2023-07-07_GrammarOfGraphics_001-intro

July 11, 2023

1 The grammar of graphics

In this notebook we approach the concept of a more rigid grammar for efficiently declaring graphics through a high-level syntax. We will gradually learn how this works based on examples and compare this with the less structured approach of creating the figures "by hand" with matplotlib.

1.1 Some context

1.1.1 Algorithmic / script-based creation of figures

- easily re-do figure when data has been updated
- adjust / fine-tune plots, change design, add features
- script is like a declaration of the figure, which is a better / more transparent way to store/archive it, than some graphics format itself
- doing graphics in python with matplotlib: in principle, we can do whatever we want! python is powerful, matplotlib has lots of functions to draw basic plots, geometric primitives, and some more sophisticated plots / high-level functions (histograms, violinplot, barchart, contour plot...)
- but we saw: complex stuff generates longer code, we have to add lots of "translations", apply data preprocessing first (e.g. get histogram or density estimate), convert data values to "plotable values" (e.g. categorical types to color or markers)

1.1.2 A good high-level interface?

- of course: in principle, these translations could be included directly into the plotting functions, which could provide a more high-level interface
- problem / challenge: if one does this too hastily / ad-hoc, the resulting library will become too rigid, or cluttered, unsystemtic
- so look for a good rigid grammar so that high-level functions know what to do, but flexible enough to be able to "do anything"

1.1.3 The grammar of graphics

- Leland Wilkinson did this in the book "The Grammar of Graphics"

- He has developed several graphics libraries from 1980s into early 2000s; so has thought about how to efficiently declare graphics for a long time. The book is essentially the concentration of this.
- It proposes a principled way to declare graphics; separating conceptual declaration from "practical translations", so that the latter can largely be automated (i.e. become clear from context, so that library knows how to do them)
- leads to very compact and transparent code; price: need to learn the language; sometimes need to make sure that data is available in right format
- but the book is pretty technical, not an easy read

1.1.4 ggplot2 and plotnine

- ggplot2 is plotting library for R that is based on this idea
- has very good documentation / tutorials available, e.g.
- https://ggplot2-book.org/
- https://r4ds.had.co.nz/index.html in particular section 3 on data visualization: https://r4ds.had.co.nz/data-visualisation.html
- plotnine is a port of ggplot2 to python https://plotnine.readthedocs.io
- in this lecture we will not do a theoretical intro to the topic
- merely try to get the basic point accross by some instructive examples
- we can discuss the usefulness of this scheme at the end

[1]:
```
# basic libraries
import numpy as np
import scipy
import pandas as pd

# matplotlib
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm
matplotlib.rc('image',cmap='viridis')
prop_cycle = plt.rcParams['axes.prop_cycle']
colors = prop_cycle.by_key()['color']

# plotnine / ggplot
from plotnine import ggplot, geom_point, geom_smooth, aes, facet_wrap,␣
 ↪after_stat
import plotnine as gg

# plotly
import plotly.express as px
import plotly.graph_objects as go
```

```
# example dataset
from plotnine.data import mpg
```

1.2 Layers

As we will see, layers are the first (and arguably major) component of the grammar. Each layer (in essence) specifies one small plot, and multiple layers can be superimposed in a single plot.

1.2.1 Mapping data variables to features

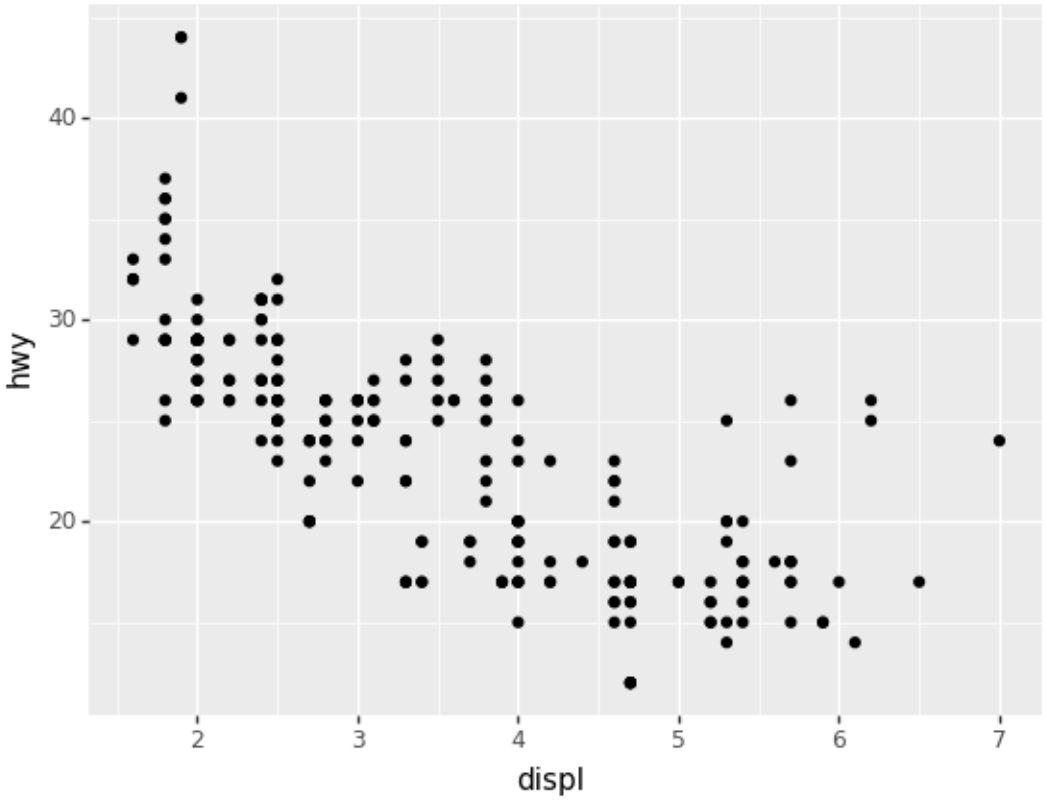
First steps

[2]:
```
# have a look at the dataset
mpg
```

[2]:
```
    manufacturer   model  displ  year  cyl       trans drv  cty  hwy fl  \
0           audi      a4    1.8  1999    4    auto(l5)   f   18   29  p
1           audi      a4    1.8  1999    4  manual(m5)   f   21   29  p
2           audi      a4    2.0  2008    4  manual(m6)   f   20   31  p
3           audi      a4    2.0  2008    4    auto(av)   f   21   30  p
4           audi      a4    2.8  1999    6    auto(l5)   f   16   26  p
..           …     …    …   …  …         …  ..  …  … ..
229   volkswagen  passat    2.0  2008    4    auto(s6)   f   19   28  p
230   volkswagen  passat    2.0  2008    4  manual(m6)   f   21   29  p
231   volkswagen  passat    2.8  1999    6    auto(l5)   f   16   26  p
232   volkswagen  passat    2.8  1999    6  manual(m5)   f   18   26  p
233   volkswagen  passat    3.6  2008    6    auto(s6)   f   17   26  p

       class
0    compact
1    compact
2    compact
3    compact
4    compact
..       …
229  midsize
230  midsize
231  midsize
232  midsize
233  midsize

[234 rows x 11 columns]
```

[3]:
```
# first encounter: mapping two colums of the dataset to x and y coordinates
ggplot(data = mpg) + \
```

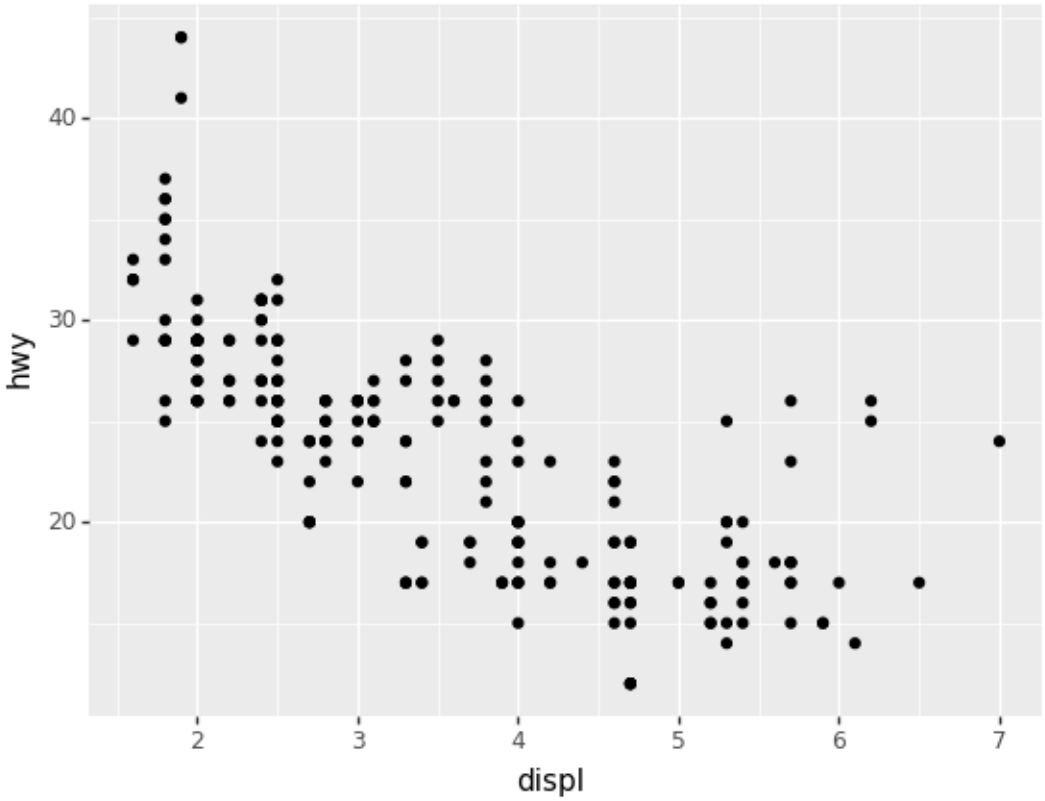
```
  geom_point(mapping = aes(x = "displ", y = "hwy"))
```



[3]: <ggplot: (8741380838500)>

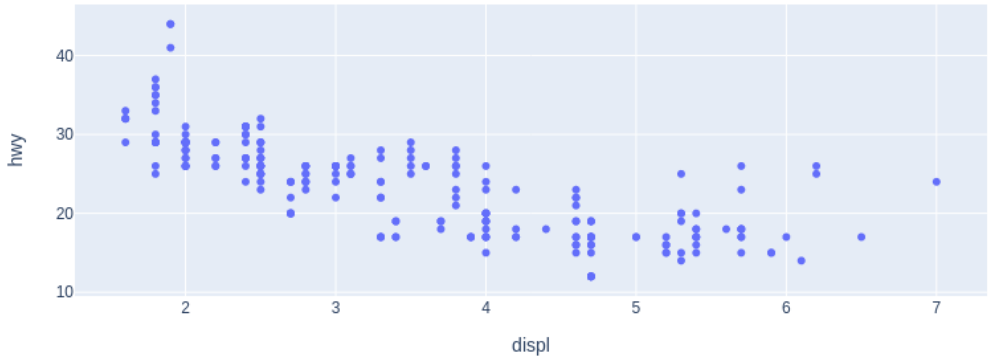
[4]:
```
# a quick glance at the syntax
a=ggplot(data = mpg)
b=geom_point(mapping = aes(x = "displ", y = "hwy"))
print("type of a: ",type(a))
print("type of b: ",type(b))
a.__add__(b)
```

```
type of a:  <class 'plotnine.ggplot.ggplot'>
type of b:  <class 'plotnine.geoms.geom_point.geom_point'>
```

