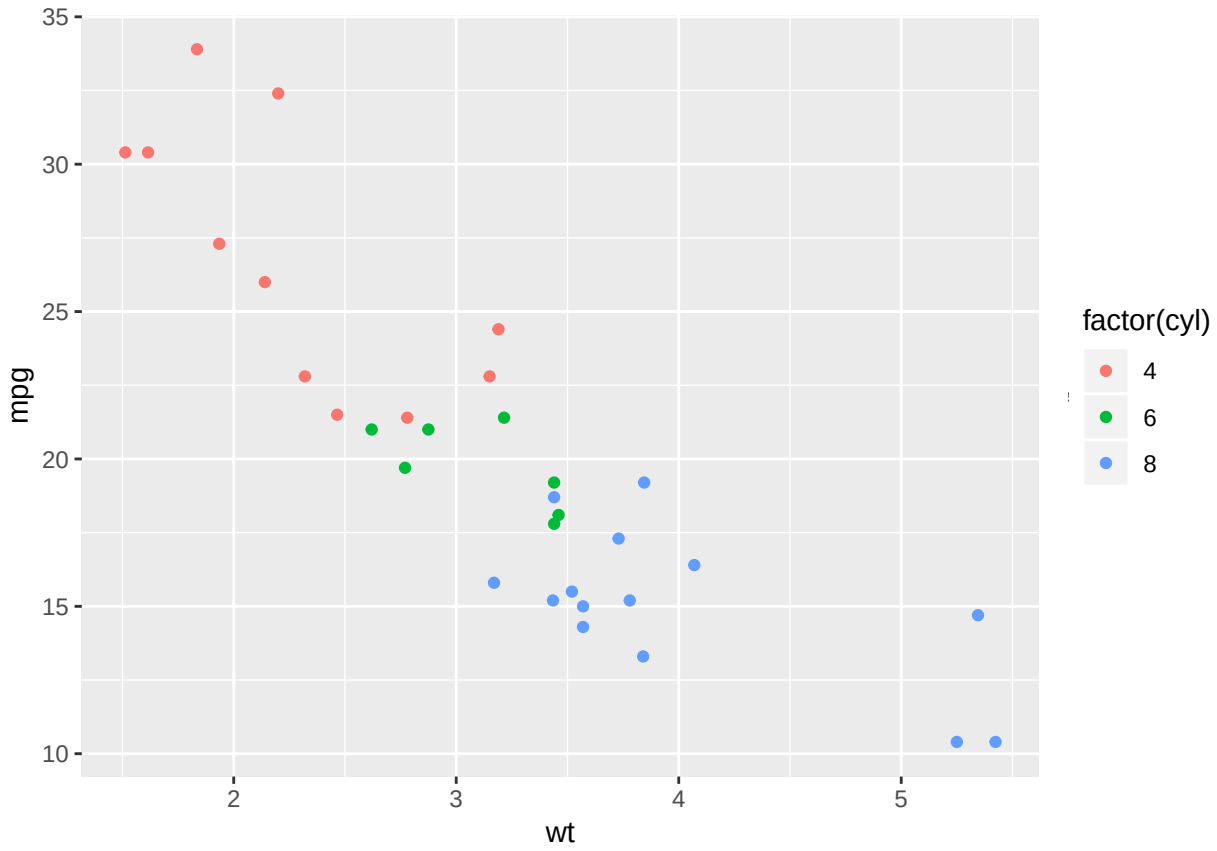
Modifying the gTable

You can visualize the layout of a `gTable` object with `gtable_show_layout()`. In the layout plot, each segment is labelled with its position.

