Data Visualization with R

Veronica

2016-11-11

Contents

1 Introduction to Graphics

Visualizing data is crucial in today's world. Without powerful visualizations, it is almost impossible to create and narrate stories on data. These stories help us build strategies and make intelligent business decisions.

R is well supported to make data visualization. It provides at least three main graphical systems: `graphics` that comes with base R, `lattice` that is an R implementation of William Cleveland's trellis graphics and `ggplot2` that is an R implementation of Leland Wilkinson's Grammar of Graphics. CRAN provides a Task View on Graphics, that is an organized list of all R packages about graphics, that includes not only the three main graphical systems but also packages to produce specialized plots.

In the following chapters, `ggplot2` graphics are shown. `ggplot2` package has become a synonym for data visualization in R and there are at least two main reasons to prefer it than other graphic systems:

- `ggplot2` is newer than other graphic systems;
- `ggplot2` is built on the idea of a semantics for graphics and there is much more emphasis on reshaping data, transformation, and assembling the elements of a plot.

`ggplot2` was written by Hadley Wickham and Winston Chang. If you are wondering, Wickham wrote also a `ggplot` package (without any number) that was archived by CRAN in 2008.

The starting point for more information about `ggplot2` is the official website providing documentation, the official mailing list and other resources.

Supposing the package is already installed, first of all `ggplot2` must be loaded.

```
require(ggplot2)
```

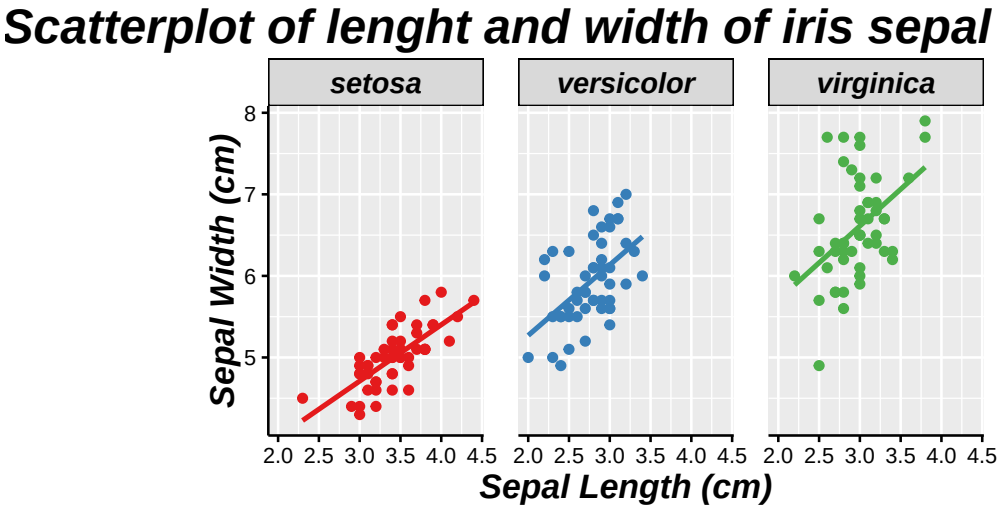
1.1 An overview of ggplot2 grammar

In order to unlock the full power of `ggplot2`, you need an overview of its undelying grammar. By understanding the grammar and how its components fit together, you can create a wider range of visualization, combine multiple source of data, and customize to your heart's content.

As we said, `ggplot2` is an R implementation of Leland Wilkinson's Grammar of Graphics, which is a formal and structured perspective on how to describe data graphics. `ggplot2` grammar is called layered as it is structured in building blocks. Its basic idea is to independently specify plot building blocks and combine them to create just about any kind of graphical display you want.

Let us see an example.

Let us consider `iris` dataset. `iris` dataset gives the measurements in centimeters of sepal length and width and of petal length and width, respectively, for 50 flowers from each of 3 species of iris. Suppose we want to visualize the relationship between length and width of iris sepals according to the different species:



The previous plot is composed of building blocks that are added to the plot one after the other.

The complete scheme of the most important building blocks of `ggplot2` grammar is displayed in the following figure:

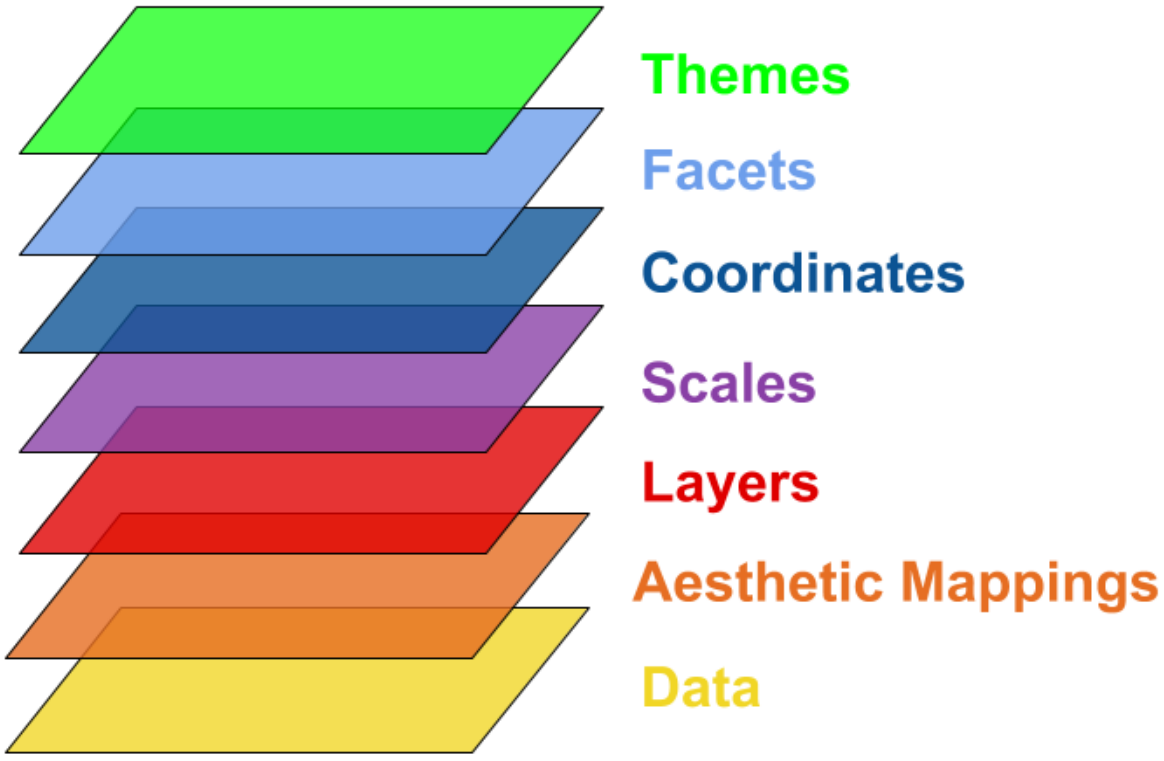


Figure 1:

The scheme must be read from bottom to top. Starting from bottom, the first three building blocks (Data , Aestethic Mappings and Layers) are fundamental to build a simple plot, indeed they are called **"key"** building blocks. The remaining building blocks (Scales , Coordinates , Facets and Themes) allow us to build a complex plot and to customize it; their use and order is not compulsory.

Let us briefly describe the task of each element of the scheme and how it helps build the previous plot:

1. Data : the dataset that we want to visualize

Figure 2:

```
# iris dataset
data(iris)
```

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

2. Aestethic Mappings : describes how variables in the data are mapped to aestethic attributes that you can perceive

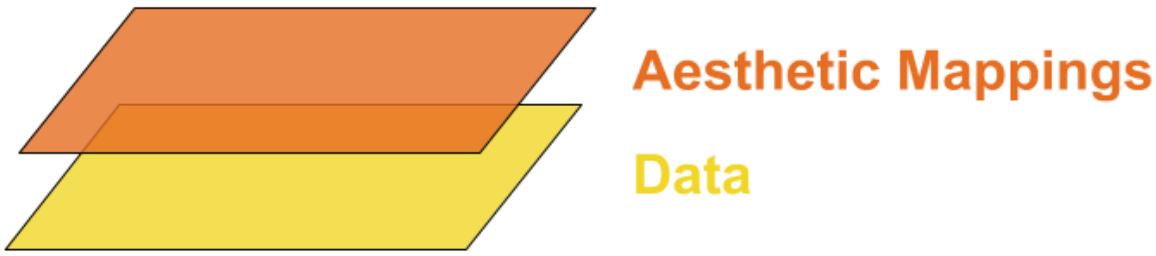


Figure 3:

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
y	x			

```
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length))
```

3. Layers : are made up by geometric elements and statistical transformations. In details, geometric objects (`geoms`) represent what we actually see on the plot: points, lines, polygons, etc. Statistical transformations (`stats`) summarise data in many useful ways. For example, binning and counting observations to create an histogram, or summarising a 2d relationship with a linear model.

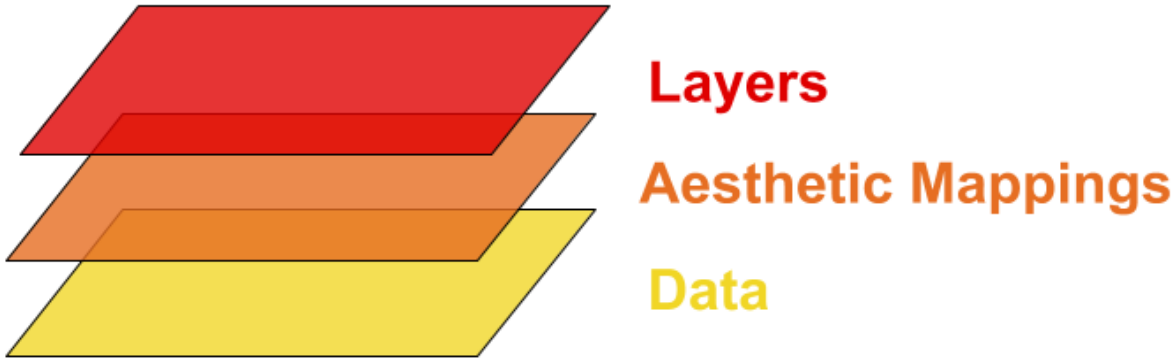
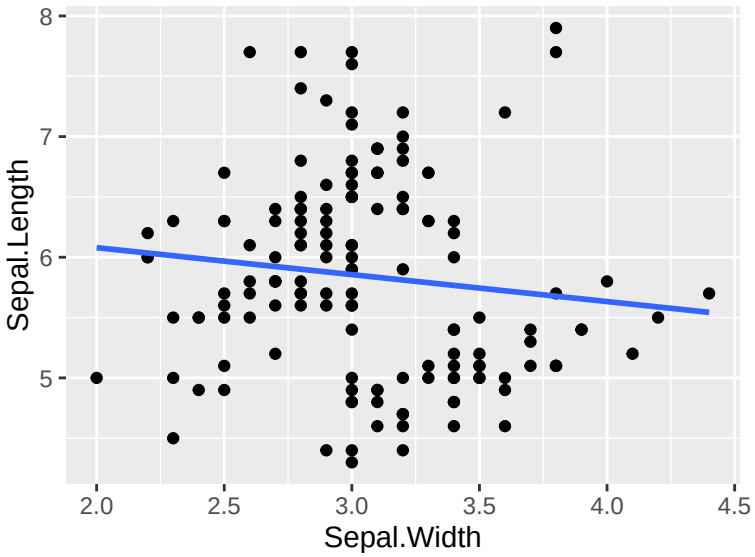


Figure 4:

```
# Scatterplot of the relationship between sepal length and sepal width with regression line
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point() + # layer 1
  stat_smooth(method = "lm", se = FALSE) # layer 2
```



4. Scales : map values in the data space to values in an aesthetic space, whether it be colour, or size or shape. Scales draw a legend on axes, which provide an inverse mapping to make it possible to read the original data values from the graph. Scales are closely related to aestethics mapped.

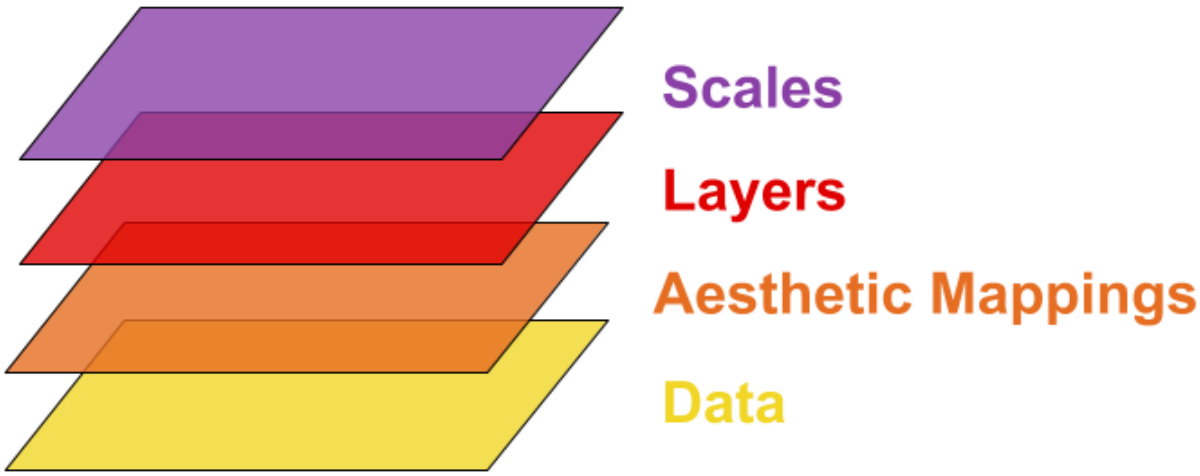
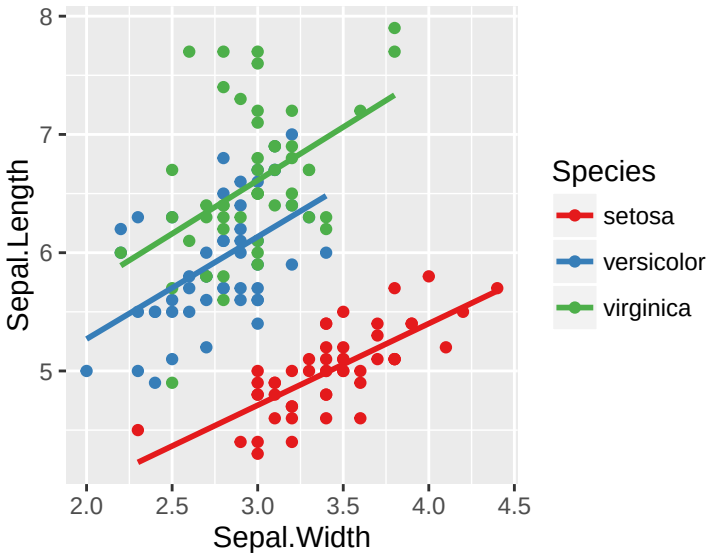


Figure 5:

```
# Map Species to colour in aes() and change the default colours of colour scale
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, colour = Species)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  scale_colour_brewer(palette="Set1")
```



5. Coordinates or coordinate system (`coord`) describe how data coordinates are mapped to the plane of the graphic. They also provides axes and gridlines to make it possible to read the graph. We normally use Cartesian coordinate system, but a number of others are available.

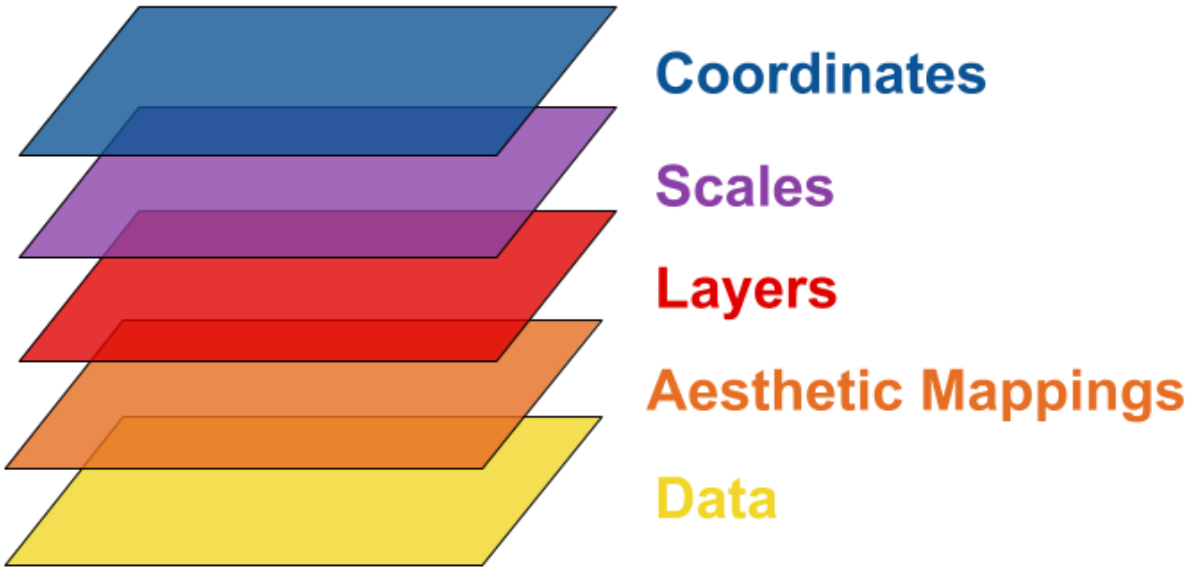
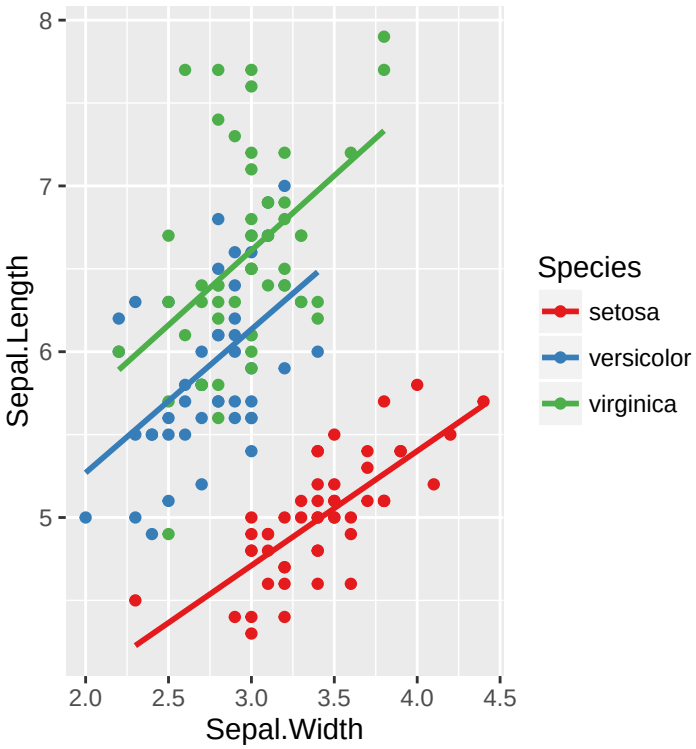


Figure 6:

```
# Change coordinate system
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, colour = Species)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  scale_colour_brewer(palette="Set1") +
  coord_equal()
```



6. Facets : describes how to break up the data into subset and how to display those subsets as small multiples.

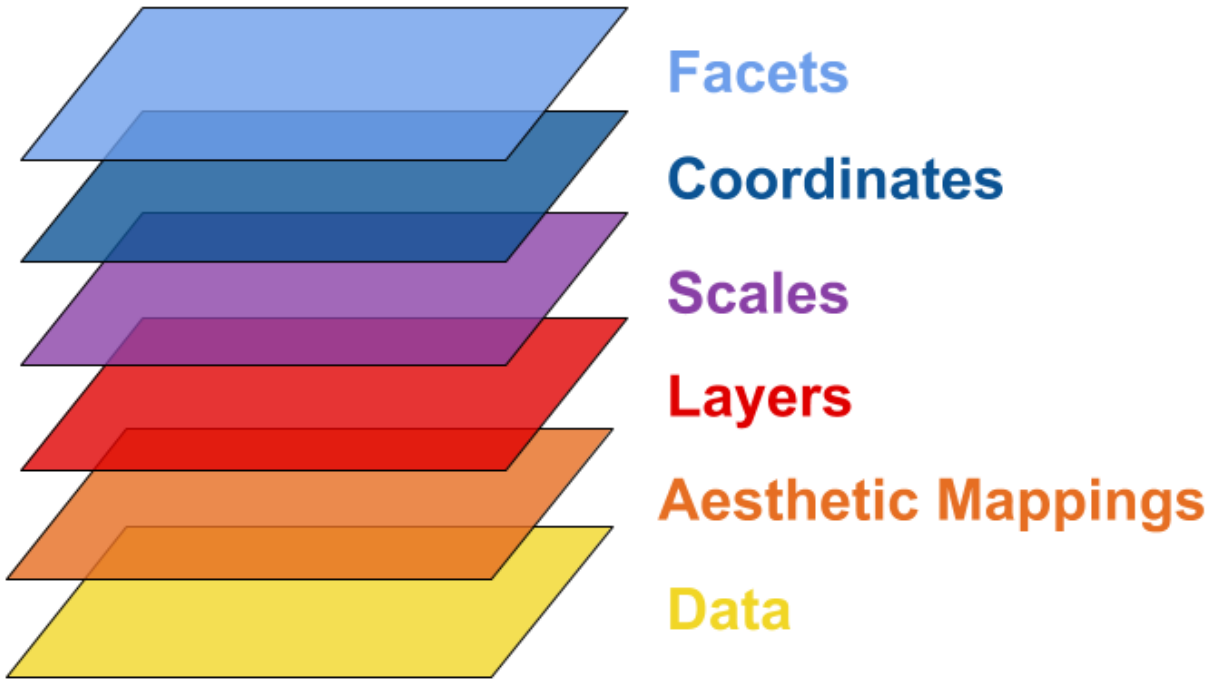
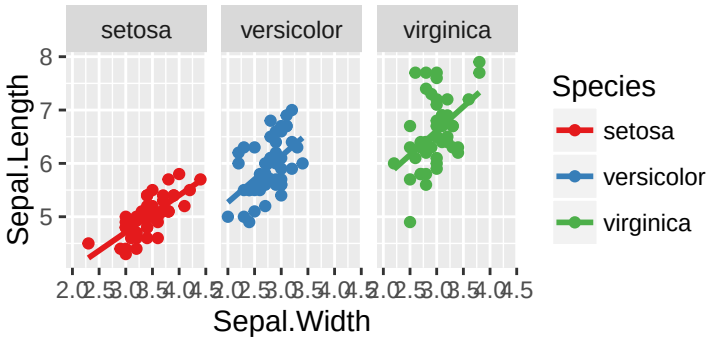


Figure 7:

```r
# Generate a plot for each iris species
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, colour = Species)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
  scale_colour_brewer(palette="Set1") +
  coord_equal() +
  facet_grid(. ~ Species)
```



7. Themes : controls all non-data elements of the plot, like the font size, background colour, etc.

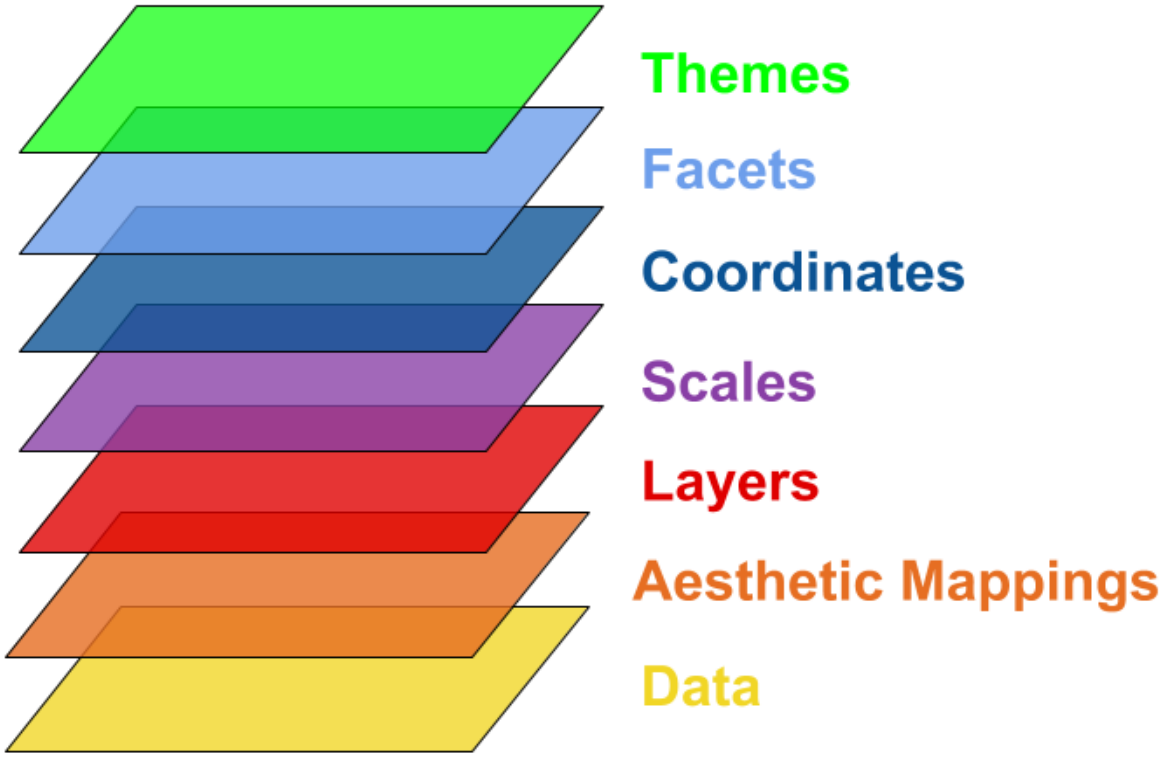
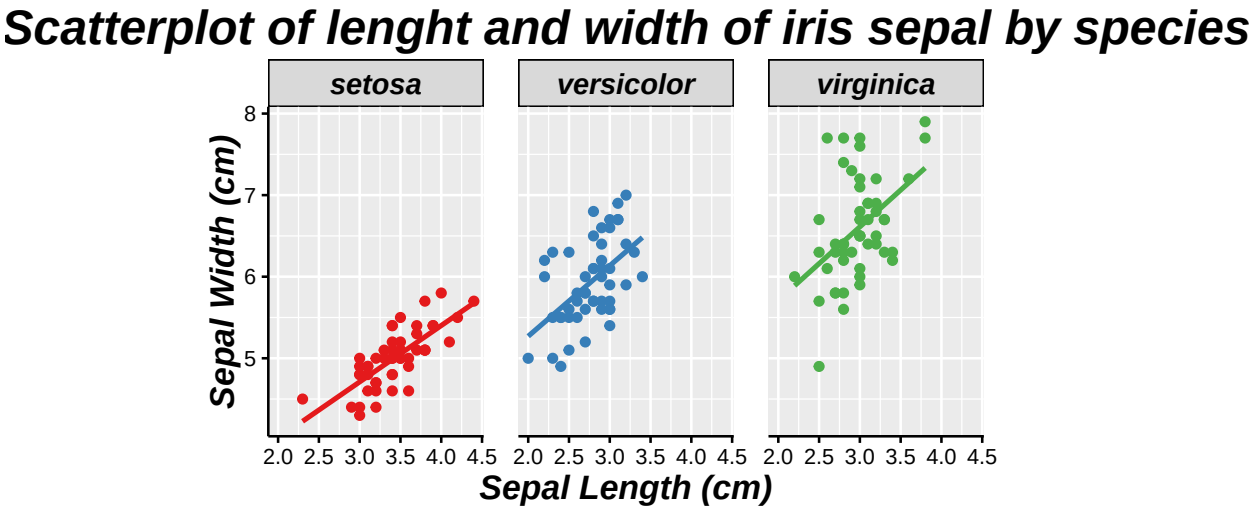


Figure 8:

```r
# Customize the appearance of the plot
ggplot(iris, aes(x = Sepal.Width, y = Sepal.Length, colour = Species)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE) +
```

```
  scale_colour_brewer(palette="Set1") +
  coord_equal() +
  facet_grid(. ~ Species) +
  ggtitle("Scatterplot of lenght and width of iris sepal by species") +
  xlab("Sepal Length (cm)") +
  ylab("Sepal Width (cm)") +
  theme(plot.background = element_blank(),
    axis.text = element_text(colour = "black"),
    axis.ticks = element_line(colour = "black"),
    axis.line.x = element_line(colour = "black"),
    axis.line.y = element_line(colour = "black"),
    axis.title = element_text(colour = "black", size = 14, face = "bold.italic"),
    strip.background = element_rect(colour = "black"),
    strip.text = element_text(colour = "black", face = "bold.italic", size = 12),
    plot.title = element_text(colour = "black", size = 20, face = "bold.italic"),
    panel.margin = unit(1, "lines"),
    legend.position="none"
)
```



1.2 Manual Contents

This manual is divided in three sections.

The first section, *Base ggplot2*, explains how to build the most important kinds of plot. It deepens the use and the functionalities of the three key components of `ggplot2` grammar (data, aesthetic mapping and layers). It introduces also some basic concepts about the other components.

The second section, *Advanced ggplot2* explains how dealing with the most common questions about plot customization and deepens the remaining components of `ggplot2` grammar (scales, coordinates, facets and themes).

The third section, *Other Topics*, talks about the `ggplot2` community and the packages built to support `ggplot2`. It shows also how to prepare plots for presentations, and how to save them to files.

1.3 Datasets

This manual is full of examples, for which the following datasets have been used:

- `bands`: provides data about process delays known as cylinder banding in rotogravure printing
- `istat`: provides the measures of weight, height, gender and geographical area ("Nord", "Centro", "Sud" and "Isole") from 1806 Italian people
- `italy`: provides information about longitude, latitude and region of the most important italian cities
- `lung`: provides information about survival in patients with advanced lung cancer from the North Central Cancer Treatment Group
- `brainbod`: provides information about the weight of body and brain of different species of animals
- `bottlecap`: provides information about of the mean diameter of the caps produced by a forging machine
- `ChickWeight`: provides weight versus age of chicks on different diets
- `ToothGrowth`: provides the effect of Vitamin C on Tooth Growth in Guinea Pigs

`bands`, `istat`, `italy`, `lung` `brainbod` and `bottlecap` are included in `qdata` package, whereas `ChickWeight` and `ToothGrowth` are included in `datasets` package, that comes with base R.

For a complete description of data, refer to the help (`?<dataset name>`).

1.4 Bibliography and References

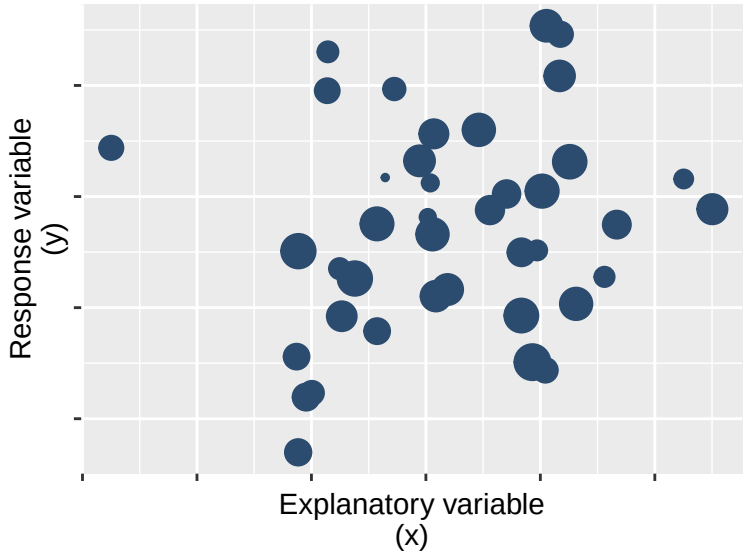
To take a deeper look about `ggplot2` you can refer to:

- Chang W., *R Graphics Cookbook*, O'Reilly, 2012 that provides a practical approach to `ggplot` and other R plots;
- Wickham H., *ggplot2: Elegant Graphics for Data Analysis*, Springer, 2009 that provides technical details about Grammar of Graphics implemented in `ggplot2`.

2 Creating a Scatter Plot

```
require(dplyr)
require(ggplot2)
require(qdata)
data(bands)
```

Scatter plots are used to display the relationship between two continuous variables. Axes represent a variable each, while each point represents an observation. This plot is often the first way to describe data when you look at it.



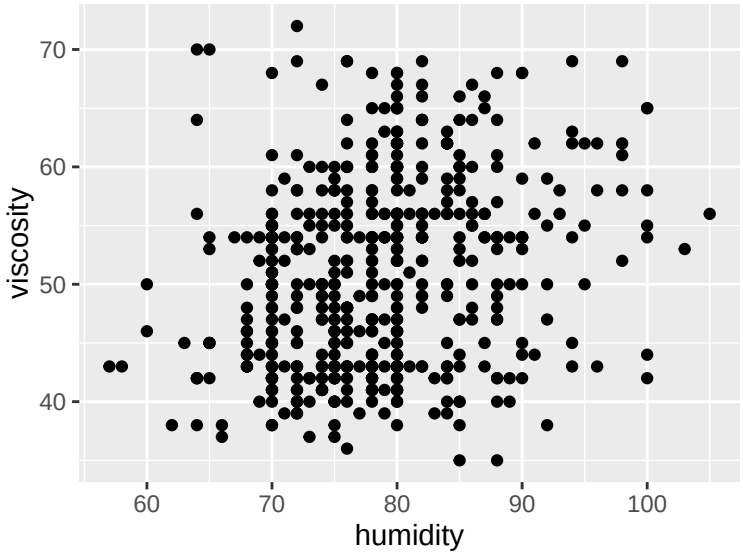
This chapter presents how to build scatter plots and introduces some basic concept of `ggplot2` grammar: aesthetic mappings and layers.

2.1 The first scatter plot

Suppose you are interested in the relationship between the humidity and the viscosity in the `bands` dataset.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point()
```

```
## Warning: Removed 6 rows containing missing values (geom_point).
```



The syntax is quite simple. The function `ggplot()` initializes the plot with the following parameters:

- *data* refers to the dataframe to use for the plot, in this case is the `bands` data frame
- *mapping* refers to the list of aesthetic mappings (visual properties) to use for plot, passed as arguments of the `aes()` function. In this case you *map* the data to *aesthetics* of plot like follow: the x-axis displays the `humidity` variable and the y-axis displays the `viscosity` variable.

Notes:

- the variable names must be unquoted
- the function is named `ggplot()` while the package is named `ggplot2`

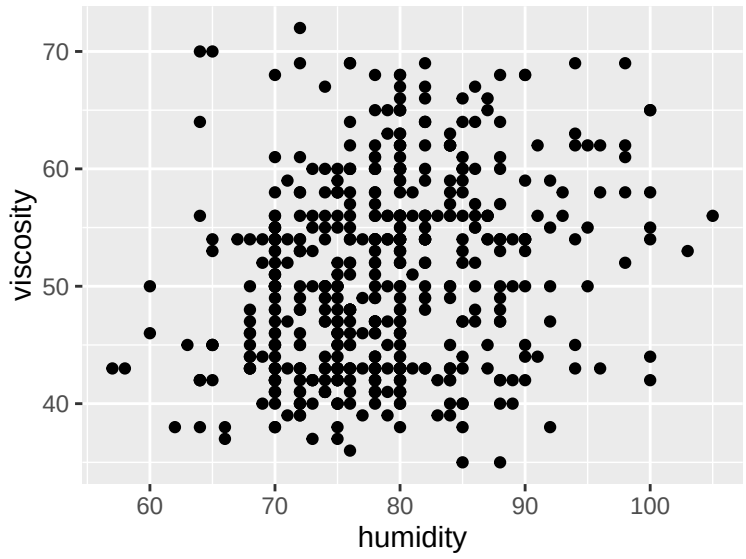
The `ggplot()` function does not return anything. You have to add to `ggplot()`, which geometric object (*geom*) you want to add. A scatter plot is made by points, and so you will use the `geom_point()` function, without any argument.

The meaning of technical terms will be cleared in the following examples.

The function returns a warning message: in six observations at least one between `humidity` and `viscosity` has missing values and points cannot be drawn. In the following examples, the warning message will not be shown.

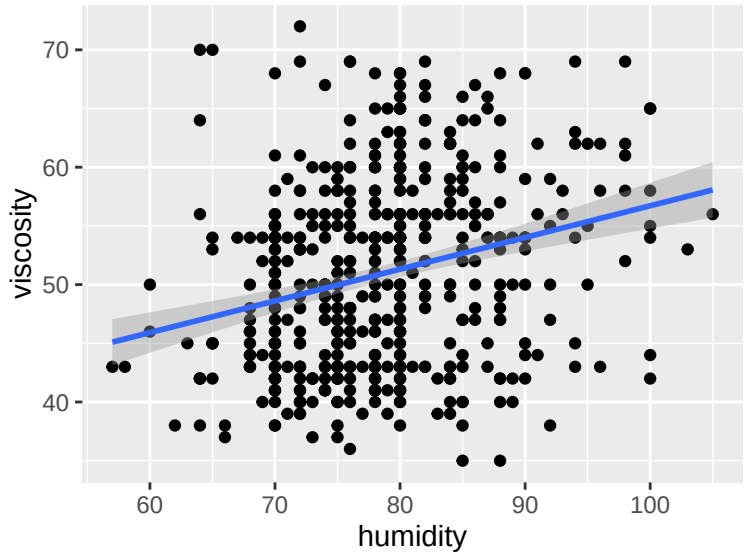
Alternatively, the plot can be initialized with `ggplot()` without any argument and the same arguments have to be passed to `geom_point()`.

```
ggplot() +
  geom_point(data=bands, mapping=aes(x=humidity, y=viscosity))
```



To understand the difference between the two examples above, you can add a new `geom` to the plot. From the previous plot it appears a (weak) positive correlation between the variables. A regression line can be added to the plot. The `geom_smooth()` with `method="lm"` adds a regression line based on the linear model.

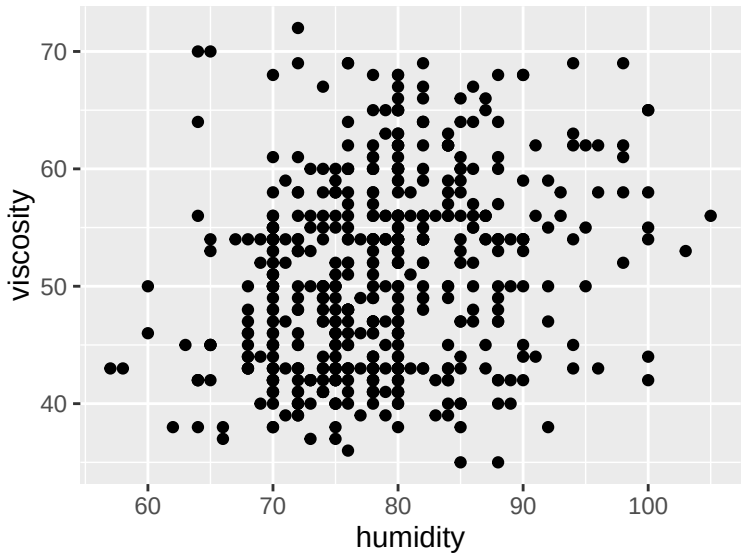
```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point() +
  geom_smooth(method="lm")
```



The plot now displays a regression line with its confidence interval.

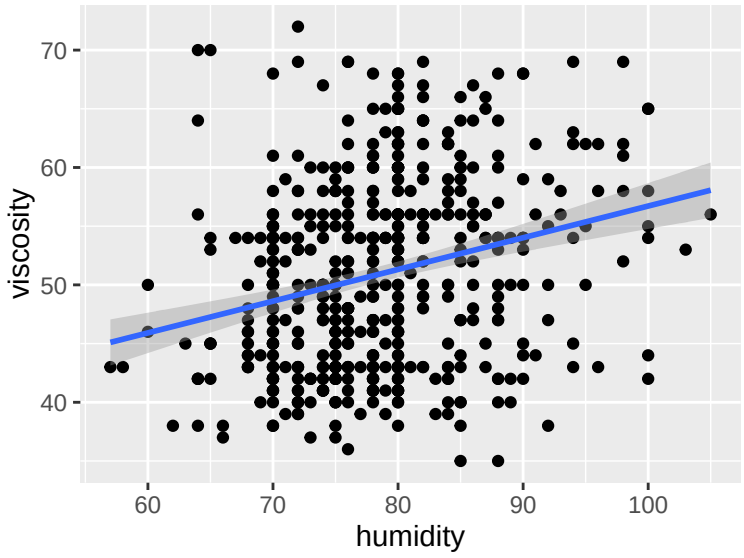
You can try to add a smoother to the plot, but starting from the code with `ggplot()` without arguments.

```
ggplot() +
  geom_point(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_smooth(method="lm")
```



There is no line. Why? The reason is quite simple: `geom_smooth()` do not know which data and which variable should be used for the linear model. You can solve the issue like follow.

```
ggplot() +
  geom_point(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_smooth(data=bands, mapping=aes(x=humidity, y=viscosity), method="lm")
```



To sum up, data and aesthetics specified in `ggplot()` are used by default by all layers. Layers are like tracing papers, you can overlay them. Each layer contains a geometry. If a layer has its own data and/or aesthetics, that layer will ignore the default values.

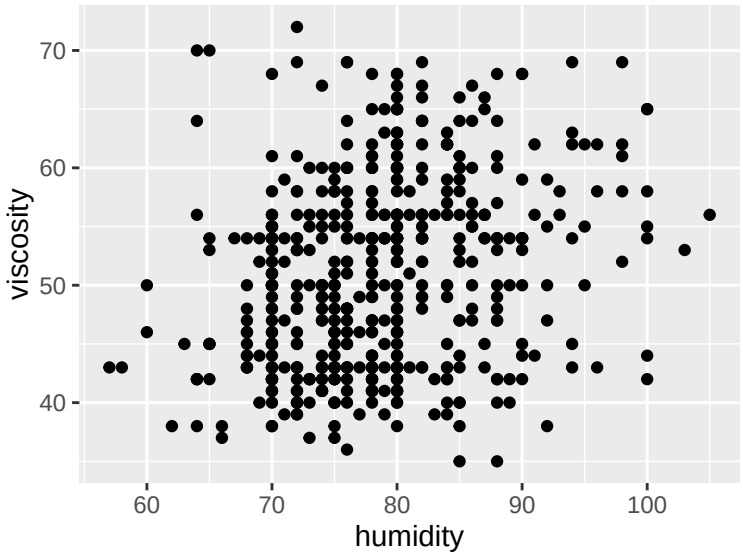
2.2 Assigning ggplots to a variable

From a formal point of view, a ggplot is an R object like anything else; so you can assign it to a variable.

```
gp1 <- ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point()
```

And now you can recall the plot named `gp1`.

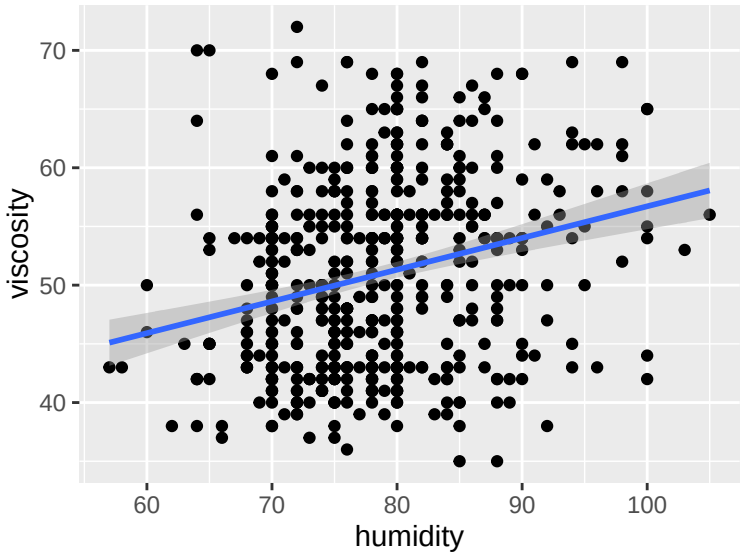
```
gp1
```



Note: you can print the plot by recalling it; indeed the run of `gp1` does the same work of `print(gp1)` command.

Once you assigned the scatter plot to `gp1`, you can add to this plot a smoother by doing:

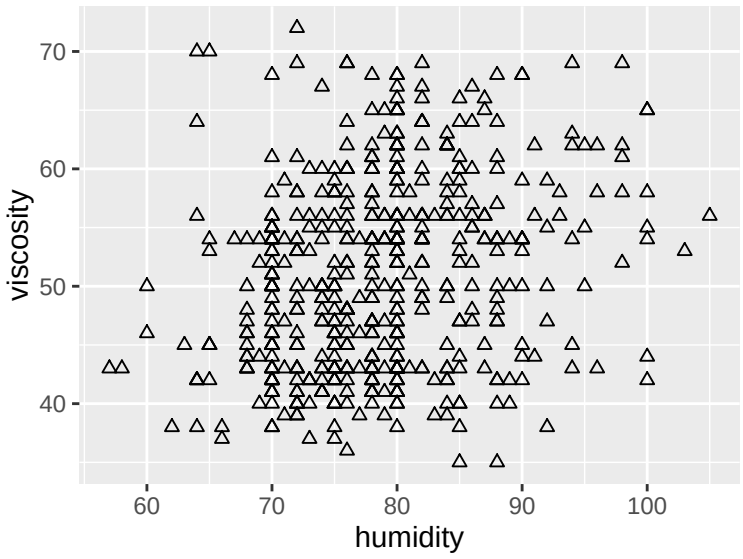
```
gp2 <- gp1 +
  geom_smooth(method="lm")
gp2
```



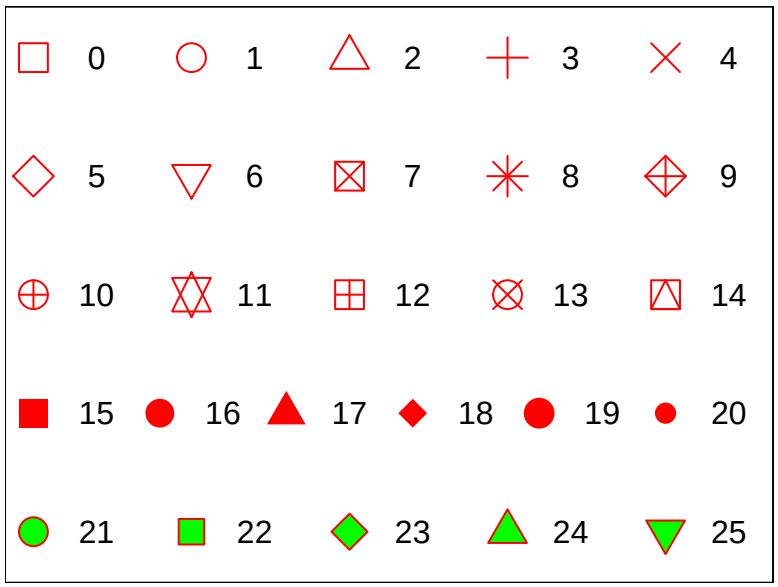
2.3 Changing the shape and size of points

At this point, you probably are interested to change the aspect of points like shape or size: it suffices to set the `shape` or the `size` as a parameter of `geom_point()`.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(shape=2)
```

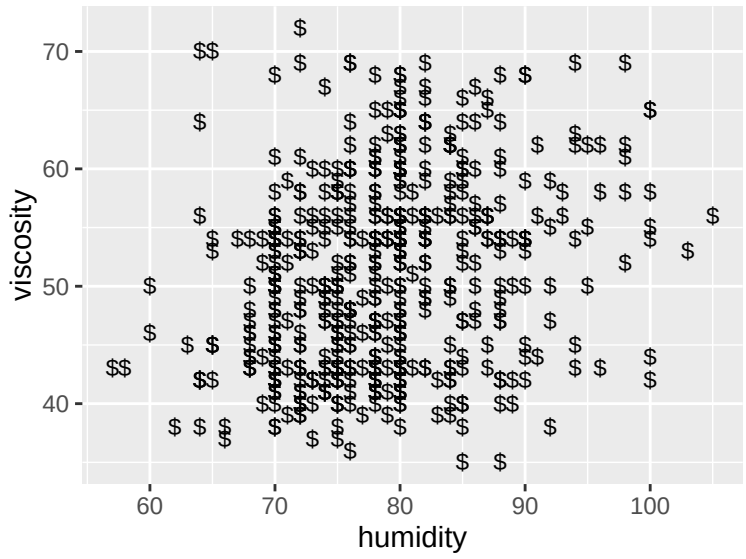


The following shapes are available in R graphics. Point shapes from 0 to 14 have just an outline, shapes from 15 to 20 are solid and shapes from 21 to 25 have both an outline and a fill. Default shape for `ggplot2` graphics is 16.



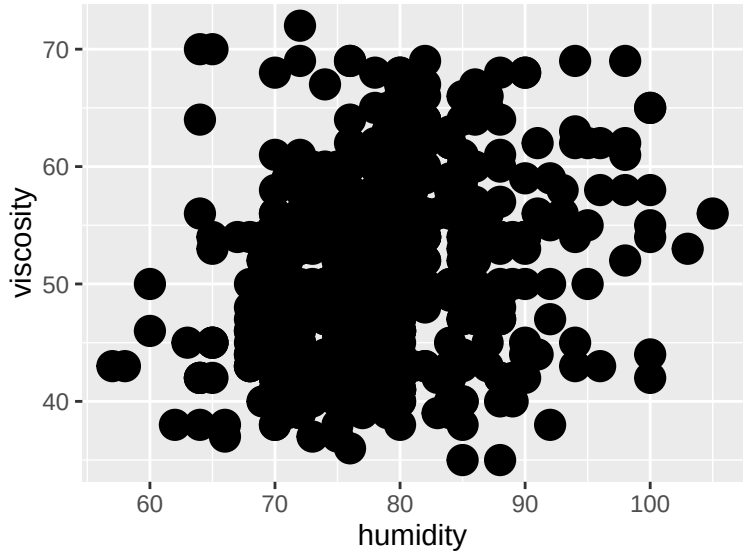
Shape can also be a (single) character string.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(shape="$", size=3)
```

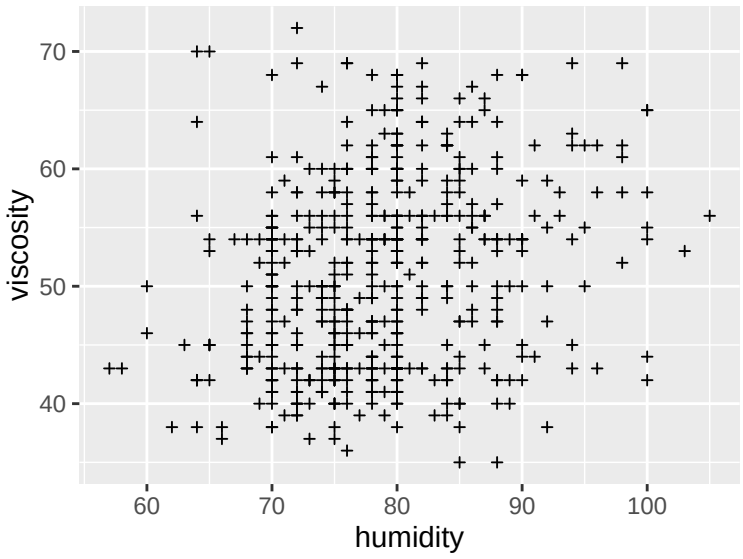


The same way works for `size` too.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(size=5)
```



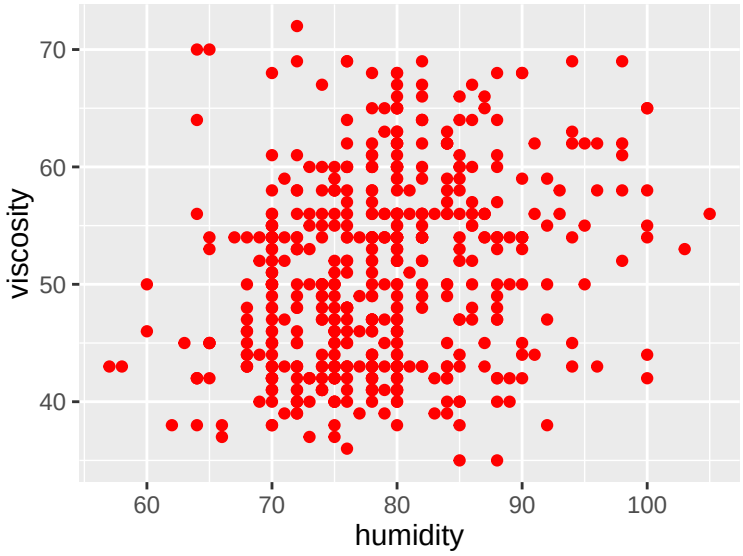

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(shape=3, size=1)
```



2.4 Changing the colour of points

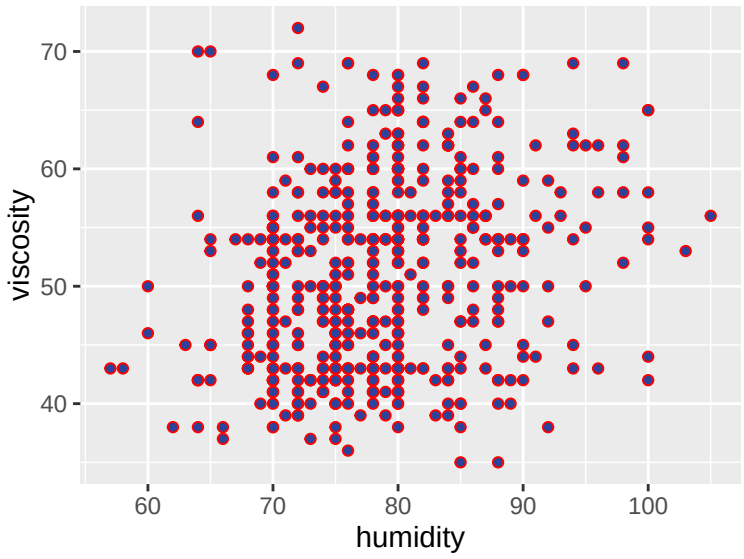
Colour can be set in a similar way than `shape` and `size`: setting `colour` as a `geom_point()` parameters. Both UK and US spellings (`color`) can be used.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(colour="red")
```



Shapes from 21 to 25 allow two colours. In these cases, `colour` set the outline and `fill` set the internal colour.

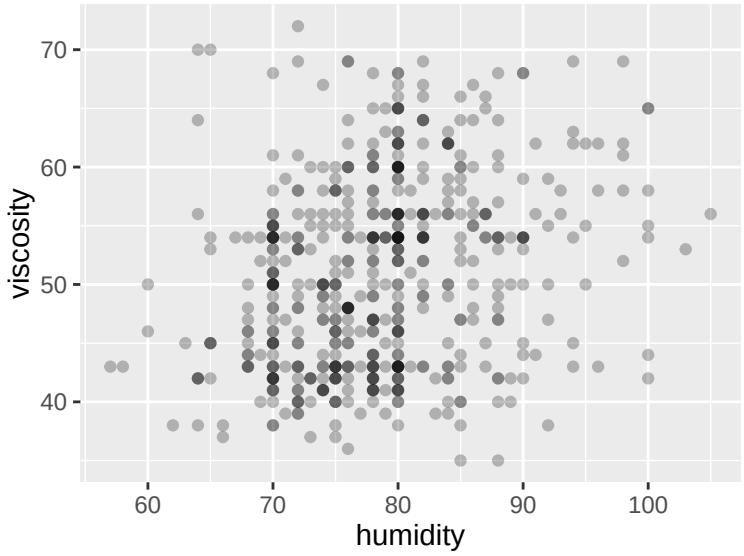
```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(shape=21, colour="red", fill="#2b439a")
```



When you need to pass a colour to R, you can use a string with the colour name. There are 657 colour names in R; just type `colours()` to view them. Alternatively, you can pass a string with the hexadecimal code (e.g. "#2b439a") or you can use `rgb()`, `hsv()` or `hcl()` if you are familiar with these colour models.

Data used in these examples has 540 observations but the plot seems have less points. This is because many points overlap. Transparencies are useful in this cases. The `alpha` aesthetic set the transparency level: legal `alpha` values are any numbers from 0 (transparent) to 1 (opaque).

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(alpha=0.25)
```



Since `alpha=0.25` (and 0.25 is 1/4) a point will be drawn as solid when four points overlap.

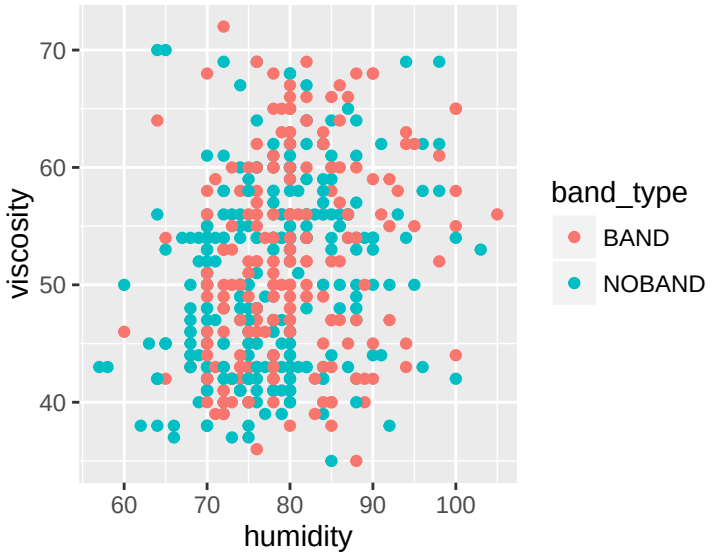
If you are familiar with base graphics in R, `shape` substitutes `pch`, `size` substitutes `cex`, `colour` substitutes `col` and `fill` substitutes `bg`. The base syntax can be used also in `ggplot2` but it is strongly suggested to migrate to the new and more intuitive syntax.

2.5 Mapping a third variable to scatter plots

Scatter plots were born to visualize the relationship between two variables: one mapped to the x-axis and one mapped to the y-axis. Sometimes, a third variable should be visualized. In these case, you can map a third variable to other aesthetics: size, shape or colour.

Suppose you're interested in the relationship between humidity and viscosity accordingly the presence or absence of `band_type`. `band_type` is a discrete variable so to perform this task, you can map it to `colour`.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, colour=band_type)) +
  geom_point()
```

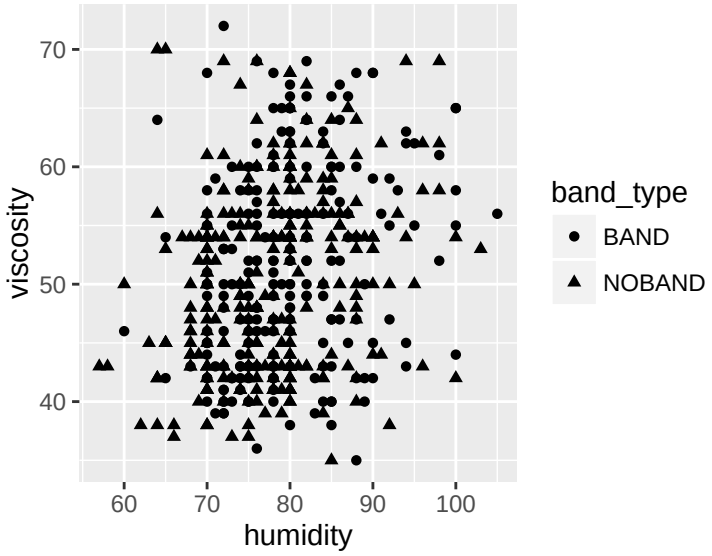


Note that mapping occurs within `aes()`, while setting occurs outside of `aes()`.

The plot shows the same points that previous ones with different colours and a legend will be added.

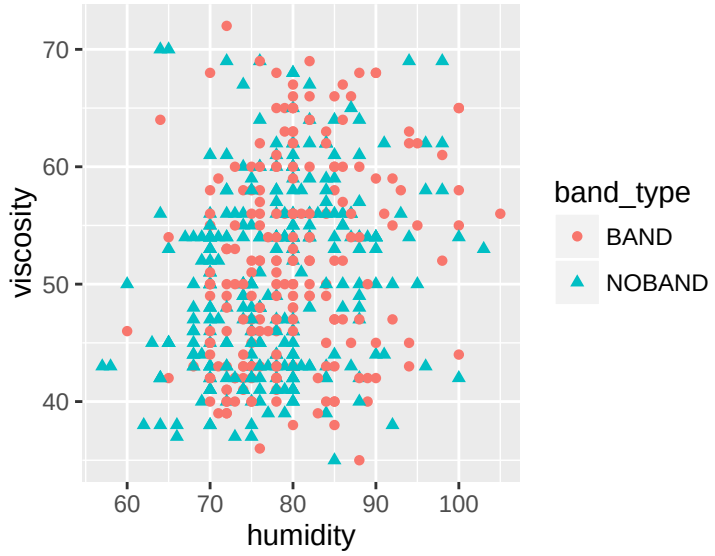
Alternatively, you can map `band_type` to another aesthetic, like `shape`.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, shape=band_type)) +
  geom_point()
```



Since different shapes are more difficult to read when you have many points, this solution provide an alternative when you are printing in black and white, without colours. You can improve your result using both aesthetics together.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, colour=band_type, shape=band_type)) +
  geom_point()
```

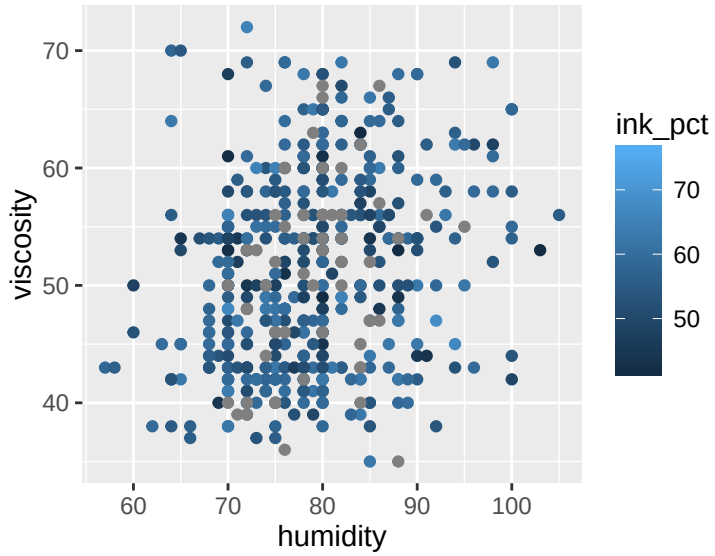


Mapping a discrete variable to `size` aestethic is not advised.

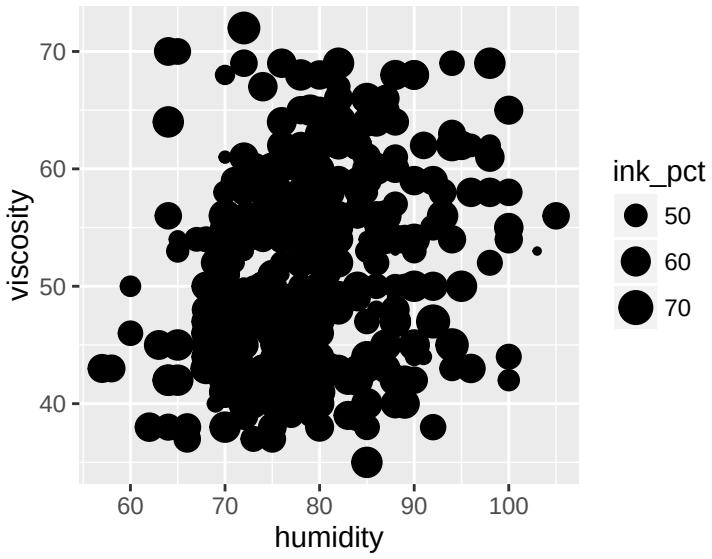
–>

If you're interested in a continuous variable as third variable you can `map` it to `colour` or to `size`. It makes no sense map a continuous value to `shape`. Let us consider `ink_pct` as third variable:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, colour=ink_pct)) +
  geom_point()
```



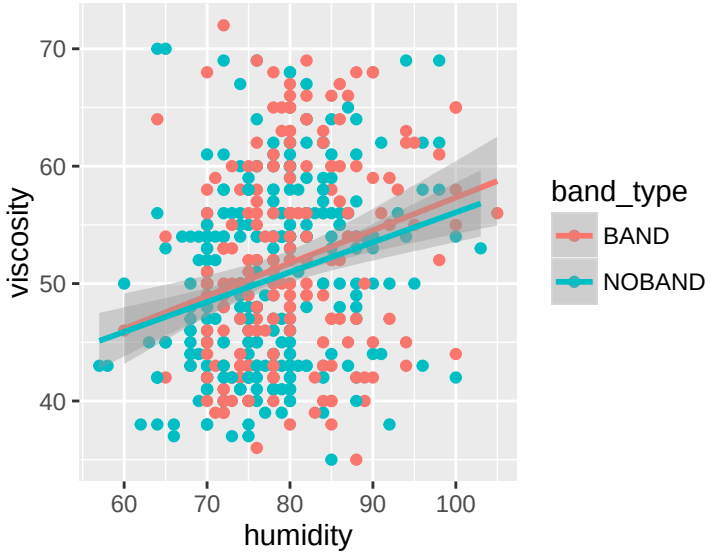

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, size=ink_pct)) +
  geom_point()
```



It is more difficult perceiving small differences in size and colour, so variable mapped to these aesthetic attributes will be interpreted with a much lower accuracy than those mapped to spatial coordinates (`x` and `y`).

The following code will produce a scatter plot of `humidity` versus `viscosity` with `band_type` mapped to `colour`.

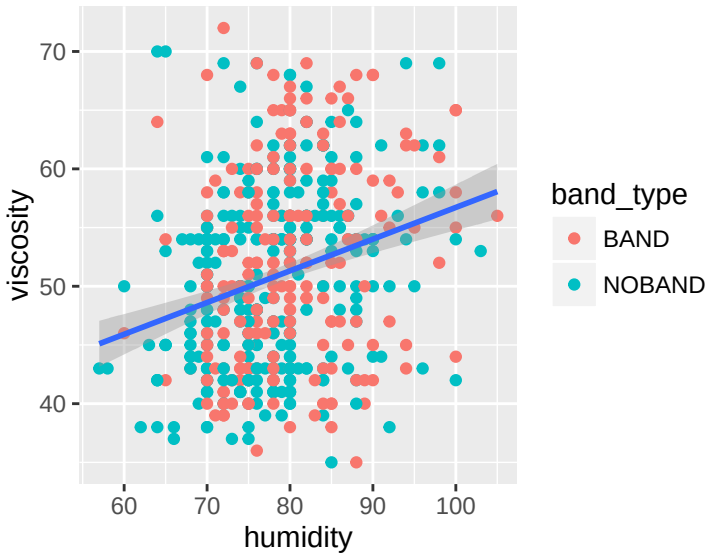
```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, colour=band_type)) +
  geom_point() + geom_smooth(method="lm")
```



The scatter plot has two regression lines. This is the expected results, as `band_type` is mapped to `colour` in the `ggplot()` function and its arguments are used not only by `geom_point()` but also by `geom_smooth()`.

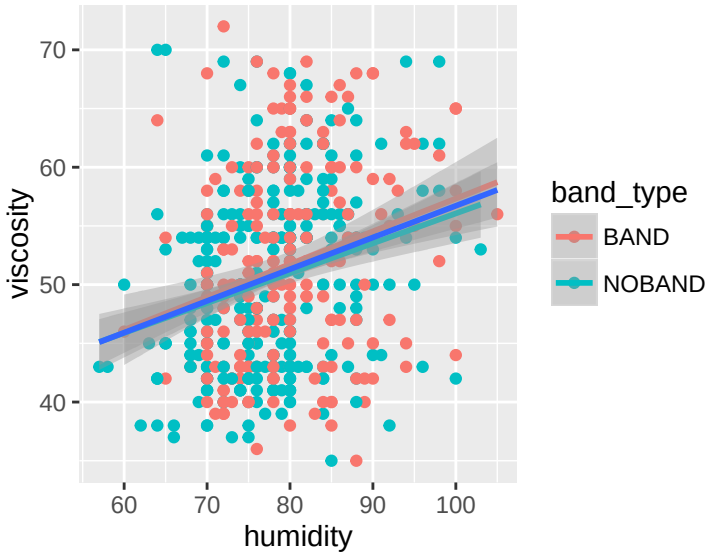
To produce a scatter plot with a single regression line the `colour` aesthetic must be mapped only to `geom_point()`.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(mapping=aes(colour=band_type)) +
  geom_smooth(method="lm")
```



Finally, this is the plot if you are interested to three regression line, one for all values and one for each level of `band_type`.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(mapping=aes(colour=band_type)) +
  geom_smooth(mapping=aes(colour=band_type), method="lm") +
  geom_smooth(method="lm")
```

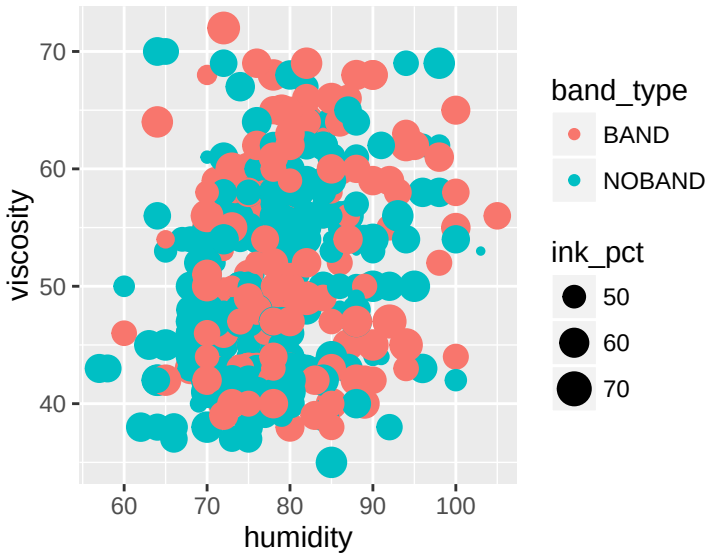


2.6 Mapping four variables to scatter plots

Although the interpretation may be difficult, different variable can map to different aesthetics at the same time. From a theoretical point of view you can map as many variable as the number of aesthetics, but it is not suggested map more than four variable.

This is the result when you are interested to the ink percentage and band type, at the same time.

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, size=ink_pct, colour=band_type)) +
  geom_point()
```

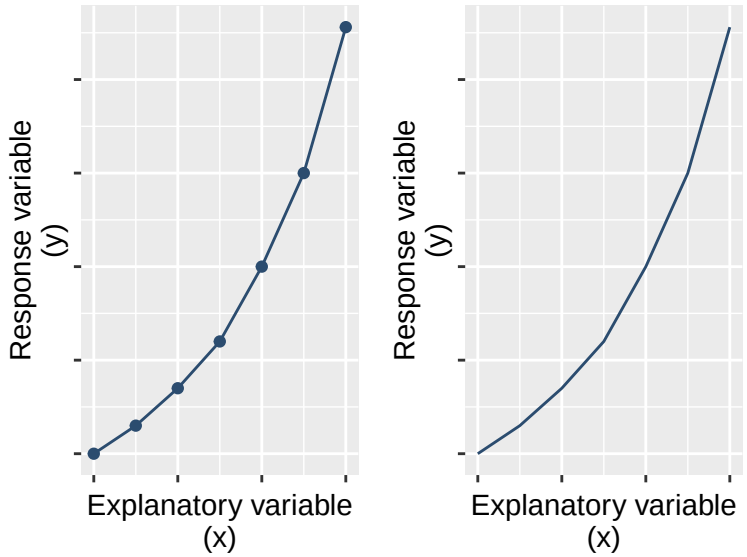


When a variable is mapped to `size` it is not suggested to map another variable to `shape`. This is because it is difficult to compare the sizes of different shapes.

3 Creating a Line Plot

```
require(dplyr)
require(ggplot2)
```

Line plots are used to display how one continuous variable, on the y-axis, changes in relation to another continuous variable, on the x-axis. It is similar to a scatter plot, except that points are ordered in the x-axis and connected by a segment. Points can also be missing.

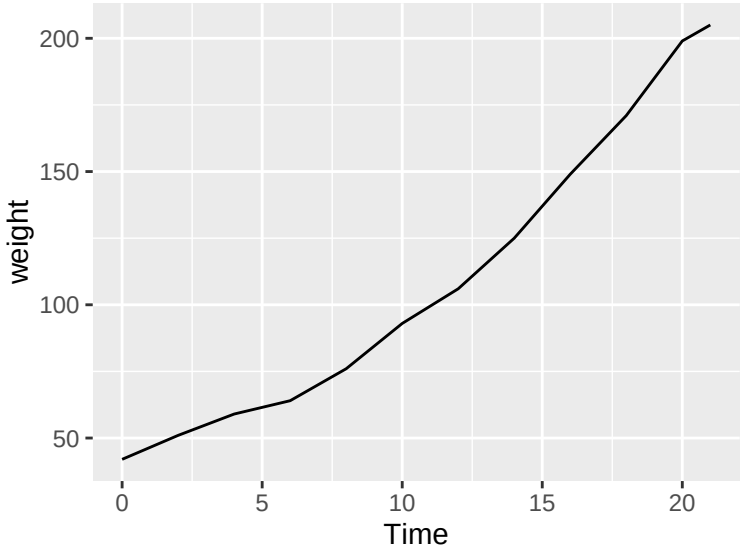


This chapter presents how to build line plots and introduces other basic concepts of `ggplot2` graphics: `group` aesthetic and manual scales.

3.1 The first line plot

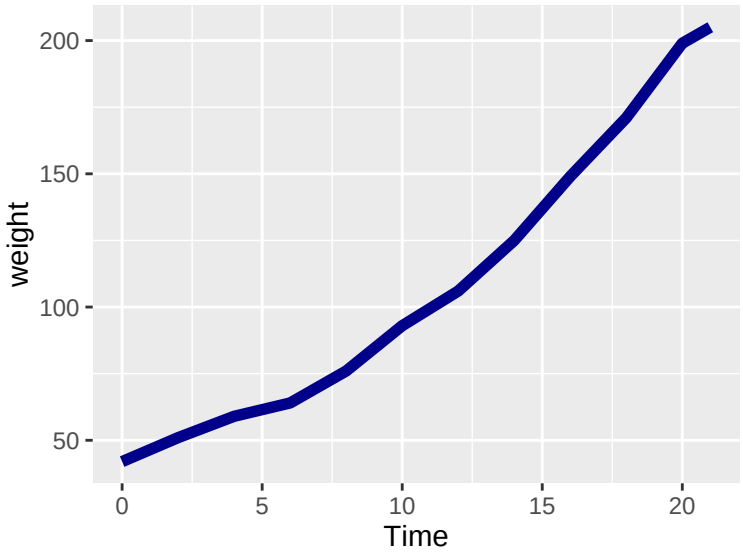
`ChickWeight` data contains the body weight of 50 chicks over time. Suppose you are interested to the growth of the first chick.

