Using ggplot2 to produce meaningful plots

Dan MacLean

May 2022

Table of contents

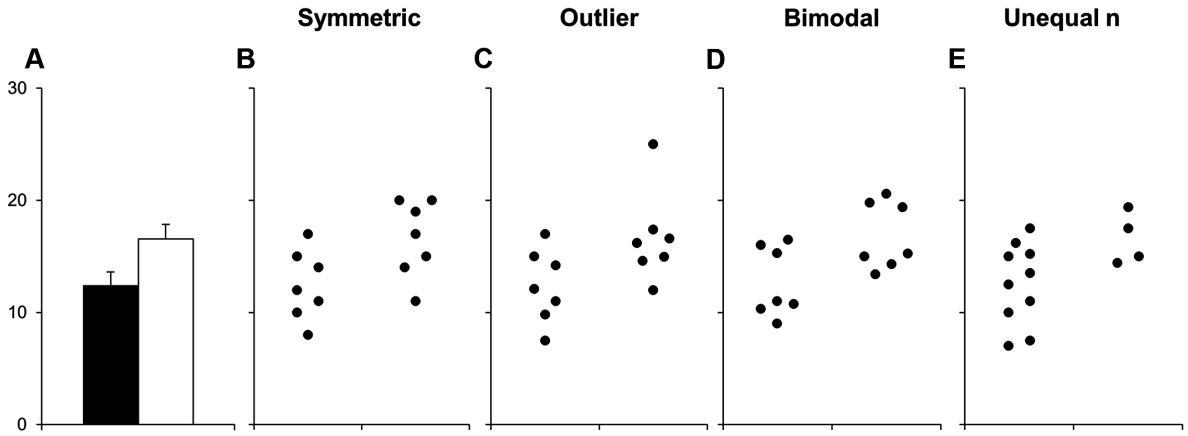
Motivation

Variability in measurements

Variability in measurements is a thing that happens as a natural consequence of working with complex systems that are affected by many variables in stochastic ways. Biological systems are some of the most variable we know. The variability in our experiments could be a function of the behaviour of the system yet it is common practice to hide that variability when we start to analyse our data by using summary plots like box-plots. Ultimately, that's bad news for our science, because the variability could be telling us something.

Summarising your data can lead to wrong conclusions

We all know that when you create a bar chart and put some error bars on it, you're really only representing two numbers, usually a mean and standard deviation. People create bar plots instinctively, and in doing so can miss important stuff. Look at this figure:



Test	p value			
T-test: Equal var.	0.035	0.050	0.026	0.063
T-test: Unequal var.	0.035	0.050	0.026	0.035
Wilcoxon	0.054	0.073	0.128	0.103

source: Weissgerber et al

The bar chart in panel A is one that came out of all those sets of numbers in the other panels. But it really hides some important stuff, like the fact the numbers are clearly separating into two groups in panel D, or that the two samples have different sizes in panel E.

Worse than any of these is that the significant difference in the t-test is coming from just one point in panel C. From this data set you might be tempted to conclude that there is a significant difference in the two samples and if you relied on the bar chart as a visualisation then you'd never suspect there was something funny.

Some enthusiastic young science communicators have even started a Kickstarter to lobby journals to stop using, in particular, bar charts! These people, calling themselves Bar Barplots, have a nice video on one of the main problems with bar charts. Have a look at this page on Kickstarter . Kickstarter - Barbarplots, especially this video Kickstarter - Barbarplots video.

Ignoring your data visualisation and just making bar plots could be an error! It's important that you spend a little time getting to know, and presenting your data as clearly and thoroughly as possible.

p - one value to fool them all?

But why would you care about this, in the end a *p*-value won't a *p*-value help you see real differences and make this all easy? Sadly, that isn't true. Let's do an experiment to test that.

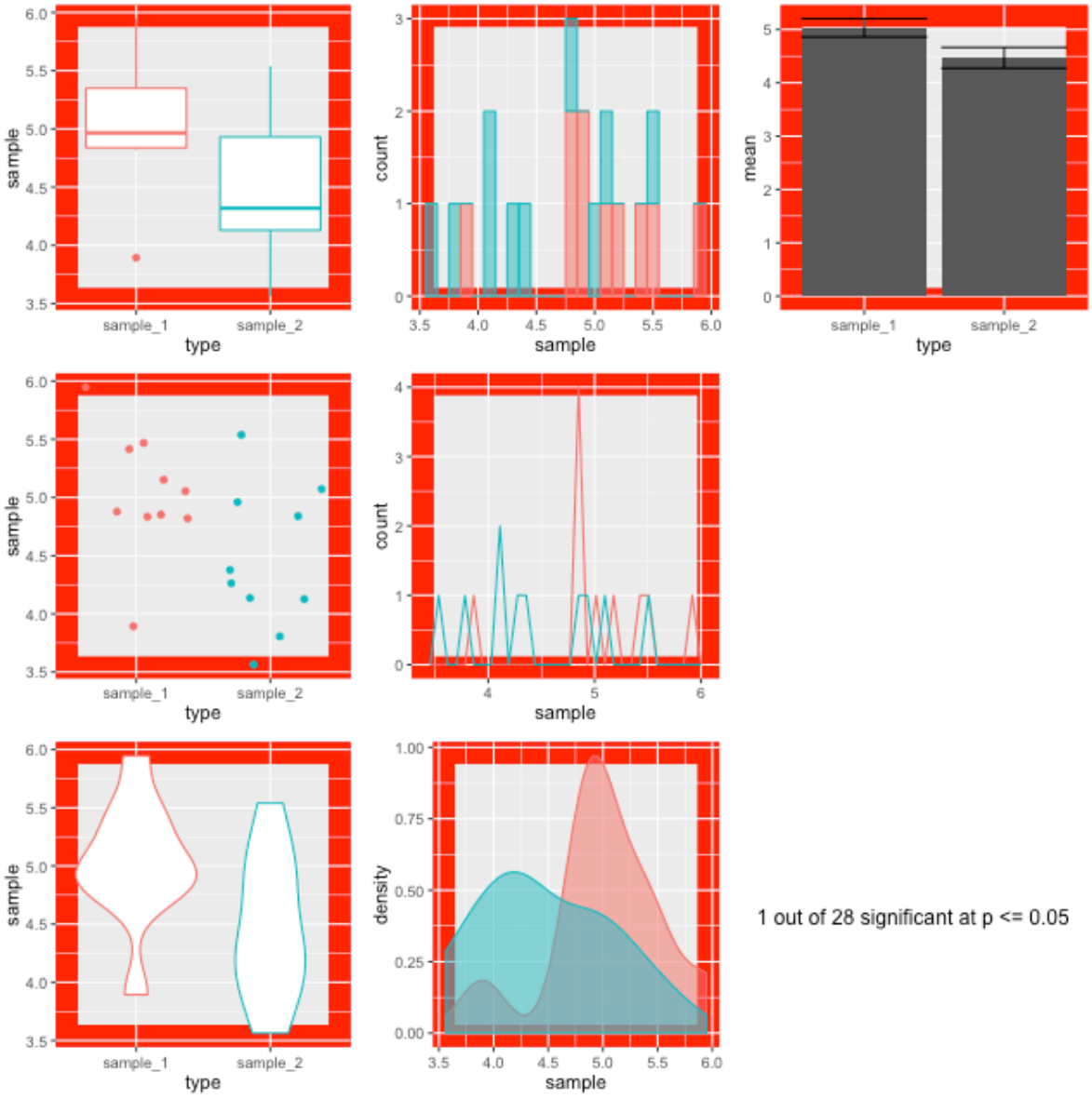
Ten Thousand Random Numbers

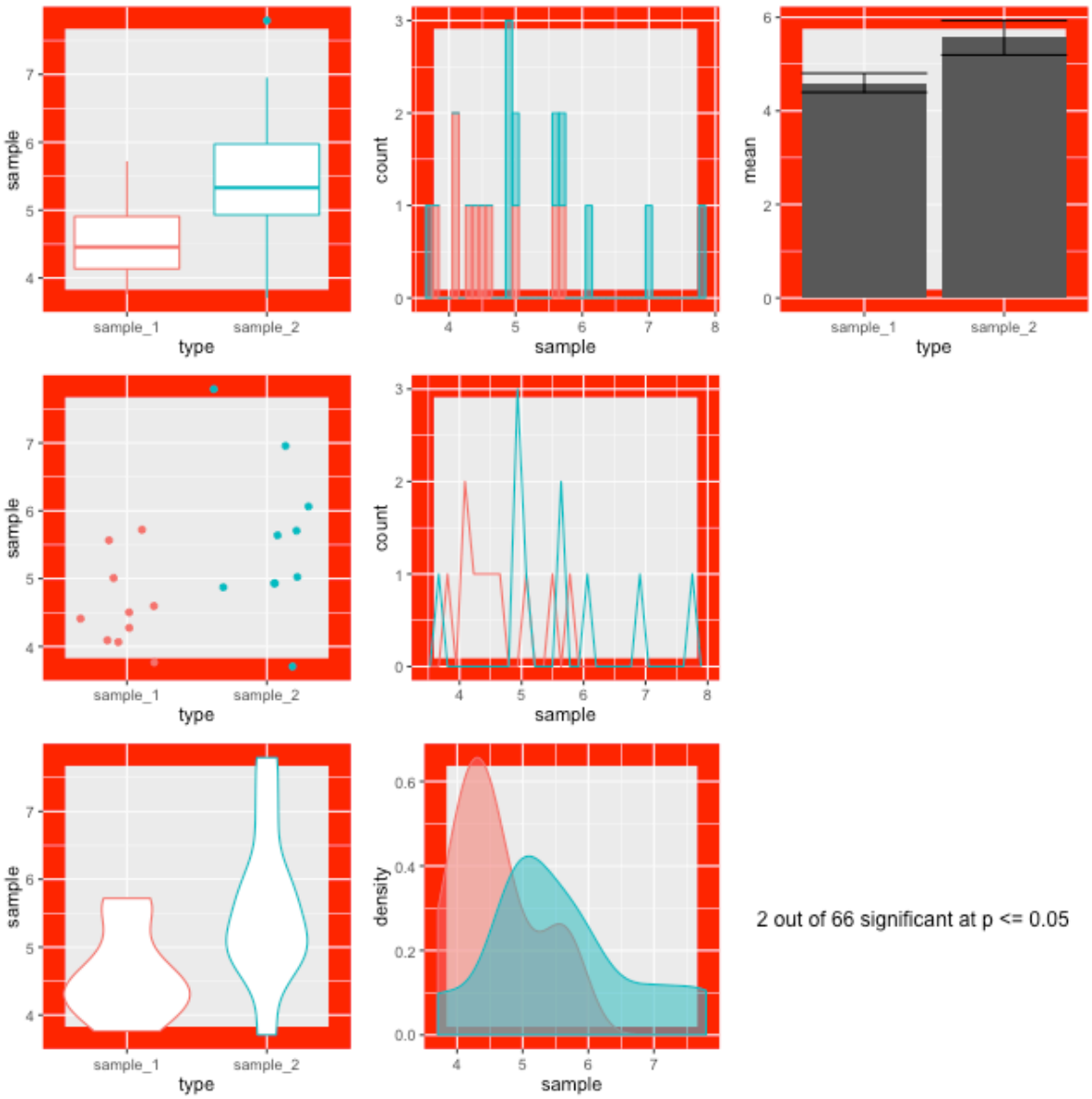
Below is a set of figures that show different views of the same set of data. Every frame of the 100 frames shows a different sampling from the same pool of 10,000 random normally distributed numbers.

Step-by-step, here's how these figures are made.

1. Generate a pool of 10,000 random numbers (mean 5, sd 1)
2. From that, select 10 and call it sample 1.
3. Select another 10, call it sample 2.
4. Draw plots comparing each sample
5. Do an independent *t*-test on the sample 1 and sample 2 to test for significant differences in means.

The figures are plotted with a red border if p comes up less than 0.05. The thing is, the samples are from the same background pool, so intuitively you might suspect that none should be different from the others. The reason that some of them do is because a p value only states that the difference observed occurs by chance in p of all events, so for 100, we'd expect 5 to be marked out by chance. In this run of the experiment we get three. Here's a couple:





Look at the different plots for each. It is observable that for all these the barplots look very convincingly different. But in the context with the other plots its clear that they aren't showing the whole of the story (or in fact much of it). The boxplots (top left) do a good job of showing the range and the violin and density plots (bottom row) do a good job of showing the shape. It is only really the point plot (first column, middle row) that reveals the positions of the data points and shows that the conclusion of the p value is likely skewed by one or two points in each sample. Concluding differences on this basis is *really* unsafe.

Hence, the conclusion from this is that a range of visualisations is necessary to allow us to

have confidence in our p values and understand the shapes of our data. Drawing box plots and sticking to p religiously is going to make us wrong more than we'd like!

ggplot2 An R package for beautiful visualisations

In this tutorial we are going to use *ggplot2* a package in R to make some clear, informative, thorough visualisations that will help us with our analysis. Here's an example of the sort of thing you can get from *ggplot*:

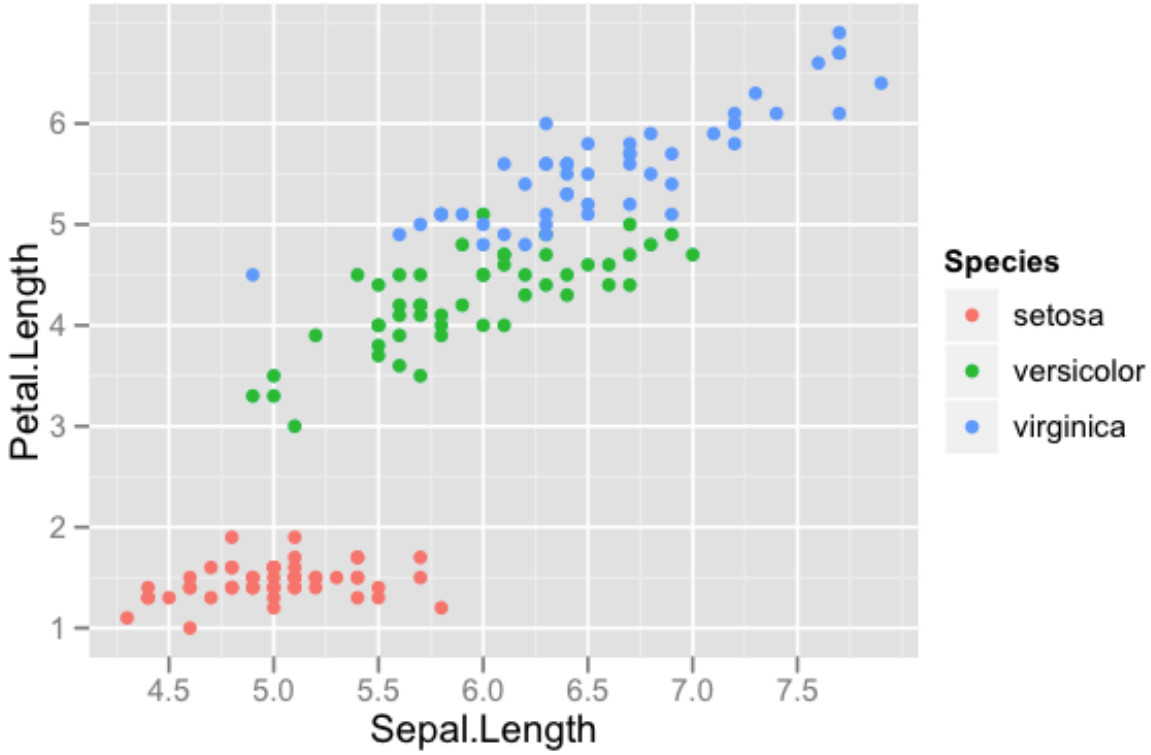


Figure 1: ggplot 2 iris data

ggplot2 is a library in the R statistical programming language - but we won't be learning to program here. The *gg* part stands for 'grammar of graphics', and *ggplot2* is a small grammar that describes plots that should be built on top of data - effectively allowing a user to write their own plot description and have the computer work out what to do, so no programming is needed, just an appreciation of the grammar that is used to describe the plot.

Part I

ggplot2 fundamentals

1 *ggplot2* Tour

1.1 About this chapter

1. Questions:

- How does ggplot2 work?

2. Objectives:

- Explain the structure of a ggplot2
- Explain the flexibilty of the structure

3. Keypoints:

- ggplot2 plots are made in user defined layers
- Using layers helps us to change plot types quickly or build progressively more complex charts

1.2 Building a plot with *ggplot2*

Loading *ggplot2* into memory so we can use it is very easy. With RStudio started, and in the console window type:

```
library(ggplot2)
```

Nothing should happen, thats a good sign!

1.3 It didn't load - I got an error

You need to go back and look at the install instructions, using the packages tab in the bottom right hand window of R studio, click `install` and type `ggplot2` into the window that appears. Select `install` and it should automatically install. If this doesn't work seek some expert help. {: .callout }

1.3.1 Loading the iris test data

R has some datasets built in that allow us to easily develop analysis. Let's look at the `iris` data

```
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
7            4.6         3.4          1.4         0.3     setosa
8            5.0         3.4          1.5         0.2     setosa
9            4.4         2.9          1.4         0.2     setosa
10           4.9         3.1          1.5         0.1     setosa
11           5.4         3.7          1.5         0.2     setosa
12           4.8         3.4          1.6         0.2     setosa
13           4.8         3.0          1.4         0.1     setosa
14           4.3         3.0          1.1         0.1     setosa
15           5.8         4.0          1.2         0.2     setosa
16           5.7         4.4          1.5         0.4     setosa
17           5.4         3.9          1.3         0.4     setosa
18           5.1         3.5          1.4         0.3     setosa
19           5.7         3.8          1.7         0.3     setosa
20           5.1         3.8          1.5         0.3     setosa
21           5.4         3.4          1.7         0.2     setosa
22           5.1         3.7          1.5         0.4     setosa
23           4.6         3.6          1.0         0.2     setosa
24           5.1         3.3          1.7         0.5     setosa
25           4.8         3.4          1.9         0.2     setosa
26           5.0         3.0          1.6         0.2     setosa
27           5.0         3.4          1.6         0.4     setosa
28           5.2         3.5          1.5         0.2     setosa
29           5.2         3.4          1.4         0.2     setosa
30           4.7         3.2          1.6         0.2     setosa
31           4.8         3.1          1.6         0.2     setosa
32           5.4         3.4          1.5         0.4     setosa
33           5.2         4.1          1.5         0.1     setosa
34           5.5         4.2          1.4         0.2     setosa
```

35	4.9	3.1	1.5	0.2	setosa
36	5.0	3.2	1.2	0.2	setosa
37	5.5	3.5	1.3	0.2	setosa
38	4.9	3.6	1.4	0.1	setosa
39	4.4	3.0	1.3	0.2	setosa
40	5.1	3.4	1.5	0.2	setosa
41	5.0	3.5	1.3	0.3	setosa
42	4.5	2.3	1.3	0.3	setosa
43	4.4	3.2	1.3	0.2	setosa
44	5.0	3.5	1.6	0.6	setosa
45	5.1	3.8	1.9	0.4	setosa
46	4.8	3.0	1.4	0.3	setosa
47	5.1	3.8	1.6	0.2	setosa
48	4.6	3.2	1.4	0.2	setosa
49	5.3	3.7	1.5	0.2	setosa
50	5.0	3.3	1.4	0.2	setosa
51	7.0	3.2	4.7	1.4	versicolor
52	6.4	3.2	4.5	1.5	versicolor
53	6.9	3.1	4.9	1.5	versicolor
54	5.5	2.3	4.0	1.3	versicolor
55	6.5	2.8	4.6	1.5	versicolor
56	5.7	2.8	4.5	1.3	versicolor
57	6.3	3.3	4.7	1.6	versicolor
58	4.9	2.4	3.3	1.0	versicolor
59	6.6	2.9	4.6	1.3	versicolor
60	5.2	2.7	3.9	1.4	versicolor
61	5.0	2.0	3.5	1.0	versicolor
62	5.9	3.0	4.2	1.5	versicolor
63	6.0	2.2	4.0	1.0	versicolor
64	6.1	2.9	4.7	1.4	versicolor
65	5.6	2.9	3.6	1.3	versicolor
66	6.7	3.1	4.4	1.4	versicolor
67	5.6	3.0	4.5	1.5	versicolor
68	5.8	2.7	4.1	1.0	versicolor
69	6.2	2.2	4.5	1.5	versicolor
70	5.6	2.5	3.9	1.1	versicolor
71	5.9	3.2	4.8	1.8	versicolor
72	6.1	2.8	4.0	1.3	versicolor
73	6.3	2.5	4.9	1.5	versicolor
74	6.1	2.8	4.7	1.2	versicolor
75	6.4	2.9	4.3	1.3	versicolor
76	6.6	3.0	4.4	1.4	versicolor
77	6.8	2.8	4.8	1.4	versicolor