The legend, that you can access with `g[[legend_index]]`, is a `gTable` itself, so you can also show its layout. It's perfectly possible to update this layout by adding new graphical objects, similar to what you saw in the video.

```
library(gtable)

# Code from before
p <- ggplot(mtcars, aes(x = wt, y = mpg, col = factor(cyl))) + geom_point()
gtab <- ggplotGrob(p)
g <- gtab$grobs
legend_index <- which(vapply(g, inherits, what = "gtable", logical(1)))
grid.draw(g[[legend_index]])
```


```r
# 1 - Show layout of legend grob
gtable_show_layout(g[[legend_index]])

# Create text grob
my_text <- textGrob(label = "Motor Trend, 1974", gp = gpar(fontsize = 7, col = "gray25"))

# 2 - Use gtable_add_grob to modify original gtab
new_legend <- gtable_add_grob(gtab$grobs[[legend_index]], my_text, 3, 2)

# 3 - Update in gtab
gtab$grobs[[legend_index]] <- new_legend

# 4 - Draw gtab
grid.draw(gtab)
```



Wonderful! Now that we can access every grob, we can also change them to meet any need we have.

ggplot2 Objects

Let's try to understand the internals of the ggplot2 object.

```
names(p)
#> [1] "data"        "layers"      "scales"      "mapping"     "theme"
#> [6] "coordinates" "facet"       "plot_env"    "labels"
p$data
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

```
p$layers
#> [[1]]
#> geom_point: na.rm = FALSE
#> stat_identity: na.rm = FALSE
#> position_identity
p$scales
#> <ggproto object: Class ScalesList, gg>
#>     add: function
#>     clone: function
#>     find: function
#>     get_scales: function
#>     has_scale: function
#>     input: function
#>     n: function
#>     non_position_scales: function
#>     scales: list
#>     super:  <ggproto object: Class ScalesList, gg>
p$theme
#> list()
```

```
p$coordinates
#> <ggproto object: Class CoordCartesian, Coord, gg>
#>     aspect: function
#>     backtransform_range: function
#>     clip: on
#>     default: TRUE
#>     distance: function
#>     expand: TRUE
#>     is_free: function
#>     is_linear: function
#>     labels: function
#>     limits: list
#>     modify_scales: function
#>     range: function
#>     render_axis_h: function
#>     render_axis_v: function
#>     render_bg: function
#>     render_fg: function
#>     setup_data: function
#>     setup_layout: function
#>     setup_panel_params: function
#>     setup_params: function
#>     transform: function
#>     super:  <ggproto object: Class CoordCartesian, Coord, gg>
p$facet
#> <ggproto object: Class FacetNull, Facet, gg>
#>     compute_layout: function
#>     draw_back: function
#>     draw_front: function
#>     draw_labels: function
#>     draw_panels: function
#>     finish_data: function
#>     init_scales: function
#>     map_data: function
#>     params: list
#>     setup_data: function
#>     setup_params: function
#>     shrink: TRUE
#>     train_scales: function
#>     vars: function
#>     super:  <ggproto object: Class FacetNull, Facet, gg>
p$plot_env
#> <environment: R_GlobalEnv>
p$labels
#> $x
#> [1] "wt"
#>
#> $y
#> [1] "mpg"
#>
#> $colour
#> [1] "factor(cyl)"
```

When you create a plot to be displayed on a screen or a file, the plot is rendered by the `ggplot_build()`

function. If we take a look at `ggplot_build(p)`, we notice three elements are shown: data, panel, and plot. The first two are the main output. The data contains the list of dataframes, one for each layer and the panel object contains all information about the axes such as limits and breaks. The last element is the plot.

`ggplot_build` is called when we print information to the screen so it contains information that have been calculated from specific statistics function such as histogram binning, box plots, and density plots, which are not found in the original data but are used in the plot. If we want to manipulate a `ggplot_build` object, we can use a `gtable`

Exploring ggplot objects

`ggplot` objects are basically just a named list that contains the information to make the actual plot. Here you'll explore the structure of this object.