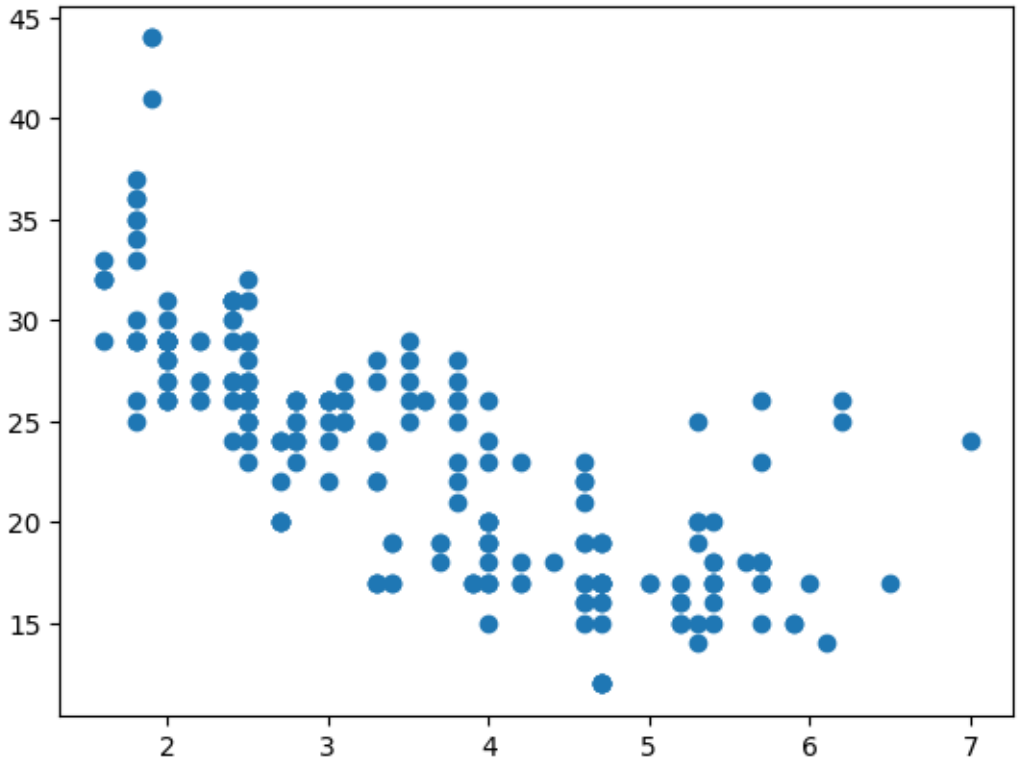


[4]: <ggplot: (8741477512150)>

[5]:
```
# plotly
fig = px.scatter(mpg, x="displ", y="hwy")
fig.show()
```



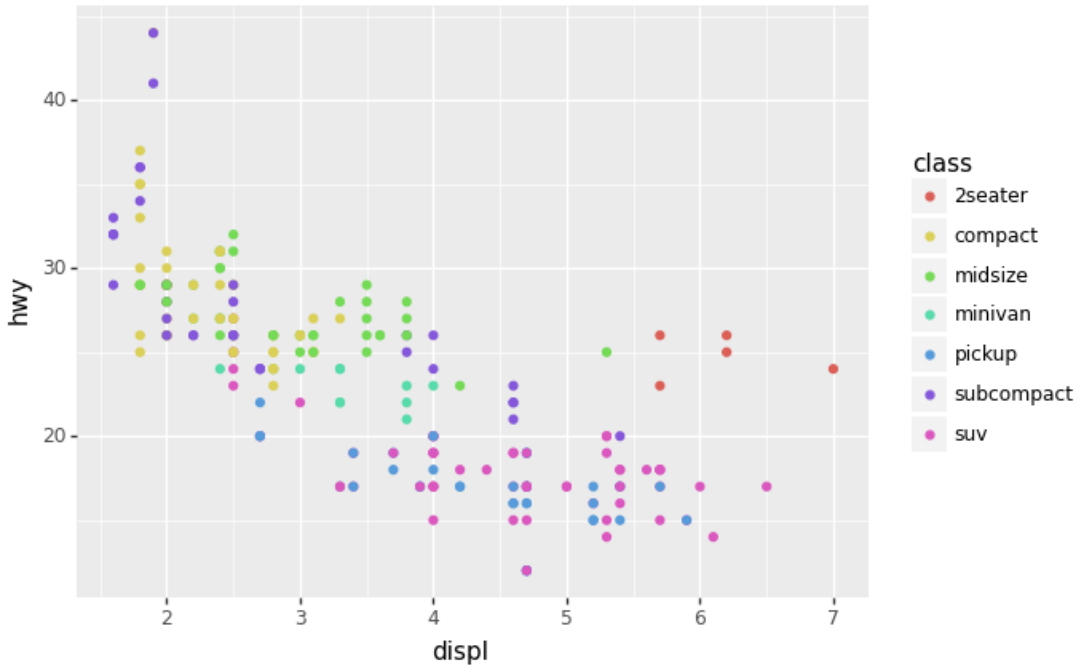
[6]:
```
# creating the (conceptually) same plot in matplotlib is comparably simple
# but: axes labels are missing, because matplotlib merely gets the numbers
# not the "contextual metadata"
plt.scatter(mpg["displ"],mpg["hwy"])
plt.show()
```



Adding more variables and features

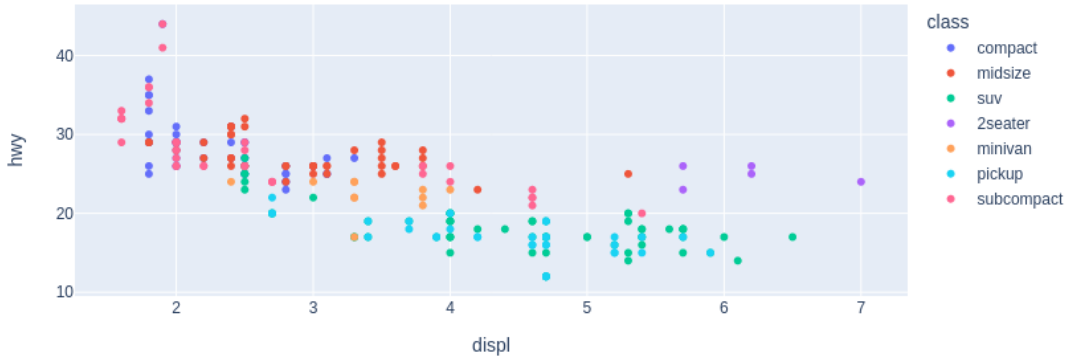
[7]:
```
# now we want to add the class column to the plot and map it to color
# conceptually this is completely clear: class is a categorical variable
# one needs to pick a corresponding discrete colormap and add a legend

# we find: with plotnine/ggplot and plotly this is straight-forward
# matplotlib is struggling, because for the color attribute it expects either␣
 ↪explicit colors
# or real numbers that can "trivially" be mapped to a standard (continuous)␣
 ↪color-scale
# of course, we can also "manually" attain our end with matplotlib, but it␣
 ↪requires more work
```

[11]:
```
ggplot(data = mpg) + \
  geom_point(mapping = aes(x = "displ", y = "hwy", color="class"))
```



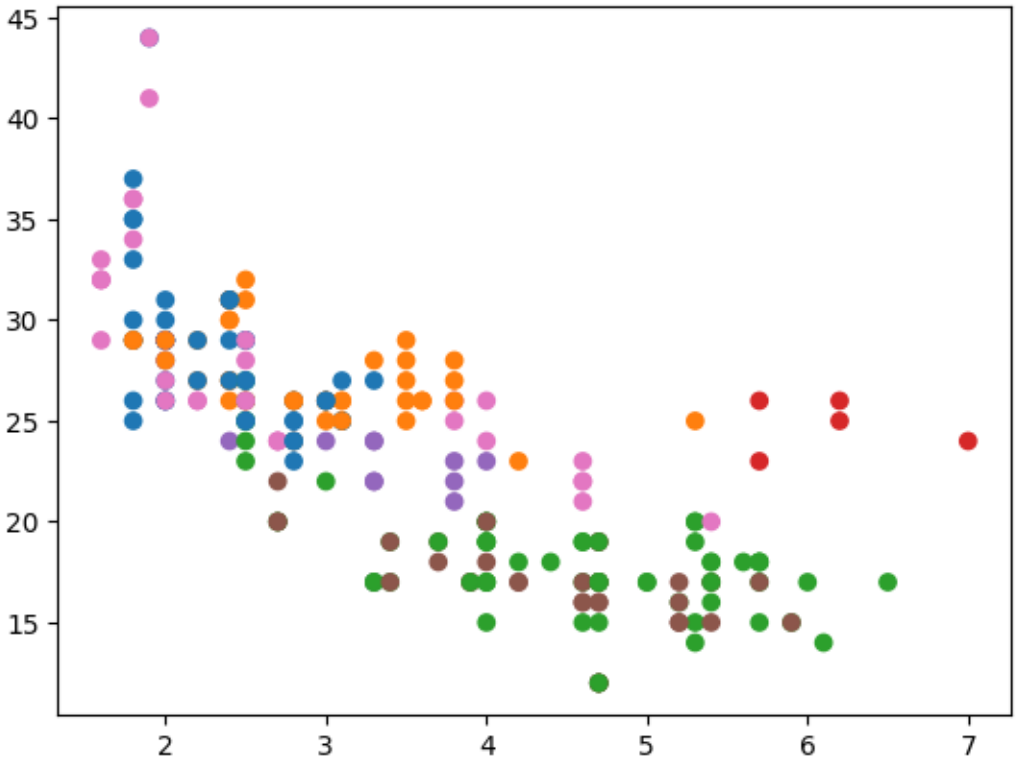
[11]: <ggplot: (8765868208285)>

[9]:
```
fig = px.scatter(mpg, x="displ", y="hwy", color="class")
fig.show()
```



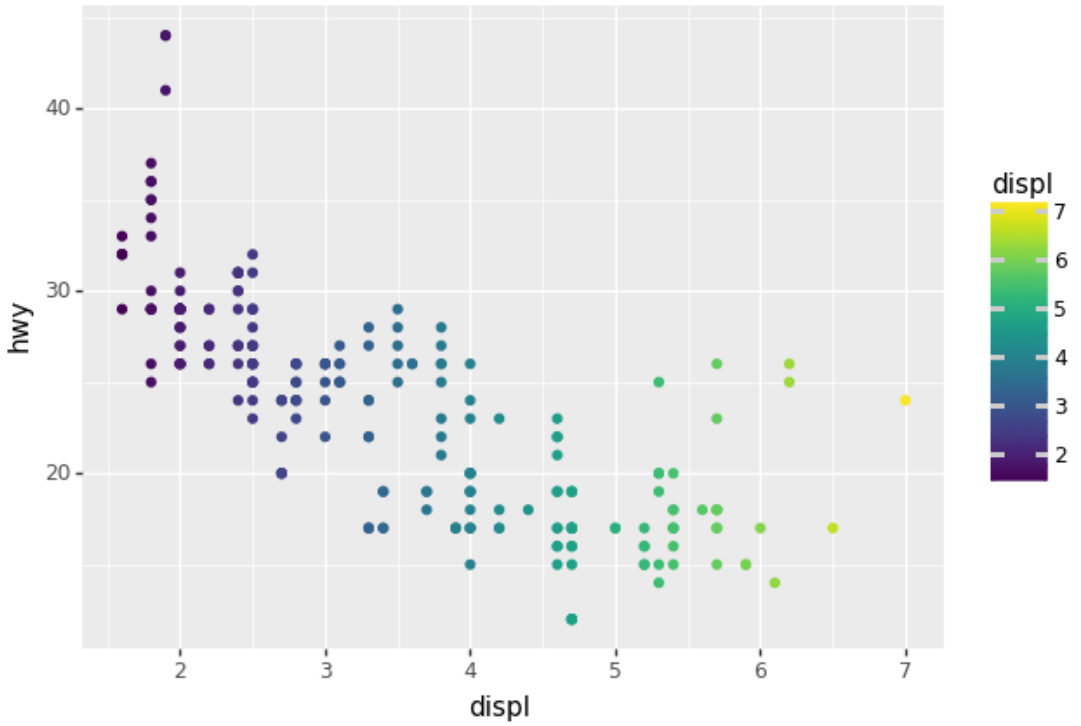
[22]:
```
classColors={x : colors[i] for i,x in enumerate(mpg["class"].unique())}
plt.scatter(mpg["displ"],mpg["hwy"],c=mpg["class"].map(classColors))
plt.show()
# this is more work, we already have to manually fix the colors,
```

```
# and we do still not have a suitable legend
```



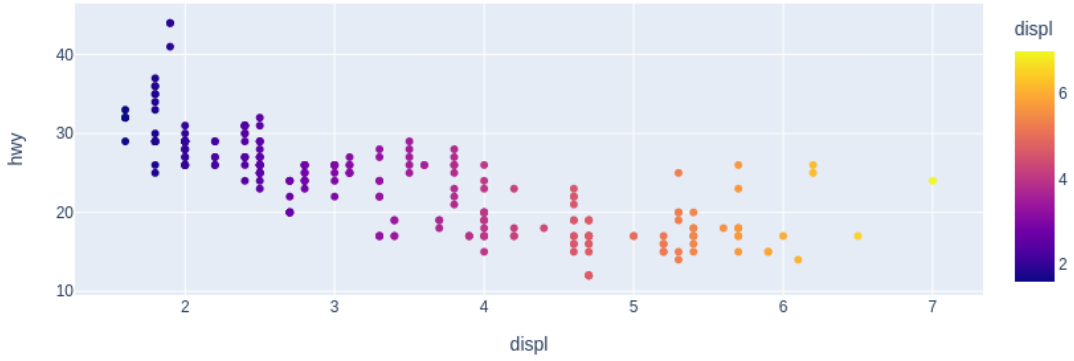
[24]:
```
# to demonstrate flexibility, play with some other choices
```

[25]:
```
# if we map a continuous value to color, we get a different colormap and legend␣
  ↪instead
ggplot(data = mpg) + \
  geom_point(mapping = aes(x = "displ", y = "hwy", color="displ"))
```