78	6.7	3.0	5.0	1.7	versicolor
79	6.0	2.9	4.5	1.5	versicolor
80	5.7	2.6	3.5	1.0	versicolor
81	5.5	2.4	3.8	1.1	versicolor
82	5.5	2.4	3.7	1.0	versicolor
83	5.8	2.7	3.9	1.2	versicolor
84	6.0	2.7	5.1	1.6	versicolor
85	5.4	3.0	4.5	1.5	versicolor
86	6.0	3.4	4.5	1.6	versicolor
87	6.7	3.1	4.7	1.5	versicolor
88	6.3	2.3	4.4	1.3	versicolor
89	5.6	3.0	4.1	1.3	versicolor
90	5.5	2.5	4.0	1.3	versicolor
91	5.5	2.6	4.4	1.2	versicolor
92	6.1	3.0	4.6	1.4	versicolor
93	5.8	2.6	4.0	1.2	versicolor
94	5.0	2.3	3.3	1.0	versicolor
95	5.6	2.7	4.2	1.3	versicolor
96	5.7	3.0	4.2	1.2	versicolor
97	5.7	2.9	4.2	1.3	versicolor
98	6.2	2.9	4.3	1.3	versicolor
99	5.1	2.5	3.0	1.1	versicolor
100	5.7	2.8	4.1	1.3	versicolor
101	6.3	3.3	6.0	2.5	virginica
102	5.8	2.7	5.1	1.9	virginica
103	7.1	3.0	5.9	2.1	virginica
104	6.3	2.9	5.6	1.8	virginica
105	6.5	3.0	5.8	2.2	virginica
106	7.6	3.0	6.6	2.1	virginica
107	4.9	2.5	4.5	1.7	virginica
108	7.3	2.9	6.3	1.8	virginica
109	6.7	2.5	5.8	1.8	virginica
110	7.2	3.6	6.1	2.5	virginica
111	6.5	3.2	5.1	2.0	virginica
112	6.4	2.7	5.3	1.9	virginica
113	6.8	3.0	5.5	2.1	virginica
114	5.7	2.5	5.0	2.0	virginica
115	5.8	2.8	5.1	2.4	virginica
116	6.4	3.2	5.3	2.3	virginica
117	6.5	3.0	5.5	1.8	virginica
118	7.7	3.8	6.7	2.2	virginica
119	7.7	2.6	6.9	2.3	virginica
120	6.0	2.2	5.0	1.5	virginica

```
121          6.9         3.2          5.7         2.3  virginica
122          5.6         2.8          4.9         2.0  virginica
123          7.7         2.8          6.7         2.0  virginica
124          6.3         2.7          4.9         1.8  virginica
125          6.7         3.3          5.7         2.1  virginica
126          7.2         3.2          6.0         1.8  virginica
127          6.2         2.8          4.8         1.8  virginica
128          6.1         3.0          4.9         1.8  virginica
129          6.4         2.8          5.6         2.1  virginica
130          7.2         3.0          5.8         1.6  virginica
131          7.4         2.8          6.1         1.9  virginica
132          7.9         3.8          6.4         2.0  virginica
133          6.4         2.8          5.6         2.2  virginica
134          6.3         2.8          5.1         1.5  virginica
135          6.1         2.6          5.6         1.4  virginica
136          7.7         3.0          6.1         2.3  virginica
137          6.3         3.4          5.6         2.4  virginica
138          6.4         3.1          5.5         1.8  virginica
139          6.0         3.0          4.8         1.8  virginica
140          6.9         3.1          5.4         2.1  virginica
141          6.7         3.1          5.6         2.4  virginica
142          6.9         3.1          5.1         2.3  virginica
143          5.8         2.7          5.1         1.9  virginica
144          6.8         3.2          5.9         2.3  virginica
145          6.7         3.3          5.7         2.5  virginica
146          6.7         3.0          5.2         2.3  virginica
147          6.3         2.5          5.0         1.9  virginica
148          6.5         3.0          5.2         2.0  virginica
149          6.2         3.4          5.4         2.3  virginica
150          5.9         3.0          5.1         1.8  virginica
```

R just printed the whole thing to screen and we end up looking at just the bottom end of it. Let's look at just the top.

```
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
```

We can see that we have the top six rows and we can see that the data is a list of measurments of the sepals and petals for some species of iris. Let's get a summary of the data set:

```
summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

Alright, that's quite clear, some summary values for each numeric column and note how R has calculated the number of rows of each distinct label for the text column.

1.3.2 A first plot

ggplot2 plots are built up of layers, the foundation layer is the data layer, thats the whole data set containing the bits we would want to plot. We define that with the `ggplot2` command.

```
library(ggplot2)
ggplot(data=iris)
```

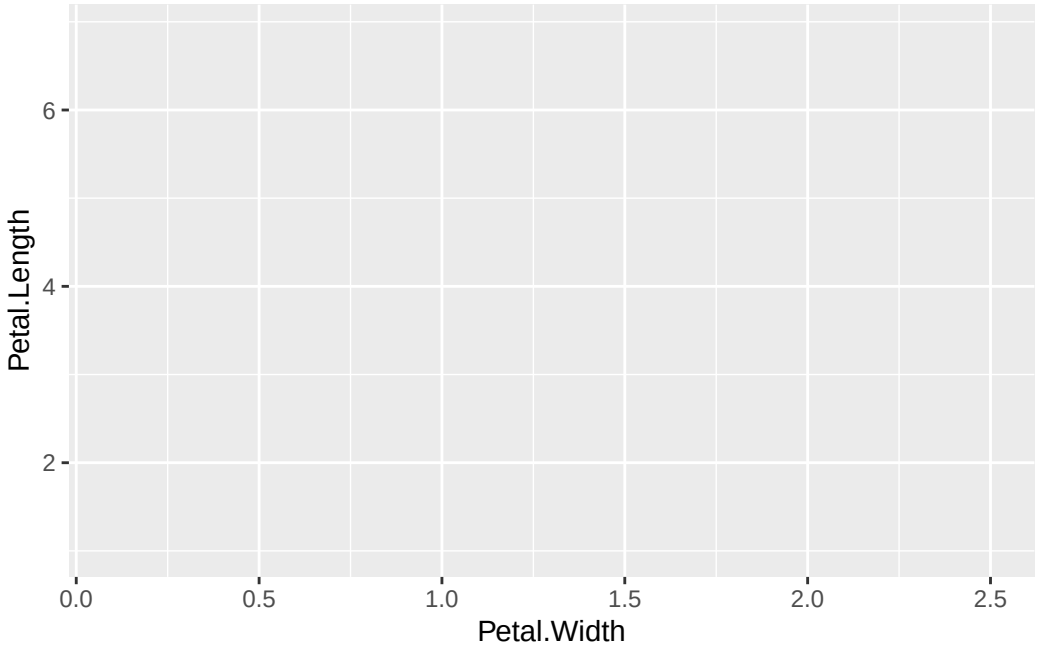
Nothing happened, you got a blank screen in the plot window to the right. That's because a data layer alone doesn't tell us what, or how to plot. It's just the source of the numbers we'll use.

The next thing we need is an `aesthetic` layer. This is basically the things to look at, and includes:

1. x and y axes (sometimes called position)
2. colour (the line colour of a thing)
3. fill (the block colour of a thing)
4. shape (e.g of points)
5. line type
6. size (e.g of points)

Let's decide to look at petal width and length. We use the `aes()` function for the aesthetic and we can add layers together with the `+` operator.

```
ggplot(data=iris) + aes(x=Petal.Width, y=Petal.Length)
```

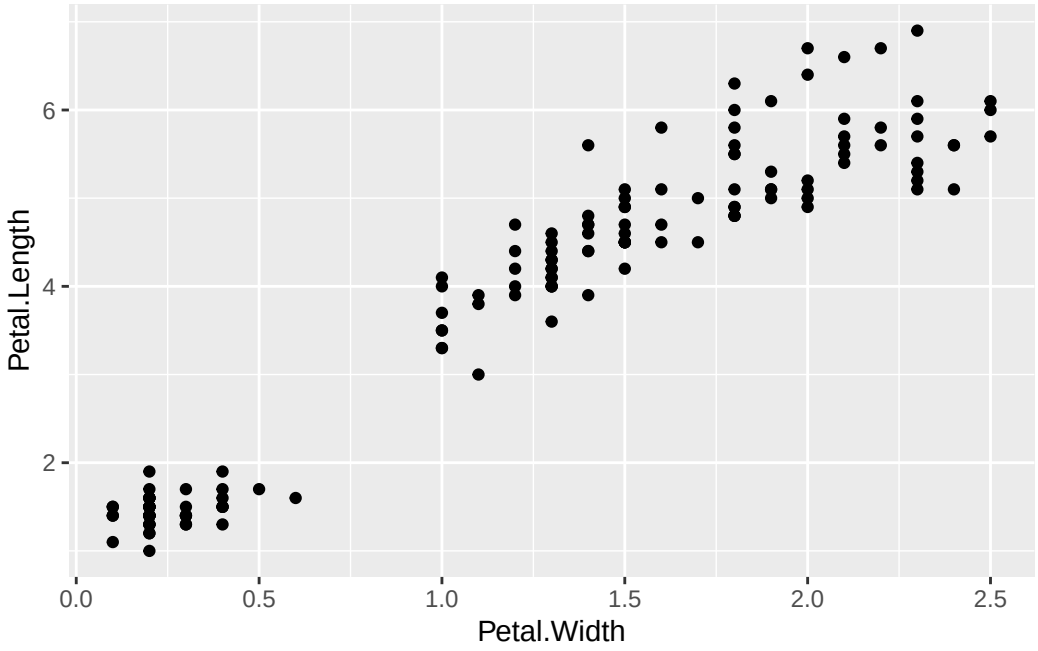


This time we get axes in the plot window. *ggplot* now knows the data source and the things that should be displayed on the axis, but it doesn't fully know *how* to display them. That is done in the `geom` (or geometric objects) layer. There are loads of geoms e.g

```
1. `geom_point()` for scatter plots
2. `geom_line()` for trend lines
3. `geom_boxplot()` for boxplots!
```

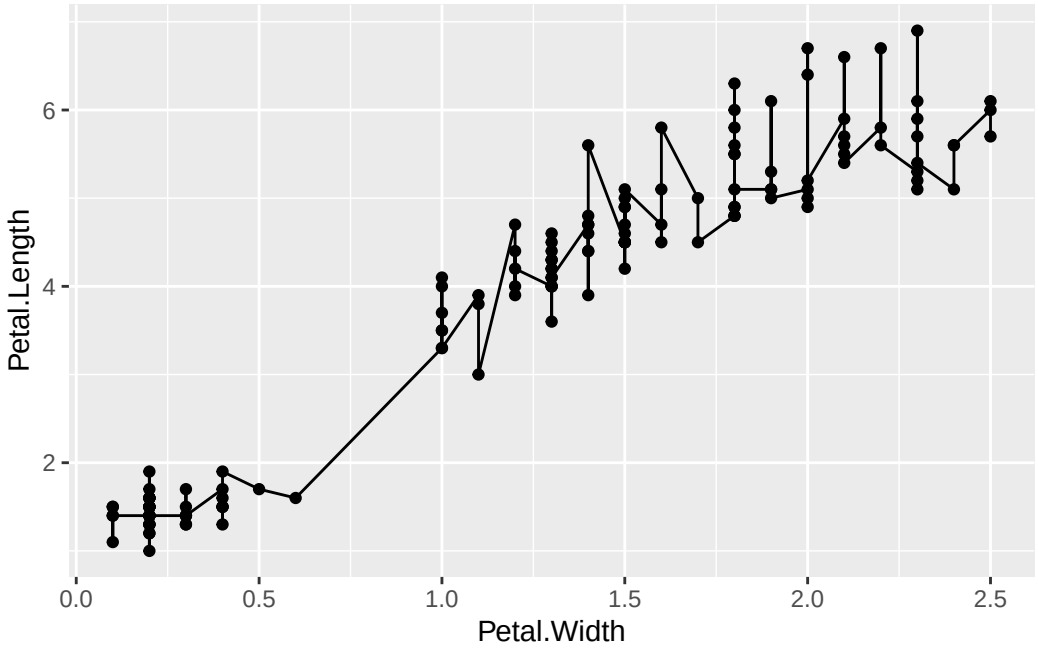
Let's add a geom layer.

```
ggplot(data=iris) + aes(x=Petal.Width, y=Petal.Length) + geom_point()
```



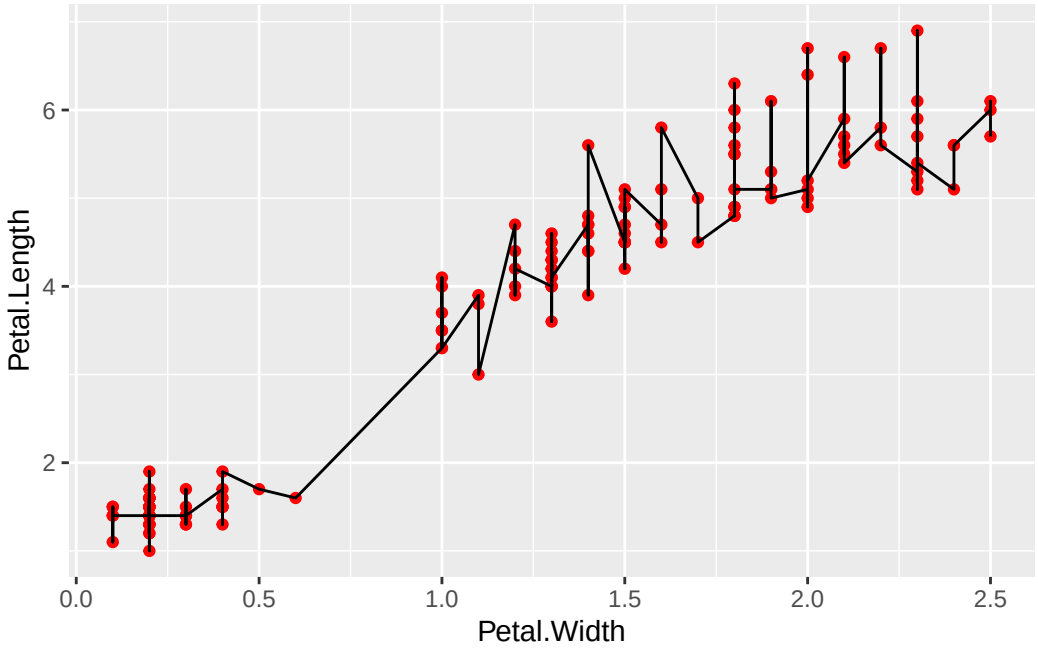
Now we see the whole plot. The data has been mapped onto the right axes and the geometric objects on top of that. Let's go crazy and add more layers.

```
ggplot(data=iris) + aes(x=Petal.Width, y=Petal.Length) + geom_point() + geom_line()
```



You can see the new geom just adds straight on top of the old one. By default, `geom_line()` is a simple join the dots sort of line, so it looks really squiggly. Different layers can have their own options set, e.g the points can be coloured.

```
ggplot(data=iris) + aes(x=Petal.Width, y=Petal.Length) + geom_point(colour="Red") + geom_l
```



1.4 Making and saving a base plot

There is actually no need to go round typing in the whole command above repetively all the time. *ggplot2* layers can be saved to R variables like this:

```
p <- ggplot(data=iris) + aes(x=Petal.Width, y=Petal.Length)
```

and the bits we want to add or change stuck on top:

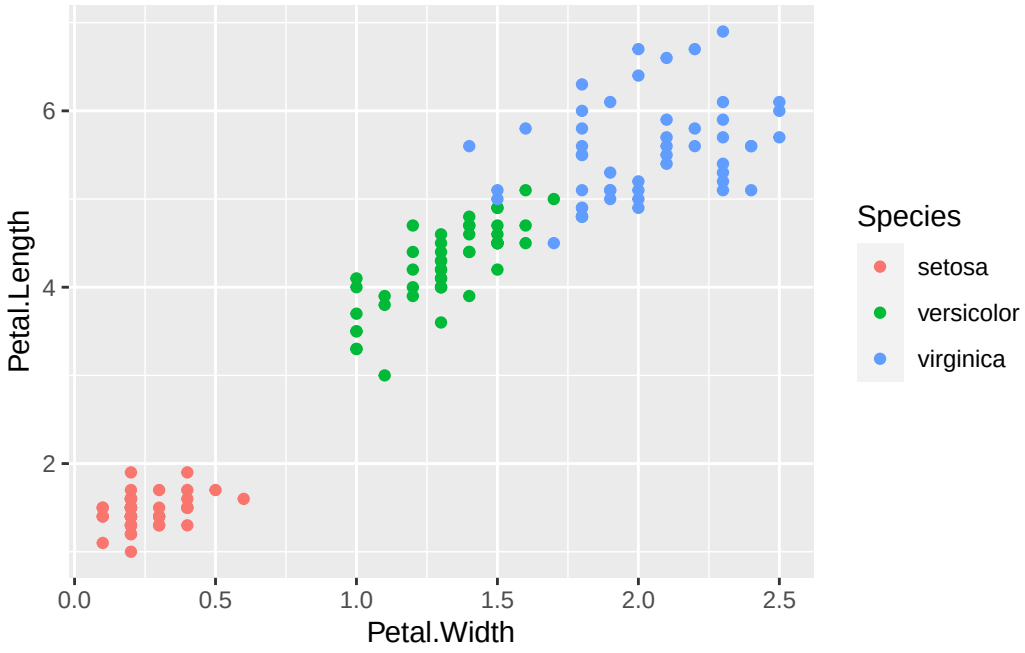
```
p + geom_point()
```

1.5 Mappings versus assignment

The power of *ggplot* to 'just do the right thing' comes from its use of mappings, these can be thought of as rules for what to do when it meets a bit of data in a particular place.

Above we set the colour, `geom_point()` to `"Red"`. This set all the points to red, it was an assignment, since ggplot didn't have anything to work out, every point is just red. By setting the colour to a column in the data we can make ggplot work colours out for us dependent on the information in that column. Try:

```
p <- ggplot(data=iris) + aes(x=Petal.Width, y=Petal.Length)
p + geom_point(aes(colour=Species))
```

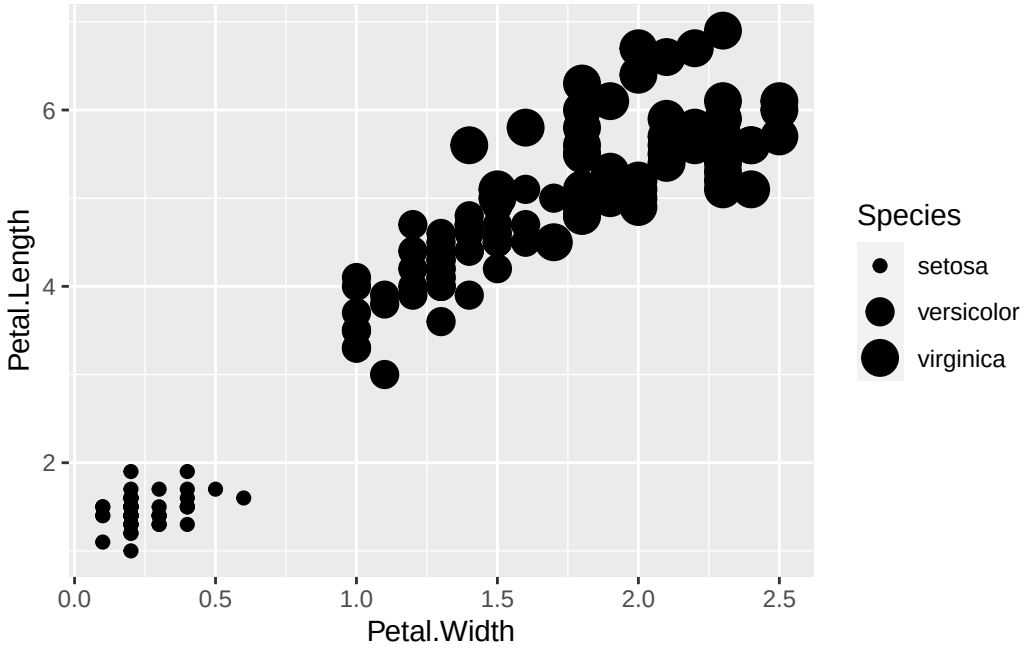


This time we told ggplot to use the value of the Species column to colour each data point, ggplot decided on a mapping for a list of colours to each different value in the Species column and drew that on the plot for us. Only aesthetics can be mappings, so we had to use an `aes()` function inside the geom.

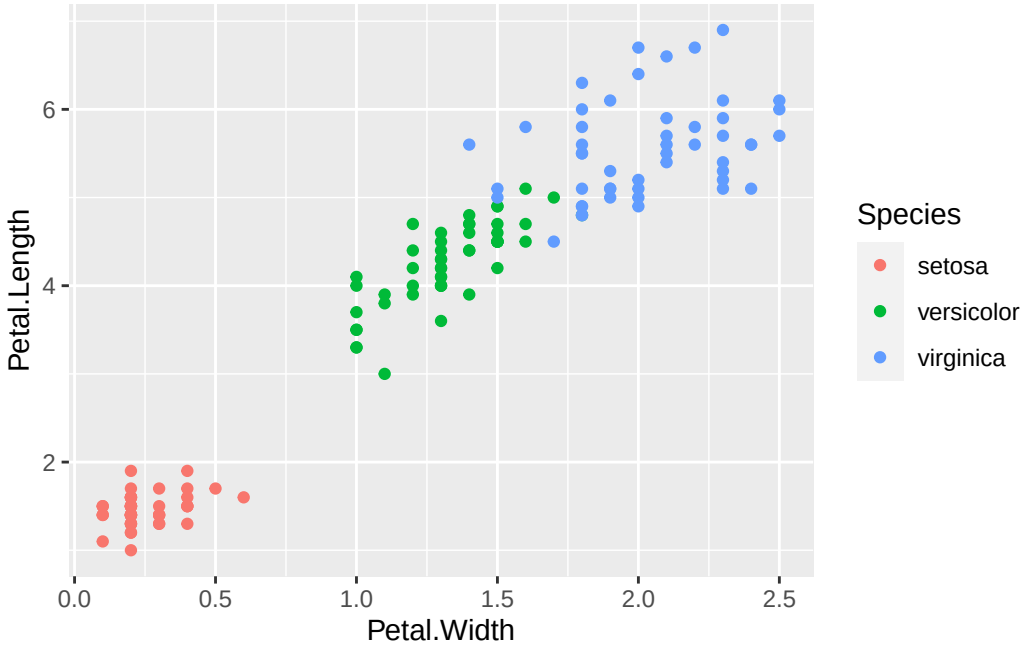
Lots of aesthetic features can be mapped to data, try `size` and `shape`, and try mixing them.