```
ggplot(data=(ChickWeight %>% filter(Chick==1)), mapping=aes(x=Time, y=weight)) +
  geom_line()
```



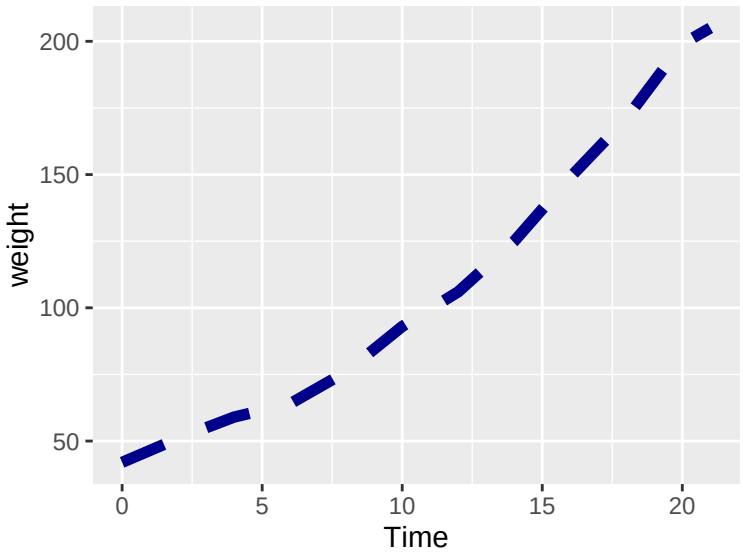
You can change the appearance of the line setting `geom_line()` aestethic arguments. For example, you can set dark blue color for line and change the size in this way:

```
ggplot(data=(ChickWeight %>% filter(Chick==1)), mapping=aes(x=Time, y=weight)) +
  geom_line(colour="darkblue", size = 2)
```

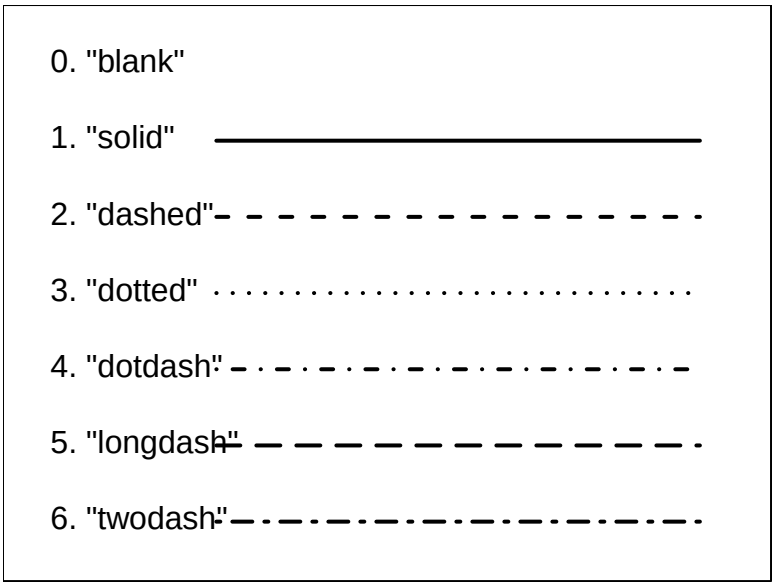


You can also choose the style of the line setting `linetype` argument. If you are experienced with R basic graphic, `linetype` has the same meaning that `lty`.

```
ggplot(data=(ChickWeight %>% filter(Chick==1)), mapping=aes(x=Time, y=weight)) +
  geom_line(colour="darkblue", size = 2, linetype=2)
```



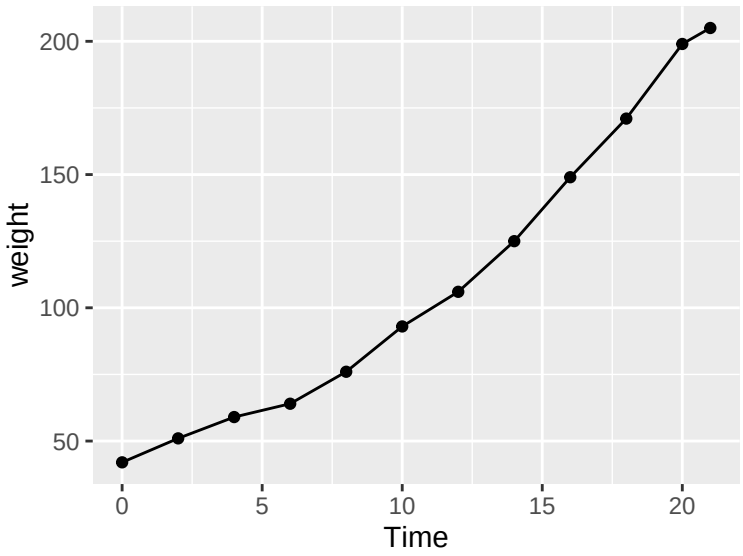
The `linetype` can be a number (0-6) or a description (like `"solid"` or `"dashed"`). Available line types are:



Available aestethics arguments for `geom_line()` are `alpha`, `colour`, `size` and `linetype`. The first three arguments have the same (intuitive) meaning already seen for `geom_point()`.

You can also add points to the plot with `geom_point()`:

```
ggplot(data=(ChickWeight %>% filter(Chick==1)), mapping=aes(x=Time, y=weight)) +
  geom_line() +
  geom_point()
```



3.2 Mapping variables to line plots

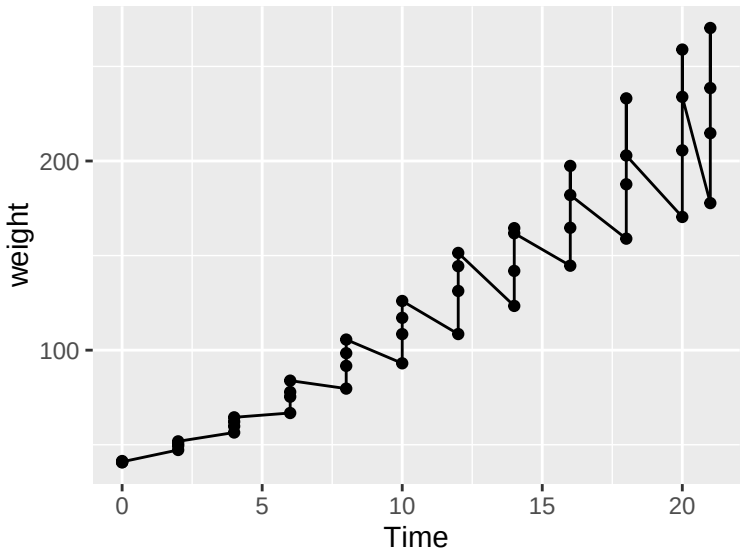
If you are interested in the average growth of chicks for the different diets, you must first summarize data.

```
ChickWeightMean <- ChickWeight %>%
  group_by(Time, Diet) %>%
  summarize(weight=mean(weight))
ChickWeightMean
```

```
## Source: local data frame [48 x 3]
## Groups: Time [?]
##
##     Time   Diet   weight
##    <dbl> <fctr>    <dbl>
## 1      0      1 41.40000
## 2      0      2 40.70000
## 3      0      3 40.80000
## 4      0      4 41.00000
## 5      2      1 47.25000
## 6      2      2 49.40000
## 7      2      3 50.40000
## 8      2      4 51.80000
## 9      4      1 56.47368
## 10     4      2 59.80000
## # ... with 38 more rows
```

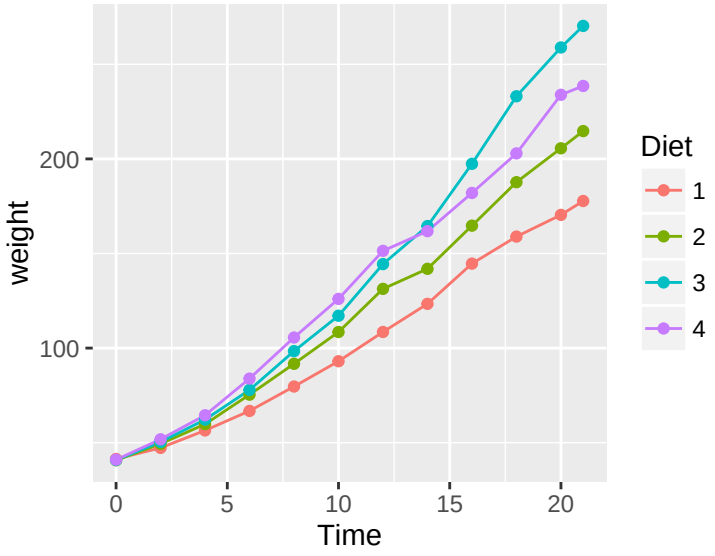
If you draw the same plot as above with new data, you will obtain a figure like this one.

```
ggplot(data=ChickWeightMean, mapping=aes(x=Time, y=weight)) +
  geom_line() +
  geom_point()
```



A jagged line appears when there are multiple data at each `x` and you tell `ggplot` to connect them. A plot like this one should sounds as a warning that something is wrong. In order to have a single point for each `x` you can summarize your data, or you can draw several lines, accordingly to a third variable, just adding a new aesthetic.

```
ggplot(data=ChickWeightMean, mapping=aes(x=Time, y=weight, colour=Diet)) +
  geom_line() +
  geom_point()
```



Sometimes, the `x` variable can be a categorical variable. This is the case when you have numerical values that are conceived as categorical ones. The `ToothGrowth` data set can be a good case in point. It provides the effect of Vitamin C on Tooth Growth in Guinea Pigs. Data can be summarized as follow to achieve the mean tooth length by supplier and dose:

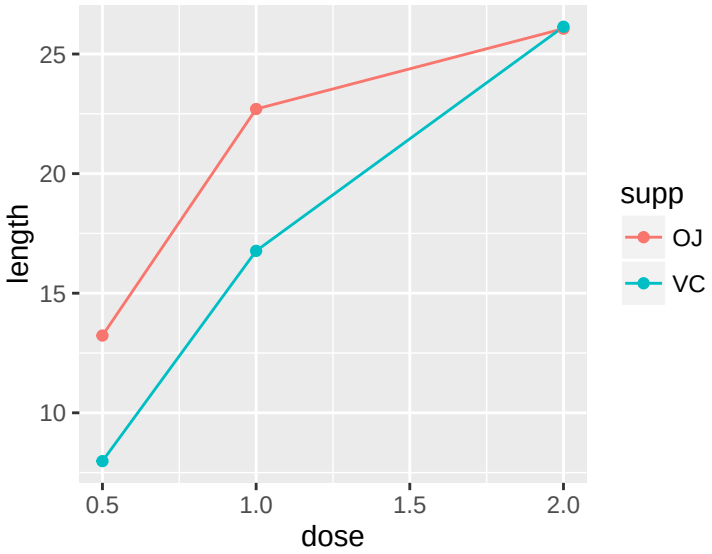
```
tg <- ToothGrowth %>%
  group_by(supp, dose) %>%
  summarize(length=mean(len))
tg
```

```
## Source: local data frame [6 x 3]
```

```
## Groups: supp [?]
##
##     supp  dose length
##   <fctr> <dbl>  <dbl>
## 1     OJ   0.5  13.23
## 2     OJ   1.0  22.70
## 3     OJ   2.0  26.06
## 4     VC   0.5   7.98
## 5     VC   1.0  16.77
## 6     VC   2.0  26.14
```

You can plot a line for each `supp` value:

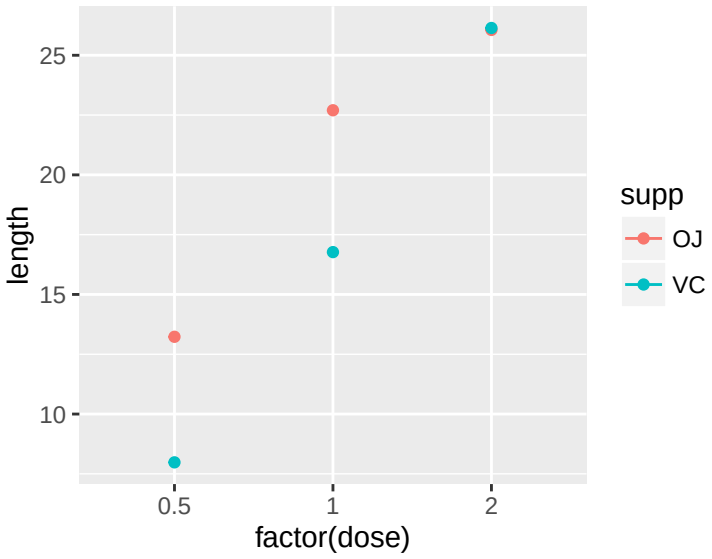
```
ggplot(data=tg, mapping=aes(x=dose, y=length, colour=supp)) +
  geom_line() +
  geom_point()
```



The plot is good but since `dose` level are fixed the space between 0.5-1.0 and 1.0-2.0 should be the same. You can fix this issue, transforming `dose` into a factor.

```
ggplot(data=tg, mapping=aes(x=factor(dose), y=length, colour=supp)) +
  geom_line() +
  geom_point()
```

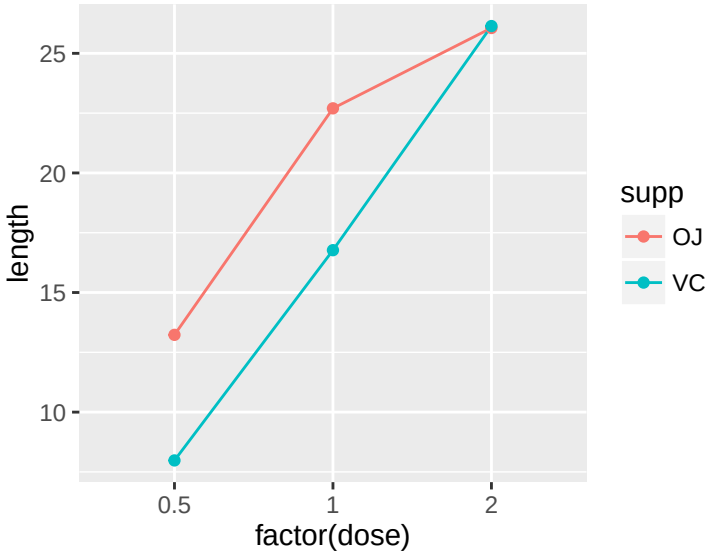
```
## geom_path: Each group consists of only one observation. Do you need to adjust the group aesthetic?
```



As you can see from the plot no lines are drawn and a warning message is returned. Probably, this is one of the most frequently problems with `geom_line()`. This appens because `geom_line()` tries to connect data points that belong to same group and the combinations of different levels of factor variables belong to different groups.

To resolve this problem, you have to manually specify the grouping, setting `group` argument:

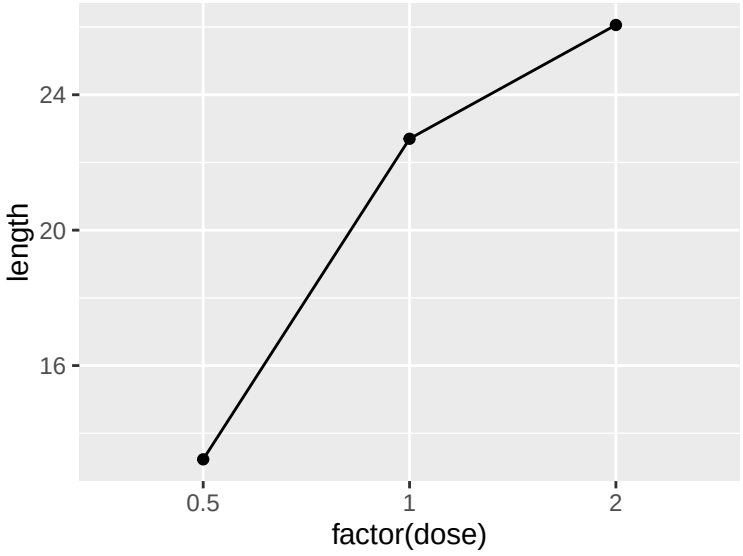
```
ggplot(data=tg, mapping=aes(x=factor(dose), y=length, colour=supp, group=supp)) +
  geom_line() +
  geom_point()
```



`group=supp` in `aes()` tells ggplot how to group data when it draws the lines.

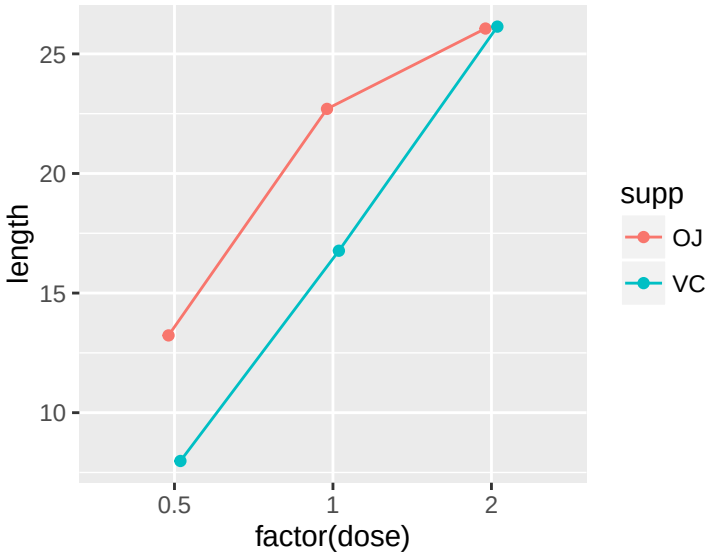
Otherwise, if you want a single line than connects all data points, you have to specify `group=1`.

```
# Consider only "OJ" supplier
ggplot(data=(tg%>%filter(supp=="OJ")), mapping=aes(x=factor(dose), y=length, group=1)) +
  geom_line() +
  geom_point()
```



When points overlap, you can *dodge* them. Dodging adjusts points left and right and must be applied to both lines and points, to avoid misalignment.

```
ggplot(data=tg, mapping=aes(x=factor(dose), y=length, colour=supp, group=supp)) +
  geom_line(position=position_dodge(0.15)) +
  geom_point(position=position_dodge(0.15))
```



`position` argument of both `geom_line()` and `geom_point()` is set equal to `position_dodge()`. `position_dodge()` is a `ggplot2` function that adjust position by dodging overlaps to the side. 0.15 is the `width` specified.

3.3 Changing the Appearance of Mapped Aesthetics

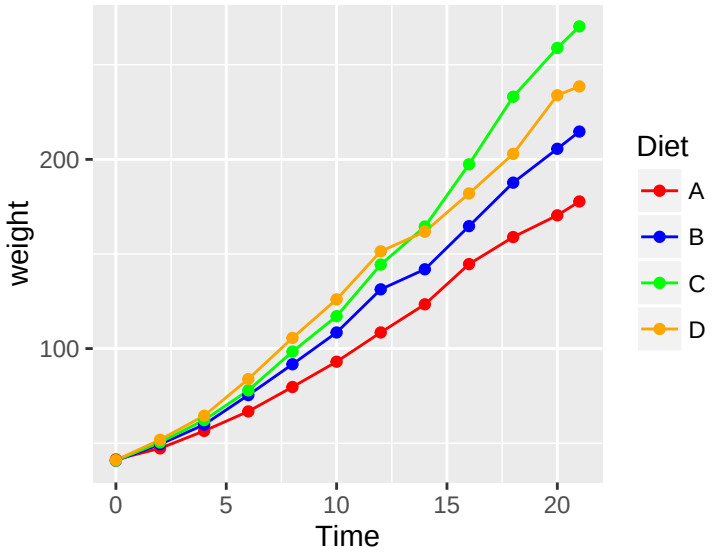
When you feed your chicks, you use the "A" fodder for diet 1, the "B" fodder for diet 2, the "C" for diet 3 and the "D" for diet 4.

```
ChickWeightMean <- ChickWeightMean %>%
  mutate(Diet = factor(Diet, levels=1:4, labels=c("A","B","C","D")))
```

"A" fodder is sold in red bags and farmers immediately identify this product with the red colour. In the same way, B is associated with the blue, C with the green and D with the orange. To improve the readability of the plot, lines for each diet should be of the colour that identify the fodder maker.

You cannot set the colour inside `ggplot()` or `geom_line()` using the `colour` argument, because this is admitted only when a variable is not *mapped* to the `colour` aesthetic. You can set manually the colour with the function `scale_colour_manual()`.

```
ggplot(data=ChickWeightMean, mapping=aes(x=Time, y=weight, colour=Diet)) +
  geom_line() +
  geom_point() +
  scale_color_manual(values=c("red", "blue", "green", "orange"))
```

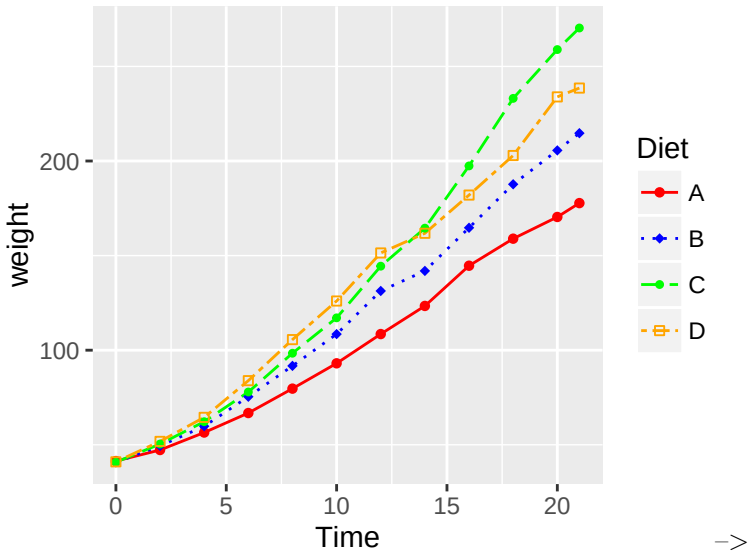


Colours in the `values` string vector must appear in the same order as they appear in the legend. You can pass a named vector to be sure of the correspondence between colour and data.

Mapped Aesthetics customization will be deepen in "Mapped Aesthetics Customization" chapter.

There are as many `scale_*_manual` function as aesthetics for which values can be set manually.

```
ggplot(data=ChickWeightMean, mapping=aes(x=Time, y=weight, colour=Diet)) +
  geom_line(mapping=aes(linetype=Diet)) +
  geom_point(mapping=aes(shape=Diet)) +
  scale_color_manual(values=c("D"="orange", "B"="blue", "A"="red", "C"="green")) +
  scale_shape_manual(values=c(16,18,20,22)) +
  scale_linetype_manual(values=c(1,3,5,6))
```

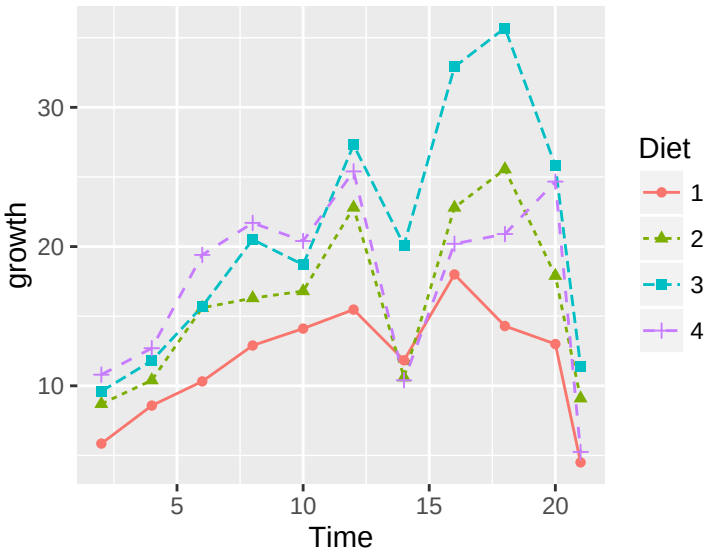


3.4 Adding horizontal and vertical lines

Line plots often contains horizontal or vertical lines.

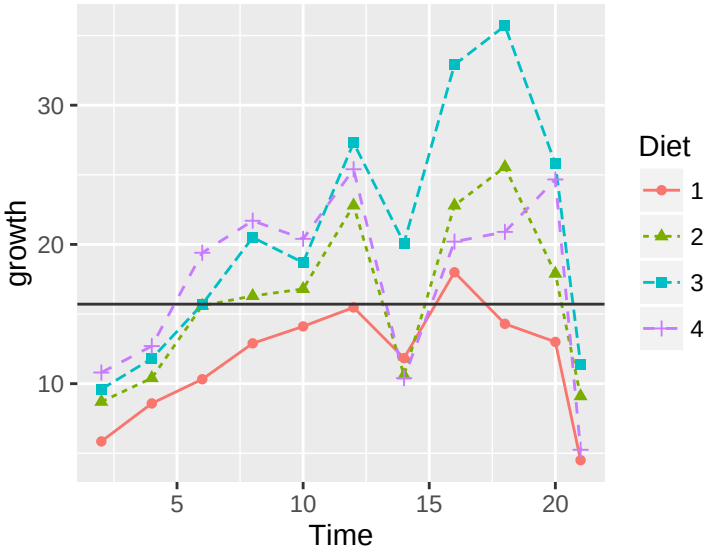
It may be interesting to analyse the average daily growth for each diet type.

```
ChickWeightGrowth <- ChickWeight %>%
  group_by(Chick) %>%
  mutate(growth=c(0,diff(weight))) %>%
  filter(growth != 0)

ChickWeightGrowthMean <- ChickWeightGrowth %>%
  group_by(Time, Diet) %>%
  summarize(growth=mean(growth))

ggp <- ggplot(data=ChickWeightGrowthMean, mapping=aes(x=Time, y=growth, colour=Diet)) +
  geom_line(mapping=aes(linetype=Diet)) +
  geom_point(mapping=aes(shape=Diet))

ggp
```



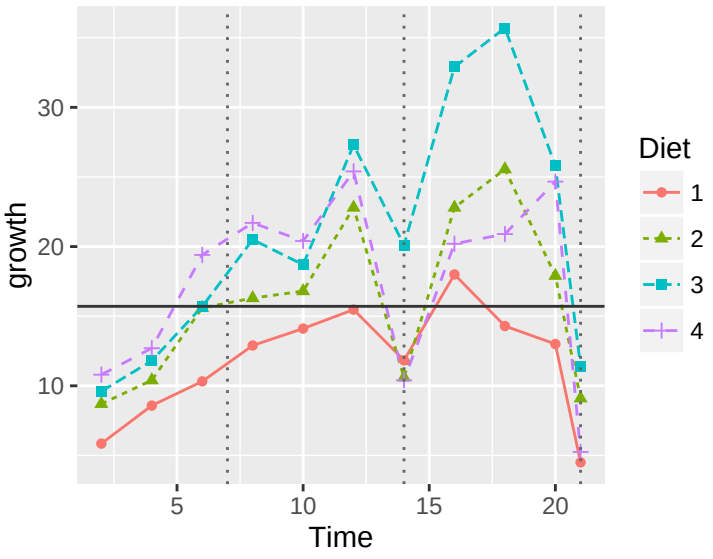
At this point you may want to add the average growth to the plot.

```
growth_avg <- ChickWeightGrowth %>%
  magrittr::use_series(growth) %>%
  mean

ggp1 <- ggp +
  geom_hline(yintercept = growth_avg, colour="grey20")

ggp1
```



Data refers to a three-weeks period. It may be interesting to highlight each week adding a vertical line at days 7, 14 and 21.

```
ggp1 +
  geom_vline(xintercept = c(7,14,21), colour="grey40", linetype=3)
```

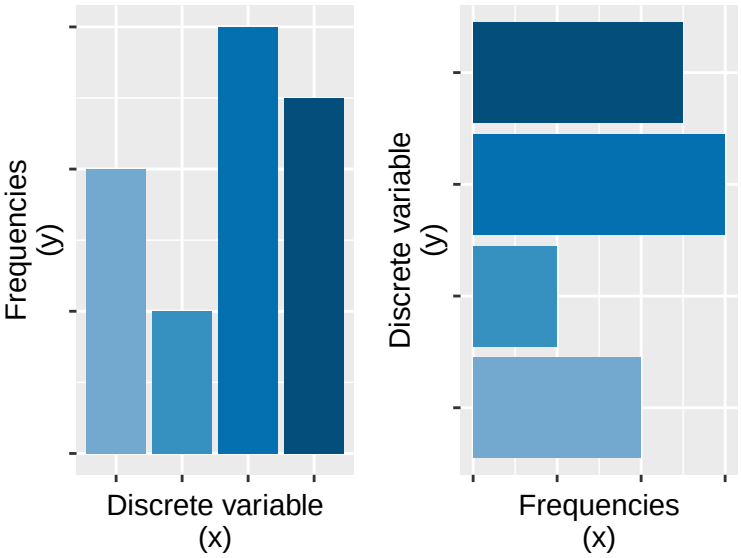


4 Creating a Bar Plot

```
require(dplyr)
require(ggplot2)
require(qdata)
data(bands)
```

Bar plots are used to display numeric values for different categories. Although they appears similar to Histograms, these plots are very different: bar plots are used for categorical x values, bars should be spaced and the weight of the bar has no meaning while histograms are used for continuous x values, bars (that are called *bins*) must not be spaced and the weight of bins depends on data.

To avoid confusion between bar plots and histograms, some authors suggest to build bars horizontally.

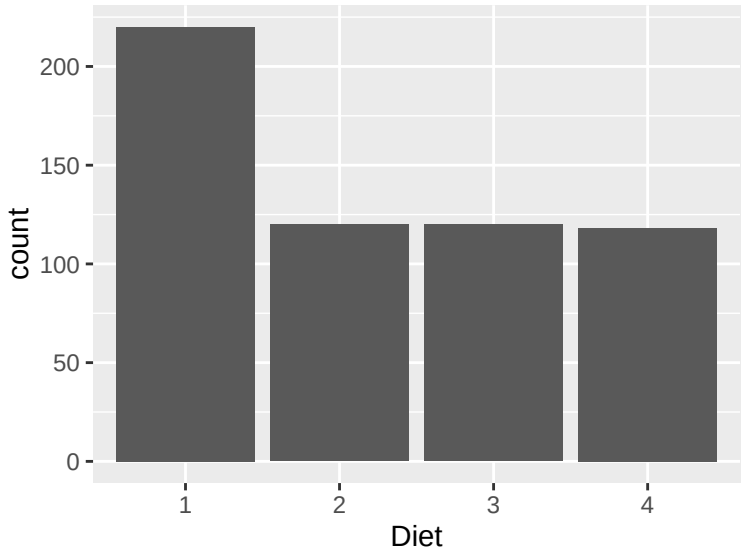


This chapter presents how to build bar plots and introduces other basic concepts of `ggplot2` grammar: `stat` (layers) and positioning.

4.1 The first bar plot

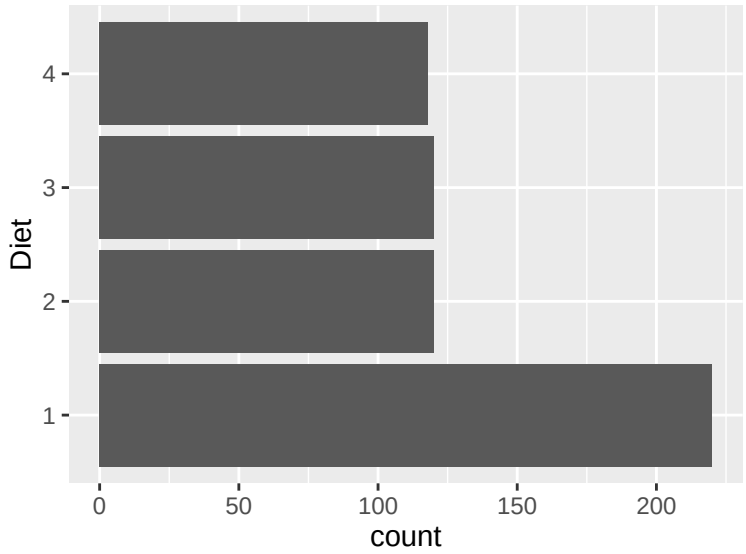
As seen in the previous chapter, chicks receive one of four diets (see `ChickWeight` data set). A graphical summary about how chicks receive each diet can be obtained.

```
ggplot(data=ChickWeight, mapping=aes(x=Diet)) +
  geom_bar()
```



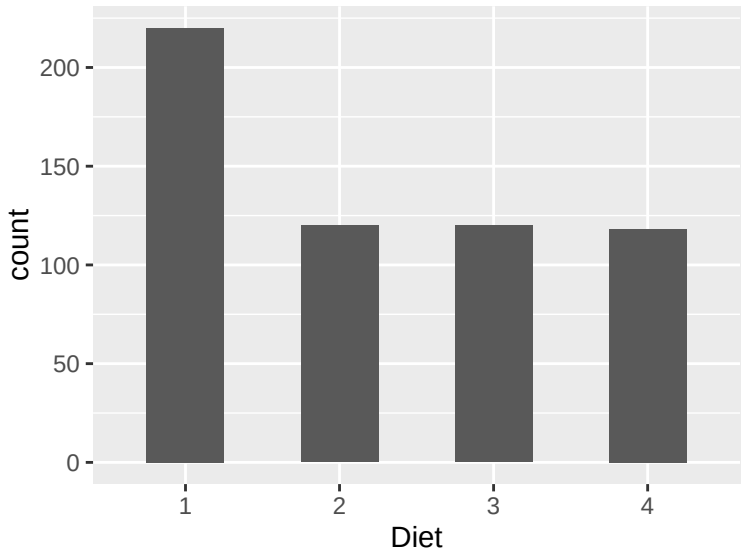
If you prefer horizontal bars, just flip the plot with `coord_flip()`.

```
ggplot(data=ChickWeight, mapping=aes(x=Diet)) +
  geom_bar() +
  coord_flip()
```



Another way to distinguish bar plots and histograms are tiny bars, to increase the space among bars.

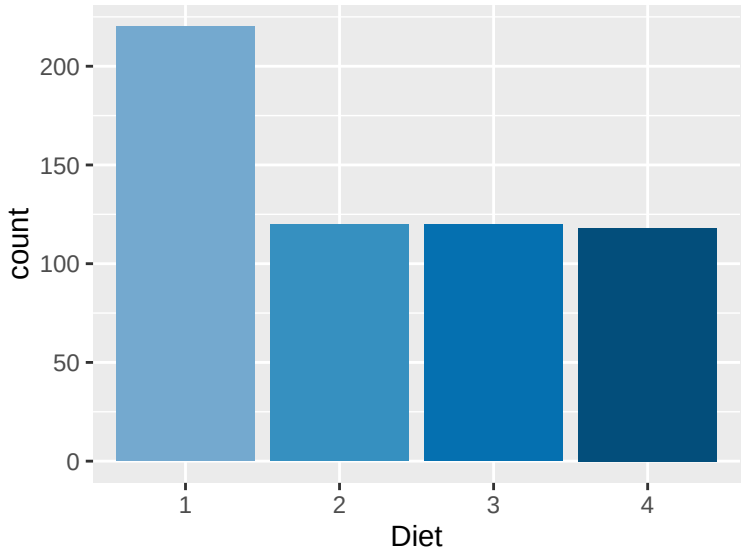
```
ggplot(data=ChickWeight, mapping=aes(x=Diet)) +
  geom_bar(width=0.5)
```



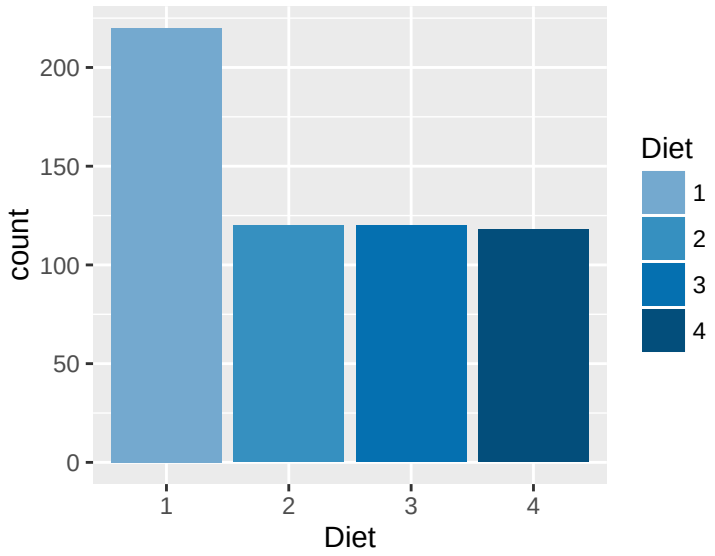
4.2 Setting and mapping variables to bar plots

Colours and other available aesthetic settings (`linetype` and `size`) can be set or mapped as usual. Remember that `colour` controls the bar outline, while `fill` controls the bar colour.

```
ggplot(data=ChickWeight, mapping=aes(x=Diet)) +
  geom_bar(fill=c("#74a9cf", "#3690c0", "#0570b0", "#034e7b"))
```

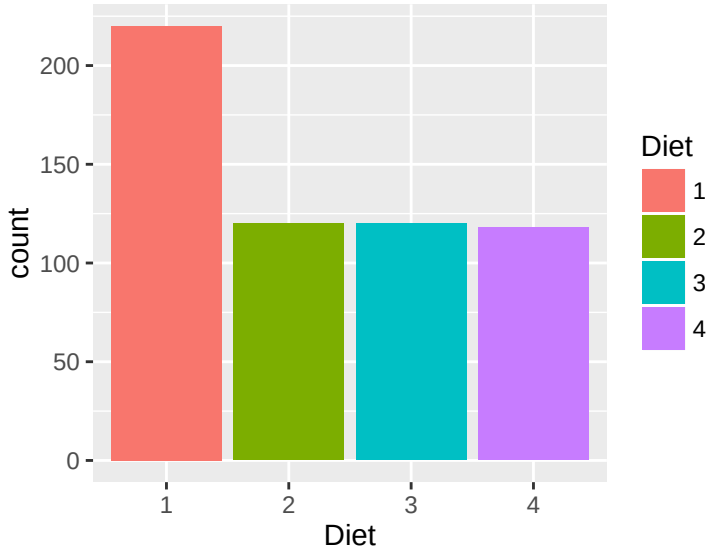



```
ggplot(data=ChickWeight, mapping=aes(x=Diet)) +
  geom_bar(mapping=aes(fill=Diet)) +
  scale_fill_manual(values=c("#74a9cf", "#3690c0", "#0570b0", "#034e7b"))
```



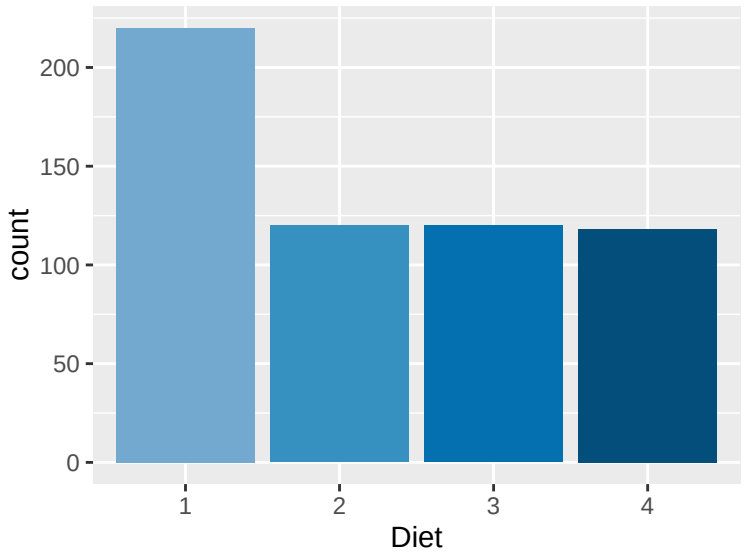
These plots are identical except that for legend, but they are based on two different approaches. In the first plot, fill colours are set. If you do not set four different fill colours, all bars have the same colours as shown above. In the second plot, fill colours are mapped to the levels of `Diet`. Since each bar represents a different type of diet, each bar has a different colour also when you do not set manually fill colours with `scale_fill_manual()`.

```
ggplot(data=ChickWeight, mapping=aes(x=Diet)) +
  geom_bar(mapping=aes(fill=Diet))
```



The use of `scale_fill_manual()` allows you to choose which colours should be used. Since fill colours are mapped with a variable, in this case the legend will be shown. In this case the legend is useless, because the same aesthetic is mapped to `x` and you already know which Diet refer to each colour. You can hide the legend in `scale_fill_manual()`.

```
ggplot(data=ChickWeight, mapping=aes(x=Diet)) +
  geom_bar(mapping=aes(fill=Diet)) +
  scale_fill_manual(values=c("#74a9cf", "#3690c0", "#0570b0", "#034e7b"), guide=FALSE)
```



–>

`ggplot2` provides some functions (e.g. `scale_*_brewer` or `scale_*_gradient`) to choose a palette (i.e. a set of coherent colours), but these functions will be shown in *Mapped Aestethics Customization* chapter. You can still use `scale_*_manual` functions and refer to some on-line tool like Color Brewer or Paletton for a better choice of colours.

4.3 Summarized data and `stats`

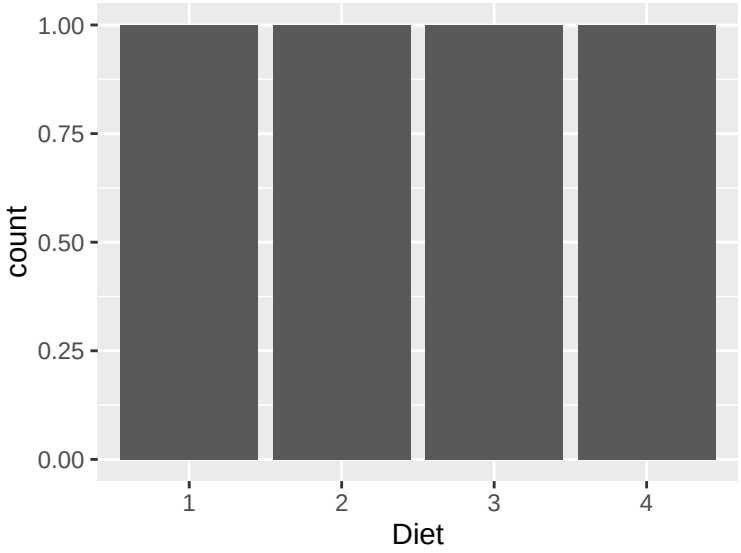
Sometimes data comes already summarized. As an example, you can have the following frequency tables without the original data.

```
ChickWeightFreq <- ChickWeight %>%
  group_by(Diet) %>%
  summarize(n=n())

ChickWeightFreq
```

```
## # A tibble: 4 × 2
##     Diet     n
##   <fctr> <int>
## 1      1   220
## 2      2   120
## 3      3   120
## 4      4   118
```

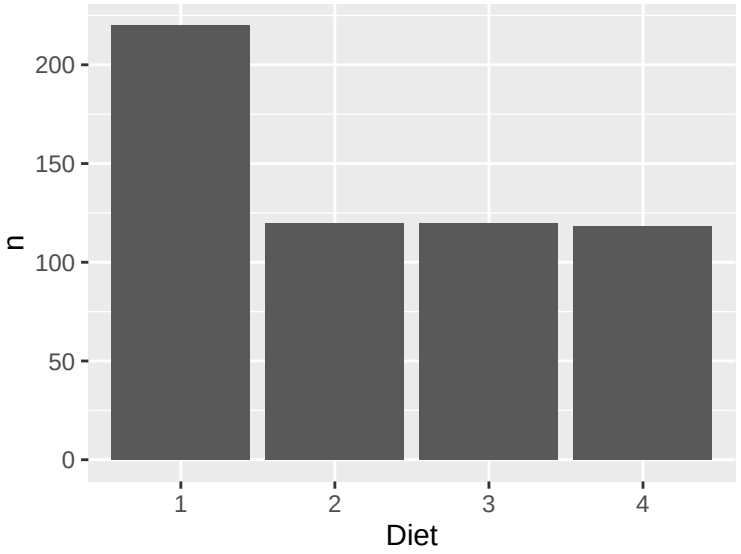
You can try to build the previous plot in the same way.

```
ggplot(data=ChickWeightFreq, mapping=aes(x=Diet)) +
  geom_bar()
```



The result is a plot with four bars of length 1. Why? The reason is quite simple. By default, `geom_bar()` scans the `Diet` column counting how many observations have Diet=1, how many have Diet=2 and so on. In this case, you must tell `ggplot` you already have the count.

```
ggplot(data=ChickWeightFreq, mapping=aes(x=Diet, y=n)) +
  geom_bar(stat="identity")
```



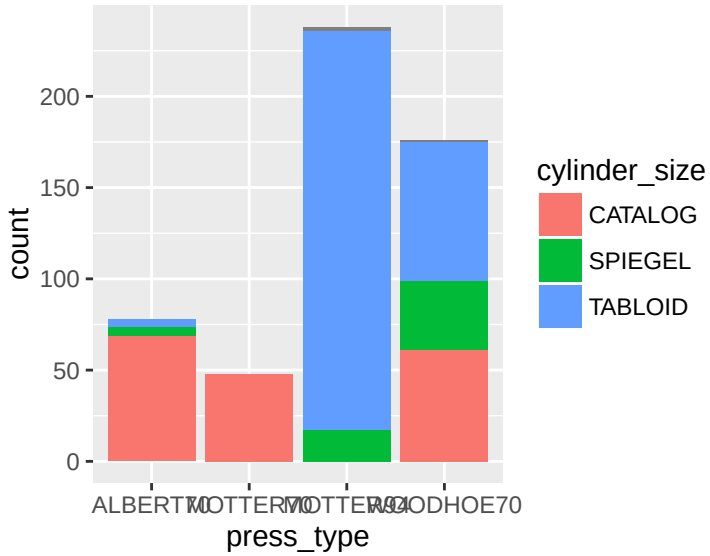
Since you have an `y` variable containing frequencies, you tell `ggplot` the variable containing counts (`y=n` in the example) and to `geom_bar` that `stat="identity"` must be used.

A statistical transformation, or `stat`, transforms the data, typically by summarizing it in some manner. All `geom`s are based on a statistical transformation. By default, almost all `geom`s seen until now uses `stat="identity"`, that do not transform data. As just seen, by default `geom_bar()` uses `stat="count"` that counts the number of cases at each `x` position. If you do not want to transform your data, `stat="identity"` must be supplied.

4.4 Stacked and grouped bar plots

If you are a quality engineer analysing the `bands` data, you may be interested to the number of presses for each type, distinguishing them by the cylinder size.

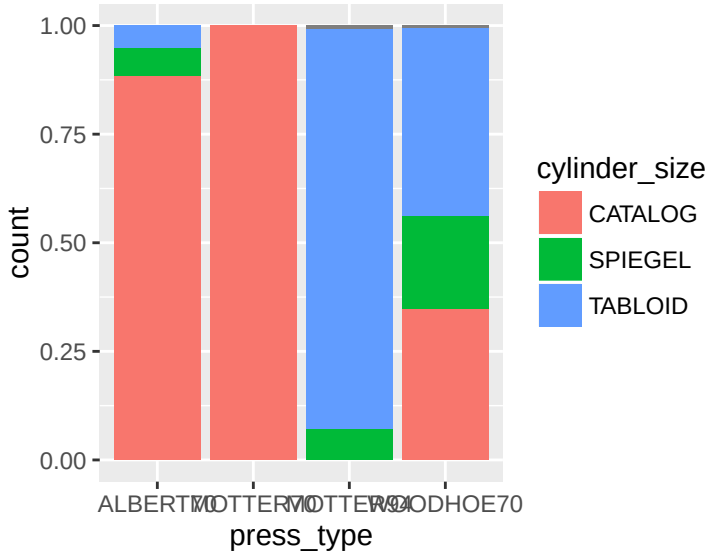
```
ggplot(data=bands, mapping=aes(x=press_type, fill=cylinder_size)) +
  geom_bar()
```



Mapping `fill` to `cylinder_size` does the job. Notice a small gray area at the top of the last two bars: this means there are few cases in which `cylinder_size` is missing.

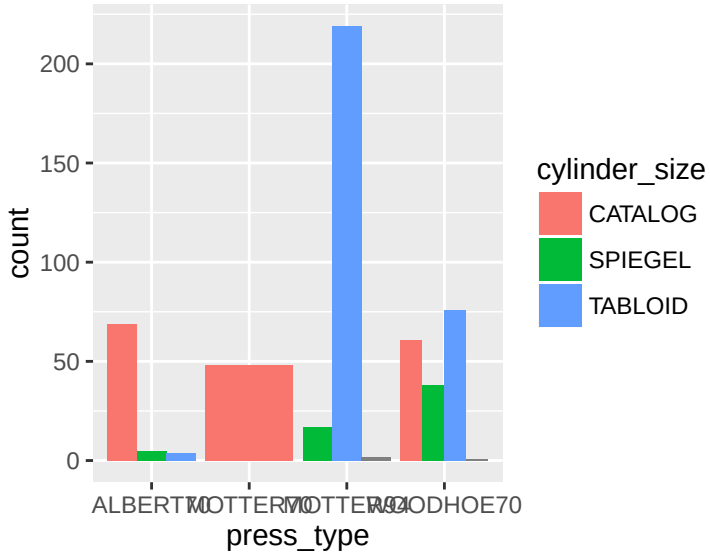
Sometimes you may be interested at the distribution of `cylinder_size` for each `press_type`.

```
ggplot(data=bands, mapping=aes(x=press_type, fill=cylinder_size)) +
  geom_bar(position="fill")
```



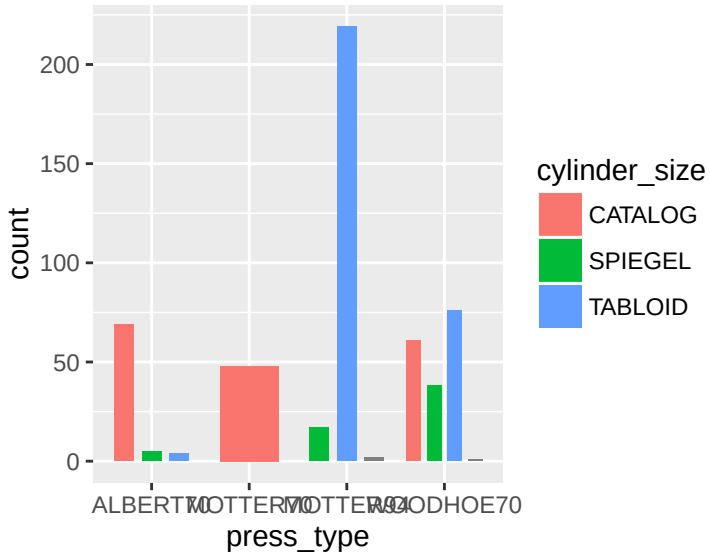
In the case you prefer a bar for each combination of `press_type` and `cylinder_size`, you can `dodge` the bars.

```
ggplot(data=bands, mapping=aes(x=press_type, fill=cylinder_size)) +
  geom_bar(position="dodge")
```



Notice that `position="dodge"` is a short key for `position=position_dodge()` with its default value and you can modify this value to add space between bars. In the case you want to add space between bars, you probably want also to modify the `width` in order to add spaces among each group of bars.

```
ggplot(data=bands, mapping=aes(x=press_type, fill=cylinder_size)) +
  geom_bar(position=position_dodge(0.85), width=0.6)
```



The `width` in `geom_bar()` determines the width of the bar; the `width` in `position_dodge()` determines the position of each bar. Probably you can easily understand their behaviour after you play with them for a while.

All data related to `MOTTER70` press has `CATALOG` as `cylinder_size`. In this case, the bars related to other `cylinder_size` are missing and the `MOTTER70` bar will expand to fill the whole space.

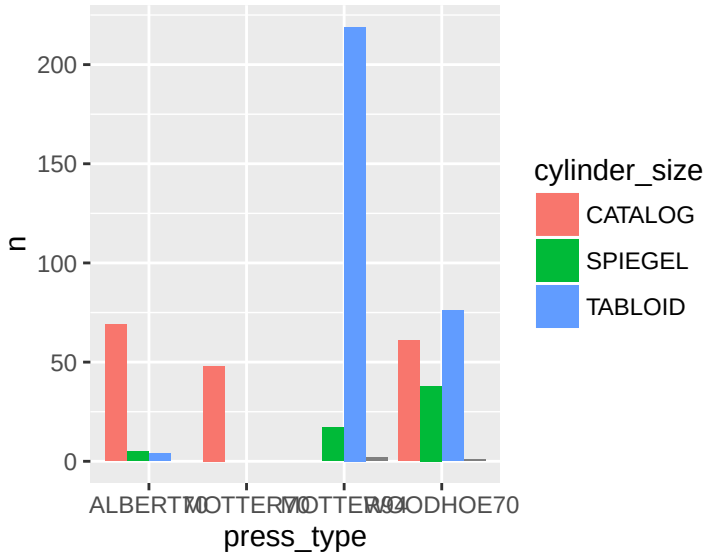
To avoid this behaviour you must have at least a `NA` value for each combination of factors. Since both `ggplot2` when you plot and `dplyr` when you build a frequency table, do not show the count for missing combinations, you should manually build a frequency table like the following one.

```
bands_freq_na <- bands %>% group_by(press_type, cylinder_size) %>% summarise(n = n()) %>%
  right_join(
    expand.grid(
      press_type = bands %>% magrittr::use_series(press_type) %>% levels,
      cylinder_size = c(bands %>% magrittr::use_series(cylinder_size) %>% levels, NA)
    )
  )
bands_freq_na
```

```
## Source: local data frame [16 x 3]
## Groups: press_type [?]
##
##    press_type cylinder_size     n
##        <fctr>        <fctr> <int>
## 1    ALBERT70       CATALOG    69
## 2    MOTTER70       CATALOG    48
## 3    MOTTER94       CATALOG    NA
## 4   WOODHOE70       CATALOG    61
## 5    ALBERT70       SPIEGEL     5
## 6    MOTTER70       SPIEGEL    NA
## 7    MOTTER94       SPIEGEL    17
## 8   WOODHOE70       SPIEGEL    38
## 9    ALBERT70       TABLOID     4
## 10   MOTTER70       TABLOID    NA
## 11   MOTTER94       TABLOID   219
## 12  WOODHOE70       TABLOID    76
## 13   ALBERT70            NA    NA
## 14   MOTTER70            NA    NA
## 15   MOTTER94            NA     2
## 16  WOODHOE70            NA     1
```

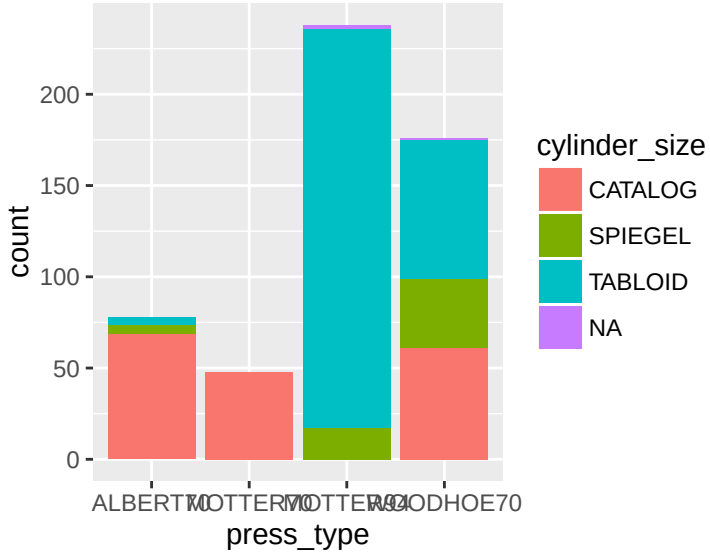
At this point, you can easily build the plot starting from frequency table. Just remember to map the counts to `y` and to add `stat="identity"` to `geom_bar()`.

```
ggplot(data=bands_freq_na, mapping=aes(x=press_type, y=n, fill=cylinder_size)) +
  geom_bar(stat="identity", position="dodge")
```



As you see in the previous plots, `cylinder_size` is a factor variable with three levels: `CATALOG`, `SPIEGEL` and `TABLOID` but it has also missing values (`NA`).
To visualize also missing values (`NA`) as `cylinder_size` level in plots legend, you have to recode its levels, adding also `NA` as level.

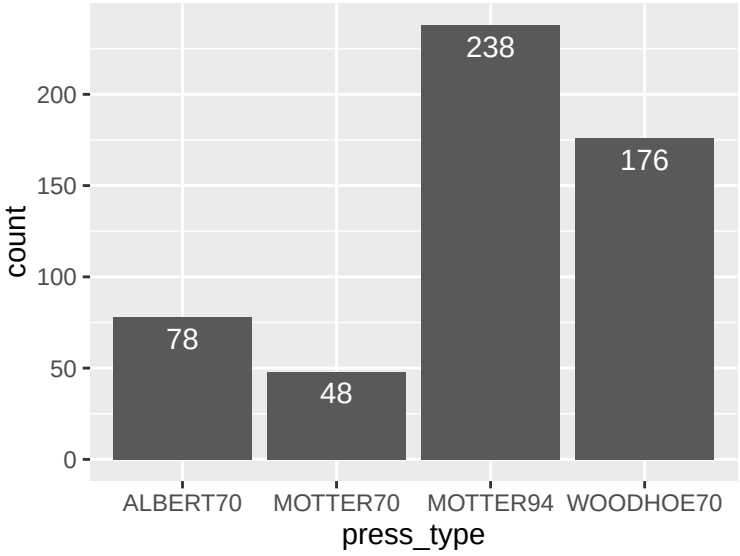
```
bands <- bands %>%
  mutate(cylinder_size = as.character(cylinder_size),
         cylinder_size =ifelse(is.na(cylinder_size),"NA",cylinder_size),
         cylinder_size = factor(cylinder_size,
                                levels= c("CATALOG", "SPIEGEL", "TABLOID", "NA"),
                                labels = c("CATALOG", "SPIEGEL", "TABLOID", "NA")))

ggplot(data=bands, mapping=aes(x=press_type, fill=cylinder_size)) +
  geom_bar()
```



4.5 Adding text to plots

Plots are useful to catch the eye of your audience but sometimes they cannot replace numbers or text. Adding text to a plot can help you to integrate information in your plot.

```
bands_freq <- bands %>% group_by(press_type) %>% summarize(n=n())

ggplot(data=bands, mapping=aes(x=press_type)) +
  geom_bar() +
  geom_text(data=bands_freq, mapping=aes(y=n, label=n), vjust=1.5, colour="white")
```

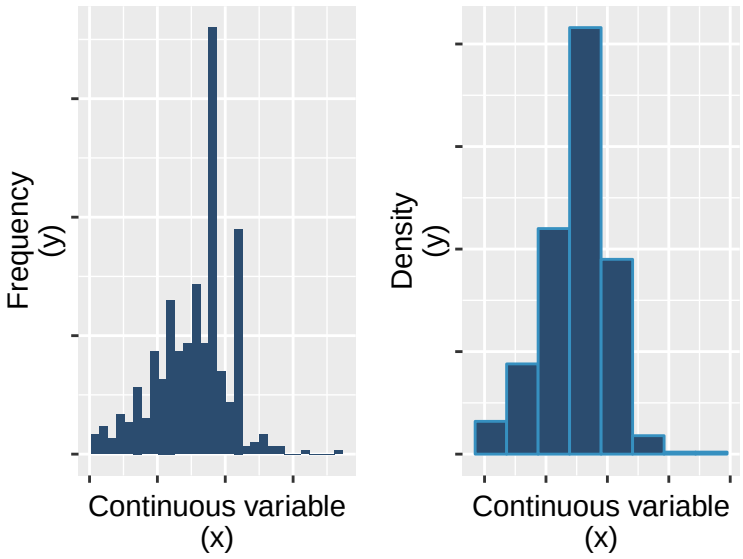


`geom_text()` works as any other `geom` seen until now. It requires `x` and `y` coordinates; in this case, `x` is `press_type` and it is inherited from `ggplot()`, while `y` should be passed. Since `bands` data do not contains the count, that is computed by `stat_count` inside `geom_bar()`, default data are overwritten in `geom_text()` in order to use frequencies. `geom_text()` requires `label` that is the text to be written. Outside `aes()`, `vjust` move down (or up) the text and `colour` set text colour.

5 Creating a Histogram

```
require(dplyr)
require(ggplot2)
require(qdata)
data(bands)
```

Histograms are used to summarize a continuous variable into classes, called *bins*. The area (and not the height) of each bin is proportional to the frequency of cases in the bin. The vertical axis is not frequency but density. When bins are equal size, a rectangle is erected over the bin with height proportional to the frequency. As the adjacent bins leave no gaps, the rectangles of a histogram touch each other to indicate that the original variable is continuous.

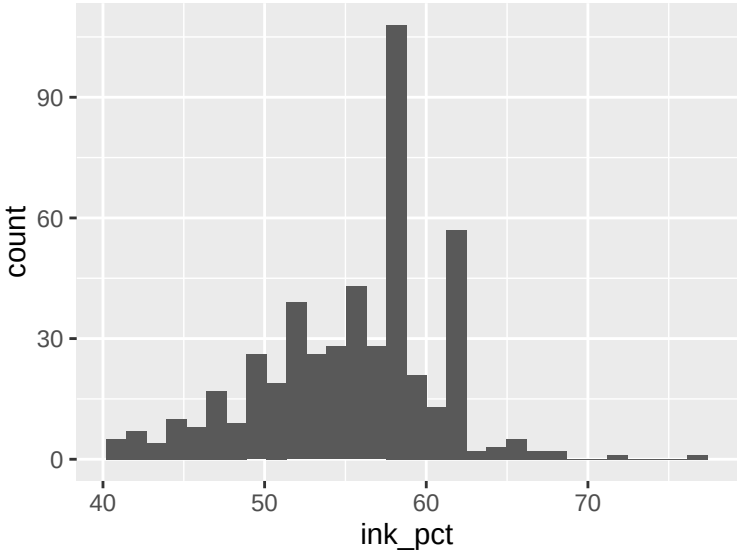


This chapter presents how to build histograms and introduces a basic concepts of `ggplot2` graphics: faceting. It shows also how to build density curves and plots and frequency polygons.

5.1 The first histogram

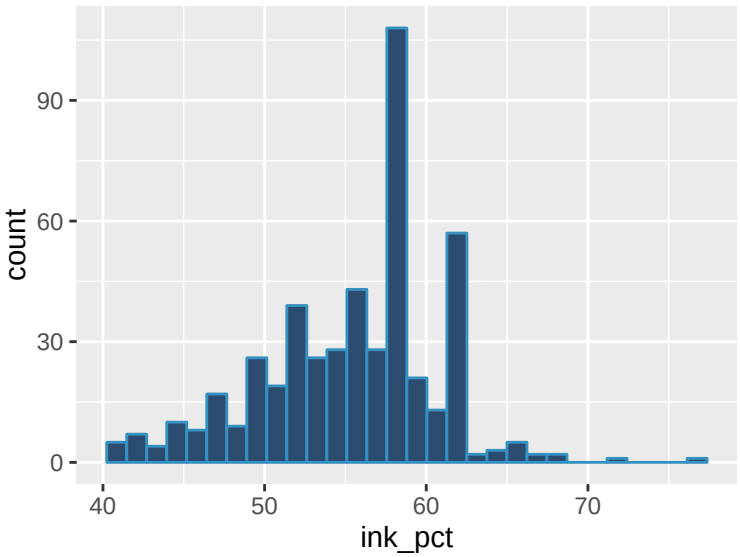
As a quality engineer, you are interested to the distribution of ink percentage in `bands` data.

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram()
```



As usual, aesthetics can be set to modify the appearance of plot.

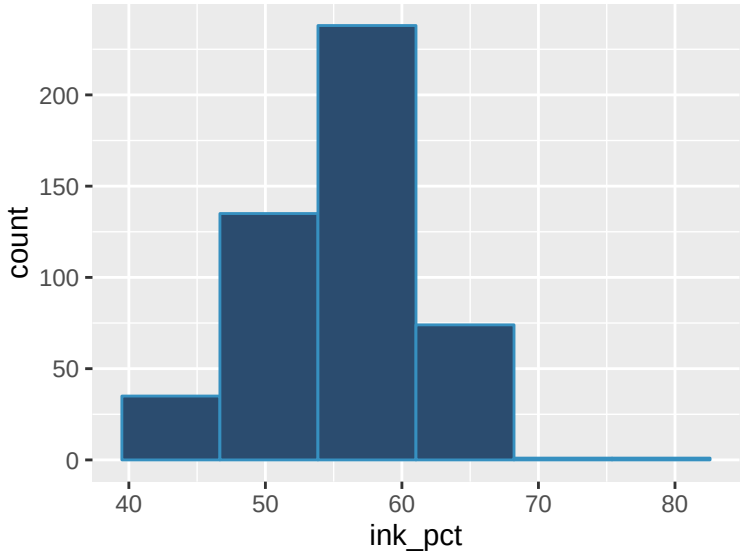
```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F", colour="#3690c0")
```



By default, the data is grouped into 30 bins. You can modify the number of bins:

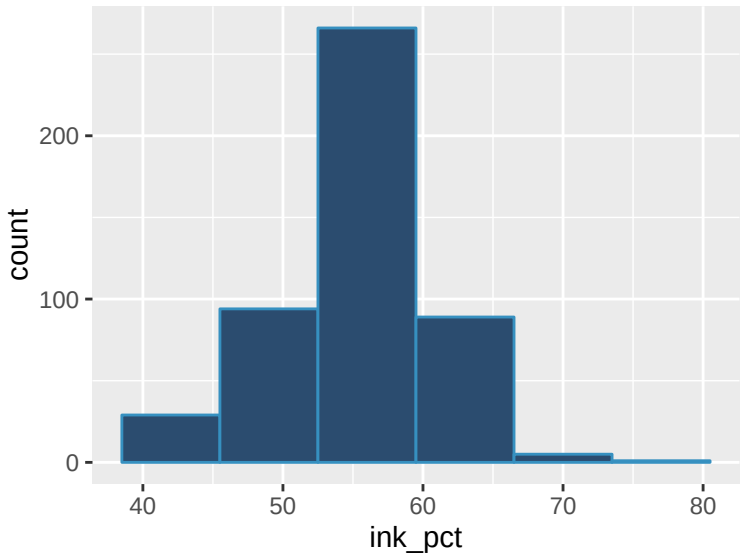
- setting the number of bins:

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F", colour="#3690c0", bins=6)
```



- setting the width of each bin:

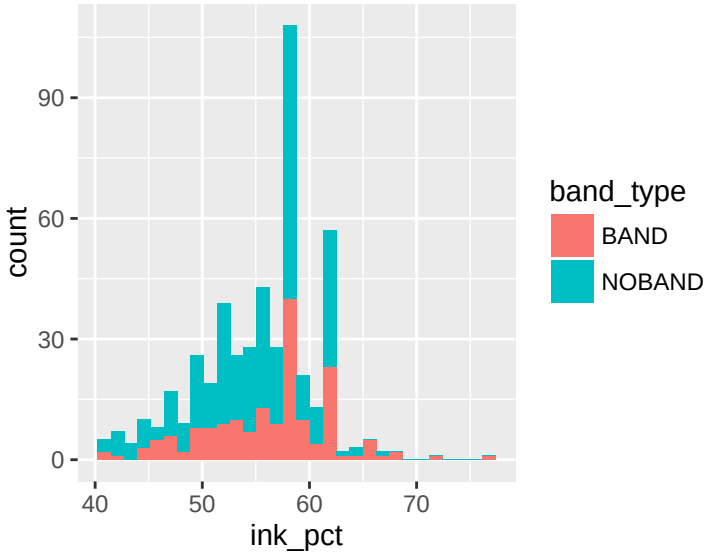
```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F", colour="#3690c0", binwidth=7)
```



5.2 Mapping variables to histograms and faceting

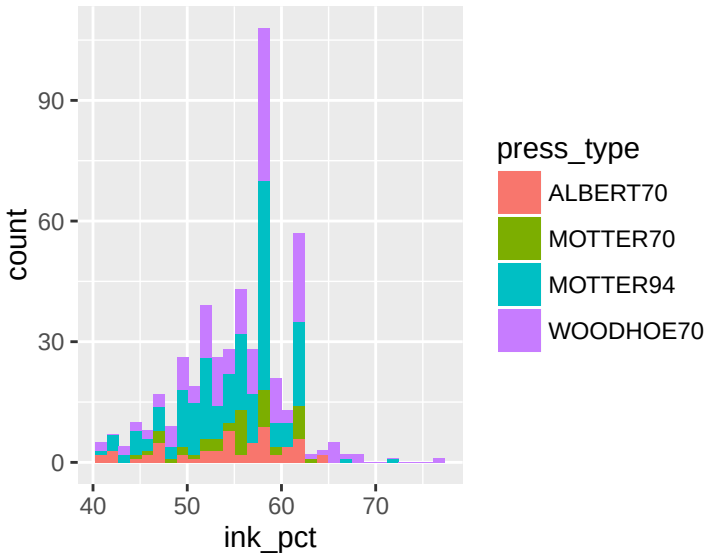
May be interesting analysing the distribution of ink percentage for each level of `band_type`.

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(mapping=aes(fill=band_type))
```



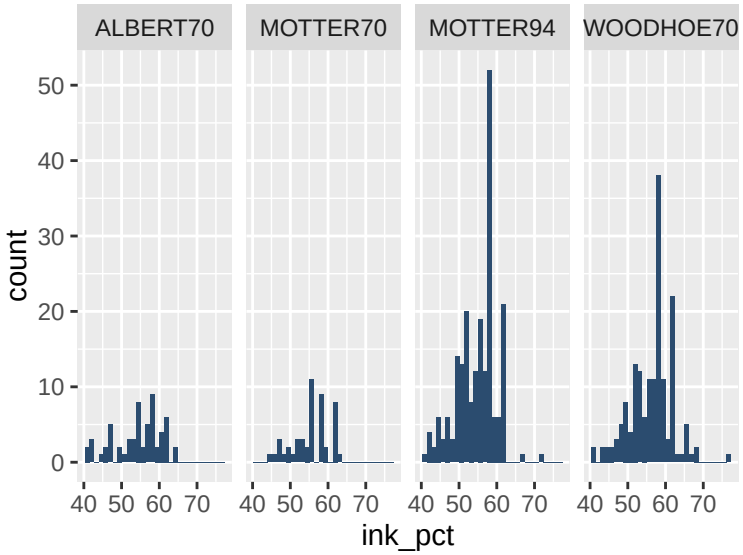
Mapping the grouping variable to `fill`, two overlapped distributions are shown. In some cases, as shown above, this solution may work. In other cases, the result may be very difficult to be understood. Suppose you are in the distribution of ink percentage for each level of `press_type`

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(mapping=aes(fill=press_type))
```



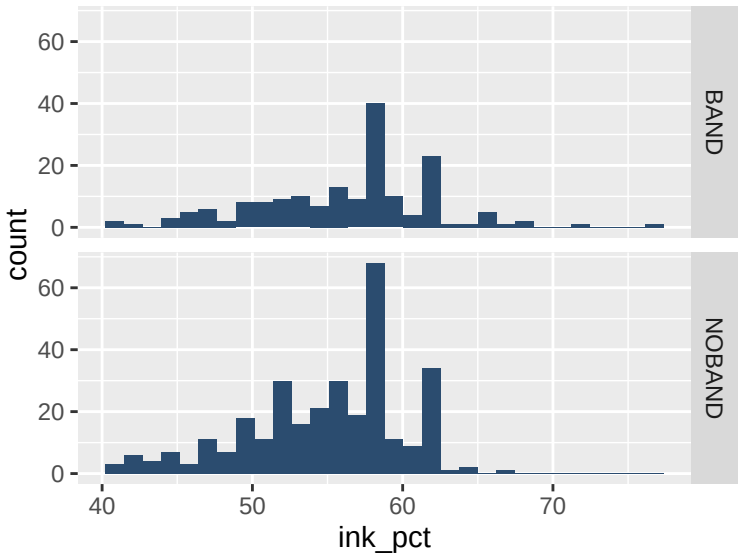
In this case, four different histograms may produce a more readable result.

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F") +
  facet_grid(. ~ press_type)
```



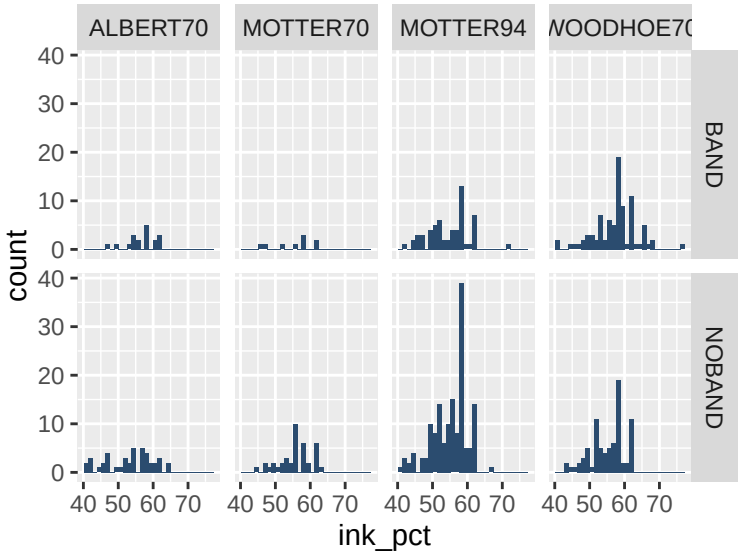
`facet_grid()` produces a different panel for each level of `press_type`. It requires a formula style: `rows ~ columns`. The dot in the formula is used to indicate there should be no faceting on this dimension (either row or column). The following example shows faceting on rows.