[25]:
```
<ggplot: (8765868034691)>
```

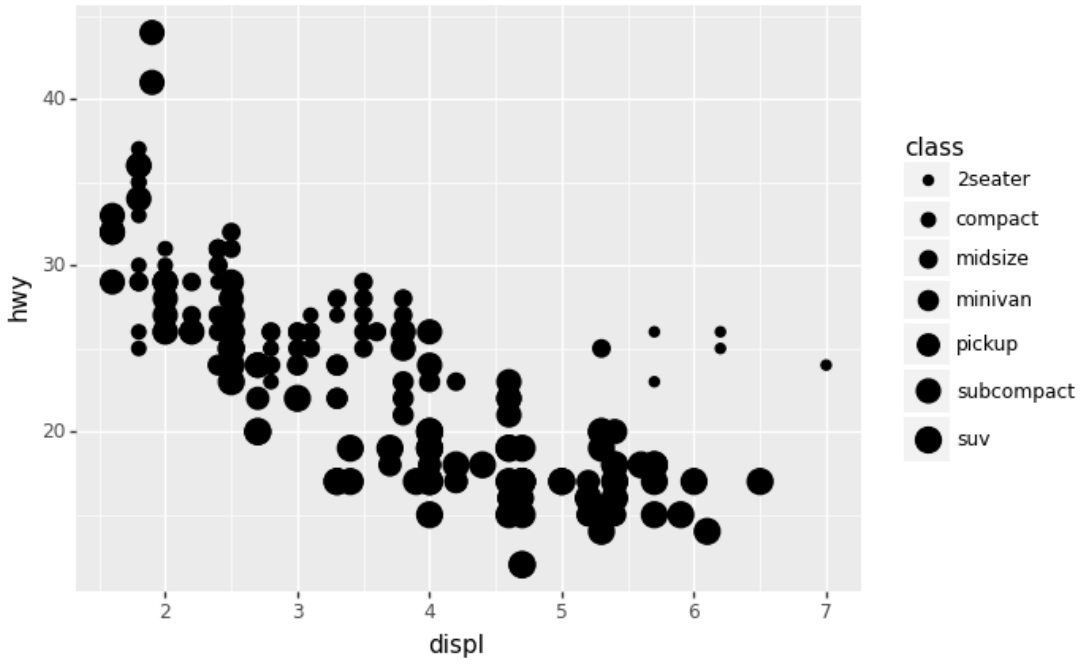
[30]:
```
fig = px.scatter(mpg, x="displ", y="hwy", color="displ")
fig.show()
```



[24]:
```
# nice touch: get a warning when an inappropriate choice is made
ggplot(data = mpg) + \
```

```
  geom_point(mapping = aes(x = "displ", y = "hwy", size="class"))
```

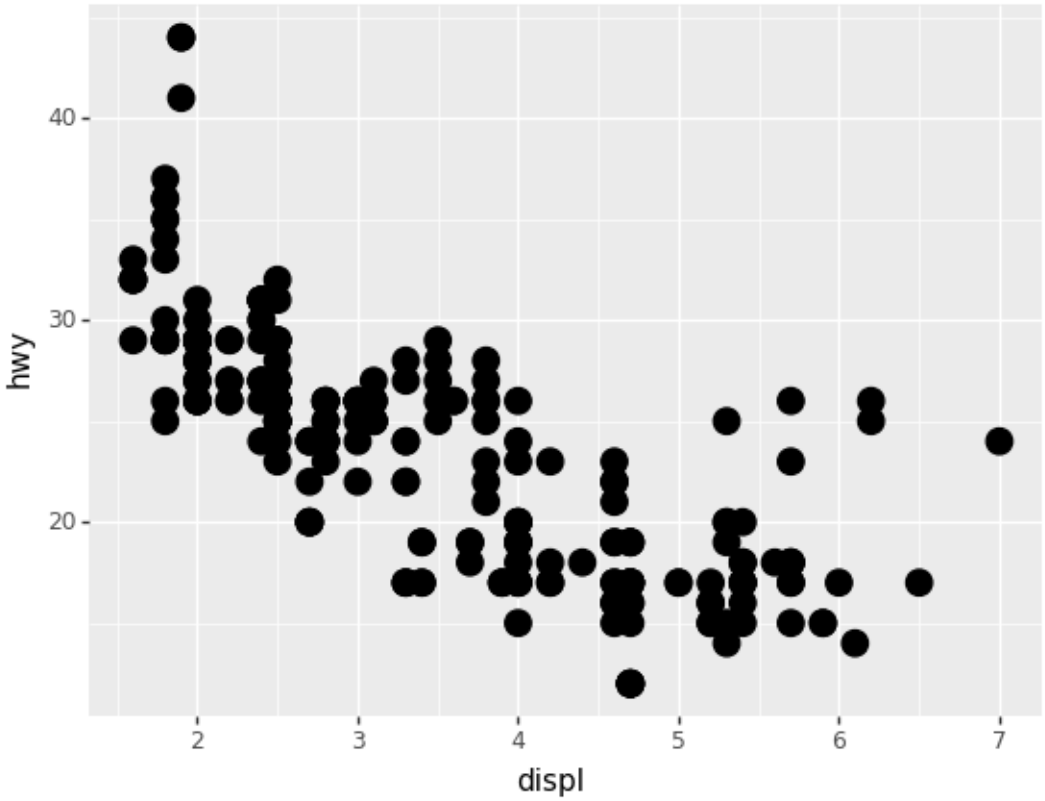
```
/home/bernhard/.local/lib/python3.10/site-
packages/plotnine/scales/scale_size.py:48: PlotnineWarning:

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



[24]: <ggplot: (8765868012169)>

[11]:
```
# plotly outright refuses to map a categoical type to size
#fig = px.scatter(mpg, x="displ", y="hwy", size="class")
#fig.show()
```

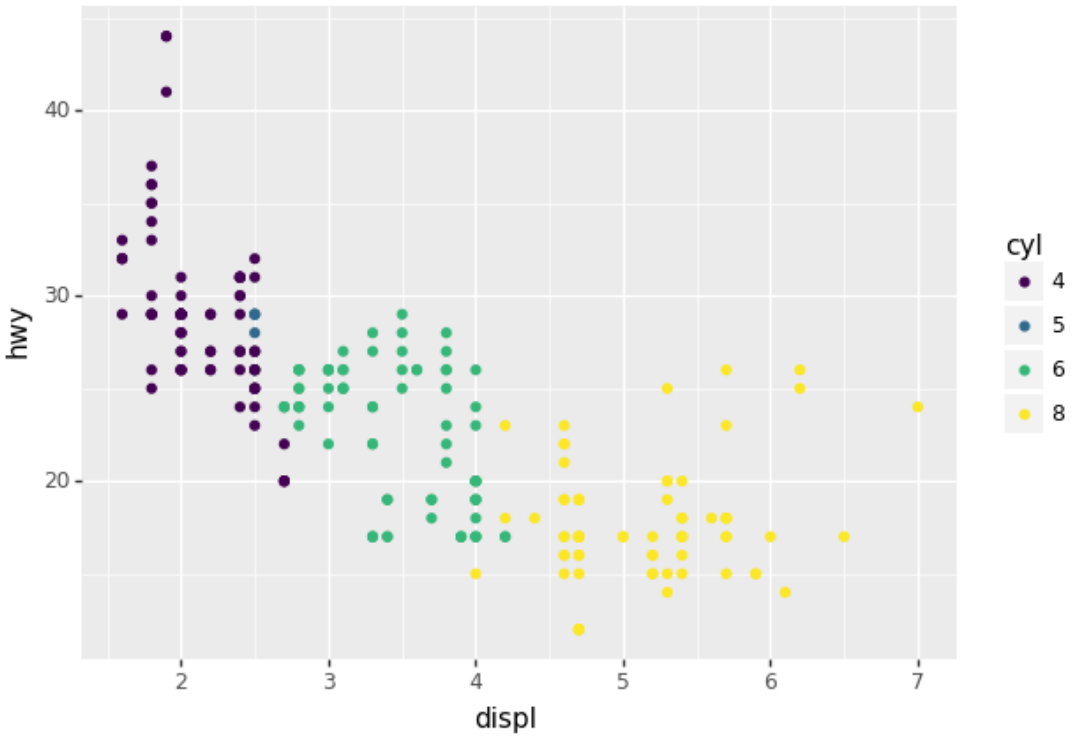
[226]:
```
# finally: distinguish mapping variable to an aesthetic vs directly specifying␣
  ↪a visual attribute
ggplot(data = mpg) + \
  geom_point(mapping = aes(x = "displ", y = "hwy"),size=5)
```



[226]: <ggplot: (8765857771031)>

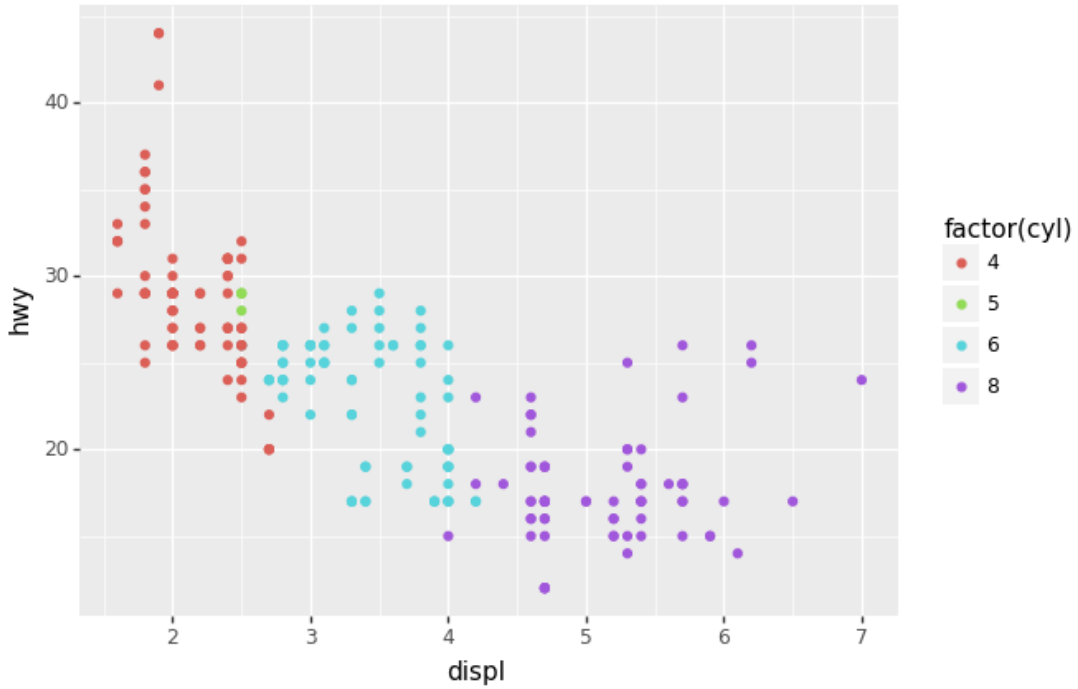
Revisit the issue of interval / ordinal / categorical variables The variable "cyl" gives number of cylinders of car, as an integer. But for visualization purposes it might be desirable to treat it as a categorical type.

[14]:
```
mpg2=mpg.copy()
mpg2["cyl"]=pd.Categorical(mpg2["cyl"],ordered=True)
#mpg2["cyl"]=pd.Categorical(mpg2["cyl"])