```
p + geom_point(aes(size=Species))
```

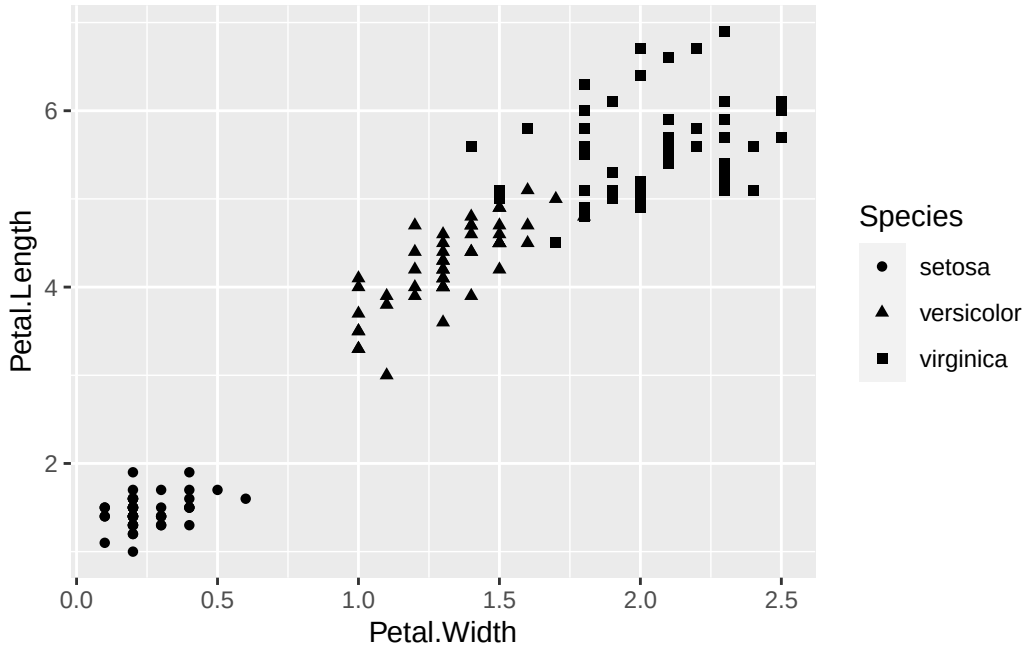
```
Warning: Using size for a discrete variable is not advised.
```




```
p + geom_point(aes(colour=Species))
```

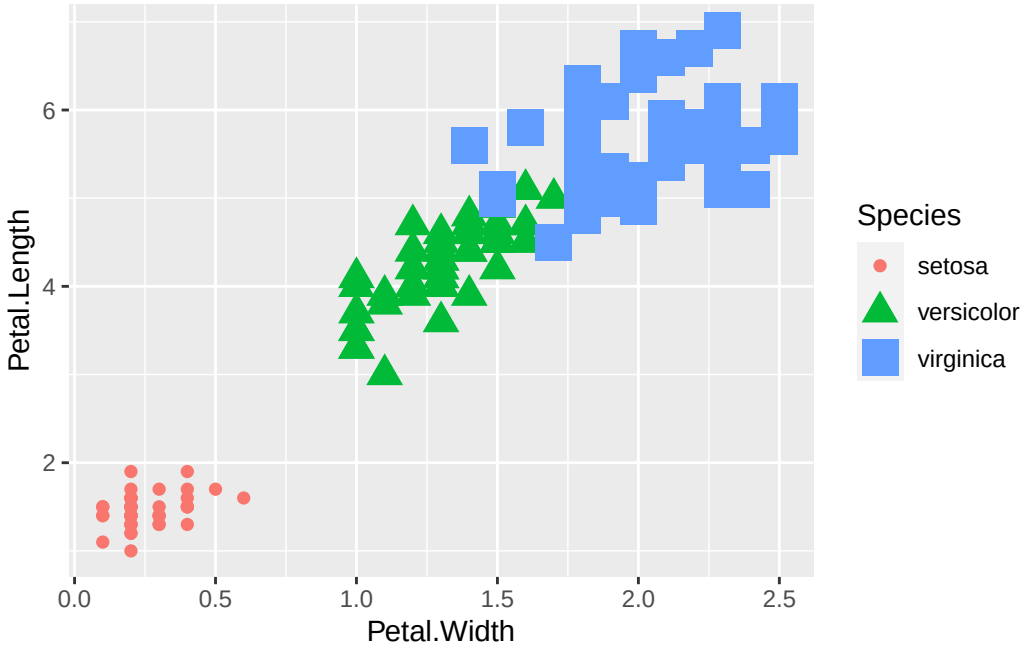


```
p + geom_point(aes(shape=Species))
```




```
p + geom_point(aes(size=Species,colour=Species,shape=Species))
```

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



1.6 Quiz

1. Use the docs at http://docs.ggplot2.org/current/ to examine the geoms that are available. Try `geom_jitter()`, why choose this over `geom_point()`?
2. Use this base plot `p <- ggplot(data=iris) + aes(x=Petal.Width, y=Petal.Length)`
3. What happens if you map a continuous variable to an aesthetic like colour? EG `aes(color=Petal.Width)`
4. Try combining `geom_smooth()` with `geom_jitter()`
5. Why doesn't `geom_boxplot()` work with a continuous varialbe like Petal.Width? (Hint: you need to think about the difference between categorical or discrete and continuous data).
6. How could you make `geom_boxplot()` show you box plots for the three species Petal.Width. (Hint: you need to think about the aesthetic and where you set it.

Part II

Making a data appropriate plot

2 Common Geoms

2.1 About this chapter

1. Questions:

- What sorts of plot can I do?

2. Objectives:

- Demonstrate the main types of plot

3. Keypoints:

- There are geoms for continuous and discrete data
- Selecting and mixing these properly can give a nice representation of your data

2.2 Continuous geoms

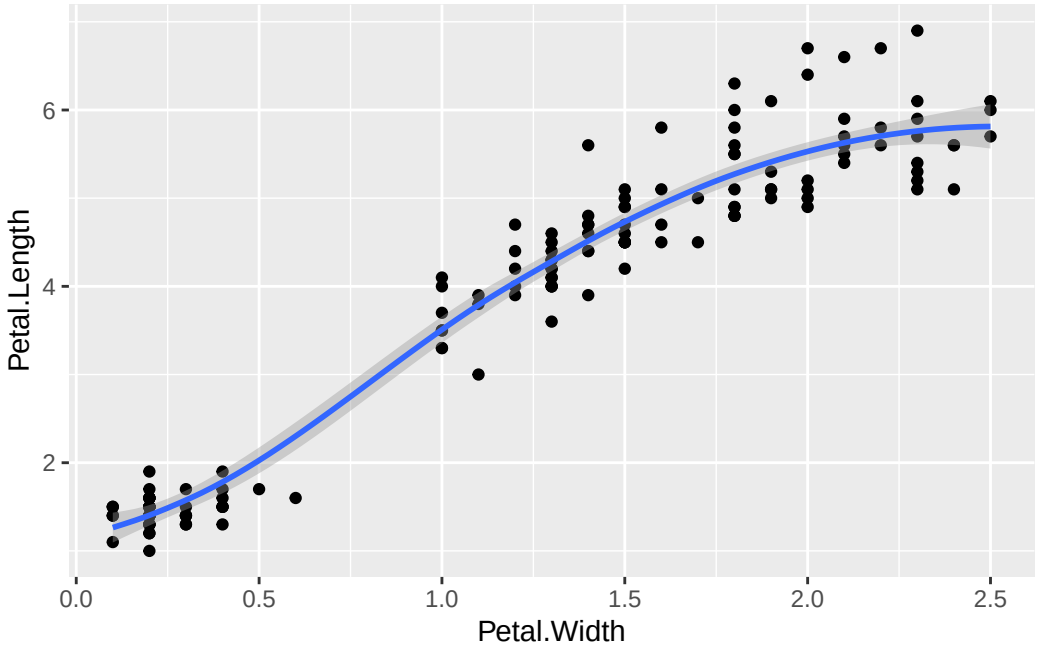
Let's look at some geoms that use continuous data on the x and y axis.

2.2.1 geom_smooth()

The built in geom `geom_smooth()` is a great one for getting a nice summary line through the data

```
p <- ggplot(iris) + aes(Petal.Width,Petal.Length) + geom_point()
p + geom_smooth()
```

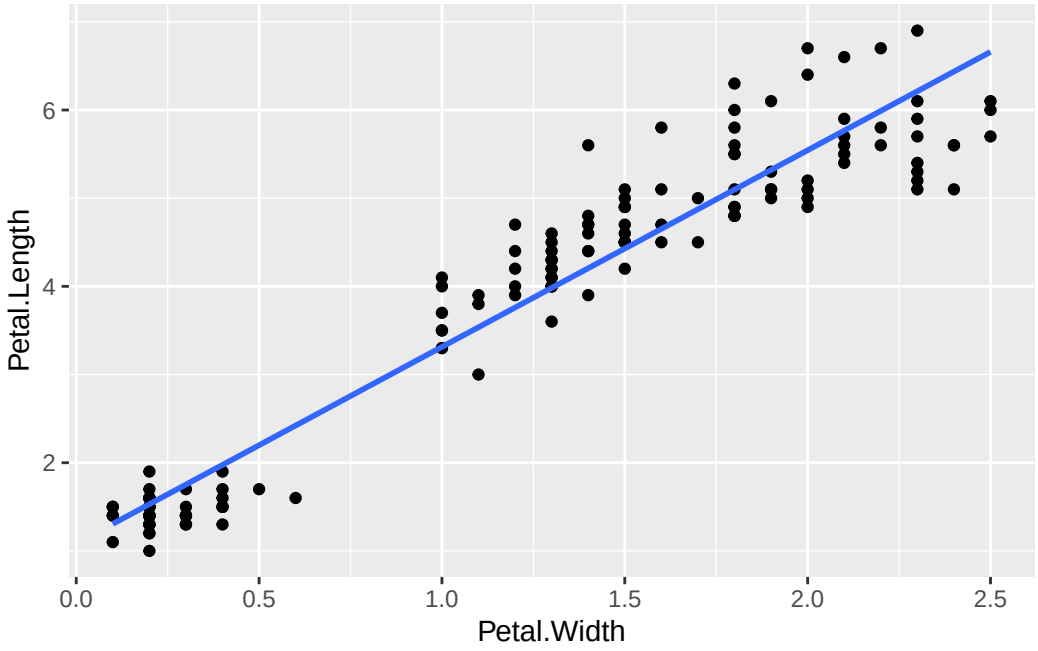
```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



By default, this isn't a simple line of best fit, as you can see the smoothed line has curves! And it has a grey region that shows the standard error of the line. To get the standard line of the form `y=mx+c`, use

```
p + geom_smooth(method = "lm", se = FALSE)
```

```
`geom_smooth()` using formula 'y ~ x'
```



2.2.2 What's the r2?

Having shown you how to put the line of best fit on the graph, you probably want to know how to get the equation and r2 value. That takes a little bit of pure R. Here's how, using the `lm` linear model function. The syntax for this is `lm(y ~ x, dataset)` so for the iris data and the graph we just made (note the order Y and X is used in not the order X and Y)

```
model <- lm(Petal.Length ~ Petal.Width, iris)
```

The result is now saved in the `model` variable we just created. This is a complex R object, which we can see a summary of using

```
summary(model)
```

```
Call:
lm(formula = Petal.Length ~ Petal.Width, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-1.33542 -0.30347 -0.02955  0.25776  1.39453
```

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.08356    0.07297   14.85   <2e-16 ***
Petal.Width  2.22994    0.05140   43.39   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4782 on 148 degrees of freedom
Multiple R-squared:  0.9271,    Adjusted R-squared:  0.9266
F-statistic:  1882 on 1 and 148 DF,  p-value: < 2.2e-16
```

This is complex, but we want model coefficients, that is the `m` value - the slope (here 2.22994) and the `c` value - the intercept (here 1.08356), and the adjusted R-squared (0.9266)

2.3 Shorthand notation

A shorthand in ggplot allows you to leave out the `data=` part of the function call, if you put the data in the first position so

`ggplot(iris)` is the same as `ggplot(data=iris)`

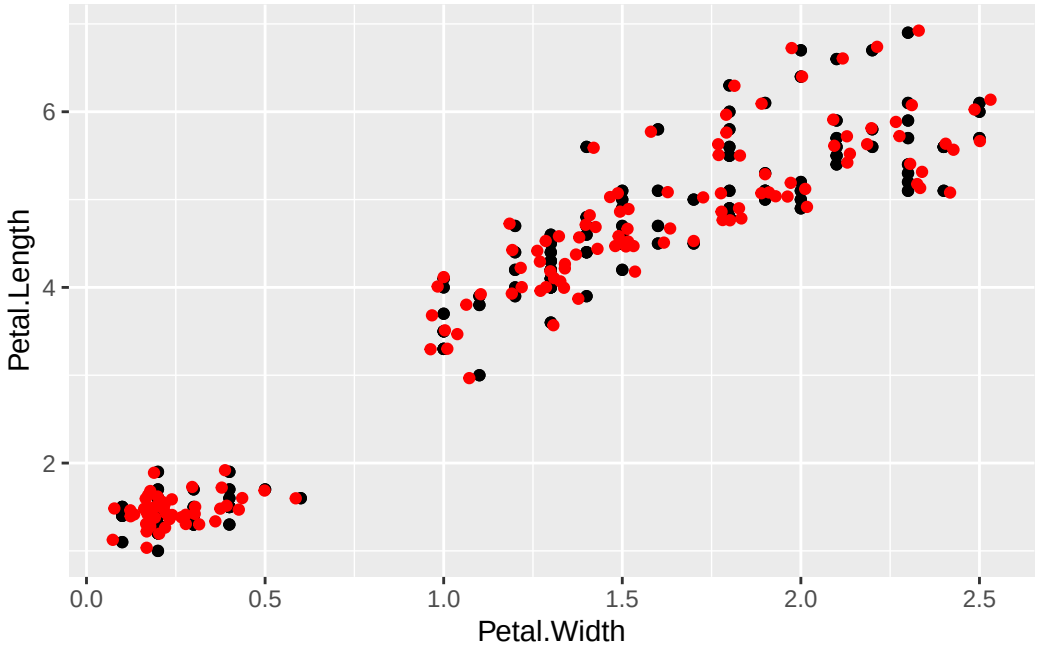
In the `aes()` function we can do the same. We can leave out the `x=` and `y=` parts and instead use the first two things in the function call for the x and y axis.

So `aes(Petal.Length, Petal.Width)` is the same as `aes(x=Petal.Length, y=Petal.Width)`

2.3.1 geom_jitter()

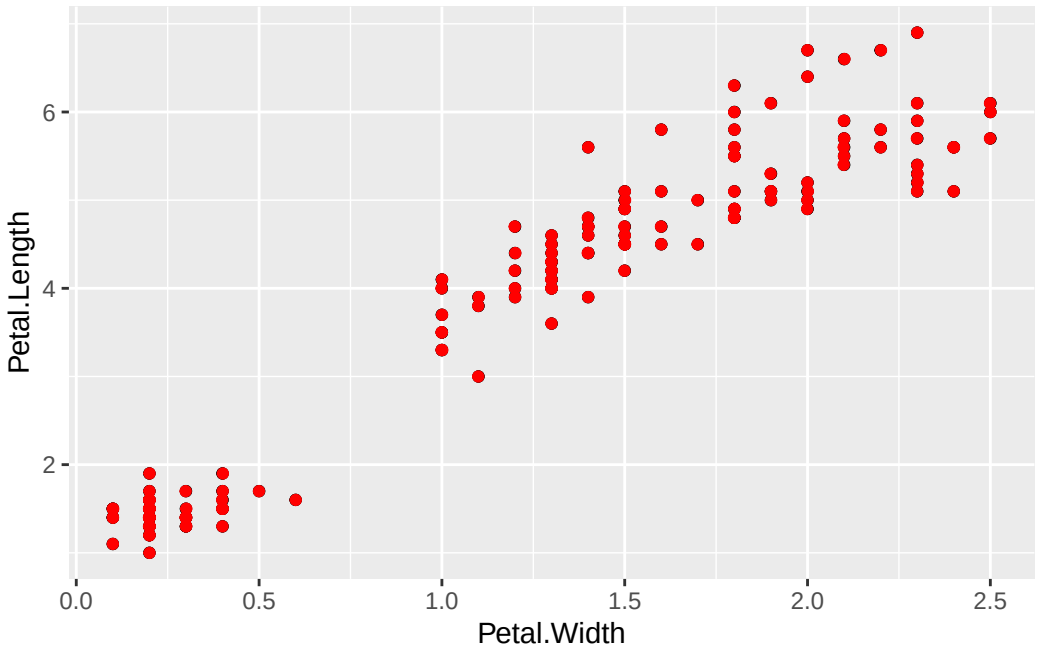
Sometimes, point plots get crowded, the points can get too close together, a visual problem called overplotting. A jitter plot lets us get over this by adding a random bit of noise to the position of the points. Here the points from the jitter geom are set to red.

```
p + geom_point() + geom_jitter(colour="Red")
```



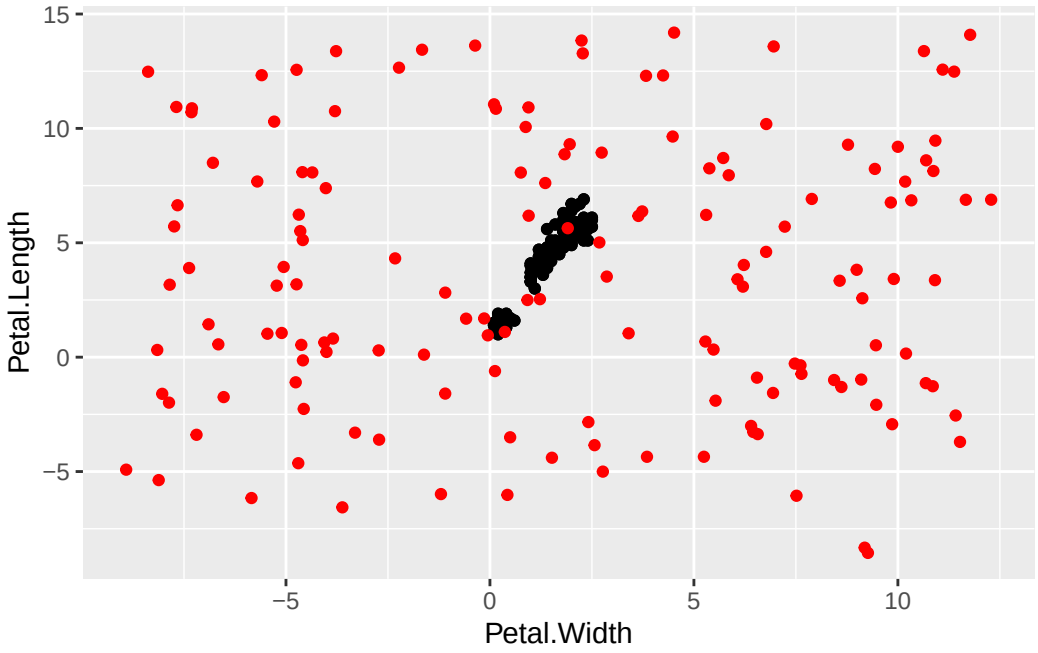
We can fiddle with the range of the jitter with `width` and `height` options

```
p + geom_point() + geom_jitter(colour="Red", width=0.001, height=0.001)
```



conversely,

```
p + geom_point() + geom_jitter(colour="Red", width=10, height=10)
```

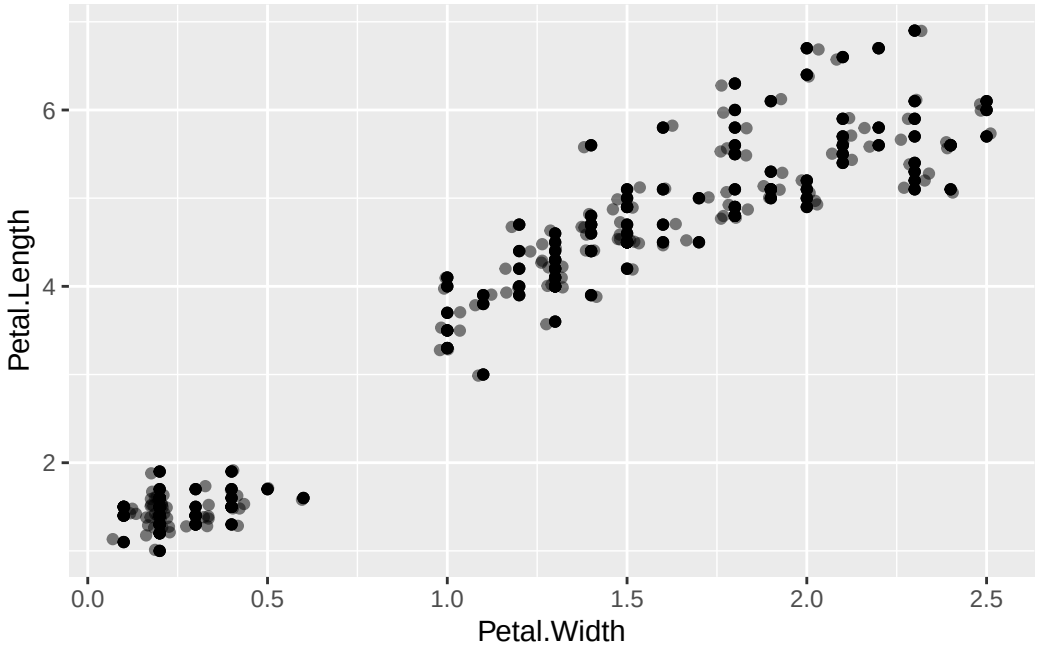


The defaults are usually a good choice though.

2.3.2 Changing opacity

Overplotting can be dealt with in other ways, changing the opacity of the geom is another. This is the `alpha` option. Choose the value in the range `0` to `1`, where `0` is invisible and `1` is solid