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F") +
  facet_grid(band_type ~ .)
```



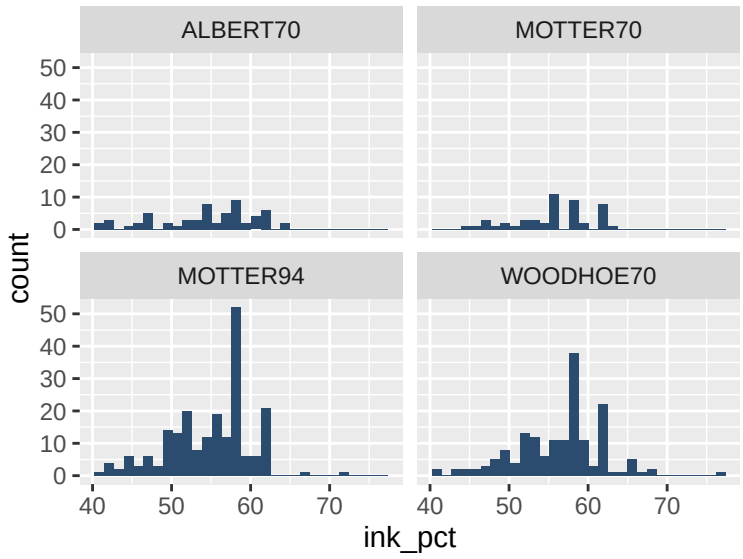
Faceting on both dimensions:

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F") +
  facet_grid(band_type ~ press_type)
```



When you have a categorical variable with many levels, it does not make sense to try and display them all in one row (or one column) and you may want to wrap it. `facet_wrap(~ variable)` wraps a sequence of panels into rows and columns, to better fit screen.

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F") +
  facet_wrap(~ press_type)
```

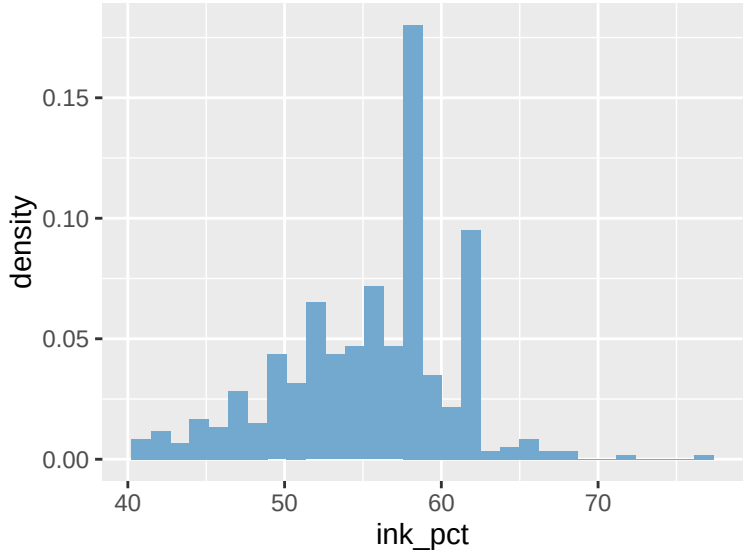


–>

5.3 Making a density histogram

To show densities instead than count (frequencies) on the y axis, the special variable `..density..` must be used.

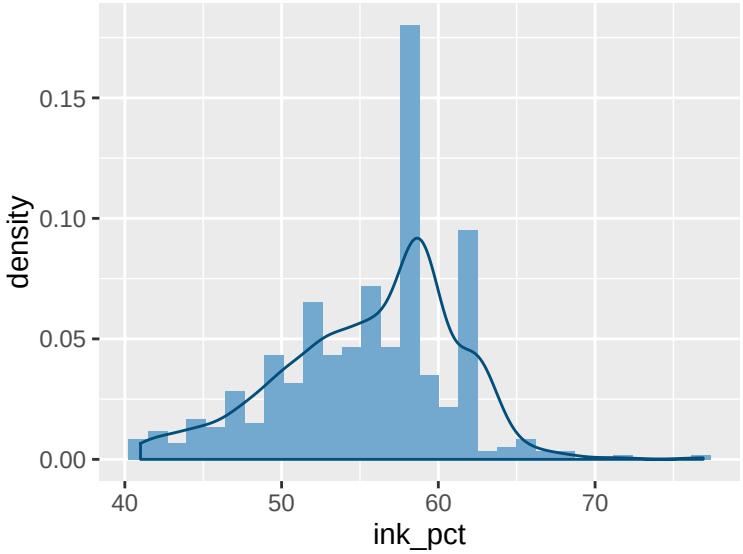
```
ggplot(data=bands, mapping=aes(x=ink_pct, y=..density..)) +
  geom_histogram(fill="#74a9cf")
```



The special variables in `ggplot` with double periods around them, as `..density..`, are returned by a `stat` transformation of the original dataset.

A density curve can be added to compare the theoretical and observed distributions.

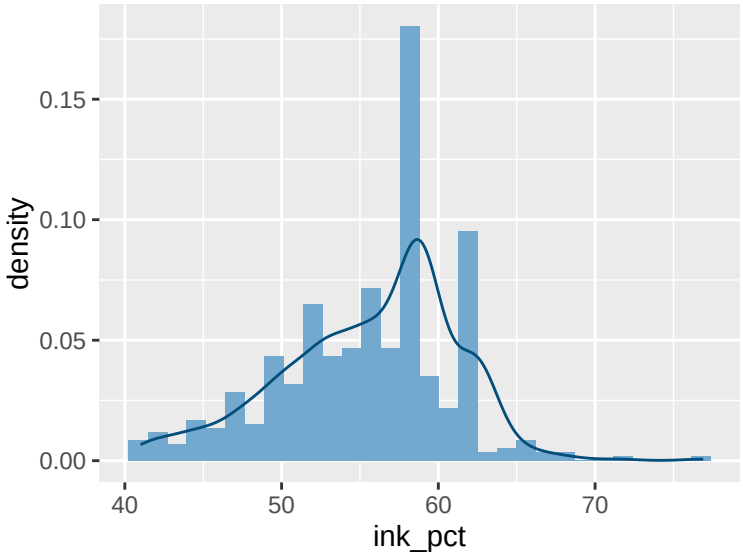
```
ggplot(data=bands, mapping=aes(x=ink_pct, y=..density..)) +
  geom_histogram(fill="#74a9cf") +
  geom_density(colour="#034e7b")
```



`geom_density()` function draws a kernel density curve, which is an estimate of the population distribution, based on the sample data.

`geom_density()` function draws a closed polygon. If you do not like the lines along the side and bottom, you can draw a density curve using `geom_line` with `stat="density"`.

```
ggplot(data=bands, mapping=aes(x=ink_pct, y=..density..)) +
  geom_histogram(fill="#74a9cf") +
  geom_line(stat="density", colour="#034e7b")
```



In particular, `geom_line()`is used with `"density"` statistical transformation, which means that a statistical transformation is applied to data, in particular `ink_pct` density is estimated.

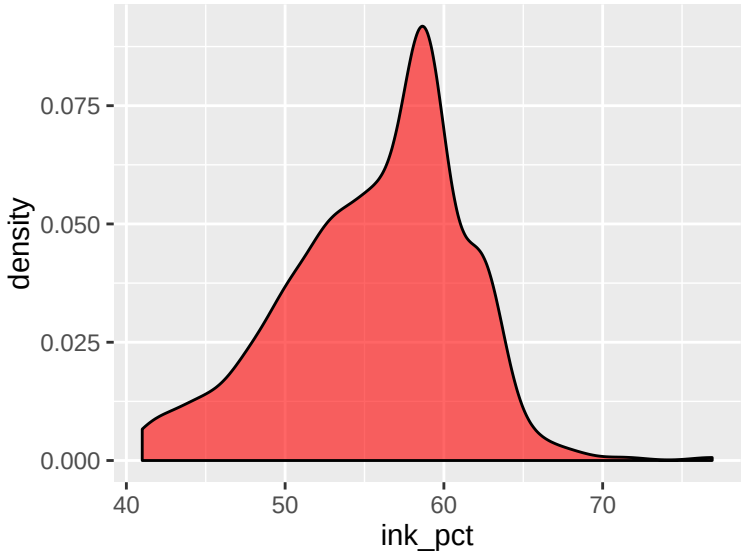
5.4 Making a Density curve

As we said, a density curve, in particular a kernel density curve, is an estimate of the population distribution, based on the sample data.

We have already seen how `geom_density()` works. Let us deepen its functionalities.

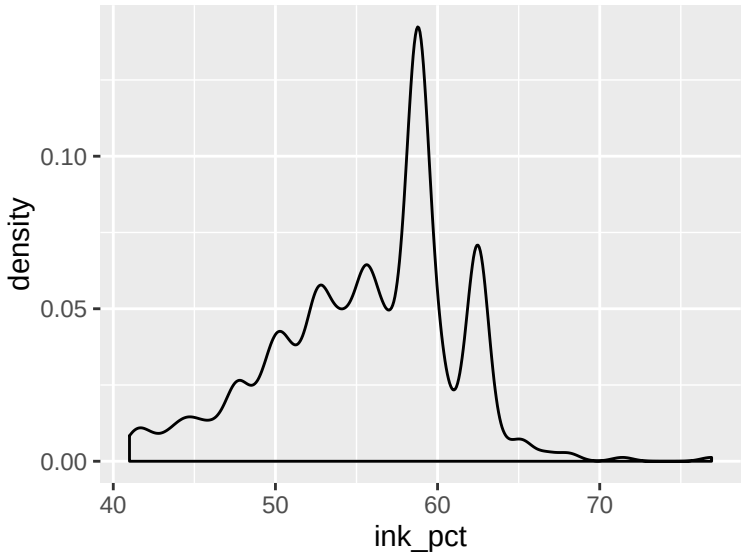
Suppose we want to analyze the density distribution estimation of `ink_pct` variable. To better visualize the area under the curve you can customize it by setting `fill` and `alpha` arguments:

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_density(fill = "red", alpha = 0.6)
```



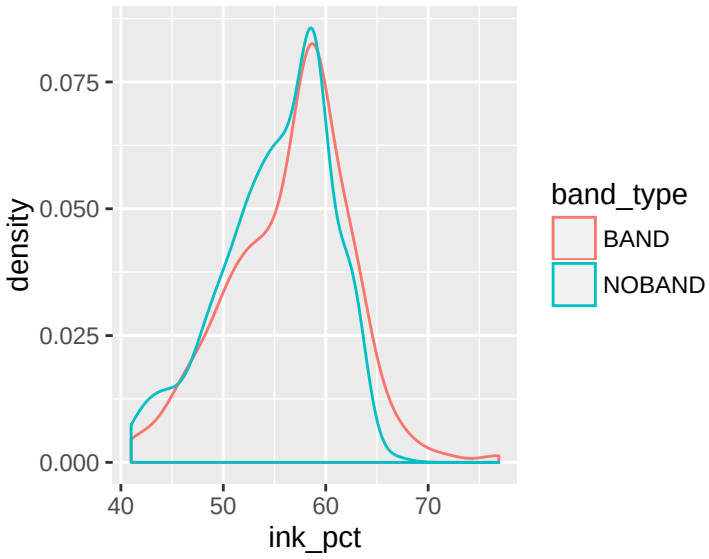
The amount of smoothing depends on the kernel bandwidth: the larger the bandwidth, the more smoothing there is. The bandwidth can be set with the adjust parameter, which has a default value of 1:

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_density(adjust = 0.5)
```



`geom_density()` can be used also to compare density distribution estimations of grouped data. The grouping variable must be map to an aesthetic like `colour` or `fill`. The grouping variable must be a factor or character vector. In this case we consider `band_type` variable:

```
ggplot(data=bands, mapping=aes(x=ink_pct, colour = band_type)) +
  geom_density()
```

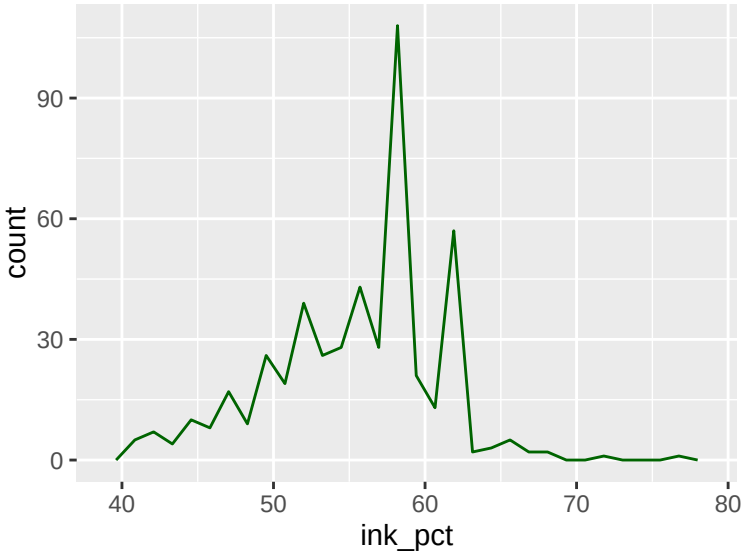


5.5 Frequency polygon

A frequency polygon appears similar to a kernel density estimate curve, see in the previous examples, but it work in the same way and it shows the same information as an histogram. In particular, both bin the data and then count the number of observations in each bin. The only difference is the display: histograms use bars and frequency polygons use lines.

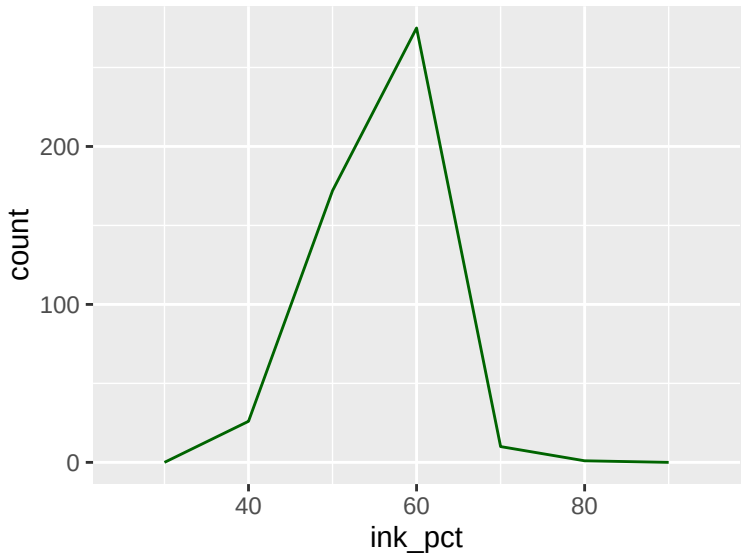
Suppose we want to analyze the distribution of `ink_pct` variable. `ggplot2` function that draws a frequency polygon is `geom_freqpoly()`:

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_freqpoly(col = "darkgreen")
```



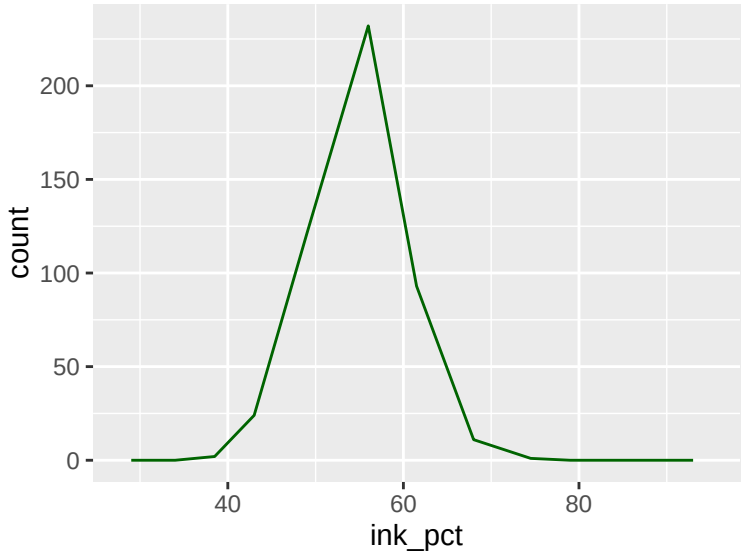
Also like a histogram, you can control the bin width for the frequency polygon:

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_freqpoly(col = "darkgreen", binwidth=10)
```



If you don't want evenly spaced bins you can set the `breaks` arguments:

```
ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_freqpoly(col = "darkgreen", breaks=c(30, 32, 36, 41, 45, 53, 59, 64, 72, 77, 81, 87, 91))
```

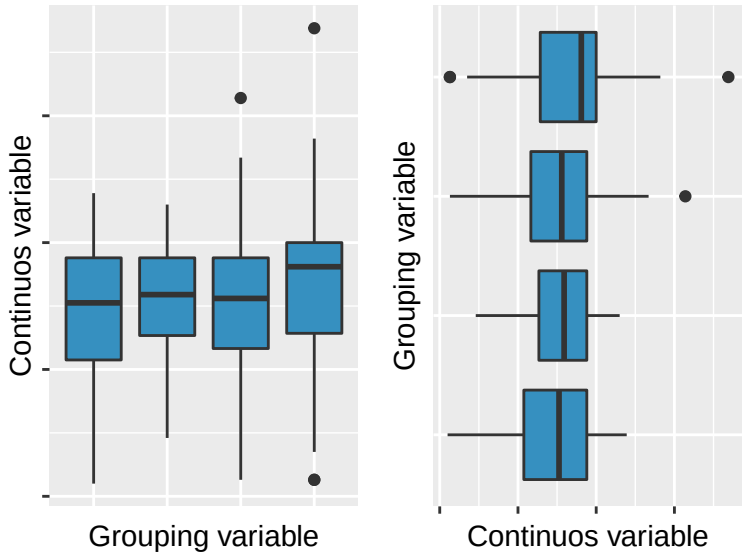


6 Creating a Box Plot

```
require(dplyr)
require(ggplot2)
require(qdata)
data(bands)
```

A Box-and-Whiskers Plot, or Box Plot, is a convenient way to draw data distribution. The box ranges from the first quartile to the third (inter-quartile range or IQR) with a line indicating the median (second quartile). The whiskers contains the lowest datum still within 1.5 IQR of the lower quartile, and the highest datum still within 1.5 IQR of the upper quartile. If there are data outside the range of whiskers, they are represented by

a dot. Box Plots are very popular among data analyst, but they are not suggested for a wider audience. Box plots can be drawn either horizontally or vertically.

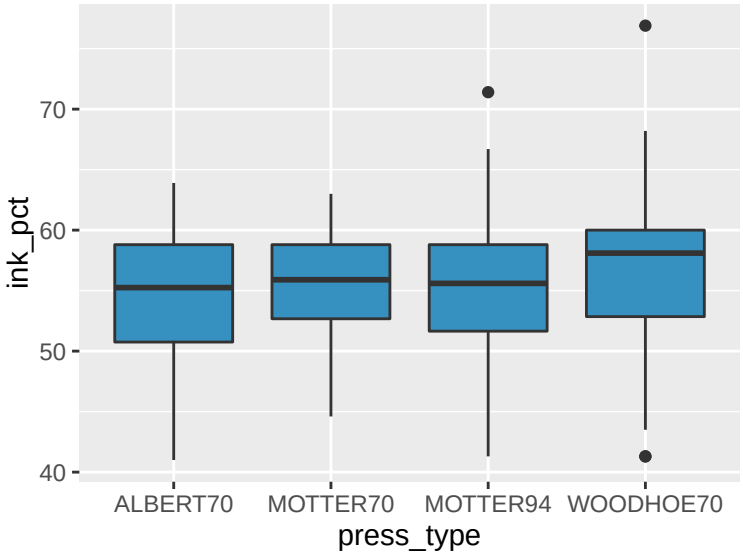


This chapter presents how to build box plots and violin plots. It show also how to add title and axis labels to the plot.

6.1 The first box plot

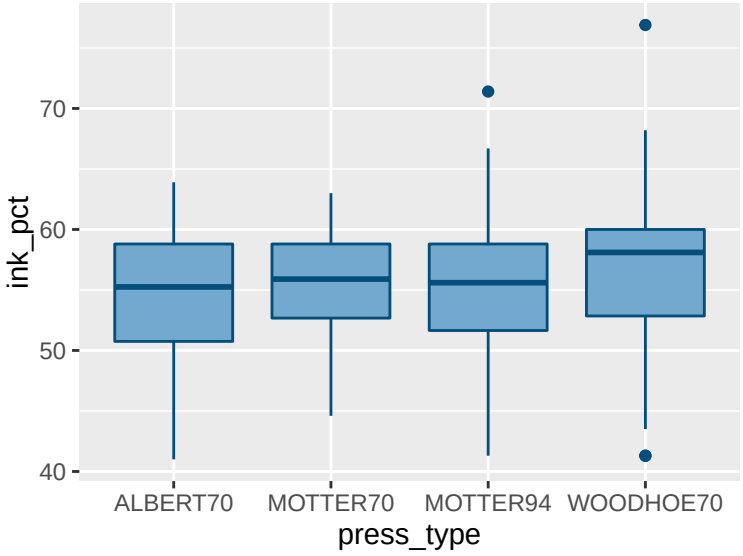
Supposing you are interested in the differences of ink percentage accordingly to the type of press, you can build four box plots to compare distributions.

```
ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#3690c0")
```



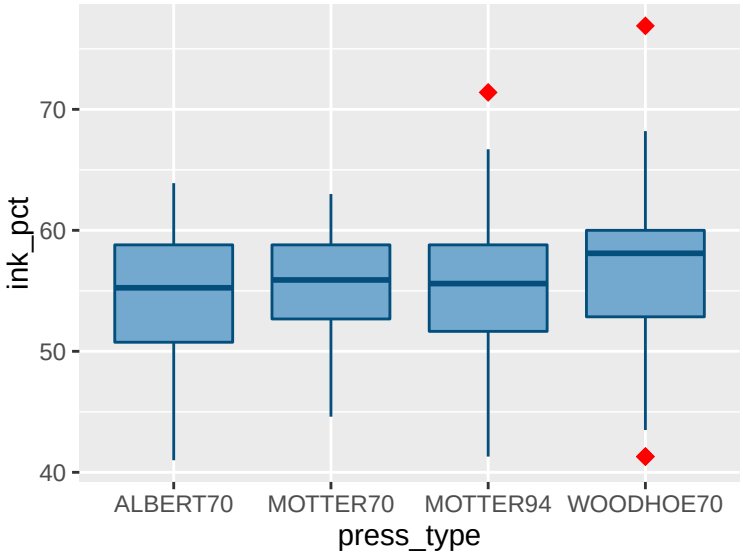
Appearance aesthetics work as seen until now; `fill` controls the box filling, while `colour` controls the box outline, whiskers and outline points.

```
ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b")
```



There are also a few `outline.*` parameter to set aesthetics for outlier points.

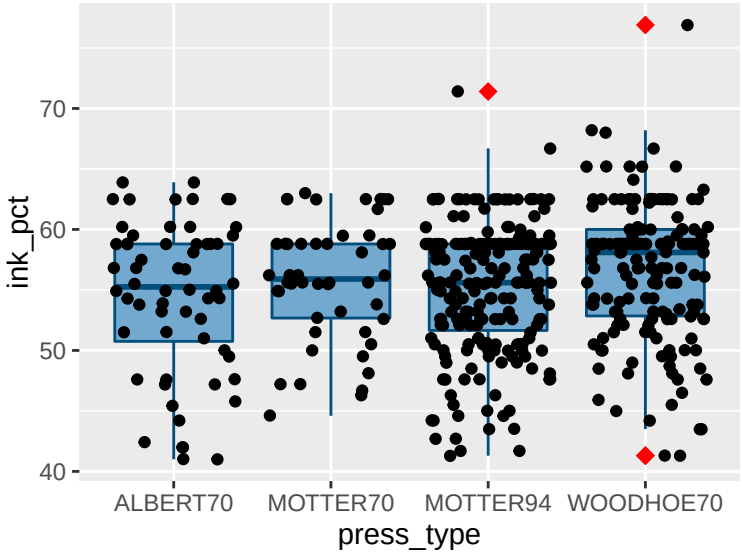
```
ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b", outlier.colour="red", outlier.shape=18, outlier.size=3)
```



6.2 Adding points to boxplot

Boxplot summarises the bulk of distributions with only five numbers: median, first quartile, third quartile, minimum and maximum. You can also add points to boxplots to identify the observations by using `geom_jitter()` function:

```
ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b", outlier.colour="red", outlier.shape=18, outlier.size=3)
  geom_jitter()
```

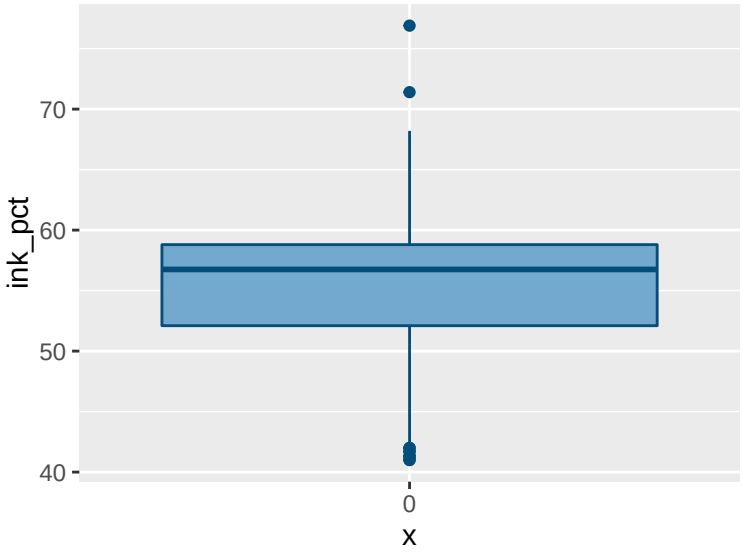


`geom_jitter()` adds a little random noise to the plot which can help avoid overlapping. Hovewer, it works with only relatively small dataset.

6.3 Making a single box plot

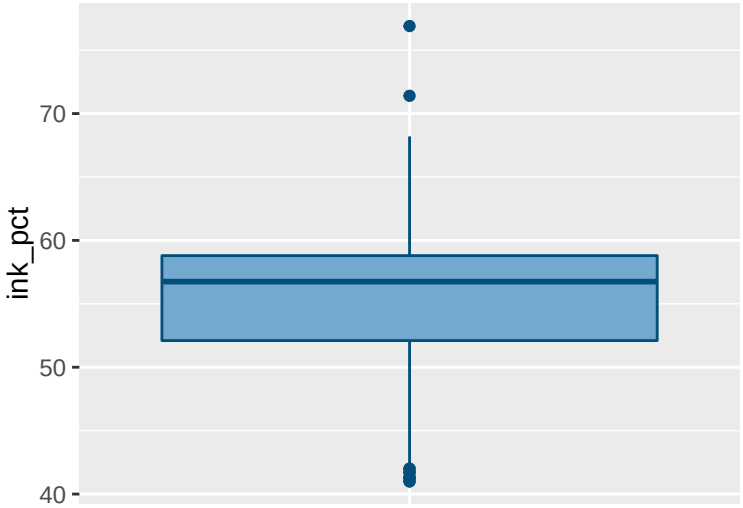
Surprisingly, it is more complicated to draw a single box plot. This is because `geom_boxplot()` requires an `x` value that is used as grouping variable. You have to set a single value for `x` when you do not have a grouping variable.

```
ggplot(data=bands, aes(x="0", y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b")
```



It does the job, however you have to remove the x-axis tick marker and label in order to have a nice result.

```
ggplot(data=bands, aes(x="0", y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b") +
  xlab("") +
  theme(axis.ticks.x = element_blank(), axis.text.x = element_blank())
```

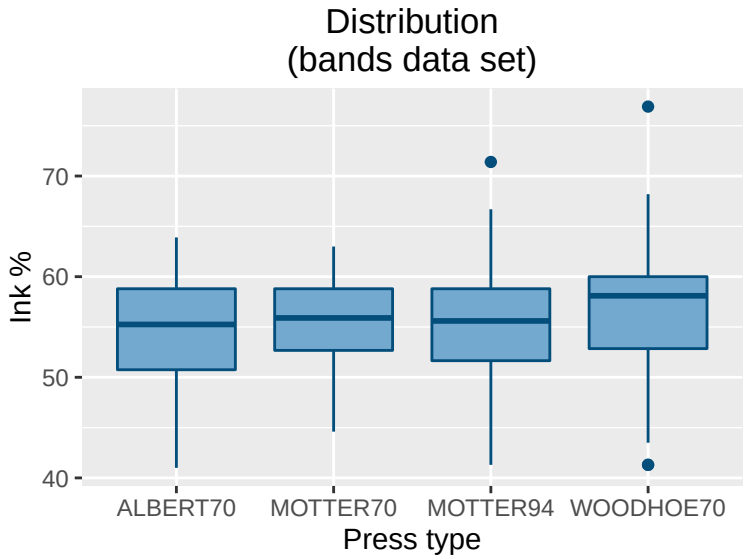


x-axis tick marker and label are removed by using `theme()` function. We remind the functionalities and features of these functions to *Axes Customization* chapter.

6.4 Adding Titles and Axis labels

The last example reveals how to remove the x-axis title. If the name of the variable in axes title are not enough clear, you can change them by using functions `ylab()` to set the y-axis title and `xlab()` to set the x-axis title, while `ggtitle()` should be used when you want a title for the whole plot. Note the use of the escape code `\n` to break lines.

```
ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b") +
  xlab("Press type") +
  ylab ("Ink %") +
  ggtitle("Distribution\n(bands data set)")
```



6.5 Making a Violin Plot

A much more flexible extension of the basic boxplot is the violin plot, constructed by combining the concept of the boxplot with that of nonparametric density estimates. Like box plots, violin plots are used to represent comparison of a variable distribution (or sample distribution) across different categories. A violin plot is more informative than a plain box plot. In fact while a box plot only shows summary statistics such as mean/median and interquartile ranges, the violin plot shows the full distribution of the data.

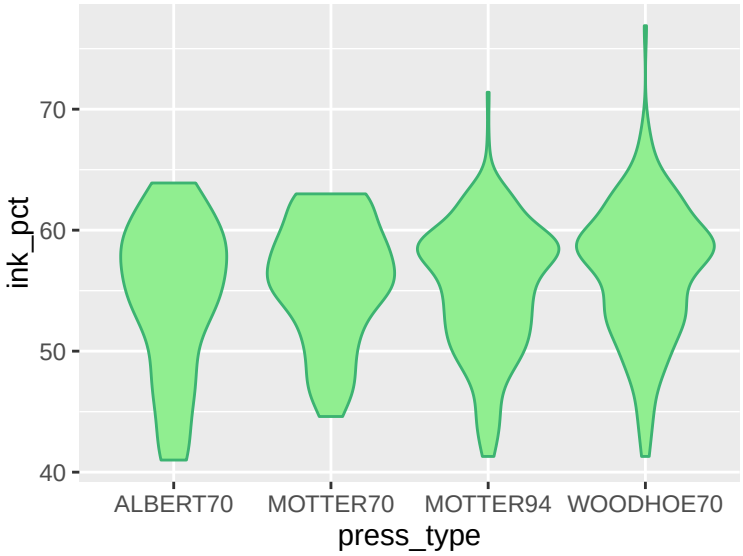
You can see that with ordinary density curves, it is difficult to compare more than just a few distributions because the lines visually interfere with each other.

A solution can be the violin plot, which is a way of comparing multiple density estimates of data distributions. In particular, a violin plot is a kernel density estimate, mirrored so that it forms a symmetrical shape.

As you can see, with a violin plot, it's easier to compare several distributions since they're placed side by side, as happens with a boxplot. –>

Suppose we want to compare the density distribution of `ink_pct` by `press_type`:

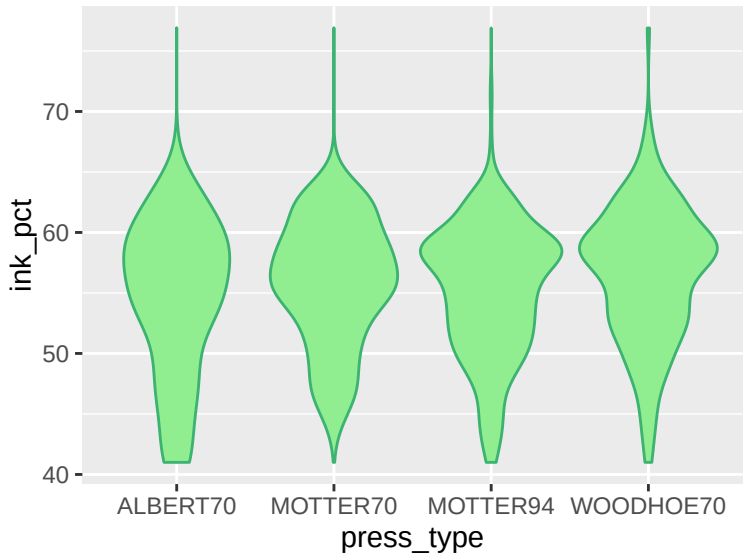
```
ggplot(data=bands, mapping=aes(x = press_type, y=ink_pct)) +
  geom_violin(fill ="lightgreen", colour = "mediumseagreen")
```



`geom_violin()` is the function that draws violin plot. Like for a boxplot, usually the continuous variable is mapped on `y` axis and the categorical variable on `x` axis.

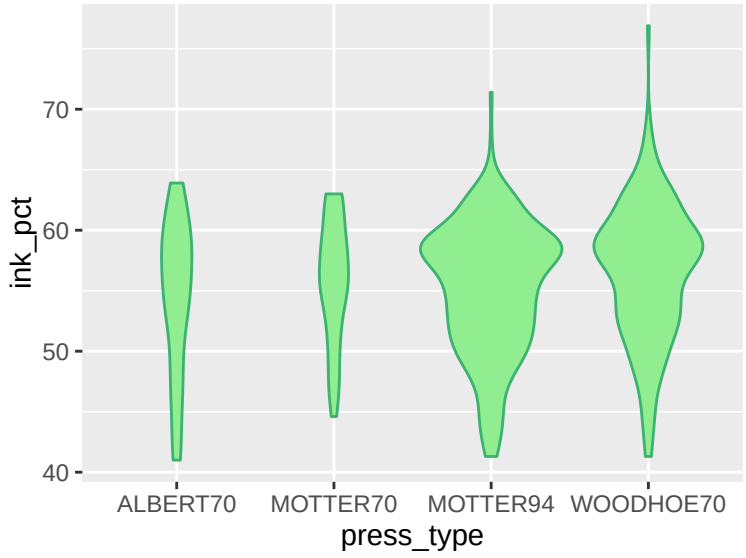
The default range of violin plot goes from the minimum to maximum data values; the flat ends of the violins are at the extremes of the data. It's possible to keep the tails, by setting `trim=FALSE`:

```
ggplot(data=bands, mapping=aes(x = press_type, y=ink_pct)) +
  geom_violin(fill ="lightgreen", colour = "mediumseagreen", trim=FALSE)
```



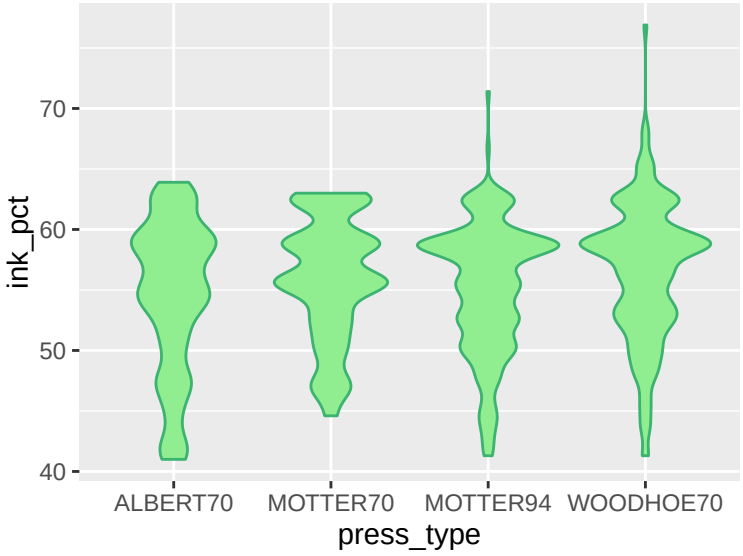
By default, the violins are scaled so that the total area of each one is the same. Instead of equal areas, you can use `scale="count"` to scale the areas proportionally to the number of observations in each group:

```
ggplot(data=bands, mapping=aes(x = press_type, y=ink_pct)) +
   geom_violin(fill ="lightgreen", colour = "mediumseagreen", scale="count")
```



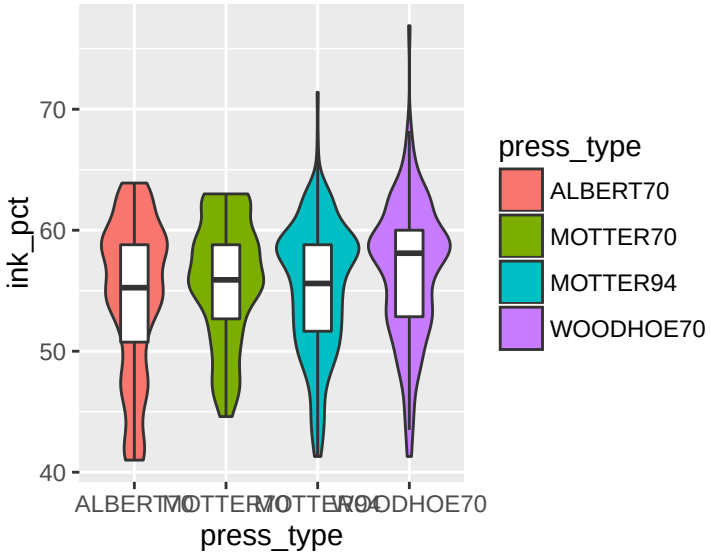
To change the amount of smoothing, use the `adjust` parameter. The default value is 1; use larger values for more smoothing and smaller values for less smoothing:

```
ggplot(data=bands, mapping=aes(x = press_type, y=ink_pct)) +
   geom_violin(fill ="lightgreen", colour = "mediumseagreen", adjust=0.5)
```



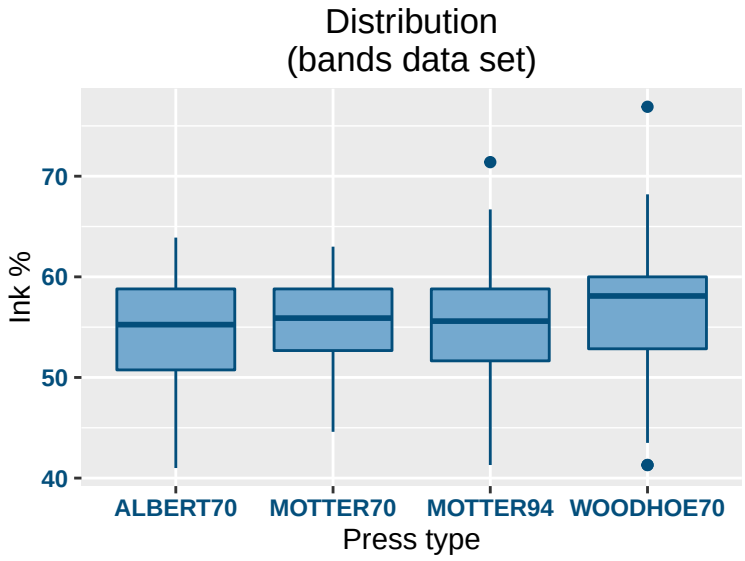
If you also need the median and interquartile range, you can overlay boxplot on violin plot:

```
ggplot(data=bands, mapping=aes(x = press_type, y=ink_pct)) +
geom_violin(aes(fill = press_type), width=0.9, bw=1.5) +
geom_boxplot(width=0.4, outlier.shape = NA)
```



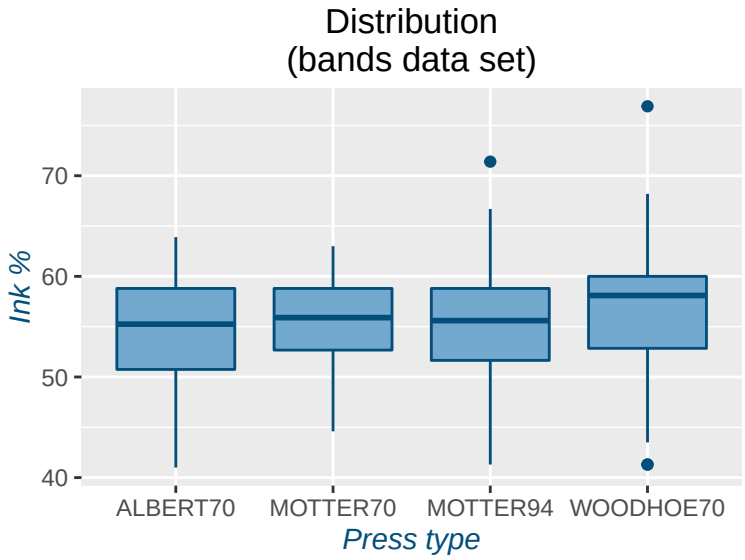
- `axis.text.x`, `axis.text.y` and `axis.text` controls the text of x, y or both axis ticks. You can suppress text with `element_blank()` or use `element_text()` to change the aspect of ticks setting font `family`, font `face`, `colour`, `size`, `angle` and other options.

```
ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b") +
  xlab("Press type") + ylab ("Ink %") + ggtitle("Distribution\n(bands data set)") +
  theme(axis.text = element_text(face="bold", colour="#034e7b"))
```



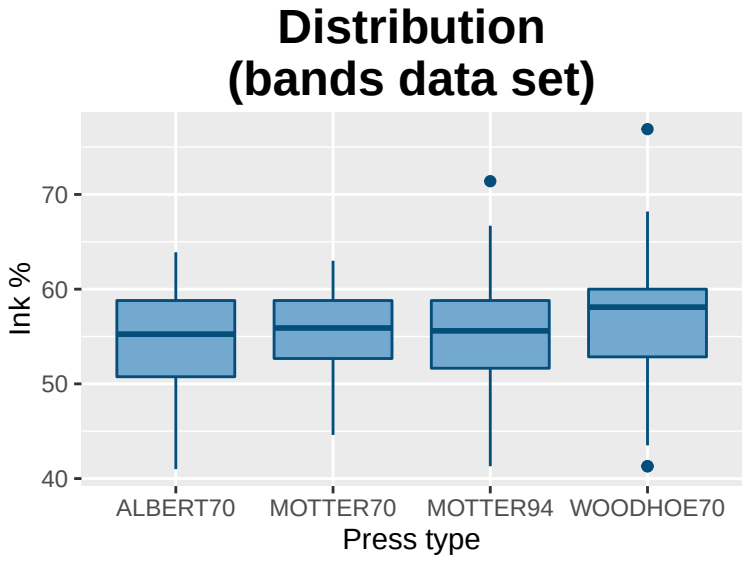
- `axis.text.x`, `axis.text.y` and `axis.text` controls the title of x, y or both axes. You can suppress text with `element_blank()` or use `element_text()` to change the aspect of axis text.

```
ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b") +
  xlab("Press type") + ylab ("Ink %") + ggtitle("Distribution\n(bands data set)") +
  theme(axis.title = element_text(face="italic", colour="#034e7b"))
```



- `plot.title` controls the text of the overall plot. You can use `element_text()` to change the aspect of title.

```
ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b") +
  xlab("Press type") + ylab ("Ink %") + ggtitle("Distribution\n(bands data set)") +
  theme(plot.title = element_text(face="bold", size=18))
```

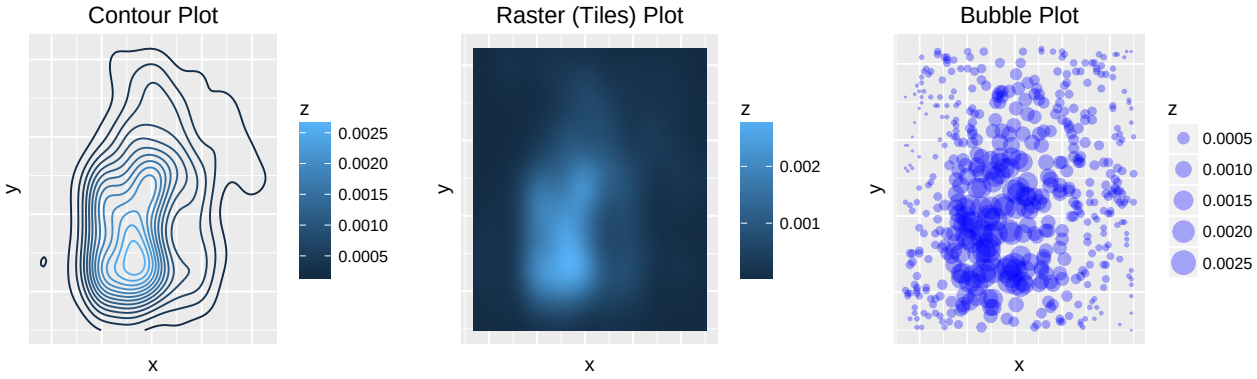


–>

7 Creating a Surface plot

```
require(ggplot2)
require(dplyr)
require(qdata)
require(MASS)
data(bands)
```

ggplot2 does not support true 3d surfaces. However it does support many common tools for representing 3d surfaces in 2d: contours, coloured raster (tiles) and bubble plots. These all works similarly, differing only in the aesthetic used for the third dimension.



This chapter explains how to build contours, coloured raster (tiles) and bubble plots.

7.1 Contour

A contour plot is a graphical technique for representing a 3-dimensional surface by plotting constant z slices, called contours, on a 2-dimensional format. That is, given a value for z, lines are drawn for connecting the (x,y) coordinates where that z value occurs.

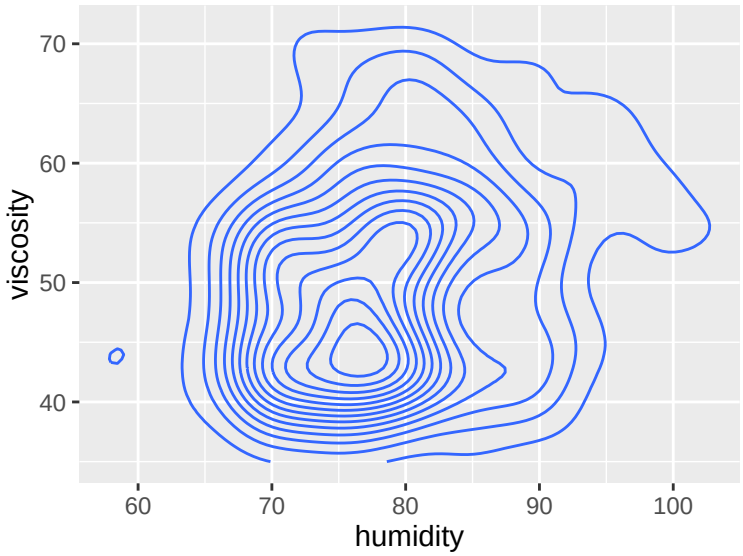
Suppose we want to analize the relationship between `humidity` and `viscosity` variables of `bands` dataset considering also the bivariate density estimate between these variables.

First of all let us compute bivariate density estimate between `humidity` and `viscosity`:

```
# remove NA from bands dataset
bands_na_rm <- bands %>% na.omit()

# Compute bivariate density estimate
f2d <- kde2d(bands_na_rm$humidity, bands_na_rm$viscosity, n =100)

# Generate a new dataset including also the newly created variable
bands_d <- expand.grid(humidity = f2d$x, viscosity = f2d$y) %>%
  tbl_df() %>%
  mutate(density = as.vector(f2d$z))
```

Contour can be plotted by `geom_contour()` function:

```
ggplot(bands_d, aes(humidity, viscosity, z = density)) +
  geom_contour()
```

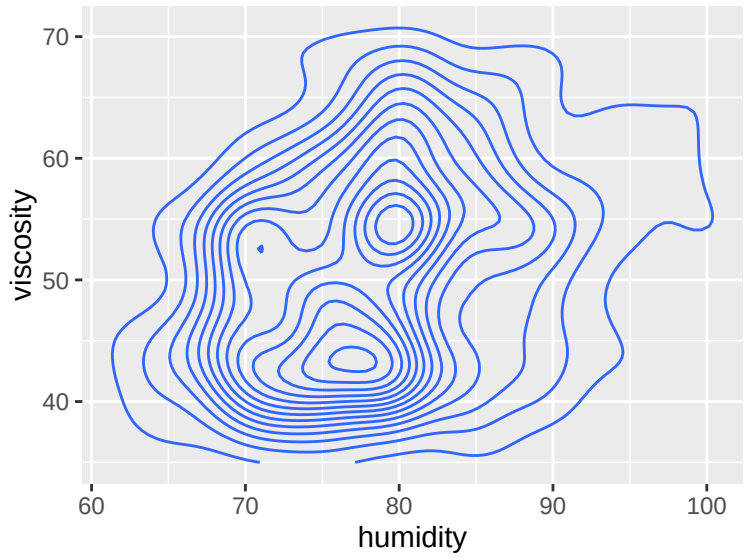


The third dimension is mapped to `z` aestethic of `ggplot()` function. In this case, `z` is set to `density`.

A similar plot can be represented also by using `geom_density_2d()` function:

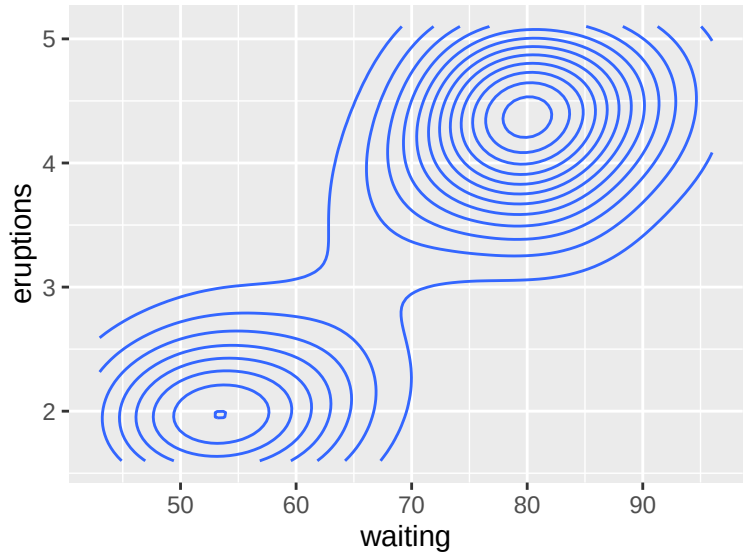
```
ggplot(bands, aes(humidity, viscosity)) +
  geom_density_2d()
```

```
## Warning: Removed 6 rows containing non-finite values (stat_density2d).
```



The difference between `geom_contour()` and `geom_density_2d()` is that `geom_density_2d()` doesn't need the definition of the third dimension as it automathically computes a two-dimensional kernel density estimate and countour is its default representation.

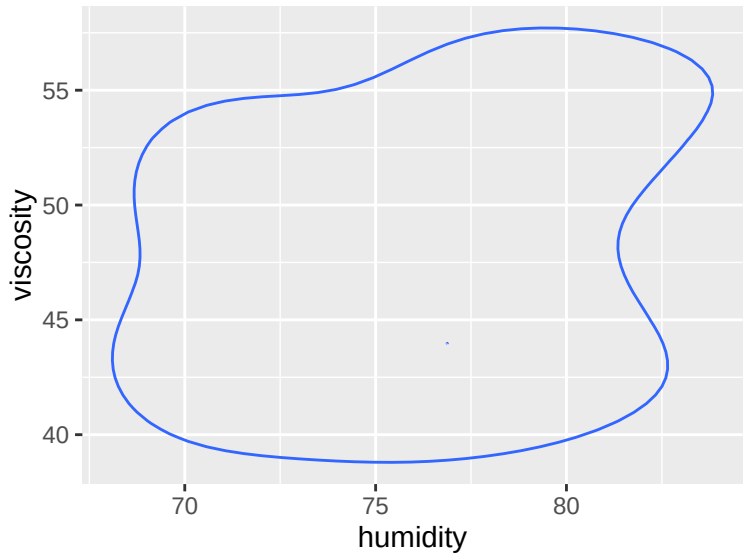
```
ggplot(faithful, aes(waiting, eruptions)) +
  geom_density_2d()
```



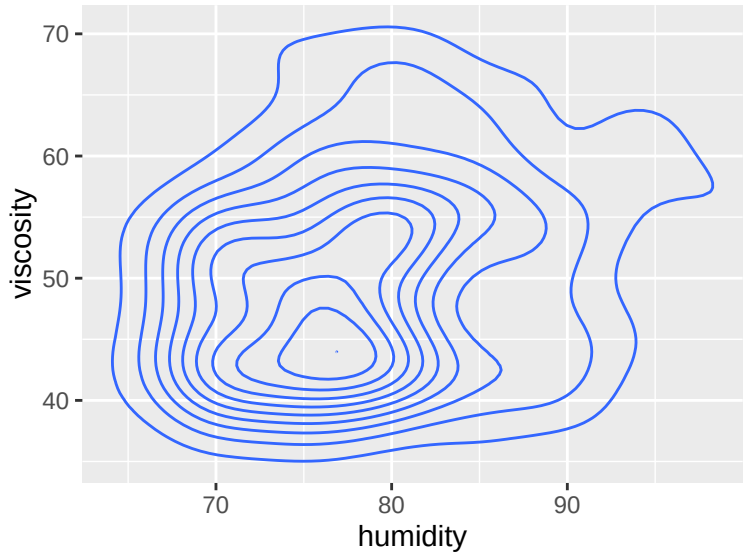
–>

You can set bins, by using `bins` argument, to generate evenly spaced contours in the range of the data:

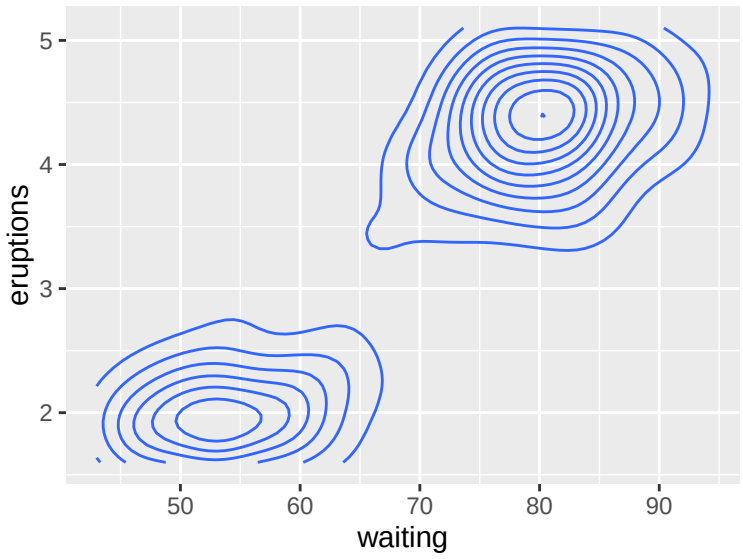
```
ggplot(bands_d, aes(humidity, viscosity, z = density)) +
  geom_contour(bins = 2)
```




```
ggplot(bands_d, aes(humidity, viscosity, z = density)) +
  geom_contour(bins = 10)
```



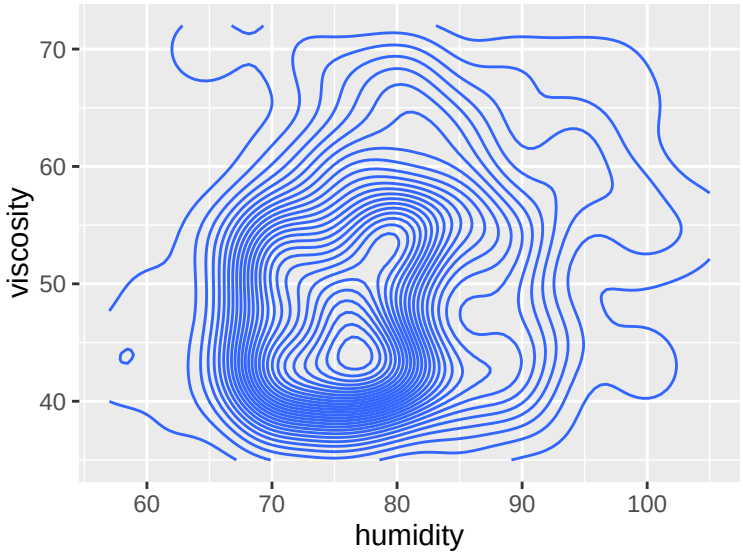

```
ggplot(faithfuld, aes(waiting, eruptions, z = density)) +
 geom_contour(bins = 10)
```



–>

You can also parameterize the distance between contours setting `binwidth` argument, which represent the binwidth of countour lines:

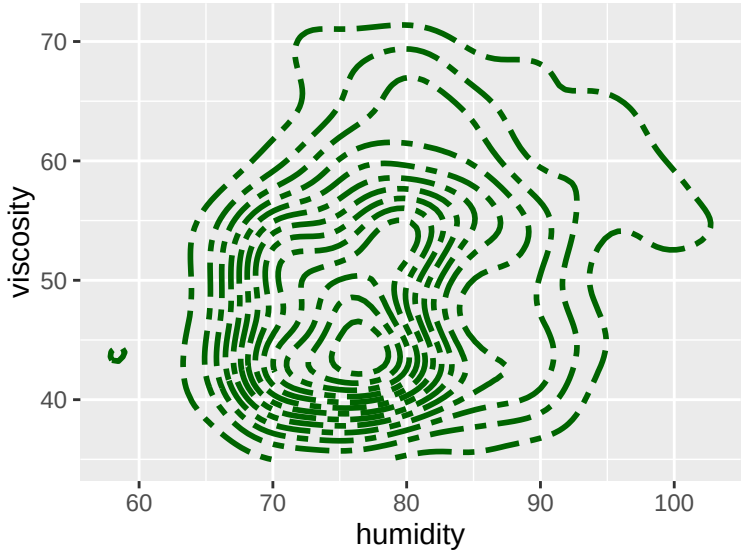
```
ggplot(bands_d, aes(humidity, viscosity, z = density)) +
  geom_contour(binwidth = 0.0001)
```



–>

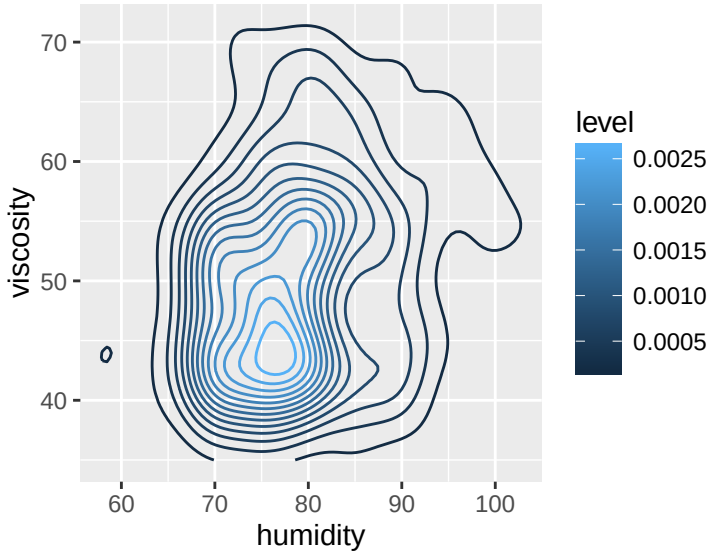
We can customize the countour plot by changing `colour`, `linetype` or line `size`:

```
ggplot(bands_d, aes(humidity, viscosity, z = density)) +
  geom_contour(colour = "darkgreen", linetype = 6, size = 1)
```



It is also possible to map the height of the density curve to the color of the contour lines, by mapping `..level..` to `colour` scale:

```
ggplot(bands_d, aes(humidity, viscosity, z = density)) +
  geom_contour(aes(colour = ..level..))
```



`..level..` is a variables generated by the statistical transformation used by `geom_contour()` function.

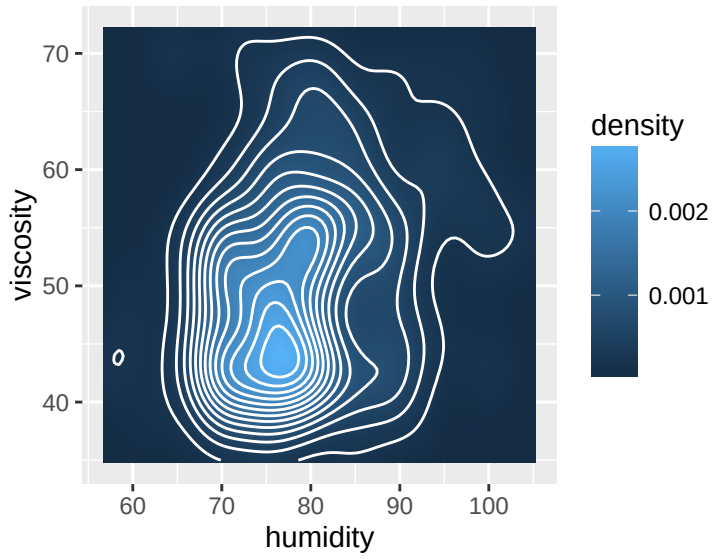
Mapping `..level..` to `colour` a legend is automathically produced.

–>

–>

You can also customize your plot by adding in background another `geom` which draws surfaces: `geom_raster()` in order to increase the 3d effect.

```
ggplot(bands_d, aes(humidity, viscosity, z = density)) +
  geom_raster(aes(fill = density)) +
  geom_contour(colour = "white")
```



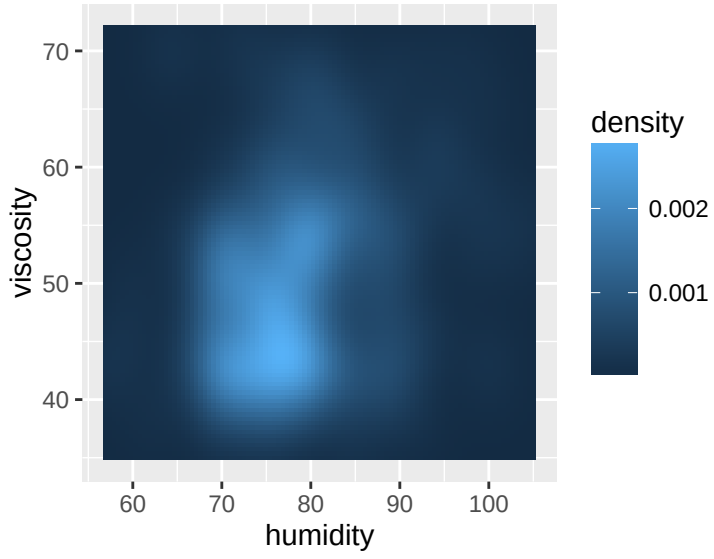
–>

7.2 Raster (Tiles) Plot

A raster (tiles) plot is a scan pattern in which an area is scanned from side to side in lines from top to bottom. It can be defined also as a pattern of closely spaced rows of dots that form an image.

As you saw in the previous example, raster plot is generated by `geom_raster()`.
`geom_raster()` is a function for drawing rectangles. The most common use for rectangles is to draw a surface. If you want to draw surfaces you have to map the variable that represent the third dimension, in this case `density`, to `fill` aestethic:

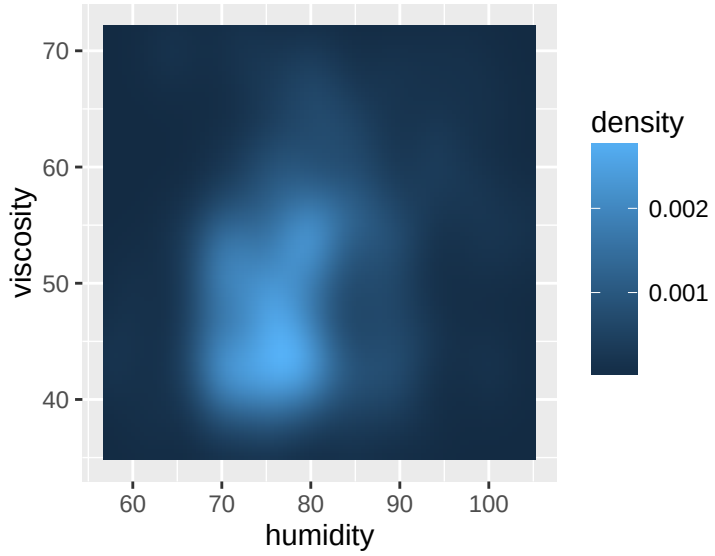
```
ggplot(bands_d, aes(humidity, viscosity)) +
   geom_raster(aes(fill = density))
```



–>

You can also add an interpolation to smooth the surface, by setting `interpolate = TRUE`. It is useful when rendering images.

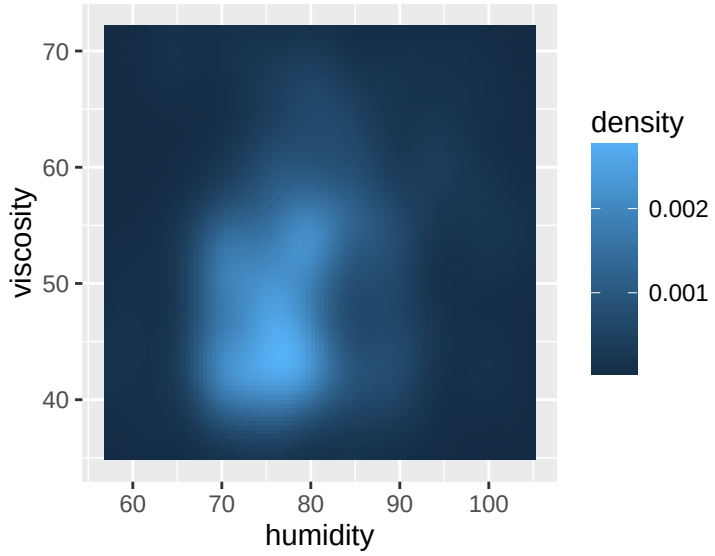
```
ggplot(bands_d, aes(humidity, viscosity)) +
   geom_raster(aes(fill = density), interpolate = TRUE)
```



–>

A similar result can be achieved by using `geom_tile()` function. `geom_tile()` is similar to `geom_raster()` as both draw rectangles. The main difference is that `geom_raster()` renders more efficiently than `geom_tile()`. In theory they should appear the same, but in practice they often do not. If you are writing to a PDF file, the appearance depends on the PDF viewer. On some viewers, when tile is used there may be faint lines between the tiles, and when raster is used the edges of the tiles may appear blurry.

```
ggplot(bands_d, aes(humidity, viscosity)) +
   geom_tile(aes(fill = density))
```



–>

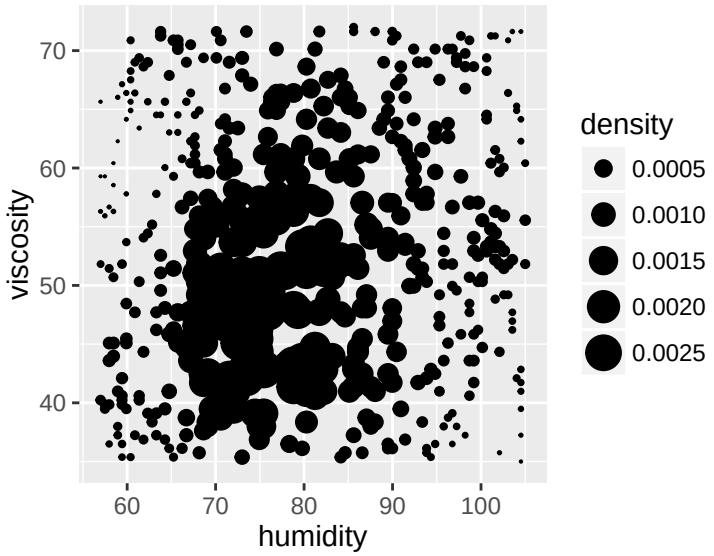
7.3 Bubble Plot

A bubble plot is another type of plot that displays three dimensions of data. Each entity with its triplet (x, y, z) of associated data is plotted as a disk that expresses x and y through the disk's xy location and z through its size. Bubble charts can be considered a variation of the scatter plot, in which the data points are replaced with bubbles.

It works better with fewer observations so we grab a sample of the original dataset:

```
bands_d_sample <- bands_d %>% sample_n(size = 500)
```

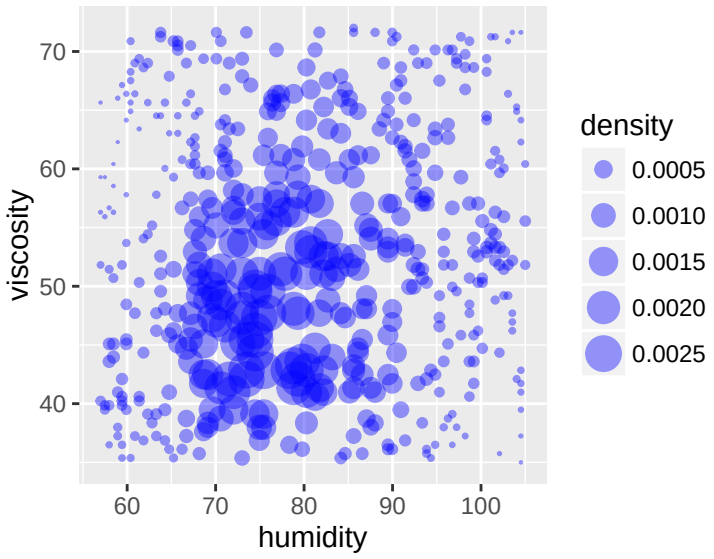
```
ggplot(bands_d_sample, aes(humidity, viscosity)) +
  geom_point(aes(size = density)) +
  scale_size_area()
```



To generate a bubble plot you have to map the third variable, in this case `density` to `size` aestethic of `geom_point()` function. Then you have to add `scale_size_area()` function to render the area of points proportional to `density`, this means that `scale_size_area()` ensures that a value of 0 is mapped to a size of 0.

You can also specify an `alpha` level, a `colour` and `fill` for bubbles as `geom_point()` settings:

```
ggplot(bands_d_sample, aes(humidity, viscosity)) +
  geom_point(aes(size = density), alpha = 0.4, colour = "blue", fill = "lightblue") +
  scale_size_area()
```



8 Mapped Aesthetics Customization

```
require(dplyr)
require(ggplot2)
require(qdata)
data(bands)
```

You can change the appearance of mapped aesthetics by using scales.
Scales control the mapping from data to aestethics. They take your data and turn into something that you can see like colour, shape, position or size. Scales provide also the tools that allow you to read the plot: legends and axes.

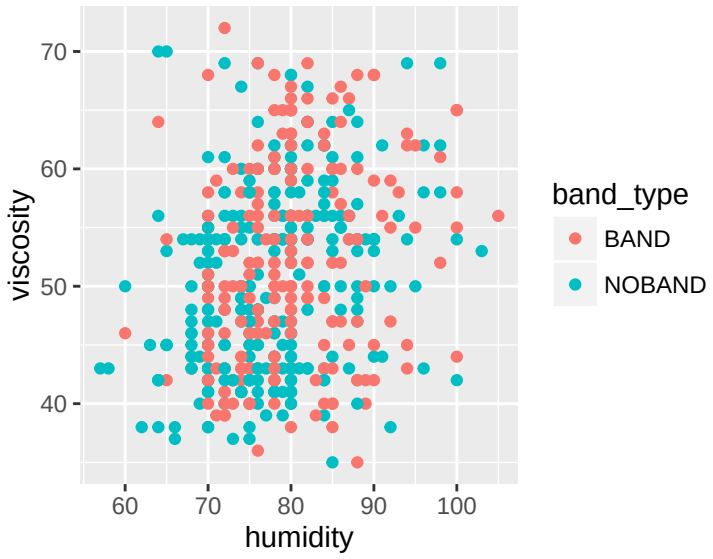
In this chapter we will discuss how customize the appearance of mapped aesthetics by exploring scales functionalities. Legends and axes will not discussed here but in *Axes Customization* and *Legend Customization* chapters, respectively.

But firstly, let us introduce some concepts about scales structure.

8.1 Scales Structure

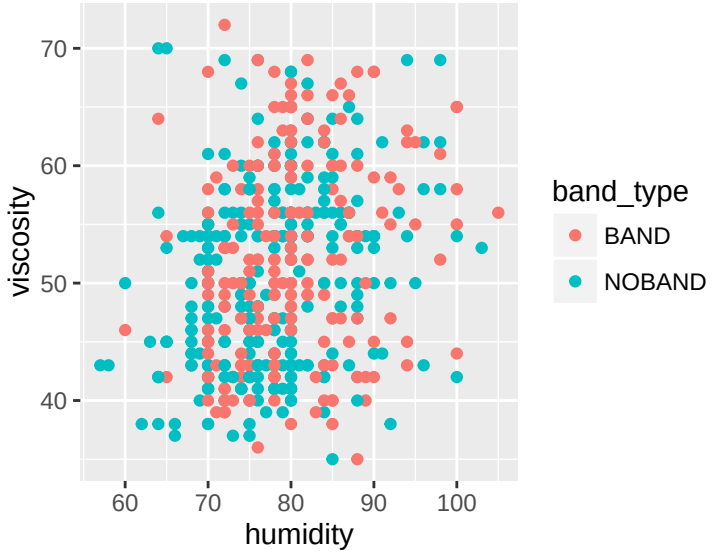
A scale is required for every aestethic used on a plot. For example, when you write:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(mapping = aes(colour=band_type))
```



What actually happens is:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(mapping = aes(colour=band_type)) +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_color_discrete()
```



The previous two specifications are equal.

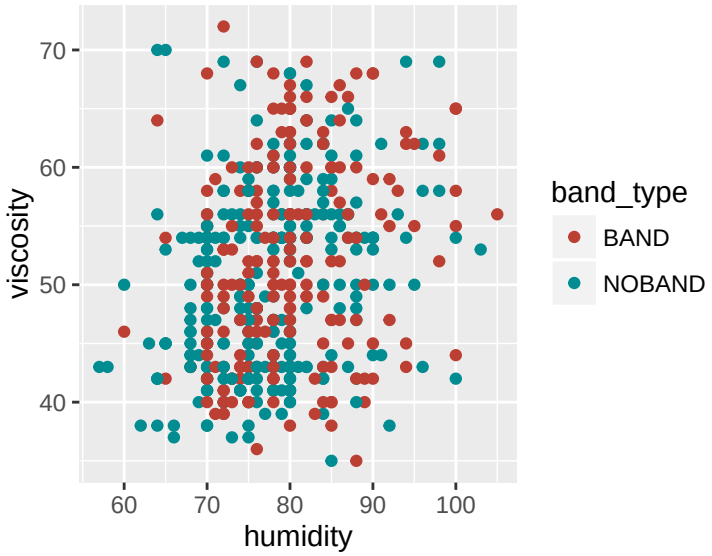
Default scales are named according to the aesthetics and the variables type. It would be tedious to manually add a scale every time a new aestethic is added, so `ggplot2` does it for you.

Each scale has default values for its argument, this means that if you want to modify one or more features of this scale you have to overwrite it/s default value. To overwrite its default value/s you have to manually add that scale.

For example, let us suppose that you want to reduce the luminance of the points colours in the plot. `colour` aesthetic mapping is controlled by `scale_color_discrete()` and the argument of `scale_color_discrete()` that control the luminance is `l`. `l` can assume values from 0 to 100 and its default value is 65. We want to

decrease the luminance and set it to 40 in this way:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity)) +
  geom_point(mapping = aes(colour=band_type)) +
  scale_color_discrete(l = 45)
```



The use of `+` to "add" a scale is a little misleading. Remember that when you `+` a scale you are not adding it but ovverwriting it.