ggplot(data = mpg2) + \
  geom_point(mapping = aes(x = "displ", y = "hwy", color="cyl"))
```



[14]: <ggplot: (8741369668825)>

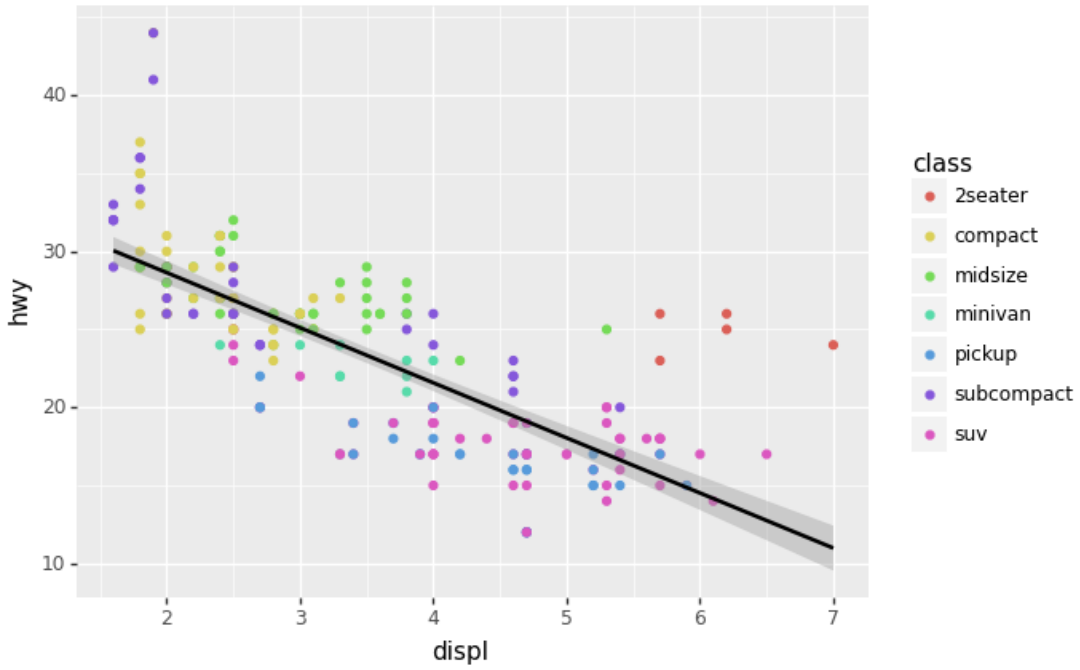
[15]:
```
# since this is a common issue, there is a keyword to do this "inline" in␣
 ↪plotnine
# note: this produces an unsorted category! so it might be cleaner to do this
# in advance and consider the factor() keyword to be a convenient hack
ggplot(data = mpg) + \
  geom_point(mapping = aes(x = "displ", y = "hwy", color="factor(cyl)"))
```



[15]: <ggplot: (8741369577974)>

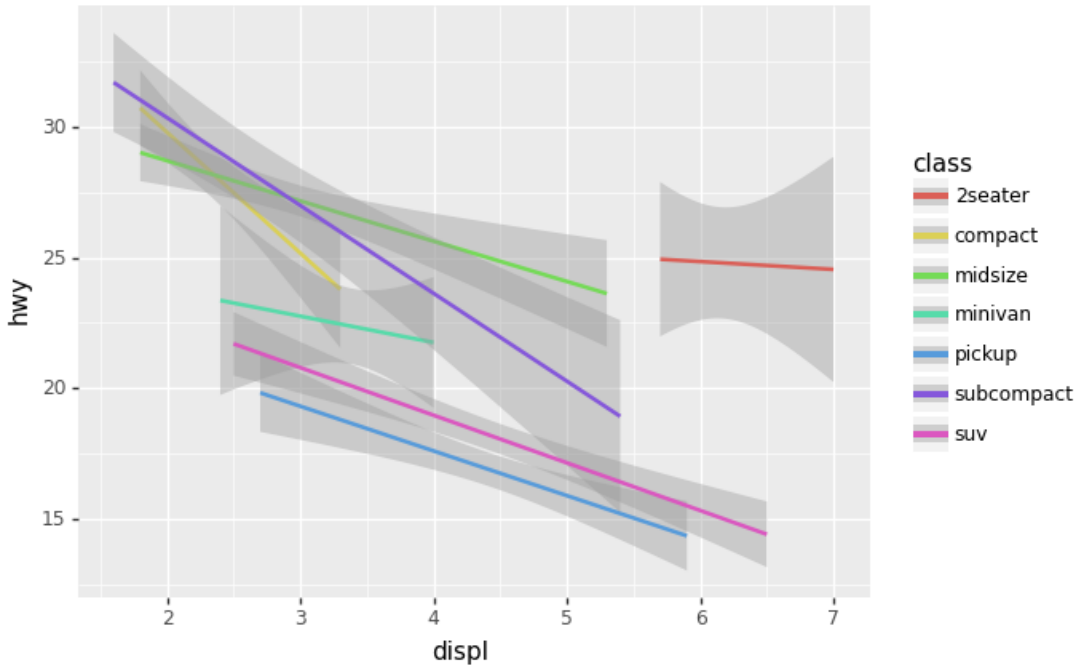
Combining multiple geoms / layers

[16]:
```
# add a second "geom" which provides a linear regression, using the same data
ggplot(data = mpg) + \
  geom_point(mapping = aes(x = "displ", y = "hwy", color="class"))+\
  geom_smooth(mapping = aes(x = "displ", y = "hwy"),method="lm")
```



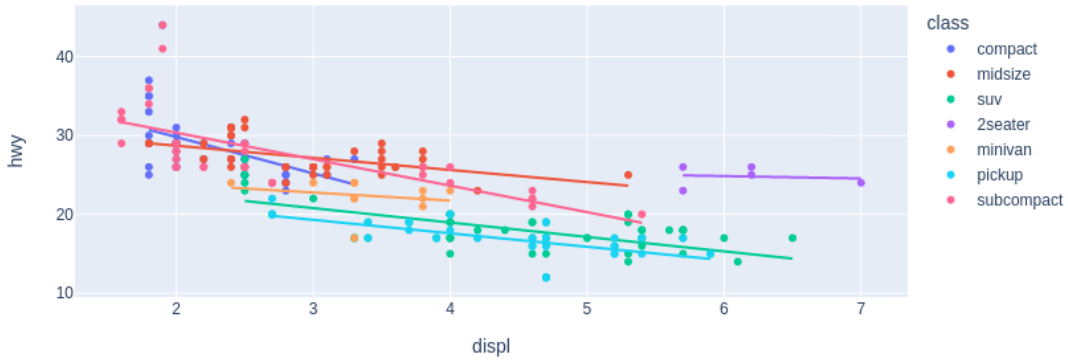
[16]: <ggplot: (8741369487327)>

[18]:
```
# can also specify an aes mapping at the level of the whole figure
# then each geom object will use that
# now the linear regression is applied to each class separately
ggplot(data = mpg, mapping = aes(x = "displ", y = "hwy", color="class")) + \
    geom_point()+\
    geom_smooth(method="lm")
```



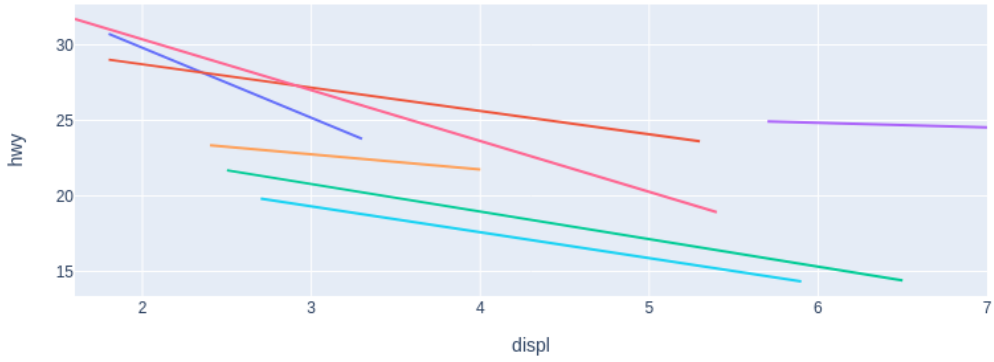
[18]: <ggplot: (8741369417650)>

[19]:
```
# plotly can also display trendlines but this is not done via a separate "geom"
# but by a keyword in the scatter plot, so it is much less "structured"
# by the way: ols stands for "ordinary least squares"
fig = px.scatter(mpg, x="displ", y="hwy", color="class",trendline="ols")
fig.show()
```



[21]:
```
# but got to leave it to plotly: the figure datastructure is quite systematic!
#fig.data
```

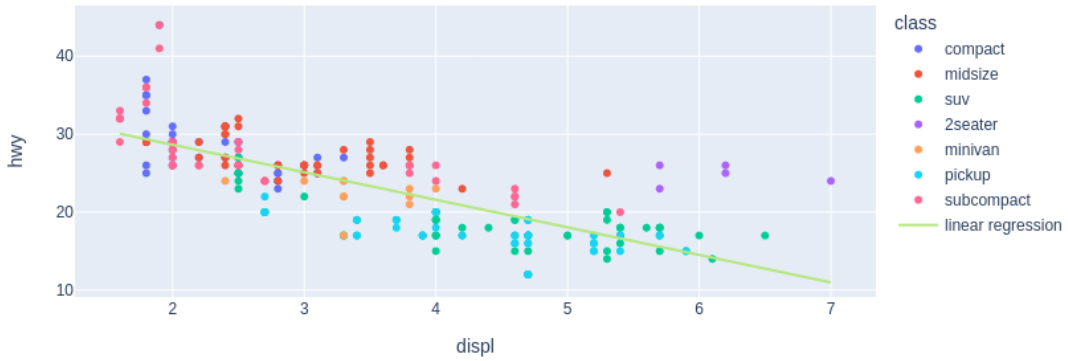
[22]:
```
# if we insisted, we could "isolate" the trendlines
fig = px.scatter(mpg, x="displ", y="hwy", color="class",trendline="ols")
fig.data=fig.data[1::2]
fig.show()
```



Obviously, generating the trendline requires some statistical analysis of the data. In the grammar of graphics there is a separate component for statistical transformations, located "between" providing the data and mapping its variables to aestetics (and which defaults to the "identity mapping" or appropriate choice, if not specified explicitly). This layer is not explicitly contained in plotly. It would therefore have to be implemented with other libraries, e.g. sklearn.

[24]:
```
from sklearn.linear_model import LinearRegression
```

[116]:
```
model = LinearRegression()
model.fit(mpg["displ"].values.reshape(-1, 1), mpg["hwy"].values.reshape(-1, 1))

x_range = np.linspace(mpg["displ"].min(), mpg["displ"].max(), 100).ravel()
y_range = model.predict(x_range.reshape(-1, 1)).ravel()

fig = px.scatter(mpg, x="displ", y="hwy", color="class")
fig.add_traces(go.Scatter(x=x_range, y=y_range, name='linear regression'))
fig.show()

# observing the overhead (and we have not even applied it "by group") that this␣
 ↪involves
# we might agree that it makes sense to include (at least some frequently used)␣
 ↪statistical
# analyses / transformations into the graphics grammar
```



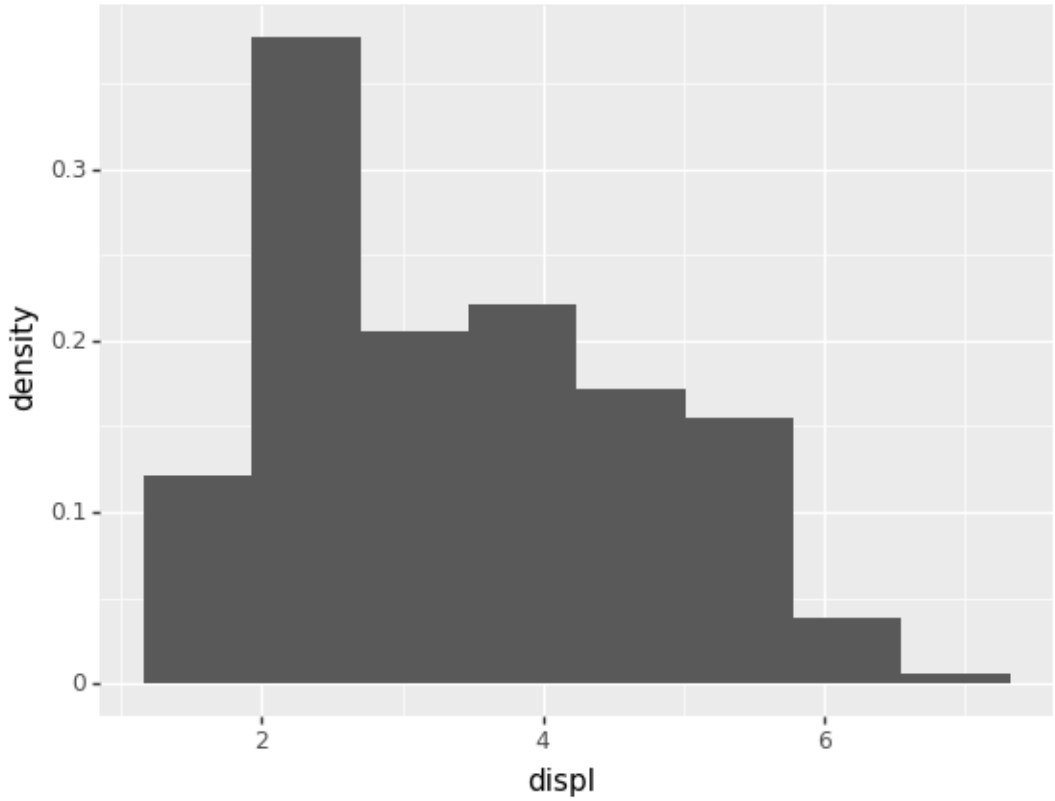
1.2.2 Stats

As we have seen in previous examples, some "geoms" implicitly apply some "statistical transformation" to the data. Let us analyze this in more detail and provide more such potential "stats" modules.

Getting started with histograms

[27]:
```
# a very relevant statistical transformation is the histogram
# note: in ggplot / plotnine stats and geoms are closely tied to each other
# so they are not invoked separately, but one calls one and specifies "the␣
 ↪other" via a
# keyword argument

mode=2
# simply get a histogram over displ
if mode==0:
    fig=ggplot(data = mpg, mapping = aes(x = "displ")) + \
        gg.geom_histogram(bins=8)