```
p + geom_point() + geom_jitter(alpha=0.5)
```

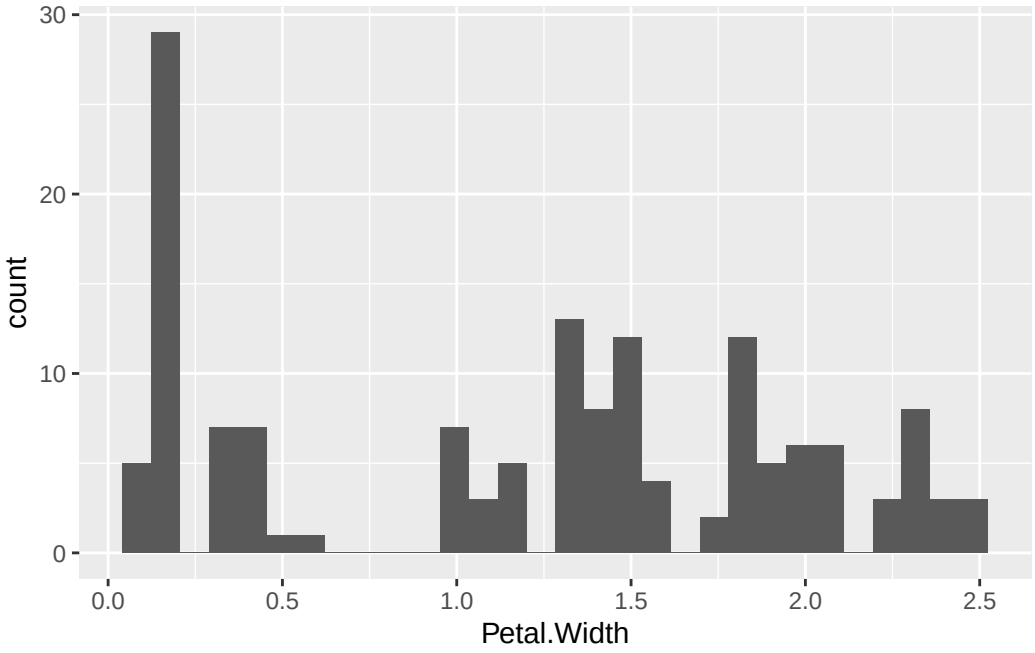


2.3.3 geom_histogram()

Plotting a histogram is done with the `geom_histogram()`. The y value for this is calculated automatically, so you provide the x value.

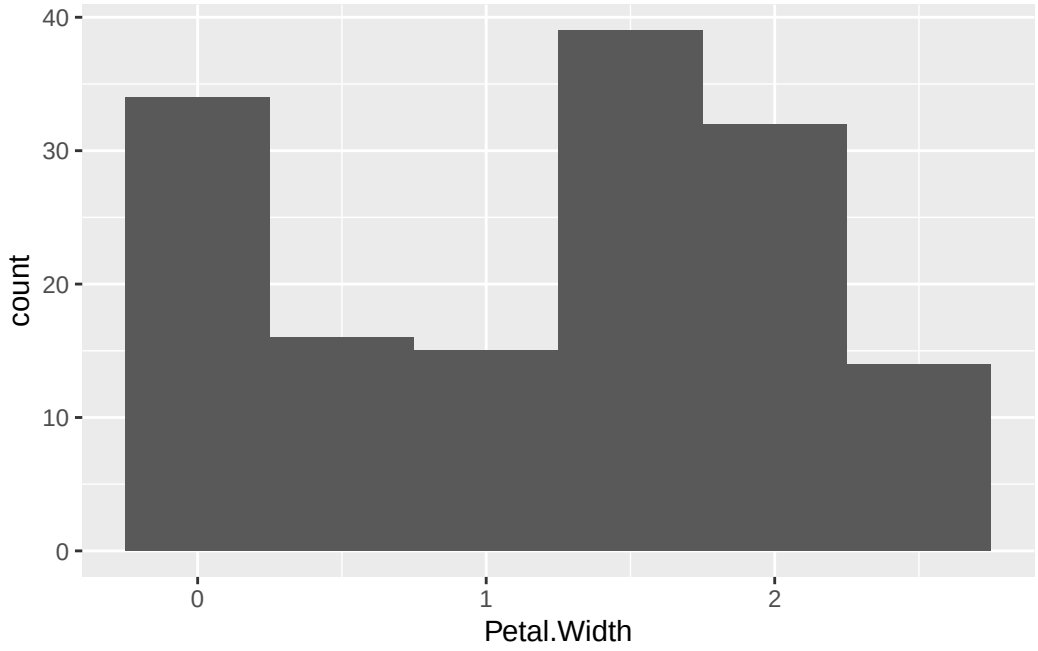
```
p <- ggplot(iris) + aes(Petal.Width)
p + geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

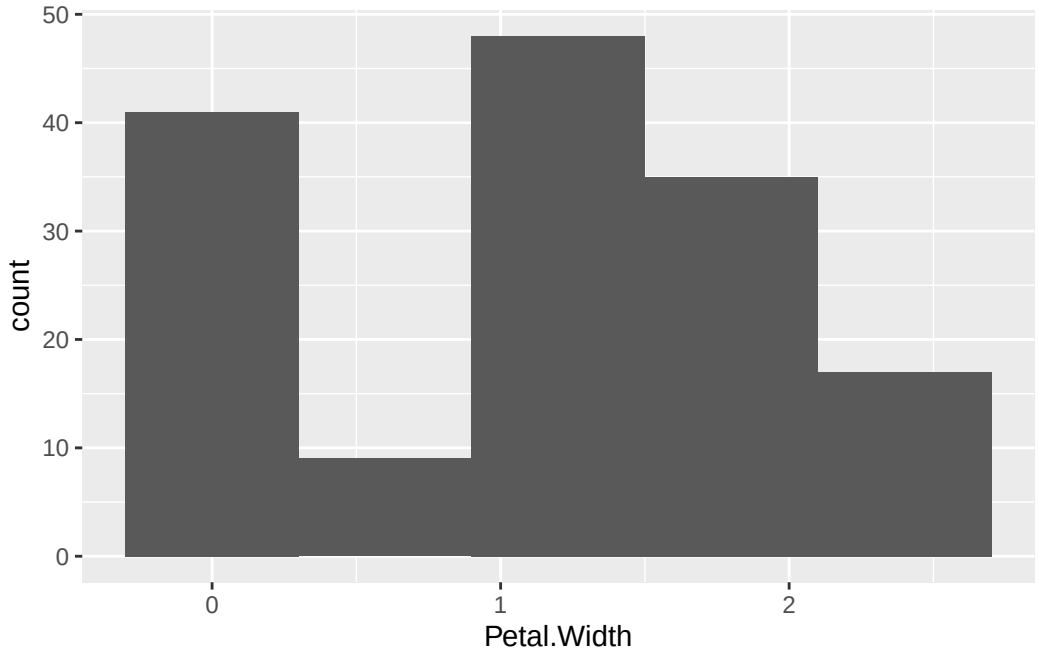


You can change the width of the bins with `binwidth`, or set the number of bins with `bins`

```
p + geom_histogram(binwidth=0.5)
```

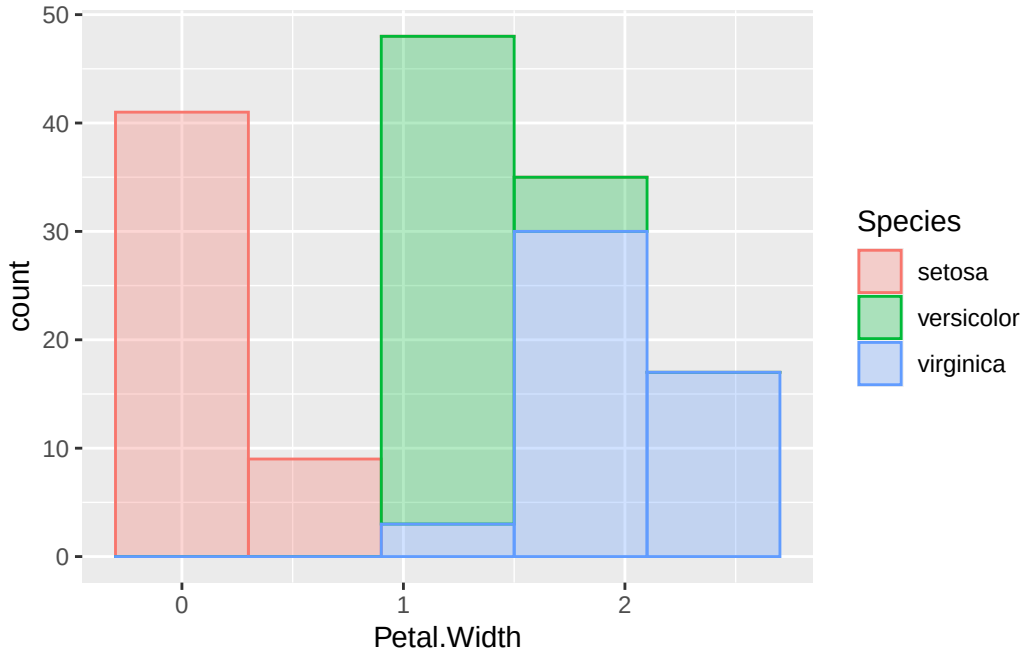



```
p + geom_histogram(bins=5)
```



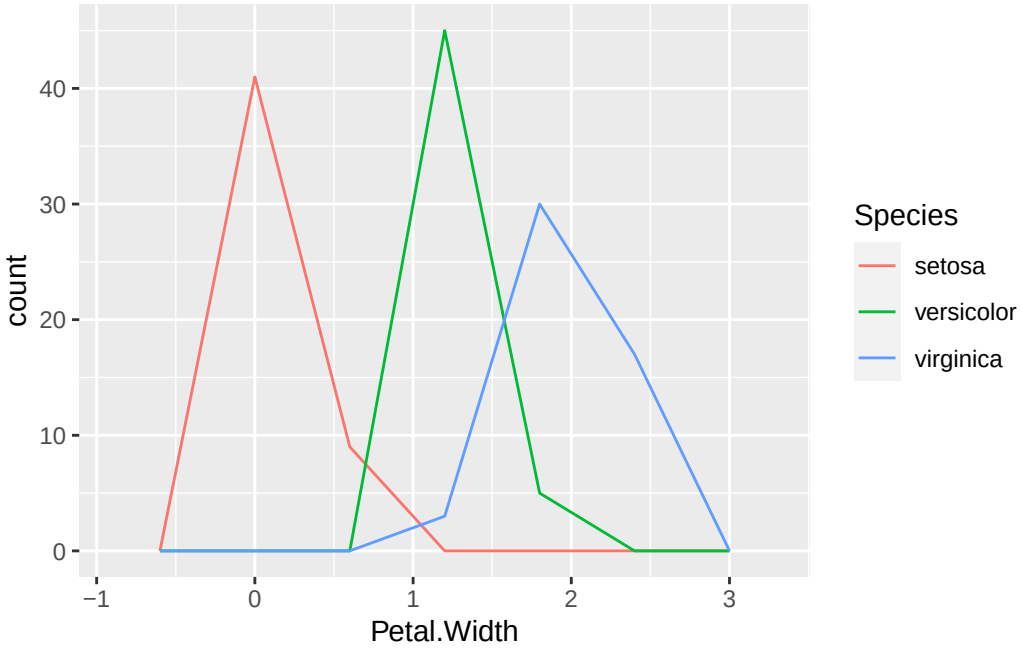
Trying to map the species to colour in this one gives us a weird sort of stacked histogram.

```
p + geom_histogram(bins=5, aes(colour=Species, fill=Species),alpha=0.3 )
```



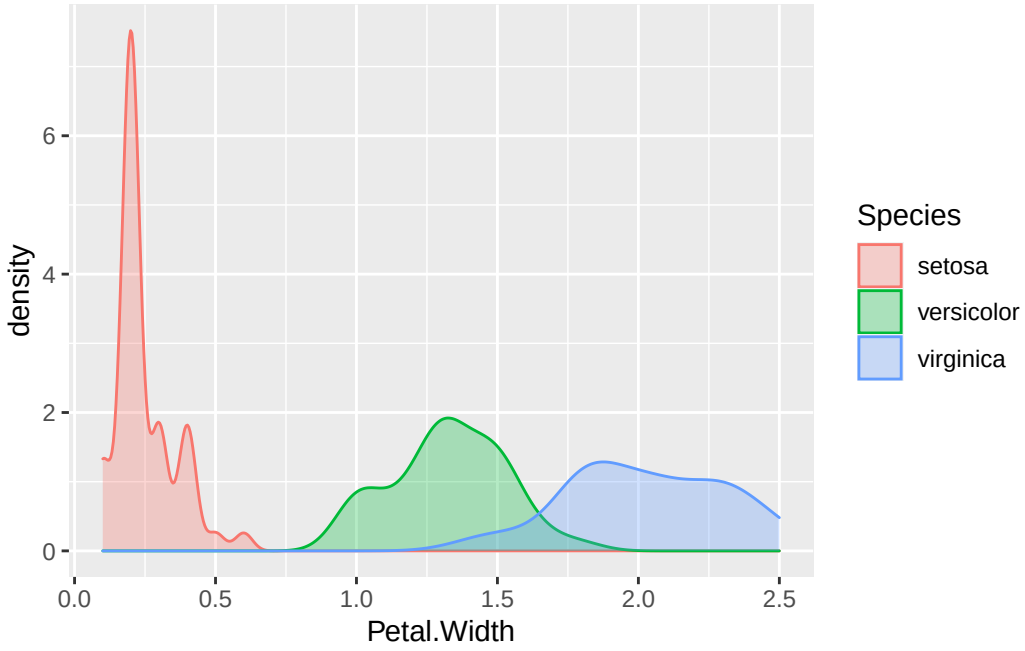
We can avoid this in a few ways, one is by using `geom_freqpoly()`, which is a line graph joining the tops of the bars of the histogram.

```
p + geom_freqpoly( aes(colour=Species), bins=5 )
```



or with `geom_density()` which gives us smoothed lines from a kernel density estimate of the data (which is a way of generating a smooth curve over histograms).

```
p + geom_density( aes(colour=Species, fill=Species), alpha=0.3)
```



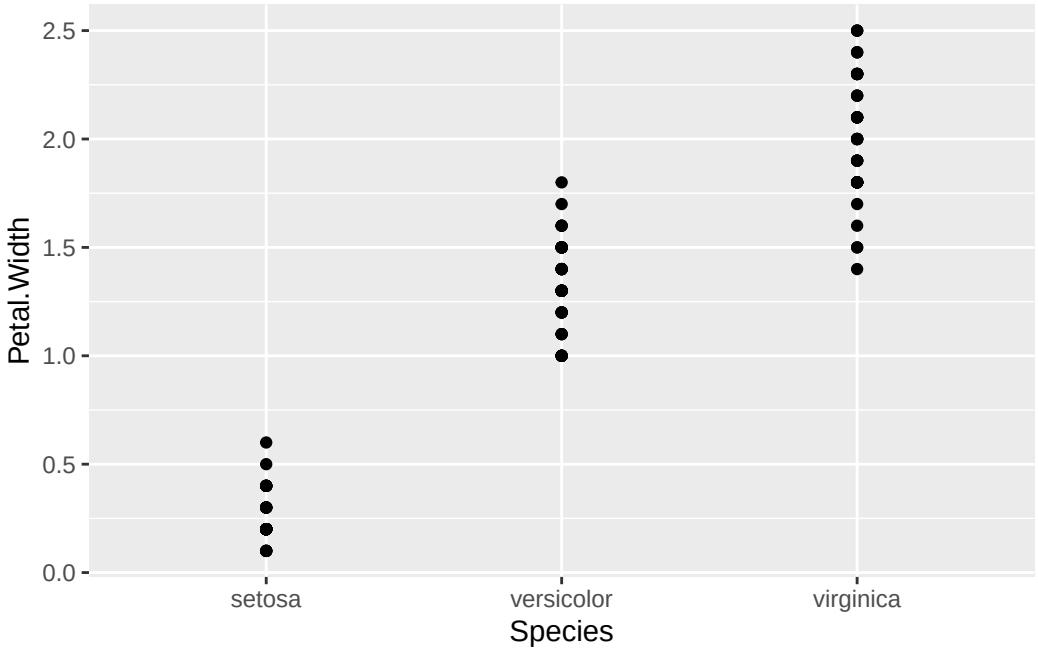
2.4 Discrete geoms

Let's look at some geoms with categories on the x and numbers on the y axis.

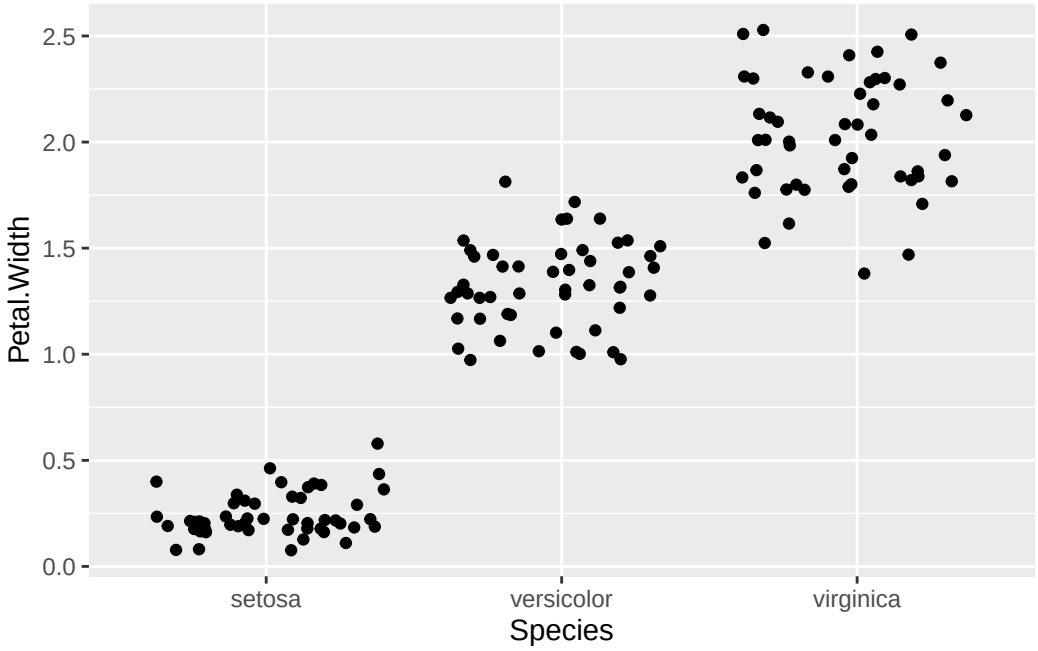
2.4.1 geom_point() and geom_jitter()

Both these geoms can be used with categoric data in one dimension. This is a useful and very honest way of showing your data points.

```
p <- ggplot(iris) + aes(x=Species, y=Petal.Width)
p + geom_point()
```



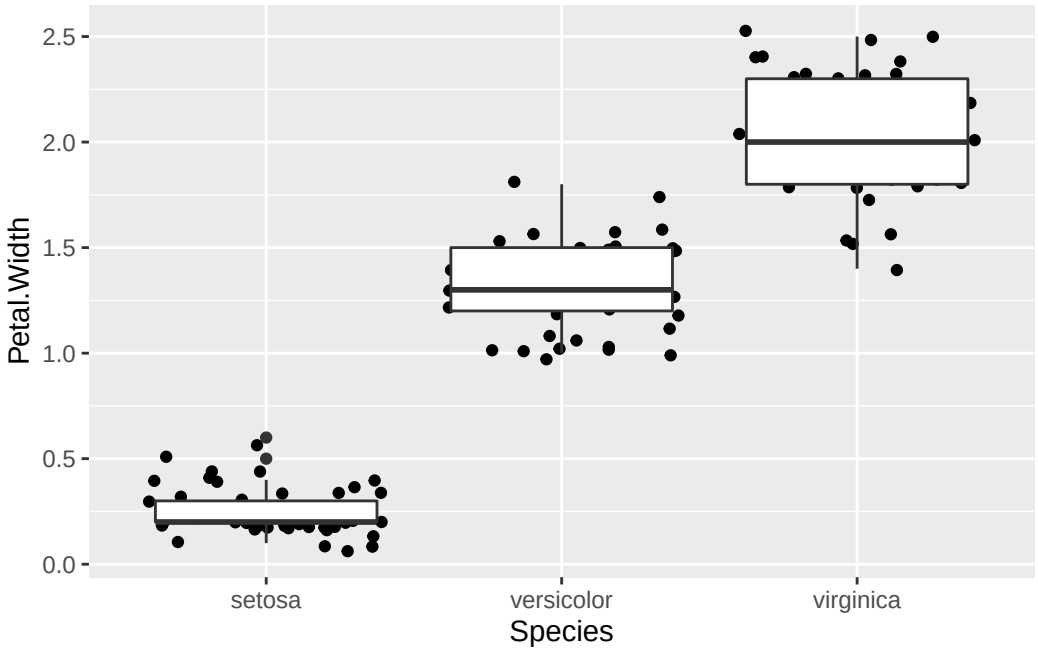

```
p + geom_jitter()
```



2.4.2 geom_boxplot() and geom_violin()

A great way to summarise the distributions of points is to use a boxplot in conjunction with the dots.