```
# Simple plot p
p <- ggplot(mtcars, aes(x = wt, y = mpg, col = factor(cyl))) + geom_point()

# Examine class() and names()
class(p)
#> [1] "gg"     "ggplot"
names(p)
#> [1] "data"        "layers"      "scales"      "mapping"     "theme"
#> [6] "coordinates" "facet"       "plot_env"    "labels"

# Print the scales sub-list
p$scales$scales
#> list()

# Update p
p <- p +
  scale_x_continuous("Length", limits = c(4, 8), expand = c(0, 0)) +
  scale_y_continuous("Width", limits = c(2, 4.5), expand = c(0, 0))

# Print the scales sub-list
p$scales$scales
#> [[1]]
#> <ScaleContinuousPosition>
#>  Range:
#>  Limits:    4 --    8
#>
#> [[2]]
#> <ScaleContinuousPosition>
#>  Range:
#>  Limits:    2 --  4.5
```

Nice one! This is pretty detailed, but the more we can access out objects the more custom our visualisations become.

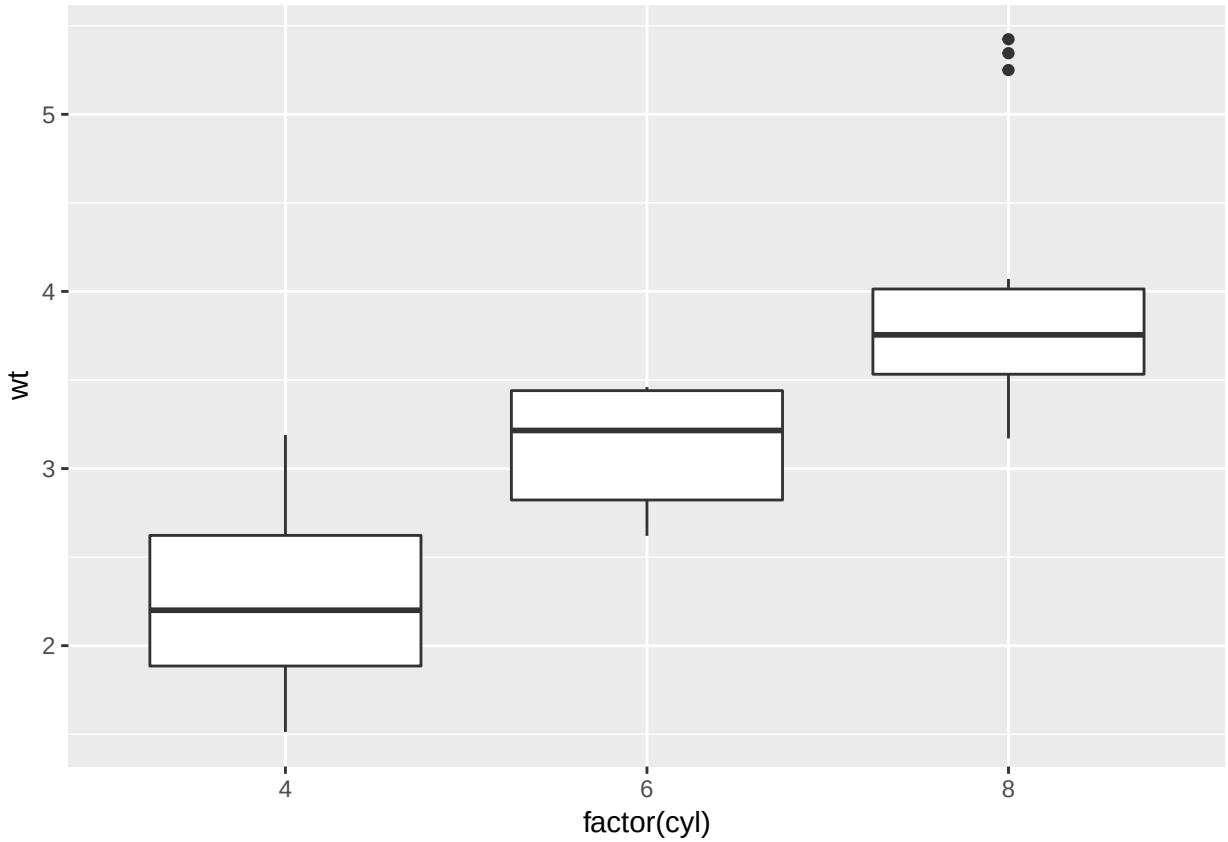
ggplot_build and ggplot_gtable

In the viewer we have produced a box plot of the mtcars dataset (called p) that you'll use to explore two key ggplot functions for accessing the object internals: `ggplot_build()` and `ggplot_gtable()`.

`ggplot_build()` is executed when you want to display or save an actual ggplot plot. It takes the data input and produces the visual output.