# here a lot of "educated guesses for default values" are used. the above is␣
 ↪equiv to:
elif mode==1:
    fig=ggplot(data = mpg, mapping = aes(x = "displ")) + \
        gg.geom_histogram(stat="bin",bins=8,mapping=aes(y=after_stat("count")))

# here: "count" is a new variable generated by the stat "bin" (which is the␣
 ↪default stat
# associated with geom_histogram), and it can be accessed via the "distinction"␣
 ↪after_stat
```

```
# alternatively: use the new variable "density"
elif mode==2:
    fig=ggplot(data = mpg, mapping = aes(x = "displ")) + \
        gg.geom_histogram(bins=8,mapping=aes(y=after_stat("density")))

fig
```

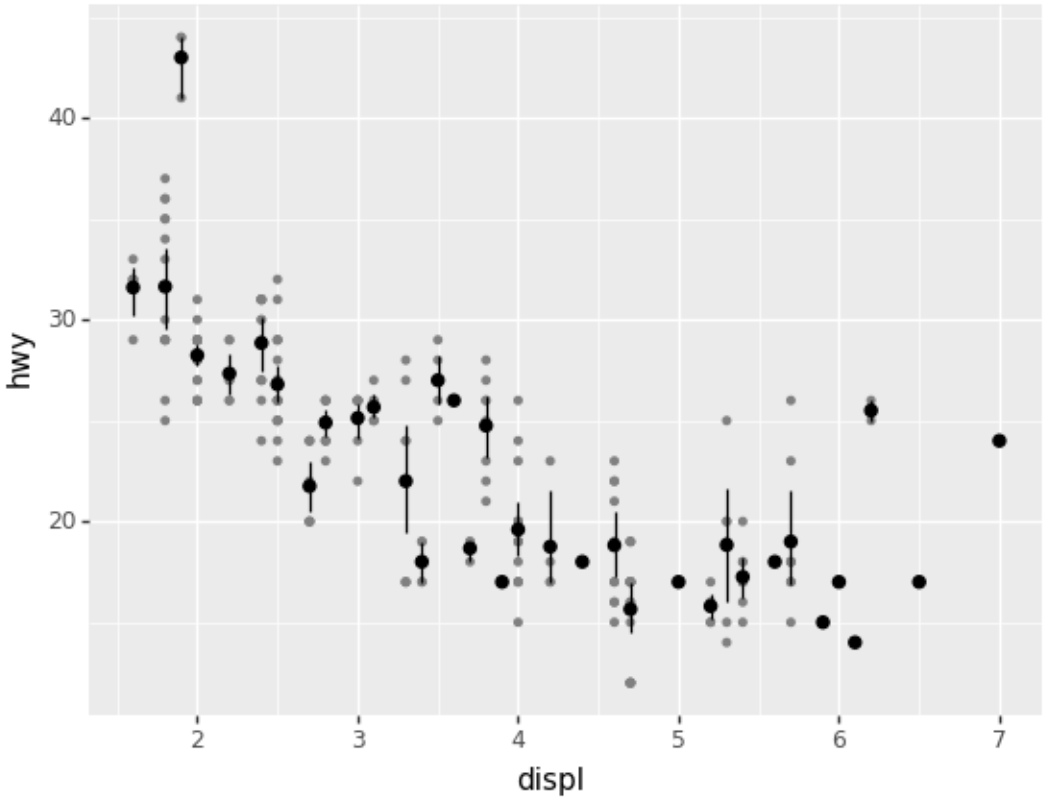


[27]: <ggplot: (8741369339944)>

A stat can be combined with different geoms

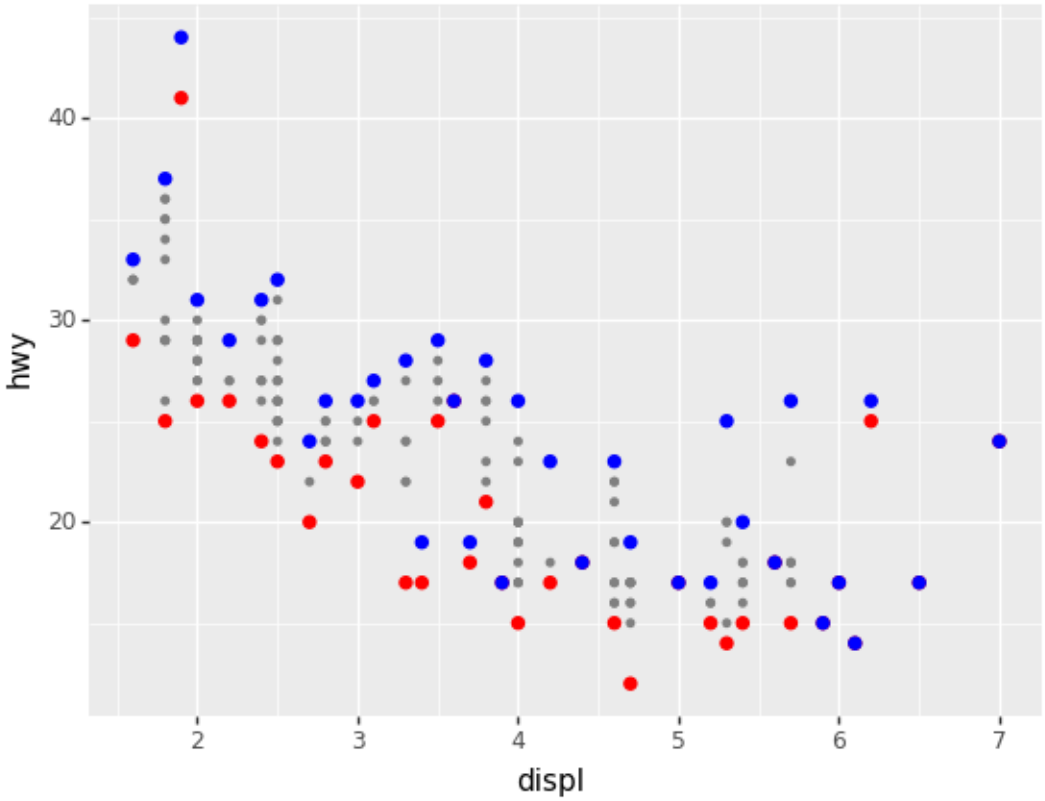
[31]:
```
# here we use another stat: "summary", which computes some function
# of the values at a given position
# by default this is combined with the "pointrange" geom
# which visualizes a "center", lower and upper end position
# one can specify different functions that compute these positions
# by default it provides an estimate for the mean and a confidence interval
ggplot(data = mpg, mapping = aes(x = "displ",y="hwy")) + \
  geom_point(color="grey",size=1)+\
  gg.geom_pointrange(stat="summary")
```

```
  #gg.geom_pointrange(mapping=aes(ymax="hwy+1",ymin="hwy-1"))
```



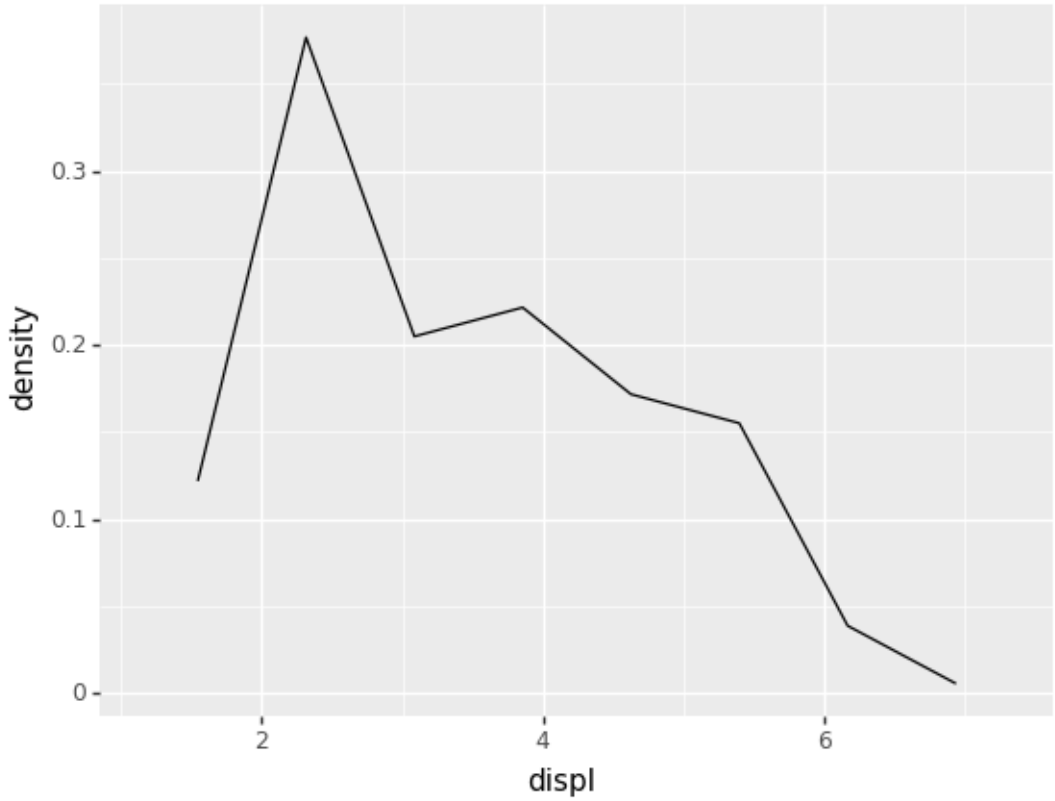
[31]: <ggplot: (8741369291518)>

[33]:
```
# but: we can use the same stat with another geom, merely use the "y"-aes and
# use a different function to compute the y from the originally provided y
ggplot(data = mpg, mapping = aes(x = "displ",y="hwy")) + \
  gg.geom_point(color="gray",size=1)+\
  gg.geom_point(stat="summary",size=2,fun_y=np.min,color="red")+\
  gg.geom_point(stat="summary",size=2,fun_y=np.max,color="blue")
```



[33]: <ggplot: (8741369132714)>

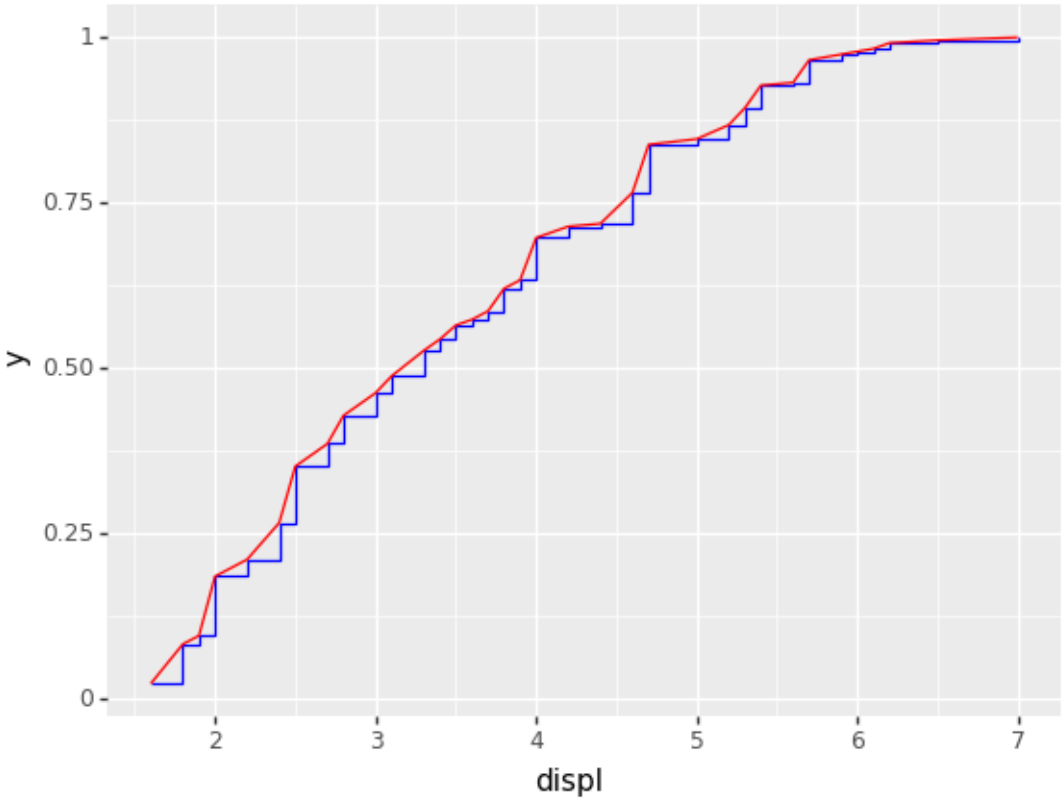
[34]:
```
# another example of combining a stat with another geom
ggplot(data = mpg, mapping = aes(x = "displ")) + \
  gg.geom_line(stat="bin",bins=8,mapping=aes(y=after_stat("density")))
```



[34]: <ggplot: (8741369059560)>

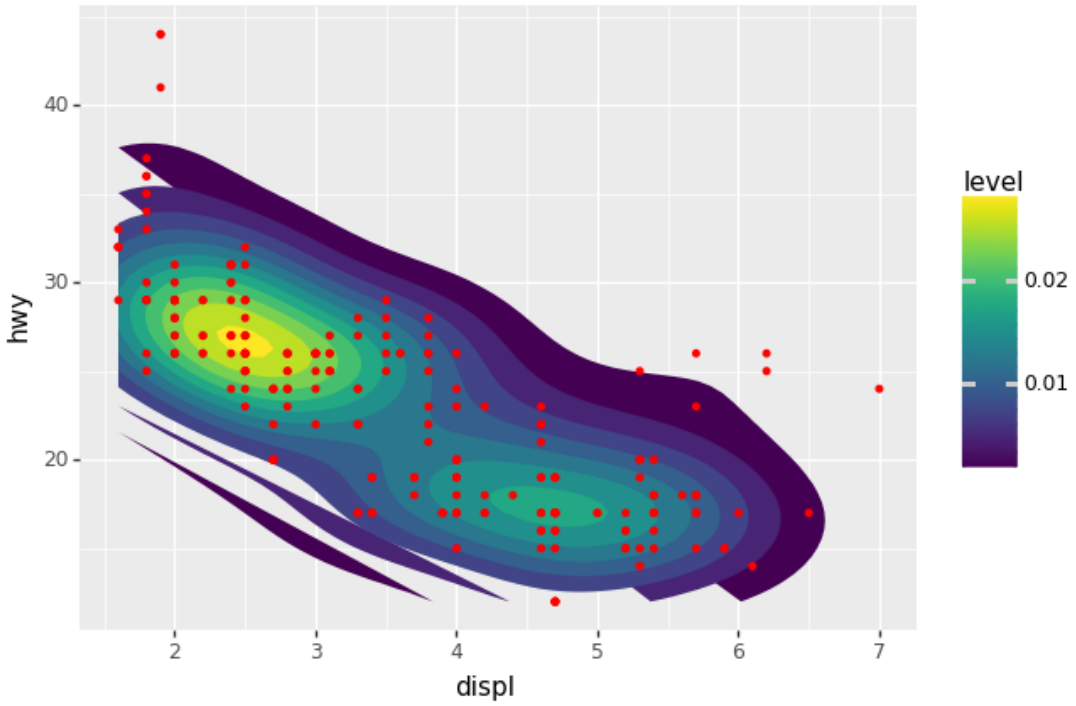
More stats and generated variables

[36]:
```
ggplot(data = mpg, mapping = aes(x = "displ")) + \
    gg.stat_ecdf(geom="step",color="blue")+\
    gg.geom_line(stat="ecdf",color="red")
```



[36]: <ggplot: (8741369020567)>

[40]:
```
# another example, where the same stat under the hood is used by different geoms
# and intuitively it is very clear that the two plots show the same data
# but merely with different visual encodings
mode=1
if mode==0:
    fig=ggplot(data = mpg, mapping = aes(x = "displ",y="hwy")) + \
        geom_point()+\
        gg.geom_density_2d(levels=10, mapping=aes(color=after_stat("level")))
if mode==1:
    fig=ggplot(data = mpg, mapping = aes(x = "displ",y="hwy")) + \
        gg.stat_density_2d(geom = "polygon",␣
 ↪mapping=aes(fill=after_stat("level")),levels=10)+\
        geom_point(color="red",size=1)