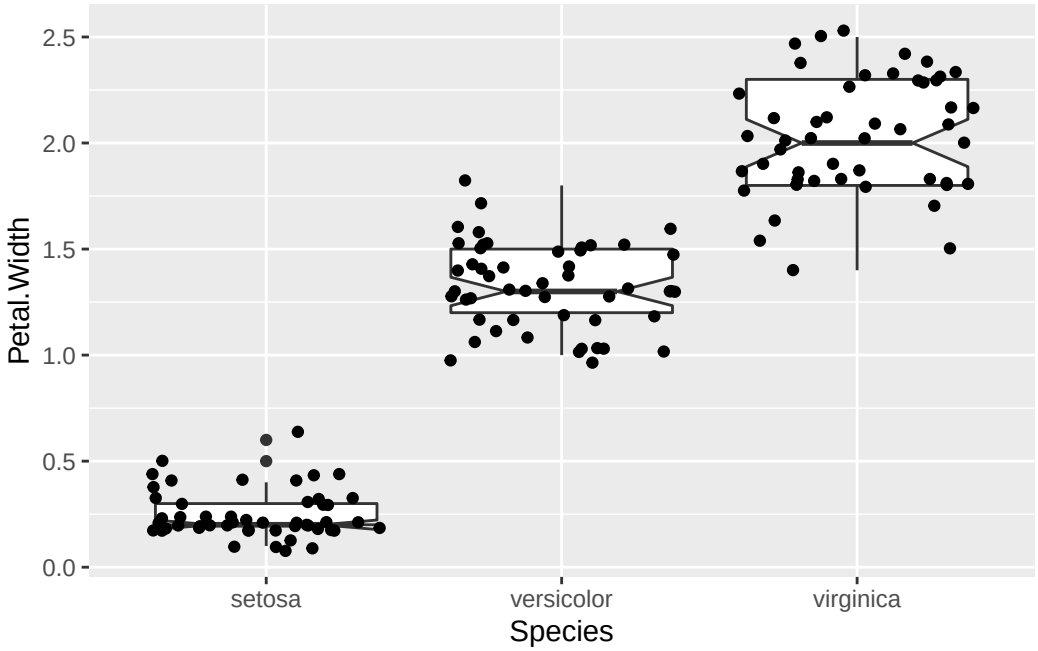
```
p <- ggplot(iris) + aes(x=Species, y=Petal.Width)
p + geom_jitter() + geom_boxplot()
```



Which unhelpfully puts the newest layer on top. Reverse the order to see the points

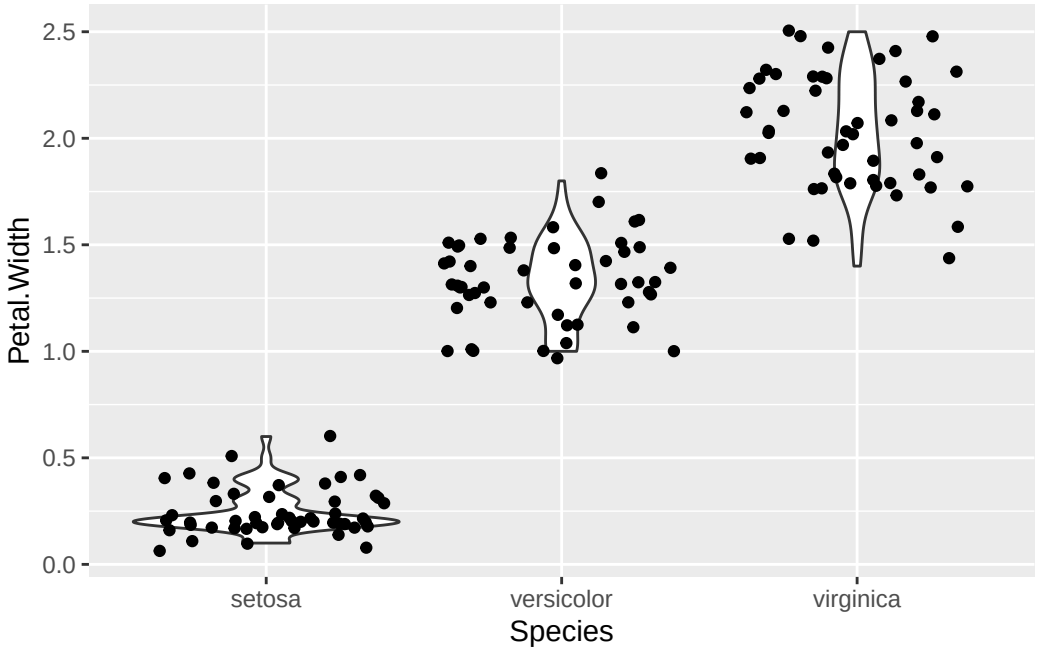
```
p + geom_boxplot(notch=TRUE) + geom_jitter()
```

```
notch went outside hinges. Try setting notch=FALSE.
```



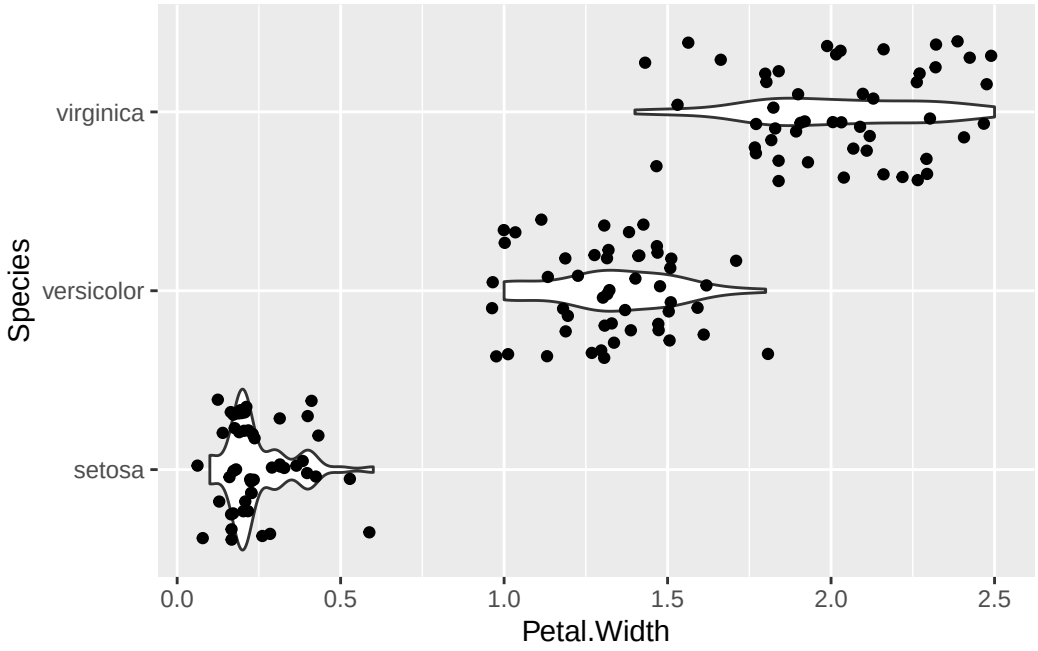
A common reason for using the boxplot is to use those notches to show the significant differences in the data. But really, these only help you assess a difference meaningfully if the data are normally distributed. In other circumstances you should be aware that the notches are misleading. Instead you can see the spread of your data much better with a violin plot.

```
p <- ggplot(iris) + aes(x=Species, y=Petal.Width)
p + geom_violin() + geom_jitter()
```



By turning your head to the side you can see the histogram curve / density distribution a bit more clearly. In fact ggplot has a way to flip a plot, one of a set of things called a transformation.

```
p + geom_violin() + geom_jitter() + coord_flip()
```



Now you can see clearly that the setosa numbers are really badly bunched down at the lower end and a bit skewed by that.

2.5 Boxplots are best for normally distributed data.

Really, these boxplots, especially the ones with the notches only help you assess a difference if the data is nicely normally distributed ggplot

2.6 Quiz

1. Incorporate a jitter and notched boxplot into the Petal.Width and Species plots we already used: `ggplot(iris) + aes(Species,Petal.Width)...`

3 Using Factors to Subset Data and Plots

3.1 About this chapter

1. Questions:

- How can I make plots that compare multiple categories?"

2. Objectives:

- Understand factors
- Understand colouring and faceting on factors
- Use factors for summaries and plot design

3. Keypoints:

- A factor is a value of a categorical variable, or the different values a label can take
- Factors are needed to subset and add attributes to data dynamically

3.2 Factors

In previous plots we've been using categories, specifically the `Species` category to split our data, colour our plots etc. These categorical columns are called Factors in R. Looking at the `diamonds` data set we can see how this is set up in R.

```
head(diamonds)
```

```
# A tibble: 6 x 10
  carat cut       color clarity depth table price     x     y     z
  <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

Here we can see the `cut`, `color` and `clarity` columns are all non-numeric, textual data. These are the factor variables of this dataset. We can confirm that by asking for the `class` of the column, that is, the type of data in it. We use the dataset `$` column name syntax for this.

```
class(diamonds$color)
```

```
[1] "ordered" "factor"
```

```
class(diamonds$depth)
```

```
[1] "numeric"
```

We can also ask for all the different values of the factor, in R called the levels

```
levels(diamonds$color)
```

```
[1] "D" "E" "F" "G" "H" "I" "J"
```

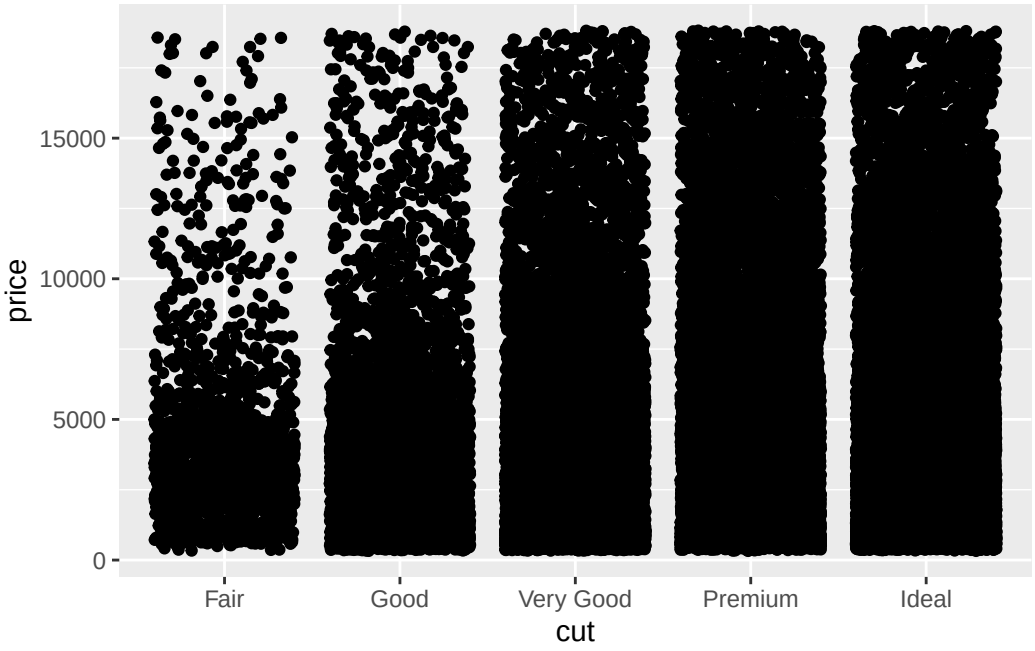
```
levels(diamonds$cut)
```

```
[1] "Fair"      "Good"      "Very Good" "Premium"   "Ideal"
```

3.3 Colouring by factors

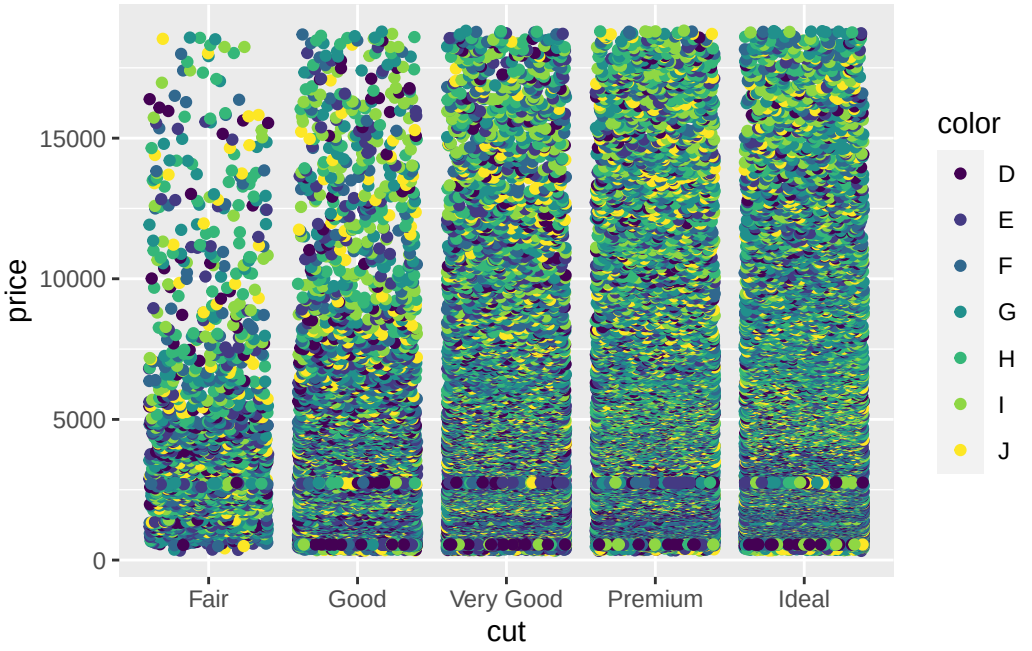
Let's look at applying mappings by a factor. Let's look at how price varies by cut.

```
p <- ggplot(diamonds) +  aes(cut,price)
p + geom_jitter()
```



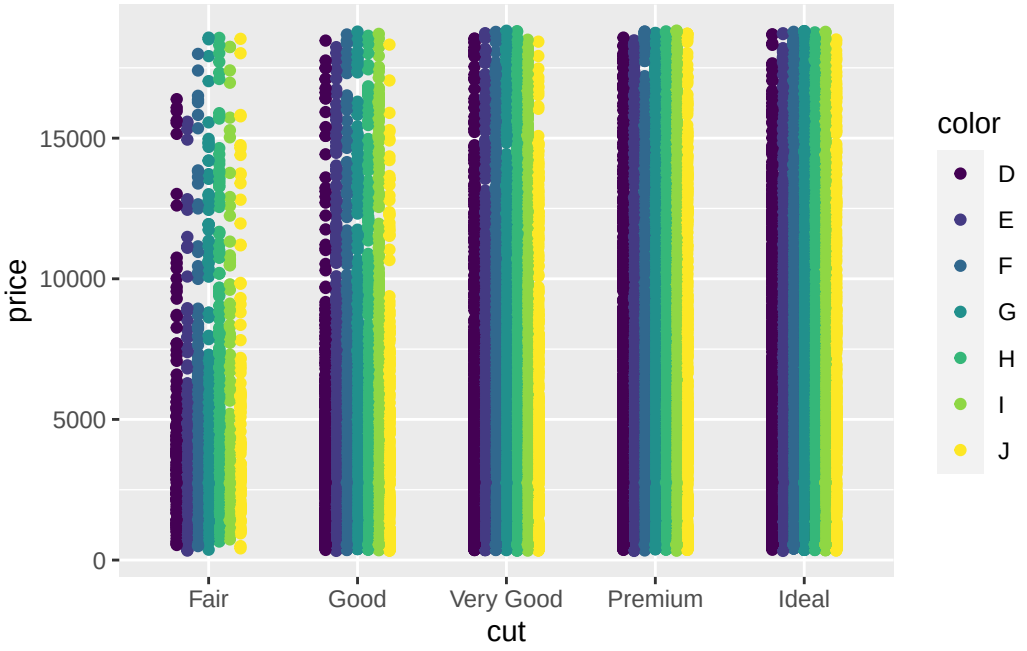
Now let's throw a second variable in there, lets see how color varies within each cut. We do this by creating a new aesthetic mapping within the geom_jitter()

```
p + geom_jitter(aes(colour=color))
```



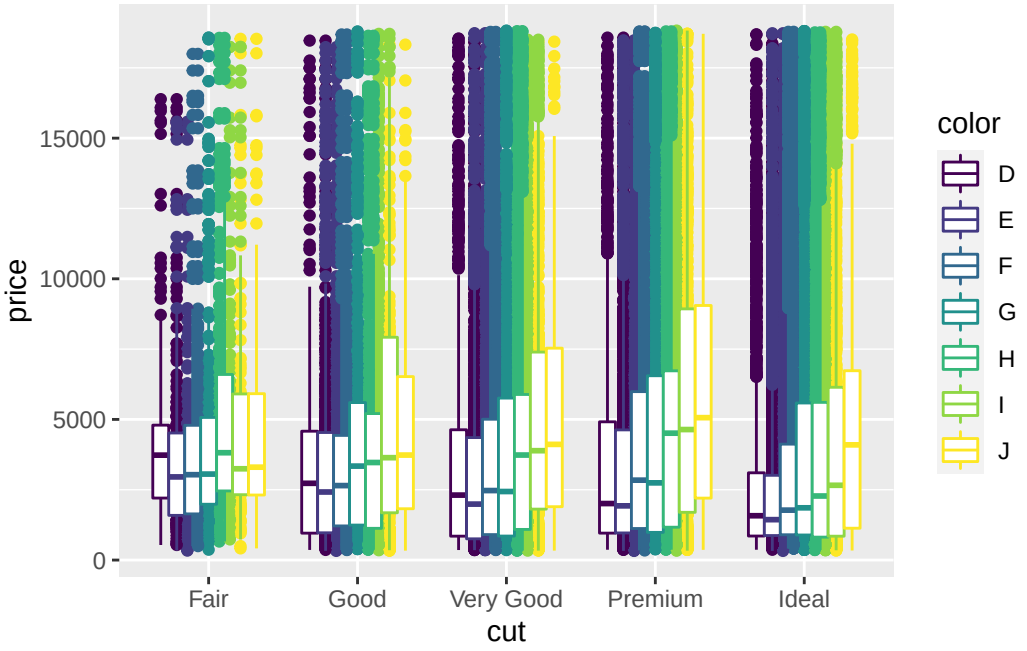
The spots are all overlapping, we can force the different colours to stay separate with the `position` option. We use `position_dodge()` to make them dodge each other. The width option tells the spots how far to stay apart.

```
p + geom_jitter(aes(colour=color), position=position_dodge(width=0.5) )
```



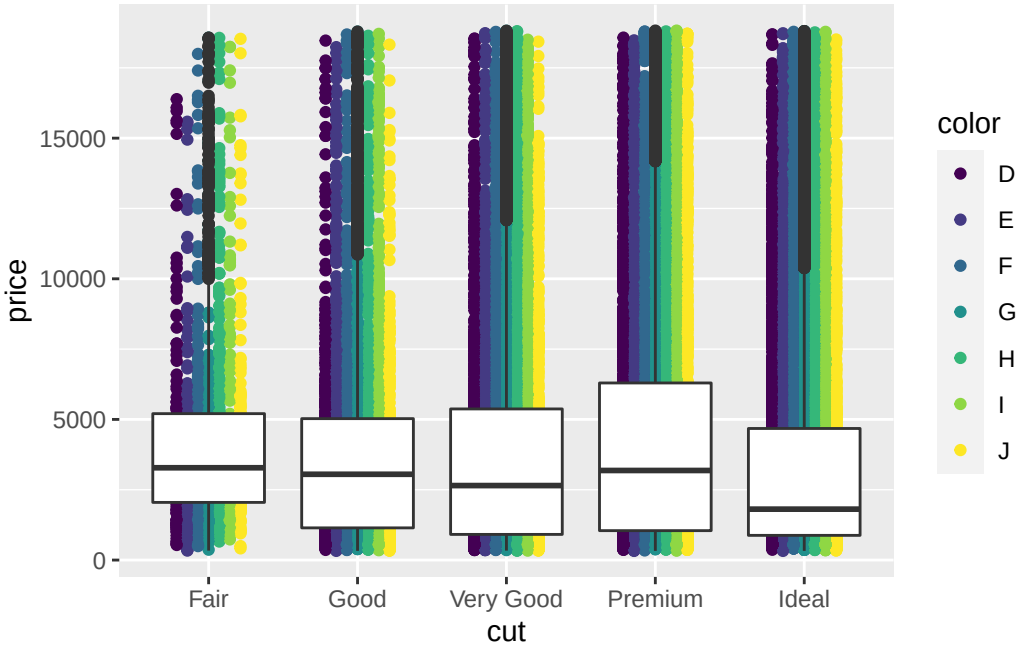
We can also throw other geoms on top in the same way. EG Boxplots for each cut and colour

```
p + geom_jitter(aes(colour=color), position=position_dodge(width=0.5) ) + geom_boxplot( ae
```



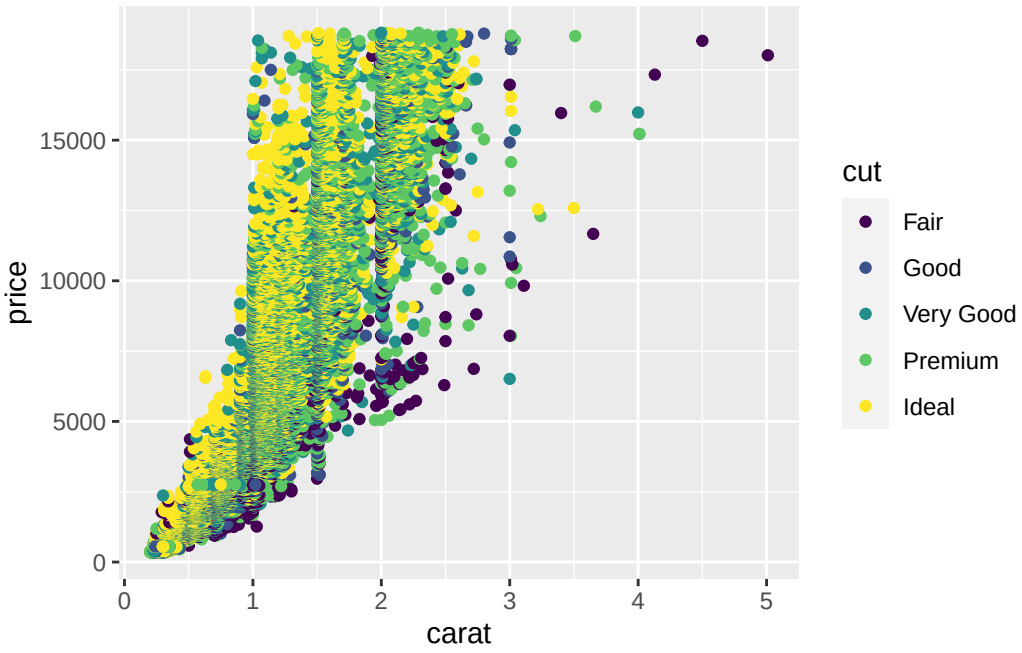
Remember layers/geoms are independent, so can be set up to show individual aspects of the data. Let's have a boxplot for the whole of the cut, irrespective of the colour.