```
# Box plot of mtcars: p
p <- ggplot(mtcars, aes(x = factor(cyl), y = wt)) + geom_boxplot()

# Create pbuild
pbuild <- ggplot_build(p)

# a list of 3 elements
names(pbuild)
#> [1] "data"   "layout" "plot"

# Print out each element in pbuild
pbuild$data
#> [[1]]
#>    ymin  lower middle   upper ymax            outliers notchupper
#> 1 1.513 1.8850  2.200 2.62250 3.19                       2.551336
#> 2 2.620 2.8225  3.215 3.44000 3.46                       3.583761
#> 3 3.170 3.5325  3.755 4.01375 4.07 5.250, 5.424, 5.345   3.958219
#>   notchlower x PANEL group ymin_final ymax_final  xmin  xmax xid newx
#> 1   1.848664 1     1     1      1.513      3.190 0.625 1.375   1    1
#> 2   2.846239 2     1     2      2.620      3.460 1.625 2.375   2    2
#> 3   3.551781 3     1     3      3.170      5.424 2.625 3.375   3    3
#>   new_width weight colour  fill size alpha shape linetype
#> 1      0.75      1 grey20 white  0.5    NA    19    solid
#> 2      0.75      1 grey20 white  0.5    NA    19    solid
#> 3      0.75      1 grey20 white  0.5    NA    19    solid
pbuild$layout
#> <ggproto object: Class Layout, gg>
#>     coord: <ggproto object: Class CoordCartesian, Coord, gg>
#>         aspect: function
#>         backtransform_range: function
#>         clip: on
#>         default: TRUE
#>         distance: function
#>         expand: TRUE
#>         is_free: function
#>         is_linear: function
#>         labels: function
#>         limits: list
#>         modify_scales: function
#>         range: function
#>         render_axis_h: function
#>         render_axis_v: function
#>         render_bg: function
#>         render_fg: function
#>         setup_data: function
#>         setup_layout: function
#>         setup_panel_params: function
#>         setup_params: function
#>         transform: function
#>         super:  <ggproto object: Class CoordCartesian, Coord, gg>
#>     coord_params: list
```

```
#>     facet: <ggproto object: Class FacetNull, Facet, gg>
#>         compute_layout: function
#>         draw_back: function
#>         draw_front: function
#>         draw_labels: function
#>         draw_panels: function
#>         finish_data: function
#>         init_scales: function
#>         map_data: function
#>         params: list
#>         setup_data: function
#>         setup_params: function
#>         shrink: TRUE
#>         train_scales: function
#>         vars: function
#>         super:  <ggproto object: Class FacetNull, Facet, gg>
#>     facet_params: list
#>     finish_data: function
#>     get_scales: function
#>     layout: data.frame
#>     map_position: function
#>     panel_params: list
#>     panel_scales_x: list
#>     panel_scales_y: list
#>     render: function
#>     render_labels: function
#>     reset_scales: function
#>     setup: function
#>     setup_panel_params: function
#>     train_position: function
#>     xlabel: function
#>     ylabel: function
#>     super:  <ggproto object: Class Layout, gg>
pbuild$plot

# Create gtab from pbuild
gtab <- ggplot_gtable(pbuild)

# Draw gtab
grid.draw(gtab)
```



Good job! Now you can see what's happening under-the-hood when you print out a plot.

Extracting Details

In the video you saw how to change the clipping parameters of a gTable object. Here, you'll see something more practical: how to extract calculated values.

Many geoms are associated with underlying descriptive statistics which are calculated and then plotted. In these cases you actually don't have the actual values that were plotted. Of course, these values are stored under the hood and you can access them in the results from ggplot_build(). This can be particularly useful for box plots. For example, since there are many methods for calculating Q1 and Q3, if you calculate your IQR and outliers outside of ggplot2 you may end up using a different method and get different results. Sometimes you want to have exactly the values that were plotted.

```
# Box plot of mtcars: p
p <- ggplot(mtcars, aes(x = factor(cyl), y = wt)) + geom_boxplot()

# Build pdata
pdata <- ggplot_build(p)$data

# confirm that the first element of the list is a data frame
class(pdata[[1]])
#> [1] "data.frame"

# Isolate this data frame
my_df <- pdata[[1]]
```

```r
# The x labels
my_df$group <- c("4", "6", "8")

# Print out specific variables
my_df[c(1:6, 11)]
```

ymin	lower	middle	upper	ymax	outliers	group
1.513	1.8850	2.200	2.62250	3.19	numeric(0)	4
2.620	2.8225	3.215	3.44000	3.46	numeric(0)	6
3.170	3.5325	3.755	4.01375	4.07	c(5.25, 5.424, 5.345)	8

Good job! Sometimes you want to get specific information that was used to creat a plot.

gridExtra

```r
library(plyr)
library(gridExtra)

my_plots <- dlply(mtcars, .(cyl), function(df) {
  ggplot(df, aes(mpg, wt)) +
    geom_point() +
    xlim(range(mtcars$mpg)) +
    ylim(range(mtcars$wt)) +
    ggtitle(paste(df$cyl[1], "cylinders"))})

length(my_plots)
#> [1] 3

names(my_plots)
#> [1] "4" "6" "8"
```

Arranging plots (1)

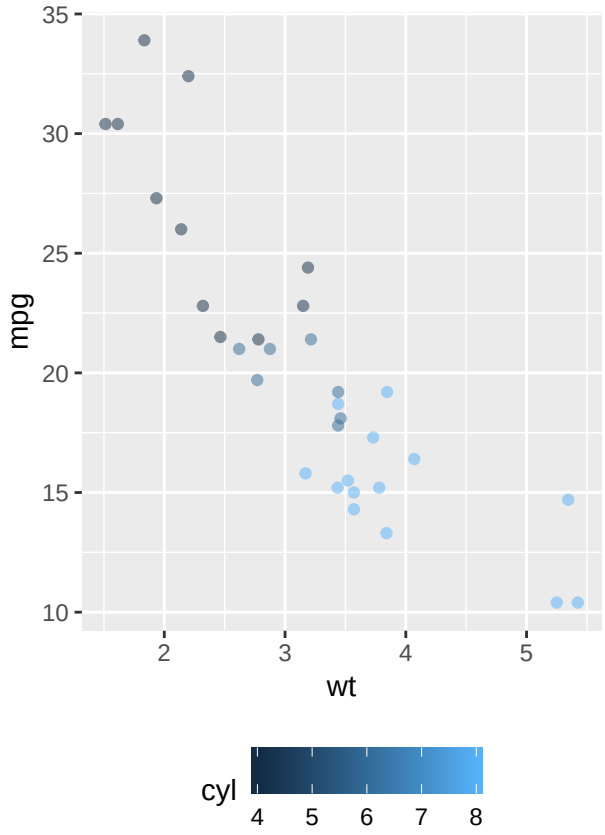
The functions in gridExtra allow you to arrange any number of plots in a variety of ways. Since you can access the legend as a separate object, that means you can also arrange multiple plots with a single legend, as shown in the viewer. This is a good alternative to faceting, since with facets it's not possible to set a different geom for each sub-plot. Here, you can combine any variety of plots and use a consistent color scale with only one legend to unify the whole image.

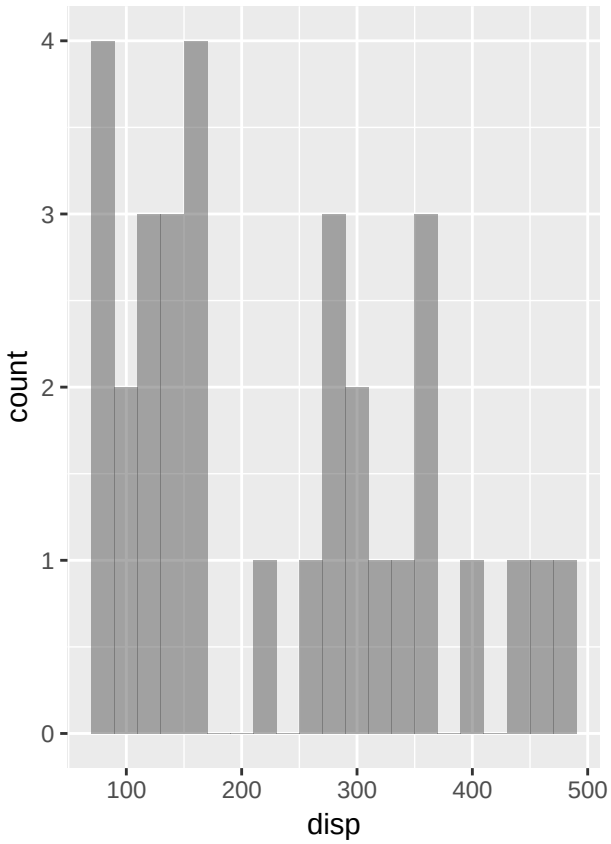
To do this you'll create a new arrange graphical object, using grid.arrange(), which will combine several pre-existing grobs. Just like with grid.rect() and rectGrob() there are two versions of the arrange grob, one grid.arrange() produces a graphics output, which means you just draw the item to the viewer, and arrangeGrob() which returns a graphical object, aka grob which can be further manipulated.

In this exercise, you'll just create your objects and arrange them using grid.arrange(). In the first steps you created two basic plots, g1 and g2. In the next exercise you'll see what to do about the legend.

```r
# Add a theme (legend at the bottom)
g1 <- ggplot(mtcars, aes(wt, mpg, col = cyl)) +
  geom_point(alpha = 0.5) +
  theme(legend.position = "bottom")

# Add a theme (no legend)
```

```
g2 <- ggplot(mtcars, aes(disp, fill = cyl)) +
  geom_histogram(position = "identity", alpha = 0.5, binwidth = 20) +
  theme(legend.position = "none")

# Load gridExtra
library(gridExtra)

# Call grid.arrange()
grid.arrange(g1, g2, ncol = 2)
```





Cool! Eventually you want to isolate the legend and have it appear as the image in the viewer, so you can let ggplot arrange it in a horizontal layout for us.

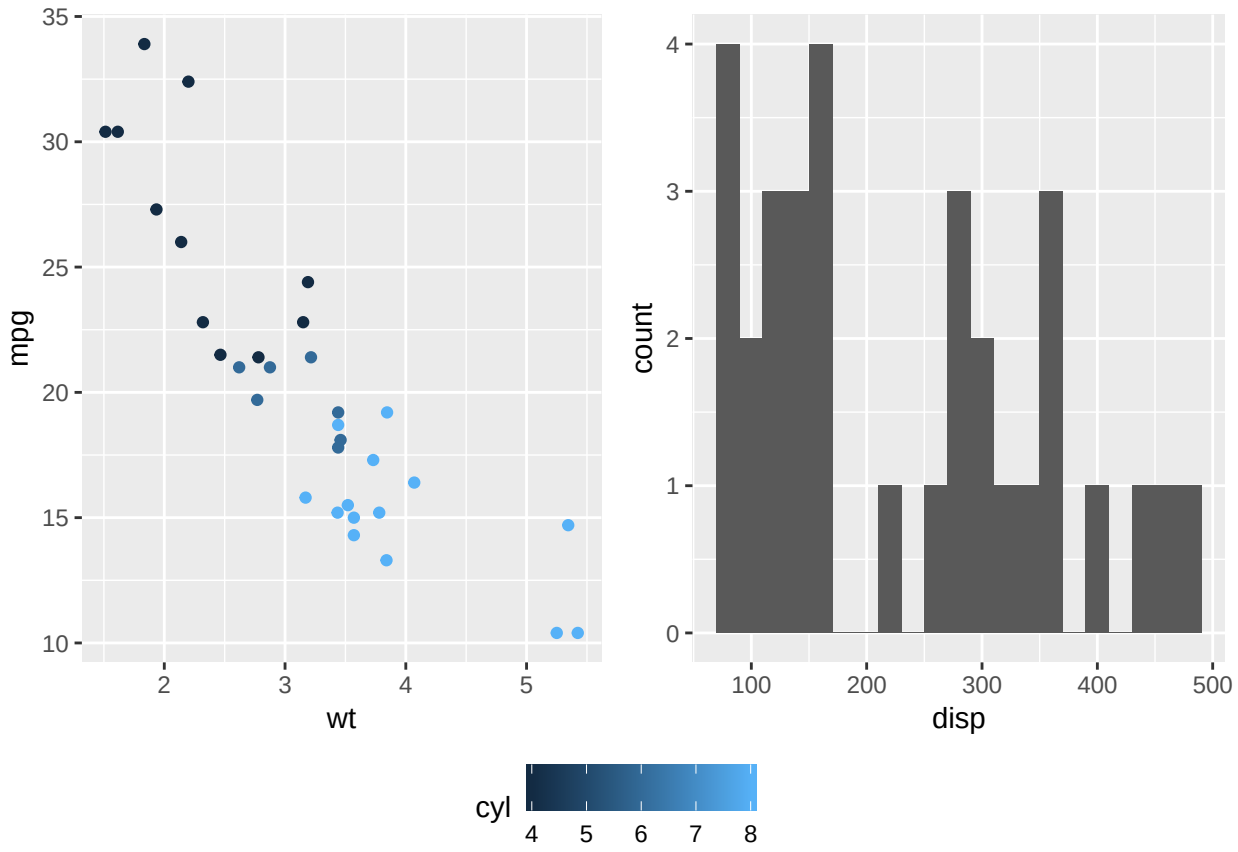
Arranging plots (2)

In the previous exercise you did a bare-bones arrangement of plots, but it would be nicer if the plot looks like the one that's shown in the viewer. You can imagine that you have three panels, not two. There are two asymmetrical rows, the small second row is where the legend is, and two symmetrical columns, where the plots are.

To obtain this plot you need to extract the legend. You already saw this in previous exercises and it has already been done for you; the legend is available as my_legend. Next you need to arrange all the items appropriately.

```
# ggplot2, grid and gridExtra have been loaded for you
# Definitions of g1 and g2
```

```r
g1 <- ggplot(mtcars, aes(wt, mpg, col = cyl)) +
  geom_point() +
  theme(legend.position = "bottom")

g2 <- ggplot(mtcars, aes(disp, fill = cyl)) +
  geom_histogram(binwidth = 20) +
  theme(legend.position = "none")

# Extract the legend from g1
my_legend <- ggplotGrob(g1)$grobs[[legend_index]]

# Create g1_noleg
g1_noleg <- g1 +
    theme(legend.position = "none")

# Calculate the height: legend_height
legend_height <- sum(my_legend$heights)

# Arrange g1_noleg, g2 and my_legend
grid.arrange(g1_noleg, g2, my_legend,
             layout_matrix = matrix(c(1, 3, 2, 3), ncol = 2),
             heights = unit.c(unit(1, "npc") - legend_height, legend_height))
```



Great work! If ggplot2 doesn't provide enough customization possibilities, there's always gridExtra