fig
# but here in the second option we see the limitations/challenges of this␣
 ↪expected seamless interaction
# between different objects
```



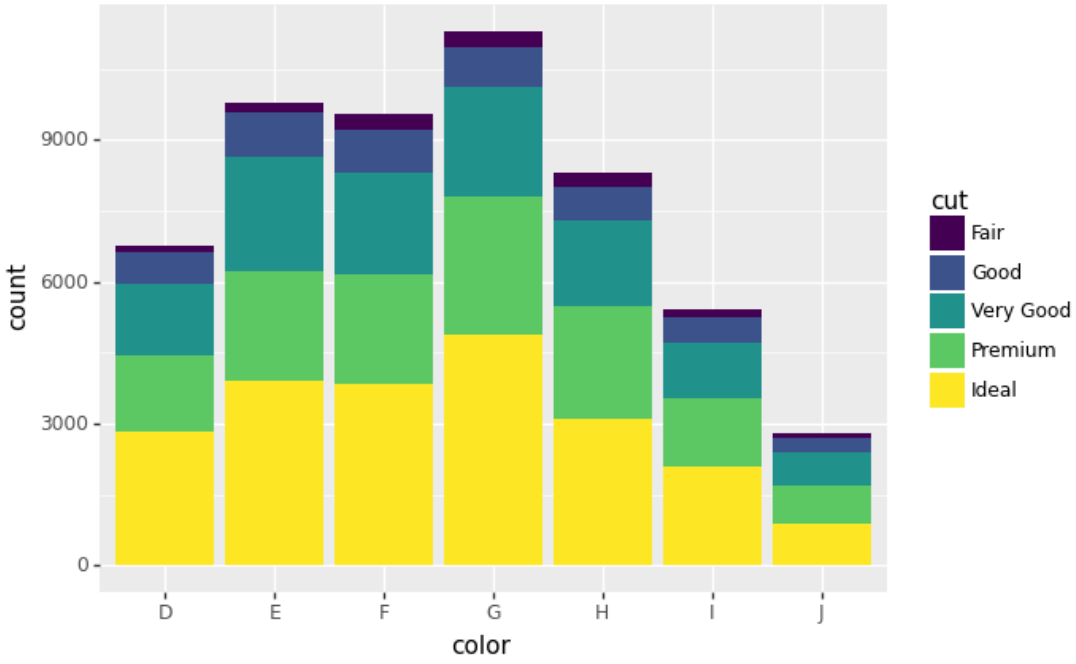
[40]: <ggplot: (8741368781831)>

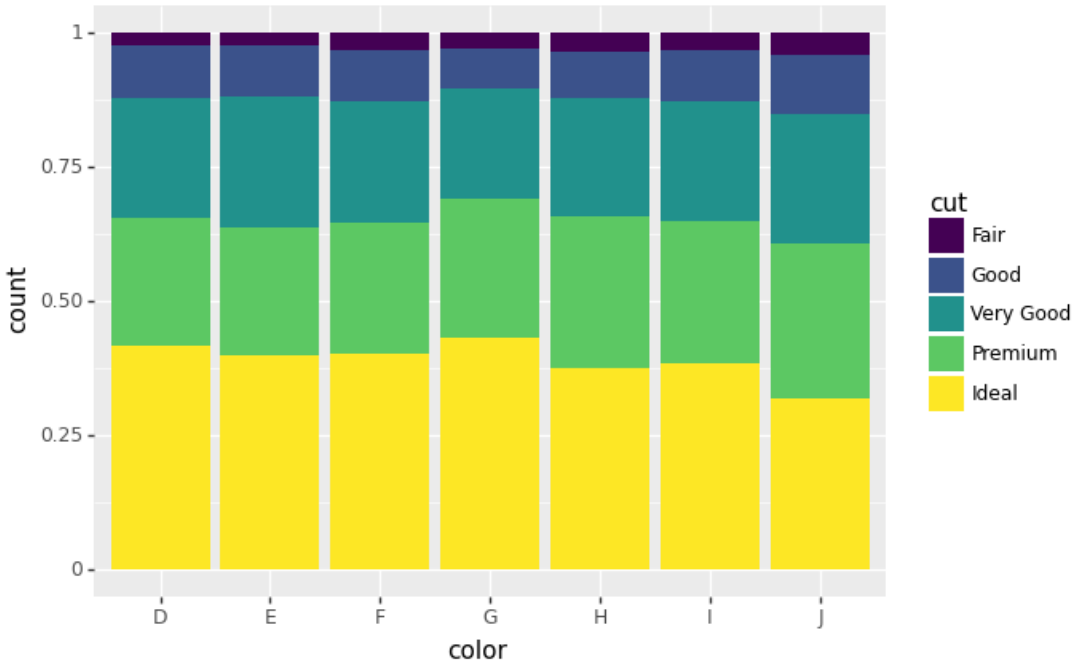
1.2.3 Position adjustments

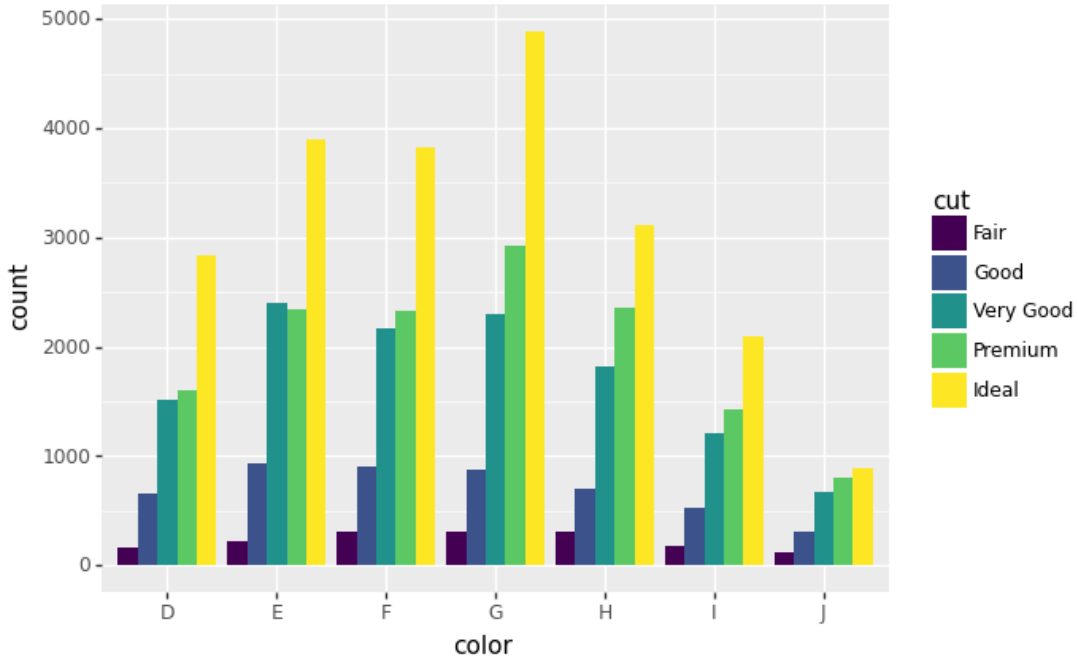
Another minor ingredient in generating plots are position adjustments applied to the geometric objects. We just give a few brief illustrations. (Our goal here is not to master the whole grammer in one session.)

[41]:
```
diamonds=gg.data.diamonds
```

[47]:
```
# here are some example position adjustments for bars
fig=ggplot(diamonds, aes(x="color", fill = "cut"))
# position stack is the default for bars, so `geom_bar()`
# is equivalent to `geom_bar(position = "stack")`.
#[fig+x for x in [gg.geom_bar(), gg.geom_bar(position = "fill"), gg.
 ↪geom_bar(position = "dodge")]]
[fig+gg.geom_bar(position=x) for x in ["stack","fill","dodge"]]
```


9000
6000
3000
0
count
D
E
F
G
H
I
J
color
cut
Fair
Good
Very Good
Premium
Ideal


1
0.75
0.50
0.25
0
count
D
E
F
G
H
I
J
color
cut
Fair
Good
Very Good
Premium
Ideal



[47]:
```
[<ggplot: (8741369669005)>,
 <ggplot: (8741369674011)>,
 <ggplot: (8741369640312)>]
```

[312]:
```
# for points there are other options, e.g. random jitter, small systematic␣
↪nudges,...
```

1.3 Overview of the grammar

1.3.1 Layers

The above has illustrated the *layer* component of the grammar. Each layer consists of the following ingredients: * data * a mapping of variables to aesthetics * a statistical transformation * a mapping to geometric objects * a position adjustment

The other components will briefly be illustrated below.

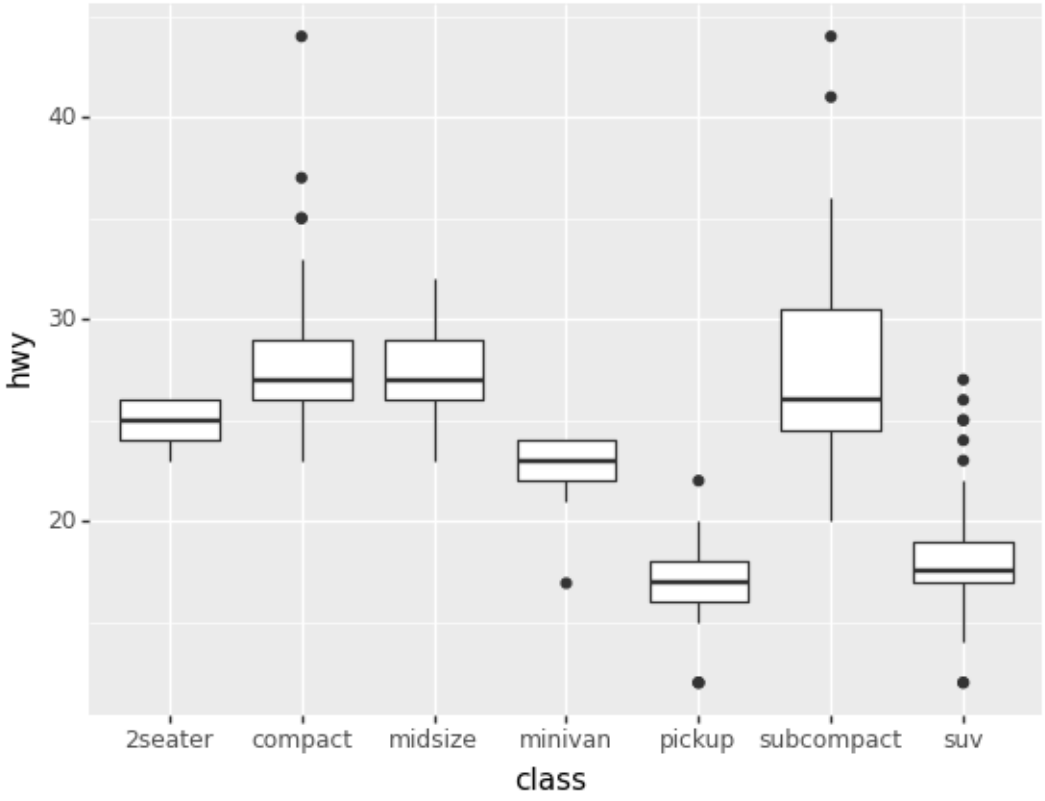
1.3.2 Scales, axes and legends

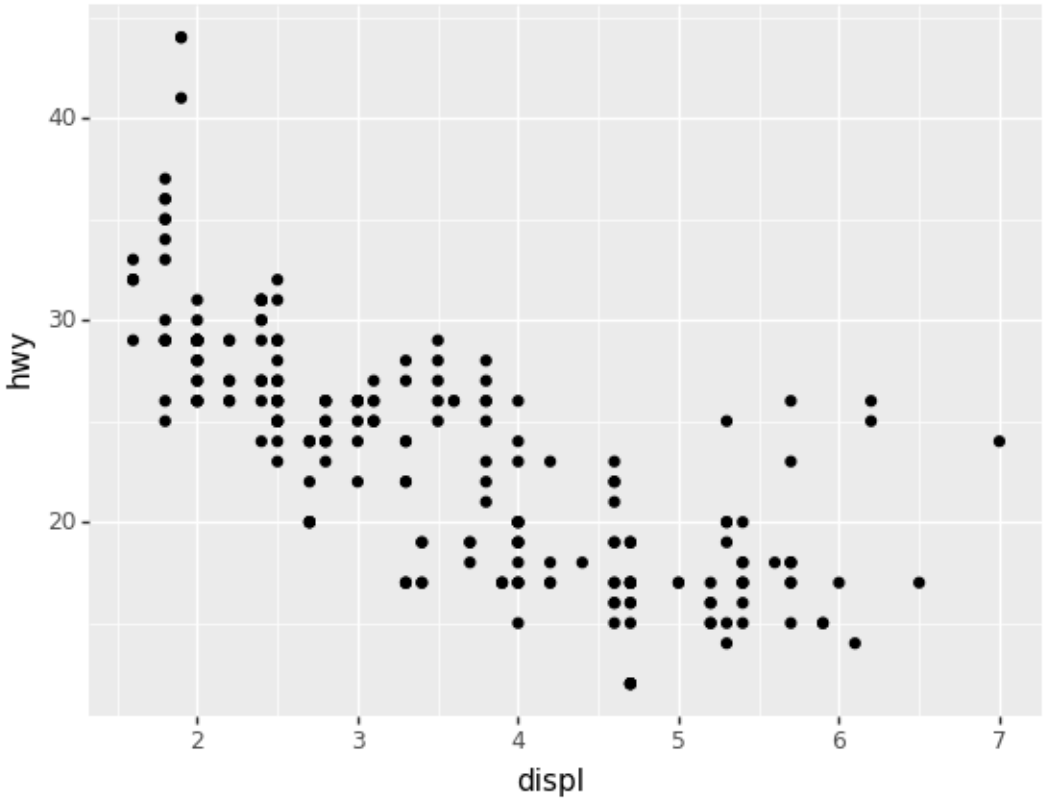
In the previous examples, variables got mapped to aesthetics, which implies a map from individual values of variables (data range) to a value in the aesthetic space. In the grammar of graphics these maps are called *scales*.