```
p + geom_jitter(aes(colour=color),position=position_dodge(width=0.5)) + geom_boxplot()
```



And of course, the whole thing still works even if we are comparing two numerical columns. We can still use the aesthetic mapping in the geom to colour our points by a factor

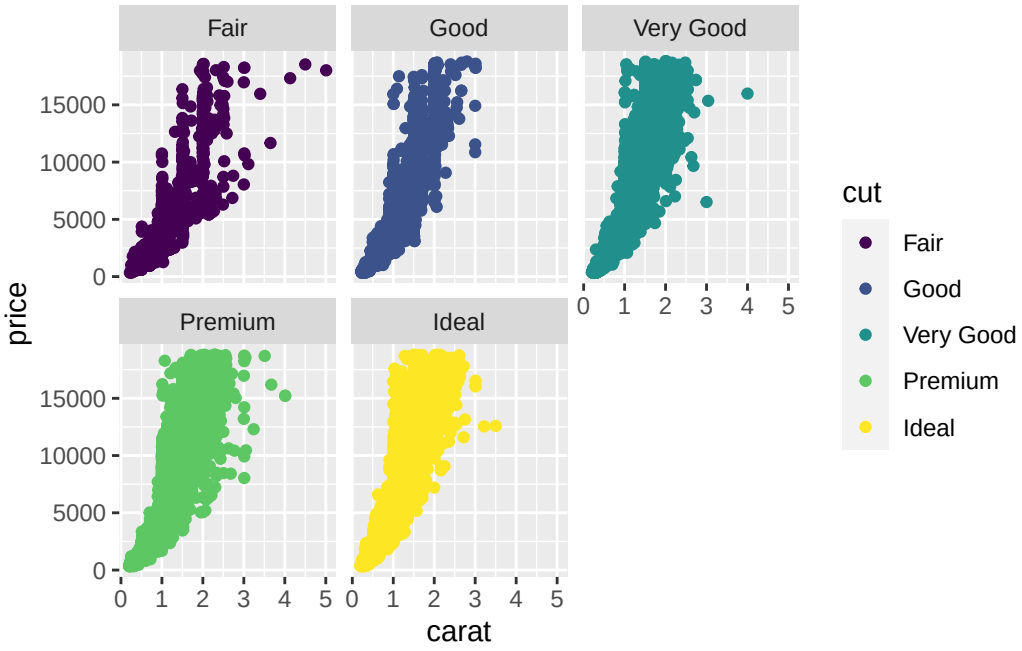
```
ggplot(diamonds) + aes(carat, price) + geom_point(aes(colour=cut))
```



3.4 Small multiple plots

Sometimes, trying to squeeze a lot of data into one plot isn't the clearest way to show it. Instead small multiple plots (different data, same settings) can be used. In ggplot, this is called faceting and is done with the `facet_wrap()` or `facet_grid()` function. We use the factors to define the facet. Let's add faceting to the previous plot

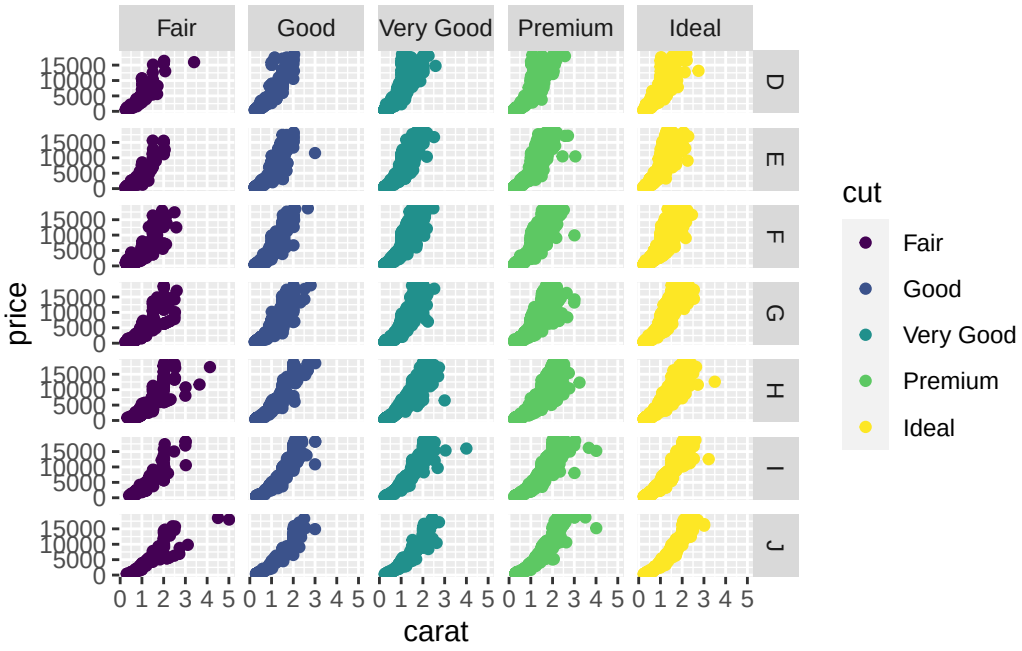
```
p <- ggplot(diamonds) + aes(carat, price)
p + geom_point(aes(colour=cut)) + facet_wrap( ~ cut)
```



Here we see the plot is divided into panels, one for each 'cut'. The `facet_wrap()` function puts all the panels into a single row, but wll wrap that row as space demands. The syntax is a bit odd, we used the `~` operator to mean 'varies by' , even though we only used one variable. It's just a quirk of ggplot.

The `facet_grid()` function forces a grid structure and can take more than one factor. Now the `~` 'varies by' syntax makes more sense:

```
p + geom_point(aes(colour=cut)) + facet_grid(color ~ cut)
```



3.5 Quiz

The built in dataset `CO2` describes measurement of CO2 uptake versus concentration for Quebec and Mississippi grasses in chilled and nonchilled tests. The dataset is as follows:

- `Type` is a factor column with two levels `Quebec` and `Mississippi`
- `Treatment` is a factor colum with two levels `nonchilled` and `chilled`
- `Uptake` is a numerical colum with CO2 uptake rate in micromoles per metre squared per second
- `Plant` is a factor with twelve levels, one for each individual plant assayed.

1. Create a plot with `geom_point()` that shows the Plant on the x-axis and the Uptake on the y-axis. Colour the points by 'Type' and `facet_wrap()` by `Treatment` to get a subplot for chilled and nonchilled.

4 Visual Customisation

1. Questions:

- How do I make my plot look the way I want it to?

2. Objectives:

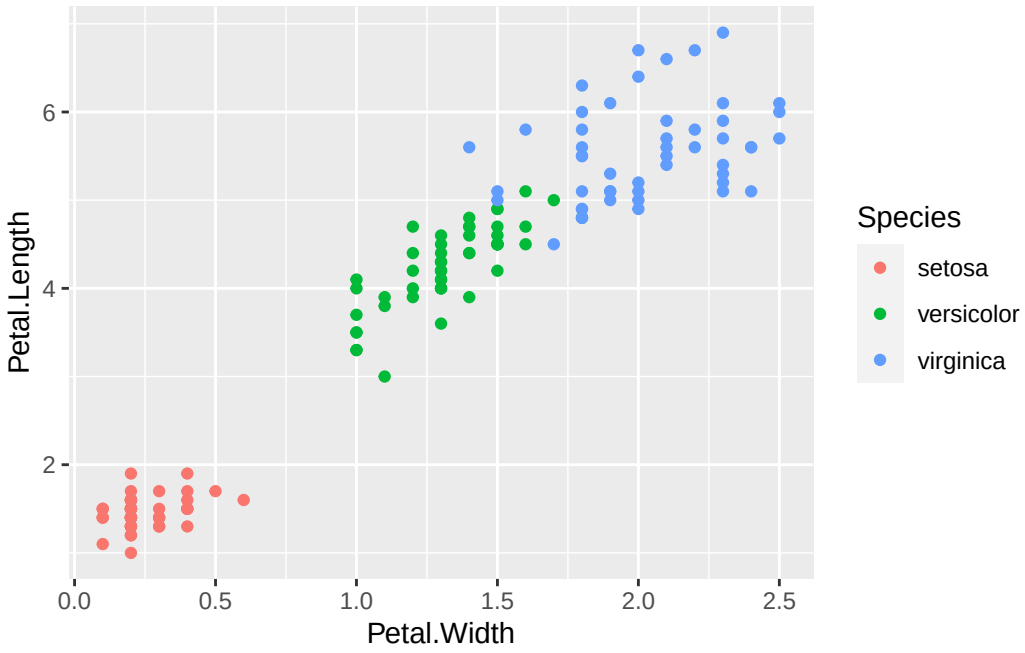
- Explain how themes are applied
- Explain how individual plot elements can be changed
- Set the order and limits on scales keypoints:
- There are a wide range of themes that can be modified
- The `theme()` function allows us to set individual theme elements
- The `scale` family of functions allows us to specify the scales

4.1 Themes

At some point you're going to want to custom/personalise or generally improve the look of your plots. So far we've concentrated on getting the data shown in the right place, now we'll look at finessing the plot to make a final version. `ggplot2` and a companion package `ggthemes` have a wide variety of ready to go themes that can be applied and modified.

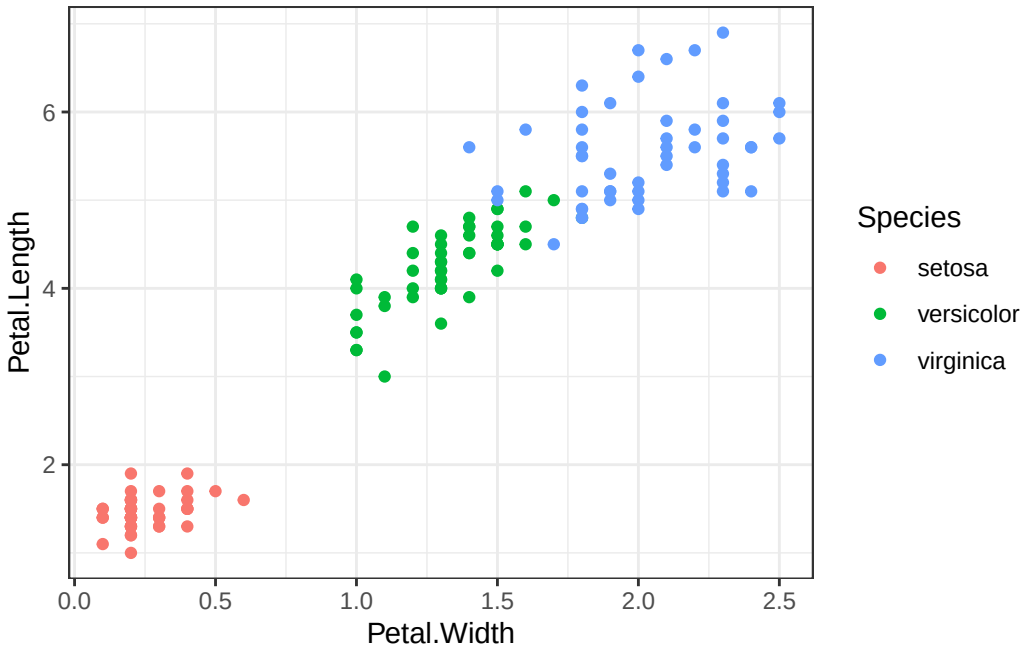
Applying a built-in theme is very easy, we can think of the theme as a new layer to add. This code will give us a standard plot

```
p <- ggplot(iris) + aes(Petal.Width,Petal.Length) + geom_point(aes(colour=Species))
p
```



Let's add a `theme_bw()` layer. Which is really a simple theme that takes away all colour you didn't explicitly ask for - so the points stay coloured.

```
p <- ggplot(iris) + aes(Petal.Width,Petal.Length) + geom_point(aes(colour=Species))
p + theme_bw()
```



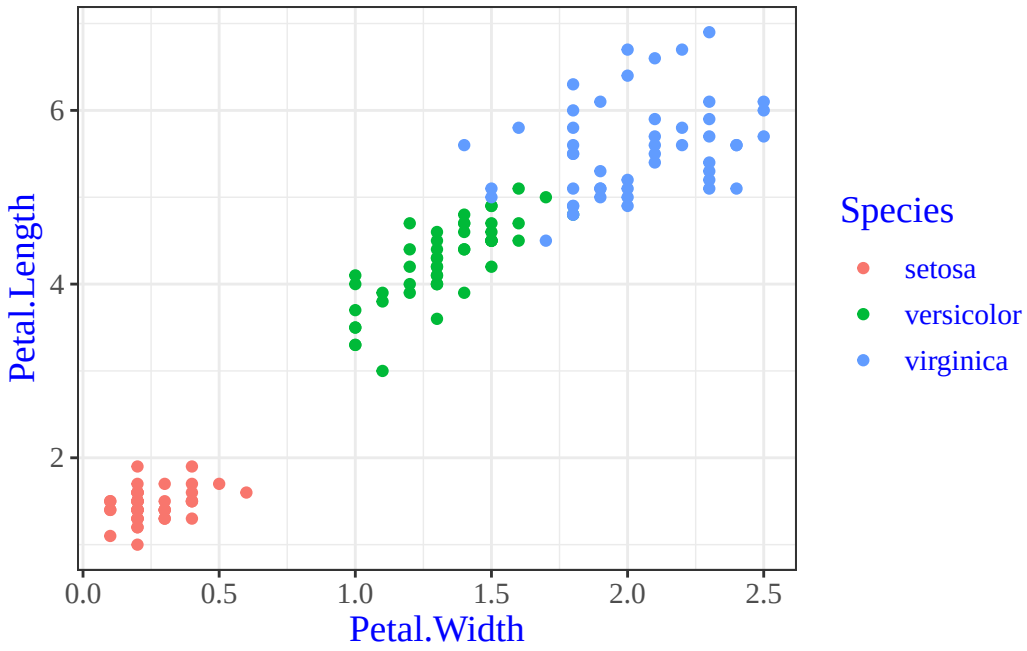
4.2 Quiz

1. `ggplot` itself only has a few themes built in. Try `theme_minimal()`, `theme_grey()` and `theme_dark()`.
2. Use the docs at https://github.com/jrnold/ggthemes to examine the themes that are available in this external package. Try loading the package and using some of the themes. Don't miss `theme_excel()`.

4.3 The `theme()` function

Changing the theme wholesale by applying a theme layer is great, but you'll usually want to change individual theme elements. This is possible too, and is done using the `theme()` function.

```
p <- ggplot(iris) + aes(Petal.Width,Petal.Length) + geom_point(aes(colour=Species))
p + theme_bw() + theme(text = element_text(family = "Times", colour = "blue", size = 14))
```



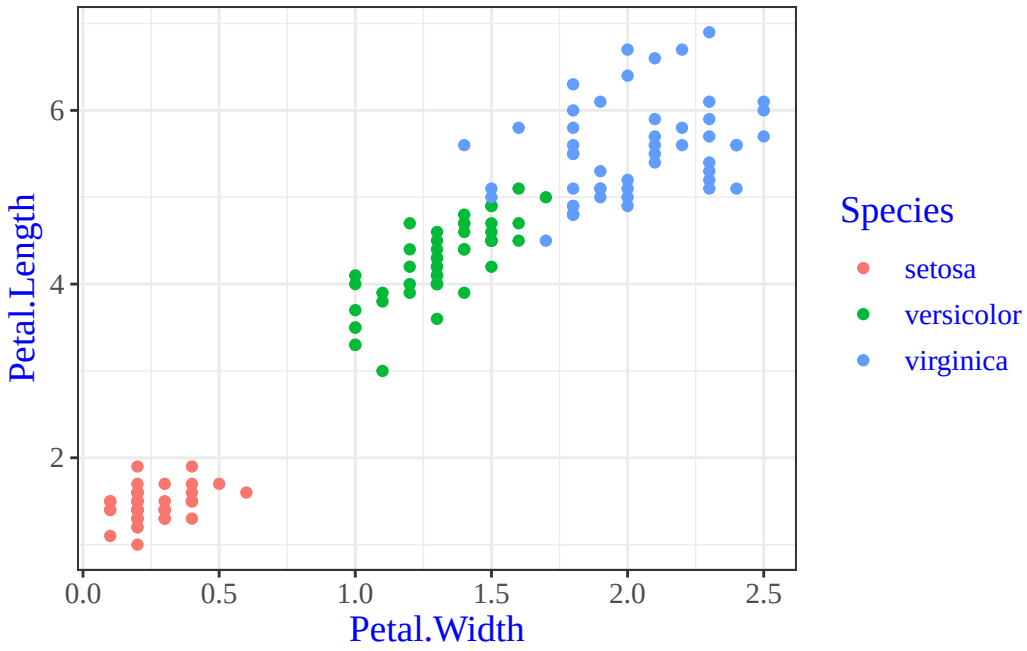
So here we built a plot, applied a theme layer and then modified an element of that theme. The theme layer is really just a list of plot elements and their current settings. Conceptually it looks like this:

```
- line = element_line(colour="black", size=0.5),
- text = element_text(family="Arial", colour="black", size=12)
```

With the thing on the left of the equals being the attribute of the plot e.g the `line` or the `text` and the thing on the right of the equals being the function that does the changing. Each plot element can be reset by using the proper function and setting the options for that function appropriately. Just like we did above!

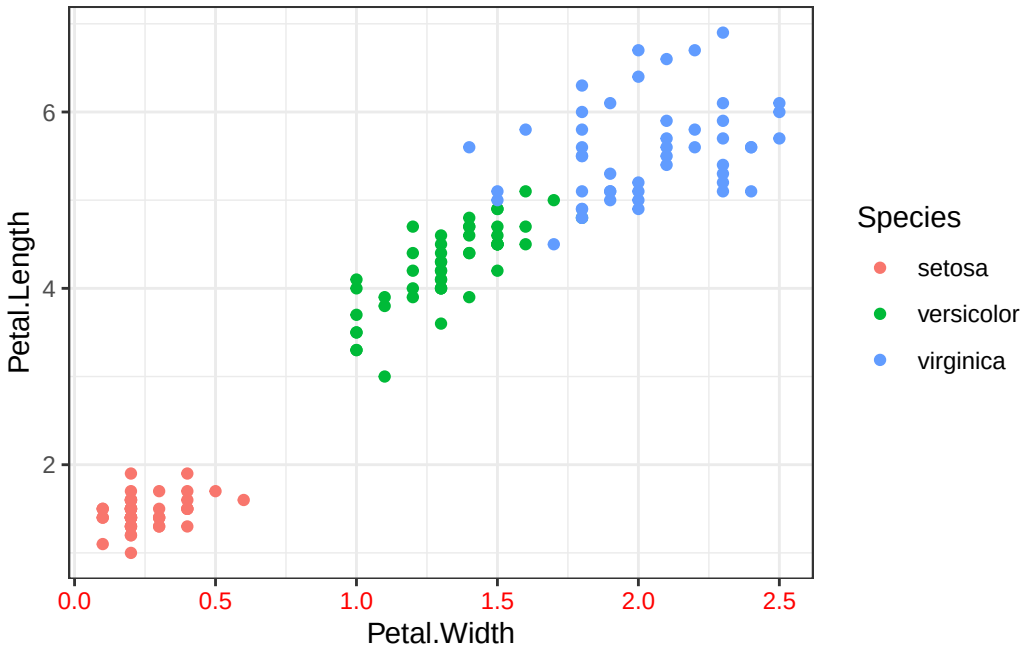
Some of the attributes apply across the whole plot, this one `text` applies to all text in the plot.

```
p + theme_bw() + theme(text = element_text(family = "Times", colour = "blue", size = 14))
```



But this one `axis.text.x` changes only the x axis text.

```
p + theme_bw() + theme(axis.text.x = element_text(colour="red"))
```



A full list of plot elements and the functions to set them are in http://docs.ggplot2.org/dev/vignettes/themes.html and here are the most important ones. The options for each element function are in the ggplot2 docs http://docs.ggplot2.org/dev/element.html

```
line =                element_line(),
rect =                element_rect(),
text =                element_text(),
axis.text =           element_text(),
strip.text =          element_text(),

axis.line =           element_blank(),
axis.text.x =         element_text(),
axis.text.y =         element_text(),
axis.ticks =          element_line(),
axis.title.x =        element_text(),
axis.title.y =        element_text(),
axis.ticks.length =   unit(),
axis.ticks.margin =   unit(),

legend.background =   element_rect(),
legend.margin =       unit(),
```

```
legend.key =          element_rect(),
legend.key.size =     unit(),
legend.text =         element_text(),
legend.title =        element_text(),

legend.position =     "right",
legend.justification = "center",

panel.background =    element_rect(),
panel.border =        element_blank(),
panel.grid.major =    element_line(),
panel.grid.minor =    element_line(),
panel.margin =        unit(),

strip.background =    element_rect(),
strip.text.x =        element_text(),
strip.text.y =        element_text(),

plot.background =     element_rect(),
plot.title =          element_text(),
plot.margin =         unit(),
```

Putting these together if we want to make our legend text a bit bigger, and use Helvetica font, in green, we'd follow this scheme:

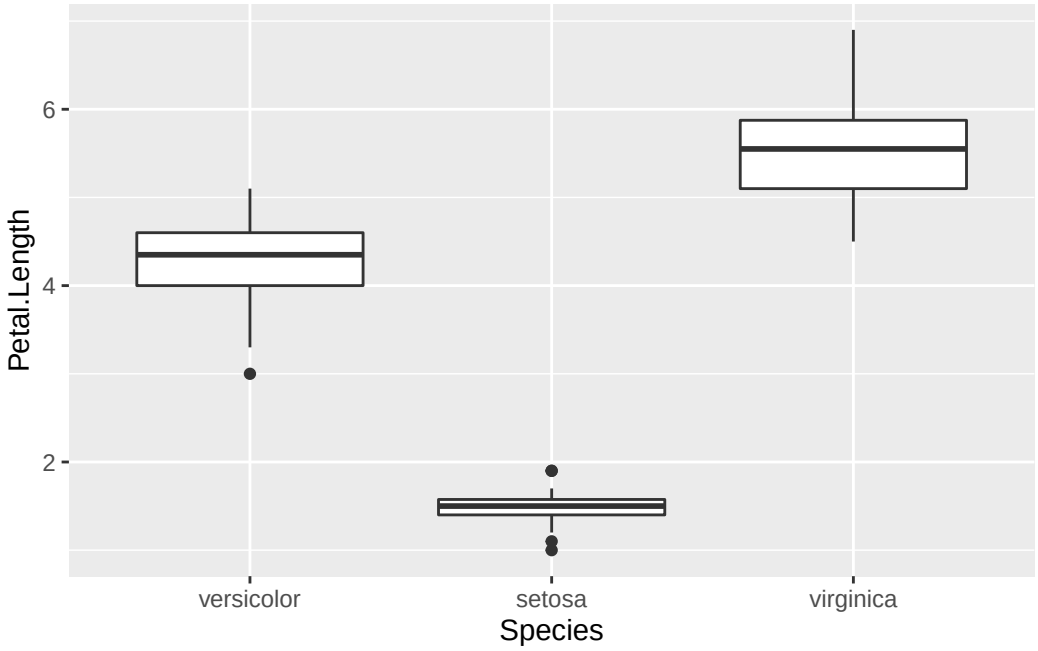
1. Use the list above to find which element is the right one for legend text. Here it will be `legend.text`
2. Read off the element function for the `legend.text`, here it is `element_text()`
3. Use the ggplot2 docs to see the options for that element function: http://docs.ggplot2.org/dev/element.html
4. Form the theme function: `theme( legend.text = element_text(size = 20, family="Helvetica", colour="green") )`
5. Add it to the plot `plot + theme( legend.text = element_text(size = 20, family="Helvetica", colour="green") )`

4.4 Changing the order of categories in the plot

The list of options above doesn't provide anything we can use to specify the order in which the different categories are displayed. Instead this is done with a new type of function, the `scale` family of functions. By using the `scale_x_discrete()` function and options (especially the

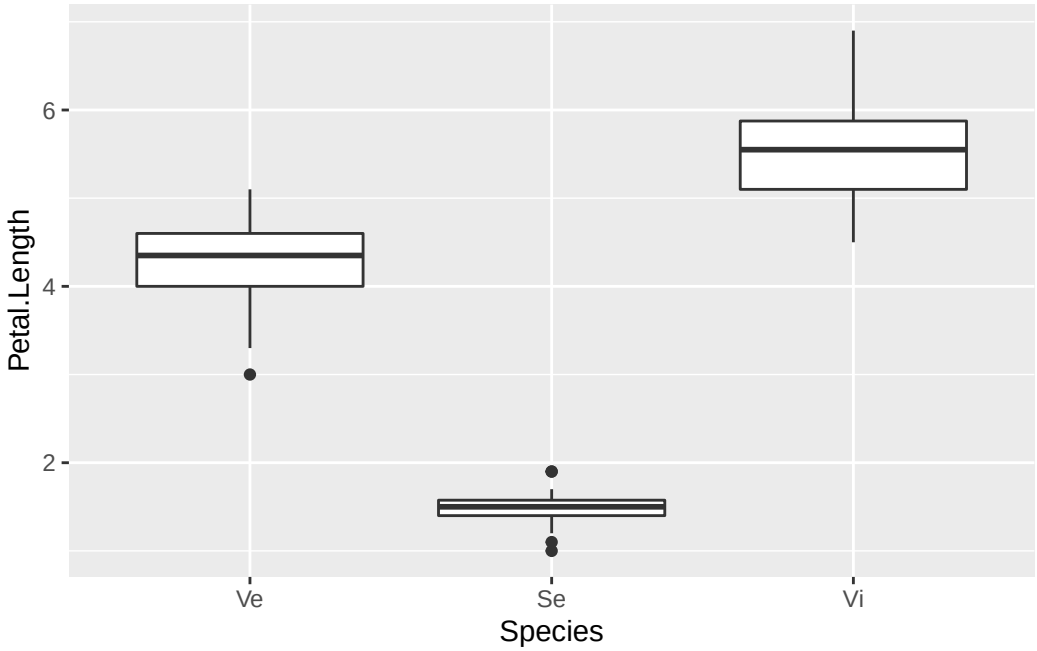
`limits`) we can set the way the scale on the axis is set. For a discrete (or categorical) variable this includes the order.

```
p <- ggplot(iris) + aes(Species,Petal.Length) + geom_boxplot()
p + scale_x_discrete(limits=c("versicolor", "setosa", "virginica"))
```



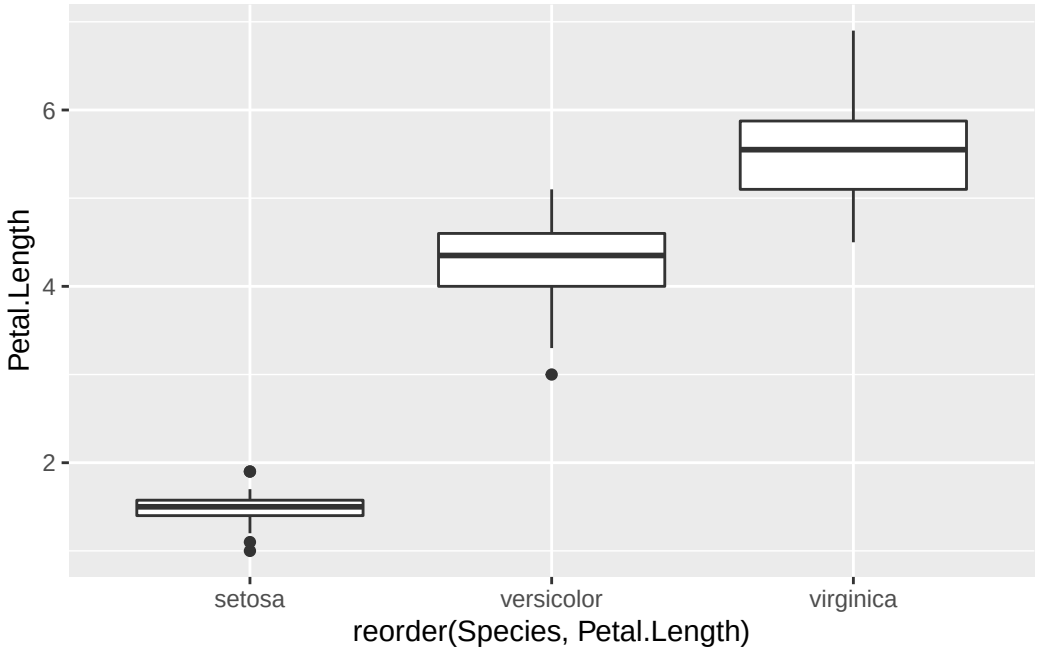
You can change labels in the same way with the `labels` option,

```
p + scale_x_discrete(limits=c("versicolor", "setosa", "virginica"), labels=c("Ve", "Se", "
```



You can reorder based on the value of some other value, e.g get the boxes ordered by the `Petal.Length` variable by squeezing in the `reorder()` function. Unusually, this is done in the `aes()` function in the aesthetic layer. We want to reorder the x-axis so we use the `reorder()` function on that. The syntax is `reorder(<variable to reorder>, <variable to reorder by)`, so here we're changing the order of the Species on the x-axis according to what is in `Petal.Length`.

```
p <- ggplot(iris) + aes(x=reorder(Species, Petal.Length), y=Petal.Length ) + geom_boxplot(
p
```



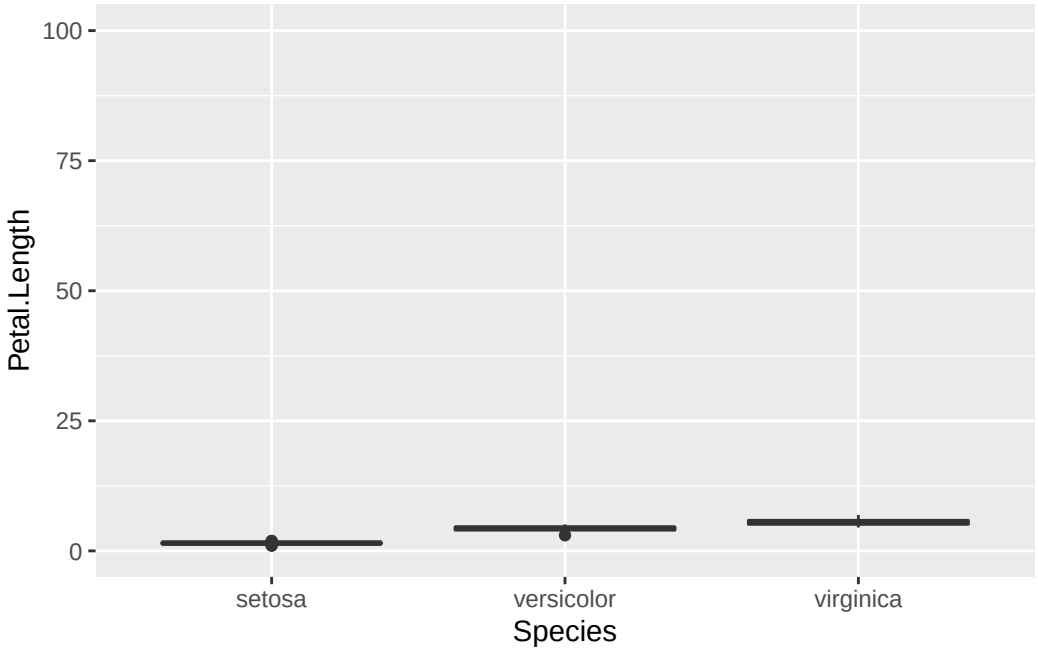
4.5 Text formatting in plots

Biological notation is frustrating because it uses text formatting to express differences between things. So the wild-type allele is referred to in italics or underlined capitals whereas a mutant is referred to in italic or underlined lower case. Programming languages have a hard time with text formatting, so tend to deal with plain text. `ggplot` is no exception and there isn't a way to make your labels italic. The best way to achieve this, therefore is to save the plot as a `.svg` file, then edit the labels manually in a graphics program like Inkscape.

4.6 Changing the limits of a continuous scale

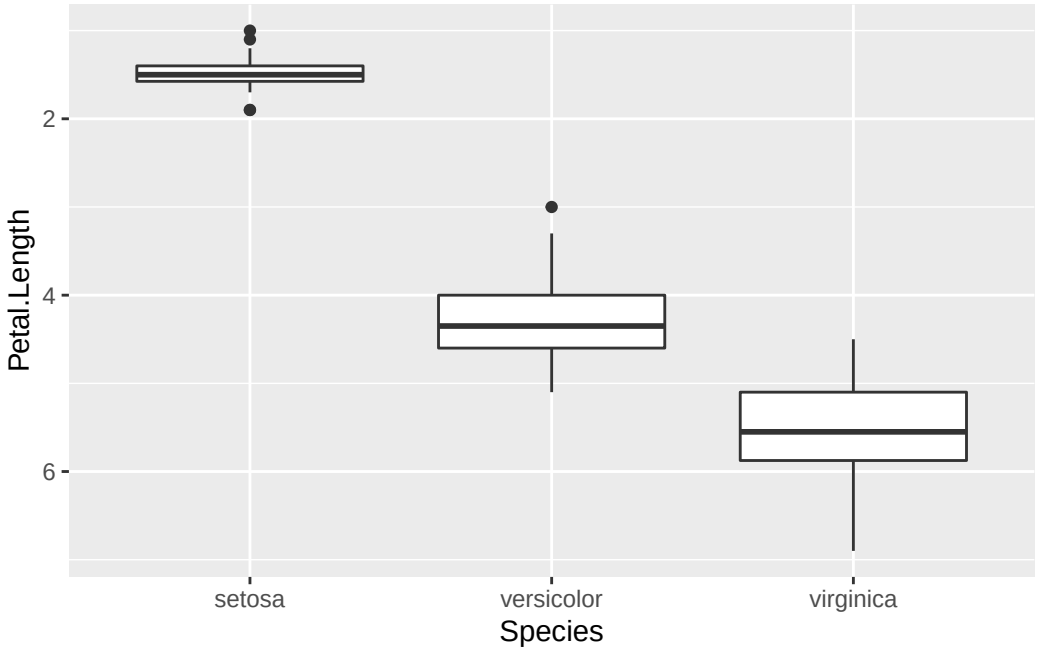
The `scale_x_discrete()` function has analogous functions for the y-axis and for continuous axes - I.E. `scale_y_discrete()` and `scale_x_continuous()` and `scale_y_continuous()`. The most common thing to want to do with a continuous scale is set the limits, the start and end points.

```
p <- ggplot(iris) + aes(Species, Petal.Length ) + geom_boxplot()
p + scale_y_continuous(limits =c(0,100))
```

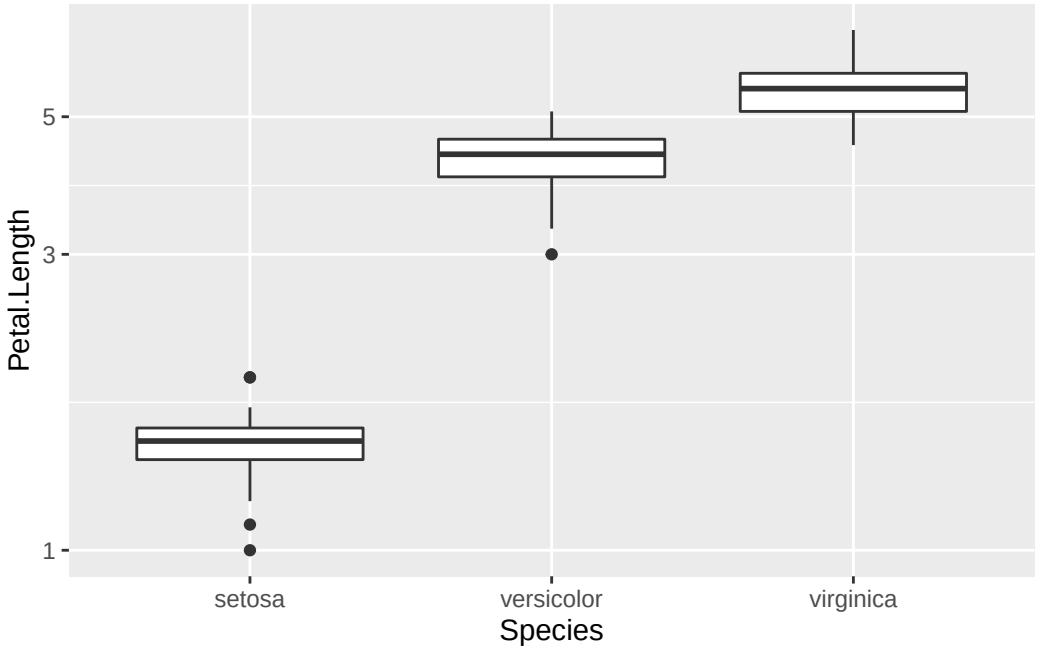


It is also possible to change the scale to a logarithmic one with the `scale_y_log10()`, function, reverse it with `scale_y_reverse()` functions.

```
p + scale_y_reverse()
```




```
p + scale_y_log10()
```



4.7 Quiz

1. Using the iris dataset, create a boxplot of Petal Width for each species
2. Overlay the actual data by adding a jitter plot
3. Remove the grey background of the plot (Hint: try `element_blank()` and `panel.background`)
4. Change the Y axis title to 'Petal Width'
5. Remove the X axis title
6. Make the species names bigger
7. Make the thick panel grid lines black, remove the thin panel grid lines.
8. Set the order of species to 'virginica', 'setosa', 'versicolor' Extra Credit: Set the values on the species axis to 'Iris virginica', 'Iris setosa', 'Iris versicolor'

Part III

Working reproducibly

5 RMarkdown for Reproducible Publishable Plots

5.1 About this chapter

1. Questions:

- How can I design a plot once and use it for many experiments?

2. Objectives:

- Use RMarkdown documents to build a plot.

3. Keypoints:

- Reproducible work is good work.
- R Markdown can helps us be reproducible and transparent

5.2 Being lazy is a virtue. Work hard to be lazy.

Writing reproducible code will save you time and effort. Computers are especially good at carrying out commands and if you are smart enough to put those commands in an executable document, rather than run the whole thing by hand every time, you'll save time, you can ensure that you'll do the same thing everytime and those who look at your work later will be absolutely clear about what you did.

Of course, this takes a little bit more effort up front, but it will pay off. And R Studio has plenty of ways to help you do just this. R Markdown documents are one such way. For the rest of this course we'll be putting our code into R Markdown.

5.3 R Markdown

Markdown is a way of adding little tags to text, to define parts of the structure of it, so that when a file written in Markdown is sent to a program that knows how to interpret it, the program can render the text as you intended.

R Studio has the ability to take a document written in Markdown, squeeze R code into it and produce the output in a pretty format. The flavour is called R Markdown. By combining this with our plotting knowledge, we can make a dynamic document that can be re-run every time we get a new dataset.

You can find more information on R Markdown in this handy cheat sheet https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf

5.3.1 A new R Markdown document

Creating an R Markdown document is easy. In R Studio, use the menu `File -> New File -> R Markdown` and you'll get a dialogue box like

Leave everything as default (making a document with output format html) and click OK. A new panel should appear in R Studio. The header looks like:

```
---
title: "Untitled"
author: "Dan MacLean"
date: "21 September 2020"
output: html_document
---
```

The top four lines are metadata about the document, R will use this to make an automatic header. You can change the values of `title`, `author` and `date` if you like. The last bit about output defines the type of document you can get.

The rest of the document is straightforward text right up to the parts with the three backticks ““‘ (weird quote things). These are the blocks of R code that will get evaluate in our R markdown. Anything between two sets of three backticks is sent to R and treated as R code, so that

```
```{r cars}

summary(cars)

```
```

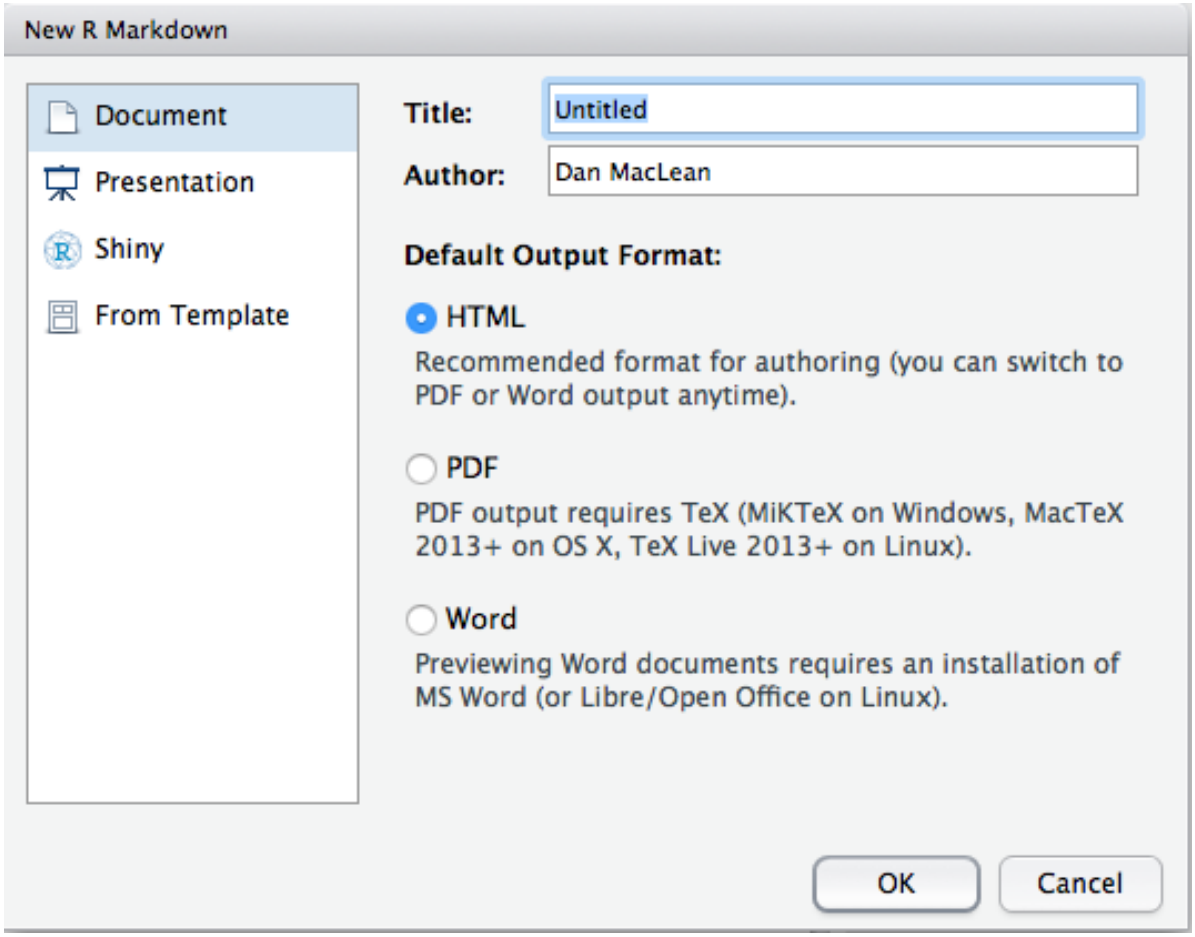


Figure 5.1: dialogue box

Gets the output of the `summary()` function in that position. To see this work, click the `Knit HTML` button and choose a filename for the R markdown document.

The eventual document produced is nicely formatted markdown with R code and results added in the proper place.

> More Information
>
> Markdown provides a rich set of tags to mark up the document to make it look as pretty as you like. Here's a cheat-sheet you can use to make your own Markdown documents https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet
> See also the course here https://danmaclean.github.io/literate_computing

5.4 Quiz

1. Make a new R Markdown document that creates and renders a plot of your choice - any of the ones you've already done will be fine. Hint: Every time you run a markdown document the computer's memory is cleared for that document, it doesn't know about what goes on outside of the document. You need to load libraries and files in the document, even if they are loaded in R Studio already

6 Loading your own data

1. Questions:

- How do I use *my* data?

2. Objectives:

- Preparing a csv file in 'tidy' format
- Understanding file system paths
- Loading a file to a dataframe
- Explicitly describing the file contents

3. Keypoints:

- Data needs to be in a particular format for `ggplot` to work
- Specifying the data type is sometimes necessary when creating a data frame.

6.1 Tidy data

There are many ways to structure data. Here are two quite common ones.

treatmenta

treatmentb

John Smith

11

2

Jane Doe

16

11

Mary Johnson

3

1

John Smith

Jane Doe

Mary Johnson

treatementa

11

16

3

treatementb

2

11

1

source: Hadley Wickham

Tables contain two things, variables and values for those variables. In these two tables there are only three variables. `treatment` is one, with the values `a` and `b` . The second is 'name', with three values hidden in plain sight, and the third is `result` which is the value of the thing actually measured for each person and treatment.

For human reading purposes, we don't need to state the variables explicitly, we can see them by interpolating between the columns and rows and adding a bit of common sense. This mixing up of variables and values across tables like this has led some to call these tables 'messy'. A computer finds it hard to make sense of a messy table.

Working with R is made much less difficult if we get the data into a 'tidy' format. This format is distinct because each variable has its own column explicitly, like this

name

treatment

result

John Smith

a

11

Jane Doe

a

16

Mary Johnson

a

3

John Smith

b

2

Jane Doe

b

11

Mary Johnson

b

1

Now each variable has a column, and each seperate observation of the data has its own row. It is *much* more verbose for a human, but R can use this easily because we are now explicit about what is called what and how it relates to everything else.

6.2 Getting your data into tidy format

The bad news here is that there is no magic function to make your data tidy. If you have an existing table then you can do this manually in Excel or some other spreadsheet package. If you have lots of data, it is possible to do it programmatically in R, see the `dplyr` and `tidyr` packages, which are complex but designed for this purpose. Also have a look at the cheat-sheet here https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf.

6.3 Loading in a CSV file

R can deal with a lot of file formats, but the most common and easily used one is 'csv', a comma-separated value file. These can be exported from virtually any spreadsheet program so it's a good interchange format to get data into R from wherever you already have it. Loading a file is done easily with the `readr` package'.

`readr` is a tool for loading data into R. It can be loaded on its own with `library(readr)`. We will use `readr` to load in data from a 'flat' `.csv` file.

6.3.1 read_csv()

The main function is `read_csv()` which can read a standard comma seperated values file from disk into an R dataframe. There are a few variants of `read_csv()` which may be appropriate for different sorts of `.csv` file, but they all work the same.

- `read_csv2()` - reads semi-colon delimited files, which are commonly used where a comma is used as a decimal separator
- `read_tsv()` - reads tab delimited files
- `read_delim()` - reads files delimited by an arbitrary character

The first argument to `read_csv()` is the path to the file to read. Here I'll read a file on my Desktop that contains the diamonds data we've been using.

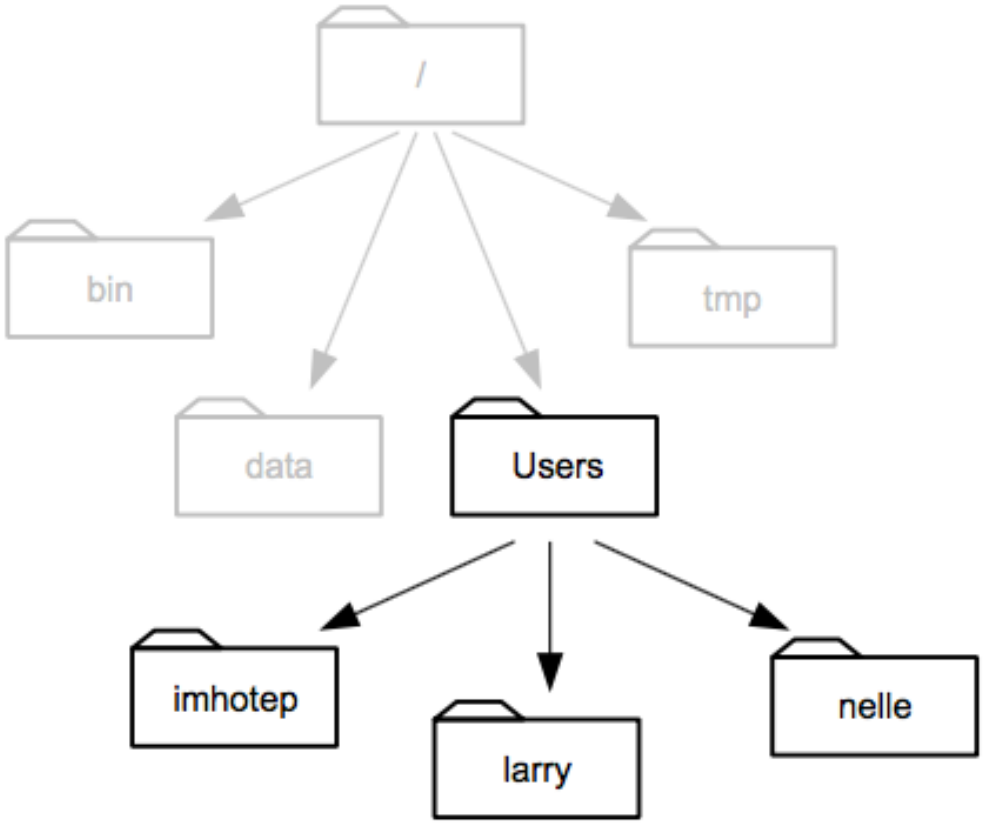
```
library(readr)
read_csv("~/Desktop/diamonds.csv")
```

This will create an object called a dataframe that can be used just like the built-in data.

6.4 Finding the file

R needs to be given the correct and full path to a file. This means the full address of the file on the hard disk of your computer. R doesn't have a file chooser so you need to know how to write this down.

Computer file systems are laid out in folders and sub-folders with files inside them. Conceptually, this results in a tree of folders and a path down the branches from the root of the tree to everything else. The root gets called '/' on Mac/Linux computers and 'C:' on Windows computers



source: Software Carpentry

This picture of an example file system shows how that is formed. When we write this down, everytime we go inside a new folder we use a slash to show we've changed folder. Most computer systems have a 'Users' or similar folder in which each users stuff is stored. Supposing we're in Larry's folder then the path would be `/Users/larry`. And a file called `my_file.txt` in that folder would be `/Users/larry/my_file.txt`.

So to write the full file path for R we can use this pattern, the first bit would be `/Users/username/` (or `C:\Documents and Settings\username\` or `C:\Users\username\`) and then the set of folders within that user area follows on. If your file `my_file.txt` is on the Desktop the full path would be `/Users/username/Desktop/my_file.txt` (or `C:/Documents and Settings/username/Desktop/my_file.txt`)

6.4.1 Make it easy on yourself

The easiest way not to have to think too hard about this stuff is to set up a consistent folder and file structure for every analysis and use RMarkdown documents to run your analysis. Here's an example scheme:

1. Create a new folder and call it something relevant to your experiment, e.g `disease_incidence_2019-11-01`
2. Within the folder create a sub-folder called `raw` and a sub-folder called `output_images`.
3. Put your tidy csv file in the `raw` folder.
4. Create a new R Markdown document and save it in the `disease_incidence_2019-11-01` folder.

Now whenever you open and run that R Markdown document, the path of your input file is `"raw/my_input_filename.csv"`. You can save your plots with the `ggsave()` function to `"output_images/filename.png"` (don't forget the quotes).

If you never mess around with the relative positions of the files and folders described, then the paths will always be the same. You can move the whole folder without worrying, just don't jumble it's contents.

6.5 Making sure the data types are correct

When we load new data we need to make sure that any header has been properly parsed as column names, and that the columns have been identified as the right sort of data

On loading with `readr` we see a column specification, `read_csv()` has guessed at what the columns should be and made those types. Its fine for the most part, but some of those columns we'd prefer to be factors. We can set our own column specification to force the column types on loading. We only have to do the ones that `read_csv()` gets wrong. Specifically, lets fix `cut` and `color` to a `factor`. We can do that with the `col_types` argument.

```
read_csv("~/Desktop/diamonds.csv",
    col_types = cols(
      cut = col_factor(NULL),
      color = col_factor(NULL)
    )
)
```

6.5.1 Parser functions

This works by assigning a parser function that returns a specific type to each column, here it's `col_factor()`. There are parser functions for all types of data, and all of them can be used if `read_csv()` doesn't guess your data properly. We won't go into detail of all of them, just remember that if your numbers or dates or stuff won't load properly, there's a parser function that can help.

Once the specification shows what you expect, then you are good to start analysing.

6.6 Quiz

```
1. Make a new folder called `analysis` on the Desktop
2. Inside `analysis` make a new folder called `raw` and put `example_ros_data_flg22.xlsx` in
3. Start a new R Markdown document and save it in `analysis`
```

2. Convert `raw/example_ros_data_flg22.xlsx` into a 'tidy' format .csv file and save to `raw`
3. Load in the data from the tidy file using `read.csv()` (Hint: You may need to save a csv version from Excel - R won't read .xlsx files.)
4. Check the datatypes and headers using `str()`, change them if necessary.
5. Create a plot that shows each data point in each treatment (Col, pp2c38, pp2c48 pp2c38/pp2c48) in each day the experiment was done.
6. Make sure the plot you generate gets saved to a folder inside `analysis` called `output_images`

Prerequisites

No specific knowledge prerequisites for this book but it will help if you are familiar with some common statistical tests, `t`, `ANOVA` and regression for the later parts. You will also find a knowledge of how to write computer file paths helpful.

You need to install the following stuff for this book:

1. R
2. RStudio
3. An R package: *ggplot2*
4. You will need to download these files and save them to somewhere on your computer: diamonds.csv and example_ros_data_flg22.xlsx,

Installing R

Follow this link and install the right version for your operating system https://www.stats.bris.ac.uk/R/

Installing RStudio

Follow this link and install the right version for your operating system https://www.rstudio.com/products/rstudio/download/

Installing R packages in RStudio.

Standard packages

Start RStudio and use the `Packages` tab in lower right panel. Click the install button (top left of the panel) and enter the package name eg `ggplot2`, then click install as in this picture

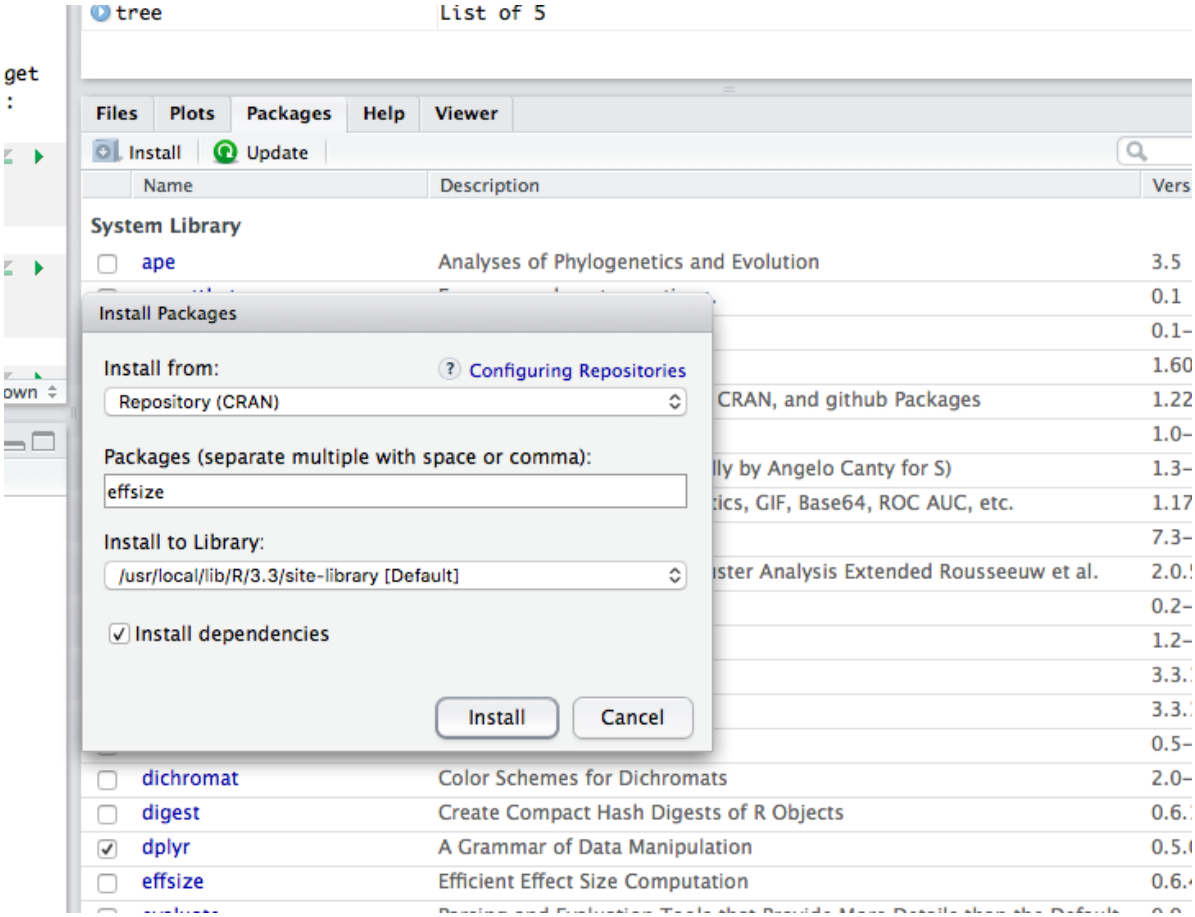


Figure 1: Installing Packages

R Fundamentals

About this chapter

1. Questions:

- How do I use R?

2. Objectives:

- Become familiar with R syntax
- Understand the concepts of objects and assignment
- Get exposed to a few functions

3. Keypoints:

- R's capabilities are provided by functions
- R users call functions and get results

Working with R

In this workshop we'll use R in the extremely useful RStudio package. For the most part we'll work interactively, meaning we'll type stuff straight into the R console in RStudio (Usually this is a window on the left or lower left) and get our results there too (usually in the consoled or in a window on the right). That's what you see in panels like the ones below - first the thing to type into R, and below it, the calculated result from R. Let's look at how R works by using it for it's most basic job - as a calculator:

```
3 + 5
```

```
[1] 8
```

```
12 * 2
```

```
[1] 24
```

```
1 / 3
```

```
[1] 0.3333333
```

```
12 * 2
```

```
[1] 24
```

```
 3 / 0
```

```
[1] Inf
```

Fairly straightforward, we type in the expression and we get a result. That's how this whole book will work, you type the stuff in, and get answers out. It'll be easiest to learn if you go ahead and copy the examples one by one. Try to resist the urge to use copy and paste. Typing longhand really encourages you to look at what you're entering.

As far as the R ouput itself goes, it's really straightforward - its just the answer with a `[1]` stuck on the front. This `[1]` tells us how far through the output we are. Often R will return long lists of numbers and it can be helpful to have this extra information

Variables

We can save the output of operations for later use by giving it a name using the assignment symbol `<-`. Read this symbol as 'gets', so `x <- 5` reads as 'x gets 5'. These names are called variables, because the value they are associated with can change.

Let's give five a name, `x` then refer to the value 5 by it's name. We can then use the name in place of the value. In the jargon of computing we say we are assigning a value to a variable.

```
x <- 5
x
```

```
[1] 5
```

```
x * 2
```

```
[1] 10
```

```
y <- 3
x * y
```

```
[1] 15
```

This is of course of limited value with just numbers but is of great value when we have large datasets, as the whole thing can be referred to by the variable.

Using objects and functions

At the top level, R is a simple language with two types of thing: functions and objects. As a user you will use functions to do stuff, and get back objects as an answer. Functions are easy to spot, they are a name followed by a pair of brackets like `mean()` is the function for calculating a mean. The options (or arguments) for the function go inside the brackets:

```
sqrt(16)
```

```
[1] 4
```

Often the result from a function will be more complicated than a simple number object, often it will be a vector (simple list), like from the `rnorm()` function that returns lists of random numbers

```
rnorm(100)
```

```
  [1] -1.55022232  0.64267350 -1.02041669  0.81040874 -1.66594573 -1.80322518
  [7] -0.02303992  1.27652097  0.67020028  0.38942804 -1.03328581 -0.11089460
 [13]  0.35429799  0.34178647  0.33522924 -0.19909834  0.21350367 -0.93112668
 [19]  0.27049638  0.77647805 -1.63033736 -1.97034872  2.02364894 -0.09120956
 [25]  0.55658902  0.52234433  0.18276791  0.18455781  0.81952242 -0.74604408
 [31] -1.66258175  0.16192874 -0.10858720 -0.48338949 -0.46596995  1.27888763
 [37] -0.67249848 -0.56294890 -0.74856173 -0.66297259 -0.59317042 -1.25551487
```

```
 [43]  0.17163724 -1.29287904  1.33106108 -0.03585880  0.02964410  0.28795277
 [49] -1.77126496 -1.08741325 -1.77784054 -1.73562790  0.49837335  0.60681509
 [55] -0.32534300  1.01860992 -0.68301170 -0.33454393 -0.78518139  0.37642710
 [61]  0.96636368  2.52059095  0.54652027  0.50892534  0.22747171  0.84123840
 [67]  0.56318178 -1.15396182  0.05282529 -0.78901900  0.18255634  0.50370595
 [73]  0.15807150 -1.24035373 -1.34490126 -1.57975374 -0.24109109  0.18281121
 [79]  1.24120507 -0.44966916 -0.37331420 -0.46666420 -0.76206726  1.87383458
 [85] -1.71818339  0.75755351 -1.55201094 -0.95884826 -2.81983510  0.25745116
 [91]  0.54921440  1.20834855  1.17924198  2.04841104 -0.97594694 -0.25358077
 [97] -0.64548053 -0.23677434  0.04228453 -0.40805164
```

We can combine objects, variables and functions to do more complex stuff in R, here's how we get the mean of 100 random numbers.

```r
numbers <- rnorm(100)
mean(numbers)
```

```
[1] 0.0008991274
```

Here we created a vector object with `rnorm(100)` and assigned it to the variable `numbers`. We than used the `mean()` function, passing it the variable `numbers`. The `mean()` function returned the mean of the hundred random numbers.

Quiz

1. Create two variables, `a` and `b`: Add them. What happens if we change a and then re-add a and b?
2. We can also assign `a + b` to a new variable, `c`. How would you do this?
3. Try some R functions: `round()`, `c()`, `range()`, `plot()` hint: Get help on a function by typing `?function_name` e.g `?c()`. Use the `mean()` function to calculate the average age of everyone in your house (Invent a housemate if you have to).

Acknowledgements