8.1.1 Scales naming scheme and scales types

The name for scaling scheme is made up by three pieces separated by `_`:

1. `scale`
2. the name of the aestethic (e.g., `colour`, `shape`, ...)
3. the name of the scale (e.g., `continuous`, `discrete`, `brewer`, ...)

There are more scale types. They can be grouped into:

- Continuous position scales, used to map data into `x` and `y` position
- Mapped aestethics scales, like colour, linetype, shape and size.

In the following paragraphs we will answer to the most frequently asked questions about mapped aesthetics scales customization for: colours, linetype, shape and size. The use and the features of continuous position scales will be discussed in *Axes Customization* chapter.

8.2 Customizing colours

Do you know something about colour theory? If the answer is no, you should read this paragraph.

At the physical level, colour is produced by a mixture of wavelenghts of lights. To characterise a colour completely we need to know the complete mixture of wavelenghts. Human eye has only three different colour receptors, so the perception of any colour can be summarised with just three numbers. You may be familiar with the RGB encoding of colour space, which defines a colour by the intensities of red, green and blue light needed to produce it. One problem with this space is that it is not percentually uniform and this make it difficult to create a mapping from a continuous variable to a set of colours. There have been many attemps to come up with colours spaces that are more percentually uniform.

We use a modern attemp called hcl colour space, which has three components of **h** (hue), **c** (chroma) and **l** (luminance):

- hue is a number between 0 and 360 (an angle) which gives the "colour" of the colour: like blue, red, orange, etc.
- chroma is the purity of a coloyr. A chroma of 0 is grey, and the maximum value of chroma varies with luminance
- luminance is the lightness of the colour. A luminance of 0 produces black, and a lumimance of 1 produces white

Hues are not perceived as being ordered, but chroma and luminance are ordered.

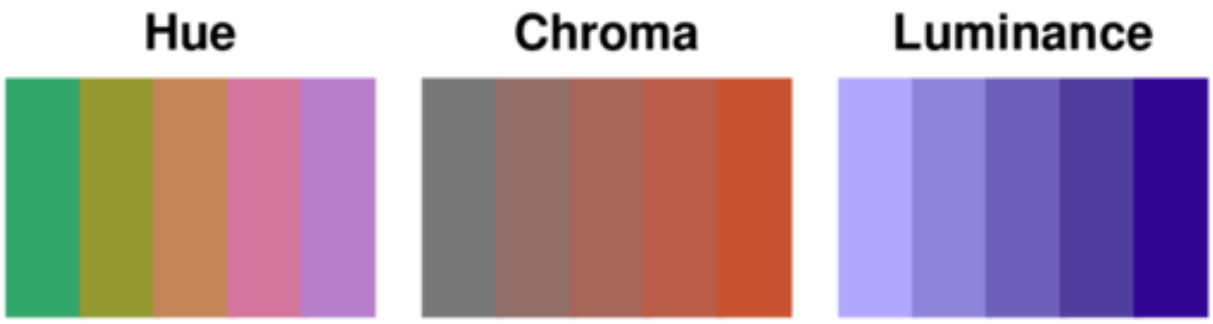
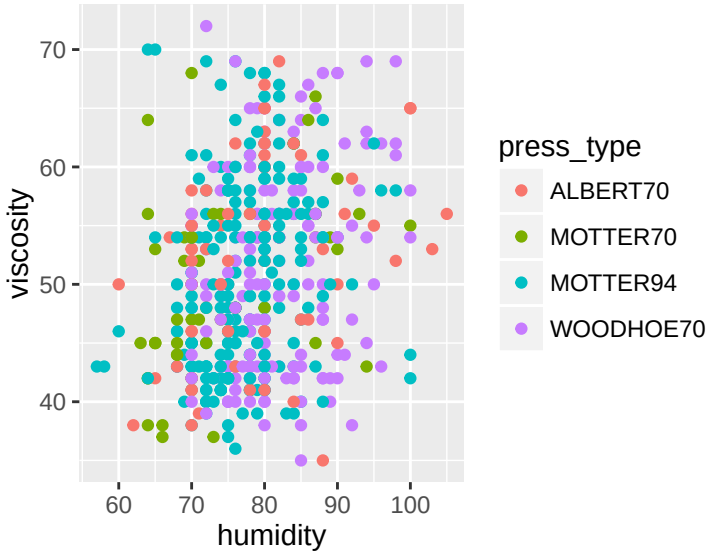


Figure 9:

8.2.1 Change the colours for a discrete mapped variable

Suppose we are interested in the relationship between `humidity` and `viscosity` by `press_type` in `bands` dataset:

```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point()
```



You want to use different colour for `press_type` levels so you want to change the default palette. A palette is a range of colours; so changing a palette means to modify the `colour` (or `fill`) scale: it involves a change in the mapping from numeric or categorical values to aesthetic attributes.

There are two types of scales that use colors: fill scales and color scales.
You can change the default colours used in the graph by choosing a function from the following table:

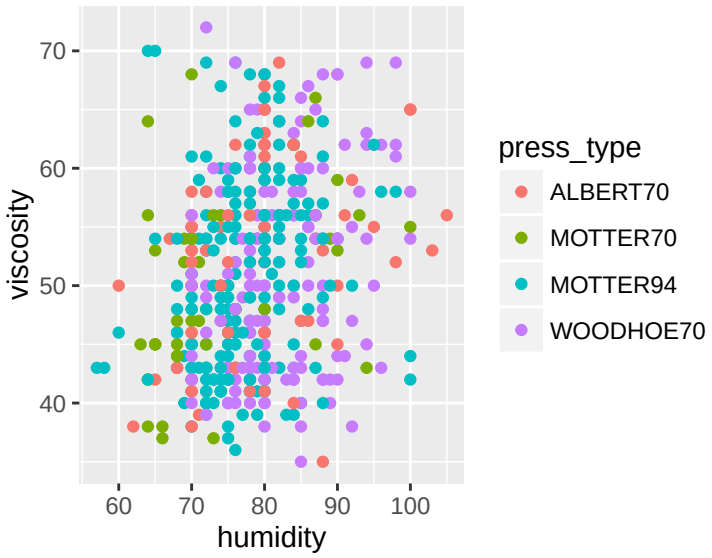
Fill scale	Colour scale	Description
`scale_fill_discrete()`	`scale_colour_discrete()`	Colors evenly spaced around the color wheel (same as hue)
`scale_fill_hue()`	`scale_colour_hue()`	Colors evenly spaced around the color wheel (same as discrete)
`scale_fill_grey()`	`scale_colour_grey()`	Grayscale palette
`scale_fill_brewer()`	`scale_colour_brewer()`	ColorBrewer palettes
`scale_fill_manual()`	`scale_colour_manual()`	Manually specified colors

By default, the colors for discrete scales are evenly spaced around a HSL color circle.

For example, if there are two colors, then they will be selected from opposite points on the circle; if there are three colors, they will be 120° apart on the color circle; and so on. The default colors used for different numbers of levels are shown here:

The default color selection uses `scale_fill_hue()` and `scale_colour_hue()`:

```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point() +
  scale_color_hue()
```



Note: in the previous plot definition `scale_color_hue()` is redundant.

With `scale_colour_hue()`, the colors are taken from around the color wheel in the HCL (hue-chroma-lightness) color space. In *Scales structure* we learn that the default lightness value is 65 on a scale from 0–100. This is good for filled areas, but it's a bit light for points and lines. To make the colors darker for points and lines, set the value of `l` (luminance/ lightness):

```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point() +
  scale_colour_hue(l=45)
```

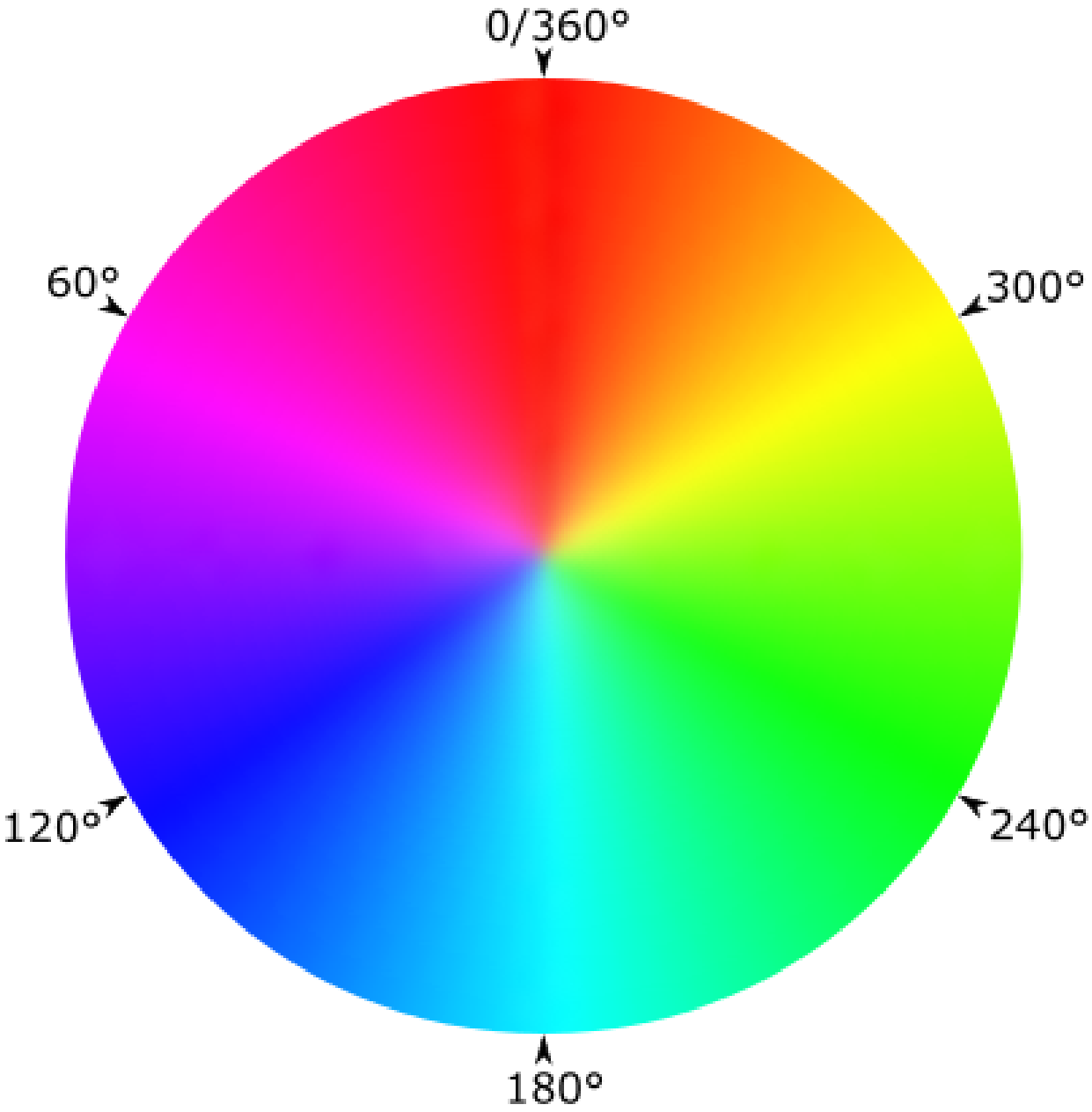


Figure 10:

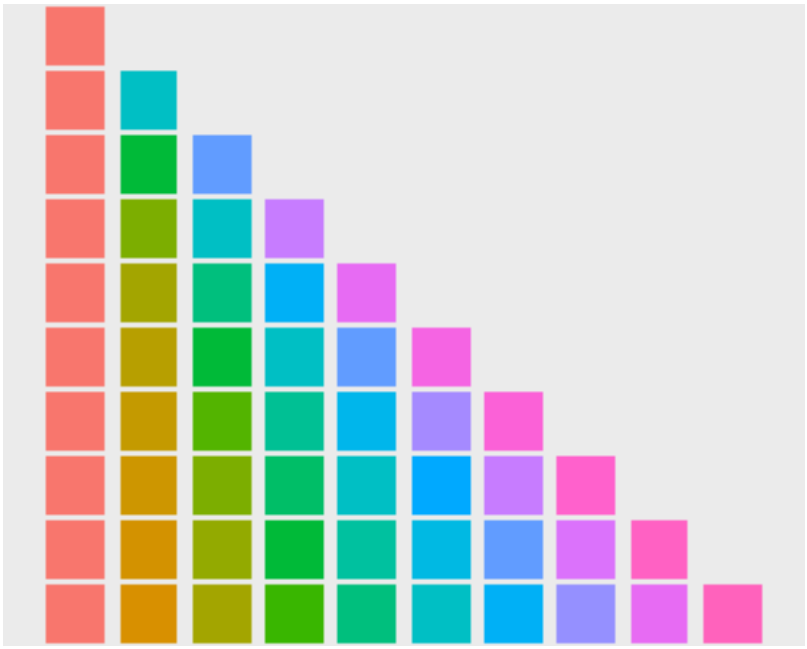
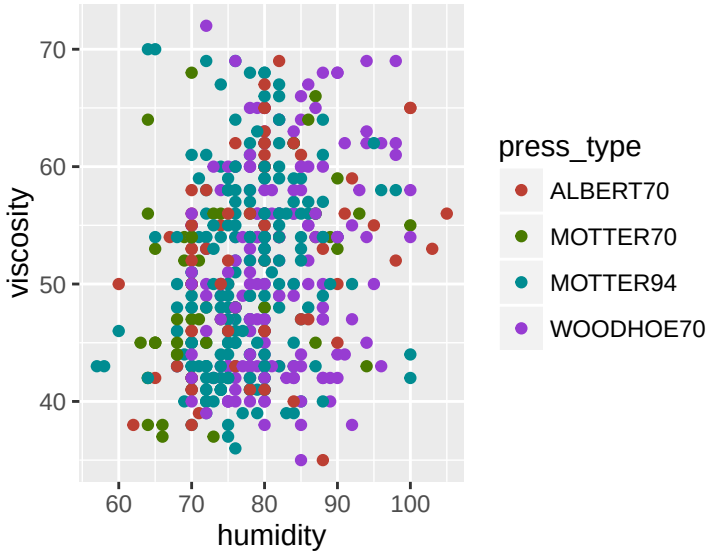


Figure 11:



You can also use other color scales, such as ones taken from the `RColorBrewer` package:

The `ColorBrewer` palettes can be selected by name:

```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point() +
  scale_colour_brewer(palette="Set1")
```

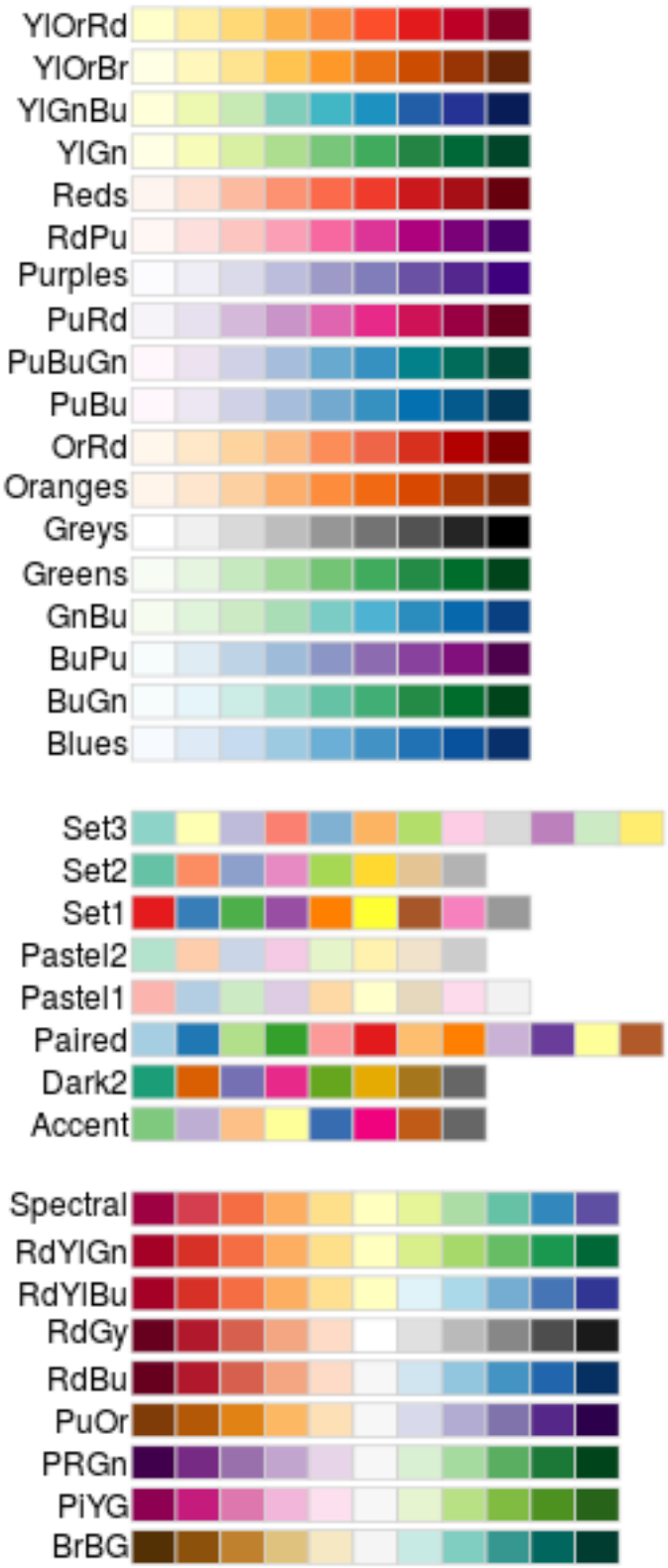
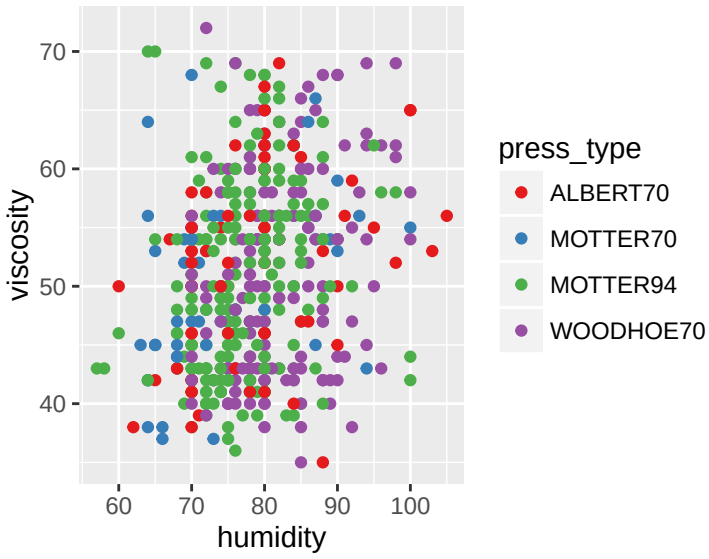
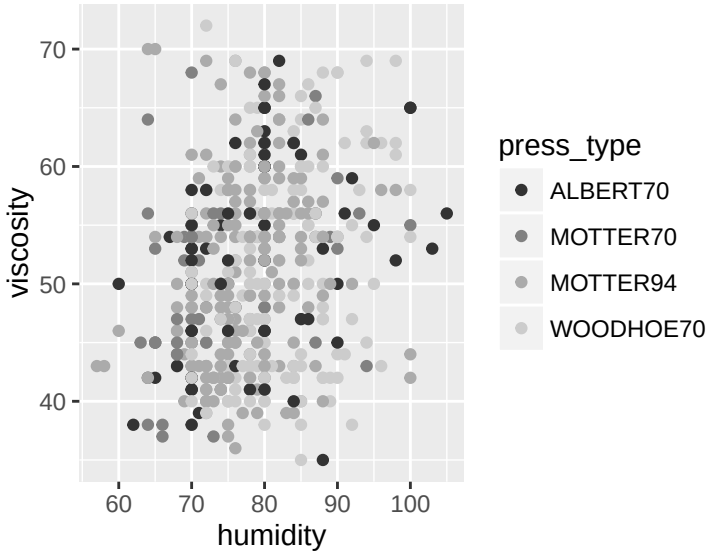

YlOrRd
YlOrBr
YlGnBu
YlGn
Reds
RdPu
Purples
PuRd
PuBuGn
PuBu
OrRd
Oranges
Greys
Greens
GnBu
BuPu
BuGn
Blues
Set3
Set2
Set1
Pastel2
Pastel1
Paired
Dark2
Accent
Spectral
RdYlGn
RdYlBu
RdGy
RdBu
PuOr
PRGn
PiYG
BrBG

Figure 12:



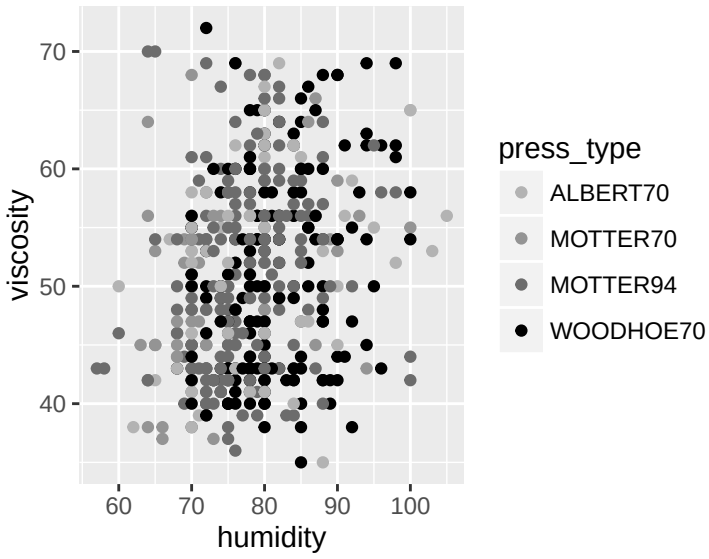
You can also use a palette of greys. This is useful for print when the output is in black and white.

```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point() +
  scale_colour_grey()
```



The default is to start at 0.2 and end at 0.8, on a scale from 0 (black) to 1 (white), but you can change the range:

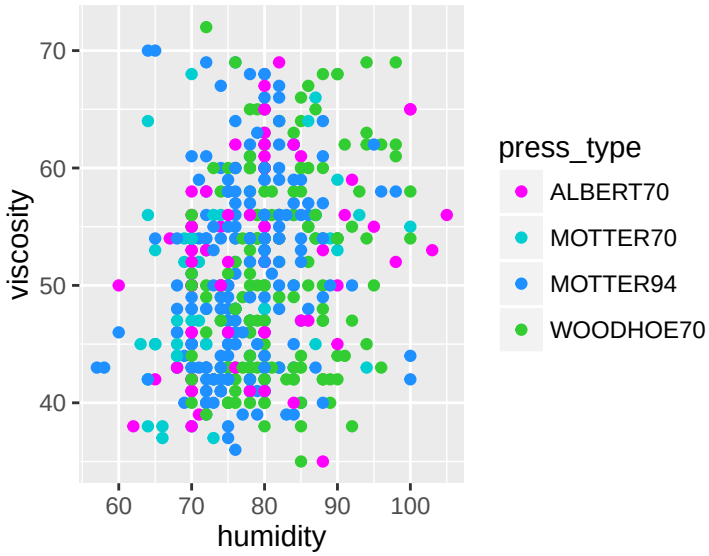
```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point() +
  scale_colour_grey(start=0.7, end=0)
```



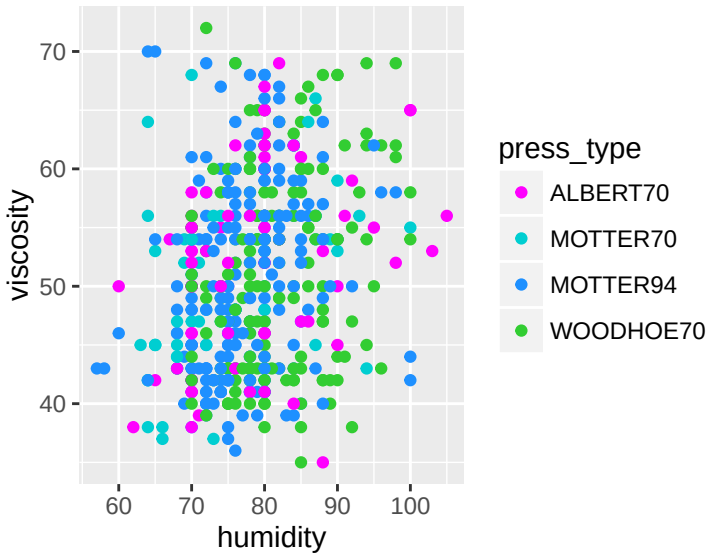
8.2.2 Using a manually defined palette for a discrete variable

Colour can be also manually specified by using `scale_colour_manual()` function. The colors can be named, or they can be specified with RGB values:

```
# named colours
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point() +
  scale_colour_manual(values = c("magenta", "dark turquoise", "dodger blue", "lime green"))
```




```
# RGB colour code
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point() +
  scale_colour_manual(values = c("#FF00FF", "#00CED1", "#1E90FF", "#32CD32"))
```



The order of the items in the values vector matches the order of the factor levels for the discrete scale.

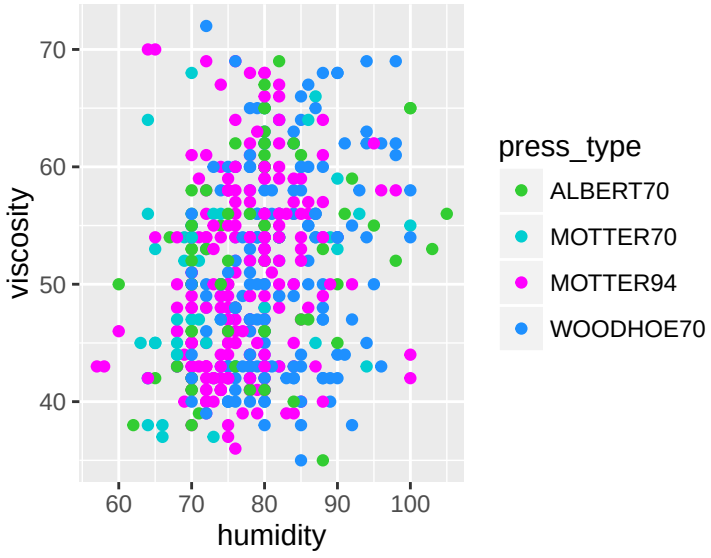
```
levels(bands$press_type)
```

```
## [1] "ALBERT70"  "MOTTER70"  "MOTTER94"  "WOODHOE70"
```

In the preceding example, the order of `press_type` is "ALBERT70", then "MOTTER70", then "MOTTER94" and "WOODHOE70", so the first in `values` argument goes with "ALBERT70", the second goes with "MOTTER70", the third with "MOTTER94" and the fourth with "WOODHOE70". If the variable is a character vector, not a factor, it will automatically be converted to a factor, and by default the levels will appear in alphabetical order.

It's possible to specify the colors in a different order by using a named vector:

```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = press_type)) +
  geom_point() +
  scale_colour_manual(values = c(ALBERT70 = "lime green", MOTTER70 = "dark turquoise", MOTTER94 = "mage
```



There is a large set of named colors in R, which you can see by running `colors()`. Some basic color names are useful: `"white"`, `"black"`, `"grey80"`, `"red"`, `"blue"`, `"darkred"`, and so on. There are many other named colors, but their names are generally not very informative, for example `"thistle3"` and `"seashell"`, so it is

often easier to use numeric RGB values for specifying colors.
RGB colors are specified as six-digit hexadecimal (base-16) numbers of the form `"#RRGGBB"`. In hexadecimal, the digits go from 0 to 9, and then continue with A (10 in base 10) to F (15 in base 10). Each color is represented by two digits and can range from 00 to FF (255 in base 10). So, for example, the color `"#FF0099"` has a value of 255 for red, 0 for green, and 153 for blue, resulting in a shade of magenta. The hexadecimal numbers for each color channel often repeat the same digit because it makes them a little easier to read, and because the precise value of the second digit has a relatively insignificant effect on appearance.

Here are some rules of thumb for specifying and adjusting RGB colors:

- higher numbers are brighter and lower numbers are darker
- to get a shade of grey, set all the channels to the same value

8.2.3 Change the colours for a continuous mapped variable

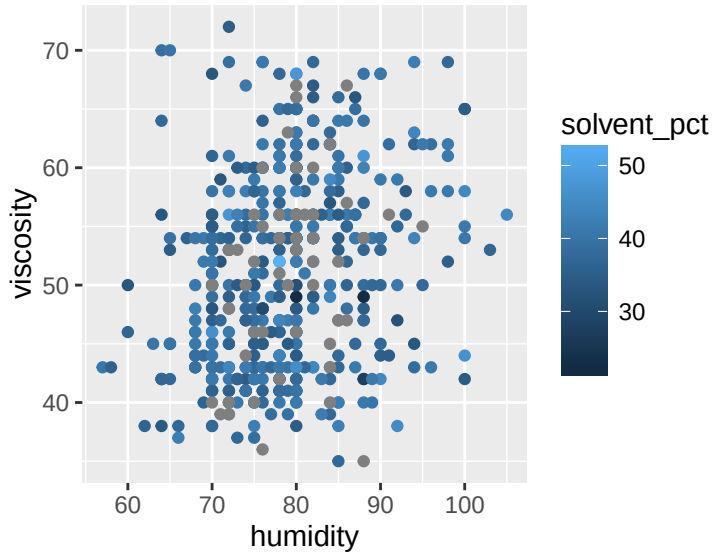
You want to use different colors for a continuous variable.

Mapping continuous values to a color scale requires a continuously changing palette of colors.
The following table lists the continuous color and fill scales:

Fill scale	Colour scale	Description
`scale_fill_gradient()`	`scale_colour_gradient()`	Two-color gradient
`scale_fill_gradient2()`	`scale_colour_gradient2()`	Gradient with a middle color and two colors that diverge from it
`scale_fill_gradientn()`	`scale_colour_gradientn()`	Gradient with n colors, equally spaced

Supposing we are interested in the relationship between `humidity`, `viscosity` and `solvent_pct` in `bands` dataset:

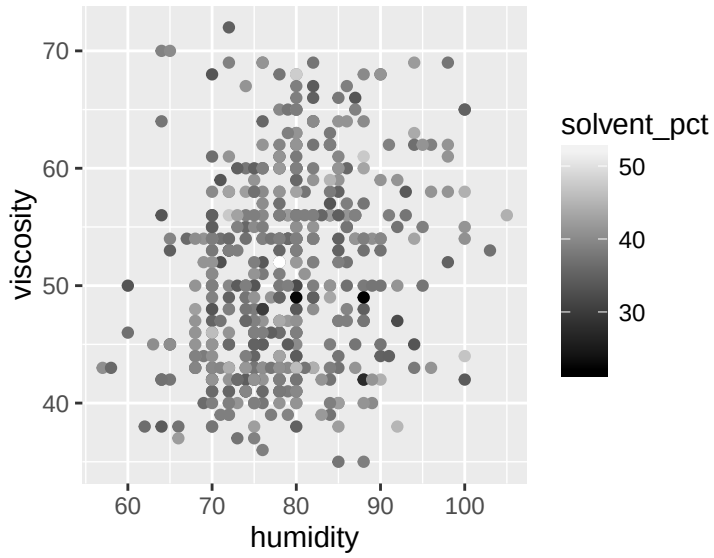
```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = solvent_pct)) +
  geom_point()
```



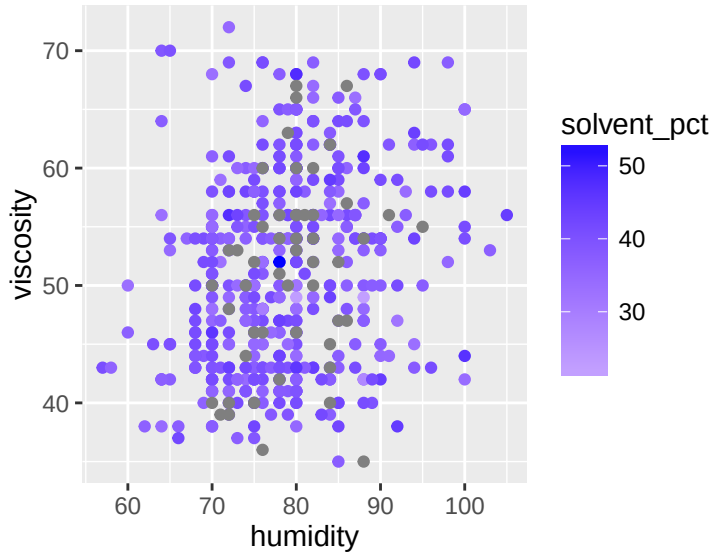
Let us see how the previous fuctions work. We will specify the colours for a continuous variable using various gradient scales. The colors can be named, or they can be specified with RGB values:

```
# scale_colour_gradient()
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = solvent_pct)) +
```

```
  geom_point() +
  scale_colour_gradient(low="black", high="white")
```

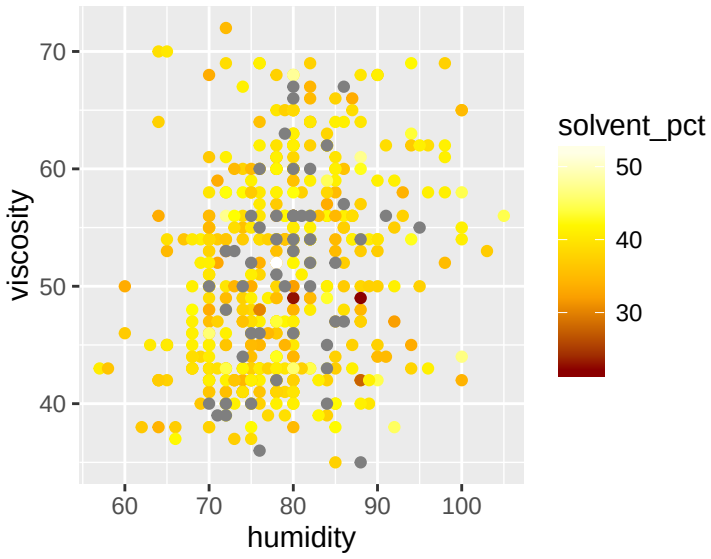



```
# scale_colour_gradient2()
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = solvent_pct)) +
  geom_point() +
  scale_colour_gradient2(low="red", mid="white", high="blue")
```



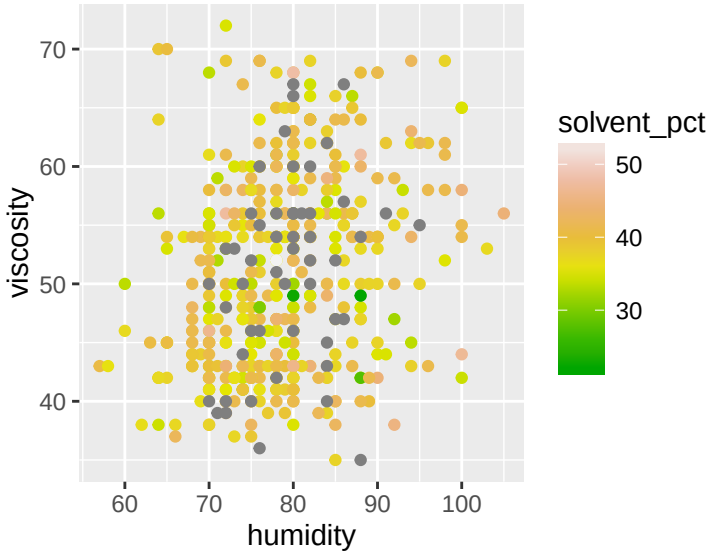
->

```
# scale_colour_gradientn()
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = solvent_pct)) +
  geom_point() +
  scale_colour_gradientn(colours = c("darkred", "orange", "yellow", "white"))
```



You can also use palettes produced by another package:

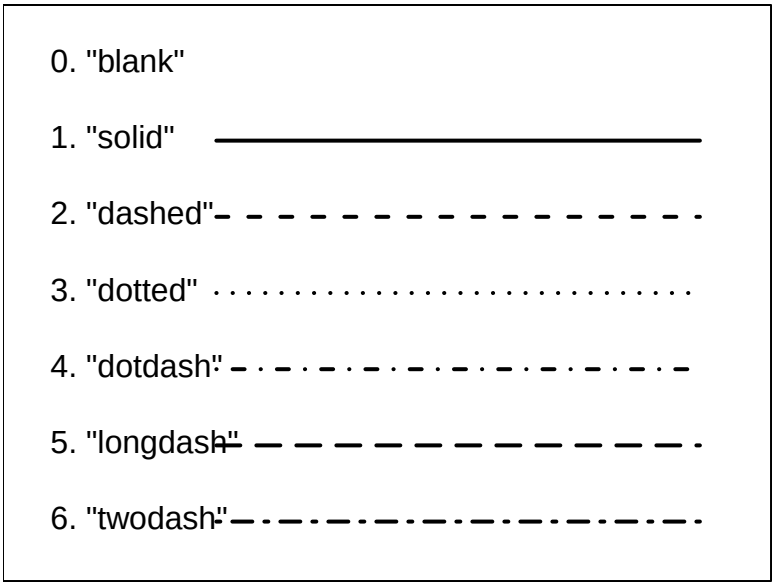
```
ggplot(data = bands, mapping = aes(x = humidity, y = viscosity, colour = solvent_pct)) +
  geom_point() +
  scale_colour_gradientn(colours = terrain.colors(n=8))
```



In this case we use `terrain.colors` palette with 8 colours. `terrain.colors` is a palette of `grDevices` package.

8.3 Customizing linetype

R provides the following line types:



`linetype` aesthetic has to be mapped to discrete variables.

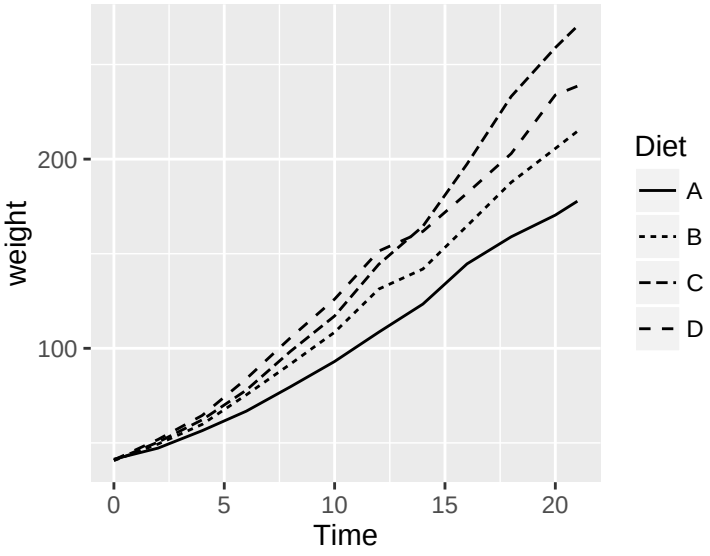
8.3.1 Changing line types

Let us consider the average growth of chicks for the different diets by using `ChickWeights` dataset, as seen in *Creating a Line Plot*:

```
# data
ChickWeightMean <- ChickWeight %>%
  group_by(Time, Diet) %>%
  summarize(weight=mean(weight)) %>%
  mutate(Diet = factor(Diet, levels=1:4, labels=c("A","B","C","D")))
```

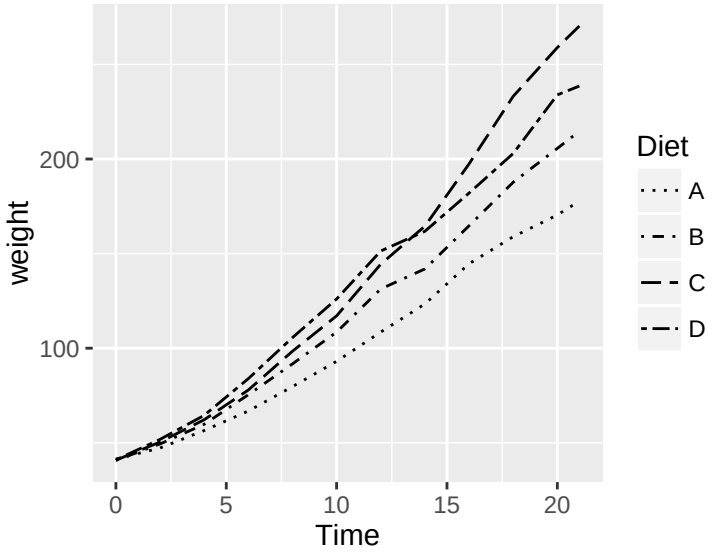
We want to se the difference in the average growth of chicks for the different diets, by mapping `Diet` variable to `linetype` aestethics:

```
ggplot(data=ChickWeightMean, mapping=aes(x=Time, y=weight)) +
  geom_line(mapping=aes(linetype=Diet))
```



To change linetypes we have to set `values` argument of `scale_linetype_manual()` function:

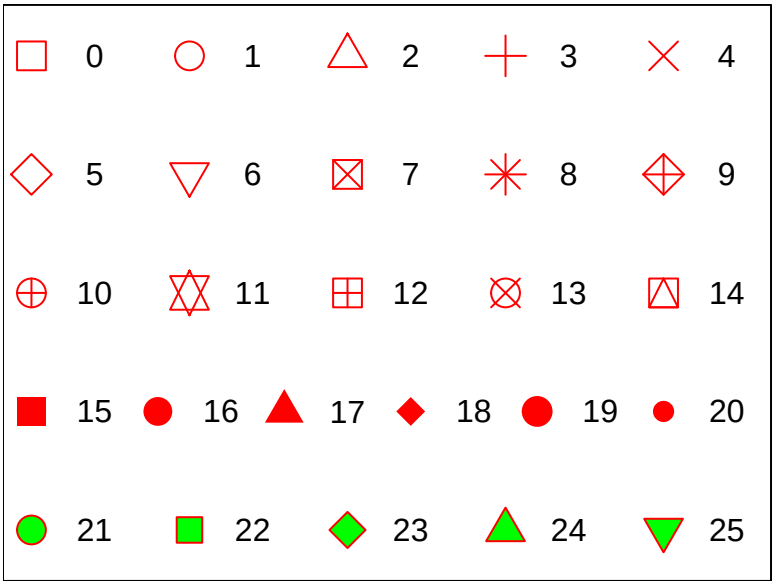
```
ggplot(data=ChickWeightMean, mapping=aes(x=Time, y=weight)) +
  geom_line(mapping=aes(linetype=Diet)) +
  scale_linetype_manual(values=c(3,4,5,6))
```



Line type values can be passed as numbers or strings.

8.4 Customizing Shape

R provides the following shapes symbols:

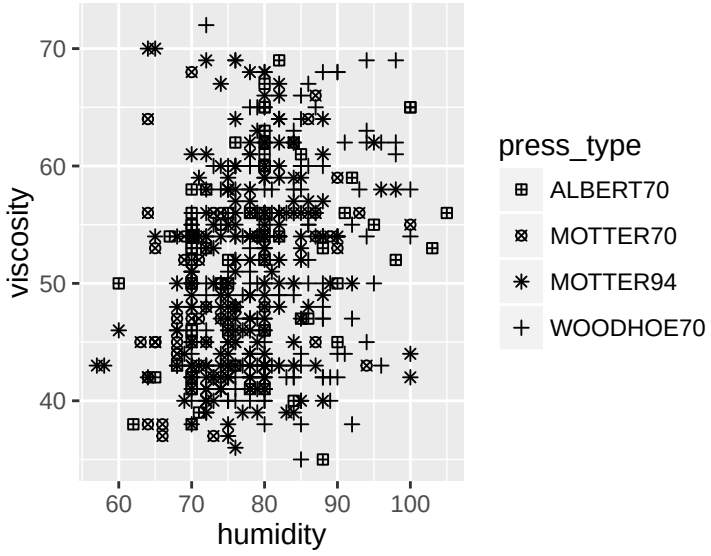


`shape` aesthetic has to be mapped to discrete variables.

8.4.1 Changing shape types

If you want to customize shape scale you have to set `values` argument of `scale_shape_manual()` function:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, shape=press_type)) +
  geom_point() +
  scale_shape_manual(values = c(12, 13, 8, 3))
```

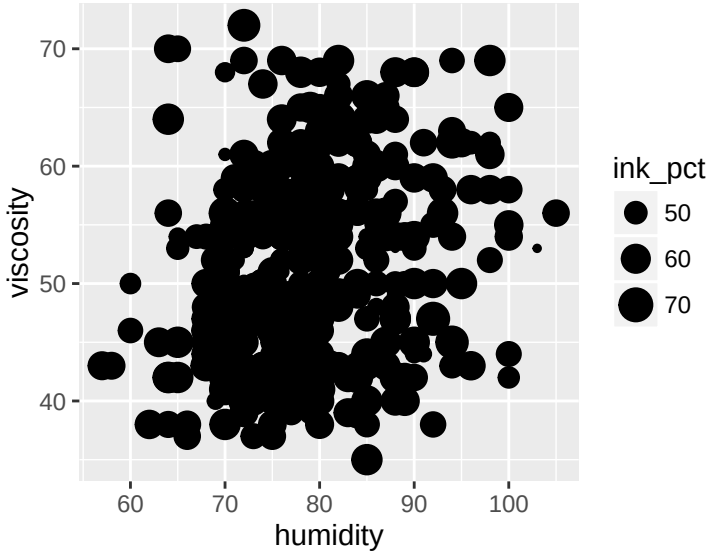


8.5 Customizing Size

The size aesthetic is most commonly used for points and text, and humans perceive the area of points (not their radius), so this provides for optimal perception. `size` can be mapped only to continuous variables.

8.5.1 Change the size of plotting symbols

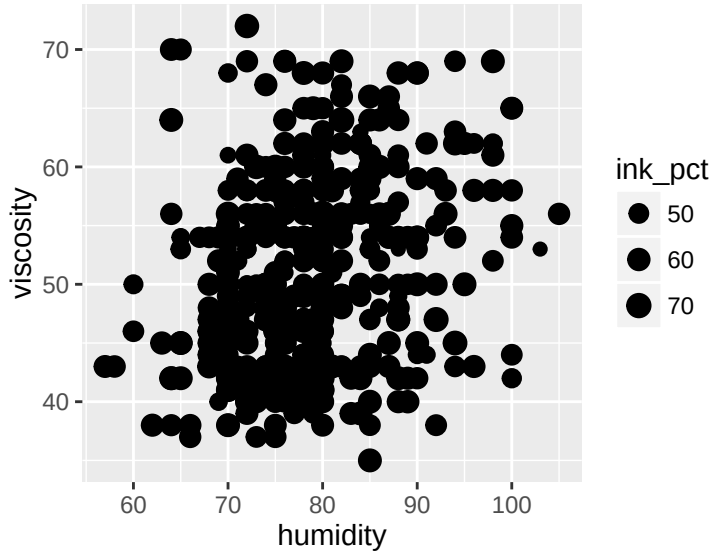
```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, size=ink_pct)) +
  geom_point()
```



When a variable is mapped to size, the results can be perceptually misleading. If it is important for the sizes to proportionally represent the quantities, you can change the range of sizes. By default the sizes of points go

from 1 to 6 mm. You could reduce the range to, say, 2 to 4 mm, with `scale_size_continuous(range=c(2, 4))`:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, size=ink_pct)) +
  geom_point() +
  scale_size(range=c(2, 4))
```

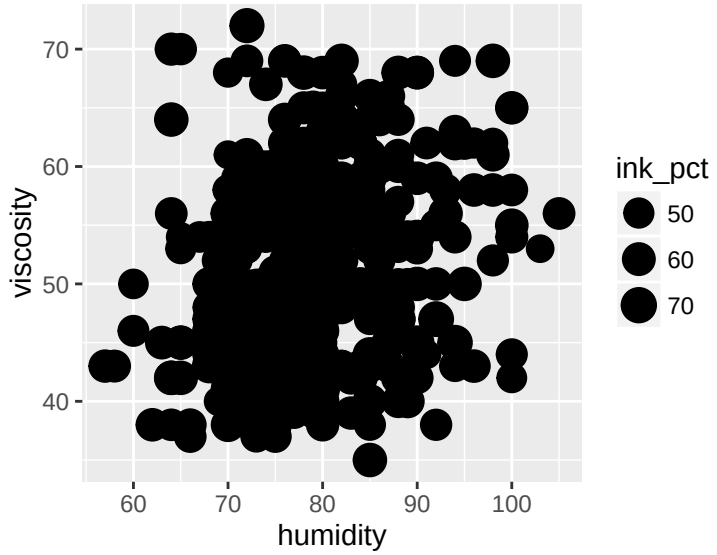


So you have to set to the `range` argument the minimum and the maximum size of the plotting symbols.

However, the point size numbers don't map linearly to diameter or area, so this still won't give a very accurate representation of the values.

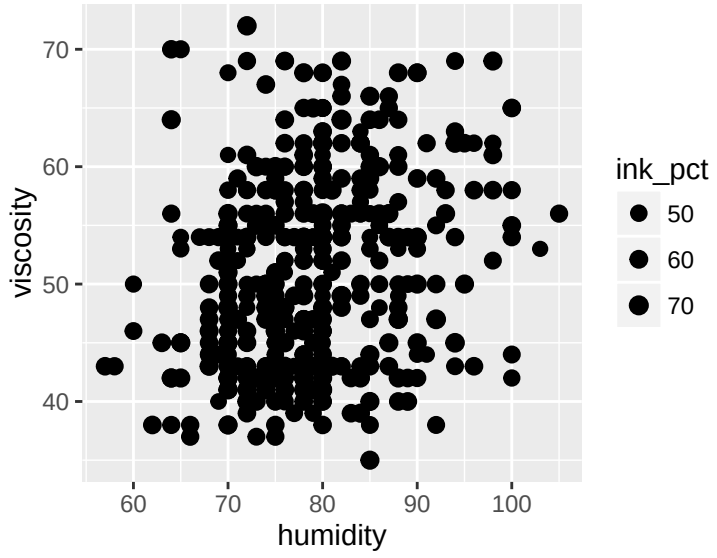
If you want that a value of zero is always mapped to size 0, use `scale_size_area()` instead of `scale_size()`:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, size=ink_pct)) +
  geom_point() +
  scale_size_area()
```



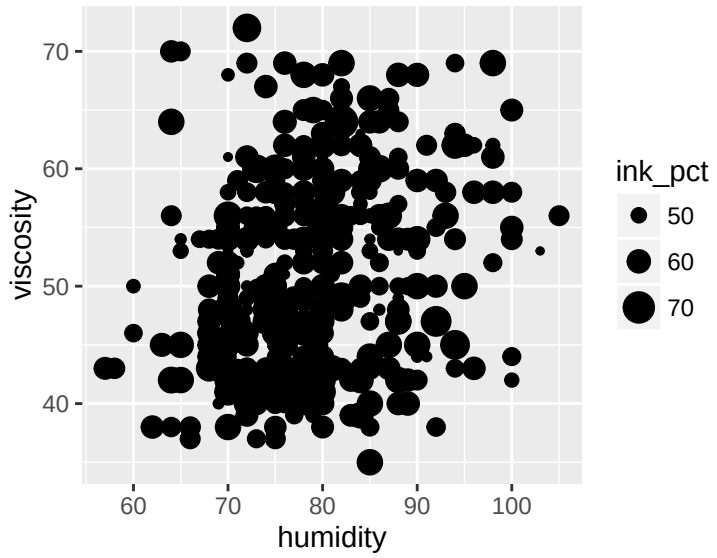
However with `scale_size_area()` you can not control the size of plotting symbols by using `range` argument. You can set only the size of largest points by setting `max_size` argument:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, size=ink_pct)) +
  geom_point() +
  scale_size_area(max_size = 3)
```



You can also use radiuos to control the size of plotting symbols by setting `scale_radius()` function:

```
ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, size=ink_pct)) +
  geom_point() +
  scale_radius()
```



This choice is always not recommended.

9 Axes Customization

```
require(ggplot2)
require(qdata)
require(scales)
```

```
data(bands)
data(brainbod)
```

Axes are more important than one thinks as they provide context for interpreting the displayed data. `ggplot2` displays the axes with defaults that look good in most cases, but you might want to control, for example, the axis scales, the axis labels, the number and placement of tick marks, the tick mark labels and so on.

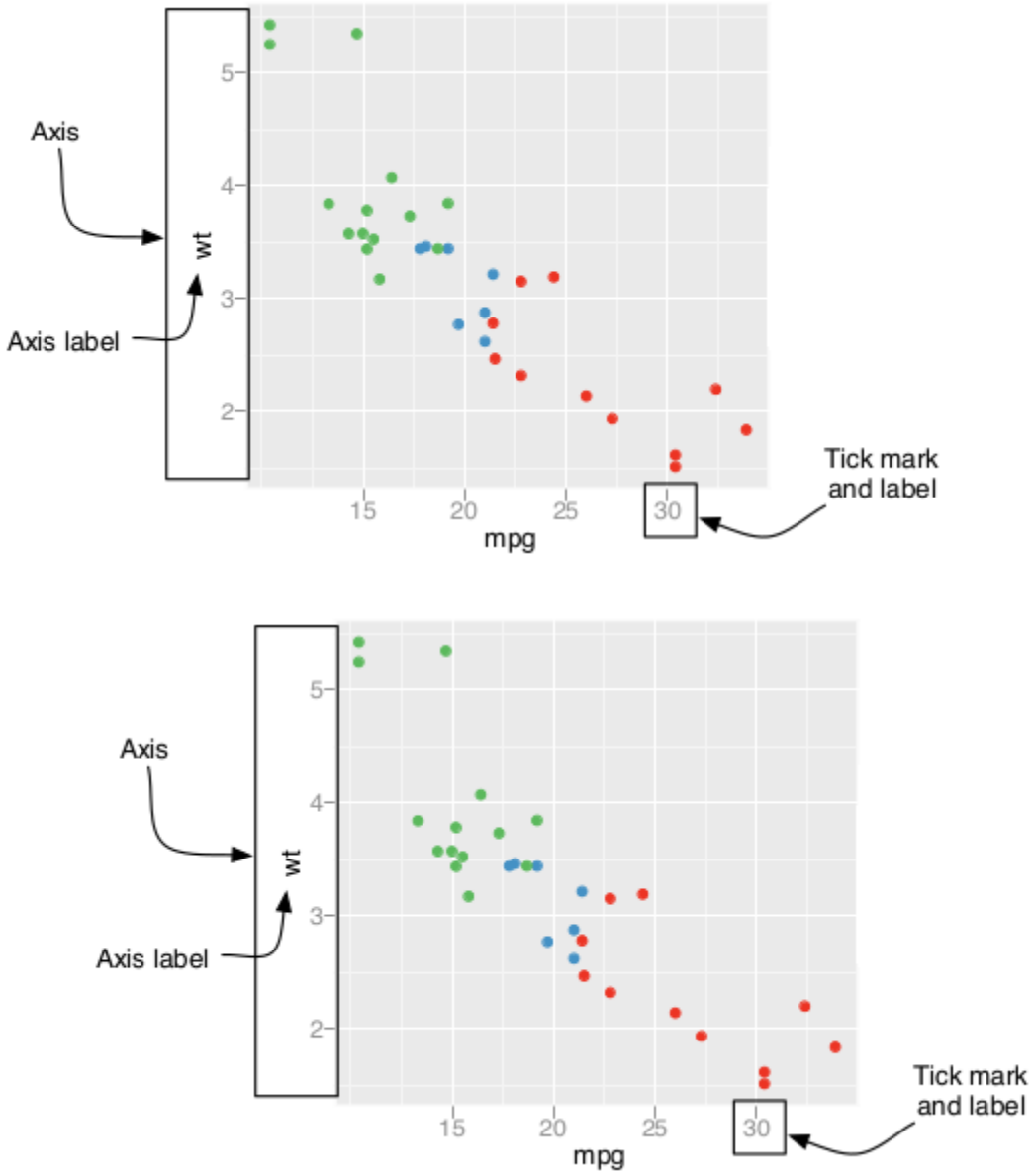


Figure 13:

`ggplot2` allows one to handle with the appearance of the axes by using three important elements of its grammar:

- `scales`, which controls the mapping from data to aestethics attributes
- `coordinates`, which describes how data coordinates are mapped to the plane of the graphic
- `themes`, which controls the non-data elements of the plot

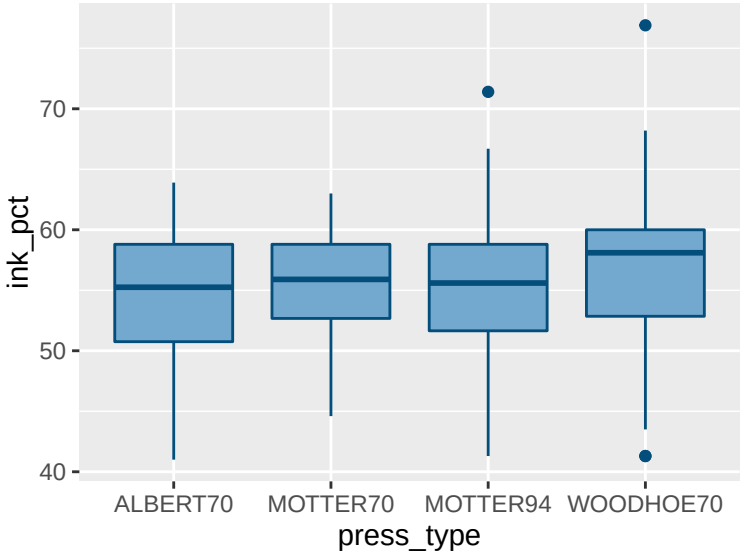
In the following paragraphs we will see how to handle with the most common questions about axes customization.

9.1 Swapping x and y axis

Swapping x and y axis is simple if you are generating a scatterplot. If you want to change what goes on the vertical axis and what goes on the horizontal axis you just have to exchange the variables mapped to `x` and `y` aesthetics, as `geom_point()` treats the x- and y-axes equally. However, the same cannot be said for Box Plot which summarize the data along the y-axis or for Line Plots, which move the lines in only one direction along the x-axis. To swap the axis of these and more others geoms which not treats the x- and y-axes equally, you have to use `coord_flip()` function as we learn in *Creating a Bar Plot* chapter.

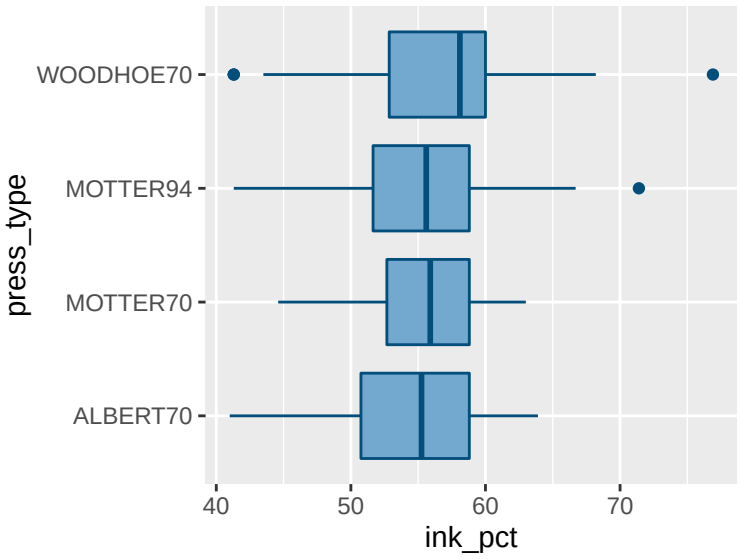
Let us see an example.
Consider `bands` dataset. Supposing you are interested in the differences of ink percentage accordingly to the type of press, you can build four box plots to compare distributions.

```
pl_1 <- ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#74a9cf", colour="#034e7b")
pl_1
```



Swapping the axes the result is:

```
pl_1 +
  coord_flip()
```



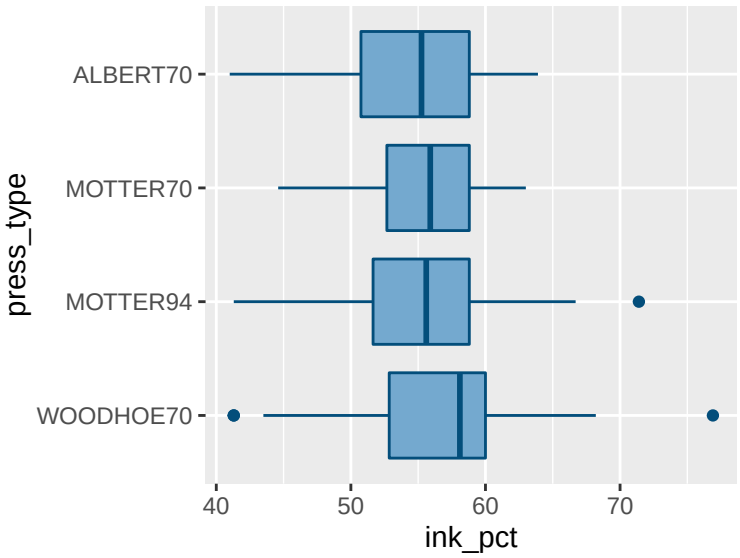
9.2 Change axis order

Let us consider a categorical (or discrete) axis with a factor mapped to it.

Sometimes when the axes are swapped, as in the previous example, the order of items will be the reverse of what you want.
On a graph with standard x- and y-axes, the x items start at the left and go to the right, which corresponds to the normal way of reading, from left to right. When you swap the axes, the items still go from the origin outward, which in this case will be from bottom to top but this conflicts with the normal way of reading, from top to bottom. Sometimes this can be a problem, so it is necessary to reverse the axis order.

This can be done by using `scale_x_discrete()` with `limits` argument set as `rev(levels(...))` in this way:

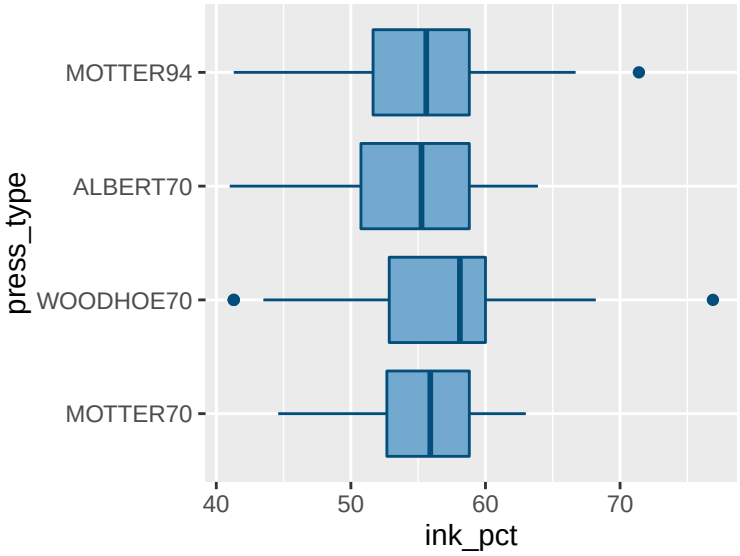
```
pl_1 +
  coord_flip() +
  scale_x_discrete(limits=rev(levels(bands$press_type)))
```



In general, to manually change the order of items in a categorical (or discrete) axis with a factor mapped to

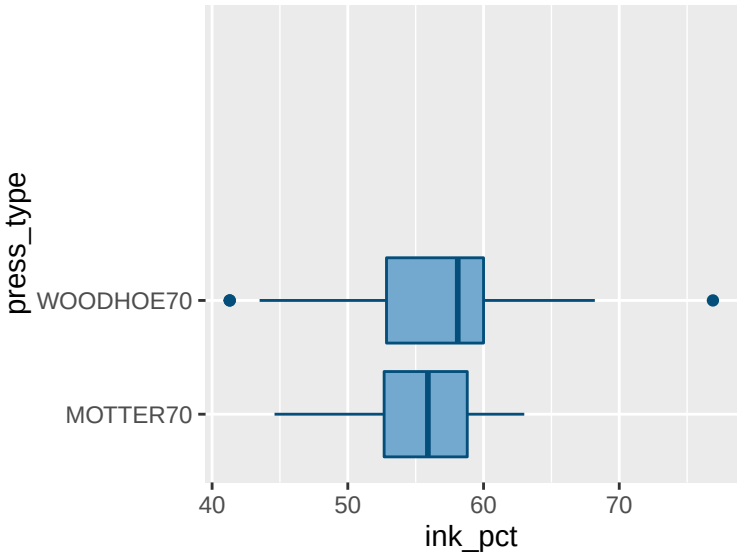
it, you have to specify `limits` argument in `scale_x_discrete()` or `scale_y_discrete()` functions with a vector of the levels in the desired order:

```
pl_1 +
  coord_flip() +
  scale_x_discrete(limits=c("MOTTER70","WOODHOE70","ALBERT70", "MOTTER94"))
```



You can also omit items with this vector, specifying only the items you want to keep:

```
pl_1 +
  coord_flip() +
  scale_x_discrete(limits=c("MOTTER70","WOODHOE70"))
```



There is empty space instead of not displayed `press_type` levels ("ALBERT70", "MOTTER94"). Maybe it is a bug of 2.1.0 version of `ggplot2`, used to realize this manual.

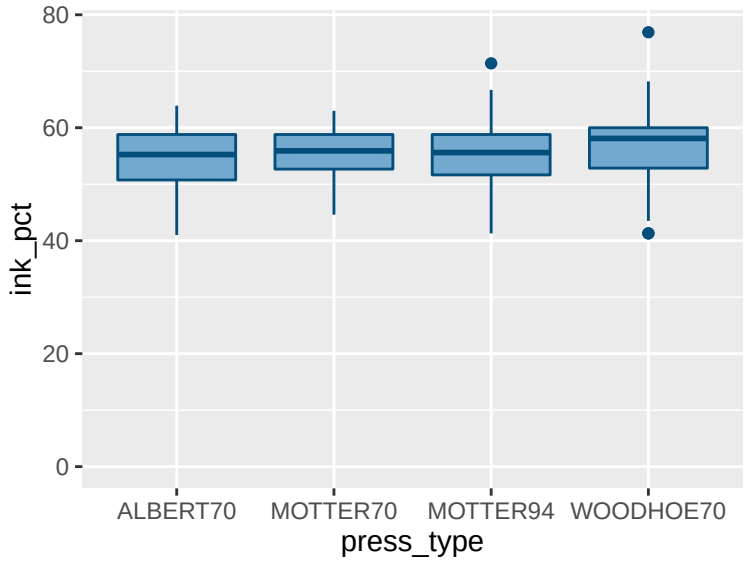
Note: the changes on axis order works also without `coord_flip()` function.

9.3 Setting the Range of a Continuous Axis

`ggplot2` computes the limits of an axes from the range of the data. But sometimes you could be interested in making the limits smaller that the range of the data to focus on an interesting area of the plot or to make the limits larger because you want multiple plots to match up.
`ggplot2` provides two ways of setting the range of the axes. The first way is to modify the scale, and the second is to apply a coordinate transformation. The first way is the most used. Let us analyze it.

To change the limits in x and/or y axes you could use `scale_y_continuous()` and/or `scale_x_continuous()`, setting the `limits` argument. `limits` argument has to be set equal to a vector of lenght two:

```
pl_1 +
  scale_y_continuous(limits=c(0, max(bands$ink_pct)))
```



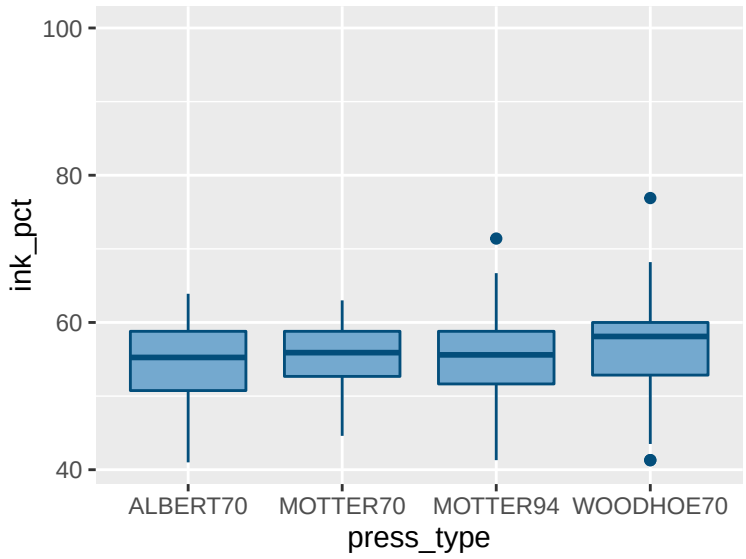
Because modifying limits is such a common task, `ggplot2` provides three shorthand: `ylim()`, `xlim()` and `lims()`.

The following two code lines produce equal results than the previous example:

```
pl_1 +
  ylim(0, max(bands$ink_pct))

pl_1 +
  lims(y = c(0, max(bands$ink_pct)))
```

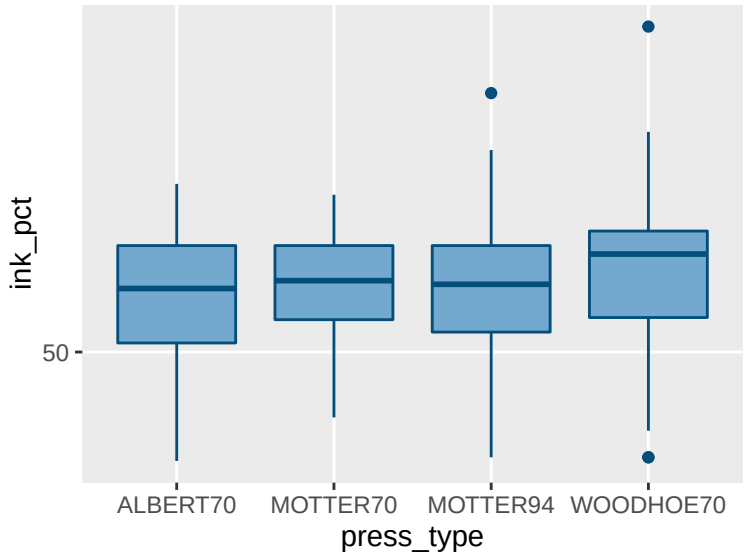
If you want to set only one limit, the other has to be set equal to `NA`. `ggplot2` computes automatically the not setted limit from the range of data:

```
pl_1 +
  ylim(limits=c(NA, 100))
```

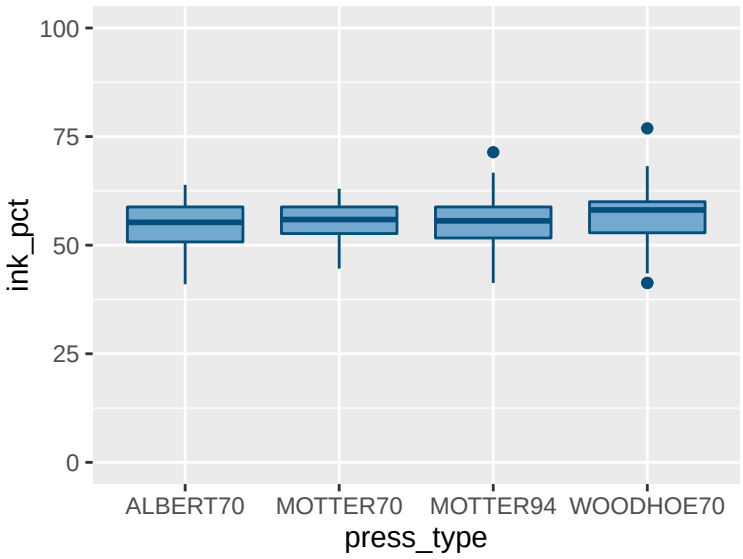


Sometimes using `ylim()` and `scale_y_continuous()` to set other properties can create problems (the same goes for `scale_x_continuous()` and `xlim`), because only the last of the directives will have an effect. In the following two examples, `ylim(0, 100)` should set the y range from 0 to 100, and `scale_y_continuous(breaks=c(0, 50, 100))` should put tick marks at 0, 50, and 100. However, in both cases, only the second directive has any effect:

```
pl_1 +
  ylim(0, 100) +
  scale_y_continuous(breaks=c(0, 50, 100))
```

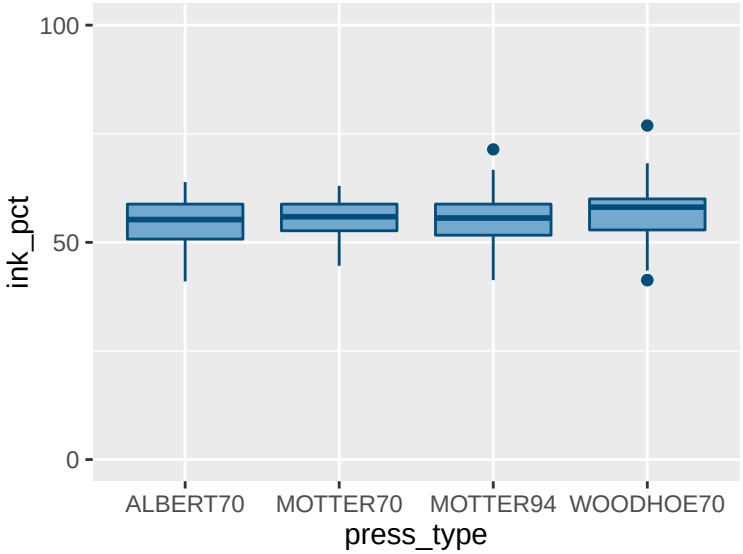



```
pl_1 +
  scale_y_continuous(breaks=c(0, 50, 100)) +
  ylim(0, 100)
```



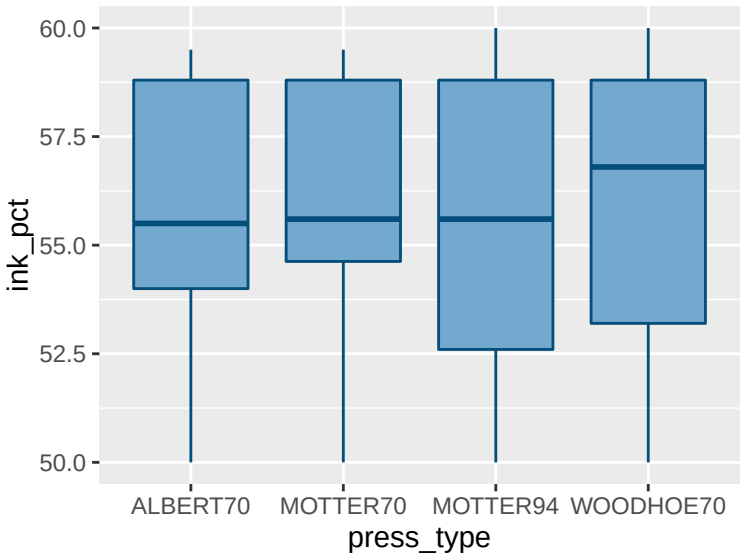
To make both changes work, we suggest you to set both limits and breaks in `scale_y_continuous()`:

```
pl_1 +
  scale_y_continuous(limits=c(0, 100), breaks=c(0, 50, 100))
```

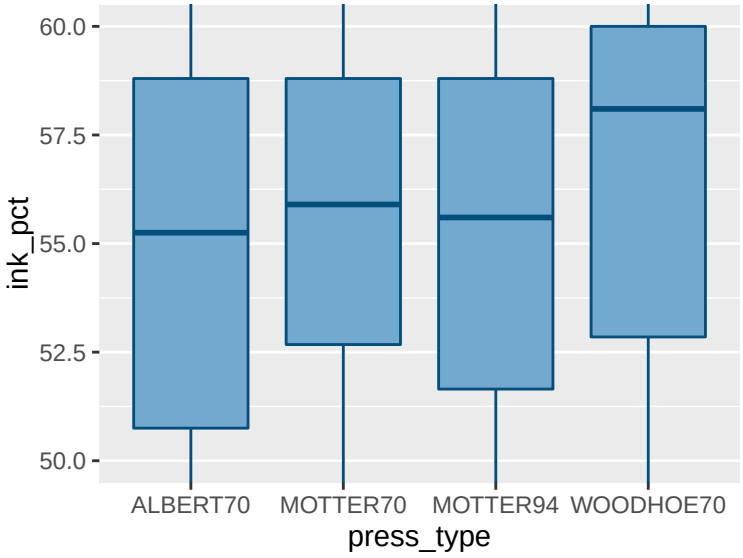


We saw that modifying the scale for changing axes limits works well. However, when you modify the limits of the x or y scale, any data outside of the limits is removed that is, the out-of-range data is not only not displayed, it is removed from consideration entirely.
With the Box Plots in these examples, if you restrict the y range so that some of the original data is clipped, the box plot statistics will be computed based on clipped data, and the shape of the box plots will change.
With a coordinate transformation for changing limits, the data is not clipped; in essence, it zooms in or out to the specified range.
Let us see the difference between these two methods:

```
# scale transformation method
pl_1 +
  scale_y_continuous(limits = c(50, 60))
```



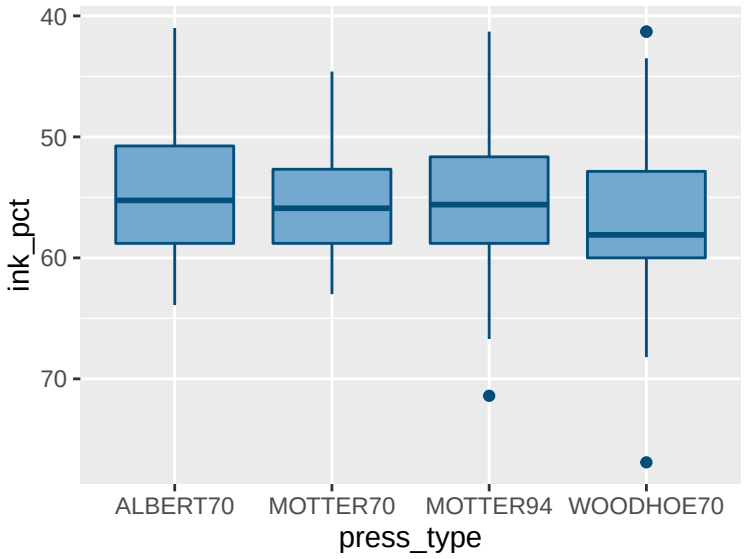

```
# coordinate transformation method
pl_1 +
  coord_cartesian(ylim = c(50, 60))
```



9.4 Reverse a continuous axis order

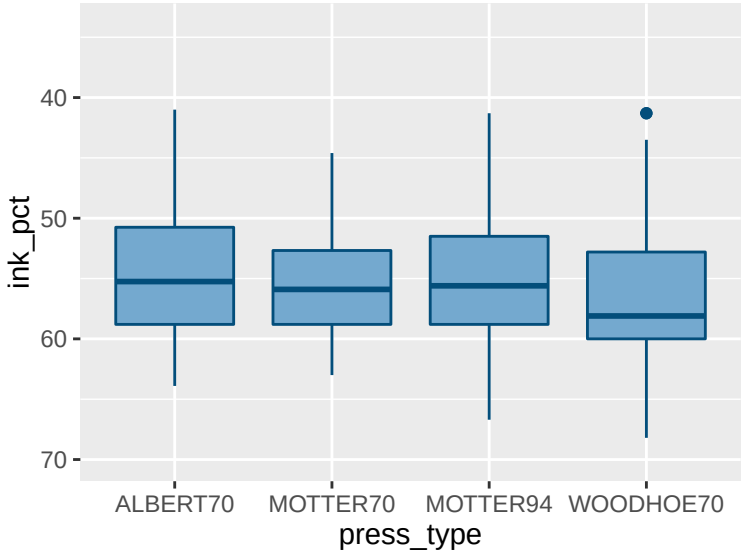
To reverse a continuous axes the function to use are: `scale_y_reverse()` or `scale_x_reverse()`:

```
pl_1 +
  scale_y_reverse()
```



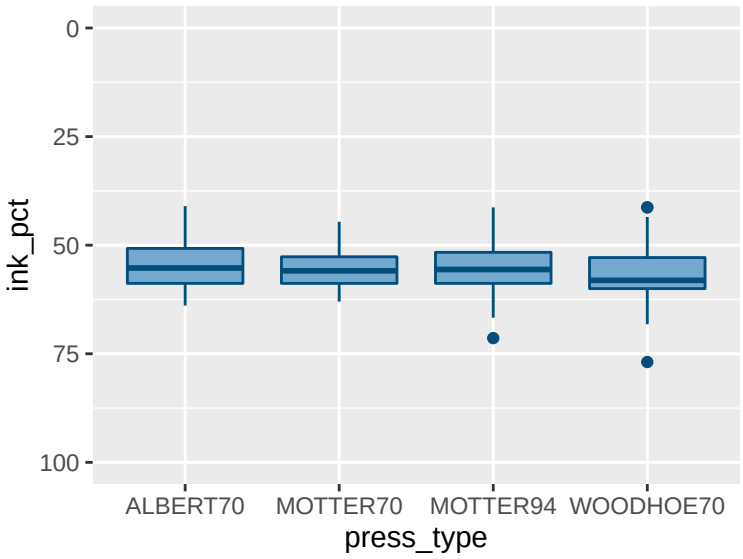
You can use also `ylim()` in this way:

```
pl_1 +
  ylim(70, 34)
```



Like `scale_y_continuous()`, `scale_y_reverse()` does not work with `ylim()`. (The same is true for the x-axis properties.) If you want to reverse an axis and set its range, you must do it within the `scale_y_reverse()` statement, by setting the limits in reversed order:

```
pl_1 +
  scale_y_reverse(limits=c(100, 0))
```



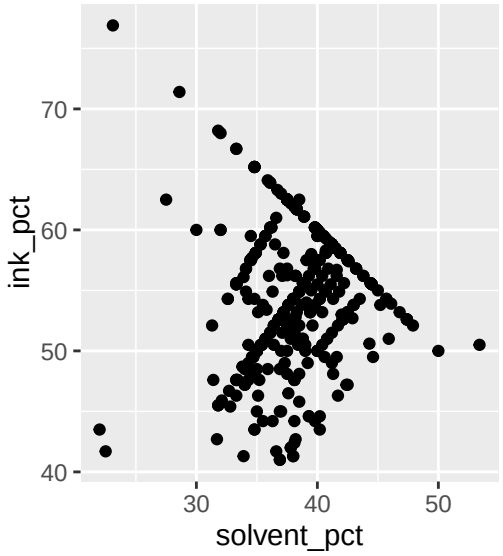
9.5 Resize axis scale

Suppose you are interested in the relationship between ink and solvent percentage in `bands` dataset:

```
pl_2 <- ggplot(data = bands, mapping = aes(y=ink_pct, x=solvent_pct)) +
  geom_point()
```

If you want the same scale for your axes, you have to use `coord_equal()` function.

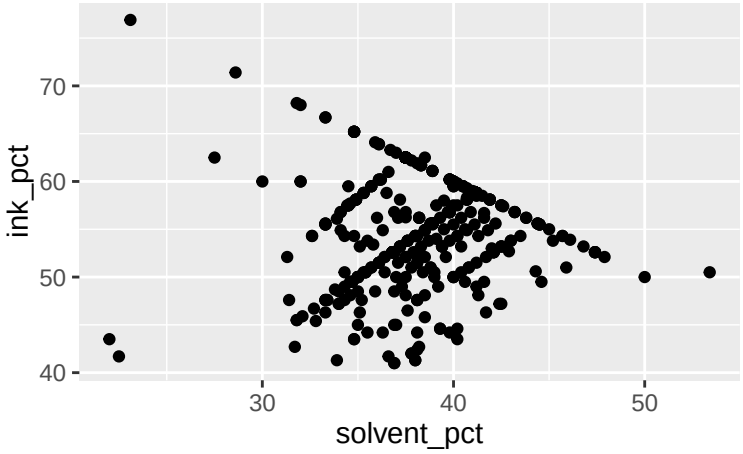
```
pl_2  +
  coord_equal()
```



By default `ratio` argument is set to 1, which ensures that one unit on the x-axis is the same length as one unit on the y-axis.

You can change it as you want to resize the scales:

```
pl_2  +
  coord_equal(ratio=1/2)
```



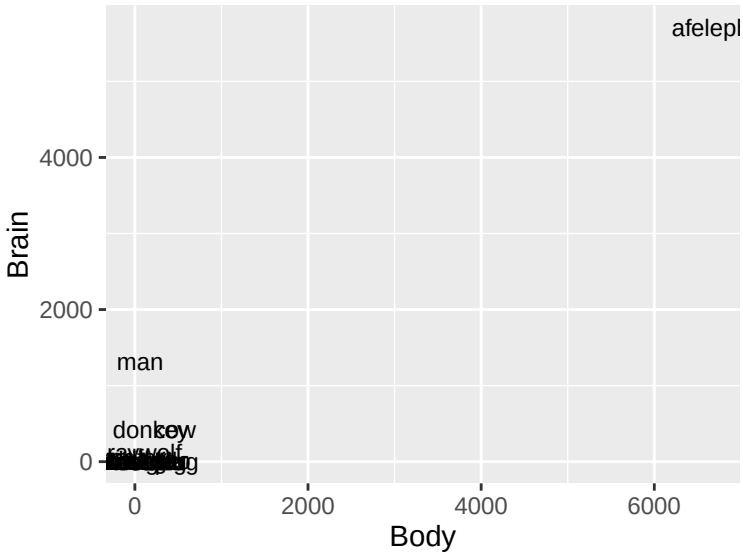
Ratio is expressed as `y / x`, so in the previous example, two unit on the x-axis correspond to one unit on the y-axis.

You can use also `coord_fixed()` to achieve the same results.

9.6 Axis scale transformations

`brainbod` dataset contains information about the weight of body and brain of different species of animals. We wants to visualize the relationship between body weight and brain weight of 15 animals species:

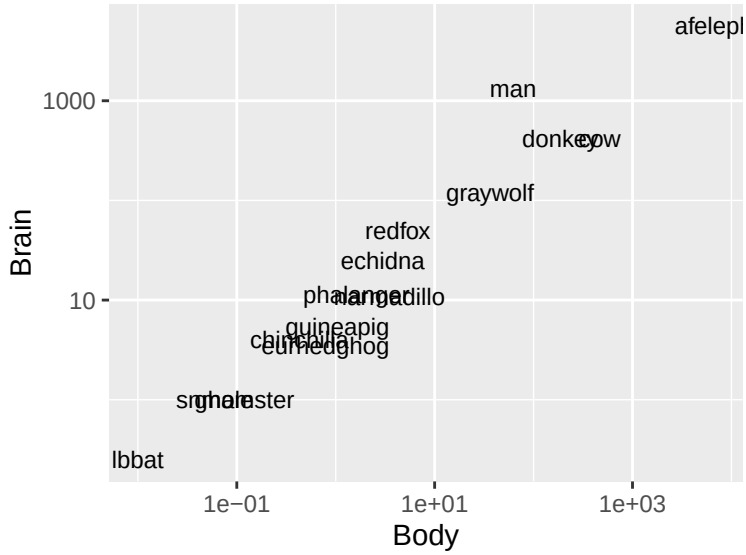
```
pl_3 <- ggplot(brainbod, aes(x=Body, y=Brain, label=Species)) +
geom_text(size=3)
pl_3
```



With the default linearly scaled axes, it's hard to make much sense of this graph. Because of the presence of elephant, the rest of the animals get squished into the lower-left corner of the graph. This is a case where the data is distributed exponentially on both axes, so a logarithmic transformations is the best solution. `scale_x_log10()` and `scale_y_log10()` functions are continuous position scales which transform x and y axis scales respectively.

```
pl_3 +
  scale_x_log10() +
```

```
  scale_y_log10()
```

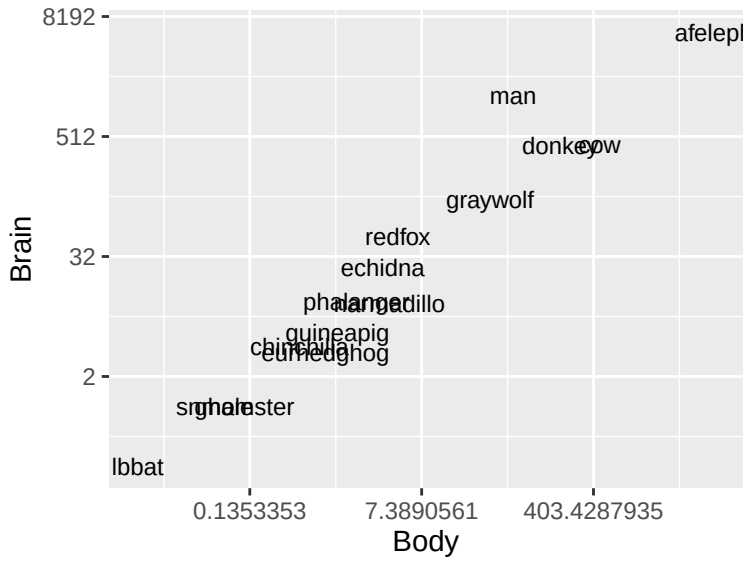


Now the plot results are intrepretable, we clearly see a linear relationship between brain weight and body weight of the analyzed animal species.

The previous example used a log_{10} transformation, but it is possible to use other transformations, such as log_2 and natural log. `scale_x_log10()` (and similarly for y) is a shortcut as it is one of the most common trasformations. A wide variety of transformations can be applied to scales of continuous axes, by setting `trans` argument of `scale_x_continuous()` and/or `scale_y_continuous()`.

Let us apply natural logarithmic transformation to x axis and log_2 transformation to y axis:

```
pl_3 +
  scale_x_continuous(trans = log_trans()) +
  scale_y_continuous(trans = log2_trans())
```



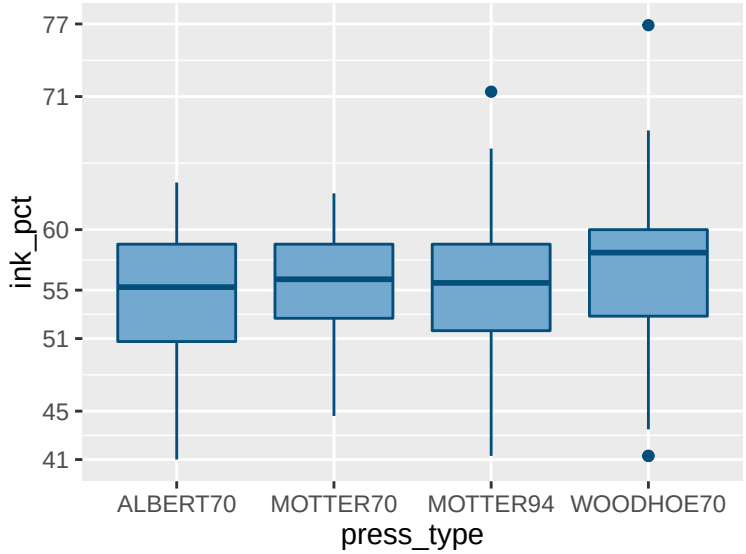
9.7 Modifying axis appearance: tick labels, tick marks, and the grid lines

There are actually three related items that can be controlled: tick labels, tick marks, and the grid lines. For continuous axes, `ggplot()` normally places a tick label, tick mark, and major grid line at each value of breaks. For categorical axes, these things go at each value of limits.

9.7.1 Change Tick marks position

Usually `ggplot()` does a good job of deciding where to put the tick marks, but if you want to change them, set breaks in the scale in this way:

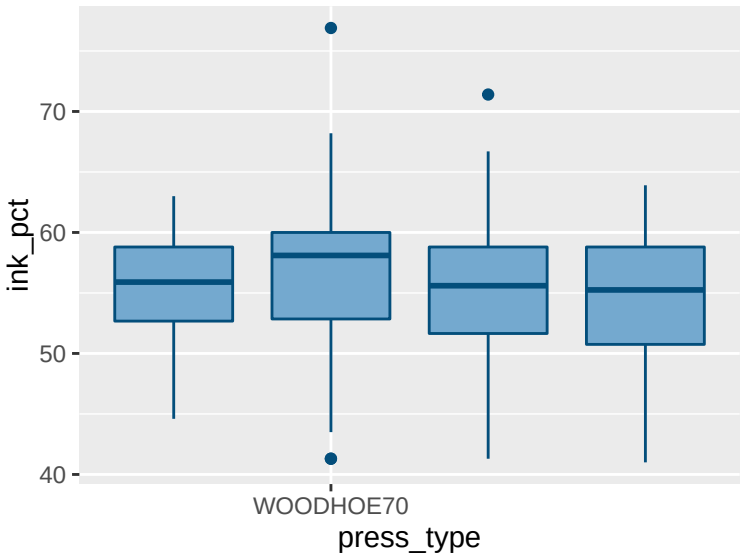
```
pl_1 +
  scale_y_continuous(breaks=c(41, 45, 51, 55, 60, 71, 77))
```



The location of the tick marks defines where major grid lines are drawn. If the axis represents a continuous variable, minor grid lines, which are fainter and unlabeled, will by default be drawn halfway between each major grid line. If the axis is discrete instead of continuous, then there is by default a tick mark for each item.

For discrete axes, you can change the order of items or remove them by specifying the limits as learn in *Change axis order* paragraph. Setting breaks will change which of the levels are labeled, but will not remove them or change their order.

```
pl_1 +
  scale_x_discrete(limits=c("MOTTER70","WOODHOE70", "MOTTER94", "ALBERT70"), breaks="WOODHOE70")
```



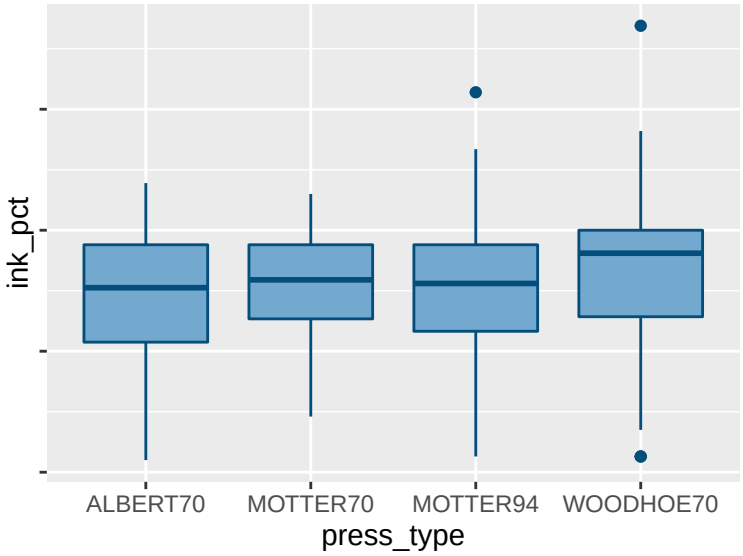
9.7.2 Removing Tick Marks and Labels

The theme function for the removal of both tick marks and labels is `element_blank()`.

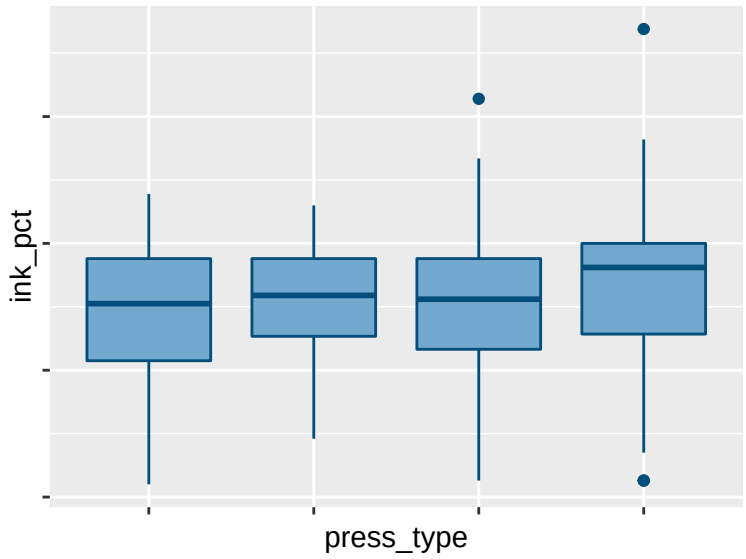
`axis.text.x`, `axis.text.y` and `axis.text` arguments of `theme()` function controls the text of x, y or both axis ticks.

To remove just the tick labels of both axes use `theme(axis.text = element_blank())`, and to remove the tick labels of y axes use `theme(axis.text.y = element_blank())` (or do the same for `axis.text.x`). This will work for both continuous and categorical axes:

```
pl_1 +
  theme(axis.text.y = element_blank())
```



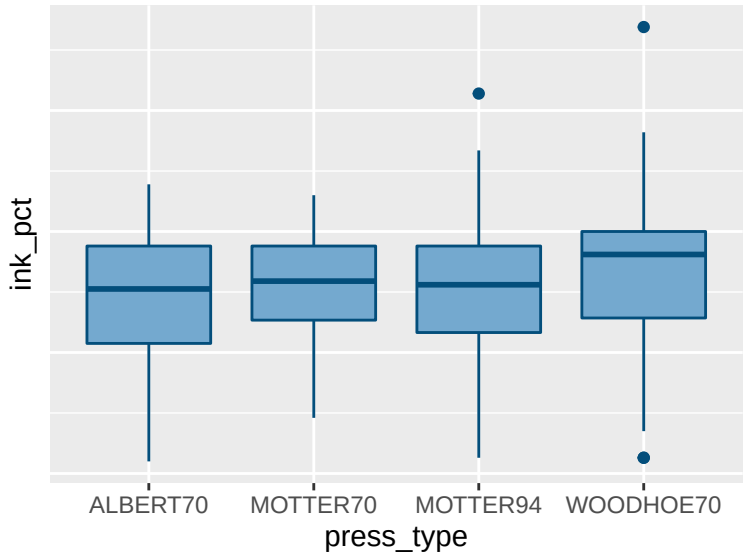

```
pl_1 +
  theme(axis.text = element_blank())
```



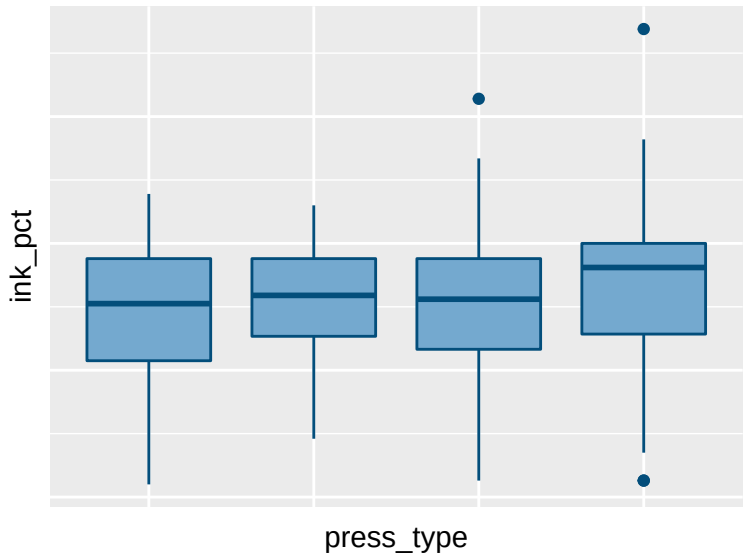
`axis.ticks.x`, `axis.ticks.y` and `axis.ticks` controls the marks of x, y or both axis ticks.

To remove the tick marks on both axes, use `theme(axis.ticks=element_blank())`, and to remove the tick marks of y axes use `theme(axis.ticks.y=element_blank())`. This will remove the tick marks.

```
pl_1 +
  theme(axis.ticks.y = element_blank(), axis.text.y = element_blank())
```

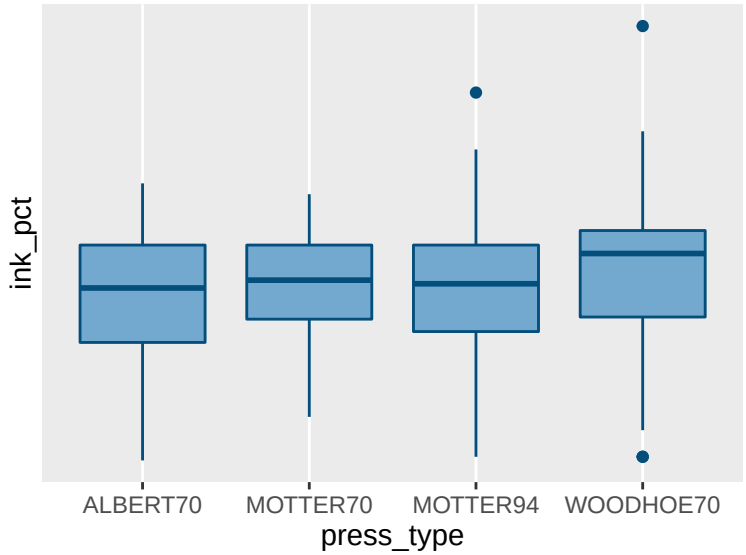



```
pl_1 +
  theme(axis.ticks = element_blank(), axis.text = element_blank())
```



To remove the tick marks, the labels, and the grid lines for continuous axis, set `breaks` to `NULL` in `scale_y_continuous` (the same for `scale_x_continuous`).

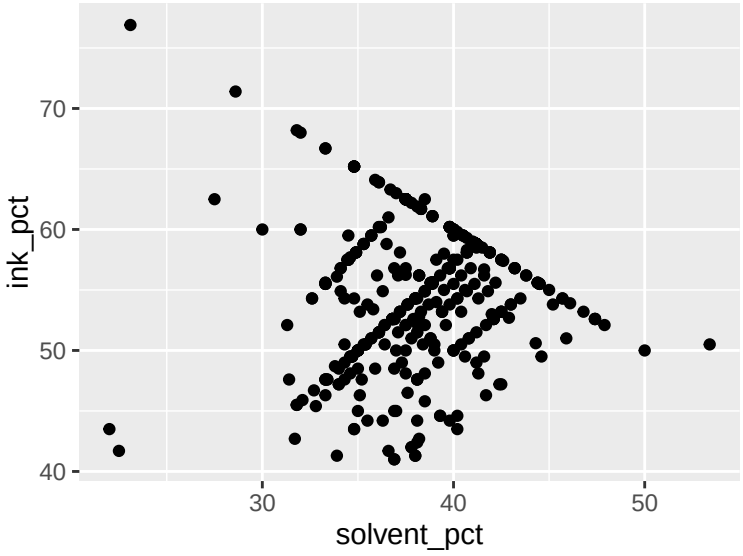
```
pl_1 +
  scale_y_continuous(breaks=NULL)
```



9.7.3 Changing the Text of Tick Labels

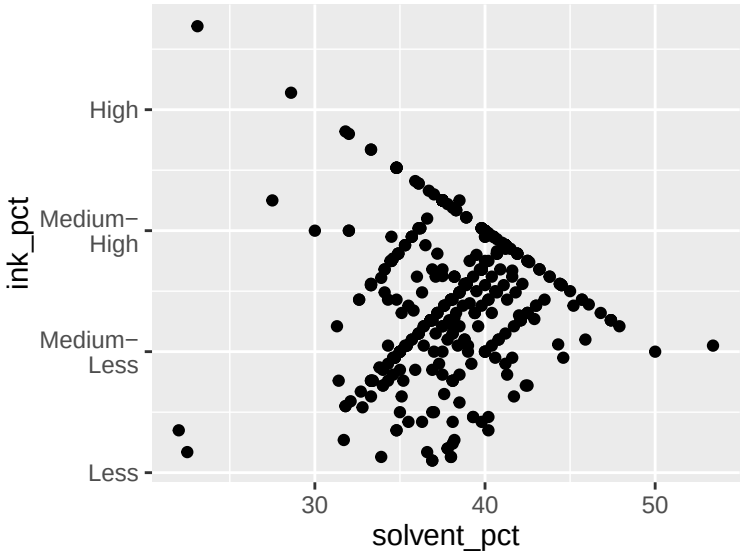
Let us consider the scatterplot showing the relationship between ink and solvent percentage in cylinder banding process in rotogravure printing, stored in `pl_2` object:

```
pl_2
```



To change the text of ticks marks in continuous axes, set `breaks` and `labels` in `scale_y_continuous` function:

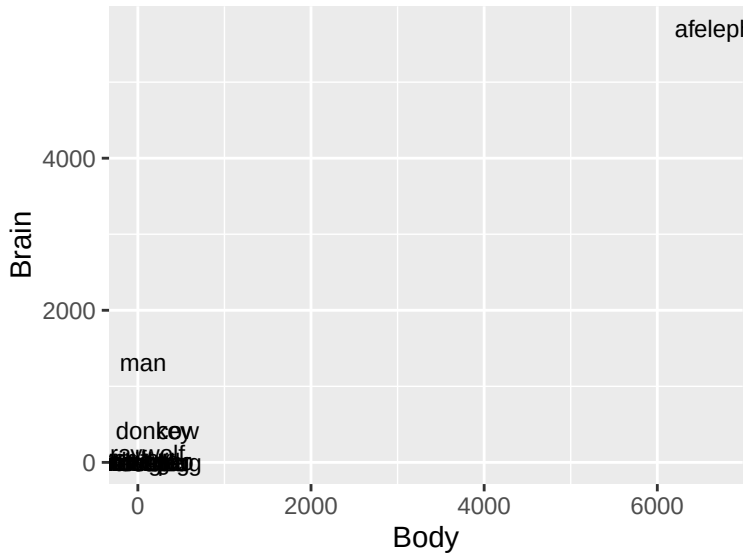
```
pl_2 +
  scale_y_continuous(breaks=c(40, 50, 60, 70), labels=c("Less", "Medium-\nLess", "Medium-\nHigh", "High
```



Pay attention to `"\n"` character, which tells `ggplot()` to put a line break.

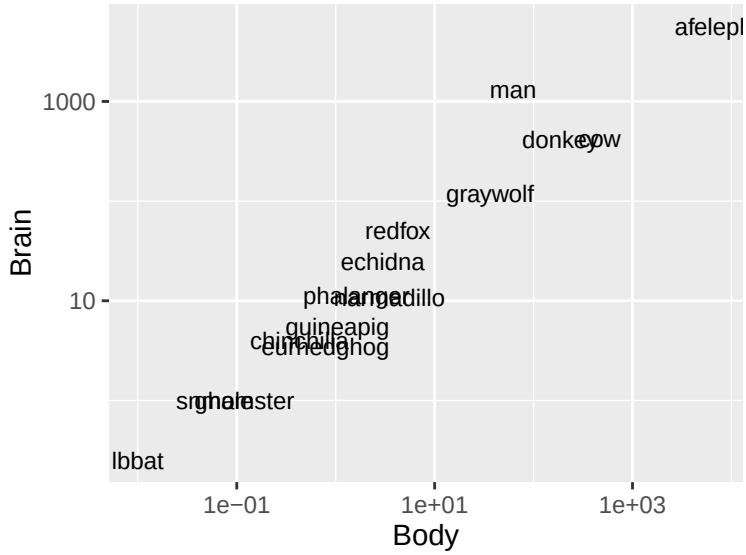
Let us consider the example described in *Axis scale transformations* paragraph, which analyze the relationship between body and brain weight of 15 species of animals, stored in `pl_3` object:

```
pl_3
```



We applied a logarithmic transformation to the axis:

```
pl_3 +
  scale_x_log10()+
  scale_y_log10()
```



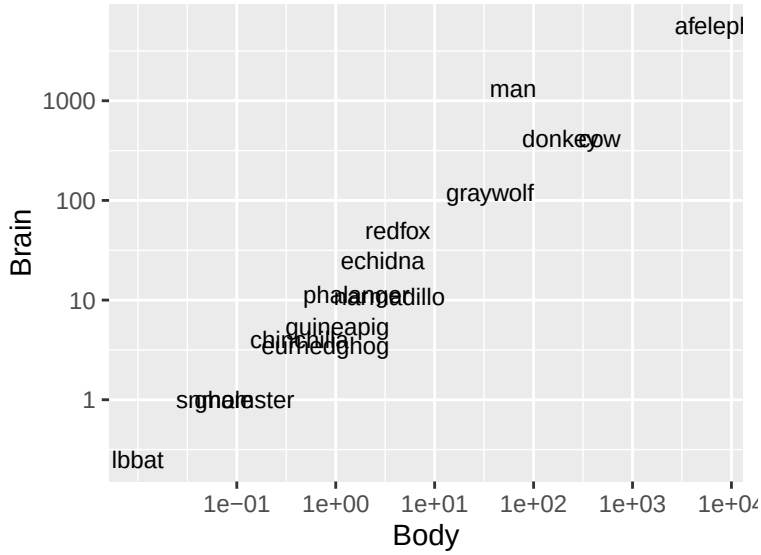
With a log axis, a given visual distance represents a constant proportional change; for example, each centimeter on the y-axis might represent a multiplication of the quantity by 10. In contrast, with a linear axis, a given visual distance represents a constant quantity change; each centimeter might represent adding 10 to the quantity.

`ggplot2` will try to make good decisions about where to place the tick marks, but if you don't like them, you can change them by specifying breaks and, optionally, labels . In the example here, the automatically generated tick marks are spaced farther apart than is ideal. For the y-axis tick marks, we can get a vector of every power of 10 from 10^0 to 10^3 like this:

```
10^(0:3)
```

```
pl_3 +
  scale_x_log10(breaks=10^(-1:5)) +
```

```
  scale_y_log10(breaks=10^(0:3))
```

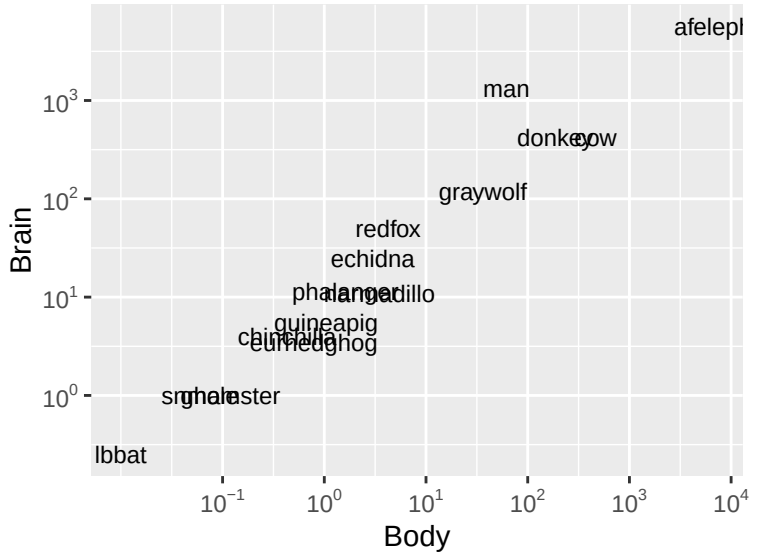


To instead use exponential notation for the break labels, use the `trans_format()` function, from the `scales` package:.

Supposing `scales` is already installed, it must be loaded:

```
require(scales)
```

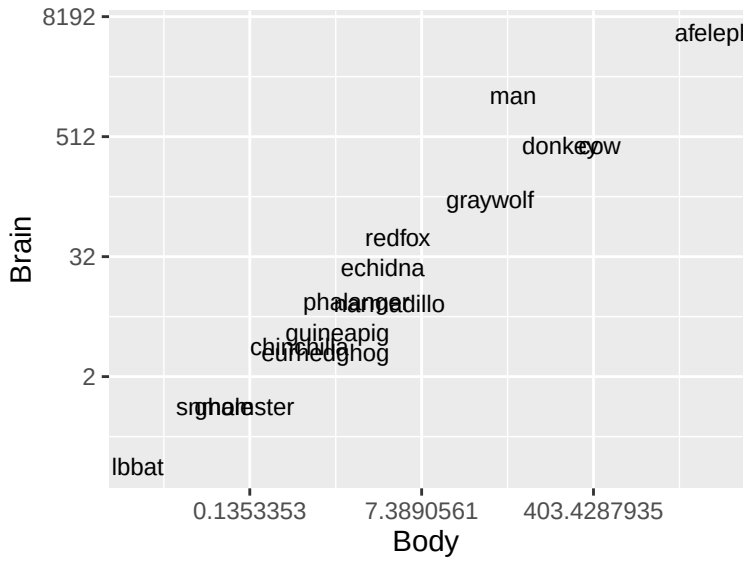
```
pl_3 +
  scale_x_log10(breaks=10^(-1:5), labels=trans_format("log10", math_format(10^.x))) +
  scale_y_log10(breaks=10^(0:3), labels=trans_format("log10", math_format(10^.x)))
```



`trans_format()` function set a format to labels after a transformation. The transformation is "log10" and the format is specified by the function `math_format()` (of `scales` package), which define the arbitrary expression to use for the label.

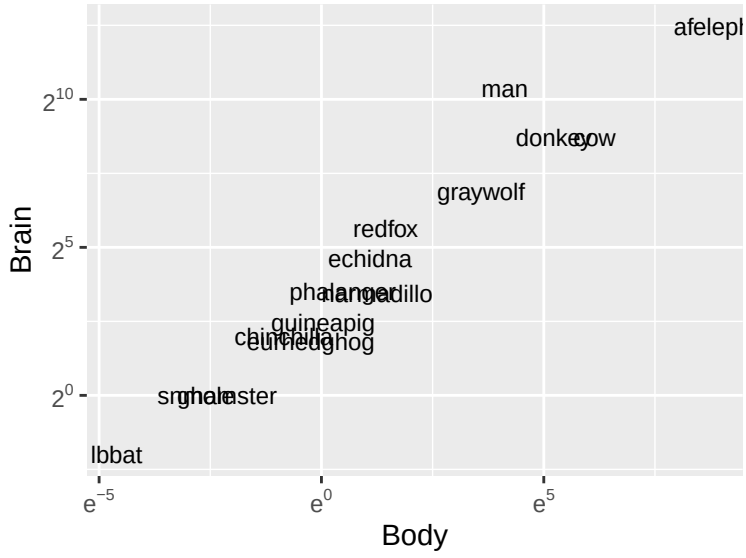
Talking about the plot in which scales have been transformed by natural log and log_2:

```
pl_3 +
  scale_x_continuous(trans = log_trans()) +
  scale_y_continuous(trans = log2_trans())
```



The labels can be modified in this way:

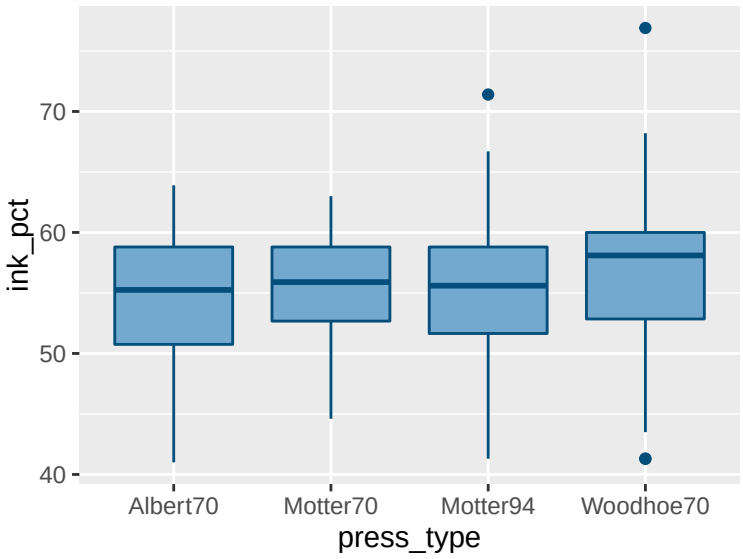
```
pl_3 +
  scale_x_continuous(trans = log_trans(), breaks = trans_breaks("log", function(x) exp(x)),
    labels = trans_format("log", math_format(e^.x))) +
  scale_y_continuous(trans = log2_trans(), breaks = trans_breaks("log2", function(x) 2^x),
    labels = trans_format("log2", math_format(2^.x)))
```



`trans_breaks()` function of `scales` package define the breaks according to the transformation applied.

Let us go back to boxplot showing the distribution of `ink_pct` by `press_type` in `bands` dataset.
To change the text of ticks marks in discrete axes, set the `labels` argument in `scale_x_discrete()` function.

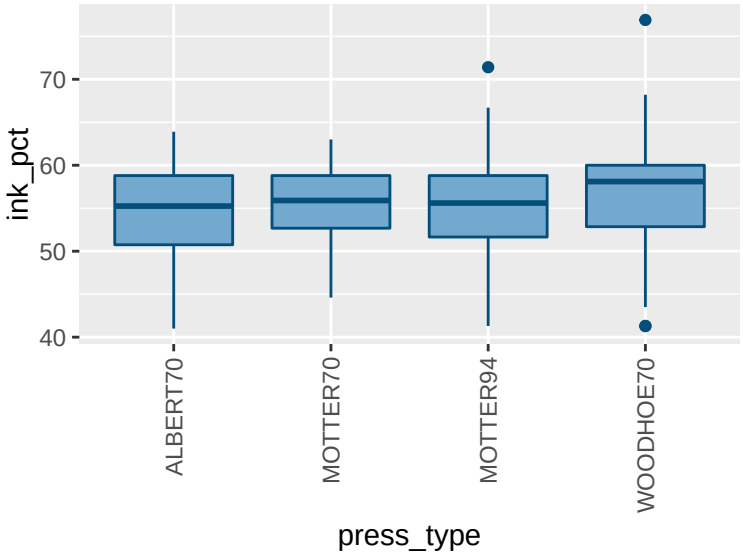
```
pl_1 +
  scale_x_discrete(labels=c("Albert70", "Motter70", "Motter94", "Woodhoe70"))
```



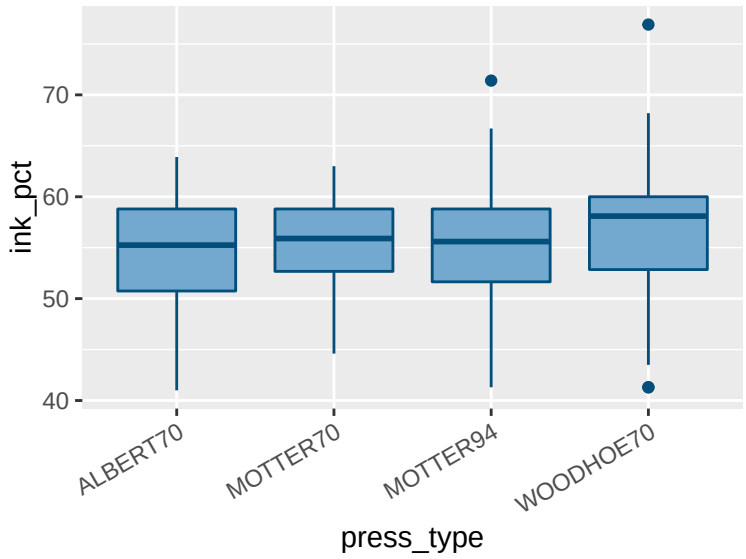
9.7.4 Changing the Appearance of Tick Labels

Text properties like rotation, size, style (bold/italic/normal), and the font family (such as Times or Helvetica) can be set with `element_text()` theme function:

```
# To rotate the text 90 degrees counterclockwise
pl_1 +
  theme(axis.text.x = element_text(angle=90, hjust=1, vjust=0.5))
```

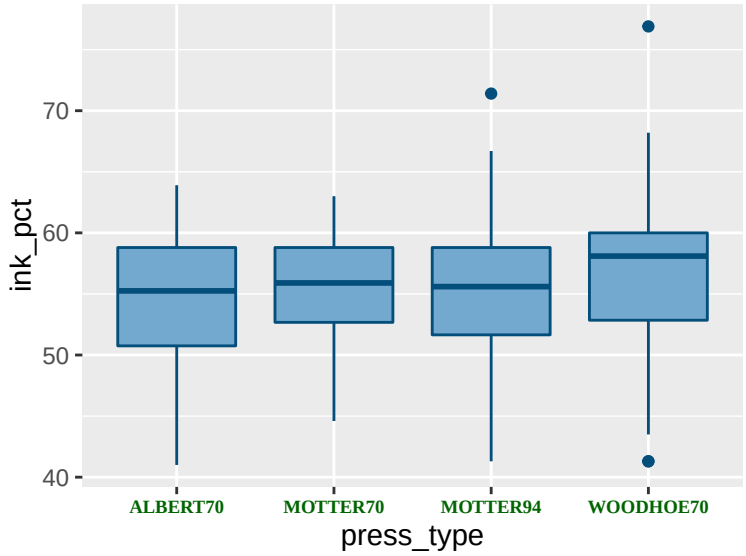



```
# Rotating the text 30 degrees
pl_1 +
  theme(axis.text.x = element_text(angle=30, hjust=1, vjust=1))
```



The `angle` setting specifies the text rotation and the `hjust` and `vjust` settings specify the horizontal alignment (left/center/right) and vertical alignment (top/middle/bottom).

```
pl_1 +
  theme(axis.text.x = element_text(family="Times", face="bold", colour="darkgreen", size=rel(0.8)))
```

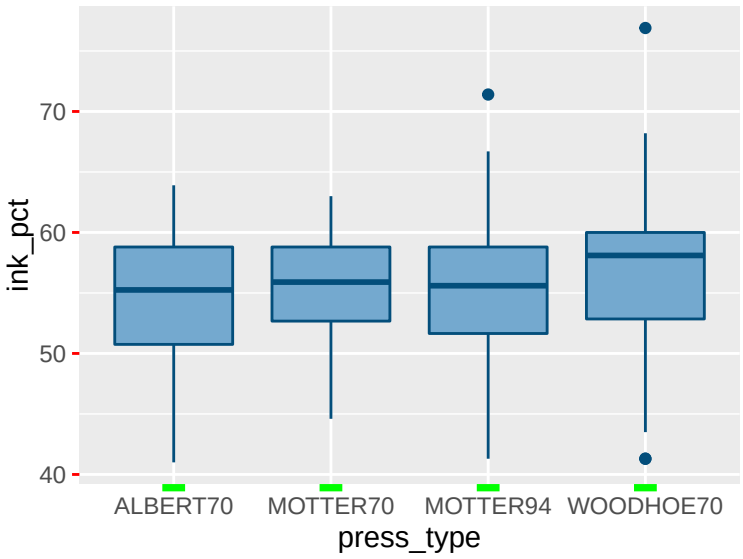


The `family` setting specifies the font family, `face` the font face, `colour` the text colour and `size` the text size. In this example, the size is set to `rel(0.8)`, which means that it is 0.8 times the size of the base font size for the theme.

9.7.5 Changing the Appearance of Tick marks

The appearance of tick marks can be changed by `element_line()` theme function.

```
pl_1 +
  theme(axis.ticks.x = element_line(colour="green", size=4), axis.ticks.y=element_line(colour="red"))
```



In particular, `colour` setting specifies the line colour and `size` the line size.

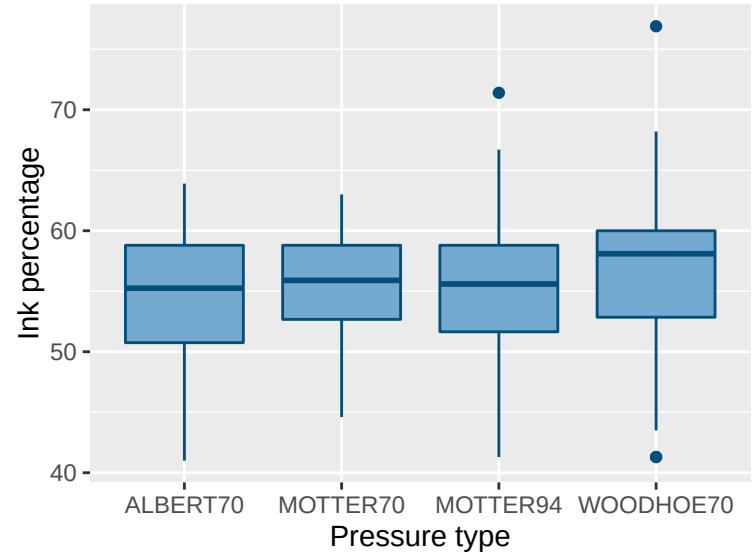
9.7.6 Changing the Text of Axis Labels

By default the graphs will just use the column names from the data frame as axis labels. This might be fine for exploring data, but for presenting it, you may want more descriptive axis labels.

`ggplot2` provides these three equivalent ways to change the axis labels:

```
pl_1 +
  xlab("Pressure type") +
  ylab("Ink percentage")
pl_1 +
  labs(x = "Pressure type", y = "Ink percentage")
pl_1 +
  scale_y_continuous(name="Ink percentage") +
  scale_x_discrete(name="Pressure type")
```

Which produces the same results:



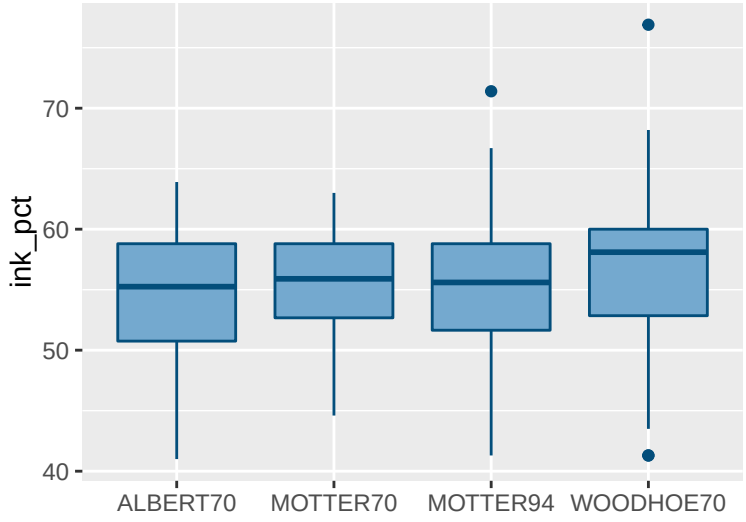
Using `scale_y_continuous()` may look a bit awkward, but it can be useful if you're also setting other properties of the scale, such as the tick mark placement, range, and so on.

9.7.7 Removing Axis Labels

Sometimes axis labels are redundant or obvious from the context, and don't need to be displayed.

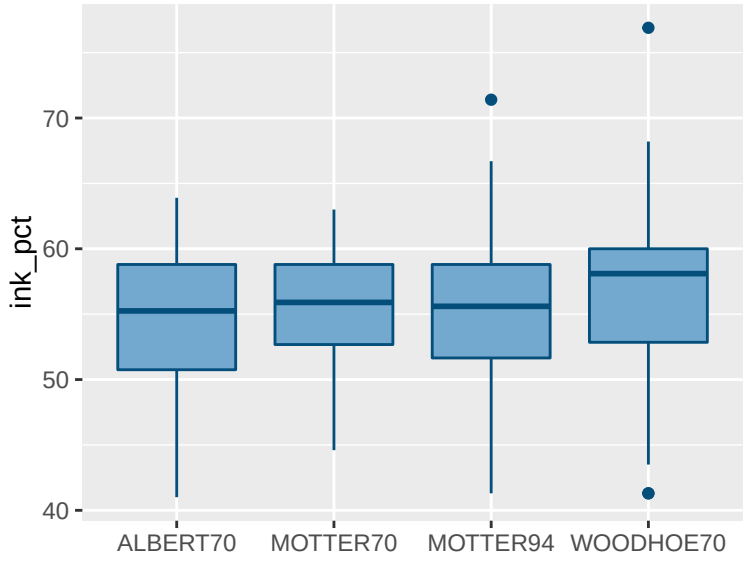
You can set x label to an empty string:

```
pl_1 +
  xlab("")
```



or you can use `theme()` function to set `axis.title.x=element_blank()`:

```
pl_1 +
  theme(axis.title.x=element_blank())
```



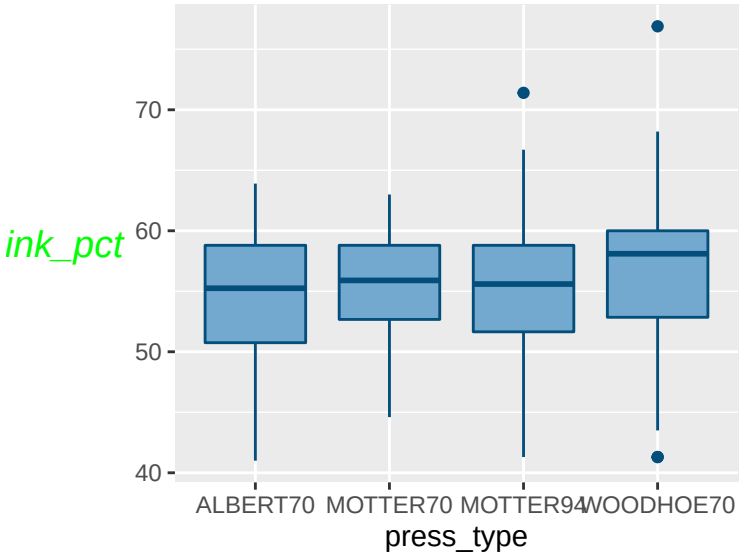
The difference between the two ways is that `theme()` function doesn't change axis name. The text is not displayed and no space is reserved for it. When you set the label to "" in `xlab()` function, the name of the scale is changed and the (empty) text is displayed.

9.7.8 Changing the appearance of axis labels

`axis.title.x`, `axis.title.y` and `axis.title` arguments of `theme()` function controls the label of x, y or both axis.

The reference function is `element_text()` and works at the same way as we saw in *Changing the Appearance of Tick Labels* paragraph.

```
pl_1 +
  theme(axis.title.y=element_text(angle=0, face="italic", size=14, colour = "green"))
```



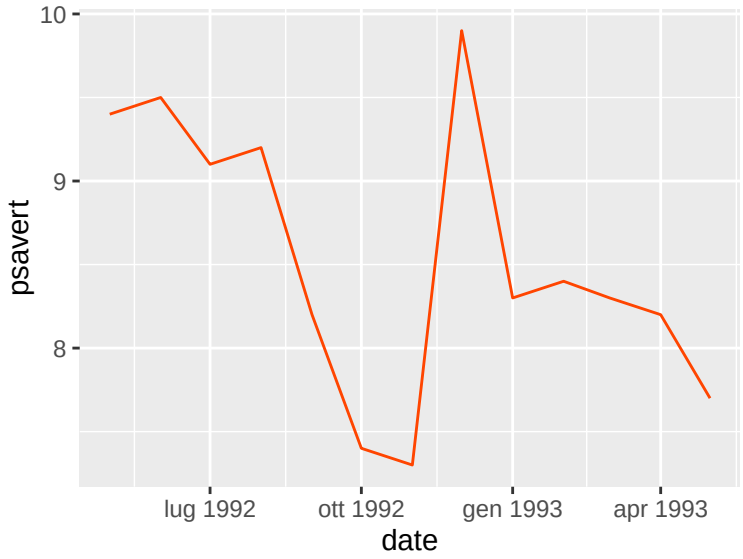
9.8 Using Dates on an Axis

`ggplot2` handles two kinds of time-related objects: dates (objects of class Date) and date-times (objects of class POSIXt). The difference between these is that Date objects represent dates and have a resolution of one day, while POSIXt objects represent moments in time and have a resolution of a fraction of a second. Specifying the breaks is similar to with a numeric axis, the main difference is in specifying the sequence of dates to use.
We use a subset of the `economics` data, provided in `ggplot2` package which ranges from may 1992 to june 1993.

```
econ <- subset(economics, date >= as.Date("1992-05-01") & date < as.Date("1993-06-01"))
```

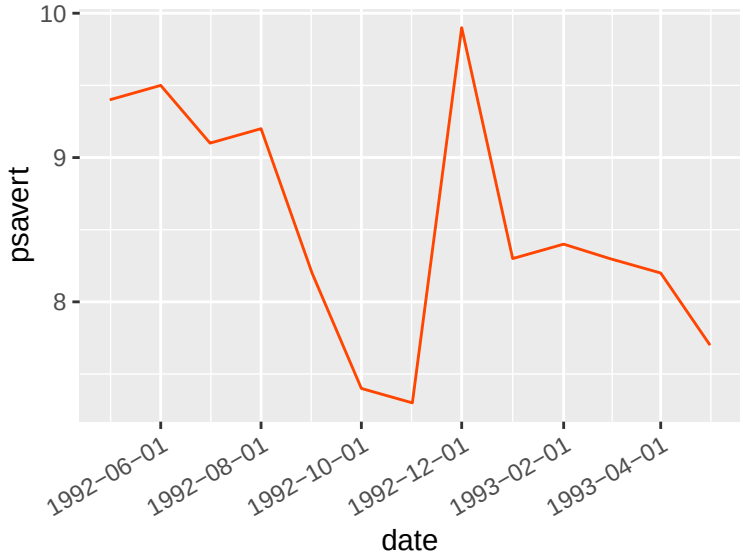
Suppose we want to generate the time series plot of personal savings rate:

```
pl_4 <- ggplot(econ, aes(x=date, y=psavert)) +
  geom_line(colour = "orangered")
pl_4
```



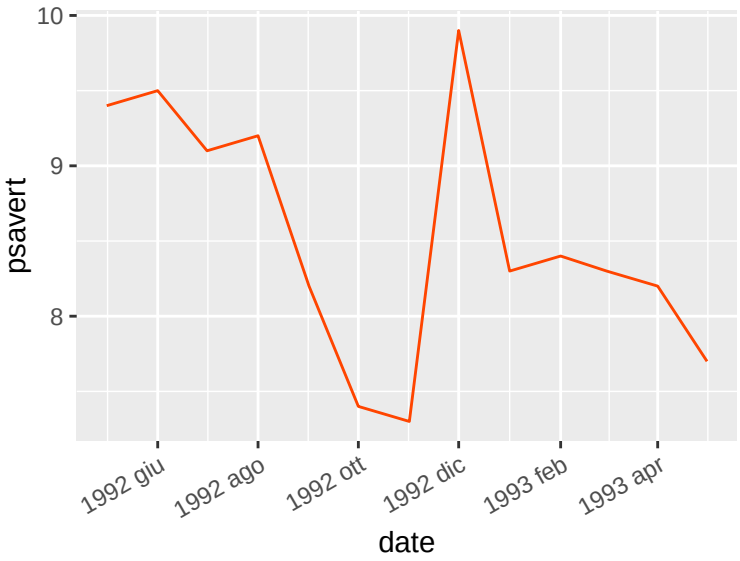
`scale_x_date()` function handle with date scales, the argument `date_breaks` specifies the distance between breaks.

```
# Use breaks, and rotate text labels
pl_4 +
  scale_x_date(date_breaks= "2 month") +
  theme(axis.text.x = element_text(angle=30, hjust=1))
```



Notice that the formatting of the breaks changed. You can specify the formatting by using the `date_format()` function from the `scales` package as `label` value:

```
pl_4 +
  scale_x_date(date_breaks= "2 month", labels=date_format("%Y %b")) +
  theme(axis.text.x = element_text(angle=30, hjust=1))
```



The format specified is `"%Y %b"`, which results like `"1992 Jun"`.

The format specified is to be put in a string that is passed to `date_format()`, and the format specifiers will be replaced with the appropriate values. For example, if you use `"%B %d, %Y"`, it will result in labels like `"June 01, 1992"`.

The following table lists the most important date-time format:

Option	Description
%Y	Year with century (2012)
%y	Year without century (12)
%m	Month as a decimal number (08)
%b	Abbreviated month name in current locale (Aug)
%B	Full month name in current locale (August)
%d	Day of month as a decimal number (04)
%U	Week of the year as a decimal number, with Sunday as the first day of the week (00–53)
%W	Week of the year as a decimal number, with Monday as the first day of the week (00–53)
%w	Day of week (0–6, Sunday is 0)
%a	Abbreviated weekday name (Thu)
%A	Full weekday name (Thursday)

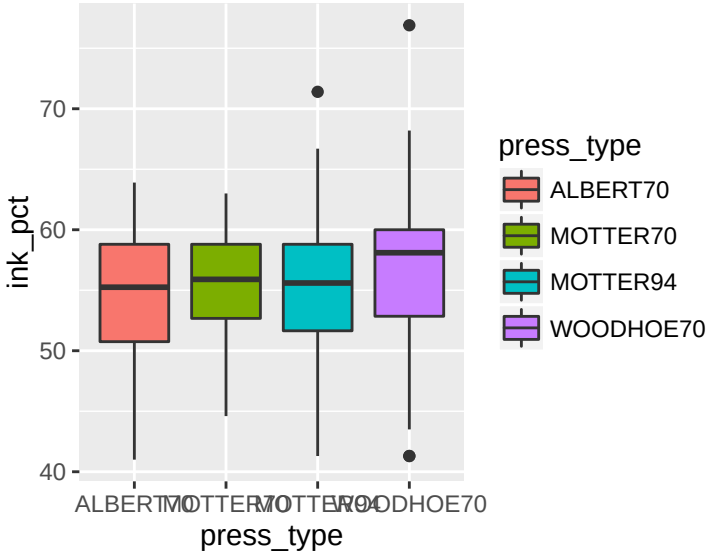
10 Legend Customization

```
require(ggplot2)
require(qdata)
require(grid)
data(bands)
```

A legend is a guide which tells us how to map visual (aesthetic) properties back to original data values. `ggplot2` automatically generates the legend according to the aesthetics mapped.

Let us see an example.
Considering the `bands` data. Supposing you are interested in the relationship between humidity and viscosity by band type, you can build four box plots to compare distributions:

```
pl_1 <- ggplot(data=bands, aes(x=press_type, y=ink_pct, fill=press_type)) +
  geom_boxplot()
pl_1
```



As you can see from the plot, the legend is automatically built showing `press_type` variable levels. In this case, `press_type` variable is mapped to `fill` aesthetic. When a variable is mapped to an aesthetic it is automatically set to a scale. The default scale in this case is `scale_fill_discrete()`, but there are also other scales according to aesthetics mapped, such as `colour` (for lines and points) or `shape` (for points).

Scales, `theme()` and `guides()` functions are the tools of `ggplot2` used to customize the appearance of the legend components.

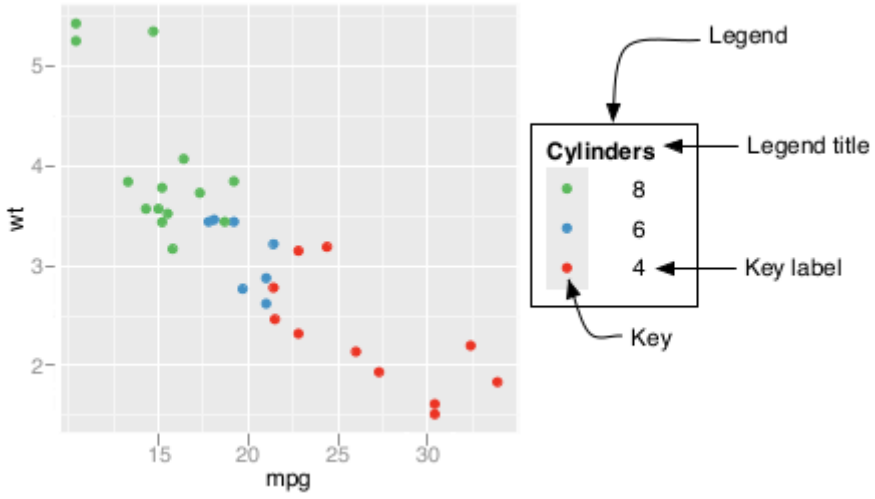


Figure 14:

In the following paragraphs we will see how to handle with the most common questions on legend customization.

10.1 Removing the legend

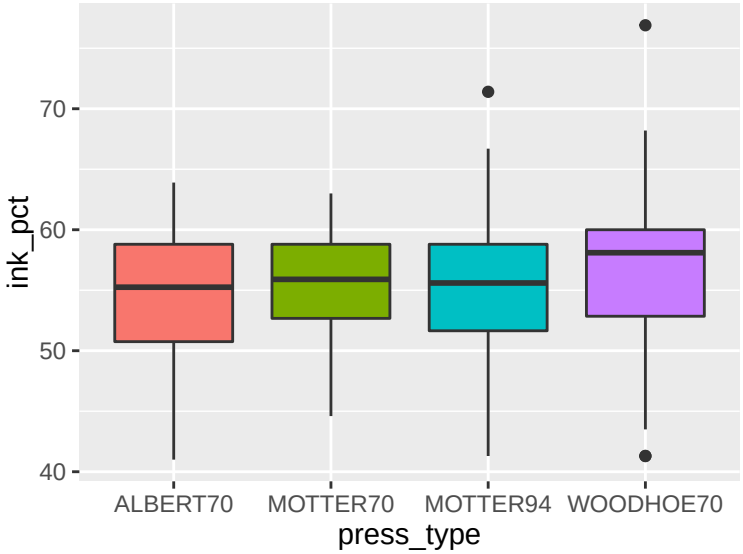
Sometimes a legend is redundant, or it is supplied in another graph that will be displayed with the current one. In these cases, it can be useful to remove the legend from a graph.
You can remove the legend in three ways:

- using `guides()` function and specify the scale that should have its legend removed
- setting `guide` argument of the `scale_xxx_xxx()` function where the aestethic is mapped to `FALSE`
- use the theming system specifying `legend.position` argument to `"none"`

In the previous example, the colors provide the same information that is on the x-axis, so the legend is unnecessary:

```
pl_1 +
  guides(fill=FALSE)
pl_1 +
  scale_fill_discrete(guide=FALSE)
pl_1 +
  theme(legend.position="none")
```

In this case, the previous three command lines produce the same result:



If you have more than one aesthetic mapped with a legend (color and shape, for example), the third is the better option because this will remove legends for all of them.

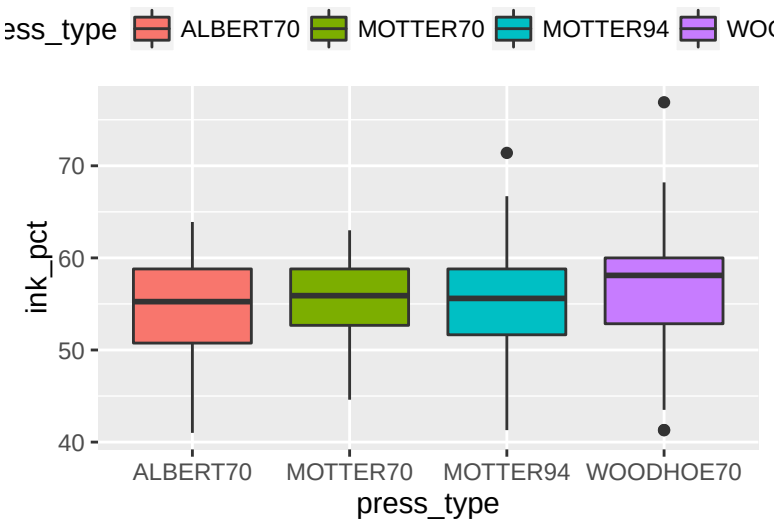
Notice that with the legend removed, the area used for graphing the data is larger.

10.2 Changing the position of a legend

If you want to move the legend from its default place, on the right side, you have to use `theme()` function. The legend can be put on the top, left, right, or bottom by using one of those strings as value of `legend.position` argument.

Suppose we want to place the position on top:

```
pl_1 +
  theme(legend.position="top")
```



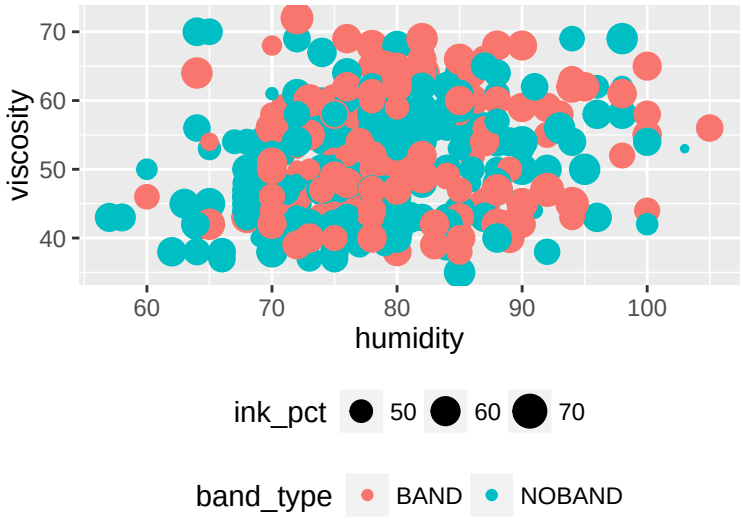
The allowed values for the argument `legend.position` are : `"left"`, `"top"`, `"right"`, `"bottom"`.

Let us consider a plot with multiple legends in plot.
Suppose we want to analyze the relationship between humidity, viscosity, ink percentage and band type in `bands` dataset:

```
pl_2 <- ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, size=ink_pct, colour=band_type)) +
  geom_point()
```

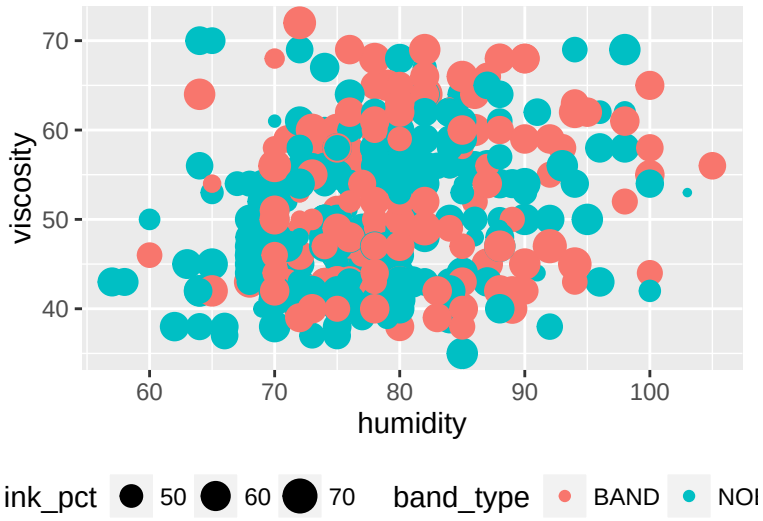
Suppose that we want also to move the legend to bottom:

```
pl_2 +
  theme(legend.position="bottom")
```



Looking at the plot we see that legends are represented in different lines. If you want to set legends in the same line set `legend.box` argument of `theme()` function as `"horizontal"`:

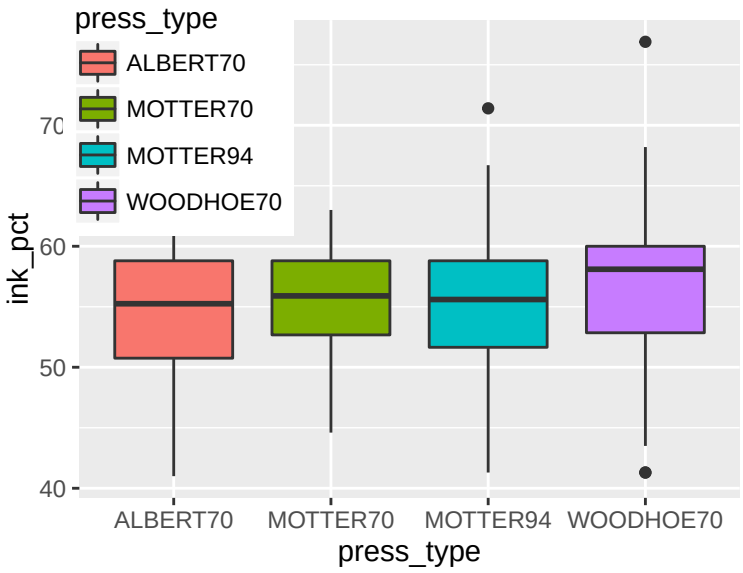
```
pl_2 +
  theme(legend.position="bottom", legend.box = "horizontal")
```



The legend can also be placed inside the graphing area by specifying a coordinate position. In this case, the argument `legend.position` has to be set to a numeric vector c(x,y), whose values should be between 0 and 1. In particular, the coordinate space starts at (0, 0) in the bottom left and goes to (1, 1) in the top right.

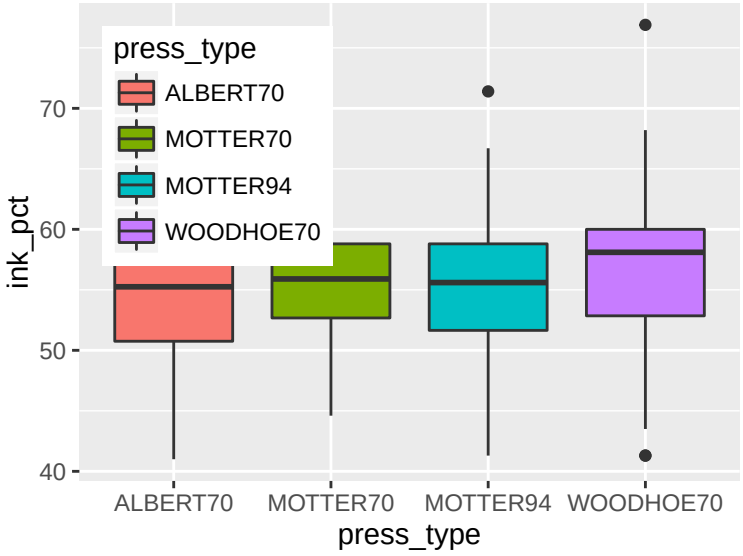
Let us see an example, considering the differences of ink percentage accordingly to the type of press in bands dataset:

```
pl_1 +
  theme(legend.position=c(0.15,0.80))
```



In the previous plot, the justification of the legend positon is centered. This means that the center of the legend (.5, .5) is placed at the coordinates set to `legend.position` argument. However, it is often useful to specify a different point. You can use `legend.justification` argument to set which part of the legend box is set to the position specified in `legend.position` argument. Let us set left-top justification for `legend.justification`:

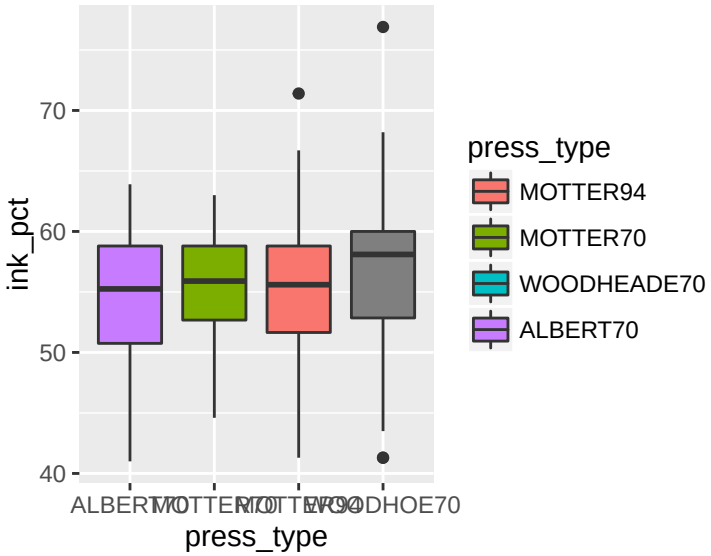
```
pl_1 +
  theme(legend.position=c(0,1), legend.justification=c(0,1))
```



10.3 Change the order of items in a legend

To change the order of the items in a legend set the limits in the scale to the desired order:

```
pl_1 +
  scale_fill_discrete(limits=c("MOTTER94", "MOTTER70", "WOODHEADE70", "ALBERT70"))
```

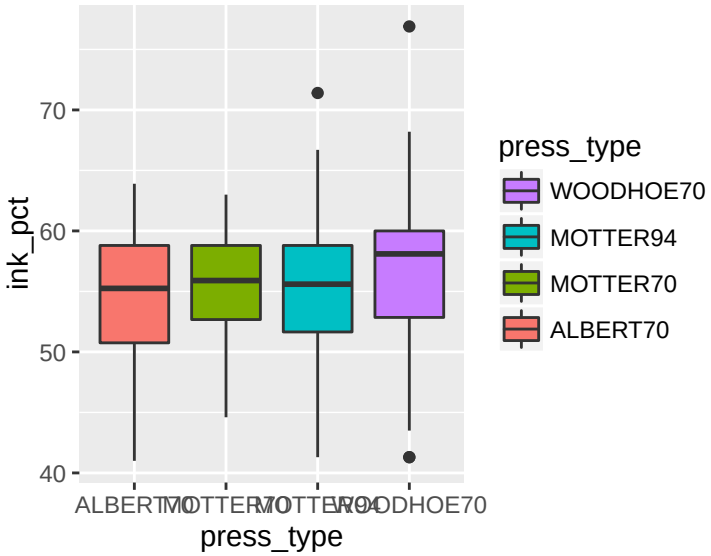


Note that the order of the items on the x-axis did not change. To do that, you would have to set the limits of `scale_x_discrete()` or change the data to have a different factor level order. For more details see *Change axis order* paragraph in *Axes Customization* chapter.

10.3.1 Reversing the order of items in a legend

Add `guides(fill=guide_legend(reverse=TRUE))` to reverse the order of the legend (for other aesthetics, replace fill with the name of the aesthetic, such as colour or size):

```
pl_1 +
  guides(fill=guide_legend(reverse=TRUE))
```

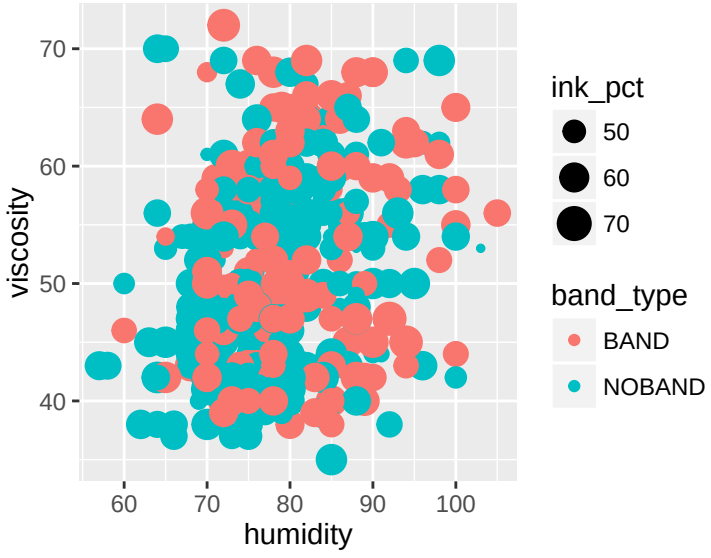


10.3.2 Change the order for multiple guides

Let us consider the example about the relationship between humidity, viscosity, ink percentage and band type in `bands` dataset.

In this case, you have multiple guides. Suppose you want to change the guides order, use `guides()` function in this way:

```
pl_2 +
  guides(color = guide_legend(order=2),
    size = guide_legend(order=1))
```



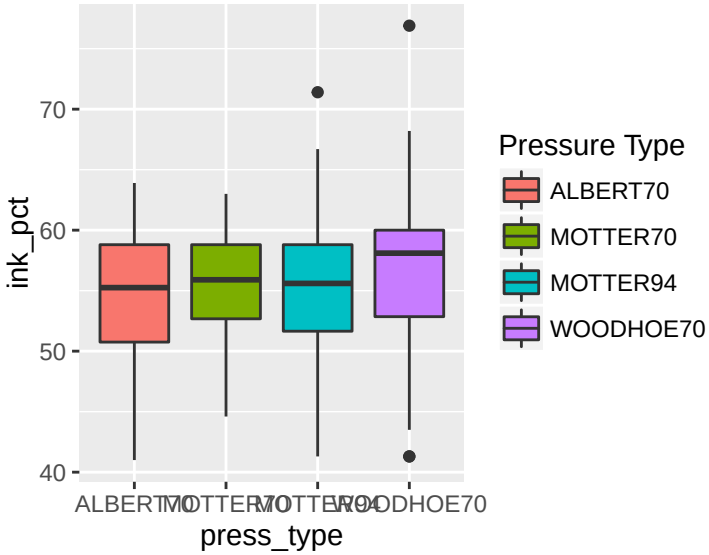
10.4 Change a legend title

The title of a legend can be changed in three ways:

- using `labs()` function and set the value of `fill`, `colour`, `shape`, or whatever aesthetic is appropriate for the legend
- in the scale specification setting `name` argument
- using `guides()` function

```
pl_1 +
  labs(fill="Pressure Type")
pl_1 +
  scale_fill_discrete(name="Pressure Type")
pl_1 +
  guides(fill=guide_legend(title="Pressure Type"))
```

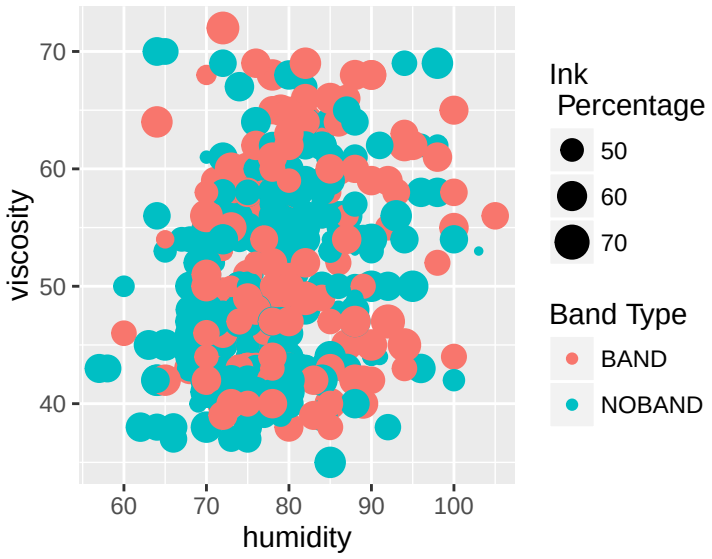
The previous three command lines proceduce the same result:



The controlling of the legend title by using the `guides()` function is a little more verbose, but it can be useful when you're already using it to control other properties.

If there are multiple variables mapped to aesthetics with a legend (those other than x and y), you can set the title of each individually.

```
pl_2 + guides(colour=guide_legend(title="Band Type"), size=guide_legend(title="Ink \n Percentage"))
```

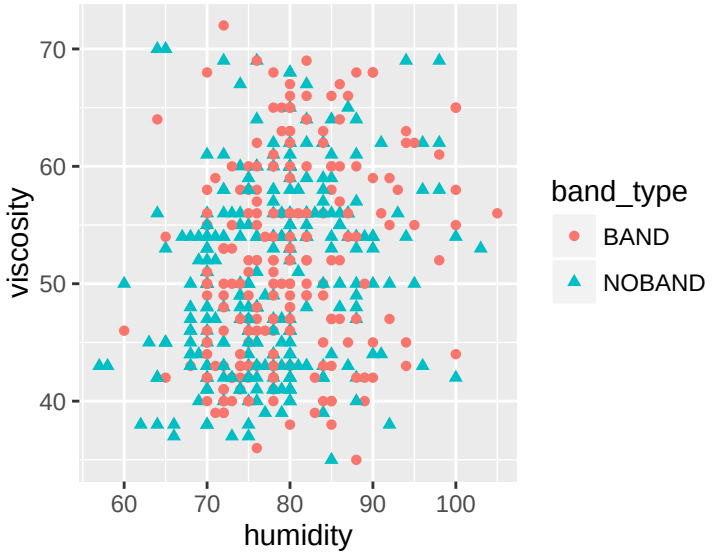


–>

If you have one variable mapped to two separate aesthetics, the default is to have a single legend that combines both.

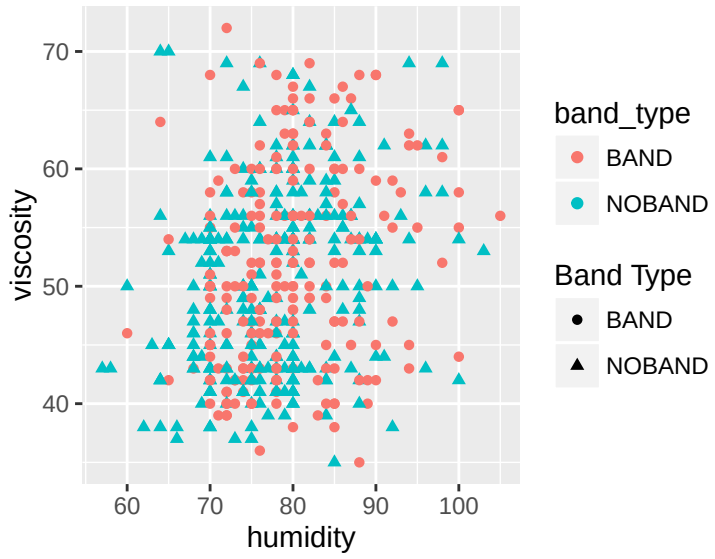
Let us analyze the relationship between humidity, viscosity and band_type in `bands` dataset:

```
pl_3 <- ggplot(data=bands, mapping=aes(x=humidity, y=viscosity, shape=band_type, colour=band_type)) +
  geom_point()
pl_3
```

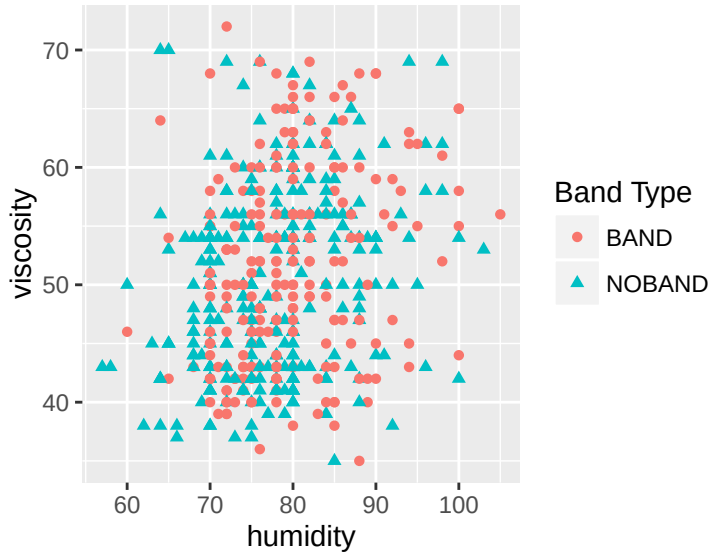


To change the title, you need to set the name for both of them. If you change the name for just one, it will result in two separate legends:

```
pl_3 +
  labs(shape = "Band Type") # two separate legend
```




```
pl_3 +
  labs(shape = "Band Type", colour = "Band Type") # one legend
```

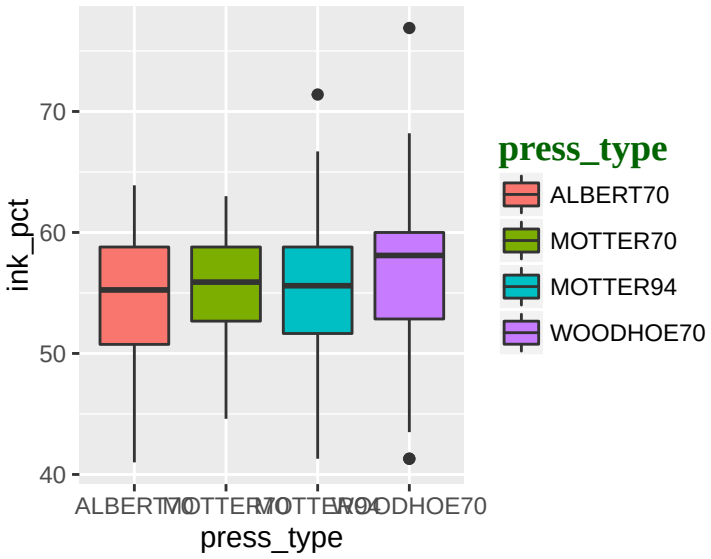


10.5 Changing the appearance of a legend

10.5.1 Changing the appearance of a legend title

The appearance of the legend title, like size, style and the font family can be changed by setting `element_text()` function to `legend.title` argument of `theme()` function in this way:

```
pl_1 +
  theme(legend.title=element_text(face="bold", family="Times", colour="darkgreen", size=14))
```

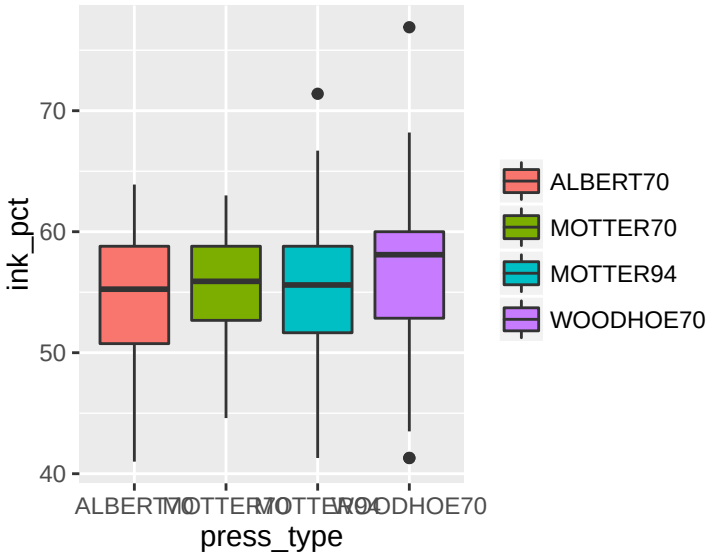


10.5.2 Removing a legend title

You can remove legend title by using `scale_xxx_xxx()` or `guides()` functions:

```
pl_1 +
  guides(fill=guide_legend(title=NULL))
pl_1 +
  scale_fill_discrete(guide = guide_legend(title=NULL))
```

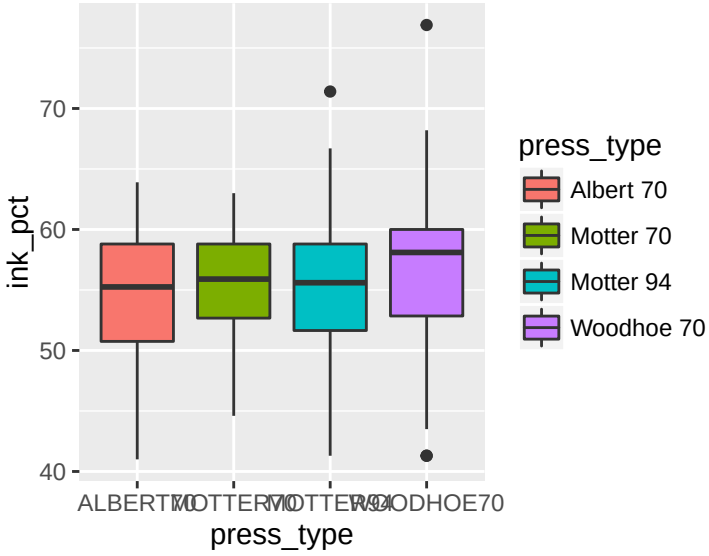
The previous command lines produce the same result:



10.5.3 Change the labels text in a legend

You can change the labels text in a legend by setting `labels` argument in the `scale_xxx_xxx()` function:

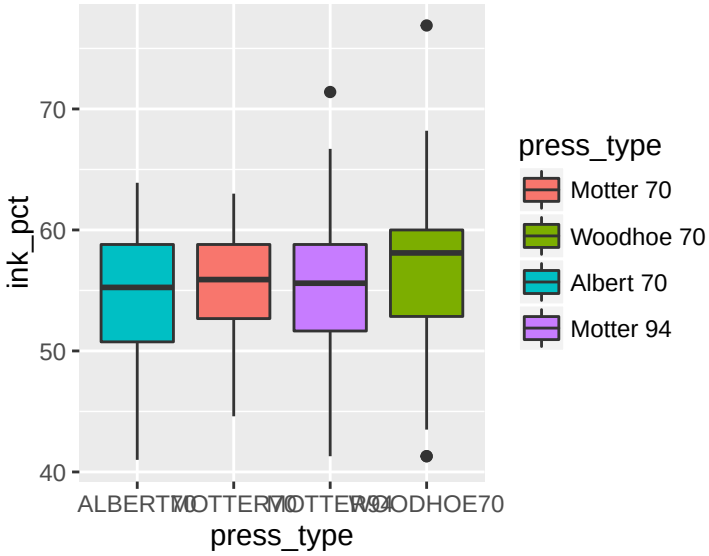
```
pl_1 +
  scale_fill_discrete(labels=c("Albert 70", "Motter 70", "Motter 94", "Woodhoe 70"))
```



Note that the labels on the x-axis did not change. To do that, you would have to set the labels of `scale_x_discrete()` (see chapter *Axes Customization*, paragraph *Changing the Text of Tick Labels*), or change the data to have different factor level names.

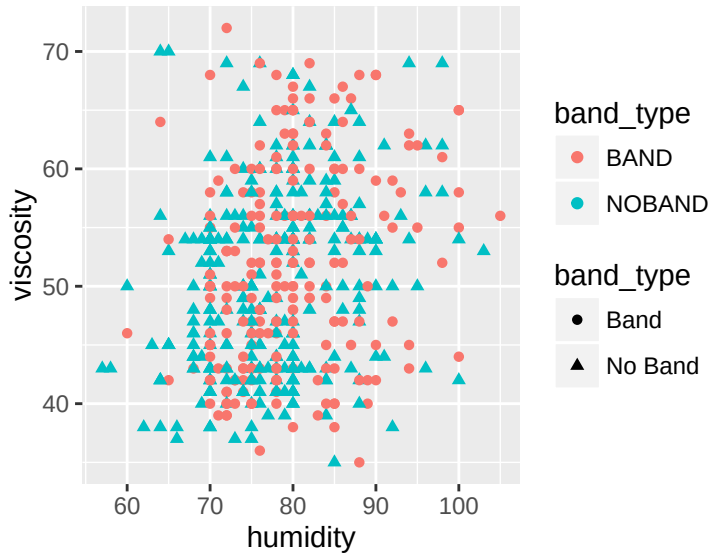
If you are also changing the order of items in the legend, the labels are matched to the items by position. In this example we change the item order, and make sure to set the labels in the same order:

```
pl_1 +
  scale_fill_discrete(limits=c("MOTTER70", "WOODHOE70", "ALBERT70", "MOTTER94"),
    labels=c("Motter 70", "Woodhoe 70", "Albert 70", "Motter 94"))
```

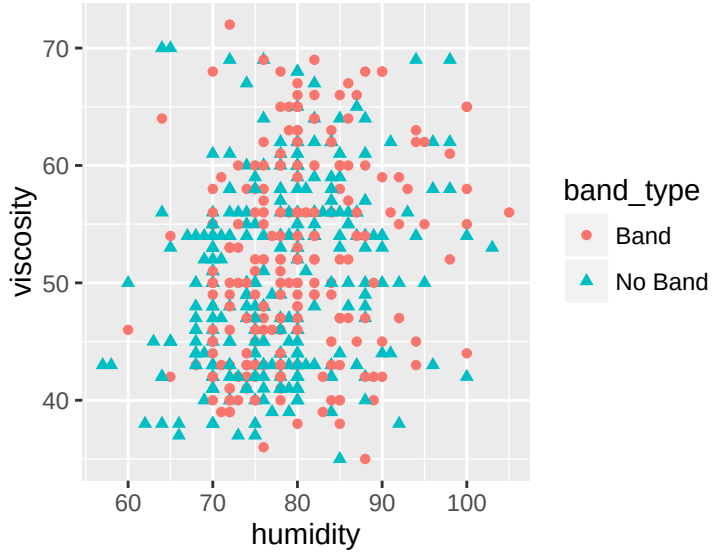


If you have one variable mapped to two separate aesthetics, the default is to have a single legend that combines both. If you want to change the legend labels, you must change them for both scales; otherwise you will end up with two separate legends:

```
# Change the labels for one scale
pl_3 +
  scale_shape_discrete(labels=c("Band", "No Band"))
```



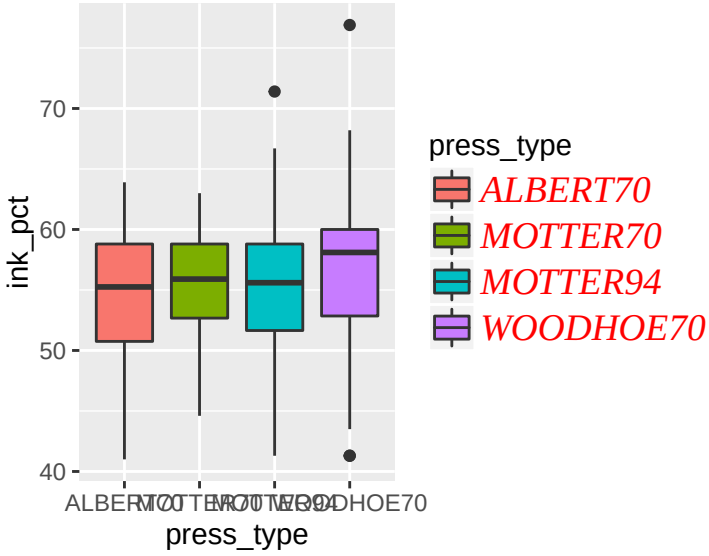

```
# Change the labels for both scales
pl_3 +
  scale_shape_discrete(labels=c("Band", "No Band")) +
  scale_colour_discrete(labels=c("Band", "No Band"))
```



10.5.4 Changing the appearance of a legend label

The appearance of the legend labels, like size, style and the font family can be changed by setting `element_text()` function to `legend.text` argument of `theme()` function in this way:

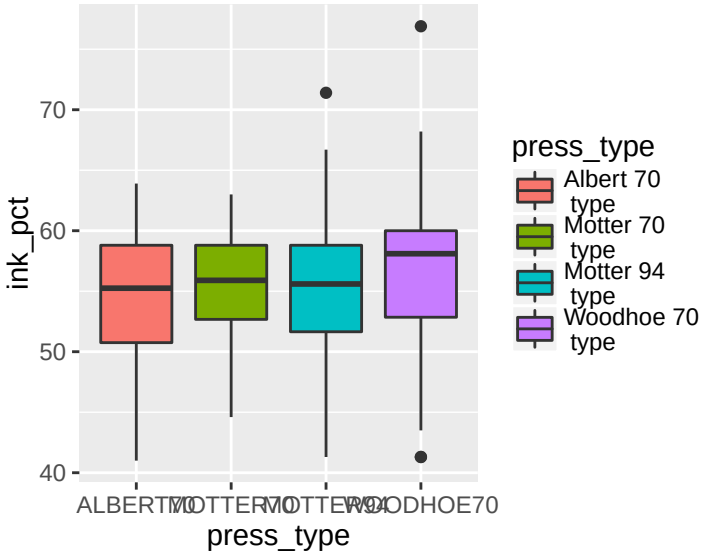
```
pl_1 +
  theme(legend.text=element_text(face="italic", family="Times", colour="red", size=14))
```



10.5.5 Using labels with a multiple lines of text

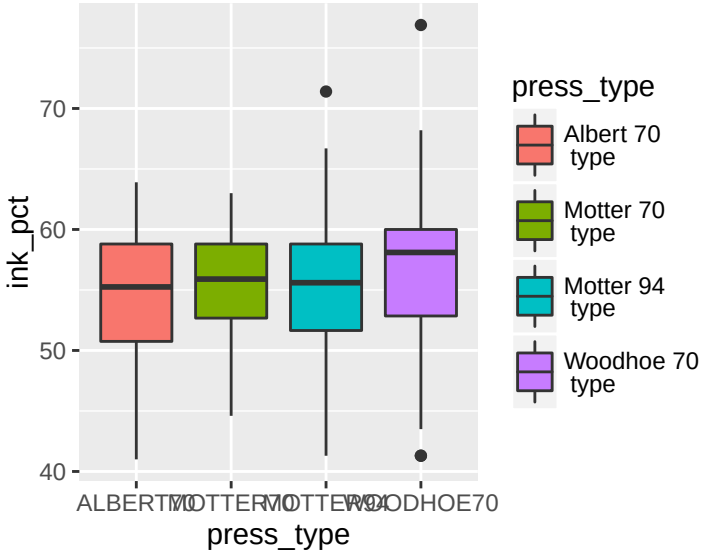
If you want to use legend labels that have more than one line of text, set the labels in the scale, using `"\n"` to represent a newline. In this example, we'll use `scale_fill_discrete()` to control the legend for the fill scale:

```
pl_1 +
  scale_fill_discrete(labels=c("Albert 70 \n type", "Motter 70 \n type", "Motter 94 \n type", "Woodhoe
```



As you can see in the previous plot, with the default settings the lines of text will run into each other when you use labels that have more than one line. To deal with this problem, you can increase the height of the legend keys and decrease the spacing between lines, using `theme()`. To do this, you will need to specify the height using the `unit()` function:

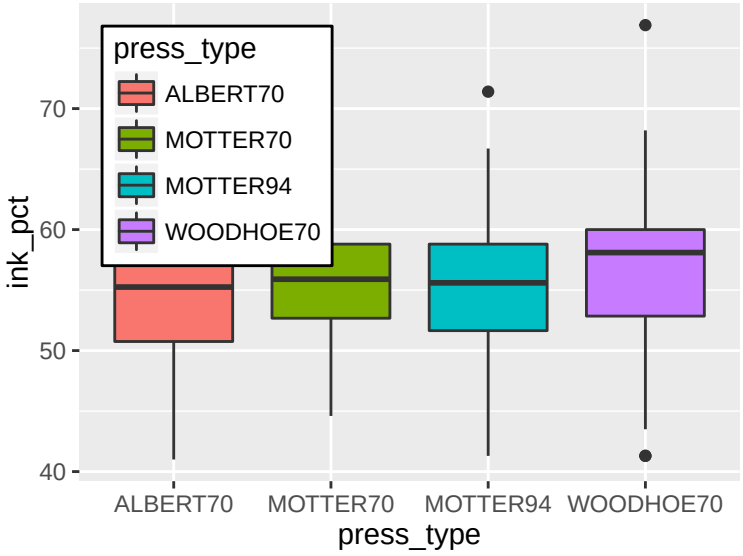
```
pl_1 +
  scale_fill_discrete(labels=c("Albert 70 \n type", "Motter 70 \n type", "Motter 94 \n type", "Woodhoe
  theme(legend.text=element_text(lineheight=.8),
    legend.key.height=unit(1, "cm"))
```



10.5.6 Customize legend background

When placing the legend inside of the graphing area, as seen in *Changing the position of a legend* paragraph, it may be helpful to add an opaque border to set it apart by setting `legend.background` argument of `theme` function:

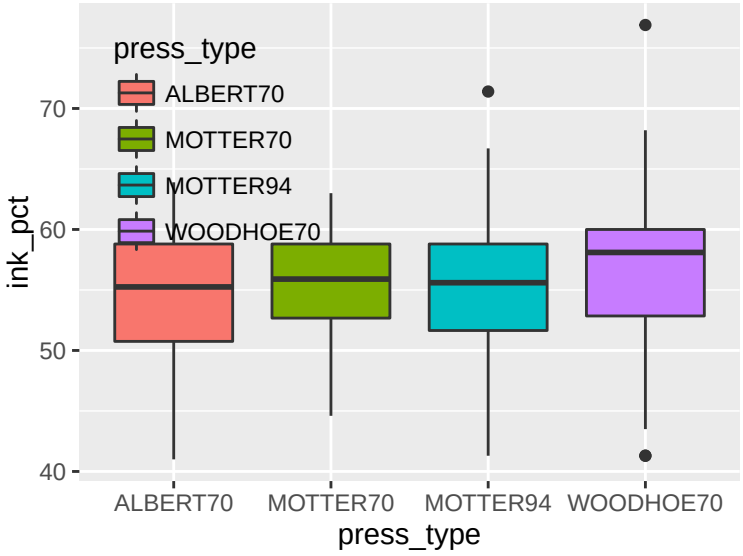
```
pl_1 +
  theme(legend.position=c(0,1), legend.justification=c(0,1),
    legend.background=element_rect(fill="white", colour="black"))
```



`element_rect()` function allows us to customize backgrounds and borders.

You can also remove the border around its elements so that it blends in by using `element_blank()` function:

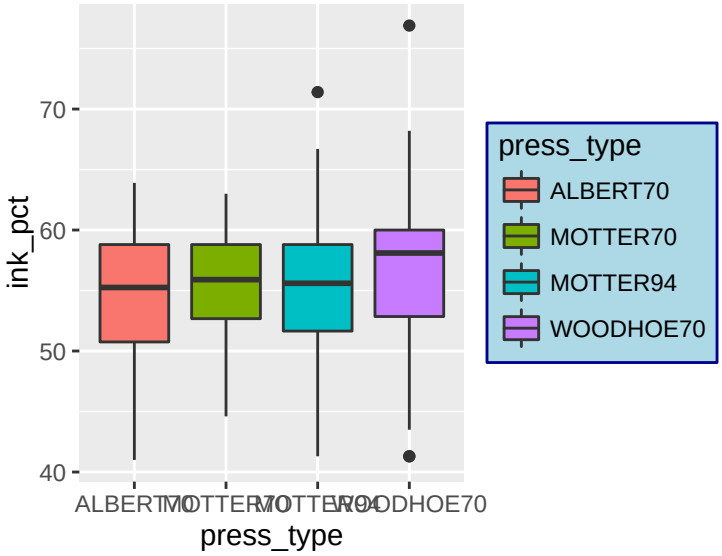
```
pl_1 +
  theme(legend.position=c(0,1),
        legend.justification=c(0,1),
        legend.background=element_blank(),  # Remove overall border
        legend.key=element_blank()) # Remove border around each item
```



`legend.key` argument of `theme()` function refers to the customization of the border around each item.

Legend background can be customized also if legend is out of the plotting area:

```
pl_1 + theme(
  legend.background = element_rect(fill="lightblue", size=0.5, linetype="solid", colour ="darkblue"),
  legend.key  = element_rect(fill="lightblue", size=0.5, linetype="solid", colour ="lightblue"))
```



11 Facets Customization

```
require(ggplot2)
require(dplyr)
require(qdata)
data(bands)
```

Faceting is one of the most useful techniques in data visualization for rendering groups of data alongside each other, and making it easy to compare them. In particular, faceting is a mechanism for automatically laying

out multiple plots in a page, splitting the data into subsets and then plots each subset in a different panel. It is an alternative to the use of aesthetics (like colour, shape or size) to differentiate groups and it is the right choice when groups overlap a lot.

In `ggplot2` there are three functions for faceting:

- `facet_null()`, which is the default and produces a single plot
- `facet_grid()`, which produces a 2d grid of panels defined by variables which form the rows and columns
- `facet_wrap()`, which "wraps" a 1d ribbon of panels into 2d

We mentioned faceting in *Creating a Histogram* chapter. In the following paragraphs we will examine in depth how to handle with the most common questions on faceting customization.

11.1 Splitting data into subplots

Let us review the basic use of `facet_grid()` and `facet_wrap()` faceting functions. Both functions plot subsets of data in separate panels. The difference between them is schematized in the following figure:

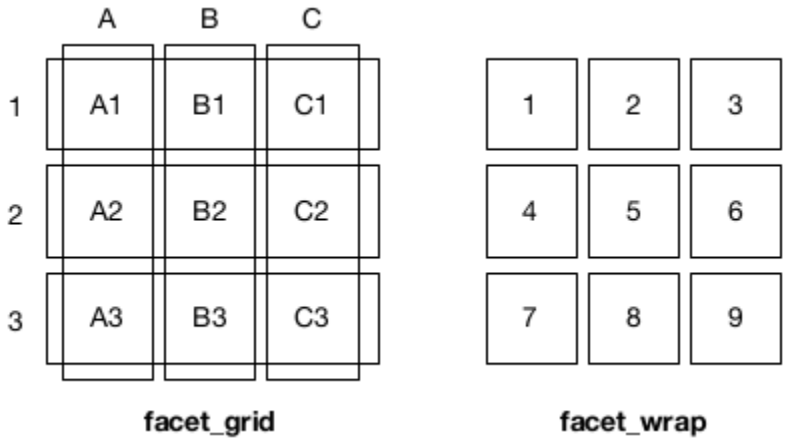
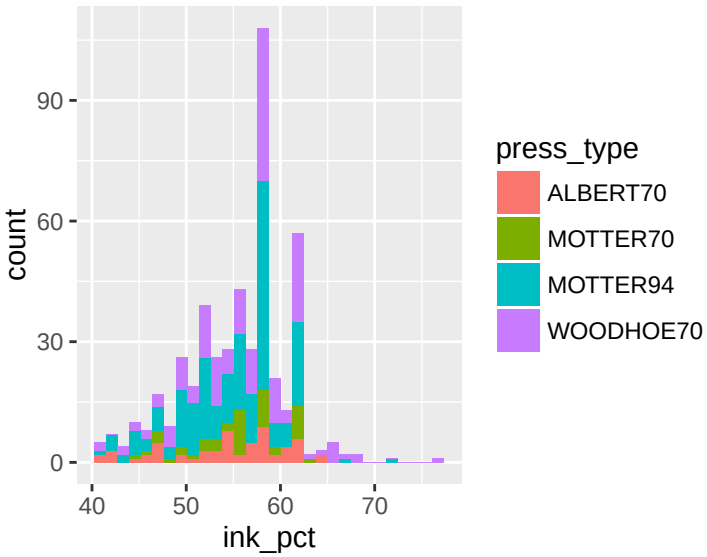


Figure 15:

Suppose you are interested in analysing the distribution of `ink_pct` by categorical variables like `press_type` or `band_type` in `bands` dataset:

```
# ink_pct by presss_type
ggplot(data=bands, mapping=aes(x=ink_pct, fill=press_type)) +
  geom_histogram()
```



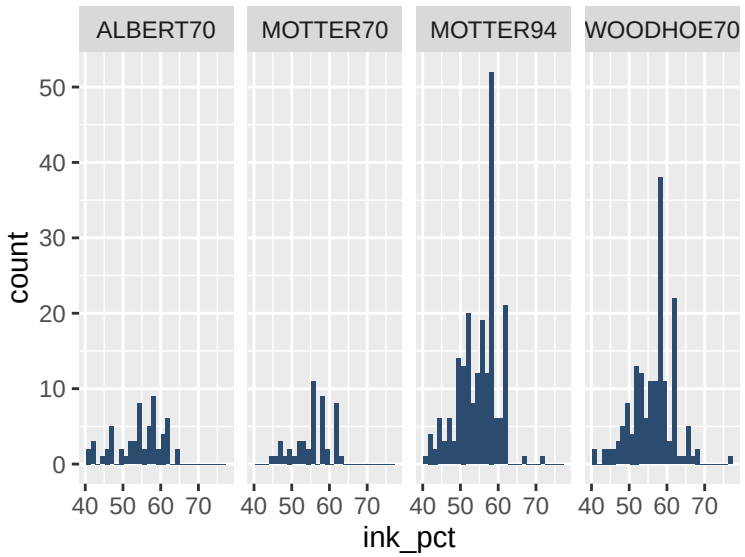
The previous plot is difficult to interpret. It is better to split data in four different histograms by using faceting.

With `facet_grid()`, you can use faceting with one or two variables. A variable to split the data into vertical subpanels and another variable to split it into horizontal subpanels are specified in formula style: `rows ~ columns`. If you want to specify only one dimension (either row or column), you have to indicate with a dot the dimension for which there should be no faceting.

In particular:

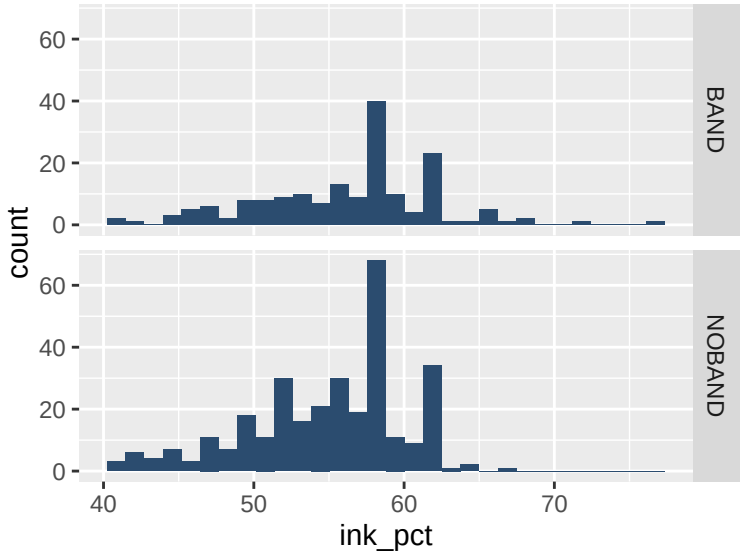
- `. ~ press_type`: spreads the values of `press_type` across the columns. The direction facilitates the comparison of y position because the vertical scales are aligned:

```
# base plot
pl <- ggplot(data=bands, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F")

# Faceted by press_type, in horizontally arranged subpanels
pl +
  facet_grid(. ~ press_type)
```



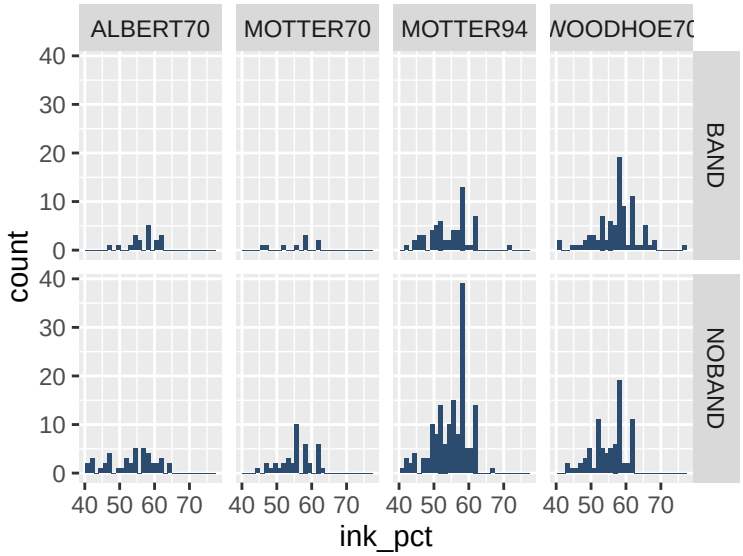
- `band_type ~ .` spreads the values of `band_type` down the rows. The direction facilitates comparison of x position because the horizontal scales are aligned. It is particularly useful for comparing distributions:

```
# Faceted by press_type, in vertically arranged subpanels
pl +
  facet_grid(band_type ~ .)
```



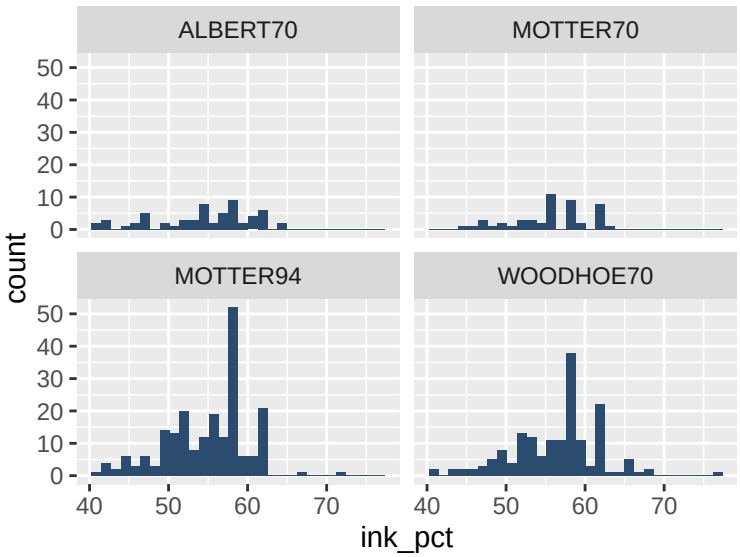
- `band_type ~ press_type` spreads `press_type` across the columns and `band_type` down the rows. Usually we put the variable with the greatest number of levels in the columns, to take advantage of the aspect ratio of the screen:

```
# Faceted by press_type and band_type
pl +
  facet_grid(band_type ~ press_type)
```



With `facet_wrap()`, you can use faceting only with one variable. The subplots are laid out horizontally and wrap around:

```
pl +
  facet_wrap(~ press_type)
```



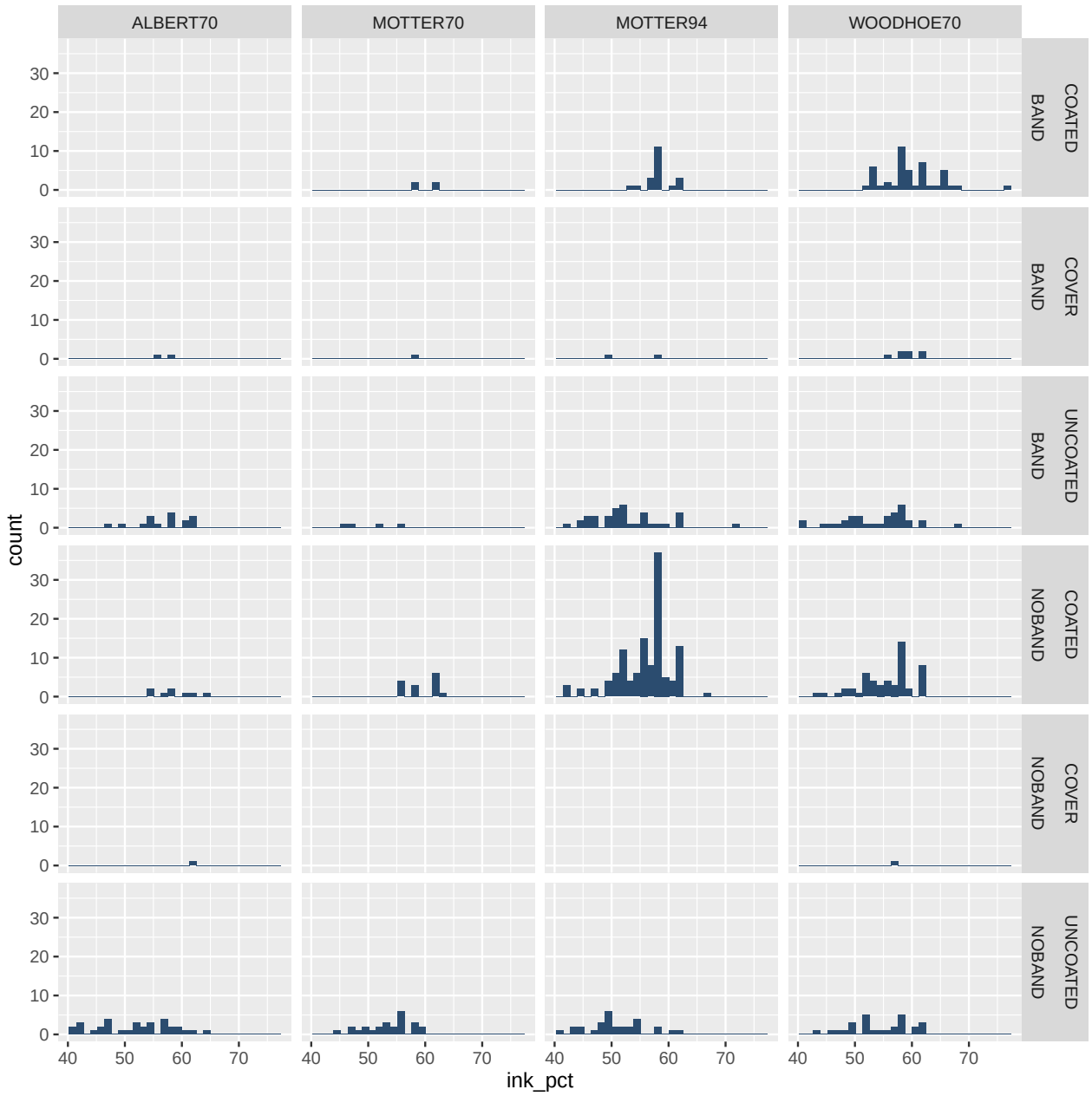
`facet_wrap()` is useful if you have a single variable with many levels and you want to arrange the plots in a more space efficient manner.

The choice of faceting direction depends on the kind of comparison you would like to encourage. For example, if you want to compare heights of bars, it's useful to make the facets go horizontally. If, on the other hand, you want to compare the horizontal distribution of histograms, it makes sense to make the facets go vertically. Sometimes both kinds of comparisons are important, there may not be a clear answer as to which faceting direction is best. It may turn out that displaying the groups in a single plot by mapping the grouping variable to an aesthetic like color works better than using facets. In these situations, you'll have to rely on your judgment.

11.2 Multiple combinations

You can add multiple variables in the rows or columns, by "adding" them together in this way:

```
# by usign facet_grid()
pl +
  facet_grid(band_type + ink_type ~ press_type)
```

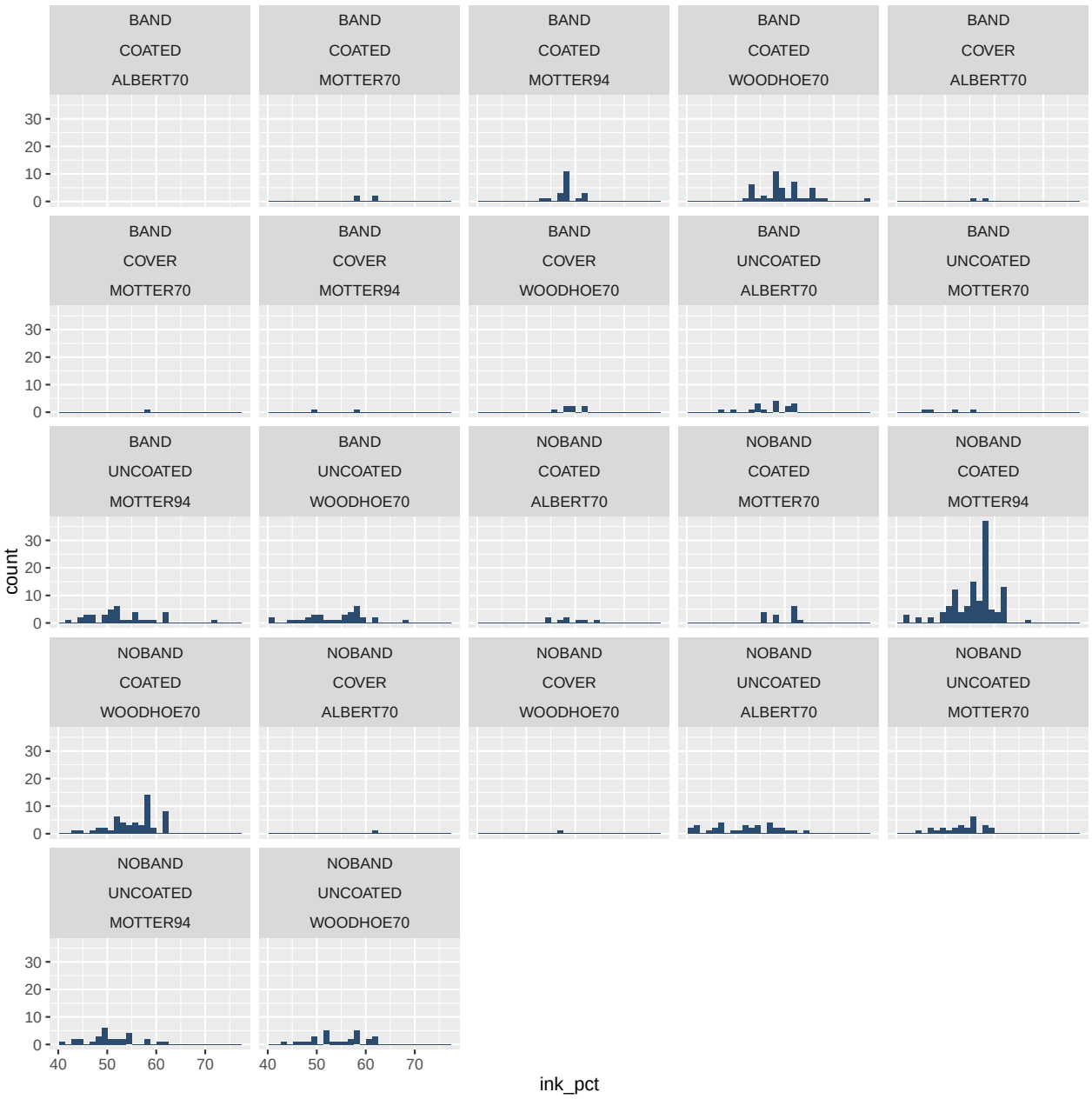


Variables appearing together on the rows or on columns are nested in the sense that only combinations that appear in the data will appear in the plot. Variables which are specified on rows and columns will be crossed: all combinations will be shown, inclusing that those doesn't appear in the original dataset: this may result in empty panels.

–>

Multiple combinations are possible also by using `facet_wrap()`:

```
# by using facet_wrap()
pl +
  facet_wrap(~ band_type + ink_type + press_type)
```

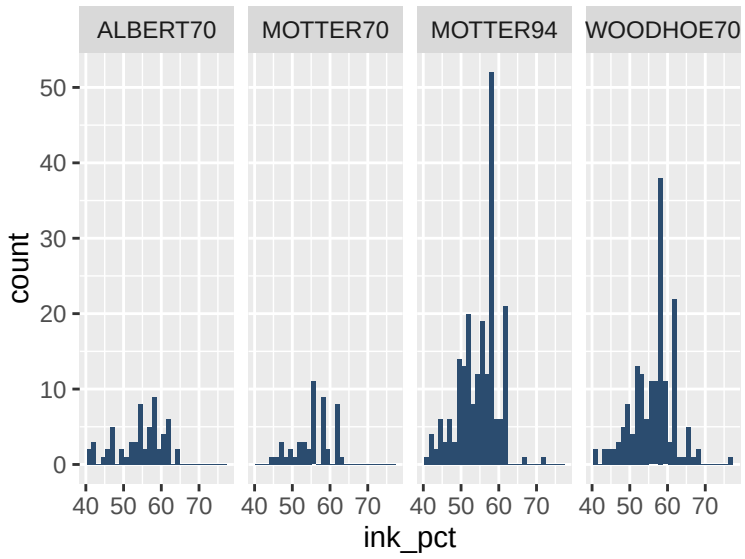


Also with `facet_wrap()` only combinations of variables that appear in the data will appear in the plot.

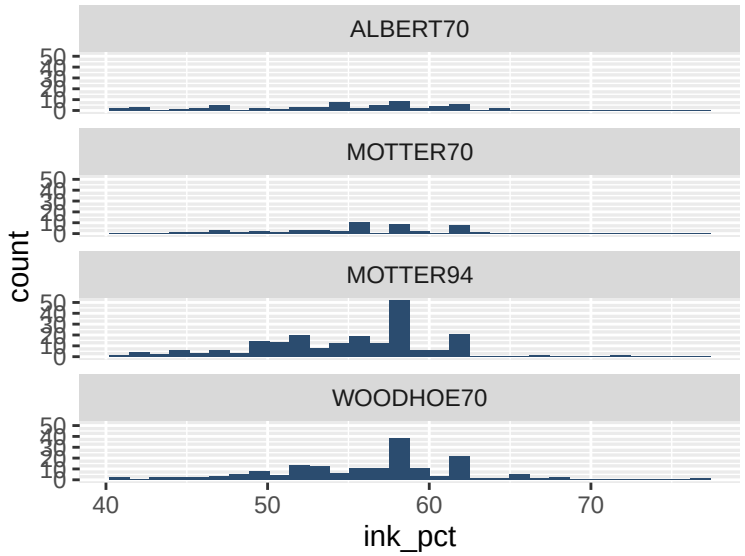
11.3 Change the order of plot in faceting

With `facet_wrap()`, the default is to use the same number of rows and columns. To change this, you can pass a value for `nrow` or `ncol` arguments:

```
pl +
  facet_wrap(~ press_type, nrow = 1)
```

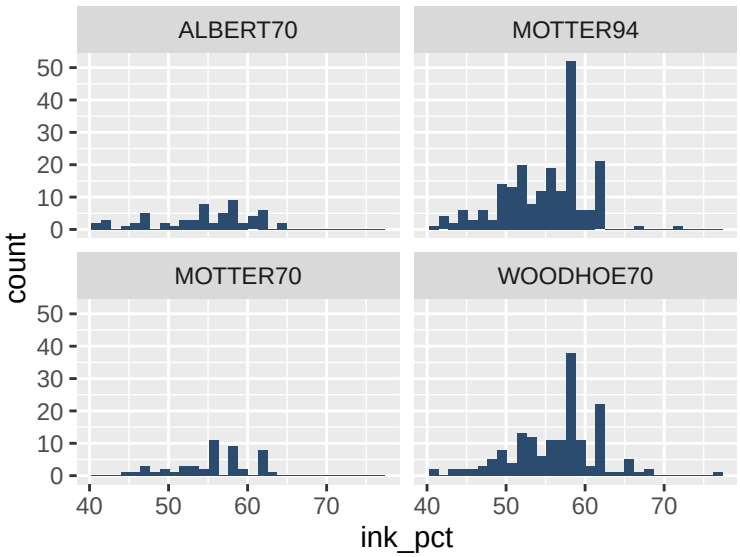



```
pl +
  facet_wrap(~ press_type, ncol = 1)
```

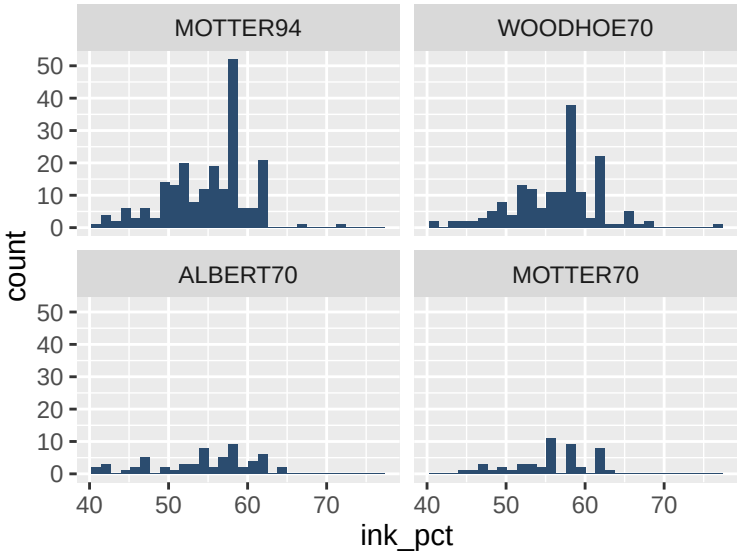


Moreover, the default wrapping direction is horizontal, but you can change it by setting `dir` argument:

```
pl +
  facet_wrap(~ press_type, dir = "v")
```




```
pl + facet_wrap(~ press_type, as.table = FALSE)
```



as.table non lo capisco (valido sia x facet_wrap che per facet_grid) –>

11.4 Using facets with different axes

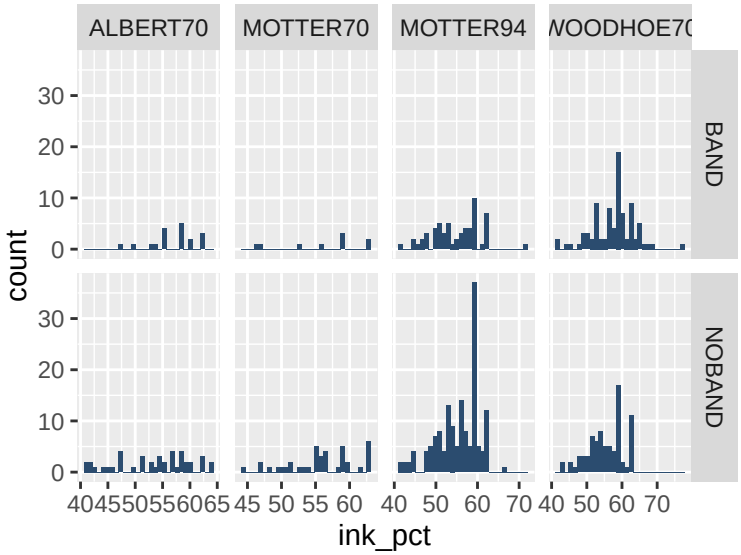
If you want subplots with different ranges or items on their axes, set `scales` argument. It is available for both `facet_wrap()` and `facet_grid()`.
It can be set as:

- `"fixed"`: default, x and y scales are fixed across all panels
- `"free_x"`: x scale is free and y scale is fixed
- `"free_y"`: y scale is free and x scale is fixed
- `"free"`: x and y scales vary across panels

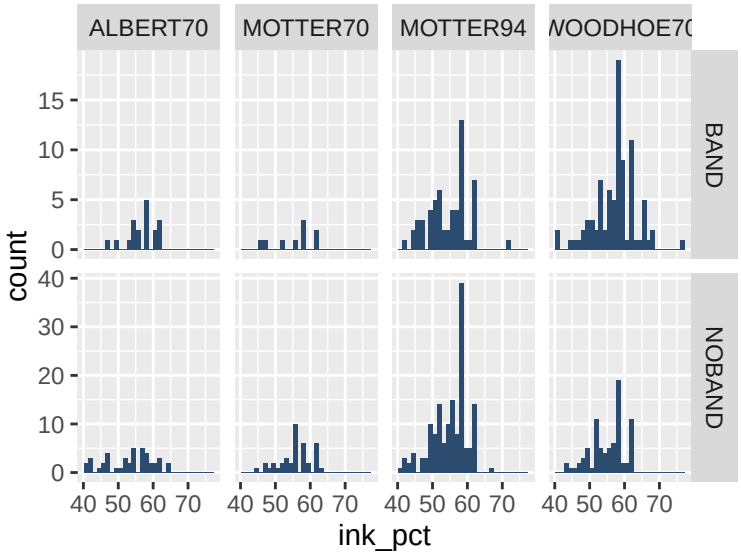
Let us see how `scales` argument works with `facet_grid()`:

```
# free x scale
pl +
```

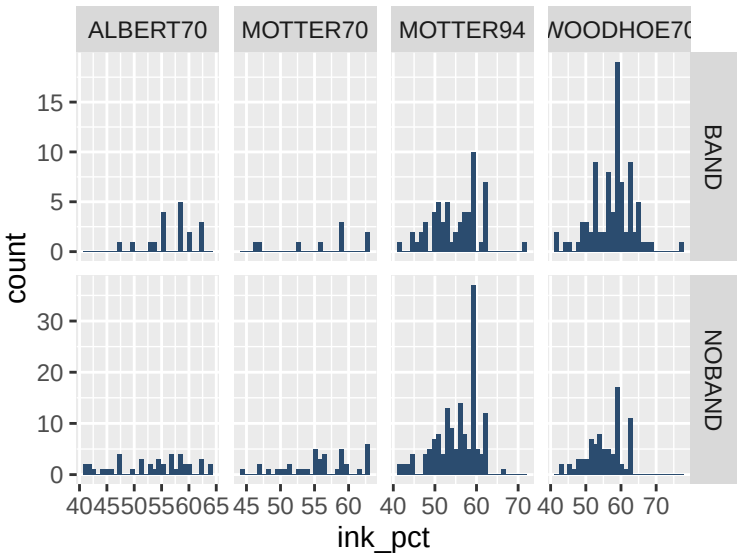
```
  facet_grid(band_type ~ press_type, scales = "free_x")
```




```
# free y scale
pl +
  facet_grid(band_type ~ press_type, scales = "free_y")
```



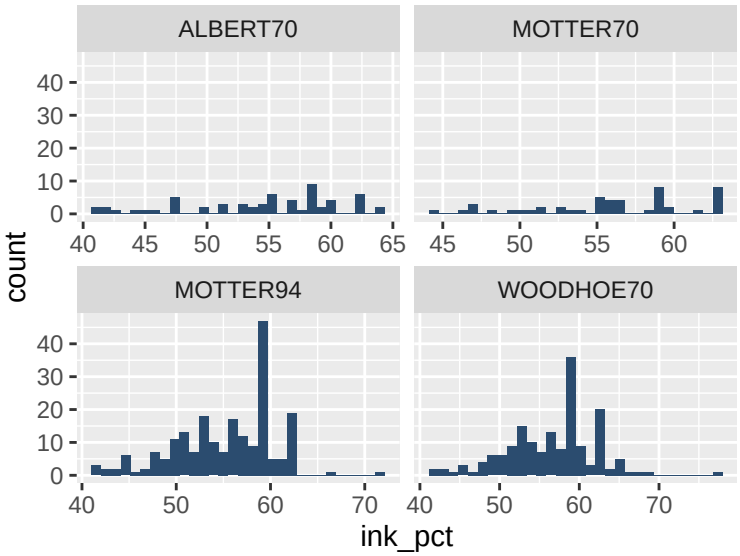

```
# free x and y scales
pl +
  facet_grid(band_type ~ press_type, scales = "free")
```



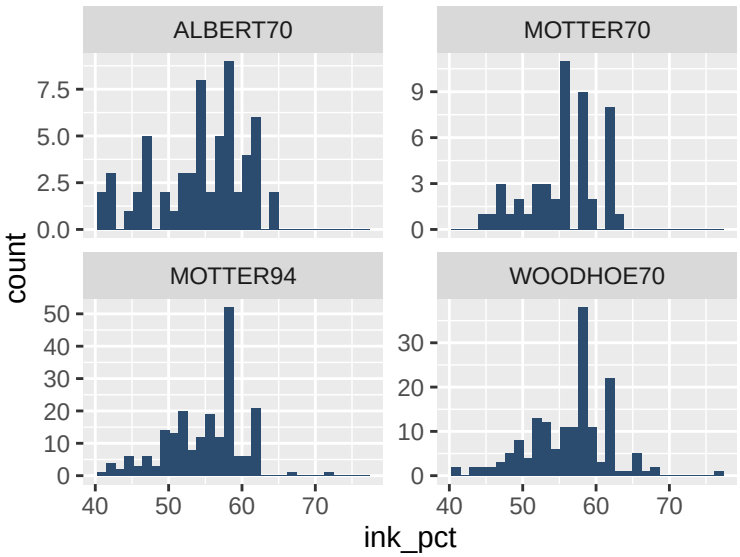
It's not possible to directly set the range of each row or column, but you can control the ranges by dropping unwanted data (to reduce the ranges), or by adding `geom_blank()` (to expand the ranges).

Let us see how `scales` argument works with `facet_wrap()`:

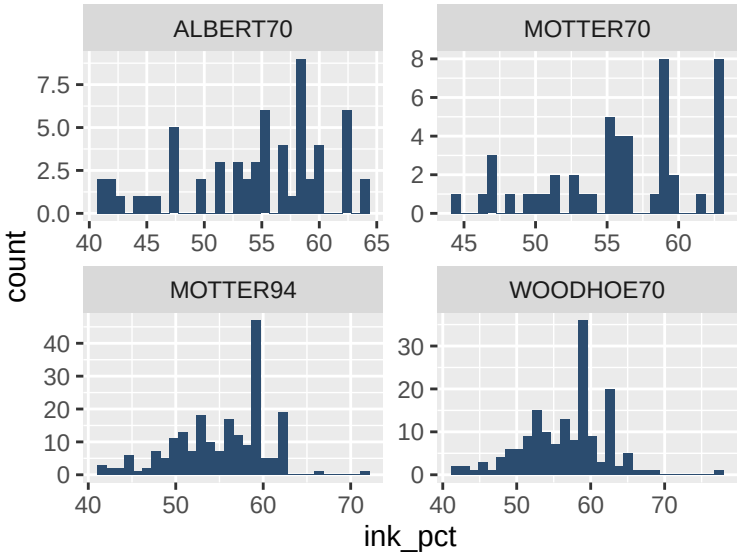
```
# free x scale
pl +
  facet_wrap( ~ press_type, scales = "free_x")
```




```
# free y scale
pl +
  facet_wrap( ~ press_type, scales = "free_y")
```




```
# free x and y scales
pl +
  facet_wrap( ~ press_type, scales = "free")
```

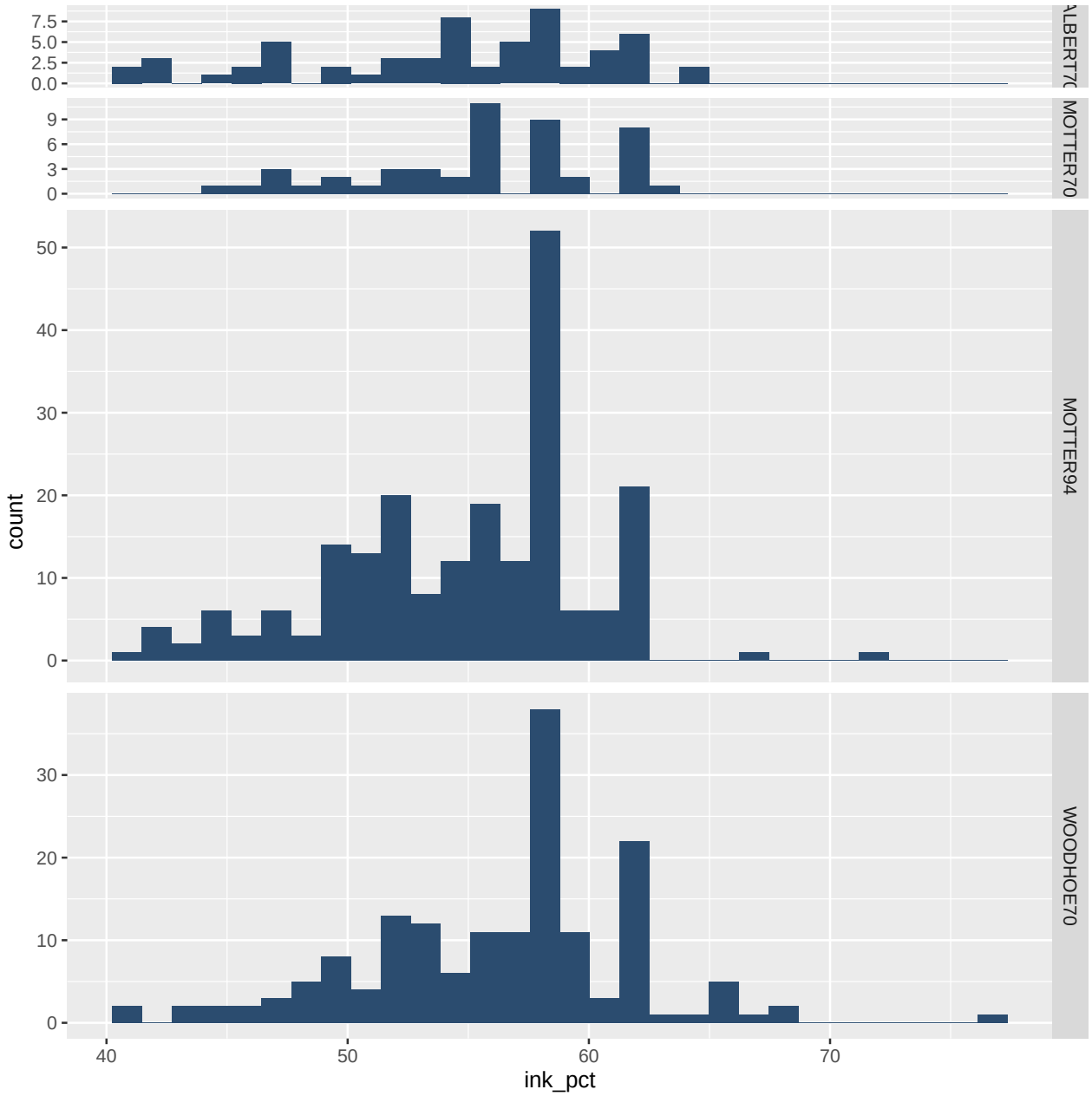


As you can see from the previous examples, fixed scales make it easier to see pattern across panels and free scales make it easier to see patterns within panels.

11.5 Controlling the space of each panel

The reserved space for each panel (width) is fixed by default but it can vary according to the range of scale for that column (or row), setting `space` argument of `facet_grid()` to `"free"`:

```
pl +
  facet_grid(press_type ~ ., space  = "free", scales = "free")
```



Remember that also `scales` argument must be set to `"free"`.

In this way, the scales are equal across the plot: 1 cm on each panel maps to the same range of data.

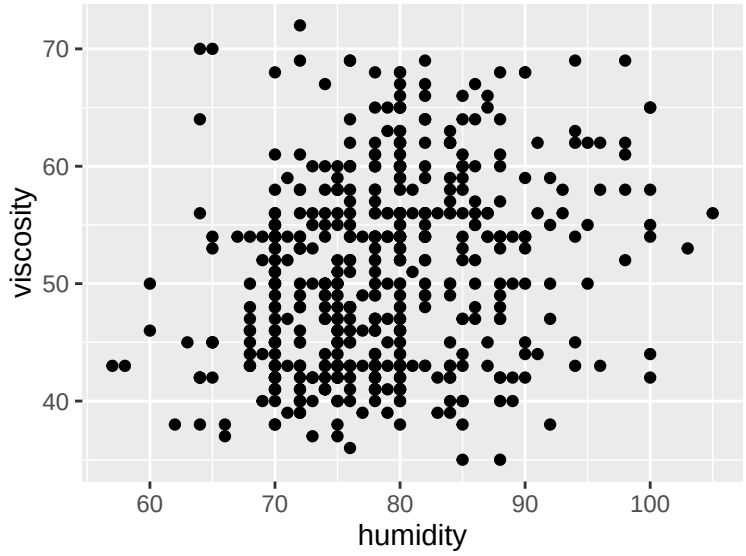
11.6 Faceting with continuous variables

Until now, we have seen how faceting categorical variables, but it is possible to facet also continuous variables, once discretised.
`ggplot2` provides three helper functions to discretise continuous variables:

- `cut_interval(x,n)`, which divides the data into `n` bins of the same length (equal range)
- `cut_width(x, width)`, which divides the data into bins of width `width`
- `cut_number(x, n)`, which divides data into `n` bins each containing (approximately) the same number of observations

Let us consider the relationship between humidity and viscosity in `bands` dataset:

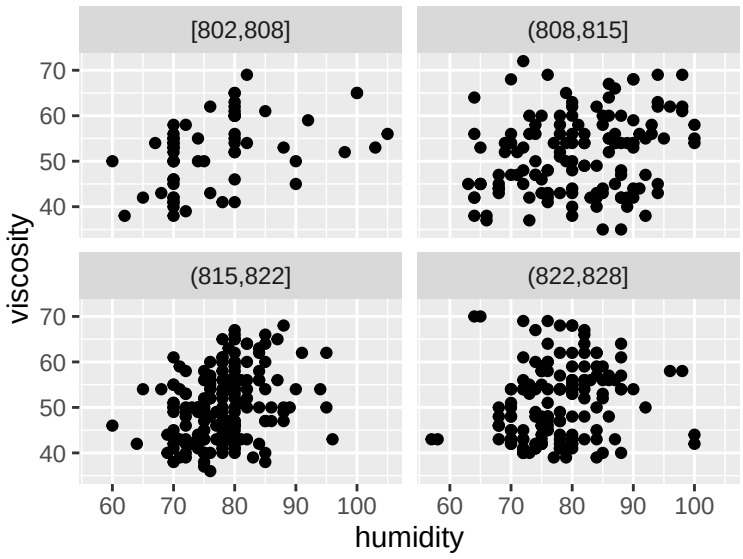
```
# Make a copy of the original data
bands2 <- bands
pl1 <- ggplot(data = bands2, mapping = aes(x = humidity, y = viscosity)) +
  geom_point()
pl1
```



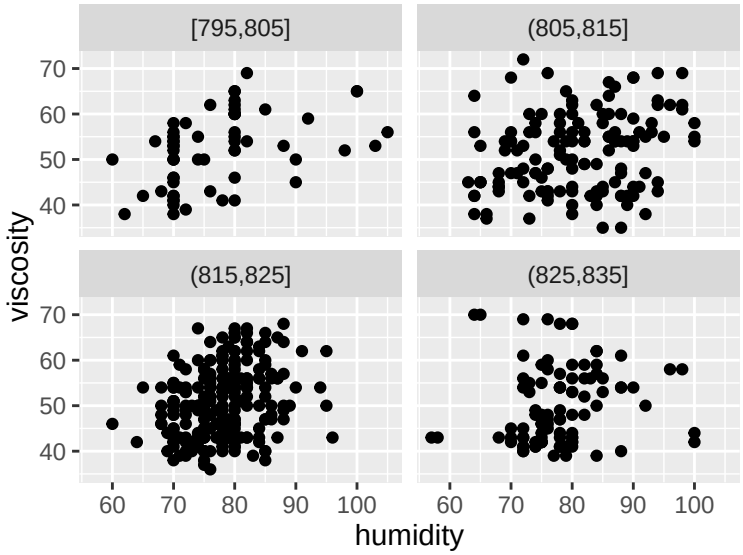
We want to verify the influence of pressure in this relationship. Firstly let us discretize `press`:

```
bands2 <- bands2 %>% mutate(
  # 4 interval with "equal" range
  press_i = cut_interval(x = press, n = 4),
  # intervals of width 10
  press_w = cut_width(x = press, width = 10),
  # 5 intervals with the "same" number of points
  press_n = cut_number(x = press, n = 5)
)
```

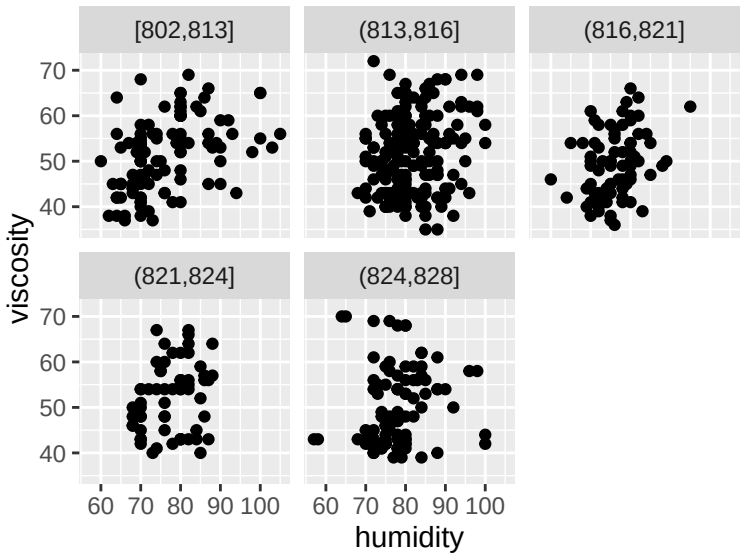
```
# 4 interval with "equal" range
ggplot(data = bands2, mapping = aes(x = humidity, y = viscosity)) +
  geom_point() +
  facet_wrap(~ press_i)
```




```
# intervals of width 10
ggplot(data = bands2, mapping = aes(x = humidity, y = viscosity)) +
  geom_point() +
  facet_wrap(~ press_w)
```




```
# 5 intervals with the "same" number of points
ggplot(data = bands2, mapping = aes(x = humidity, y = viscosity)) +
  geom_point() +
  facet_wrap(~ press_n)
```

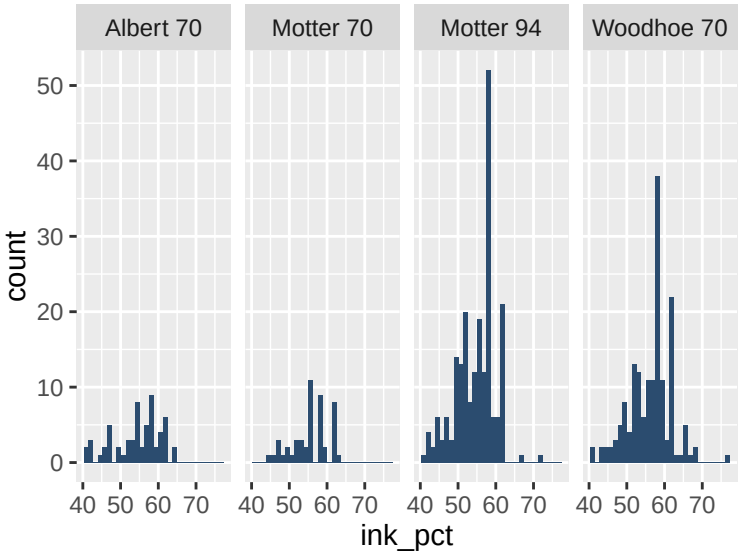


11.7 Changing the text of facets labels

Unlike with scales where you can set the labels, to set facet labels you must change the data values.

```
# Make a copy of the original data
bands3 <- bands

# Rename "BAND" to "Band", "NOBAND" to "No Band"
levels(bands3$press_type)[levels(bands3$press_type)=="ALBERT70"] <- "Albert 70"
levels(bands3$press_type)[levels(bands3$press_type)=="MOTTER70"] <- "Motter 70"
levels(bands3$press_type)[levels(bands3$press_type)=="MOTTER94"] <- "Motter 94"
levels(bands3$press_type)[levels(bands3$press_type)=="WOODHOE70"] <- "Woodhoe 70"
```

```
ggplot(data=bands3, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F") +
  facet_grid(. ~ press_type)
```



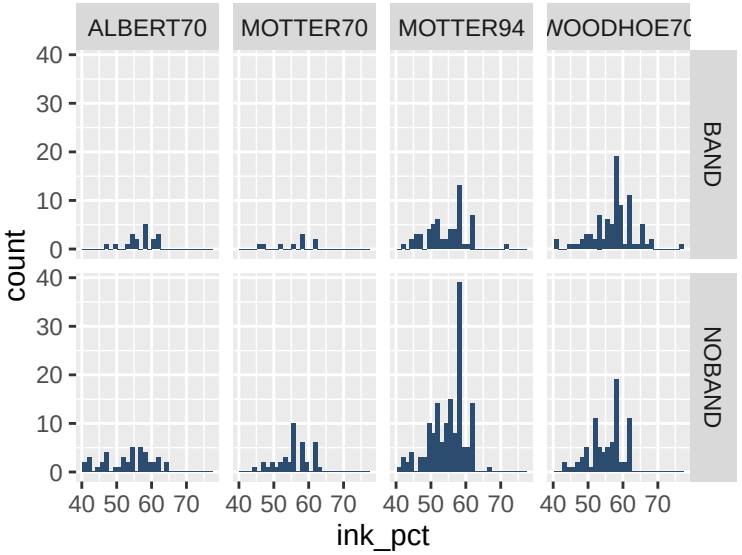
The appearance of the labels can be controlled also setting a function to `labeller` argument of `facet_grid()`

or `facet_wrap()`.

The most important labeller functions are:

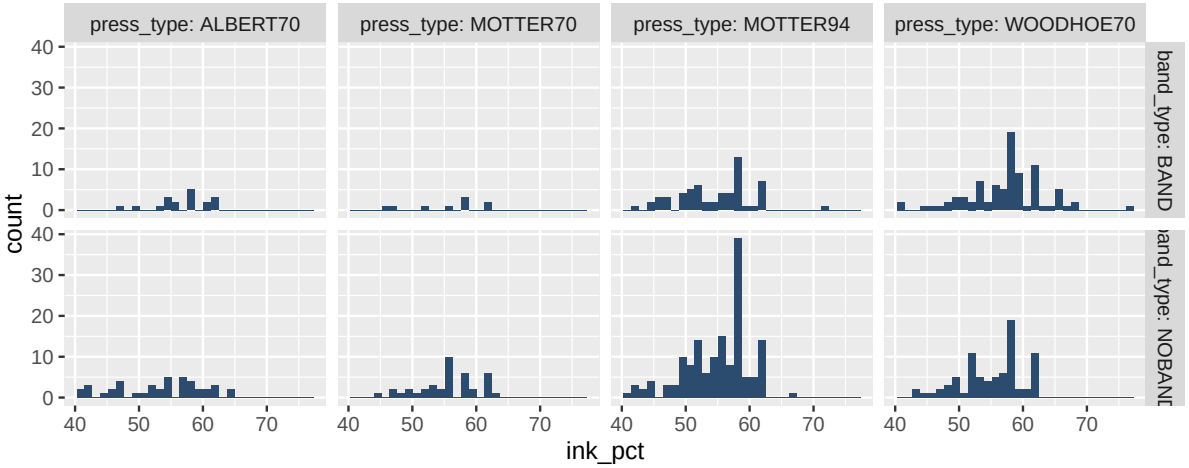
- `label_value`, the default

```
pl +
  facet_grid(band_type ~ press_type, labeller = label_value)
```



- `label_both`, which prints out both the name of the variable and the value of the variable in each facet

```
pl +
  facet_grid(band_type ~ press_type, labeller = label_both)
```

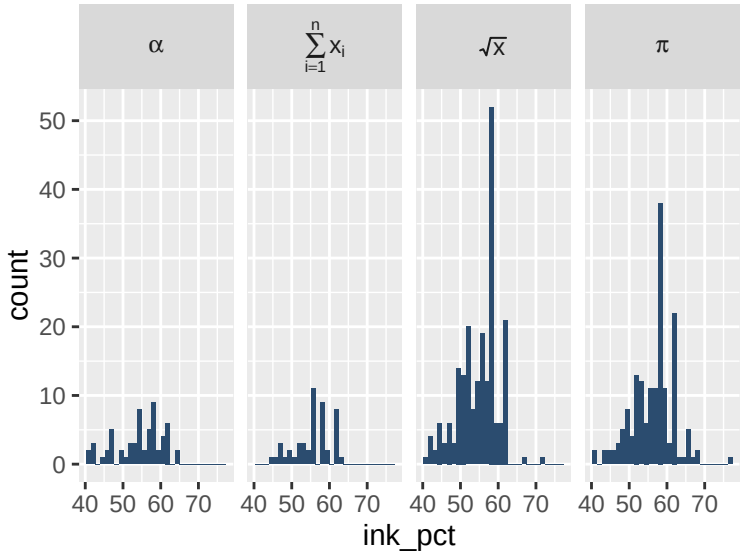


Now instead of the facet labels at the top being `"ALBERT70"`, `"MOTTER70"`, `"MOTTER94"` and `"WOODHOE70"`they are `"press_type: ALBERT70"`, `"press_type:  MOTTER70"`, `"press_type:  MOTTER94"` and `"press_type: WOODHOE70"`. Similarly, the vertical labels are `"band_type: BAND"`, `"band_type:  NOBAND"`. There is not a way to set the labels for the horizontal and vertical separately.

- `label_parsed`, which takes strings and treats them as R math expressions

```
# modify the dataset
bands4 <- bands
levels(bands4$press_type)[levels(bands4$press_type)=="ALBERT70"] <- "alpha"
levels(bands4$press_type)[levels(bands4$press_type)=="MOTTER70"] <- "sum(x[i], i==1, n)"
```

```
levels(bands4$press_type)[levels(bands4$press_type)=="MOTTER94"] <- "sqrt(x)"
levels(bands4$press_type)[levels(bands4$press_type)=="WOODHOE70"] <- "pi"

ggplot(data=bands4, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F")  +
  facet_grid(. ~ press_type, labeller = label_parsed)
```



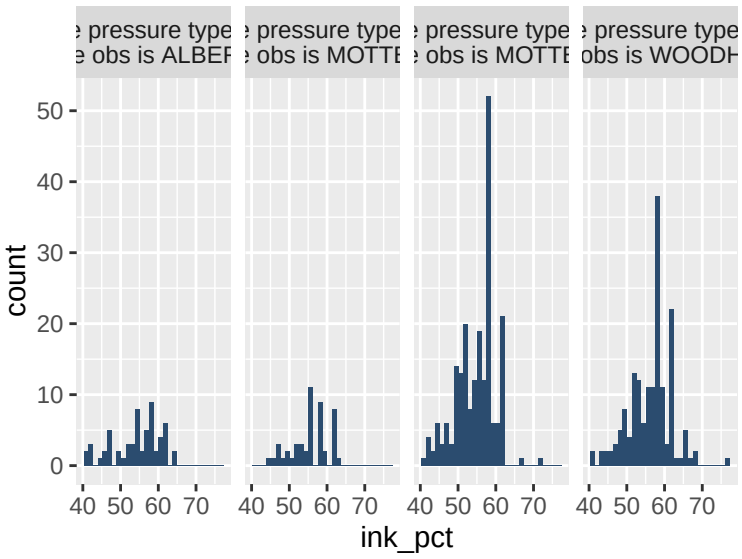
It is also possible to write labeller functions and pass them to `labeller` argument.

The following is an example of a labeller which wraps long labels at a fixed character count so that they take up less space. This can be useful since the space for facets is not expanded to guarantee that the entire label is visible.

```
label_wrap <- function(variable, value) {
  lapply(strwrap(as.character(value), width=25, simplify=FALSE),
         paste, collapse="\n")
}
```

The structure of a labeller is a function that takes two arguments: `variable` and `value`. `variable` is a length 1 character vector with the name of the variable which is being faceted on. `value` is the ordered (not necessarily unique) values of the facets.

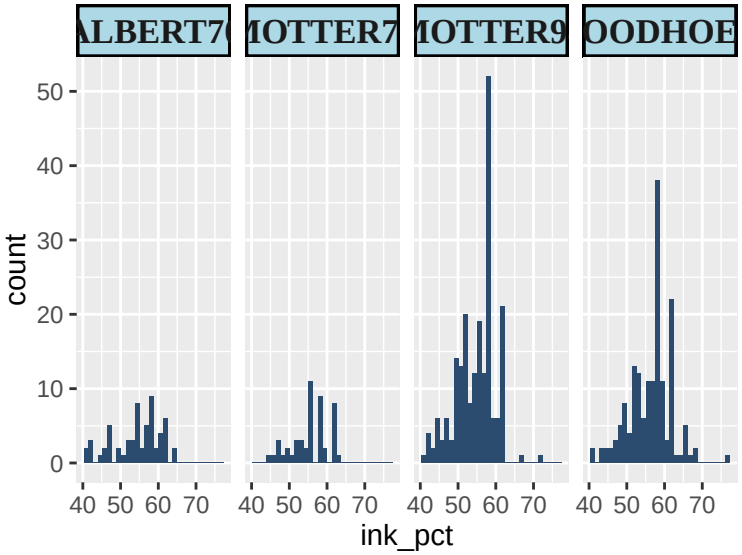
```
# modify the dataset
bands5 <- bands
levels(bands5$press_type)[levels(bands5$press_type)=="ALBERT70"] <- "The pressure type for these obs is
levels(bands5$press_type)[levels(bands5$press_type)=="MOTTER70"] <- "The pressure type for these obs is
levels(bands5$press_type)[levels(bands5$press_type)=="MOTTER94"] <- "The pressure type for these obs is
levels(bands5$press_type)[levels(bands5$press_type)=="WOODHOE70"] <- "The pressure type for these obs is

ggplot(data=bands5, mapping=aes(x=ink_pct)) +
  geom_histogram(fill="#2B4C6F") +
  facet_grid(. ~ press_type, labeller = label_wrap)
```



11.8 Changing the appearance of facet labels and headers

If you want to change the appearance of facet labels (like size, style and the font family), and headers (like colour, border colour and size), you have to use the theming system, setting `strip.text` argument, which control the text appearance, equal to `element_text()` function and `strip.background` argument, which control the background appearance, equal to `element_rect()` function:

```
pl +
  facet_grid(. ~ press_type) +
  theme(strip.text = element_text(face="bold",family = "Times", size=rel(1.2)),
        strip.background = element_rect(fill="lightblue", colour="black", size=1))
```

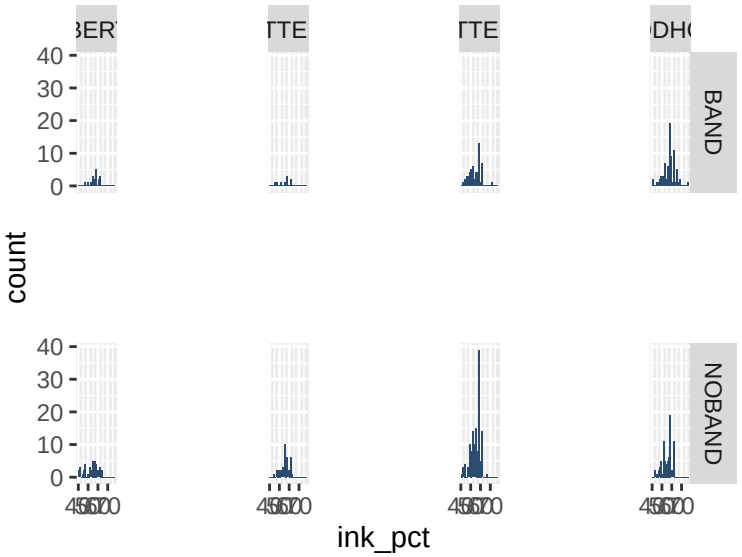


Using `rel(1.2)` makes the label text 1.2 times the size of the base text size for the theme and using `size=1` for the background makes the outline of the facets 1 mm thick.

11.9 Modify the margin between panels

If you want to modify the margin between panels you have to set `panel.margin` argument of `theme()` function by using `unit()` function in this way:

```
pl +
  facet_grid(band_type ~ press_type) +
  theme(panel.margin = unit(2, "cm"))
```



`unit()` is a function that creates a unit object of the correct length to use for setting margins. You have to specify the margin length and its units (of measure).

It works both for `facet_grid()` and `facet_wrap()`.

12 More on Plot Overall Appearanc

```
require(ggplot2)
require(ggthemes)
require(xtable)
require(qdata)
data(bands)
```

The overall appearance of plots is controlled by the theming system, an important component of `ggplot2` grammar. We started the exploration of theming system in the previous chapters, in particular in *Legend Customization*, *Axes Customization* and *Facet Customization*, where we saw how themes give us the control of non-data elements of the plot like fonts, ticks, panel strip, legend keys, ..

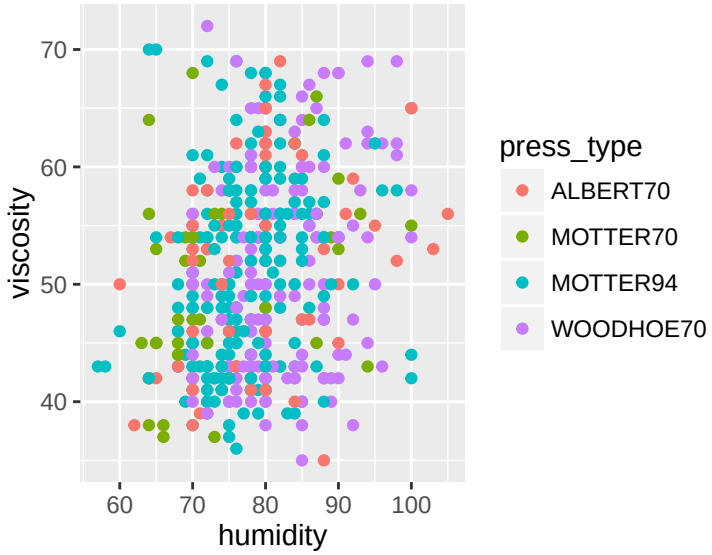
In this chapter we will deeepen the theming system structure and we will answer to the most frequently asked questions about plot customization.

12.1 Theming system structure

`ggplot2` has a default theme, called `theme_grey()`, with a light grey background and white gridlines.
A theme, or **theme function** defines the settings of a collection of theme elements for the purpose of creating a specific style of graphics production.

Let us see an example, considering the relationship between humidity and viscosity by pressure type in `bands` dataset:

```
pl <- ggplot(data = bands, mapping = aes(x= humidity, y = viscosity, colour =press_type)) +
  geom_point()
pl
```



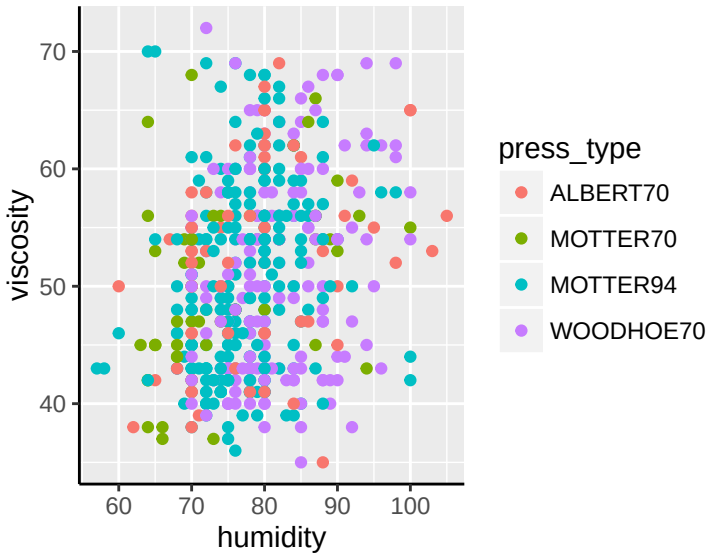
In particular, a **theme function** is composed of:

- **theme elements**, which refer to individual attributes of a graphic that are independent of the data and that you can control, such as font size, axis ticks, appearance of grid lines or background color of a legend;
- **theme element functions**, which enables you to modify the settings of certain theme elements. In particular, each theme element is associated with an element function, which describes the visual properties of that element.

Each theme element has a default value that can be locally modified in a specific ggplot object by using **`theme()` function**.

Suppose we want to modify the colour of axis lines:

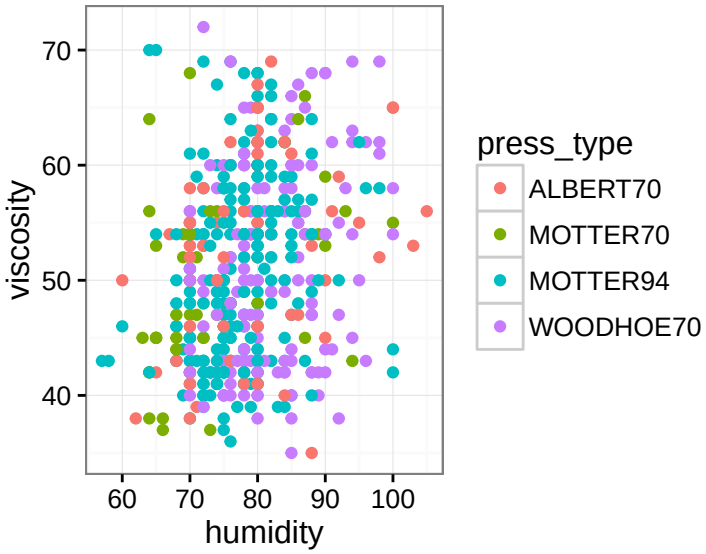
```
pl +
  theme(axis.line.x = element_line(colour = "black"),
        axis.line.y = element_line(colour = "black"))
```



In particular, `axis.line.x` and `axis.line.y` are the theme elements that control the appearance of x and y axis respectively and `element_line()` is the theme element function that allows us to modify the theme elements.

Moreover, if you don' like `theme_grey()` you can decide to totally replace it by setting another **theme function** (complete theme). Let us set `theme_bw()`, a theme with white background and thin grey grid lines:

```
pl +
  theme_bw()
```



Understanding how dealing with **theme element functions** and **theme elements** is very important in plot customization phase.

The most important **theme element functions** are:

- `element_text()`: controls the drawing of labels and headings.
- `element_line()`: draws lines and segments such as graphics region boundaries, axis tick marks and grid lines.
- `element_rect()`: draws rectangles. It is mostly used for background elements and legend keys.

- `element_blank()`: draws nothing. This function has no arguments.

There are around 40 unique **theme elements** that controls the appearance of the plots. They can be roughly grouped into five categories: plot, axis, legend, panel and facet.

Let us schematize them:

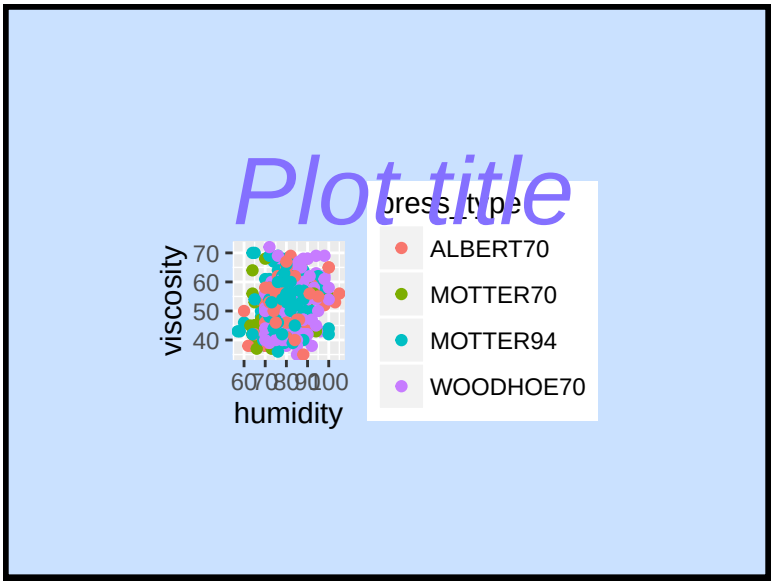
- plot theme elements

Theme Element	Description	Element Function Associated
`plot.background`	plot background	`element_rect()`
`plot.title`	plot title	`element_text()`
`plot.margin`	margins around plot	`unit()`

`unit()` is a theme function exported by `ggplot2` from `grid` package. It controls grid.

Let us see an example:

```
pl +
  labs(title = "Plot title") +
  theme(plot.title = element_text(size = 36, hjust = 0, colour = "lightslateblue", face = "italic"),
        plot.background = element_rect(fill = "lightsteelblue1", colour = "black", size = 2, linetype =
        plot.margin = unit(c(2, 2, 2, 2), "cm"))
```



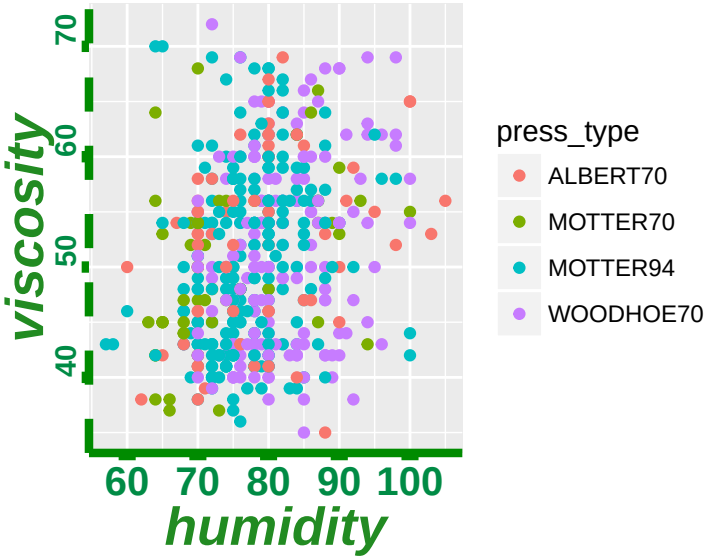
- axis theme elements

Theme Element	Description	Element Function Associated
`axis.line`	line parallel to axis (hidden in default theme)	`element_line()`
`axis.text`	tick labels	`element_text()`
`axis.text.x`	x-axis tick labels	`element_text()`
`axis.text.y`	y-axis tick labels	`element_text()`
`axis.title`	axis titles	`element_text()`
`axis.title.x`	x-axis title	`element_text()`
`axis.title.y`	y-axis title	`element_text()`
`axis.ticks`	axis tick marks	`element_line()`
`axis.ticks.length`	length of tick marks	`unit()`

Theme Element	Description	Element Function Associated
`axis.ticks.margin`	width of axis tick margin	`unit()`

Let us see an example:

```
pl +
  theme(
    axis.line.x = element_line(colour = "green4", size = 1.5),
    axis.line.y = element_line(colour = "green4", linetype = "dashed", size = 1.5),
    axis.text = element_text(color = "springgreen4", size = 15, face = "bold"),
    axis.text.y = element_text(angle = 90, size = rel(0.7), hjust = 0),
    axis.ticks = element_line(colour = "green4", size = 2),
    axis.ticks.x = element_line(size = rel(1.5)),
    axis.title = element_text(size = 20, color = "forestgreen", face = "bold.italic")
)
```

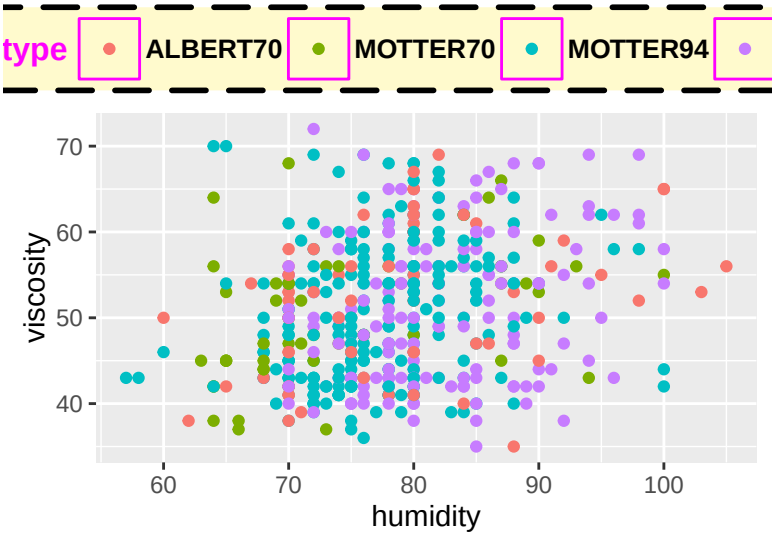


- legend theme elements

Theme Element	Description	Element Function Associated (or Value)
`legend.background`	legend background	`element_rect()`
`legend.key`	background of legend keys	`element_rect()`
`legend.key.size`	legend key size	`unit()`
`legend.key.height`	legend key height	`unit()`
`legend.key.width`	legend key width	`unit()`
`legend.margin`	legend margin	`unit()`
`legend.text`	legend labels	`element_text()`
`legend.text.align`	legend label alignment	numeric value
`legend.title`	legend name	`element_text()`
`legend.title.align`	legend name alignment	numeric value
`legend.position`	position of legend	“left”, “right”, “bottom”, “top”
`legend.direction`	direction of legend keys	“horizontal” or “vertical”
`legend.justification`	justification of legend	numeric value
`legend.box`	position of multiple legend boxes	“horizontal” or “vertical”

Let us see an example:

```
pl +
  theme(
    legend.position = "top",
    legend.box = "horizontal",
    legend.background = element_rect(fill = "lemonchiffon", color = "black", size = 1, linetype = "longe
    legend.key = element_rect(fill = "lemonchiffon", color = "magenta"),
    legend.key.width = unit(0.8, "cm"),
    legend.key.height = unit(0.8, "cm"),
    legend.text = element_text(face = "bold", size = 10),
    legend.title = element_text(face = "bold", size = 12, colour = "magenta")
    )
```

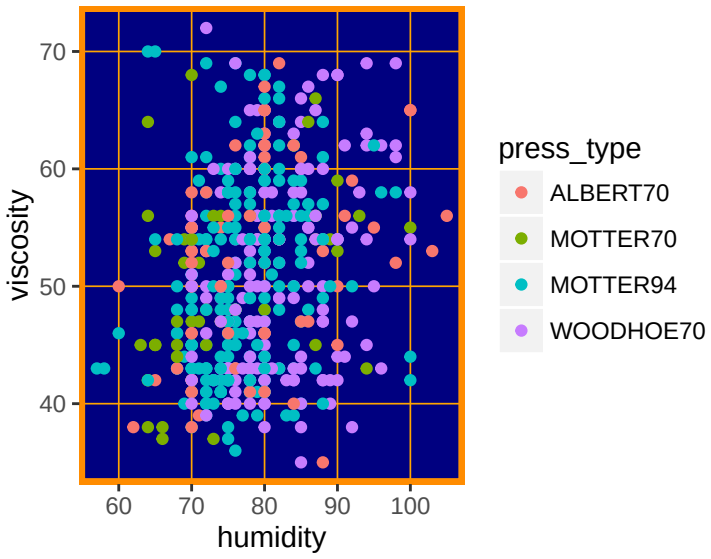


- panel theme elements

Theme Element	Description	Element Function Associated (or Value)
`panel.background`	background of graphics region	`element_rect()`
`panel.border`	border of graphics region	`element_rect()`
`panel.grid.major`	major grid lines	`element_line()`
`panel.grid.major.x`	vertical major grid lines height	`element_line()`
`panel.grid.major.y`	horizontal major grid lines	`element_line()`
`panel.grid.minor`	minor grid lines	`element_line()`
`panel.grid.minor.x`	vertical minor grid lines	`element_text()`
`panel.grid.minor.y`	horizontal minor grid lines	`element_line()`
`panel.margin`	margin between facets	numeric value
`aspect.ratio`	plot aspect ratio	numeric value

Let us see an example:

```
pl +
  theme(
    panel.background = element_rect(fill = "navy", color = "orange", size = 2),
    panel.border = element_rect(fill = NA, colour = "darkorange", size = 2),
    panel.grid.major = element_line(color = "orange", size = 0.3),
    panel.grid.minor = element_blank()
    )
```

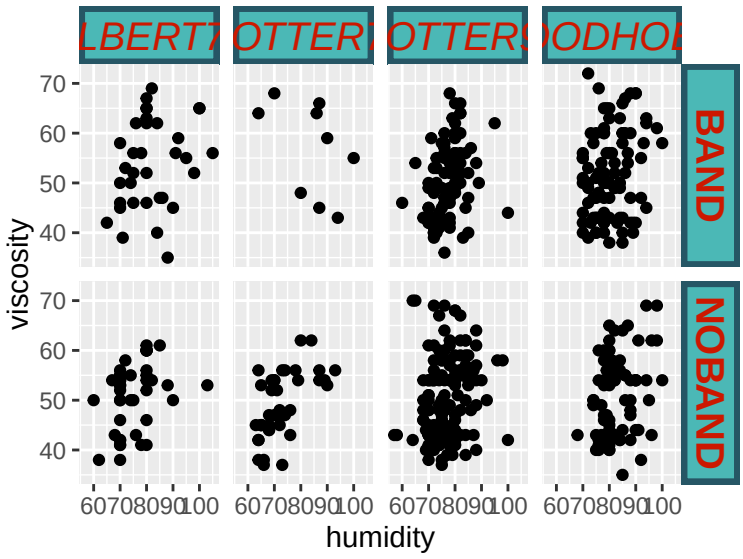


- facet theme elements

Theme Element	Description	Element Function Associated
`strip.background`	background of panel strips	`element_rect()`
`strip.text`	strip text	`element_text()`
`strip.text.x`	horizontal strip text	`element_text()`
`strip.text.y`	vertical strip text	`element_text()`

Let us see an example:

```
ggplot(data = bands, mapping = aes(x= humidity, y = viscosity)) +
  geom_point() +
  facet_grid(band_type ~ press_type) +
  theme(
    strip.background = element_rect(fill = "#4bb8b6", color = "#265665", size = 2),
    strip.text = element_text(face = "italic", size = 15, colour = "#CC1800"),
    strip.text.y = element_text(face = "bold")
  )
```



We have already learn about most of these `ggplot2` theme elements in the previous chapters, in particular about: axis, legend and facet categories. But what's about plot and panels theme elements?
With the term "plot" we means what is included outside the plotting area and with "panel" what is included in the plotting area.

In the following paragraphs we will see how to handle with the most common questions about plot and panel customization. We will talk also about the customization of the whole theme function.

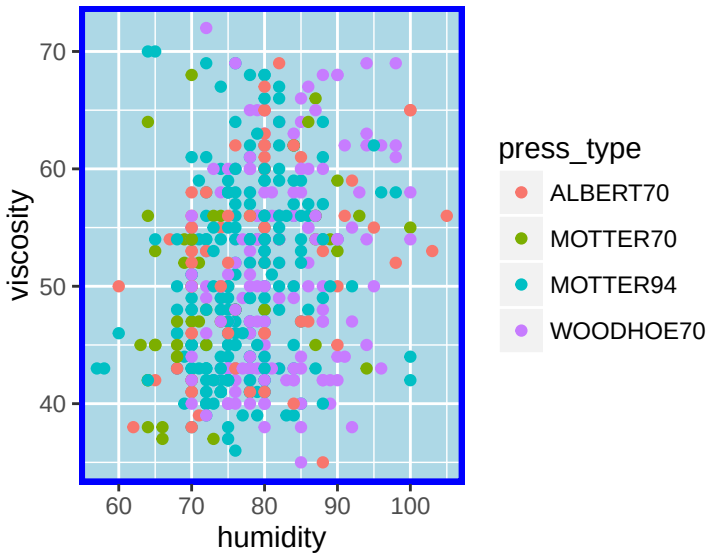
12.2 Customize the appearance of plotting area

If you want to change the appearance of plotting area you have to set `panel_xxx` arguments of `theme()` function.

12.2.1 Change the background of plotting area

`panel.background` element allows us to modify the background of graphical region and `panel.border` to modify the border of graphical region. Both `panel.background` and `panel.border` are modified by using `element_rect()` function.

```
pl +
  theme(
    panel.background = element_rect(fill="lightblue"),
    panel.border = element_rect(colour="blue", fill=NA, size=2)
)
```

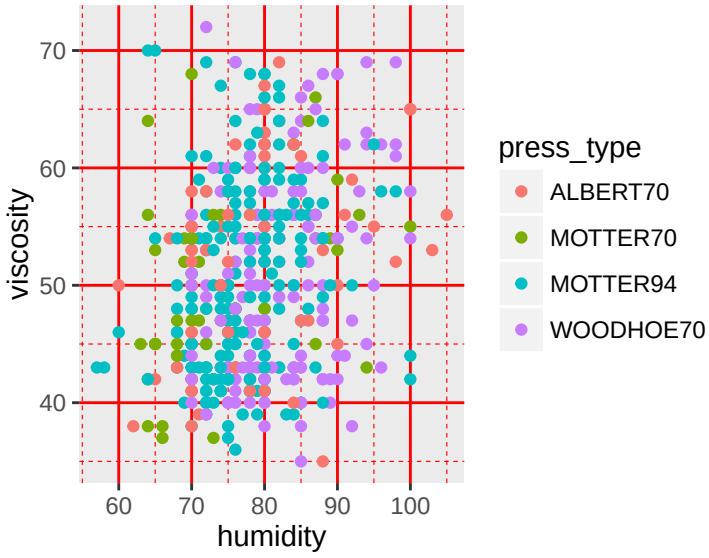


Pay attention to `fill` argument for `panel.border`: you have to specify a blank fill (setting it to `NA`) for not covering panels.

12.2.2 Customize and remove grid lines

`panel.grid.major` and `panel.grid.minor` arguments allows us to customize grid lines by using `element_line()` function:

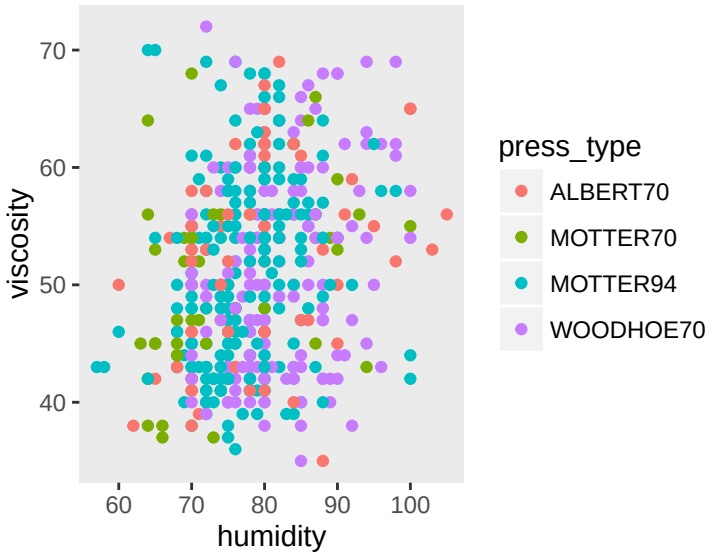
```
pl +
  theme(
    panel.grid.major = element_line(colour="red"),
    panel.grid.minor = element_line(colour="red", linetype="dashed", size=0.2)
)
```



If you want to remove grid lines you have to set `panel.grid.major` and/or `panel.grid.minor` equal to `element_blank()`:

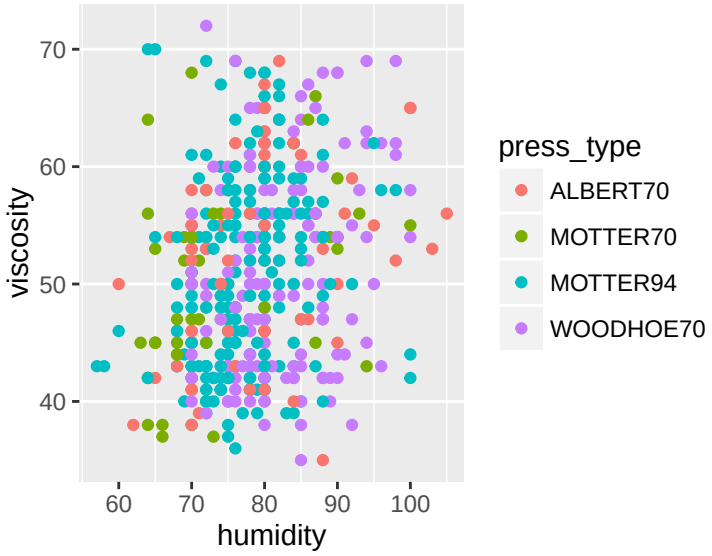
```
pl +
  theme(
```

```
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank()
)
```



It's possible to hide just the vertical or horizontal grid lines with `panel.grid.major.x`, `panel.grid.major.y`, `panel.grid.minor.x` and `panel.grid.minor.y` arguments of `theme()` function:

```
pl +
  theme(
    panel.grid.major.x = element_blank(), # remove horizontal grid major lines
    panel.grid.minor.y = element_blank()  # remove vertical grid minor lines
)
```



12.3 Customize the plot appearance outside plotting area

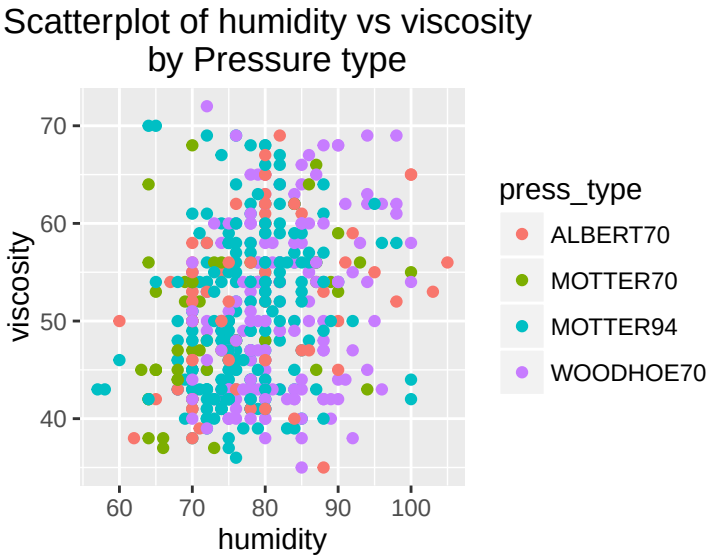
If you want to modify plot appearance outside plotting area you have to set `plot_xxx` arguments of `theme()` function.

12.3.1 Add the title to a plot and change its appearance

As we saw in *Creating a Boxplot* chapter, title can be added by using `ggtitle()` or `labs` function in this way:

```
pl +
  ggtitle("Scatterplot of humidity vs viscosity \n by Pressure type")
pl +
  labs(title = "Scatterplot of humidity vs viscosity \n by Pressure type")
```

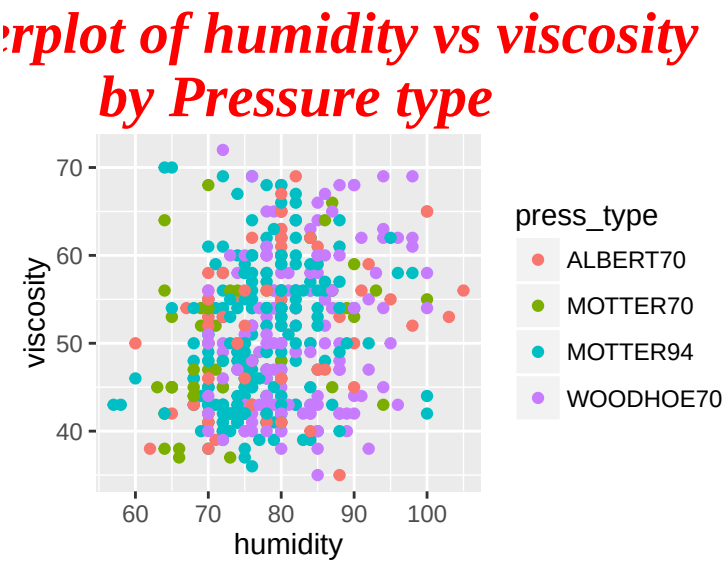
The previous two command lines produce the same result:



`"\n"` is used to break the lines.

Title appearance can be changed by setting `plot.title` argument of `theme()` function:

```
pl +
  ggtitle("Scatterplot of humidity vs viscosity \n by Pressure type") +
  theme(
    plot.title=element_text(size=rel(2), lineheight=0.9, family="Times", face="bold.italic", colour="red
        )
```



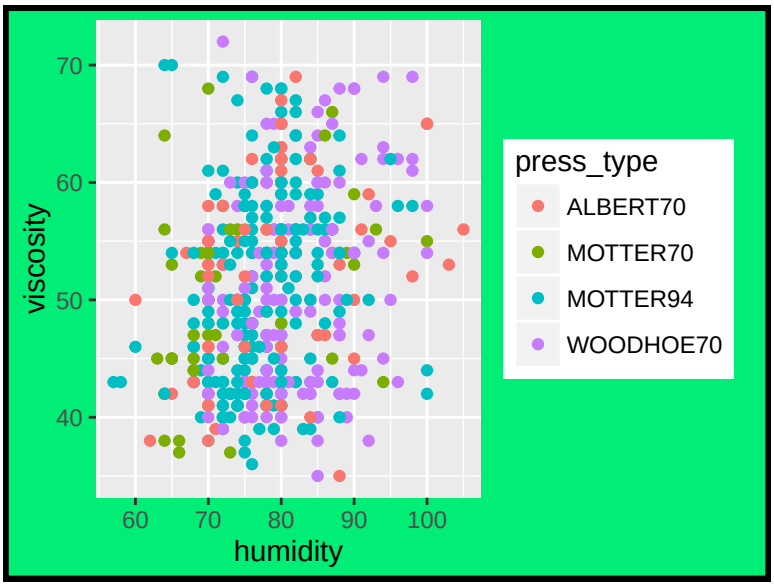
`lineheight` arguments refers to interlinear space in plot title.

12.3.2 Change the background outside plotting area

To modify the background set `plot.background` element of `theme()` function.

You can change the colour of the background and add a contour line.

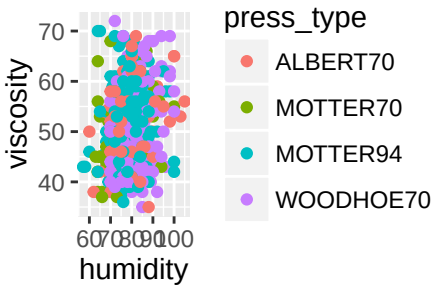
```
pl +
  theme(
    plot.background = element_rect(fill = "springgreen2",linetype = "solid", colour = "black", size = 2
)
```



12.3.3 Change the margins around graphical area

To modify margins around graphical area set `plot.margin` element of `theme()` function:

```
pl +
  theme(
    plot.margin = unit(x = c(2, 2, 2, 2), units = "cm")
)
```



`unit()` is a function that creates a grid unit object of the correct length to use for setting margins. `x` argument have to be specified as a vector of lenght 4 containing the measure of margins for the four margin of the plotting area, and `units` argument represents the measurements units. If you want more details about the supported measurements units have a look at the help of `unit()` function (`?unit`)

12.4 Customize theme function

12.4.1 Change the default theme

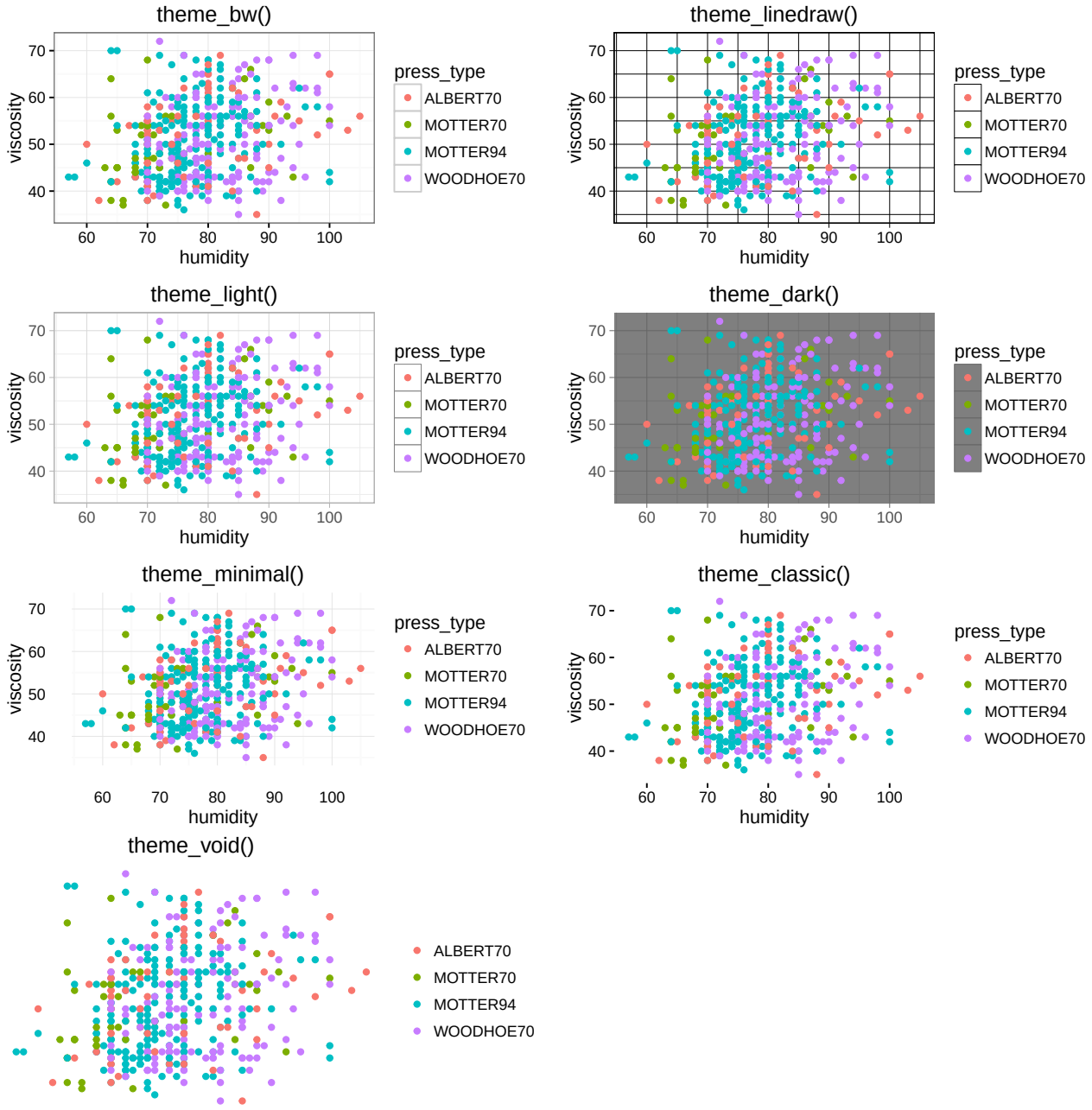
If you want to change theme function you have to overwrite the defaut one. Other theme functions are provided by `ggplot2` and `ggthemes` packages.

`ggplot2` provides the following theme functions:

- `theme_bw()`: a variation on `theme_grey()` that uses a white background and thin grey grid lines
- `theme_linedraw()`: a theme with only black lines of various widths on white backgrounds, reminiscent of a line drawing

- `theme_light()`: similar to `theme_linedraw()` but with ligth grey lines and axes, to direct more attention towards the data
- `theme_dark()`: the dark cousin of `theme_light()`, with similar line sizes but a dark background. Useful to make thin coloured lines pop out
- `theme_minimal()`: a minimalistic theme with no background annotations
- `theme_classic()`: A classic-looking theme, with x and y axis lines and no gridlines. In `ggplot` 2.1.0 axes are not visible because of a bug
- `theme_void()`: a completely empty theme

```
pl1 <- pl + theme_bw() + ggtitle("theme_bw()")
pl2 <- pl + theme_linedraw() + ggtitle("theme_linedraw()")
pl3 <- pl + theme_light() + ggtitle("theme_light()")
pl4 <- pl + theme_dark() + ggtitle("theme_dark()")
pl5 <- pl + theme_minimal() + ggtitle("theme_minimal()")
pl6 <- pl + theme_classic() + ggtitle("theme_classic()")
pl7 <- pl + theme_void() + ggtitle("theme_void()")

gridExtra::grid.arrange(pl1, pl2, pl3, pl4, pl5, pl6, pl7, ncol=2)
```

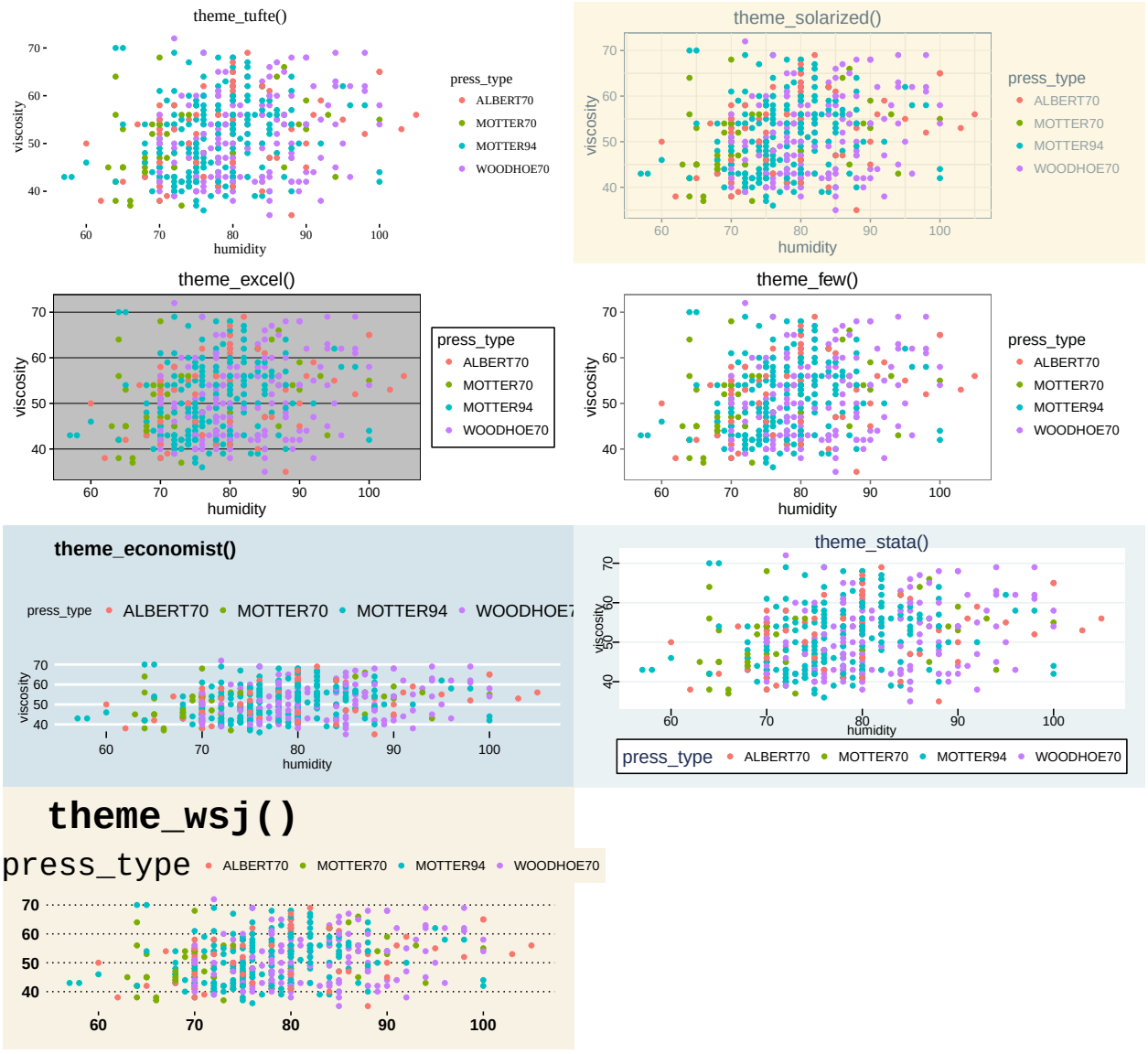


`ggthemes` package provides lots of theme functions. Some of them are listed here:

- `theme_tufte()`: a minimal ink theme based on Tufte's The Visual Display of Quantitative Information
- `theme_solarized()`: a theme using the solarized color palette
- `theme_excel()`: a theme replicating the classic gray charts in Excel
- `theme_few()`: theme from Stephen Few's "Practical Rules for Using Color in Charts"
- `theme_economist()`: a theme based on the plots in the The Economist magazine
- `theme_stata()`: themes based on Stata graph schemes
- `theme_wsj()`: a theme based on the plots in the The Wall Street Journal

```
pl8 <- pl + theme_tufte() + ggtitle("theme_tufte()")
pl9 <- pl + theme_solarized() + ggtitle("theme_solarized()")
pl10 <- pl + theme_excel() + ggtitle("theme_excel()")
pl11 <- pl + theme_few() + ggtitle("theme_few()")
```

```
pl12 <- pl + theme_economist() + ggtitle("theme_economist()")
pl13 <- pl + theme_stata() + ggtitle("theme_stata()")
pl14 <- pl + theme_wsj() + ggtitle("theme_wsj()")

gridExtra::grid.arrange(pl8, pl9, pl10, pl11, pl12, pl13, pl14, ncol=2)
```



12.4.2 Chang the default theme for more than one plot

If you want to change the default theme (`theme_grey()`) for all plots generated in the current R session, you can use `theme_set()` function. For example, if you want to use white background for all plots run:

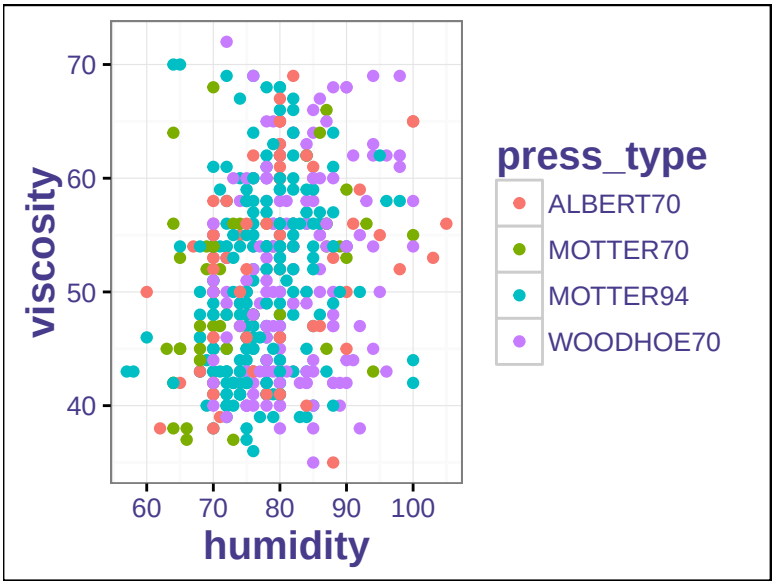
```
theme_set(theme_bw())
```

12.4.3 Creating your own theme

You can create your own theme by adding elements to an existing theme:

```
mytheme <- theme_bw() +
theme(text = element_text(colour="slateblue4"),
      axis.title = element_text(size = rel(1.25), face = "bold"),
      legend.title = element_text(size = rel(1.25), face = "bold"),
      plot.background = element_rect(colour ="black")
      )
```

```
pl +
  mytheme
```



13 More on Stats

```
require(ggplot2)
require(RColorBrewer)
require(dplyr)
require(qdata)
data(istat)
```

As we said in *Introduction to Graphics* chapter, layer building block contains two elements of `ggplot2` grammar: statistic (`stats`) and geometric (`geoms`).

`stats` can be seen as an alternative way to build a layer. A statistical transformation, or `stat`, transforms the data, typically by summarising it in some manner. Indeed, some plots, like Box Plot, Histogram, Bar Plot, etc., visualize a transformation of the original data set.

We can divide statistic layer (`stat`) into two groups:

- Statistics with geoms
- Statistics outside geoms

Both categories of functions begin with `stat_`.

In the examples of the following paragraph we will consider `istat` dataset, which provides the measures of weight, height, gender and geographical area ("Nord", "Centro", "Sud" and "Isole") from 1806 Italian people.

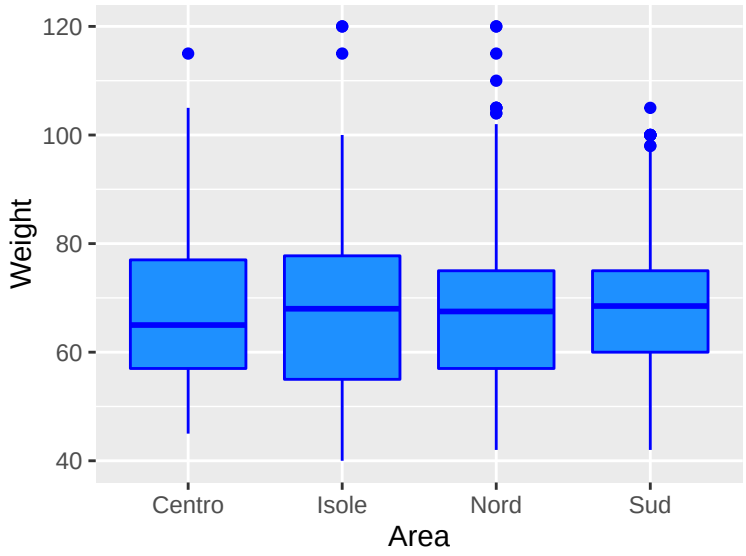
13.1 Statistics with geoms

Statistics with geoms group includes `stats` functions that are combined with `geoms` functions to make a layer. You have already used many of `ggplot2`'s `stats` because, usually, they are used behind the scenes to generate the most important `geoms`.

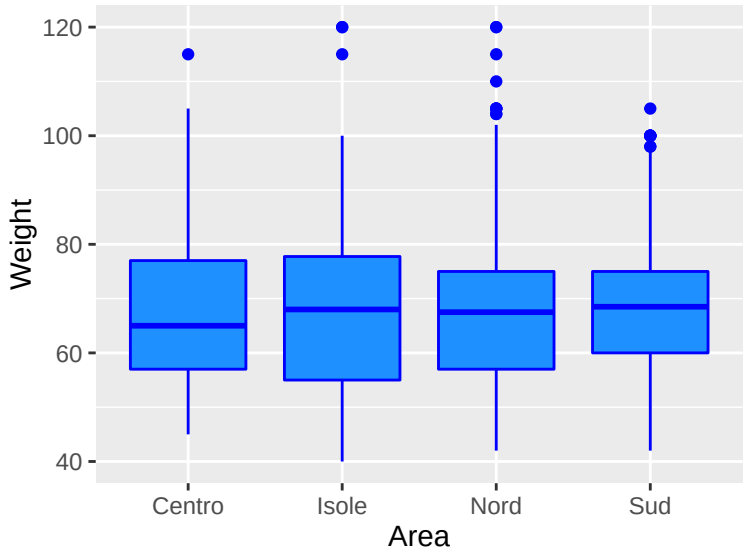
`stats` functions and `geoms` functions both combine a stat with a geom to make a layer.

For example, `"boxplot"` is the default statistical transformation allowed by `geom_boxplot()`, which summarises the observations in a given group. `geom_boxplot(stat="boxplot")` does the same as `stat_boxplot(geom="boxplot")`.

```
ggplot(data=istat, mapping=aes(y=Weight, x=Area)) +
  geom_boxplot(stat="boxplot", col="blue", fill ="dodger blue")
```



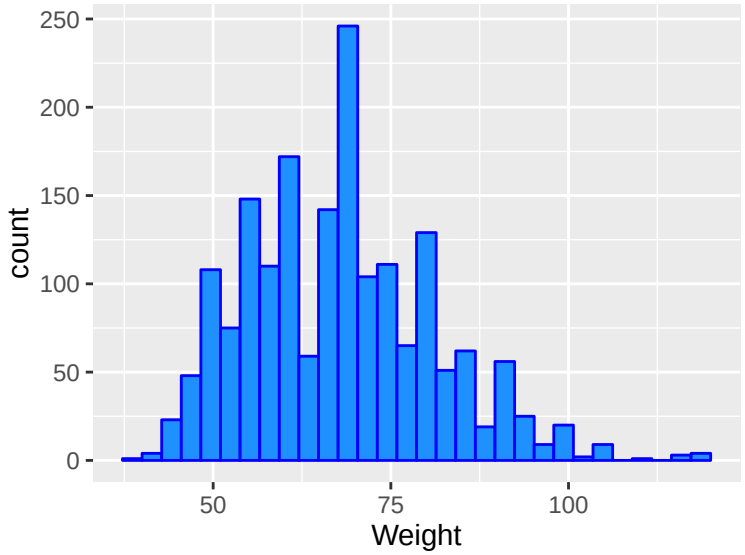

```
ggplot(data=istat, mapping=aes(y=Weight, x=Area)) +
  stat_boxplot(geom="boxplot", col="blue", fill ="dodger blue")
```



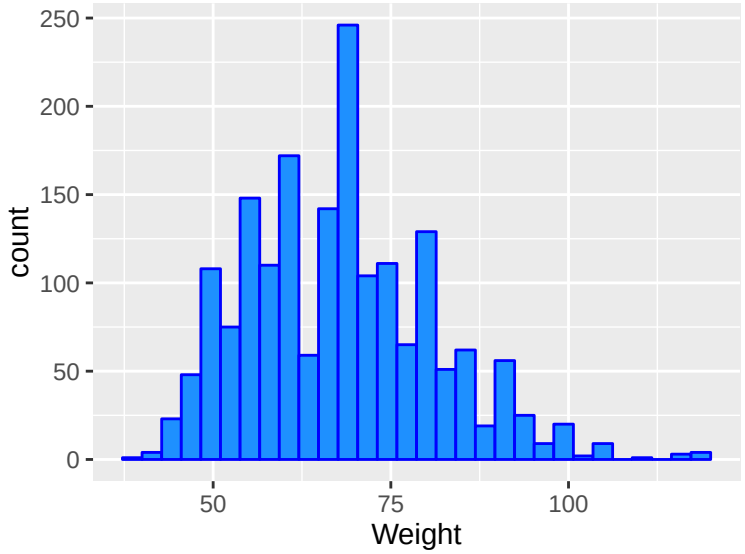
All `geoms` are based on a default statistical transformation. Some `geoms` allows us to use different `stats` in order to achieve different results.

For example, `"bin"` is a statistical transformation allowed by `geom_bar()`, which bins the data in ranges. It is not the default statistical trasformation, which is `"count"`. `geom_bar(stat="bin")` does the same as `stat_bin(geom="bar")`:

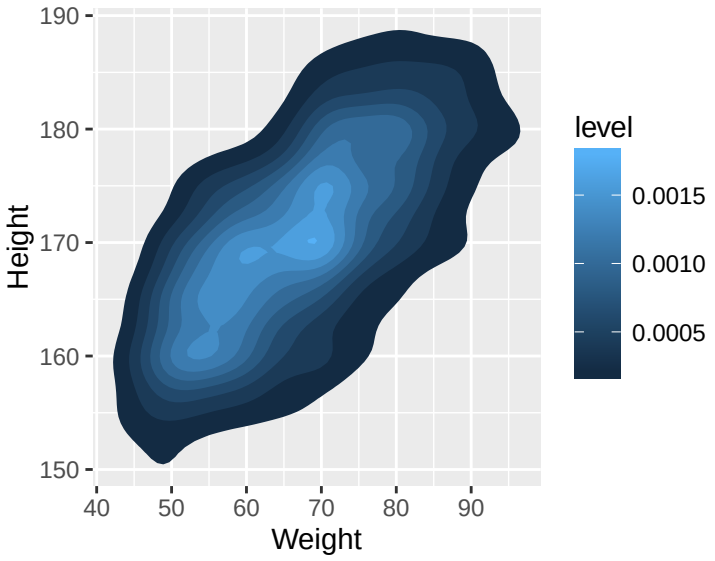
```
ggplot(data=istat, mapping=aes(x =Weight)) +
  geom_bar(stat="bin", col="blue", fill ="dodger blue")
```




```
ggplot(data=istat, mapping=aes(x=Weight)) +
  stat_bin(geom="bar", col="blue", fill ="dodger blue")
```




```
ggplot(istat, aes(Weight, Height)) +
        geom_polygon(aes(fill = ..level..), stat = "density2d", n = 100)
```

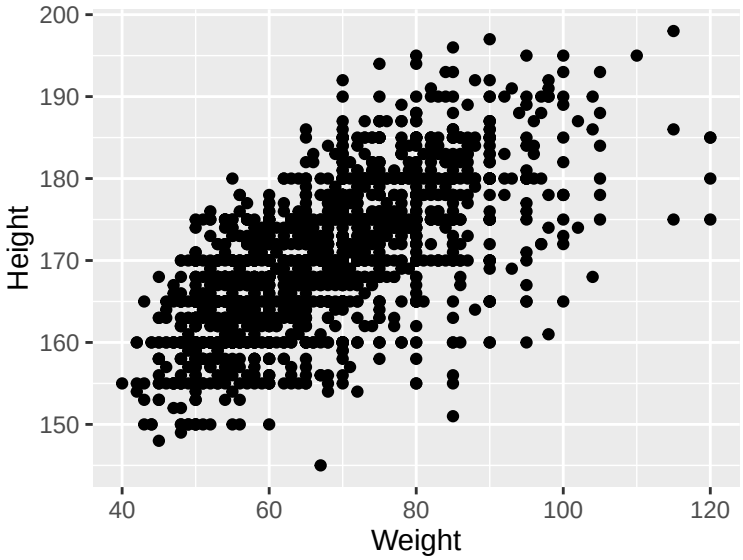


`..level..` variable, created by `stat_density2d()` function, draws a surface plot (a raster plot). –>

By default, most of the `geoms` seen until now uses `stat="identity"`, that do not transform data.

For example, `stat="identity"` is the default `stat` of `geom_point()`:

```
ggplot(data=istat, mapping=aes(x=Weight, y=Height)) +
  geom_point()
```



As you can see, points are not transformed, they are only plotted on the graph.

The following table schematizes the corrispondence between `geom` and the default `stat` function. The `geoms` for which `stat="identity"` is the default transformation are not included in table.

geom	stat
geom_histogram()	stat_bin()
geom_bar()	stat_bin()
geom_freqpoly()	stat_bin()
geom_smooth()	stat_smooth()
geom_boxplot()	stat_boxplot()

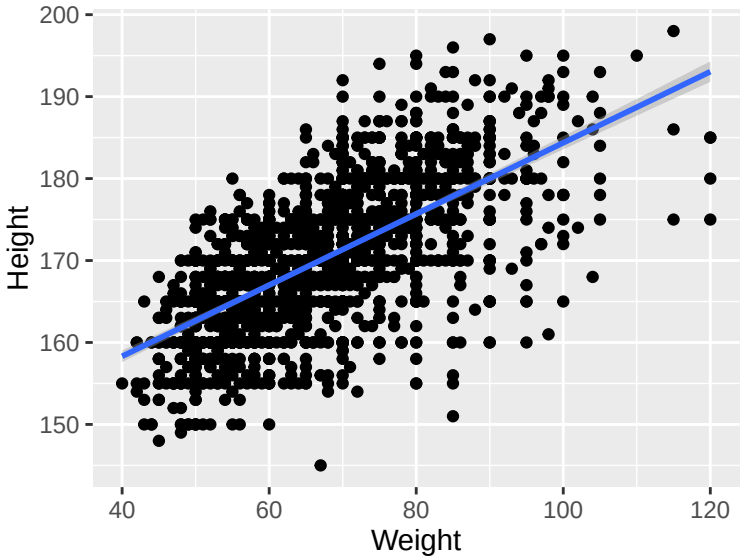
geom	stat
geom_dotplot()	stat_bindot()
geom_bin2d()	stat_bin2d()
geom_hex()	stat_binhex()
geom_contour()	stat_contour()
geom_quantile()	stat_quantile()
geom_count()	stat_sum()

In the following paragraph we will see the functionalities of some statistical transformation listed in the previous table.

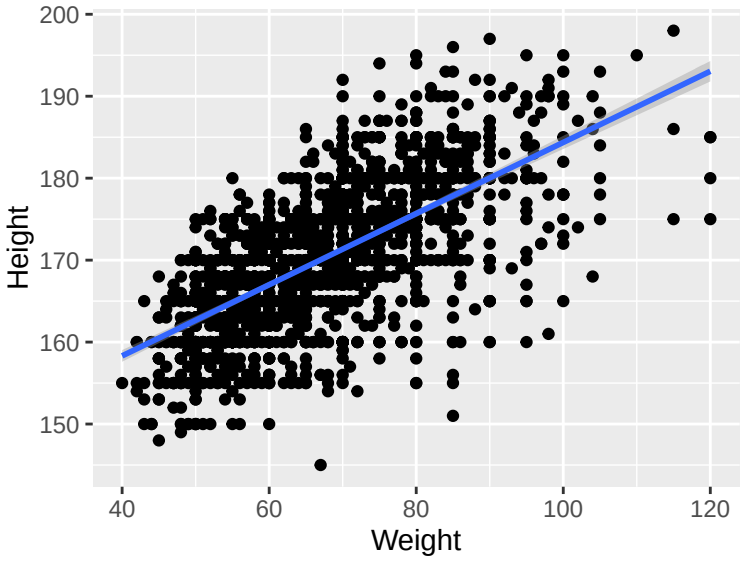
13.1.1 Adding Fitted Regression Model Lines

Suppose you want to fit a model to the data. The statistic that fits a model and represents a regression line is `"smooth"`.
Let us consider `istat` dataset, we want to plot a regression model line between `Weight` and `Height`.
This can be done in two ways:

```
ggplot(istat, aes(Weight, Height)) +
  geom_point() +
  geom_smooth(method=lm)
```



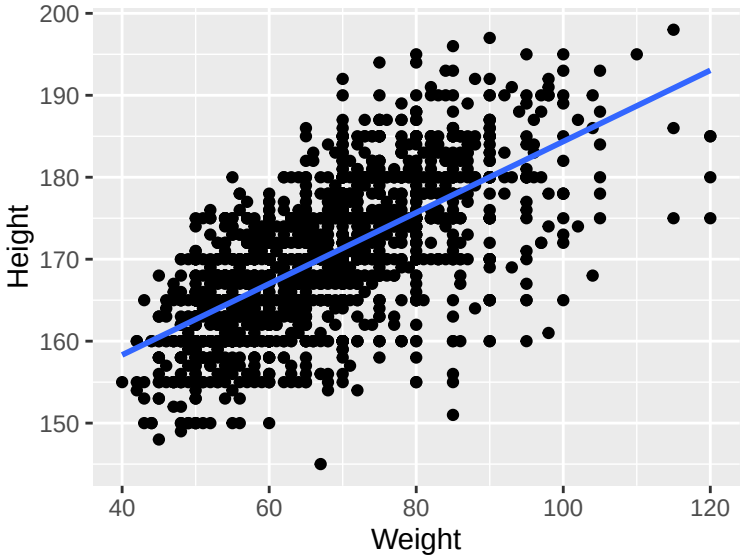

```
ggplot(istat, aes(Weight, Height)) +
  geom_point() +
  stat_smooth(method=lm)
```



As you can see the result is the same. In the following examples we will analyze `stat_smooth()`. Remember that what we say for `stat_smooth()` also applies to `geom_smooth()`.

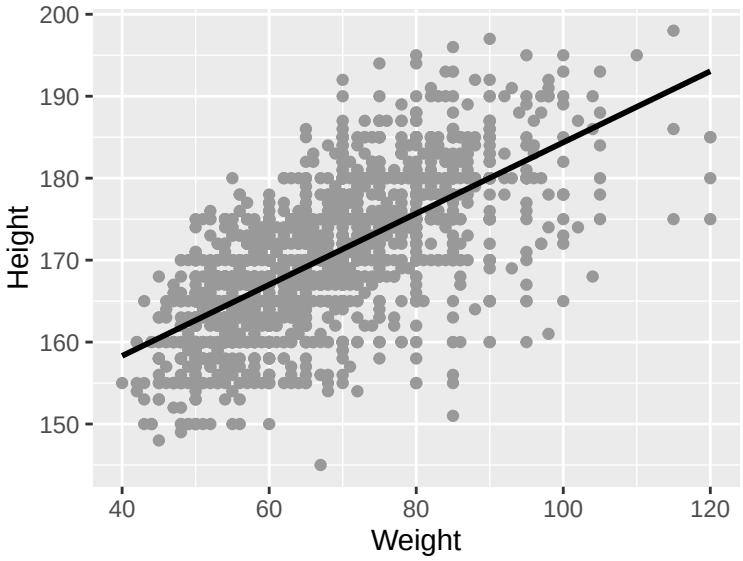
`method` argument is specified as `lm`. This means that the data is fitted with the `lm()` (linear model) function and a linear regression line is represented.
By default, a 95% confidence region for the regression fit is added. The confidence interval can be disabled specifying `se=FALSE`:

```
ggplot(istat, aes(Weight, Height)) +
  geom_point() +
  stat_smooth(method=lm, se=FALSE)
```



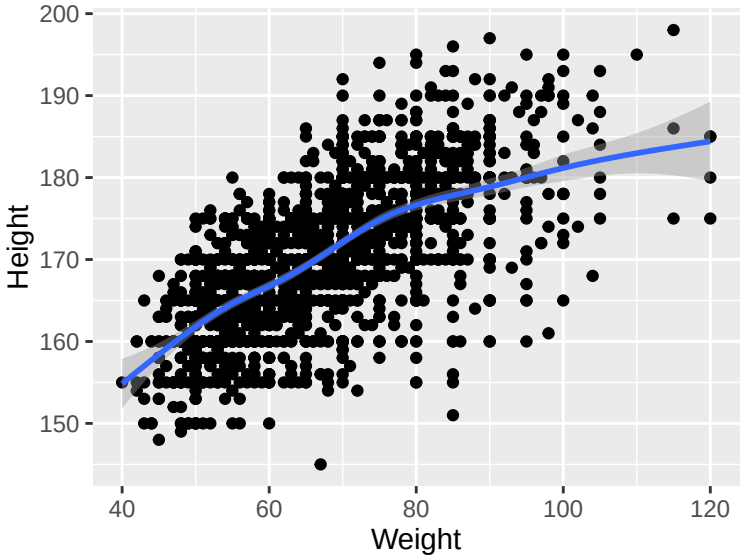
The default color of the fit line is blue. This can be change by setting `colour`. As with any other line, the attributes `linetype` and `size` can also be set. To emphasize the line, you can make the dots less prominent by setting `colour`:

```
ggplot(istat, aes(Weight, Height)) +
  geom_point(colour="grey60") +
  stat_smooth(method=lm, se=FALSE, colour="black")
```



The linear regression line is not the only way of fitting a model to the data, in fact, it is not even the default. If you add `stat_smooth()` without specifying the method, it will use a `loess` (locally weighted polynomial) curve. LOESS smoothing is a non-parametric form of regression that uses a weighted, sliding-window, average to calculate a line of best fit:

```
ggplot(istat, aes(Weight, Height)) +
  geom_point() +
  stat_smooth()
```



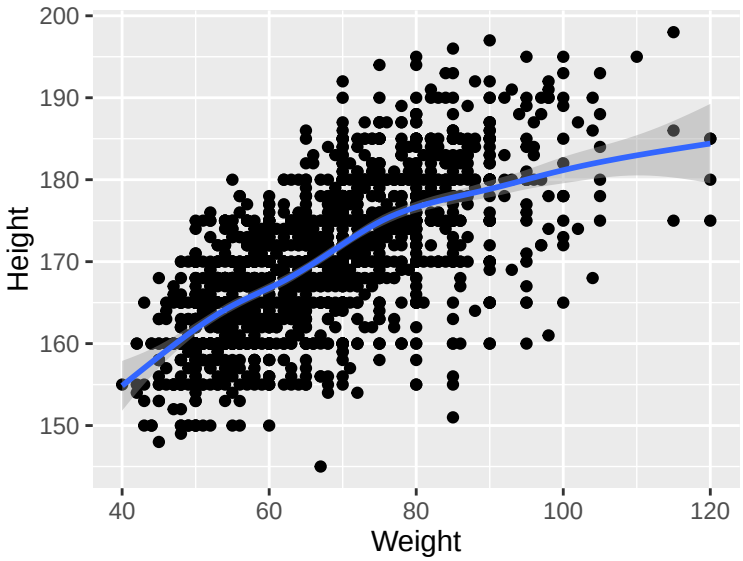
Remember that the previus code is equal to:

```
ggplot(istat, aes(Weight, Height)) +
  geom_point() +
  stat_smooth(method = loess)
```

Additional parameters can be passed along to the `loess()` function by just passing them to `stat_smooth()`. For example, we can control the degree of smoothing by setting `span` argument:

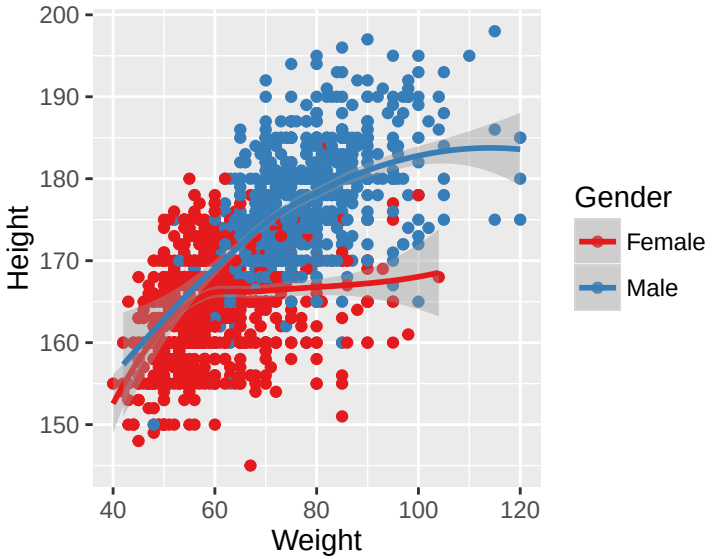
```
ggplot(istat, aes(Weight, Height)) +
  geom_point() +
```

```
  stat_smooth(span = 0.7)
```



Suppose we want to group points by `Gender` variable. If your scatter plot has points grouped by a factor, using `colour` or `shape`, one fit line will be drawn for each group:

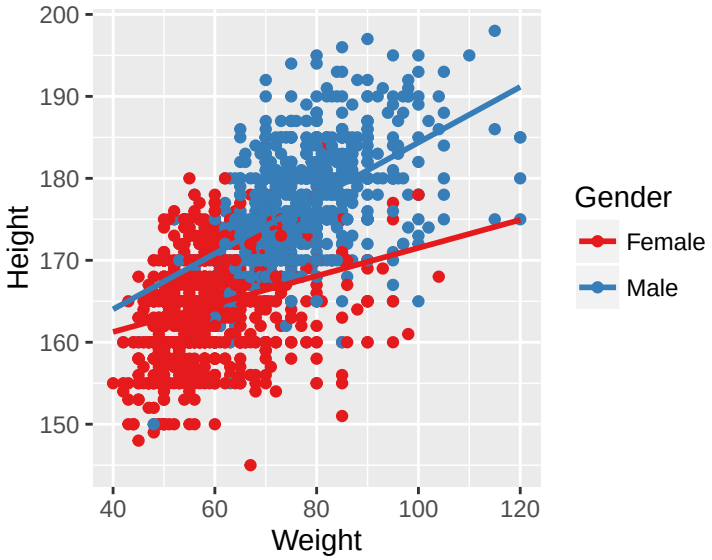
```
ggplot(istat, aes(Weight, Height, col =Gender)) +
  geom_point() +
  scale_colour_brewer(palette="Set1") +
  geom_smooth()
```



As you can see from the plot, a regression line has been drawn for Female (red line) and one for Male (blue line)

Notice that the red line, for Female, doesn't run all the way to the right side of the graph. There are two reasons for this. The first is that, by default, `stat_smooth()` limits the prediction to within the range of the predictor data (on the x-axis). The second is that even if it extrapolates, the `loess()` function only offers prediction within the x range of the data. If you want the lines to extrapolate from the data, you must use a model method that allows extrapolation, like `lm()`, and pass `stat_smooth()` the option `fullrange=TRUE`:

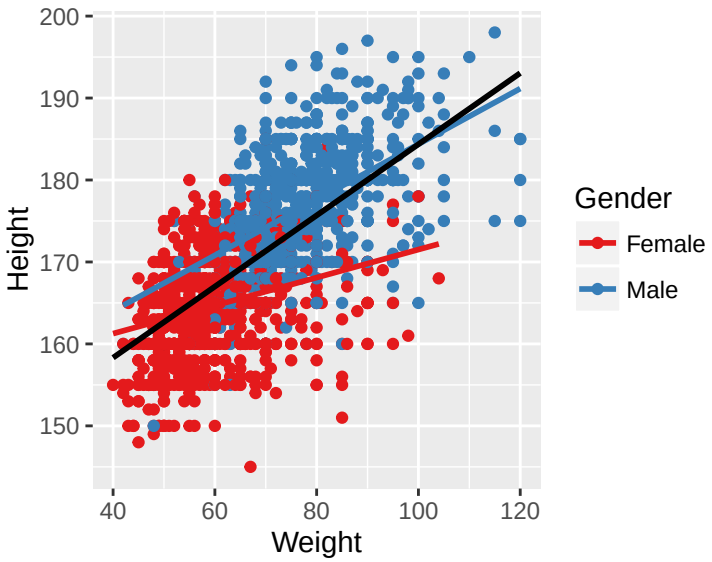
```
ggplot(istat, aes(Weight, Height, col =Gender)) +
  geom_point() +
  scale_colour_brewer(palette="Set1") +
  geom_smooth(method=lm, se=FALSE, fullrange=TRUE)
```



In this example with the `istat` data set, the default settings for `stat_smooth()` (with LOESS and no extrapolation) make more sense than the extrapolated linear predictions, because we don't grow linearly and we don't grow forever.

Suppose we want to add to the plot also a regression line for the whole observations. It is possible by adding another `stat_smooth()` function:

```
ggplot(istat, aes(Weight, Height, col = Gender)) +
  geom_point() +
  scale_colour_brewer(palette="Set1") +
  stat_smooth(method=lm, se=FALSE) +
  stat_smooth(mapping = aes(group = 1), method = "lm", se = F, col = "black")
```

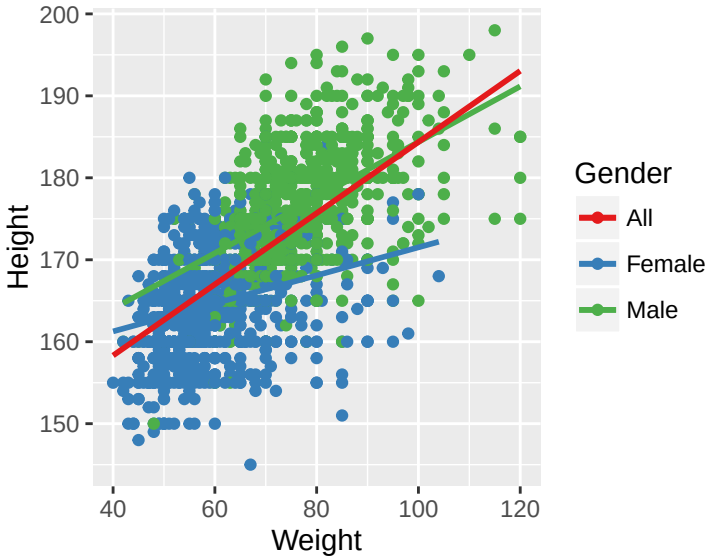


`group` aestethic set to 1 means that the whole observations must be considered for drawing the regression

line.

However, this plot presents a problem because there is a black line on our plot that is not included in the legend. To get this, we need to map something to `colour` as an aesthetic, not just set `colour` as an attribute.

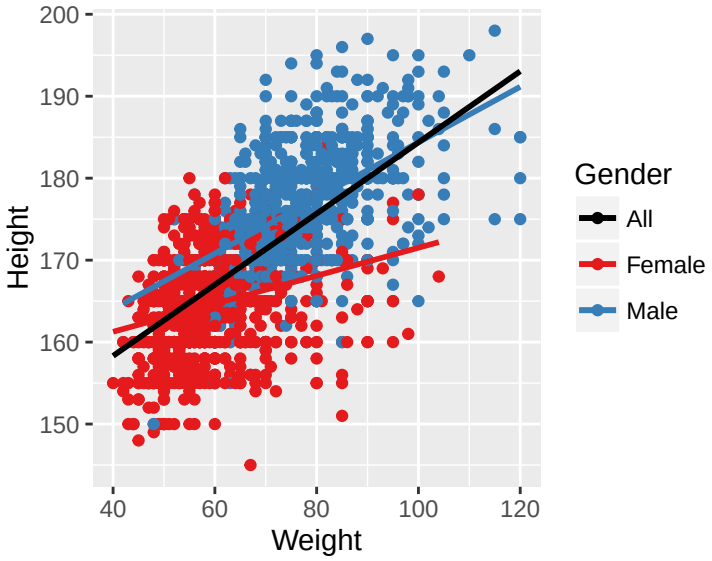
```
ggplot(istat, aes(Weight, Height, col =Gender)) +
  geom_point() +
  scale_colour_brewer(palette="Set1") +
  stat_smooth(method=lm, se=FALSE) +
  stat_smooth(mapping = aes(group = 1, col = "All"), method = "lm", se = F)
```



So, `colour` has been added to the `aes()` function in the second `stat_smooth()`, setting it to `"All"`. This will name the line properly. Pay attention that the `colour` attribute has been removed in the second `stat_smooth()`, otherwise, it would be overwrite the `colour` aesthetic. Now we should see our `"All"` model in the legend, but it's not black anymore.

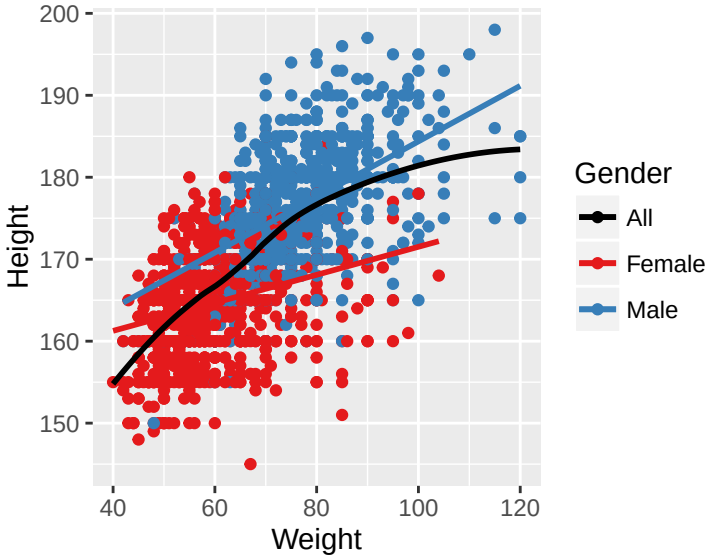
This is a way to change the colours:

```
# colours definition
myColors <- c( "black", brewer.pal(2, "Set1"))

ggplot(istat, aes(Weight, Height, col =Gender)) + geom_point() +
  stat_smooth(method = "lm", se = F) +
  stat_smooth(method = "lm", aes(group = 1, col="All"), se = F) +
  scale_color_manual("Gender" , values = myColors)
```



Nothing prevents us to use different estimate method in our plots: a linear model for groups and a loess model for whole observations.

```
ggplot(istat, aes(Weight, Height, col =Gender)) +
  geom_point() +
  stat_smooth(method = "lm", se = F) +
  stat_smooth(method = "loess", aes(group = 1, col="All"), se = F) +
  scale_color_manual("Gender", values = myColors)
```



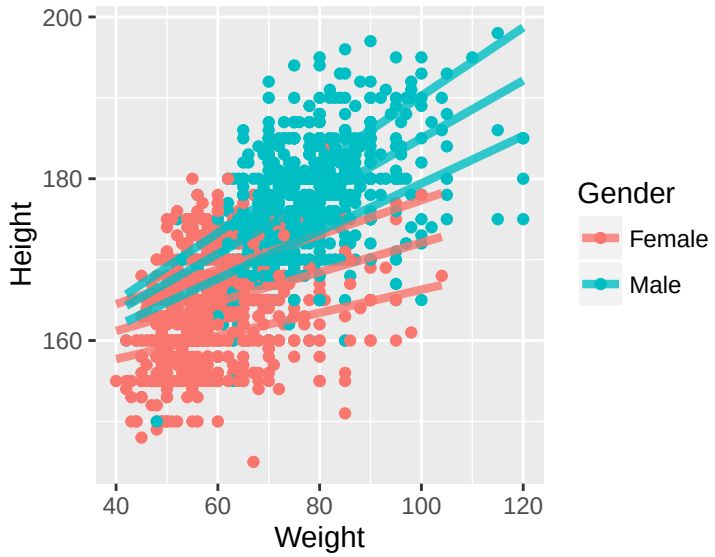
13.1.2 Add quantile lines from a quantile regression

Suppose you want to fit and represent quantiles in your plot. The statistics that add quantile lines from a quantile regression is `"quantile"`.

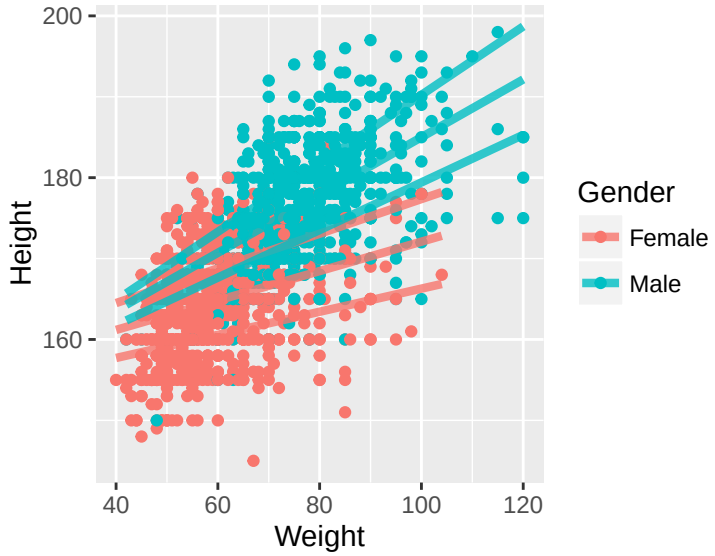
Let us consider the relationship between `Weight` and `Height` in `istat` dataset:

```
ggplot(istat, aes(Weight, Height, y, col = Gender, group = Gender)) +
  geom_point() +
```

```
  geom_quantile(alpha = 0.8, size = 1.5)
```



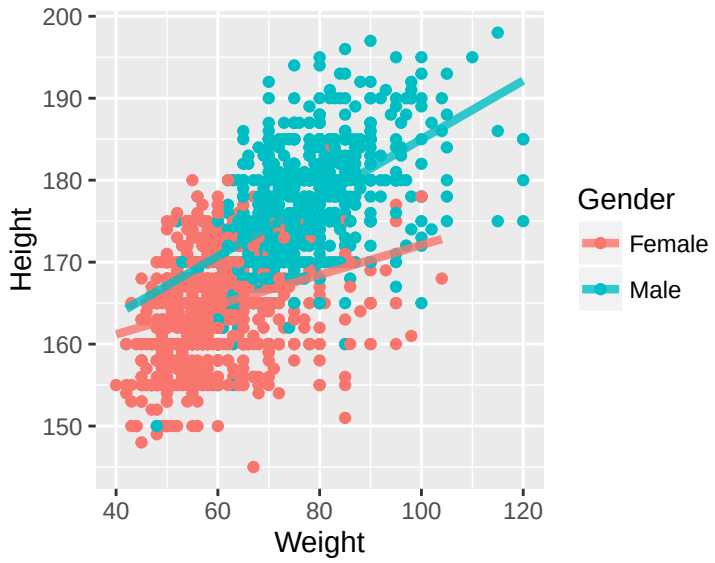

```
ggplot(istat, aes(Weight, Height, y, col = Gender, group = Gender)) +
  geom_point() +
  stat_quantile(alpha = 0.8, size = 1.5)
```



As you can see the result is the same, you can use either `stat_quantile()` or `geom_quantile()`.

Three quartiles are been drawn by default, which are: first quartile (0.25), median (0.5) and third quartile (0.75). If you want to fit and represent only the median, set `quantiles` argument to 0.5:

```
ggplot(istat, aes(Weight, Height, col = Gender, group = Gender)) + geom_point() +
  stat_quantile(alpha = 0.8, size = 1.5, quantiles = 0.5)
```



–>

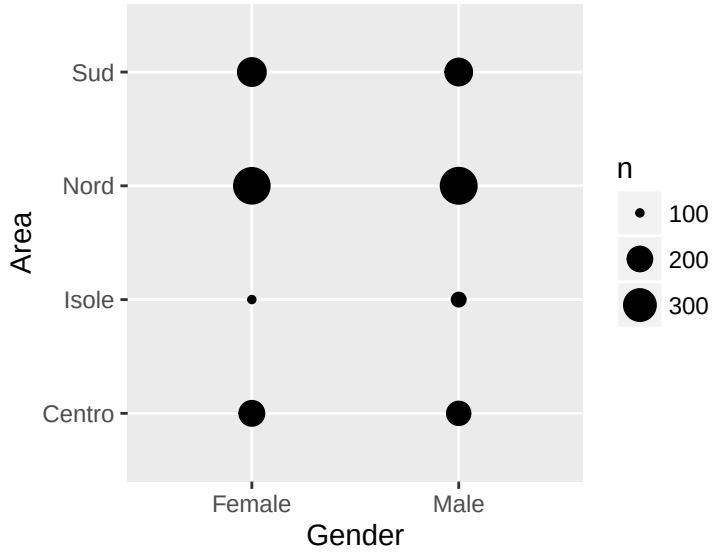
13.1.3 Count the number of observations at each location

Suppose you want to count the number of observation in each location. The reference statistic is `"sum"`.

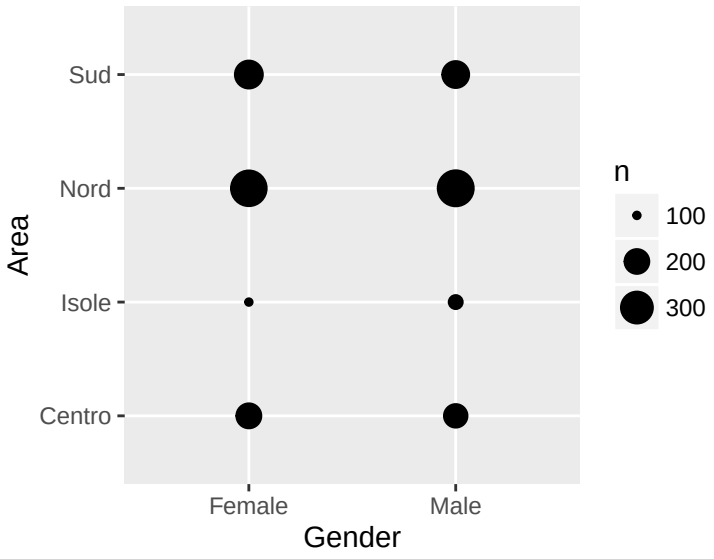
It is usaully used when you have discrete data.

Let us consider `Gender` and `Area` of `istat` dataset:

```
ggplot(istat, aes(Gender, Area)) +
  stat_sum()
```




```
ggplot(istat, aes(Gender, Area)) +
  geom_count()
```



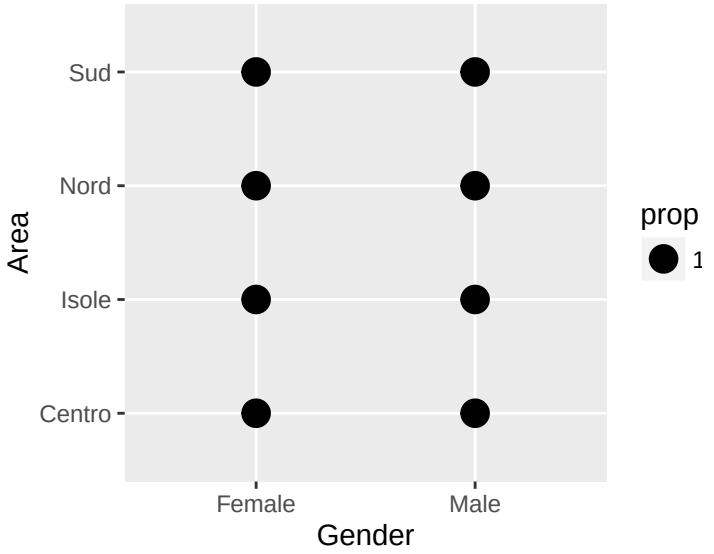
As you can see the result is the same, you can use either `stat_sum()` or `geom_count()`.

`"sum"` statistic creates two variables: `n` (count) and `prop`.

Each `stat` creates additional variables to map aesthetics to. Indeed, a stat internally takes a data frame as input and returns a data frame as output, and so a `stat` can add new variables to the original dataset. These variables use a common `..name..` syntax. This prevents confusion in case the original dataset includes a variable with the same name as a generated variable, and it makes it clear to any later reader of the code that this variable was generated by a stat. Each statistic lists the variables that it creates in its documentation.

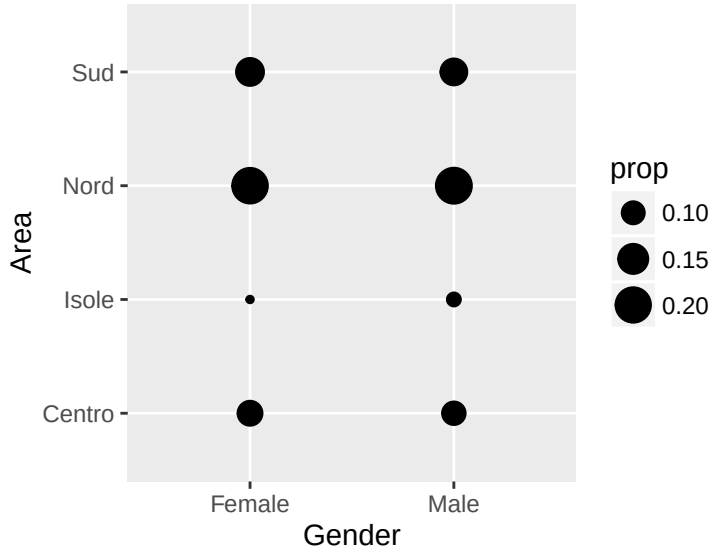
We can use these variables for building the plot, in this case you can display proportions instead of counts, mapping `size` aesthetic to `..prop..` variable:

```
ggplot(istat, aes(Gender, Area)) +
  stat_sum(aes(size = ..prop..))
```



By default, all categorical variables in the plot form the groups. Specifying `stat_sum()` without a group identifier leads to a plot which is not useful. We need to specify which group the proportion is to be calculated over by mapping `group` aesthetics:

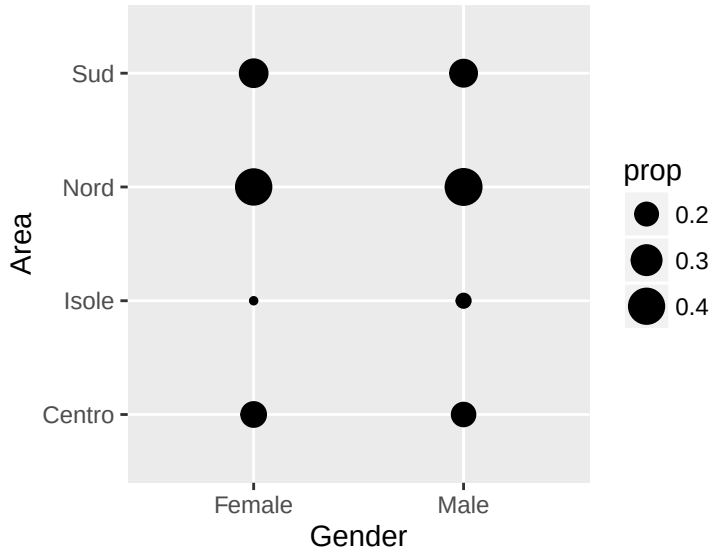
```
ggplot(istat, aes(Gender, Area)) +
  stat_sum(aes(size = ..prop.., group = 1))
```



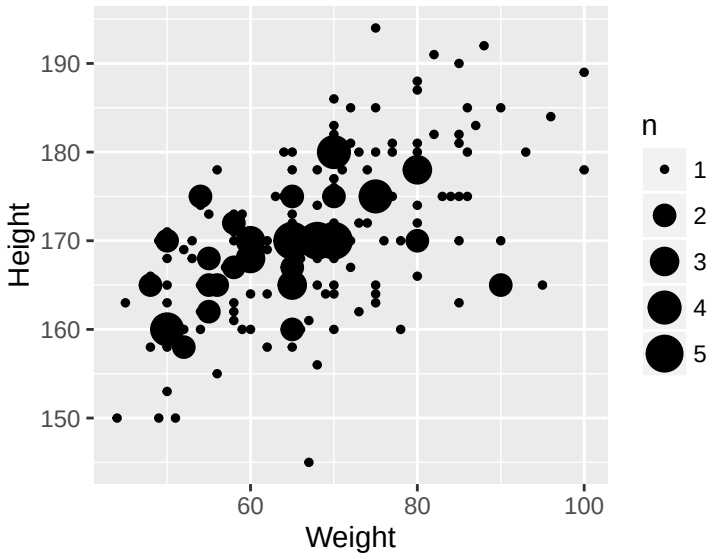
In this case the proportion is to be calculated over is the whole dataset, as `group = 1`.

We can map group also to `Gender` or `Area`. For example, if we map group to `Gender` the proportions will be computed over it:

```
ggplot(istat, aes(Gender, Area)) +
  stat_sum(aes(size = ..prop.., group = Gender))
```




```
ggplot(istat_sample, aes(Weight, Height)) +
   geom_count()
```

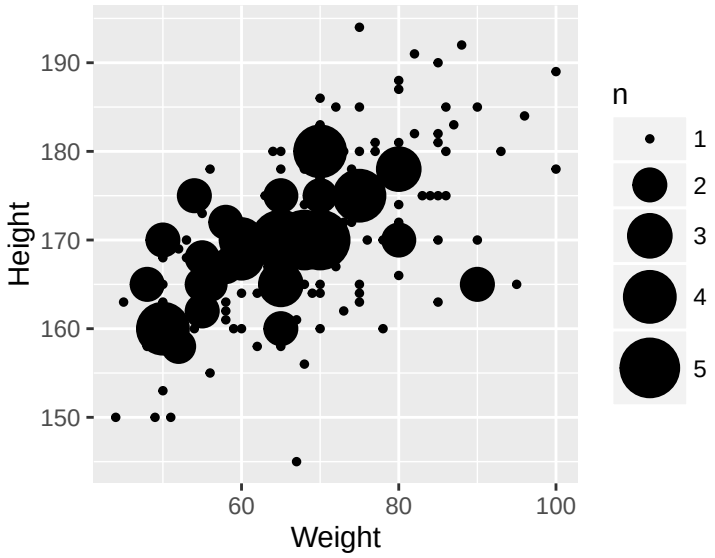


As you can see the result is the same, you can use either `stat_sum()` or `geom_count()`.

We worked on a sample as with the whole dataset the result is incomprehensible.

You can adjust the size scale with the generic `scale_size()` function, by setting the range argument (minimum and maximum dot sizes) as c(1,10):

```
ggplot(istat_sample, aes(Weight, Height)) +
  stat_sum() +
  scale_size(range = c(1,10))
```



–>

13.2 Statistics outside geoms

Statistics outside geoms group includes statistics that can't be created with a `geom_` function.
The most important are:

- `stat_summary()`: summarises values
- `stat_function()`: plots a function

- `stat_qq()`: performs computations for quantile-quantile plot
- `stat_ecdf()`: represents the empirical cumulative density function

In the following paragraph we will see the functionalities of the statistical transformation listed in the previous bulleted list.

13.2.1 Plotting a Function

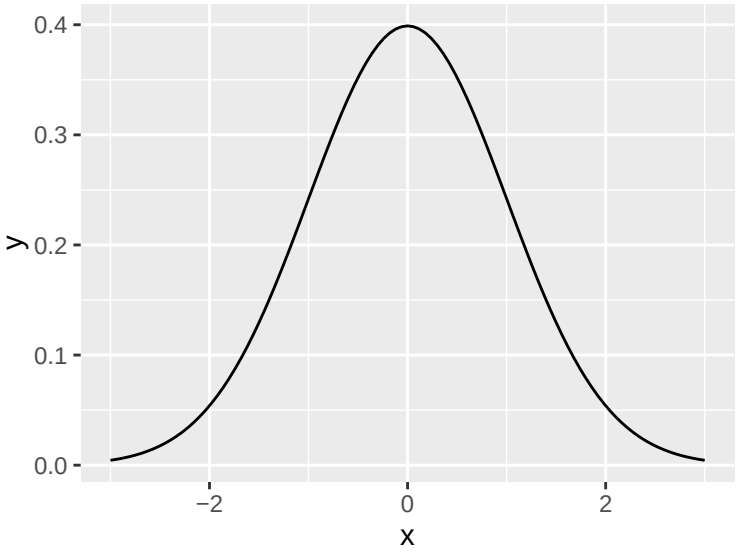
If you want to plot a function, you have to use `stat_function()`.

Suppose we want to plot the density function of a sample.

```
df <- data.frame(x=seq(from = -3, to = 3, length.out = 100))
```

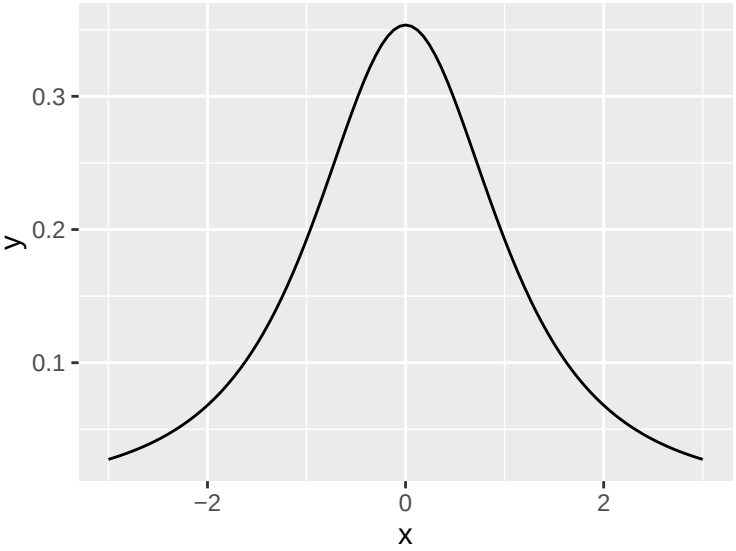
`dnorm()` is the function that gives the density of the normal distribution, so `fun` argument of `stat_function()` must be specified equal to `dnorm()`:

```
ggplot(df, aes(x=x)) +
  stat_function(fun = dnorm)
```



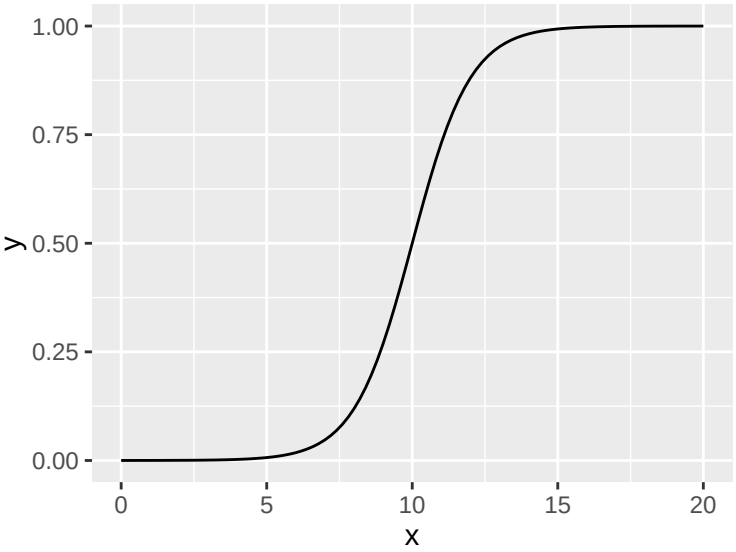
Some functions take additional arguments. For example, `dt()`, the function for the density of the t-distribution, takes a parameter for degrees of freedom. These additional arguments can be passed to the function by putting them in a list and giving the list to `args` argument of `stat_function()`:

```
ggplot(df, aes(x=x)) +
  stat_function(fun=dt, args=list(df=2))
```



You can also define your own functions. It should take an `x` value for its first argument, and it should return a `y` value. In this example, we define a sigmoid function:

```
# Function definition
myfun <- function(xvar) {
  1/(1 + exp(-xvar + 10))
}

df <- data.frame(x=c(0, 20))
# Plotting function
ggplot(df, aes(x=x)) +
  stat_function(fun=myfun)
```



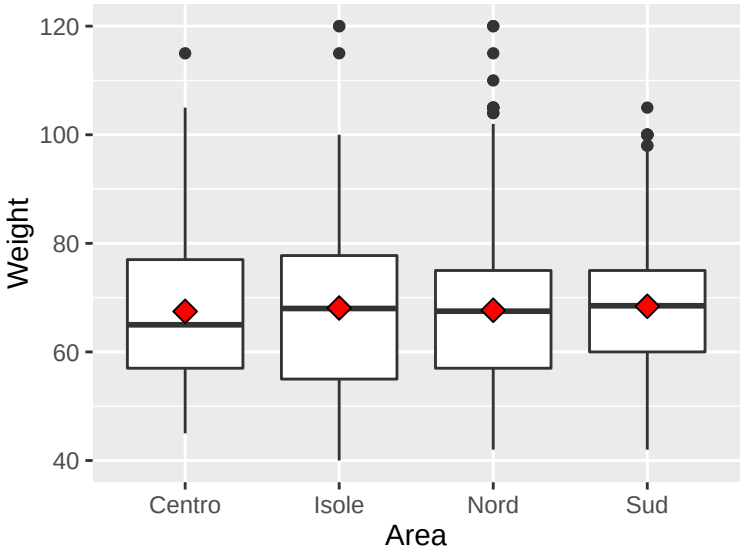
13.2.2 Adding means to a Box Plot

The horizontal line in the middle of a box plot displays the median, not the mean. For data that is normally distributed, the median and mean will be about the same, but for skewed data these values will differ.

The mean can be added to a Box Plot by using `stat_summary()` function.

Let us consider the distribution of `Weight` by `Area` in `istat` dataset:

```
ggplot(data=istat, aes(x=Area, y=Weight)) +
  geom_boxplot() +
  stat_summary(fun.y="mean", geom="point", shape=23, size=3, fill="red")
```



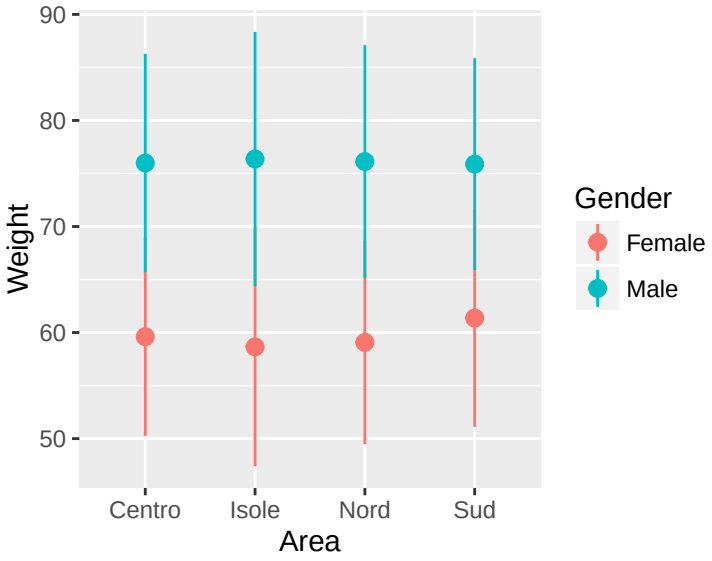
`fun.y` argument of `stat_summary()` function can be specified equal to the statistics that we want to compute, in this case the `"mean"`. There is also `geom` argument which specifies which `geom` has to be used to represent the computed means in the plot, in this case `"point"`.

Apart from mean, you can compute lots of summaries on your distribution.

`ggplot2` offers a selection of summary functions from `Hmisc` package to make it easy to use with `stat_summary()`.
Let us see some examples.

`mean_sdl()` function is a wrapper of `smean.sdl()` function of `Hmisc` package and computes the mean plus or minus a constant times the standard deviation.

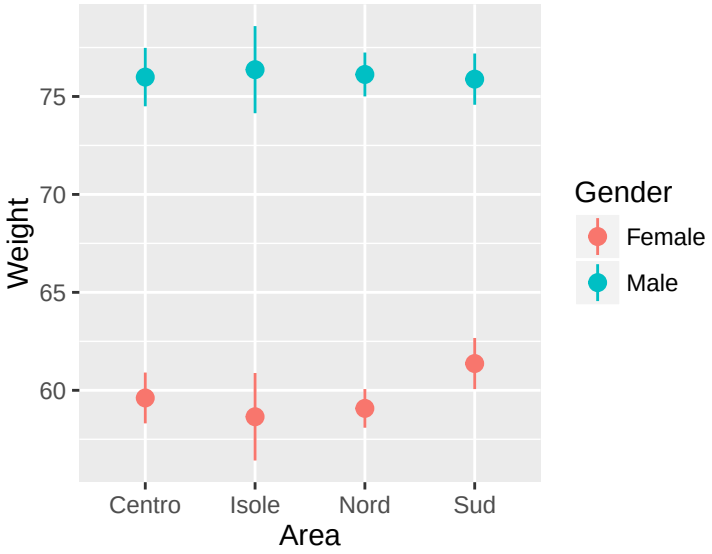
```
ggplot(data = istat, mapping=aes(x=Area, y=Weight, colour=Gender, fill=Gender, group=Gender)) +
  stat_summary(fun.data = mean_sdl, fun.args = list(mult = 1))
```



In this case the statistic function, `mean_sdl`, is given to `fun.data` argument, because `mean_sdl` returns a data frame with variables ymin, y, and ymax. In the previous example we gave `mean` function to `fun.y` argument because `mean` returns a single number.
`mult = 1` value passed to `fun.args` means that the `mult` argument of `mean_sdl` function must be set to 1. `mult` argument represent themultiplier of the standard deviation used in obtaining a coverage interval about the sample mean.

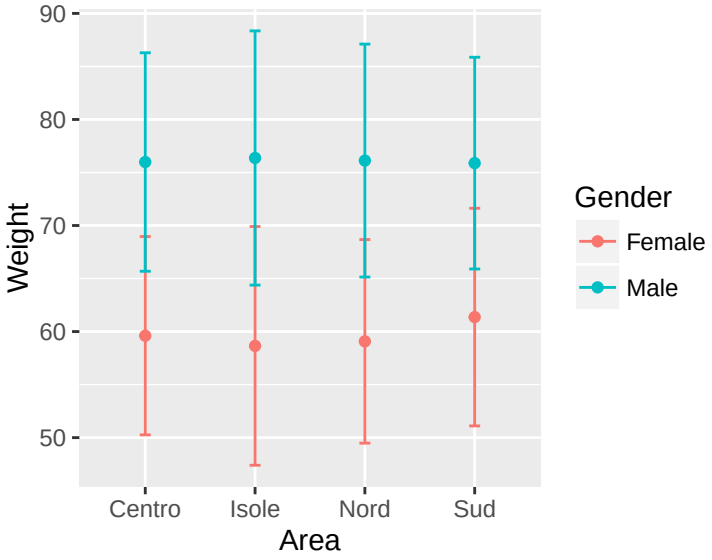
`mean_cl_normal()` is a wrapper of `smean.cl.normal()` function of `Hmisc` package that computes 3 summary variables: the sample mean and lower and upper Gaussian confidence limits based on the t-distribution (95% Confidence Interval):

```
ggplot(data = istat, mapping=aes(x=Area, y=Weight, colour=Gender, fill=Gender, group=Gender)) +
  stat_summary(fun.data = mean_cl_normal)
```



You can plot an error bar of the mean in this way:

```
ggplot(data = istat, mapping=aes(x=Area, y=Weight, colour=Gender, fill=Gender, group=Gender)) +
  stat_summary(fun.y = mean, geom = "point") +
  stat_summary(fun.data = mean_sdl, fun.args = list(mult = 1), geom = "errorbar", width = 0.1)
```



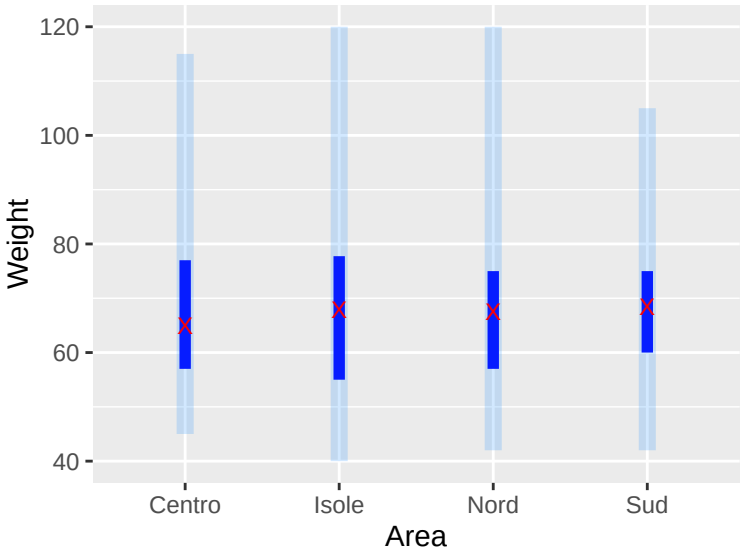
You can also pass your own functions to `stat_summary()`. Suppose we want to produce a different version of boxplot.

Firstly we have to define two functions: one that returns the range of observations and one that computes the median and the interquartile range.

```
# Function definition: range of obs
pl_range <- function(x) {data.frame(ymin = min(x), ymax = max(x))}
# Function definition: median and interquartile range
med_iqr <- function(x) {data.frame(y = median(x), ymin = quantile(x, 0.25), ymax = quantile(x, 0.75))}
```

Our version of boxplot can be produced in this way:

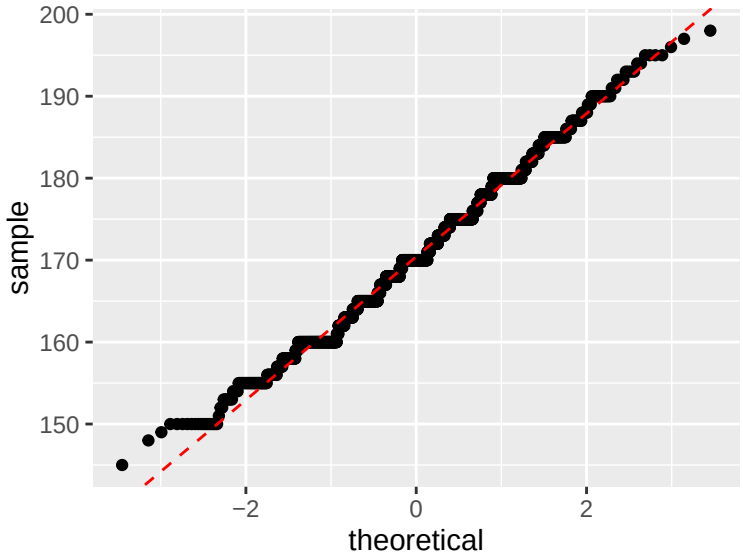
```
ggplot(data = istat, mapping=aes(x=Area, y=Weight)) +
  stat_summary(fun.data = med_iqr, geom = "linerange", size = 2, col = "blue") +
  stat_summary(fun.data = pl_range, geom = "linerange", size = 3, alpha = 0.2, col = "dodgerblue") +
  stat_summary(fun.y = median, geom = "point", size = 3, col = "red", shape = "X")
```



13.2.3 Calculation for quantile-quantile plot.

Suppose you want to plot the quantile-quantile plot of `Height` variable in `istat` dataset. It is not simple to generate a quantile-quantile plot with `ggplot2`.
This can be a way:

```
ggplot(data = istat, mapping = aes(sample = Height)) +
  stat_qq() +
  geom_abline(mapping = aes(intercept=mean(Height),slope=sd(Height)), color="red", linetype=2)
```



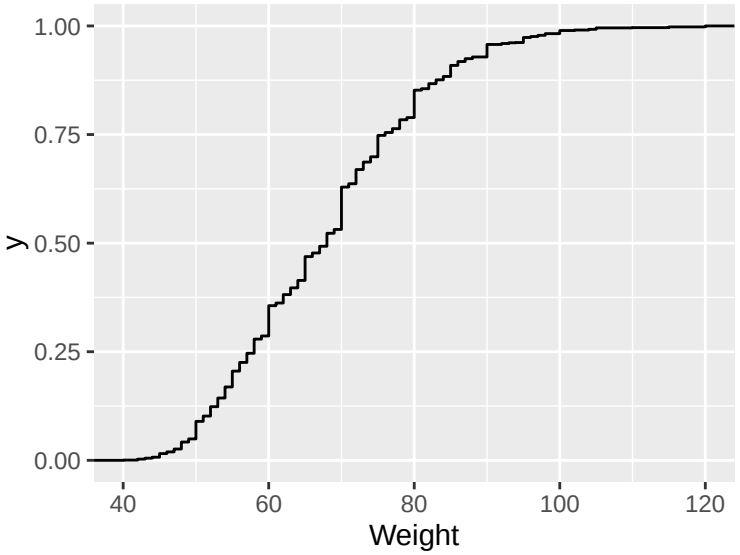
You have to specify sample quantiles in aesthetics, in this case the sample quantile is `Height` variable.

In particular, base graphics `qqnorm()` function is reproduced by `stat_qq()` and `qqline()` by `geom_abline()`, with intercept equal to the mean of the variable for which the qq-plot has to be drawn and the slope equal to the variance.

13.2.4 Creating a Graph of an Empirical Cumulative Distribution Function

Suppose you want to graph the empirical cumulative distribution function (ECDF) of `Weight` variable of `istat` dataset. You have to use `stat_ecdf()` function

```
ggplot(istat, aes(x=Weight)) +
  stat_ecdf()
```



The ECDF shows what proportion of observations are at or below the given x value. Because it is empirical, the line takes a step up at each x value where there are one or more observations.

14 ggplot2 community

```
require(ggplot2)
require(GGally)
require(ggmap)
require(qdata)
require(survival)
data(lung)
data(italy)
data(bottlecap)
data(istat)
```

ggplot2 is a community that is flourishing day by day. There are packages developed starting from ggplot2 grammar, the so-called ggplot2 extensions, like GGally and packages designed to work with ggplot2, like ggmap. All of them contribute to the growth of ggplot2 community. This means that also you can easily create your own stats, geoms and positions, and provide them in other packages, which will become ggplot2 extensions. You can find a list of the official ggplot2 extensions at www.ggplot2-exts.org. These packages are available on CRAN and/or on Github. Enjoy yorself with them!

In this chapter we will analyze the two most important packages linked to ggplot2:

- GGally, an extension of ggplot2 for correlation matrix and survival plots
- ggmap, for Spatial Visualization with ggplot2

14.1 GGally

GGally is a convenient package built upon ggplot2 for correlation matrix and survival plots written and mantained by Barret Schloerke. It reduces the complexity of combining geometric objects with transformed data.

GGally extends ggplot2 by providing several functions including:

- ggcor(): for pairwise correlation matrix plot
- ggpairs(): for scatterplot plot matrix
- ggsurv(): for survival plot

Supposing the package is already installed, first of all GGally must be loaded.

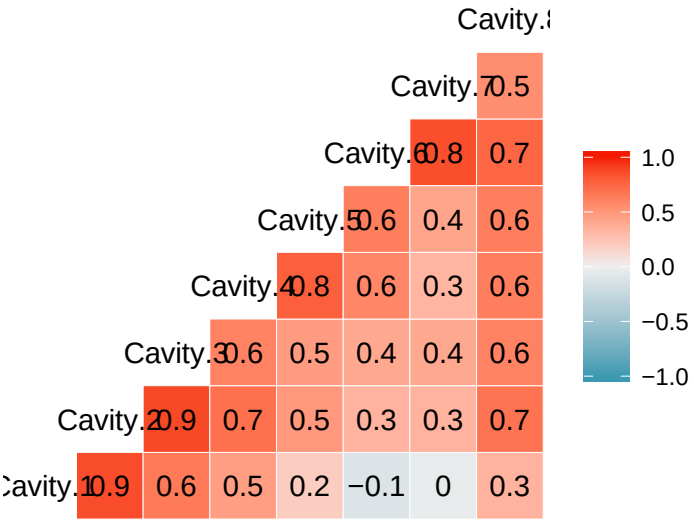
```
require(GGally)
```

14.1.1 ggcorr(): Plot a correlation matrix

The function ggcorr() draws a correlation matrix plot using ggplot2.

Let us see an example considering bottlecap dataset. bottlecap contains measures of the mean diameter of the caps produced by 8 different cavity of a forging machine during the quality control phase. Suppose we want to know if there is correlation between the measures performed by the 8 cavity in order to verify if it is possible to reduce the measuring cavity.

```
ggcorr(data = bottlecap, palette = "RdBu", label = TRUE)
```



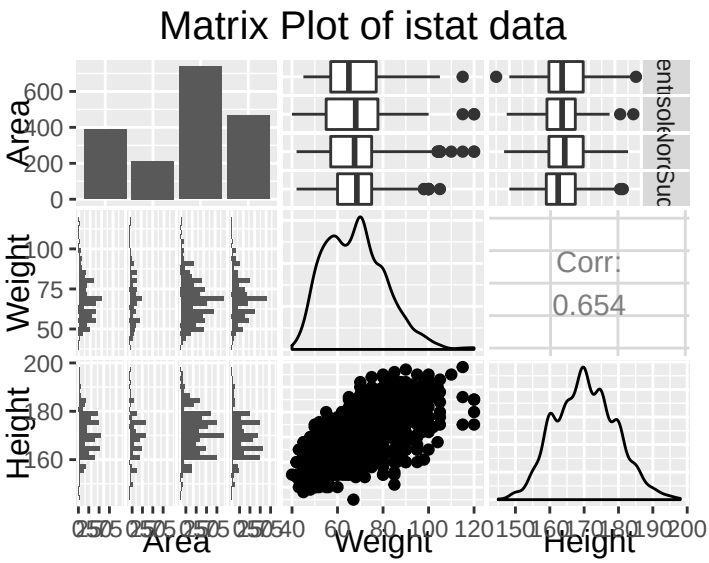
`label` argument is set to `TRUE` in order to add correlation coefficients to the plot.

14.1.2 `ggpairs()`: for scatterplot plot matrix

`ggpairs()` function contains templates for different plots to be combined into a plot matrix. It is a nice alternative to the more limited `pairs` function of `graphics` package.

Let us see an example considering `istat` dataset.

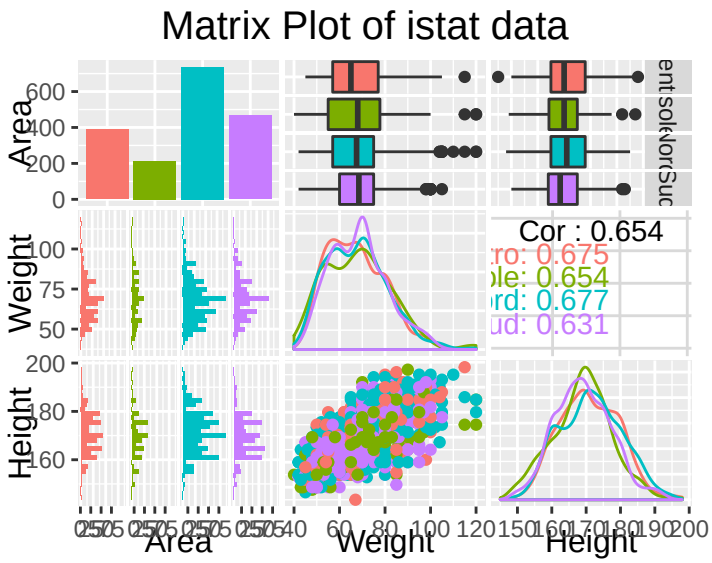
```
ggpairs(data=istat, # dataframe with variables
        columns = 2:4, # columns to be used to make plots
        title="Matrix Plot of istat data") # title of the plot
```



Plots like the one above are very helpful, among others things, in the pre-processing stage of a classification problem, where you want to analyze your predictors given the class labels. It is particularly amazing that we can now use the arguments `colour`, `shape`, `size` and `alpha` provided by `ggplot2`:

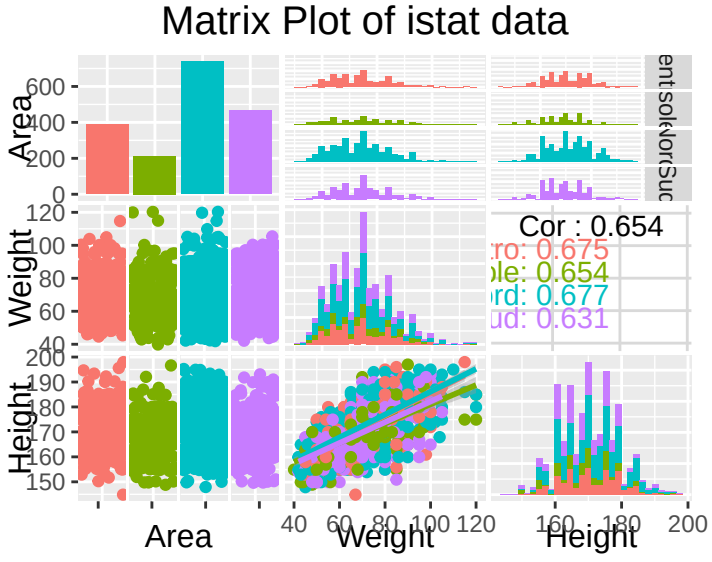
```
ggpairs(data=istat, # data.frame with variables
        mapping = aes(colour=Area), # esthetic mapping (besides x and y)
```

```
        columns = 2:4, # columns to be used to make plots
        title="Matrix Plot of istat data") # title of the plot
```



We have some control over which type of plots to use. We can choose which type of graph will be used for continuous vs. continuous (`continuous`) and discrete vs. discrete (`discrete`) and continuous vs. discrete (`combo`). We can also have different plots for the upper diagonal (`upper`), for the diagonal (`diagonal`) and for the lower diagonal (`lower`).

```
ggpairs(data=istat,
        mapping = aes(colour=Area),
        columns=2:4,
        upper=list(continuous = 'cor', discrete = 'facetbar', combo ='facethist'),
        lower=list(continuous = 'smooth', discrete = 'facetbar', combo ='dot'),
        diag=list(continuous = 'barDiag', discrete = 'barDiag'),
        title="Matrix Plot of istat data")
```



For example, the code above, set to use in the upper diagonal: correlations for continuous vs. continuous variables, faceted bar plot for discrete vs. discrete variables and faceted histograms for continuous vs. discrete variables.

14.1.3 ggsurv(): for survival plot

ggsurv() produces Kaplan-Meier plots using ggplot2.

Let us see an example, considering lung data, which is about survival in patients with advanced lung cancer.

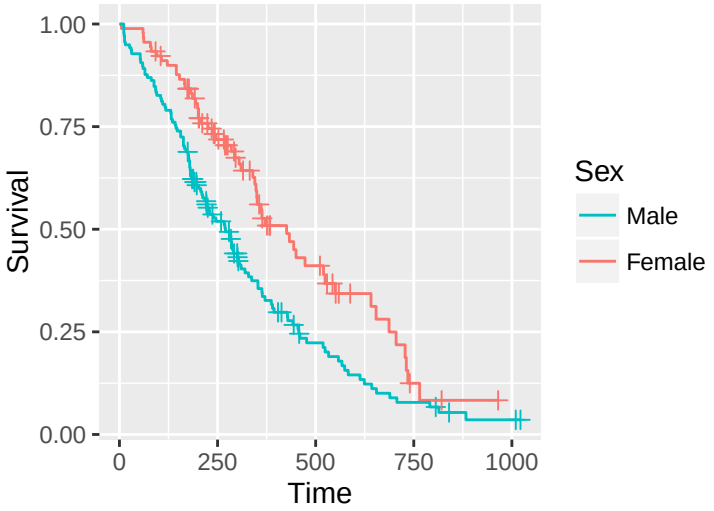
```
lung <- lung[, c(2,3,5)]
```

We consider only the following variables:

- time: Survival time in days
- status: censoring status 1 = censored, 2 = dead
- sex: Male = 1; Female = 2

As a first argument ggsurv() needs a survfit object, created by the survival package:

```
# Fit survival functions
surv <- survfit(Surv(time, status) ~ sex, data = lung)
# Plot survival curves
ggsurv(surv) +
  guides(linetype = FALSE) +
  scale_colour_discrete(name   = 'Sex', breaks = c(1,2),
                        labels = c('Male', 'Female'))
```



14.2 ggmap

ggmap is a package for Spatial Visualization with ggplot2 written by David Kahle and Hadley Wickham.

In particular, ggmap enables such visualization by combining the spatial information of static maps from Google Maps, OpenStreetMap, Stamen Maps or CloudMade Maps with the layered grammar of graphics implementation of ggplot2.

ggmap plots have the same elements of ggplot2, but certain elements are fixed to map components :

- the x aesthetic is fixed to longitude,
- the y aesthetic is fixed to latitude,
- the coordinate system is fixed to the Mercator projection

The basic idea driving ggmap is to take a downloaded map image, plot it as a context layer using ggplot2, and then plot additional content layers of data, statistics, or models on top of the map.

In ggmap this process is broken into two pieces:

1. downloading the images and formatting them for plotting, done with `get_map()`
2. making the plot, done with `ggmap()`.

Let us see an example.

We want to identificate the positions of italian most important cities.

1. First of all we download italian map:

```
# get italy map
italy_map <- get_map(location="Italy", zoom = 6, maptype = "satellite")
```

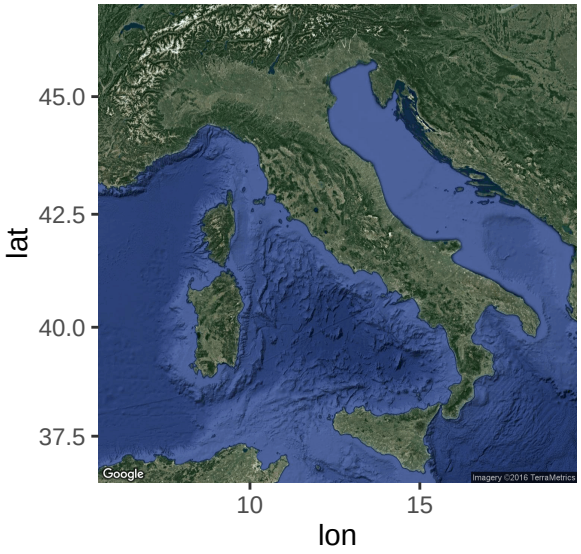
```
## Map from URL : http://maps.googleapis.com/maps/api/staticmap?center=Italy&zoom=6&size=640x640&scale=
```

```
## Information from URL : http://maps.googleapis.com/maps/api/geocode/json?address=Italy&sensor=false
```

`location` argument is specified as an "address", it can be specified also by a latitude/longitude pair.
`zoom` argument specify the level of map zoom; it can be spacified from 3 (continent) to 21 (building) the default is 10 (city). 6 level of zoom returns countries. `maptype` specify map theme. Some of the options are: `"terrain"`, `"toner"`, `"watercolor"`, `"roadmap"`, `"satellite"`, `"hybrid"`.

2. We make the plot visualizing firstly only the downloaded map, by using `ggmap()` function:

```
ggmap(italy_map)
```



We can add information about the location of most important cities included in `italy` dataset:

```
head(italy)
```

```
## # A tibble: 6 × 5
##            city      lat       lon      pop         region
##           <chr>    <dbl>     <dbl>    <dbl>          <chr>
## 1       Potenza 40.64200 15.798996  69060.0     Basilicata
## 2    Campobasso 41.56300 14.655997  50762.0         Molise
## 3         Aosta 45.73700  7.315003  34062.0  Valle d'Aosta
## 4        Modena 44.65003 10.919995 175034.5 Emilia-Romagna
## 5       Crotone 39.08334 17.123337  59313.5       Calabria
## 6 Vibo Valentia 38.66659 16.100040  32168.0       Calabria
```

```
ggmap(italy_map) +
  geom_point(aes(colour = region), data = italy)+
  geom_text(aes(label = city, colour = region), data = italy, size = 4, check_overlap = T,
```

```
          hjust = 0, nudge_x = 0.05) +
ggtitle("Map of most the important italian cities")+
labs(colour="Region", label ="Region") +
theme(axis.title=element_blank(),
      plot.title=element_text(size = 22, face = "bold"),
      legend.title=element_text(size = 16))
```

15 Presenting Plots

```
require(dplyr)
require(ggplot2)
require(qdata)
data(bands)
```

Winston Chang wrote on his book:

> Broadly speaking, visualizations of data serve two purposes: discovery and communication. In the discovery phase, you'll create exploratory graphics, and when you do this, it's important to be able try out different things quickly. In the communication phase, you'll present your graphics

to others. When you do that, you'll need to tweak the appearance of the graphics, and you'll usually need to put them somewhere other than on your computer screen.

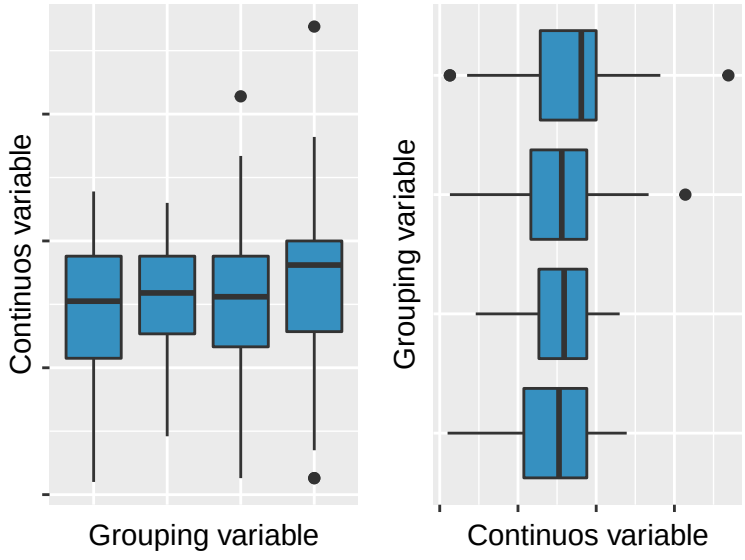
This chapter show how to output your plots for presentation.

15.1 Reassembling Plots for Printing

When you produce plots for printing, you may want several plots in a single image. The `grid.arrange()` function from `gridExtra` package, provide an easy-to-use solution. You just provide `ggplot` objects and the number of columns (or rows). Before you use the `gridExtra` package for the first time, may be you need to install it.

The following example returns the first plot of the previous chapter, with two box plots put side by side.

```
gp0 <- ggplot(data=bands, aes(x=press_type, y=ink_pct)) +
  geom_boxplot(fill="#3690c0") +
  ylab("Continuos variable") + xlab("Grouping variable") +
  theme(
    axis.text.x = element_blank(), axis.text.y = element_blank()
  )

gp1 <- gp0 + theme(axis.ticks.x = element_blank())

gp2 <- gp0 + theme(axis.ticks.y = element_blank()) + coord_flip()

gp <- gridExtra::grid.arrange(gp1, gp2, ncol=2)
```

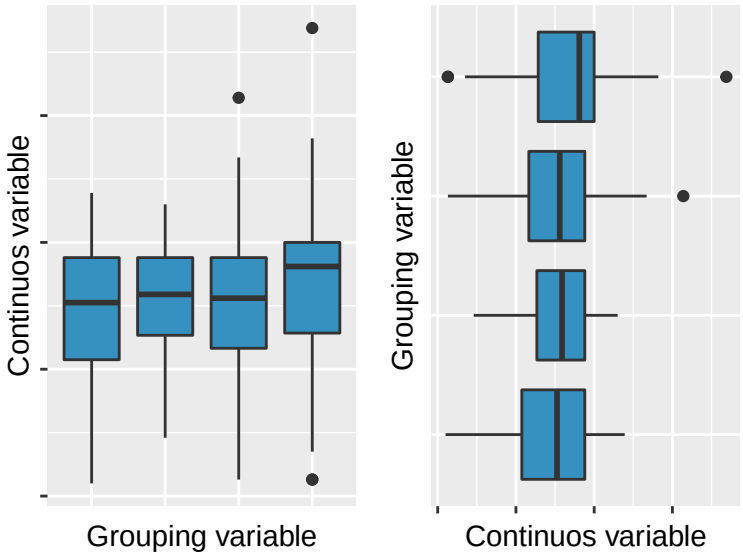


Notice that you have to use the `plot()` function to get the object saved by `grid.arrange()`.

```
gp
```

```
## TableGrob (1 x 2) "arrange": 2 grobs
##   z     cells    name           grob
## 1 1 (1-1,1-1) arrange gtable[layout]
## 2 2 (1-1,2-2) arrange gtable[layout]
```

```
plot(gp)
```



15.2 Outputting Plots to Vectorial Images

A *vectorial* image is a file containing the instructions to draw a plot. You can reshape images without any quality loss. Most used vectorial formats are PDF and SVG. Vectorial images can be edited with the well known commercial software "Adobe Illustrator" or its most famous free alternative "Inkscape".

To save an image to PDF, you can use `pdf()` or `cairo_pdf()` functions to open a new graphical device. The last uses the Cairo device that returns better images also on Mac and Linux. However sometimes `cairo_pdf` may return a *bitmap* PDF. If you want to be sure a vectorial PDF is produced, use `pdf()` function. To save an image to SVG, you can use `svg()`.

All plots called after the device is open, will be redirect to the file. At the end, you must close the device with `dev.off()`. If something goes wrong and plots are not returned, you must execute `dev.off()` to be sure that the graphical device is closed. Notice that you can open the file only after you run `dev.off()`.

```
pdf("gp1.pdf")
plot(gp)
dev.off()
```

```
cairo_pdf("gp2.pdf")
plot(gp)
dev.off()
```

```
svg("gp.svg")
plot(gp)
dev.off()
```

The most commonly options are `width` and `height` to set image dimension in inches.

```
cairo_pdf("gp3.pdf", width=12, height=8)
plot(gp)
dev.off()
```

An alternative to `pdf()`, `cairo_pdf()` and `svg()` functions is `ggsave()`. `ggsave` redirect the plot to an external file, and does not require opening and closing a new device. It guess the right device from file name extension. Unfortunately, it can save only one plot at time, so it does not work with `grid.arrange()` outputs.

By default `ggsave()` saves the last plot:

```
ggsave("boxplot1.pdf", width = 20, height = 20, units = "cm")
```

If you want to save another plot, you must specify its name:

```
ggsave("boxplot2.pdf", plot=gp1, width = 20, height = 20, units = "cm")
```

15.3 Outputting Plots to Bitmap Images

A *bitmap* image is a file containing which colour should be returned for each pixel. When you increase the size of a bitmap image, you get a loss of quality. Most used vectorial formats are PNG and TIFF. TIFF is an high quality bitmap format and may be required by your publisher. R supports other bitmap formats (BMP, JPEG, ...) but there is no reason to use them. Bitmap images can be edited with the well known commercial software "Adobe Photoshop" or its most famous free alternative "The Gimp".

Images with geometric objects should always saved as vectorial images, while images with a lot of details (e.g. photos) are bitmap images. Following this rule, it is better to save a plot as a vectorial image but there is one main exception. If you have one million points overlapped, the vectorial file contains one million instructions to draw points and the file will be much larger than bitmap image that should draw only a single point.

```
png("gp.png", width=1200, height=800, res=300)
plot(gp)
dev.off()
```

By default with bitmap formats, **`width`** and **`height`** are not in inches but in pixel. You can change the **`units`**. You must also specify resolution (**`res`**), that is how many pixels there are in an inch. Increasing the **`res`** value, you get better plots but bigger files. Good choices are 72 for screen use (e.g. presentations or web sites) and 300 or 600 when printing.

When only one plot should be saved, `ggsave()` can be used. Note that **`width`** and **`height`** are in **`inch`**es, but you can change the **`units`**. You can also specify resolution (**`dpi`**, dot per inch).

```
ggsave("boxplot.png", width = 4, height = 3, dpi = 300)
```