Axes and legends play the role of the *inverse function* that allow the viewer to decode the aesthetics, back to data values. So even if they look very different, they play the same role.

In the above we noticed that plotnine/ggplot chose various appropriate scales automatically, e.g. for continuous, discrete or categorical variables. But in principle, their declaration is part of the grammar.

[303]:
```
# discrete vs continuous x-axis
fig1=ggplot(data = mpg, mapping = aes(x = "class",y="hwy")) + \
        gg.geom_boxplot()
fig2=ggplot(data = mpg, mapping = aes(x = "displ",y="hwy")) + \
        gg.geom_point()
(fig1,fig2)
```



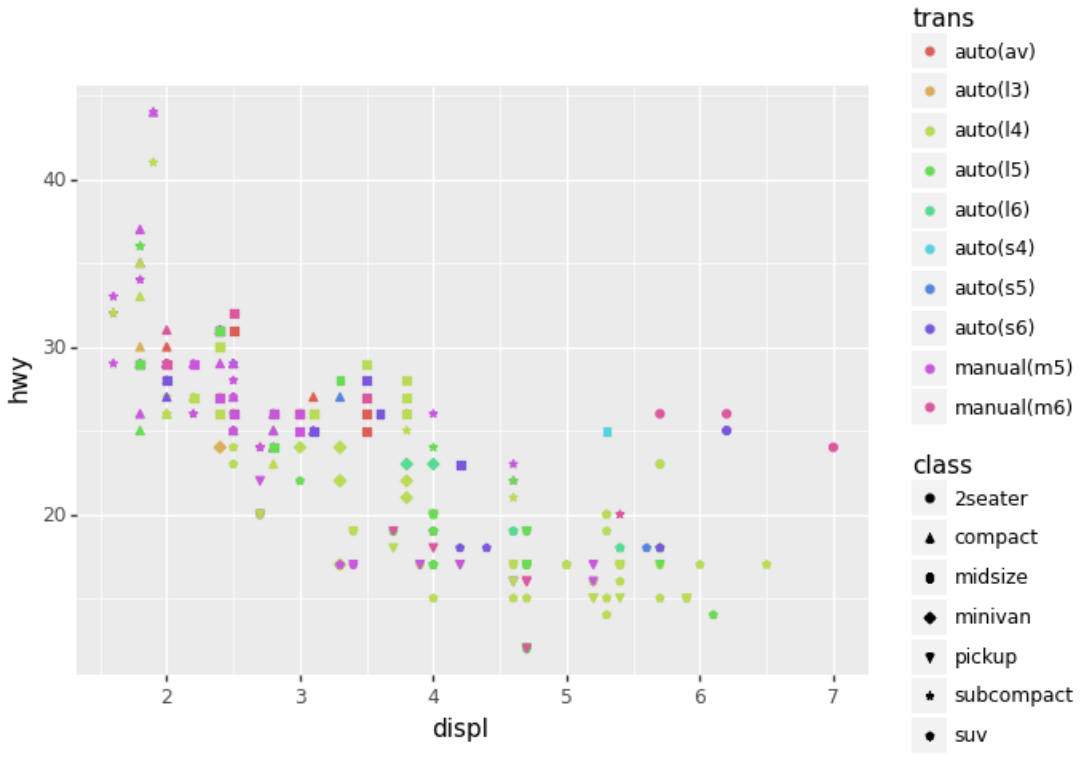


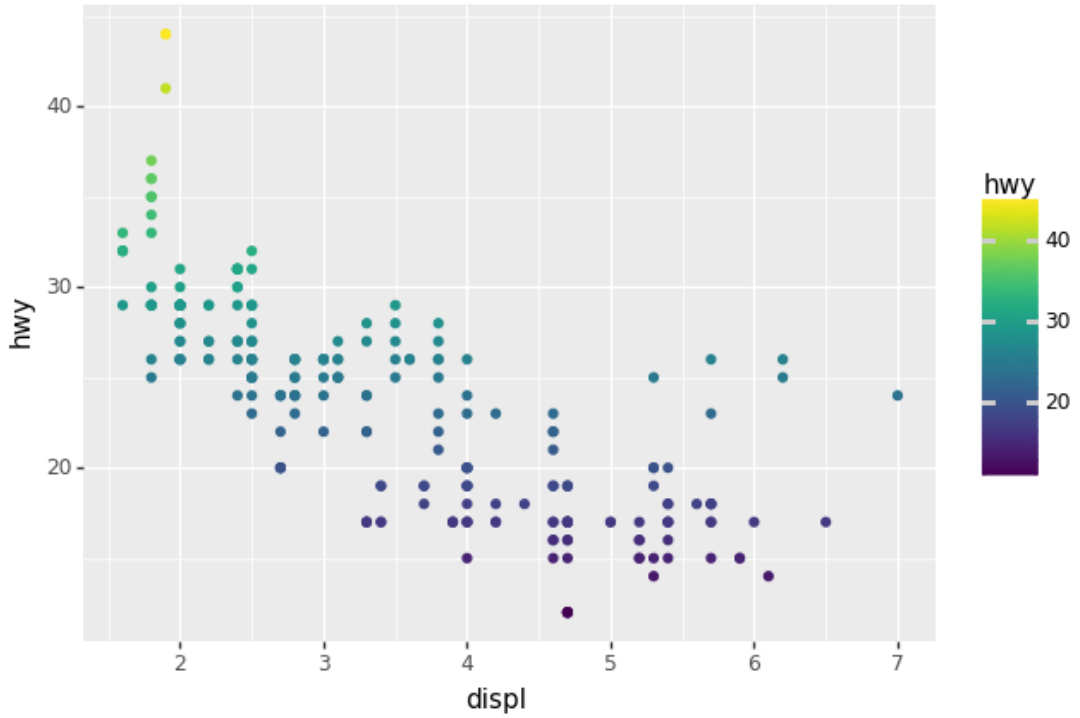
[303]:
```
(<ggplot: (8765853242134)>, <ggplot: (8765853251864)>)
```

[306]:
```
mpg.columns
```

[306]:
```
Index(['manufacturer', 'model', 'displ', 'year', 'cyl', 'trans', 'drv', 'cty',
       'hwy', 'fl', 'class'],
      dtype='object')
```

[307]:
```
# and as before: discrete vs continuous colormaps
fig1=ggplot(data = mpg, mapping = aes(x =␣
 ↪"displ",y="hwy",color="trans",shape="class")) + \
       gg.geom_point()
fig2=ggplot(data = mpg, mapping = aes(x = "displ",y="hwy",color="hwy")) + \
       gg.geom_point()
(fig1,fig2)
```


trans
auto(av)
auto(l3)
auto(l4)
auto(l5)
auto(l6)
auto(s4)
auto(s5)
auto(s6)
manual(m5)
manual(m6)
class
2seater
compact
midsize
minivan
pickup
subcompact
suv
hwy
displ
40
30
20
2
3
4
5
6
7

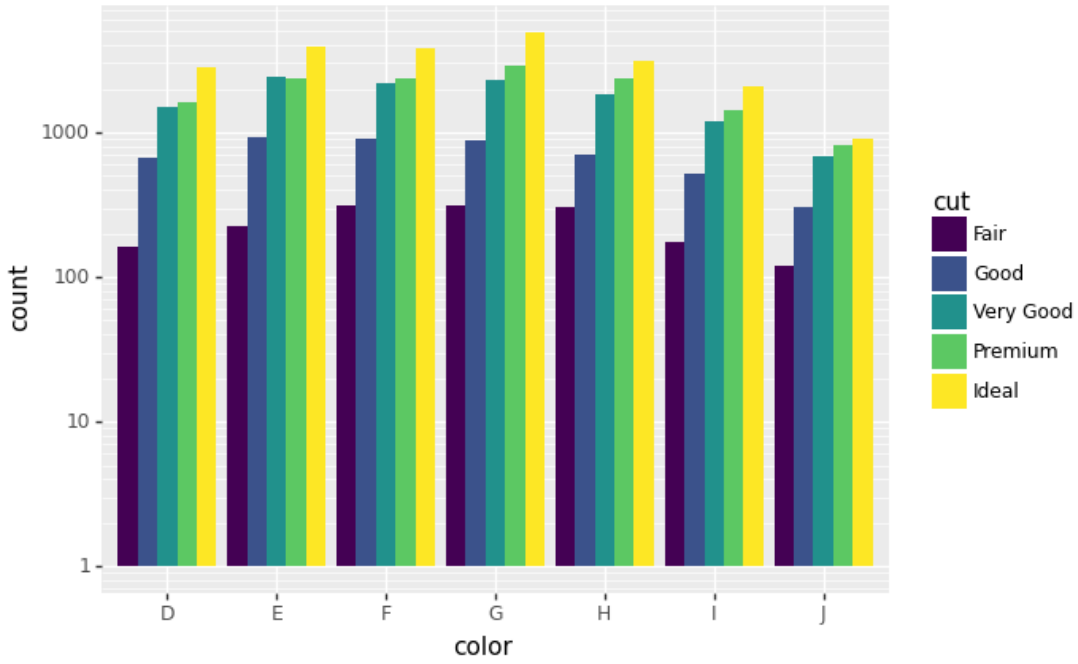

hwy
displ
hwy
40
30
20
2
3
4
5
6
7

[307]: (<ggplot: (8765853218342)>, <ggplot: (8765853070901)>)

The set of scales is approximately given by: * position scales (with axes) * color scales * markers, size, line width and style

(By the way, as seen above: merely because a graph was generated with the grammar, does not automatically make it good or even meaningful!)

[49]:
```
# scales are also where logarithmic axis scaling is implemented
ggplot(diamonds, aes(x="color", fill = "cut"))+\
  gg.geom_bar(position = "dodge")+\
  gg.scale_y_log10()
```



[49]: <ggplot: (8741368816481)>

1.3.3 Coordinates

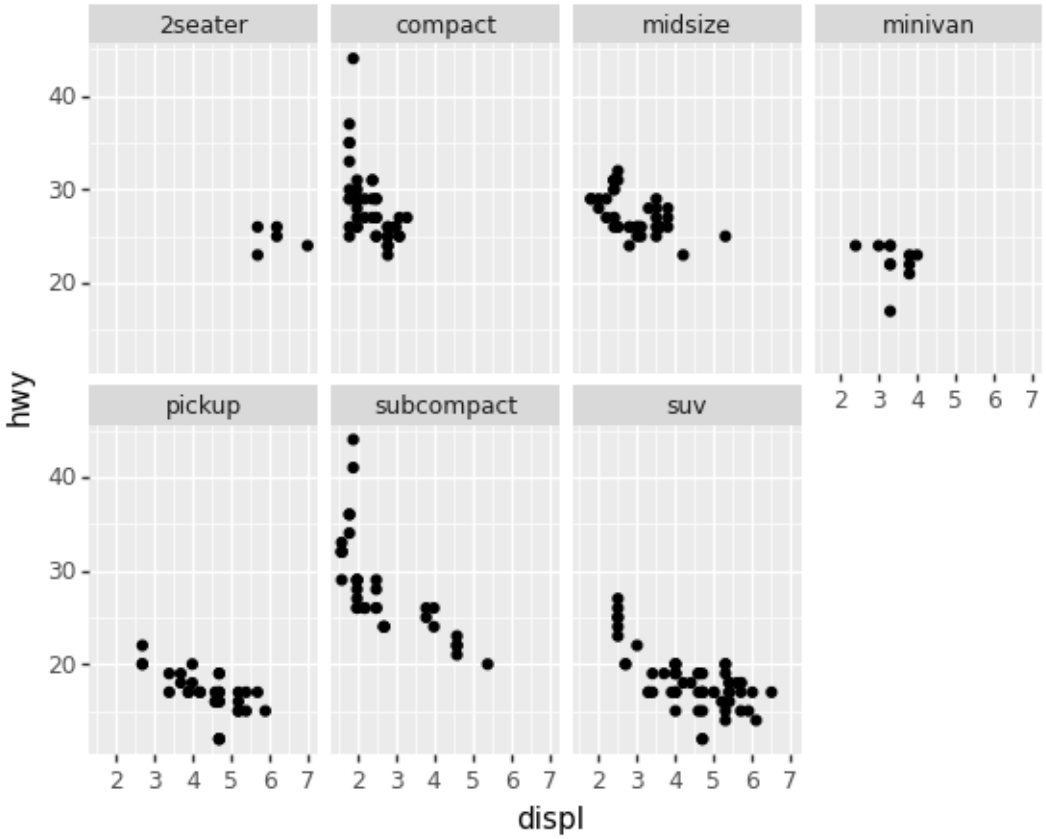
All of the above was in Cartesian coordinates for the x and y aesthetics. (With the exception of a logarithmic scaling.) But in principle there are other coordinate systems, such as polar coordinates, or map projections. These are implemented as separate component in the grammar. In principle, polar coordinates could be used to turn bar charts into pie charts. Everyone may decide for themselves how useful or profound that is. Either way: unfortunately, polar coordinates are not

yet implemented in plotnine at this point (of course, they are for the original ggplot) and thus cannot be demonstrated here.

1.3.4 Faceting

Instead of mapping the variable class to an aesthetic, we might also consider a small multiple over its various values. This is called "facting" in the grammar.

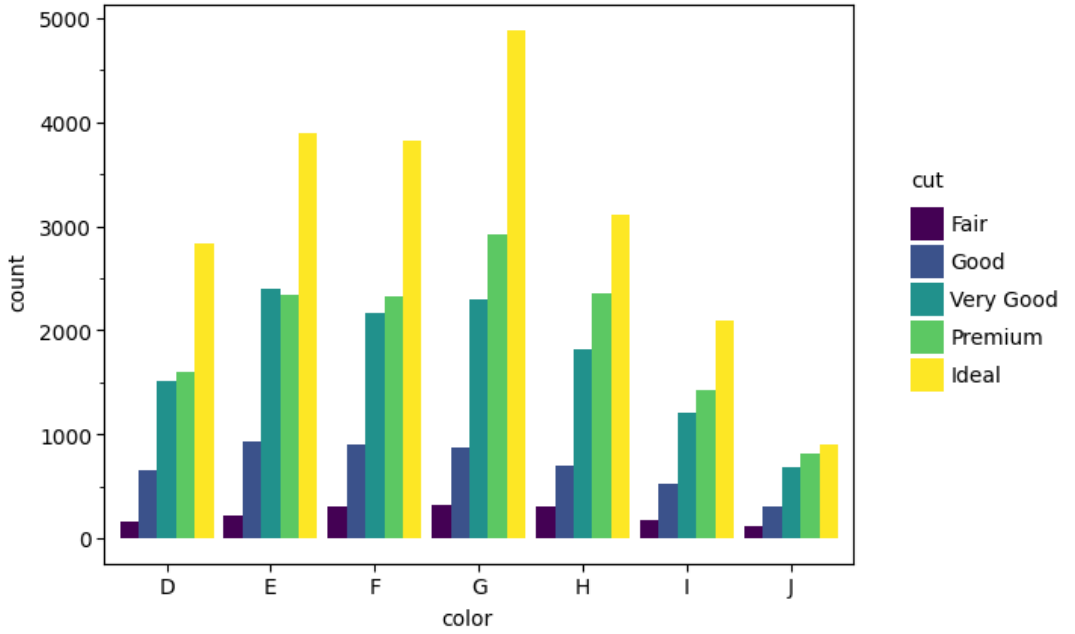
[121]:
```
ggplot(data = mpg) + \
  geom_point(mapping = aes(x = "displ", y = "hwy")) + \
  facet_wrap("~class", ncol = 4)
```

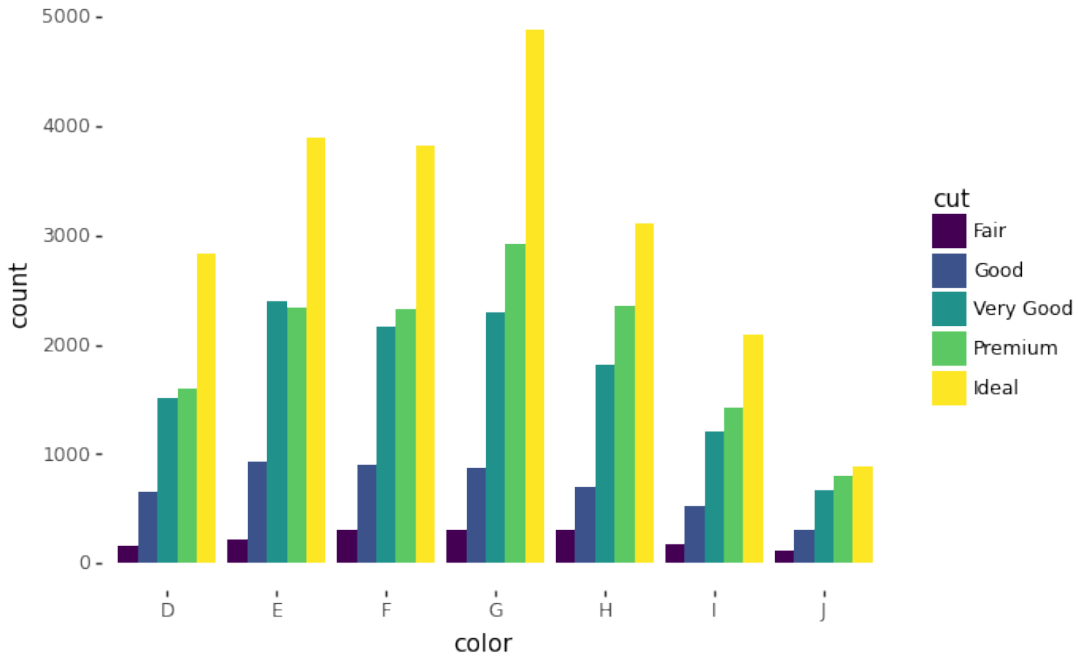


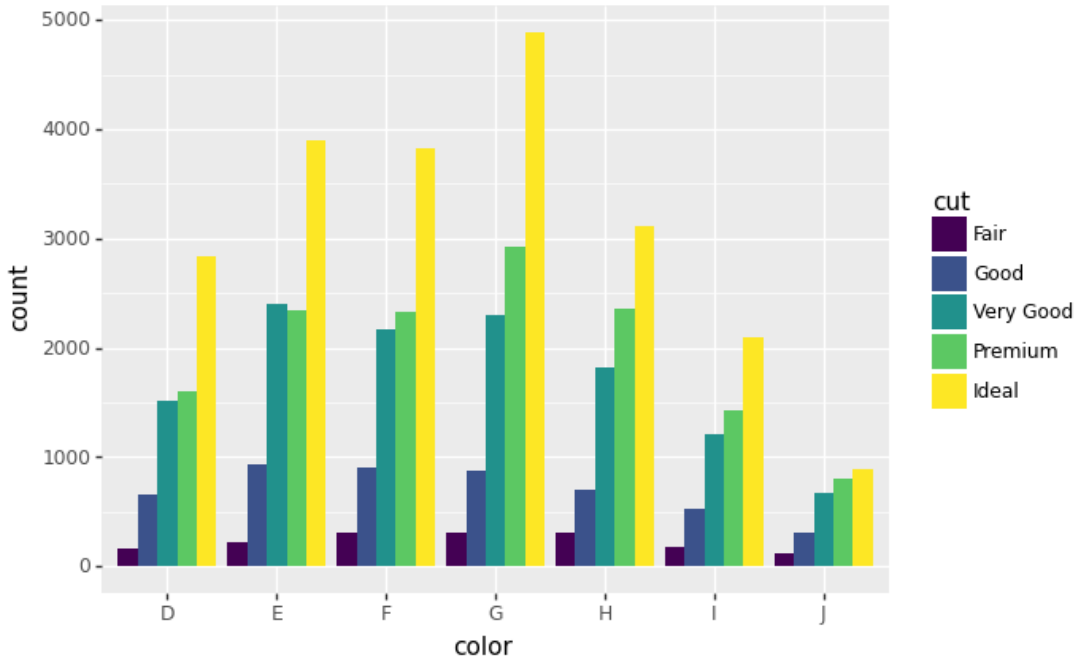
[121]: <ggplot: (8765866053359)>

1.3.5 Themes

[51]:
```
# scales are also where logarithmic axis scaling is implemented
fig=ggplot(diamonds, aes(x="color", fill = "cut"))+\
  gg.geom_bar(position = "dodge")
[fig+x for x in [gg.theme_matplotlib(),gg.theme_tufte(),gg.theme_grey()]]
```







[51]:
```
[<ggplot: (8741368243813)>,
 <ggplot: (8741368306301)>,
 <ggplot: (8741368196832)>]
```

Of course, one also has the option to adjust all styles individually. But it is nice to know that in principle one command at the end can make the plot visually compatible with some other design constraints (e.g. a corporate identity).

[ ]: