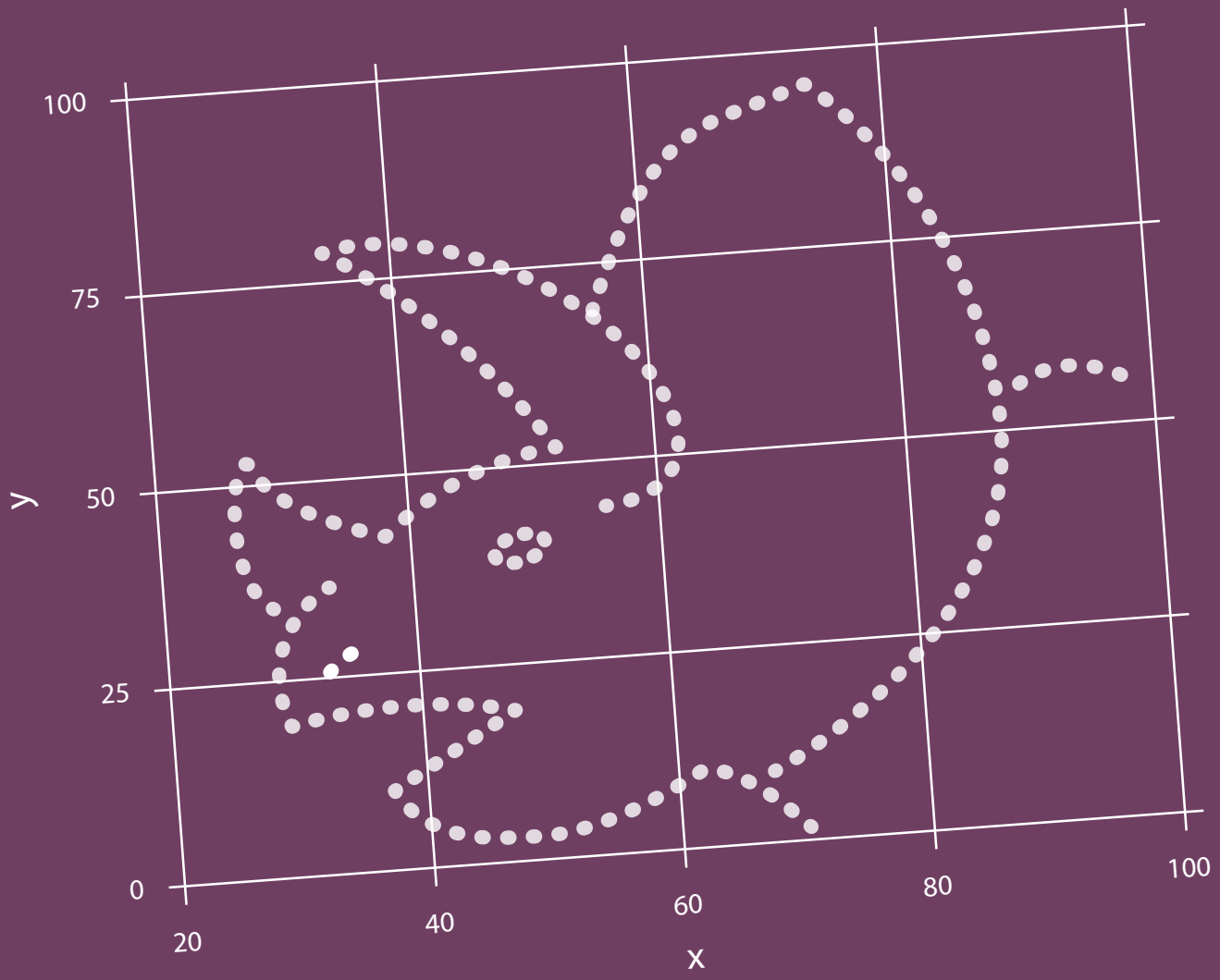
THE HITCHHIKER'S GUIDE TO PLOTNINE

Don't panic and create beautiful plots with Python



Jodie Burchell
Mauricio Vargas

The Hitchhiker's Guide to Plotnine

Don't panic and create beautiful graphs with Python

by

Jodie Burchell, Mauricio Vargas

The Hitchhiker's Guide to Plotnine

Jodie Burchell & Mauricio Vargas

This book is for sale at https://leanpub.com/plotnine-guide

This version was published on 28th October, 2019.

Contents

What to expect from this book

As users of R, we are both massive fans of the elegant `ggplot2` which makes it simple to create beautiful plots with just a few lines of code. However, when moving to Python we found that `matplotlib` was not quite as intuitive or easy to use. Fortunately, `plotnine` has been created[1], which is a stable and comprehensive port of `ggplot2` to Python. `plotnine` sits over `matplotlib`, which allows it to use the graphing functionality already built into Python with the grammar of `ggplot` to give it a much greater ease of use.

This book was inspired by our previous book[2] on using `ggplot` in R; however, the code in this book is completely new and tailored for using `plotnine` in Python. The focus of the book is technical, and aims to get you to create your own customised graphs in `plotnine` as quickly as possible. In this sense, the book resembles a recipe book with detailed instructions on how to make each change and customisation in your own plots. It is assumed that you know the basics of Python and want to focus on making beautiful plots with `plotnine`; as such, the basics of using Python or core packages such as `pandas` and `numpy` will not be covered in this book. However, we have included some resources in our further reading list if you need help getting started with Python and `pandas` more generally.

Each chapter will explain how to create a different type of plot, and will take you step-by-step from a basic plot to a highly customised graph. Each chapter is independent from the others. You can read the whole book or go to a section of your interest, and we are sure that it will be easy to understand the instructions and reproduce our examples without reading the earlier chapters. Chapters 1-4 are based on an international trade dataset created from different sources (Chile Customs, Central Bank of Chile and General Directorate of International Economic Relations). The rest of the chapters are based in classic example datasets (`diamonds`, `galton` and `airquality`).

Digital formats allow us to update this book on a constant frequency. In this version we are using Python 3.7.1. To make sure you are using the same version

[1] `https://plotnine.readthedocs.io/en/stable/`
[2] `https://leanpub.com/hitchhikers_ggplot2`

of the packages as this version of the book, the materials for this book also include a requirements file (`plotnine_requirements.txt`).

We invite you to stay in touch and read the authors' blogs where they publish articles about Python, R and Statistics. Jodie's blog is Standard error[3] and Mauricio's blog is Pachá (Batteries Included)[4]. We'd also like to thank the talented Aga Kozmic for designing the cover of this book. You can see more of her work on her website[5].

Don't panic!

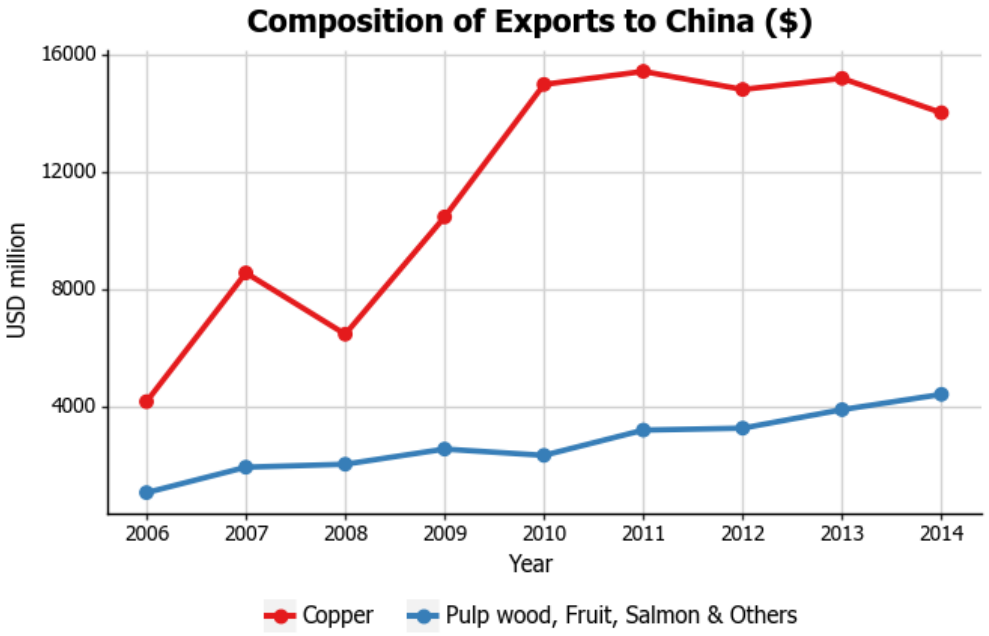
[3] `http://t-redactyl.io/`
[4] `https://pacha.hk/`
[5] `https://agakozmic.com/`

Chapter 1

Line plots

1.1 Introduction

In this chapter, we will work towards creating the line plot below. We will take you from a basic line plot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from pandas import DataFrame
```

For this chapter, we'll be using a trade dataset put together by the book's authors. The exact sample we've used in this chapter can be downloaded from here[1]. You can load the data into Python like so:

```
copper = pd.read_csv("https://git.io/JecIS")
```

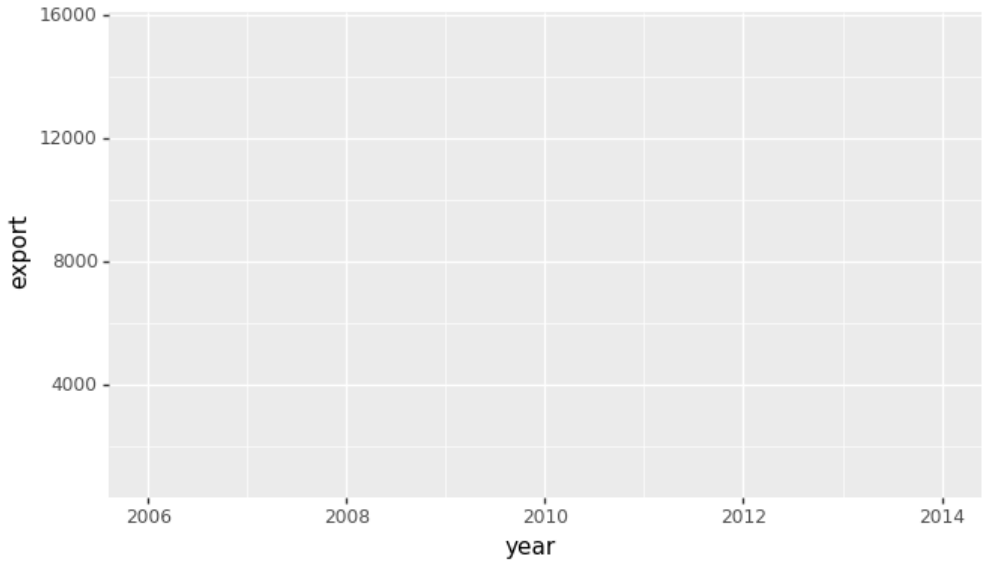
1.2 Basic ggplot structure

In order to initialise a line plot we tell ggplot that `copper` is our data, and specify that our x-axis plots the `year` variable, our y-axis plots the `export` variable, and our grouping variable is called `product`. You may have noticed that we put our variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `year` to the x-axis and `export` to the y-axis. We'll see how it handles `product` in a minute.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell `ggplot` how we want to visually represent it.

```
p1 = ggplot(copper, aes("year", "export", colour="product"))
p1
```

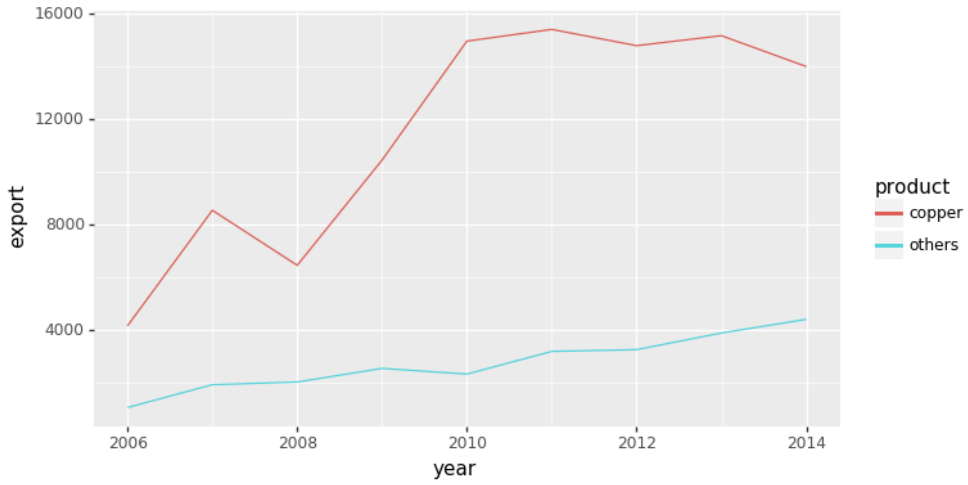
[1] `https://git.io/JecIS`



1.3 Basic line plot

We can do this using `geoms`. In the case of a line plot, we use the `geom_line()` geom.

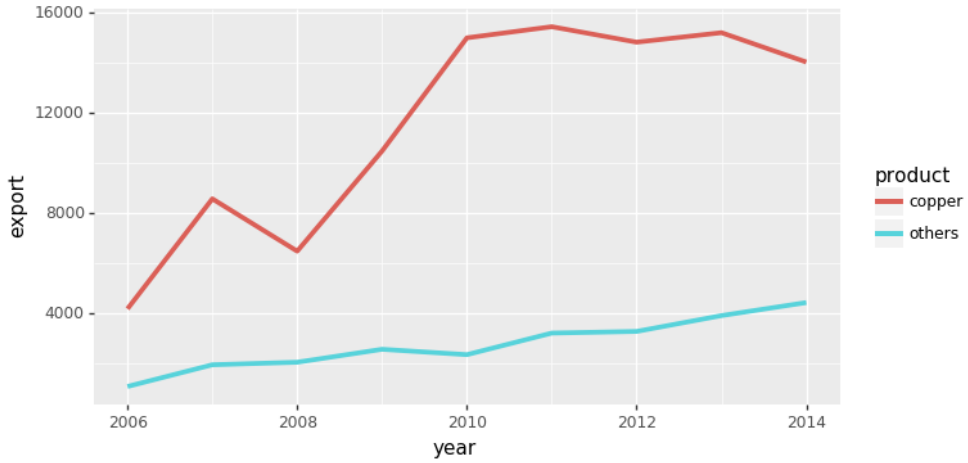
```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
      + geom_line()
)
p1
```



1.4 Adjusting line width

To change the line width, we add a size argument to `geom_line`.

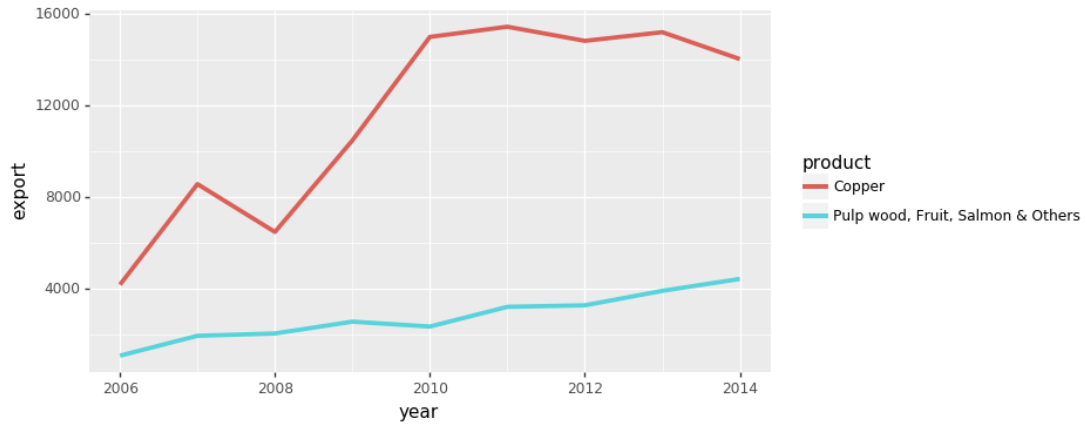
```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
      + geom_line(size=1.5)
)
p1
```



1.5 Changing group labels

To change the names of the groups in the legend, we need to rename the levels of the `product` variable. We can do this by creating a dictionary called `prod_name`, create another dictionary within *that*, and put in the values we want to change as key-value pairs. We then pass this dictionary to the `replace` method, making sure to set the `inplace` argument to `True` to make sure we overwrite the previous values of the column.

```
prod_name = {
    "product": {
        "copper": "Copper",
        "others": "Pulp wood, Fruit, Salmon & Others"
    }
}
copper.replace(prod_name, inplace=True)

p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
      + geom_line(size=1.5)
)
p1
```



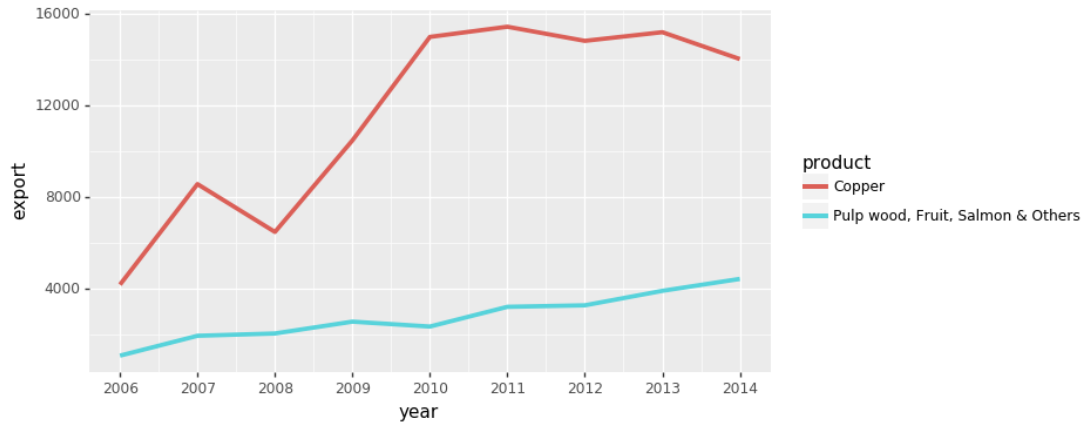
1.6 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, if we also wanted to change the y-axis we could use the `scale_y_continuous` option. Here we will change the x-axis to every year, rather than every 2 years. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[2] which generates a sequence from your selected start, stop and step values respectively. Note that because of Python's indexing when, you need to set the `stop` argument to be one number more than your desired maximum when using `np.arange`.

Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis (passed as a list), although we haven't done so in this plot.

[2] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html`

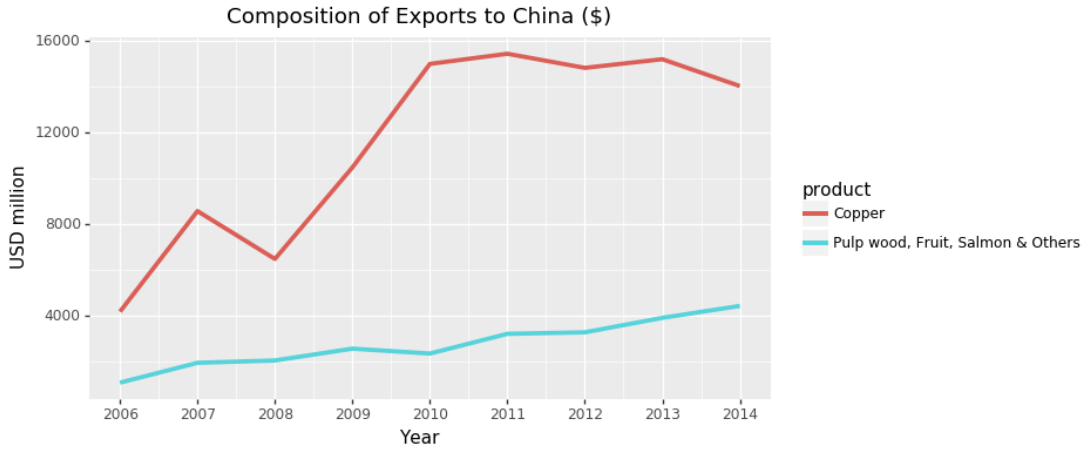
```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
)
p1
```



1.7 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
)
p1
```



1.8 Adjusting the colour palette

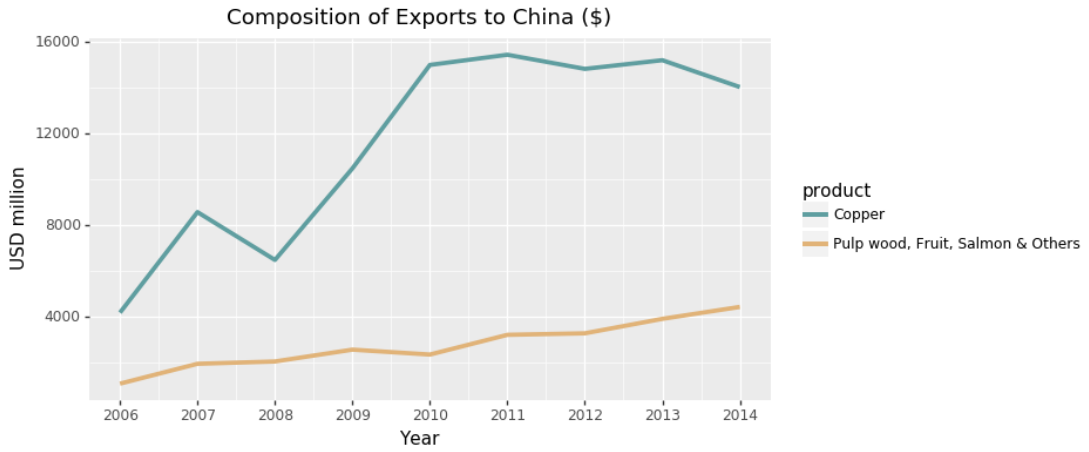
There are a few options for adjusting the colour. The most simple is to change each level to a named colour using `scale_colour_manual`. `plotnine` uses the colour palette utilised by `matplotlib`, and the full set of named colours recognised by `ggplot` is here[3]. Let's try changing our lines to `rebeccapurple` and `gold`.

```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_manual(["rebeccapurple", "gold"])
)
p1
```

[3] `https://matplotlib.org/examples/color/named_colors.html`

We can also change the colours using specific HEX codes. Let's change the lines to `#5F9EA0` and `#E1B378`.

```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_manual(["#5F9EA0", "#E1B378"])
)
p1
```



An alternative to using manual colours is to use the schemes from ColorBrewer[4]. Here we have used the `scale_colour_brewer` option with the qualitative scale `Accent`. More information on using `scale_colour_brewer` is here[5].

```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_brewer(type="qual", palette="Accent")
)
p1
```

1.9 Adjusting the legend

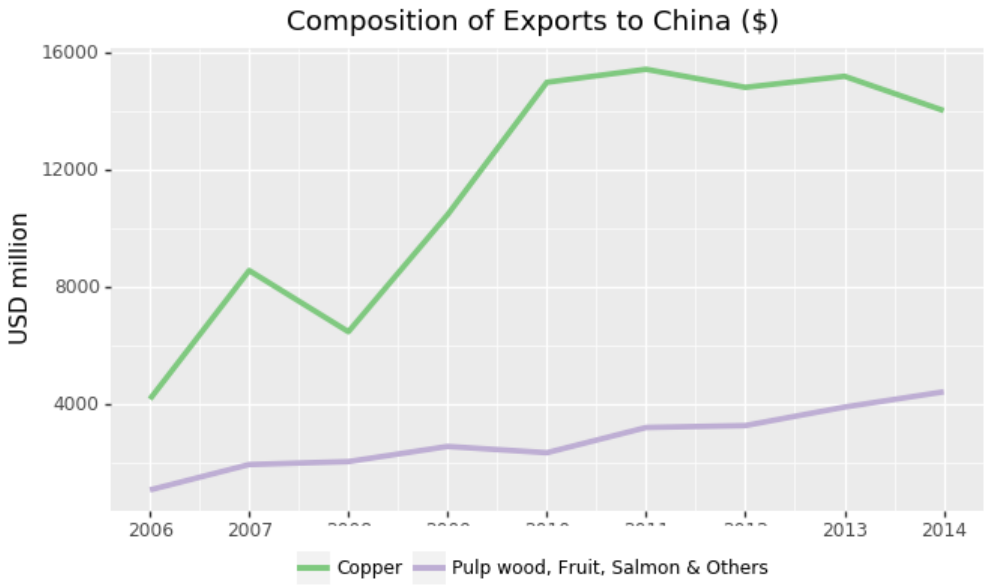
To adjust the position of the legend from the default spot of right of the graph, we add the `theme` option and specify the `legend_position="bottom"` argument. We can also change the legend shape using the `legend_direction="horizontal"` argument. Finally, we can centre the legend title position using the argument `legend_title_align="center"`.

[4] `http://colorbrewer2.org/`
[5] `http://plotnine.readthedocs.io/en/stable/generated/plotnine.scales.scale_color_brewer.html#plotnine.scales.scale_color_brewer`

```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
    )
)
p1
```

Let's also get rid of the legend title. We can do this by adding `legend_title` with the argument `element_blank()` to `theme`. If we instead wanted to change the name of the title, we could add the argument `name="Product"` to the `scale_colour_brewer()` option. Note that you can also do this with the `scale_colour_manual()` option we used above.

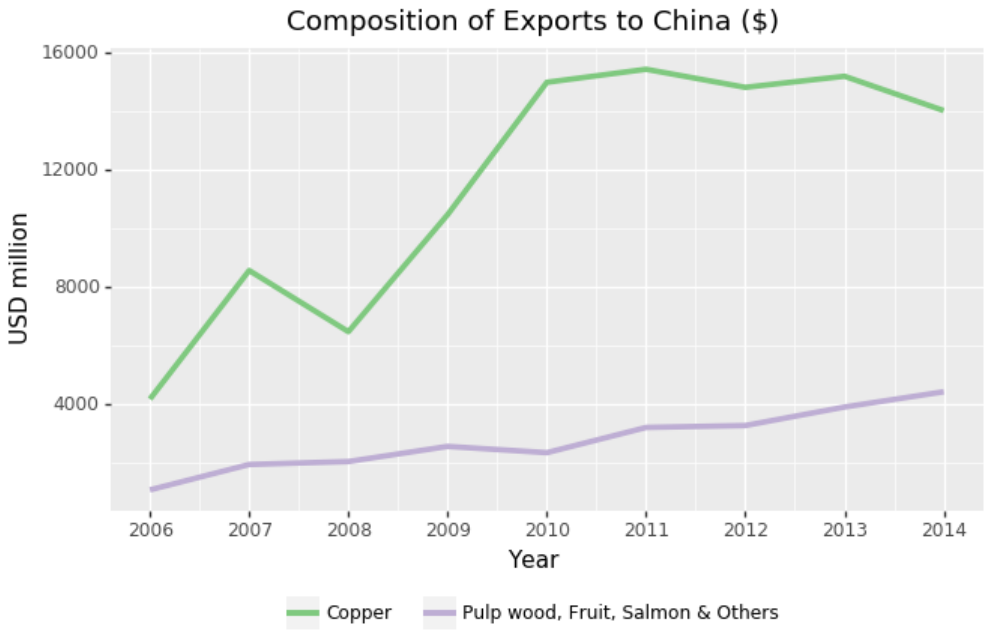
```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
    )
)
p1
```



Unfortunately, the legend is now sitting over our x-axis. We can fix this using the `legend_box_spacing=0.4` option within `theme`. This allows us to move the

legend down as much as needed. The legend is also looking a little squashed. We can space it out a bit more using the `legend_entry_spacing_x=15` argument.

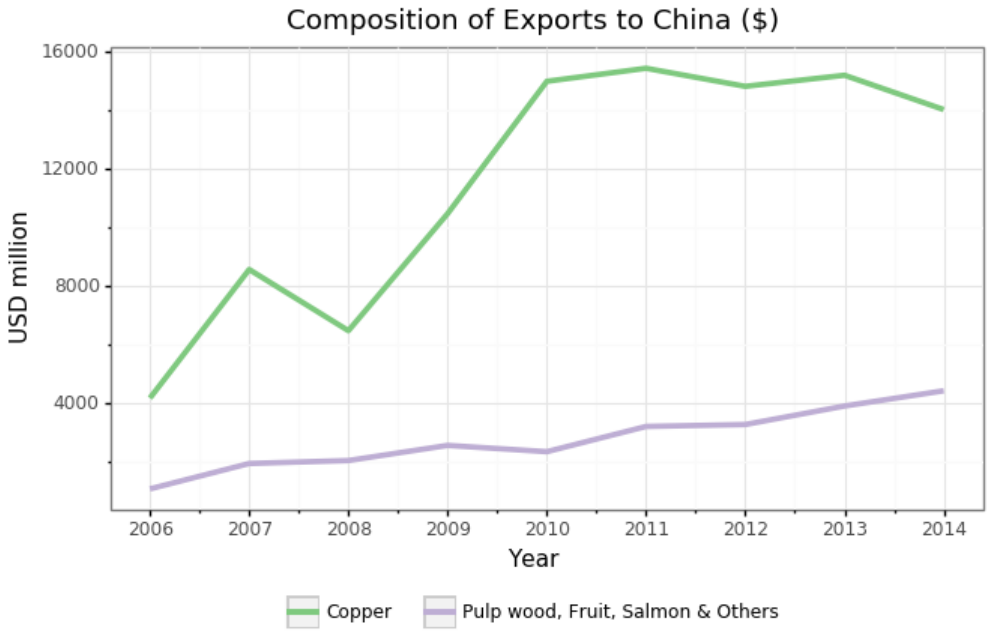
```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
    )
)
p1
```



1.10 Using the white theme

We can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()`.

```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_brewer(type="qual", palette="Accent")
    + theme_bw()
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
    )
)
p1
```



1.11 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[6]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[7]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

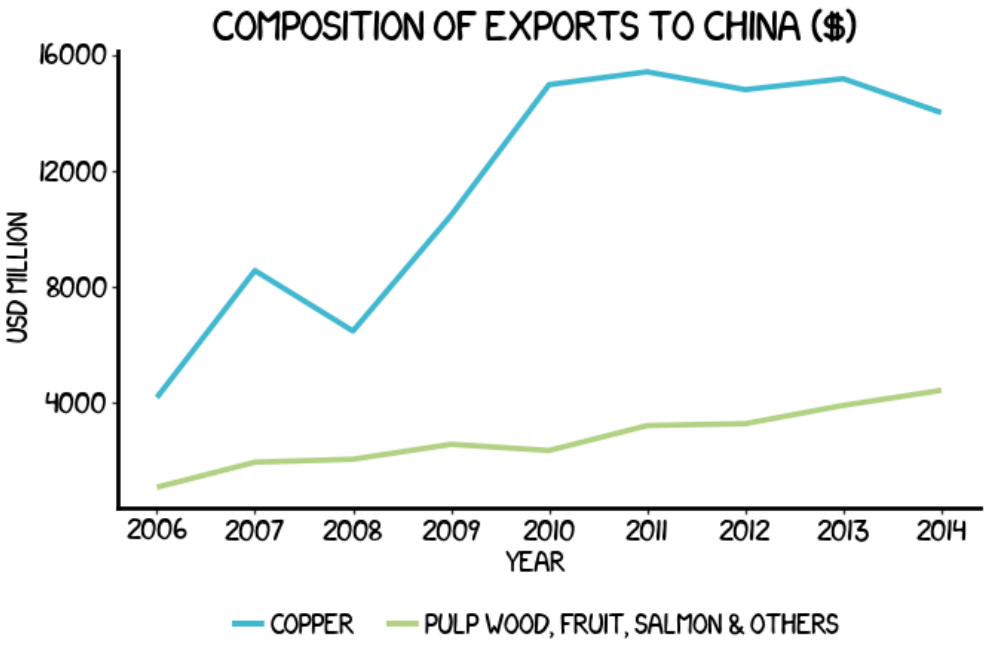
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;

[6] `xkcd.com/1350/xkcd-Regular.otf`
[7] `https://matplotlib.org/api/font_manager_api.html`

- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_manual(["#40b8d0", "#b2d183"])
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.5,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        legend_key=element_blank(),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p1
```



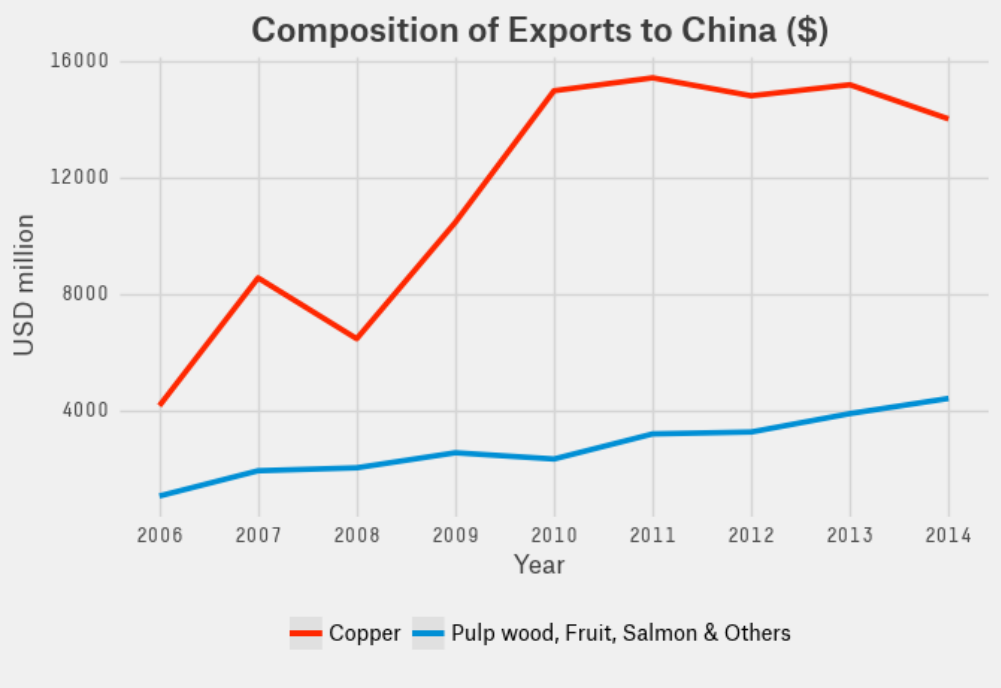
1.12 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[8]). Below we've applied `theme_538()`, which approximates graphs in the nice FiveThirtyEight website. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[9] and 'Decima Mono Pro'[10] in `legend_text`, `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
legend_text = fm.FontProperties(fname=agr)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)
```

[8] http://plotnine.readthedocs.io/en/stable/api.html#themes
[9] https://commercialtype.com/catalog/atlas/atlas_grotesk
[10] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
# Alter size and weight of font objects
title_text.set_size(16)
legend_text.set_size(10)
axis_text.set_size(12)
body_text.set_size(10)

p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_manual(["#FF2700", "#008FD5"])
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.5,
        legend_title=element_blank(),
        legend_text=element_text(fontproperties=legend_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p1
```



1.13 Creating your own theme

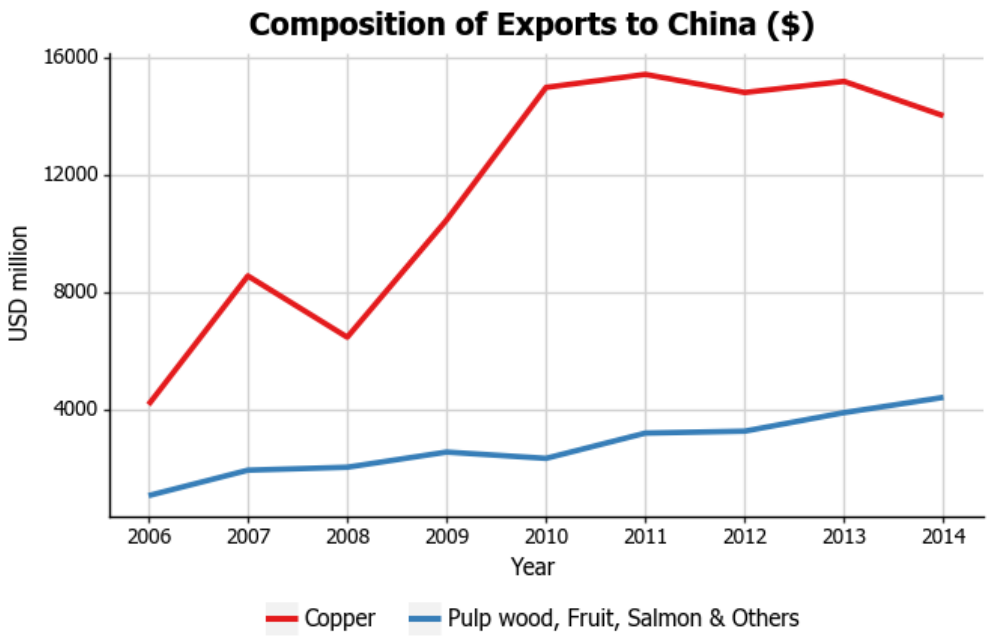
Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

We've also changed our colour scheme to another ColorBrewer theme, the quantitative scale `Set1`.

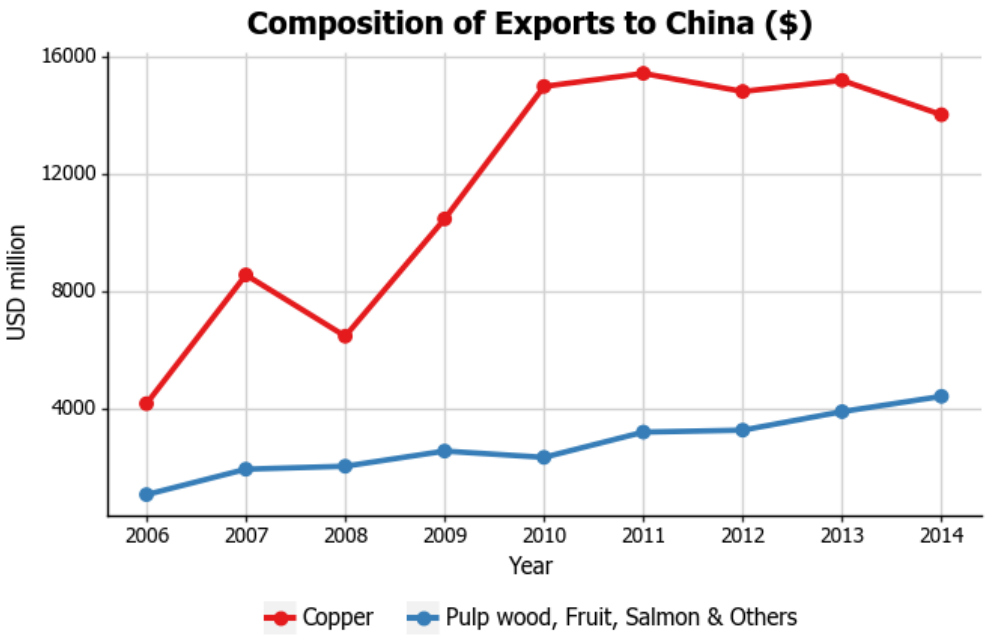
```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
```

```
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_brewer(type="qual", palette="Set1")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p1
```



Finally, to add points to create a marked line we add the `geom_point(size=3)` geom.

```
p1 = (
    ggplot(copper, aes("year", "export", colour="product"))
    + geom_line(size=1.5)
    + geom_point(size=3)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_colour_brewer(type="qual", palette="Set1")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p1
```



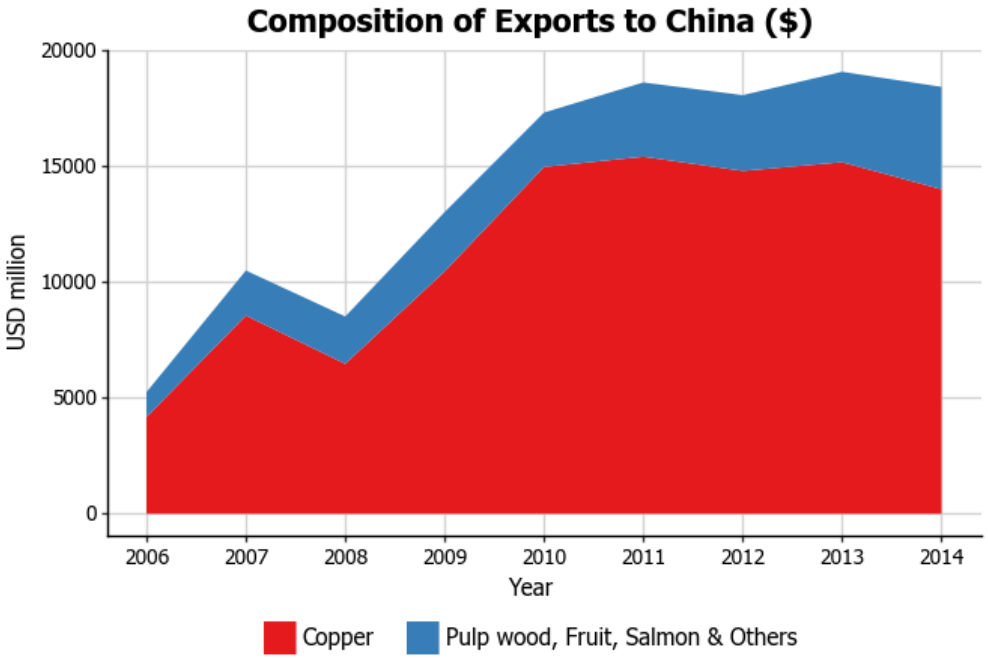
With all of these customisations, we now finally have the graph we presented at the beginning of this chapter.

Chapter 2

Area plots

2.1 Introduction

In this chapter, we will work towards creating the area plot below. We will take you from a basic area plot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import pandas as pd
import numpy as np

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from pandas import DataFrame
```

For this chapter, we'll be using a trade dataset put together by the book's authors. The exact sample we've used in this chapter can be downloaded from here[1]. You can load the data into Python like so:

```
copper = pd.read_csv("https://git.io/JecIS")
```

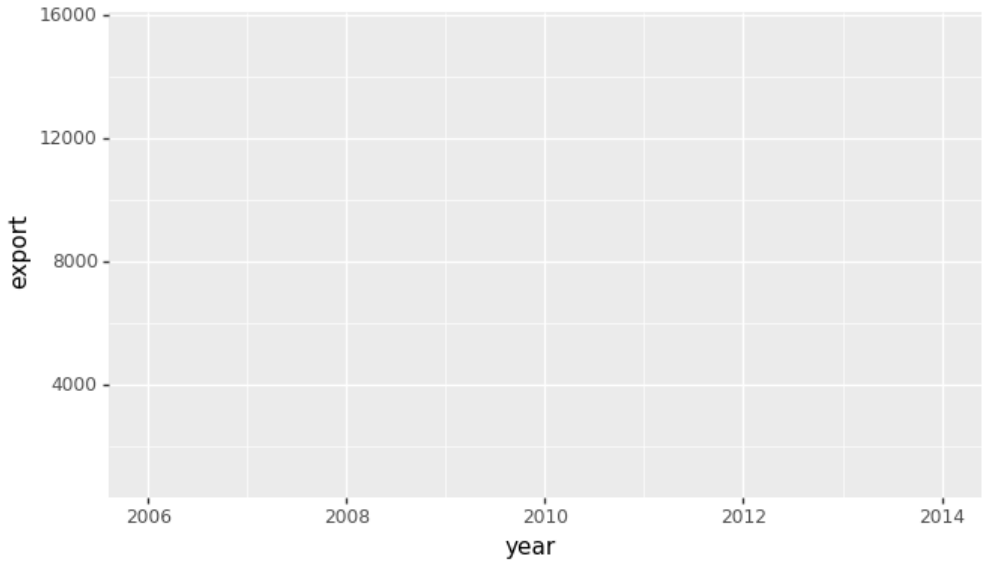
2.2 Basic ggplot structure

In order to initialise an area plot we tell ggplot that `copper` is our data, and specify that our x-axis plots the `year` variable, our y-axis plots the `export` variable, and our grouping variable is called `product`. You may have noticed that we put our variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `year` to the x-axis and `export` to the y-axis. We'll see how it handles `product` in a minute.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell `ggplot` how we want to visually represent it.

```
p2 = ggplot(copper, aes("year", "export", fill="product"))
p2
```

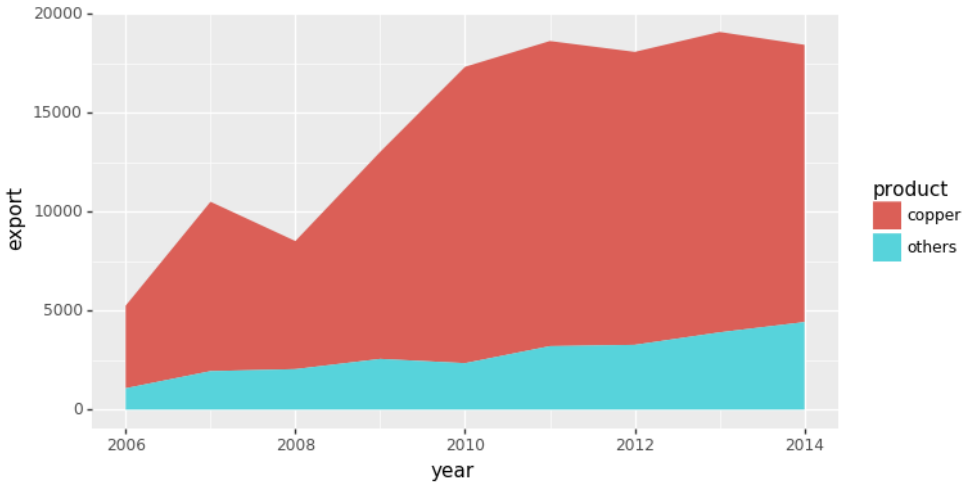
[1] `https://git.io/JecIS`



2.3 Basic area plot

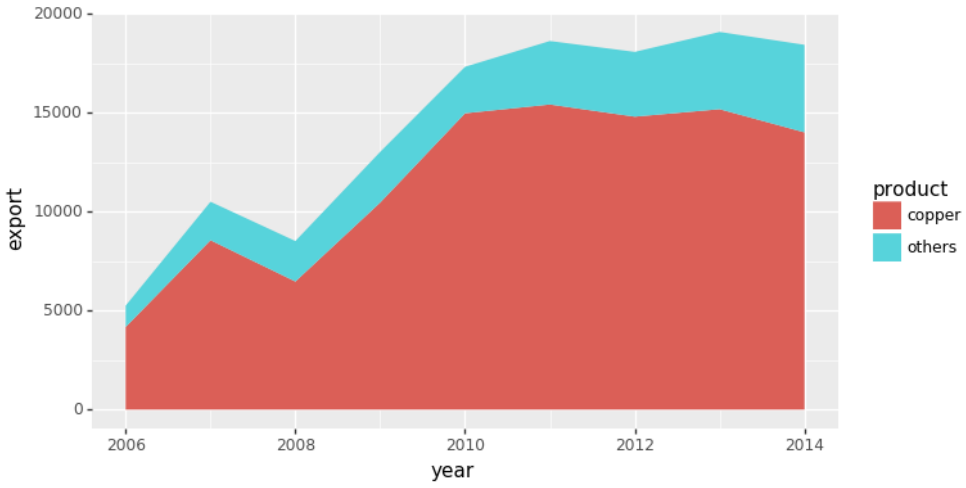
We can do this using `geoms`. In the case of an area plot, we use the `geom_area()` geom.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area()
)
p2
```



From now and ongoing we will stack in the opposite order. Adding the argument `position=position_stack(reverse=True)` to `geom_area` allows us to do that.

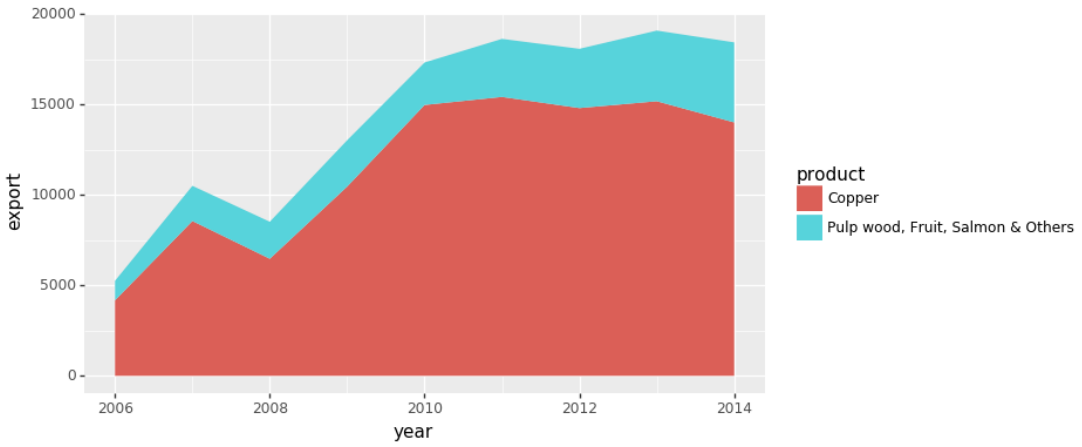
```
p2 = (
    ggplot(copper, aes("year", "export", fill="product")) +
      geom_area(position=position_stack(reverse=True))
)
p2
```



2.4 Changing group labels

To change the names of the groups in the legend, we need to rename the levels of the `product` variable. We can do this by creating a dictionary called `prod_name`, create another dictionary within *that*, and put in the values we want to change as key-value pairs. We then pass this dictionary to the `replace` method, making sure to set the `inplace` argument to `True` to make sure we overwrite the previous values of the column.

```
prod_name = {
    "product": {
        "copper": "Copper",
        "others": "Pulp wood, Fruit, Salmon & Others"
    }
}
copper.replace(prod_name, inplace=True)

p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
)
p2
```

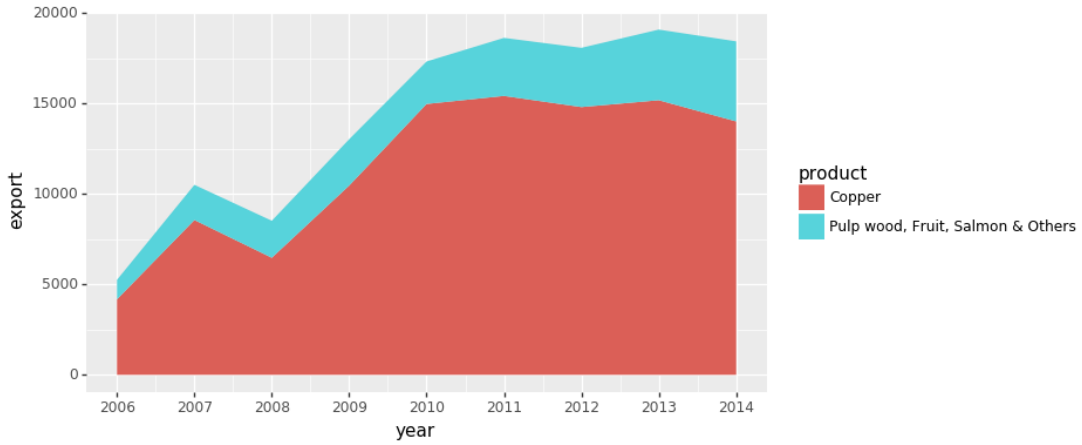


2.5 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, if we also wanted to change the y-axis we could use the `scale_y_continuous` option. Here we will change the x-axis to every year, rather than every 2 years. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[2] which generates a sequence from your selected start, stop and step values respectively. Note that because of Python's indexing, you need to set the `stop` argument to be one number more than your desired maximum when using `np.arange`.

Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis (passed as a list), although we haven't done so in this plot.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
)
p2
```

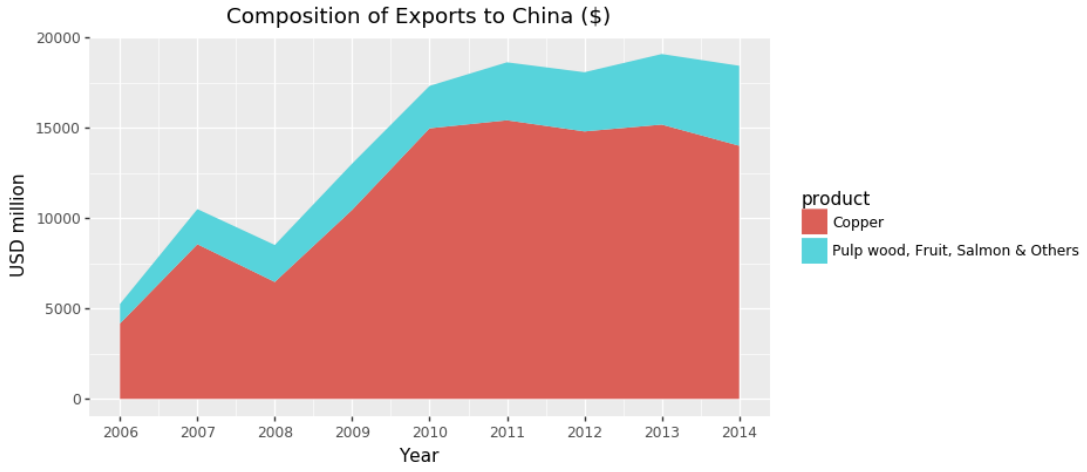


[2] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html`

2.6 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
)
p2
```

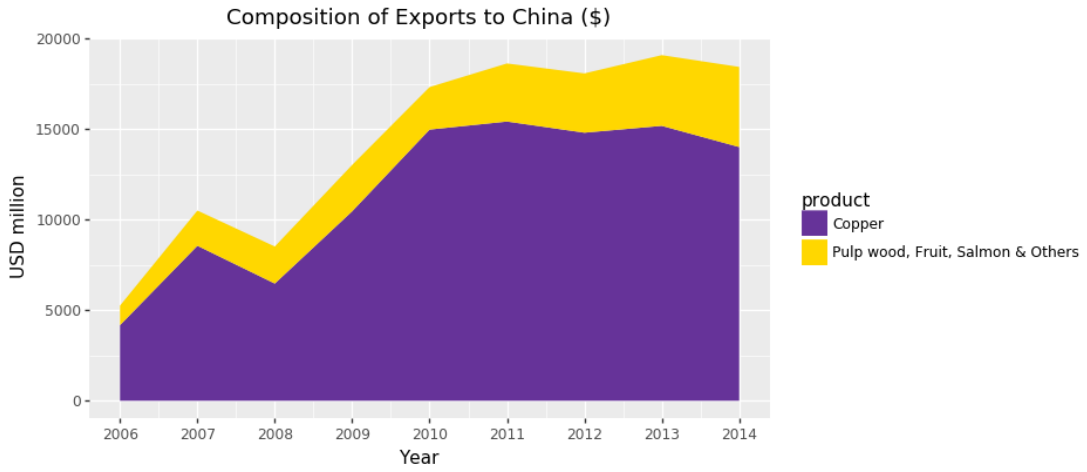


2.7 Adjusting the colour palette

There are a few options for adjusting the colour. The most simple is to change each level to a named colour using `scale_fill_manual`. `plotnine` uses the colour palette utilised by `matplotlib`, and the full set of named colours recognised by `ggplot` is here[3]. Let's try changing our areas to `rebeccapurple` and `gold`.

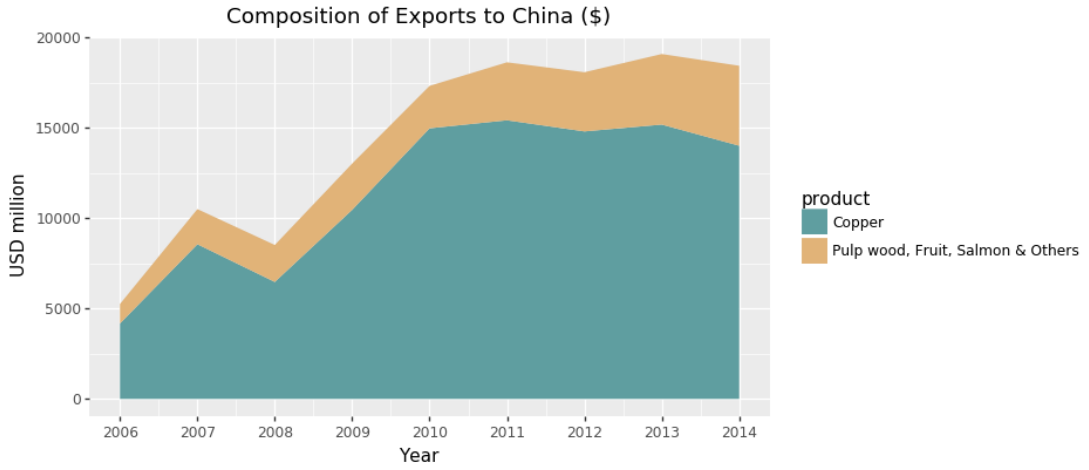
[3] `https://matplotlib.org/examples/color/named_colors.html`

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["rebeccapurple", "gold"])
)
p2
```



We can also change the colours using specific HEX codes. Let's change the areas to `#5F9EA0` and `#E1B378`.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["#5F9EA0", "#E1B378"])
)
p2
```

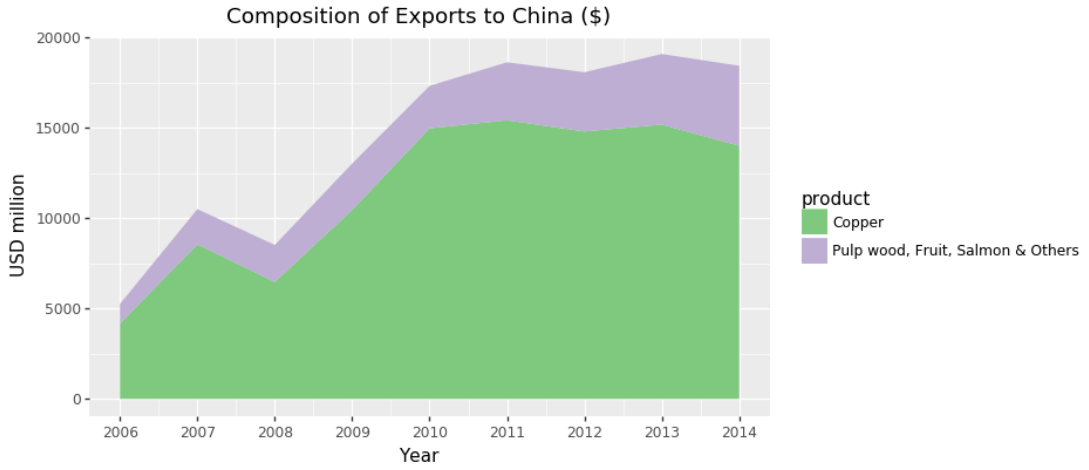


An alternative to using manual colours is to use the schemes from ColorBrewer[4]. Here we have used the `scale_colour_brewer` option with the qualitative scale `Accent`. More information on using `scale_colour_brewer` is here[5].

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
)
p2
```

[4] http://colorbrewer2.org/

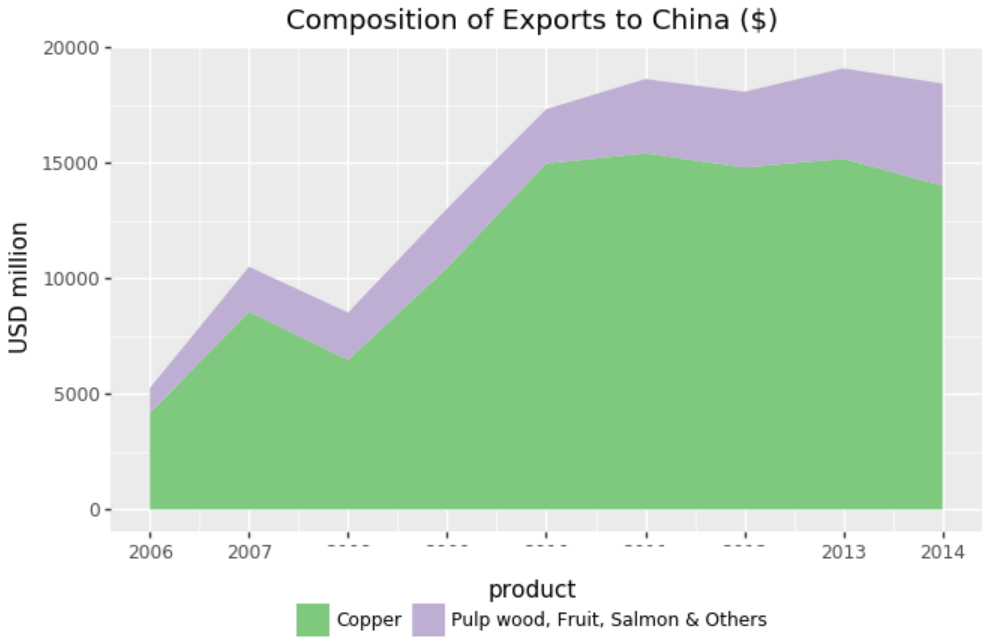
[5] http://plotnine.readthedocs.io/en/stable/generated/plotnine.scales.scale_color_brewer.html#plotnine.scales.scale_color_brewer



2.8 Adjusting the legend

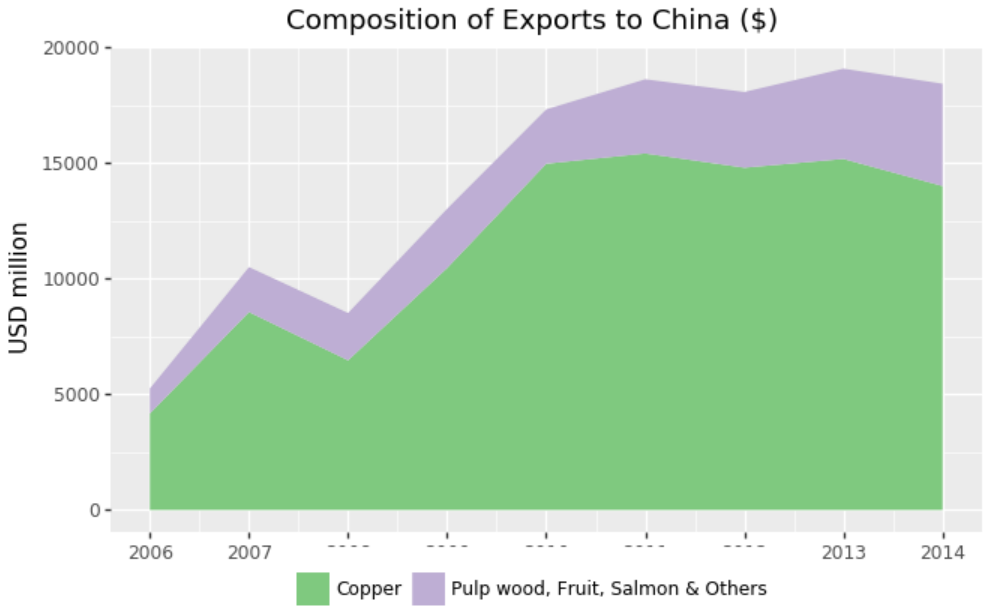
To adjust the position of the legend from the default spot of right of the graph, we add the `theme` option and specify the `legend_position="bottom"` argument. We can also change the legend shape using the `legend_direction="horizontal"` argument. Finally, we can centre the legend title position using the argument `legend_title_align="center"`.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
    )
)
p2
```



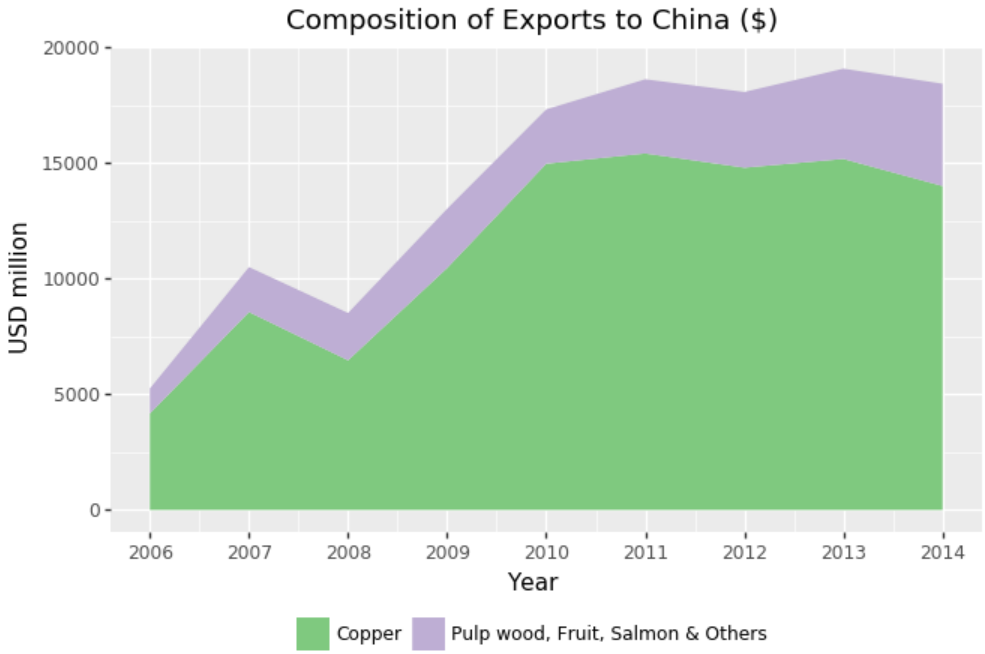
Let's also get rid of the legend title. We can do this by adding `legend_title` with the argument `element_blank()` to `theme`. If we instead wanted to change the name of the title, we could add the argument `name = 'Product'` to the `scale_colour_brewer()` option. Note that you can also do this with the `scale_colour_manual()` option we used above.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
    )
)
p2
```



Unfortunately, the legend is now sitting over our x-axis. We can fix this using the `legend_box_spacing=0.4` option within `theme`. This allows us to move the legend down as much as needed.

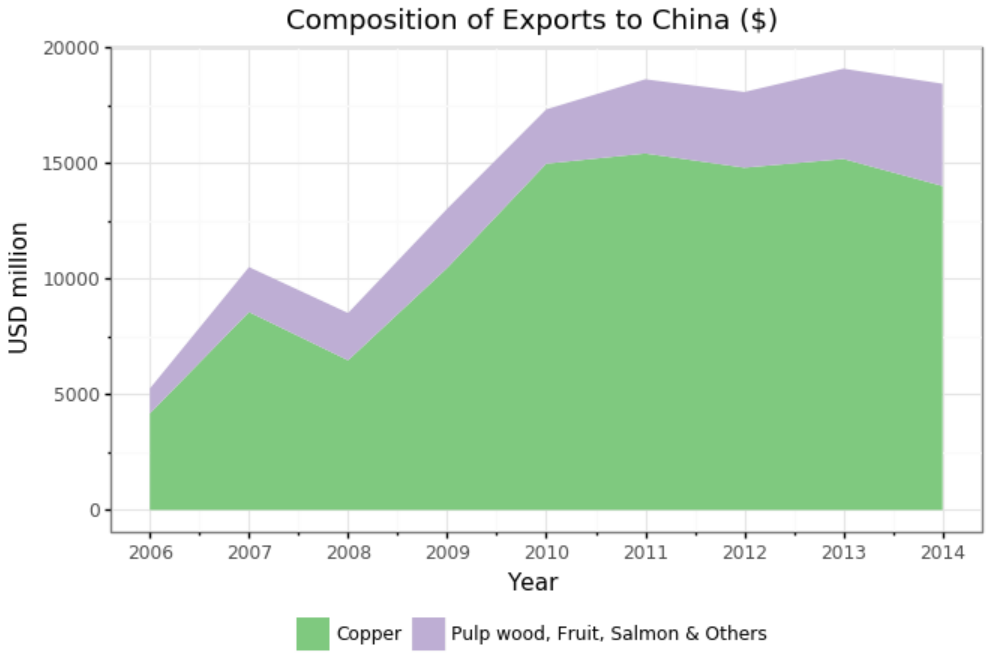
```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
        legend_box_spacing=0.4,
    )
)
p2
```



2.9 Using the white theme

We can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after `ggplot()`.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year") + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme_bw()
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
        legend_box_spacing=0.4,
    )
)
p2
```



2.10 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[6]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we

[6] `xkcd.com/1350/xkcd-Regular.otf`

want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[7]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

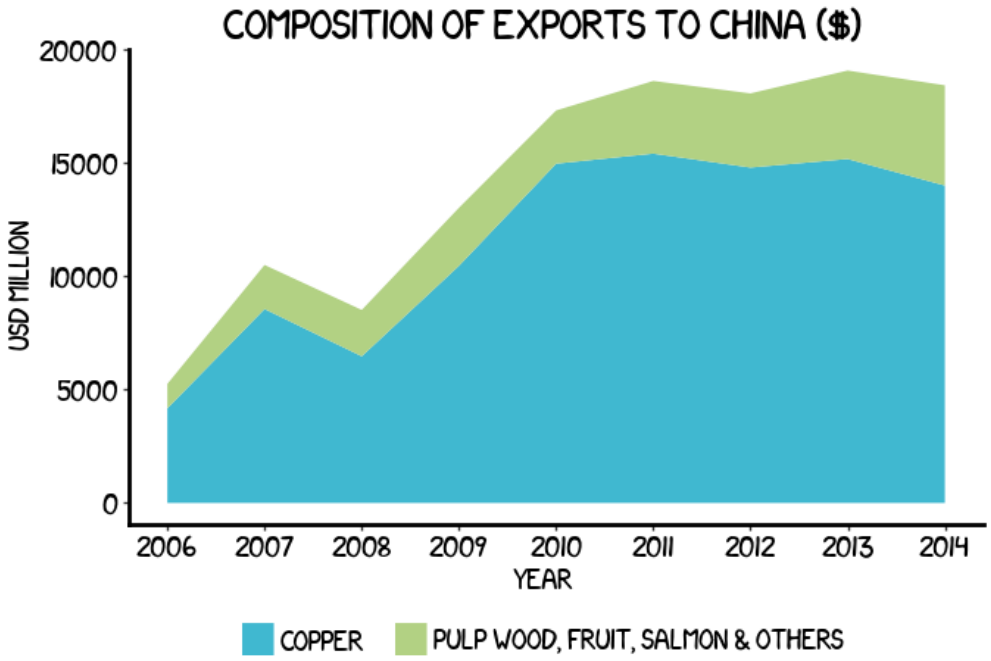
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["#40b8d0", "#b2d183"])
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
```

[7] `https://matplotlib.org/api/font_manager_api.html`

```
        legend_title_align="center",
        legend_box_spacing=0.5,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        legend_key=element_blank(),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p2
```

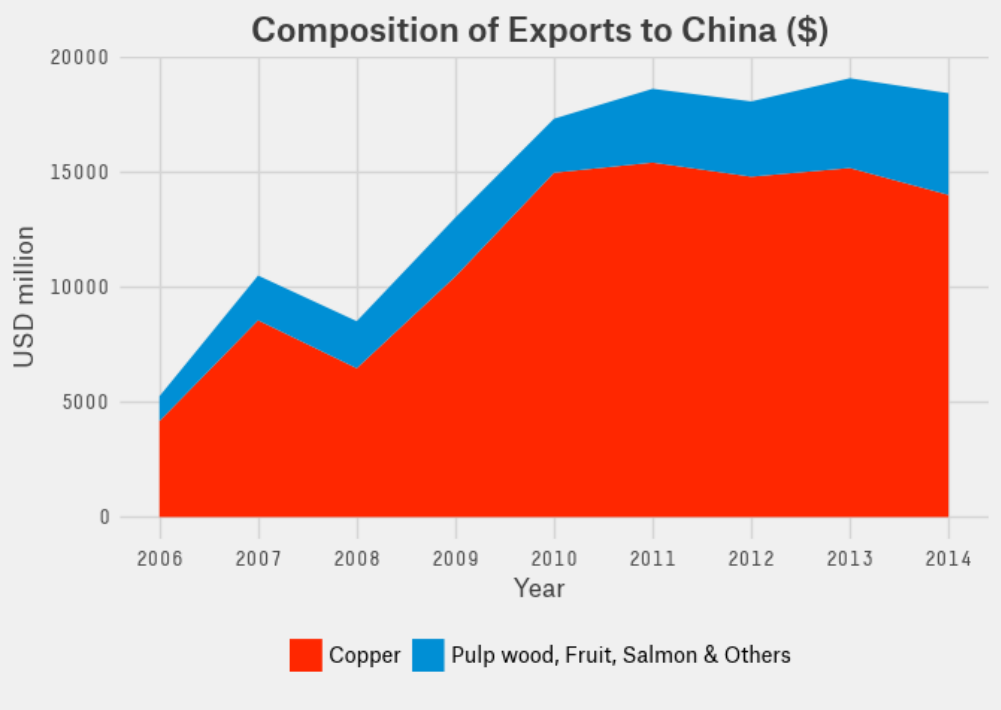


2.11 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[8]). Below we've applied `theme_538()`, which approximates graphs in the nice FiveThirtyEight website. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[9] and 'Decima Mono Pro'[10] in `legend_text`, `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
legend_text = fm.FontProperties(fname=agr)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
legend_text.set_size(10)
axis_text.set_size(12)
body_text.set_size(10)


p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year") + ylab("USD million")
    + scale_fill_manual(["#FF2700", "#008FD5"])
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        legend_position="bottom", legend_direction="horizontal",
        legend_box_spacing=0.5, legend_title=element_blank(),
        legend_text=element_text(fontproperties=legend_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p2
```

[8] http://plotnine.readthedocs.io/en/stable/api.html#themes
[9] https://commercialtype.com/catalog/atlas/atlas_grotesk
[10] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/



2.12 Creating your own theme

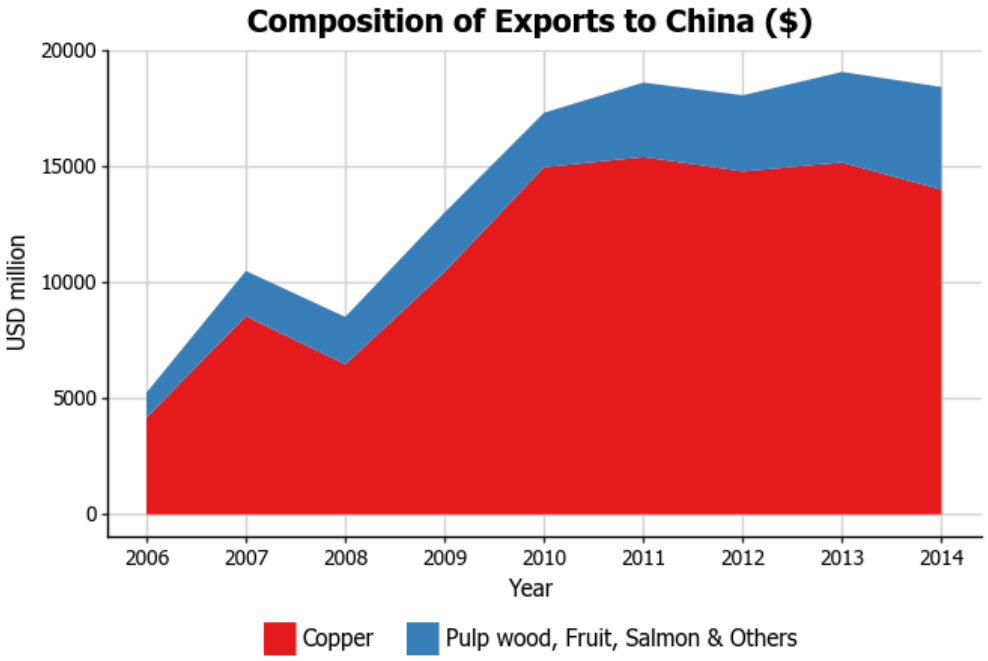
Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

We've also changed our colour scheme to another ColorBrewer theme, the quantitative scale `Set1`.

With all of these customisations, we now finally have the graph we presented at the beginning of this chapter.

```
p2 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_area(position=position_stack(reverse=True))
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Set1")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p2
```

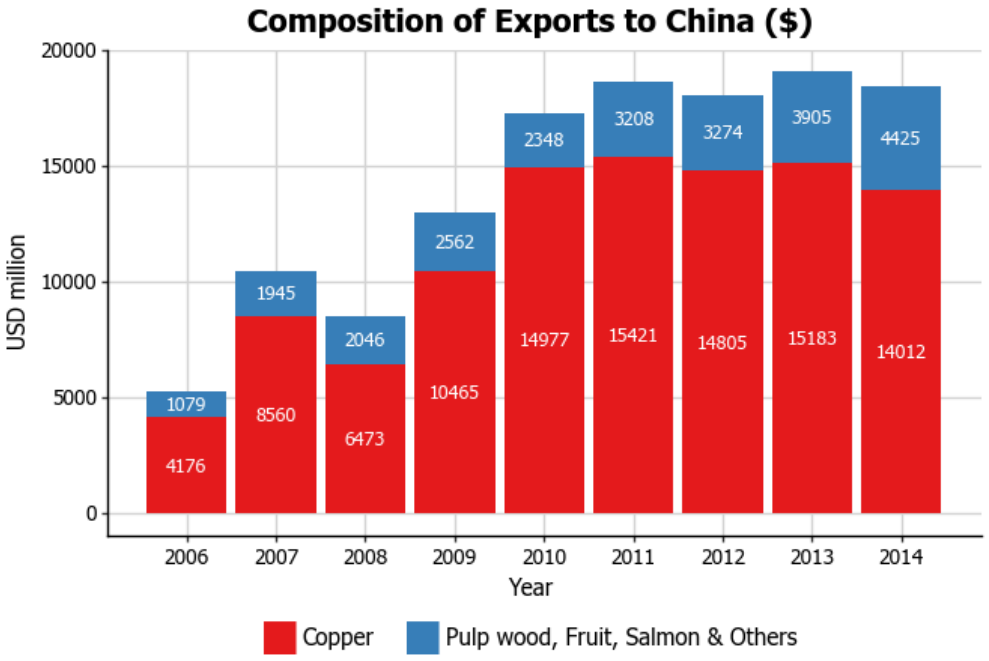

Composition of Exports to China ($)
20000
15000
10000
5000
0
USD million
2006
2007
2008
2009
2010
2011
2012
2013
2014
Year
Copper
Pulp wood, Fruit, Salmon & Others

Chapter 3

Bar plots

3.1 Introduction

In this chapter, we will work towards creating the bar plot below. We will take you from a basic bar plot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from pandas import DataFrame
```

For this chapter, we'll be using a trade dataset put together by the book's authors. The exact sample we've used in this chapter can be downloaded from here[1]. You can load the data into Python like so:

```
copper = pd.read_csv("https://git.io/JecIS")
```

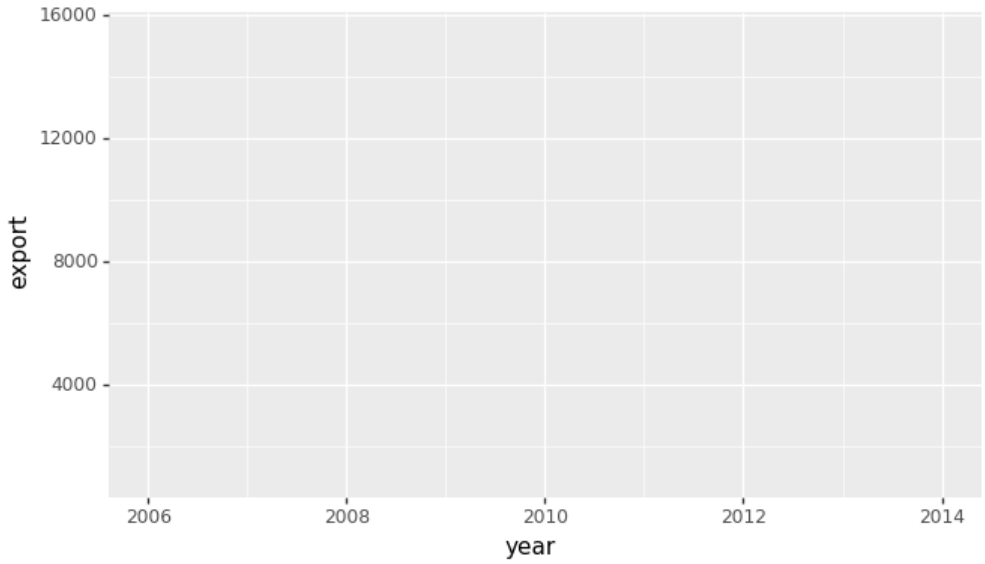
3.2 Basic ggplot structure

In order to initialise a bar plot we tell ggplot that `copper` is our data, and specify that our x-axis plots the `year` variable, our y-axis plots the `export` variable, and our grouping variable is called `product`. You may have noticed that we put our variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `year` to the x-axis and `export` to the y-axis. We'll see how it handles `product` in a minute.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

```
p3 = ggplot(copper, aes("year", "export", fill="product"))
p3
```

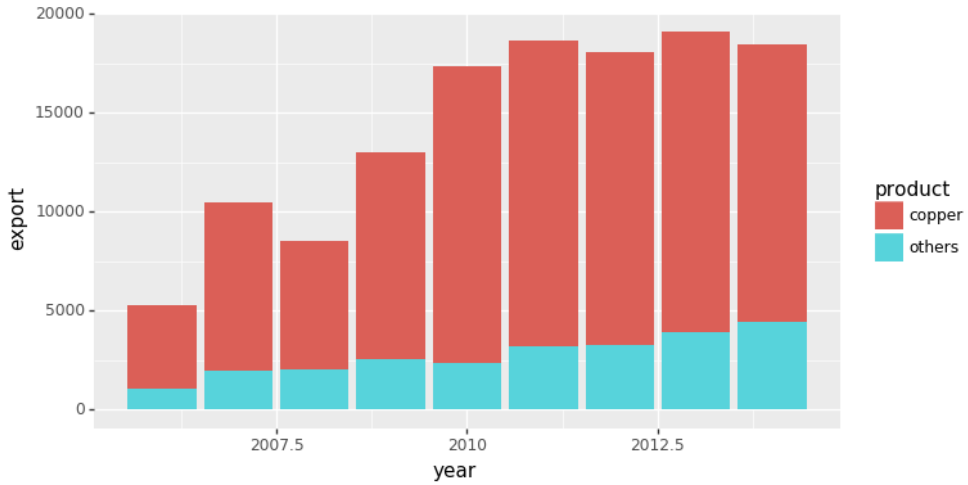
[1] `https://git.io/JecIS`



3.3 Basic bar plot

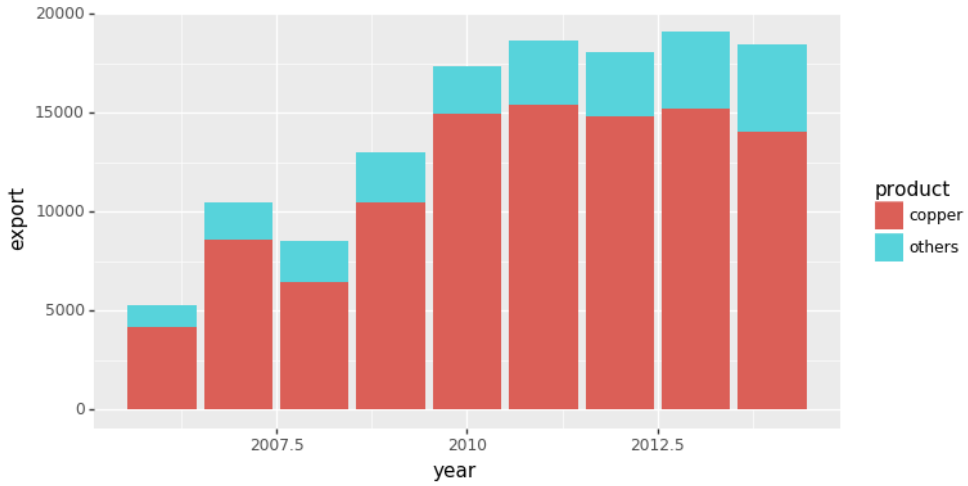
We can do this using `geoms`. In the case of a bar plot, we use the `geom_col()` geom. (We could have also used `geom_bar(stat = 'identity')`, but `geom_col()` is just a bit cleaner.)

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
      + geom_col()
)
p3
```



From now and ongoing we will stack in the opposite order. Adding the argument `position=position_stack(reverse=True)` to `geom_col()` allows us to do that.

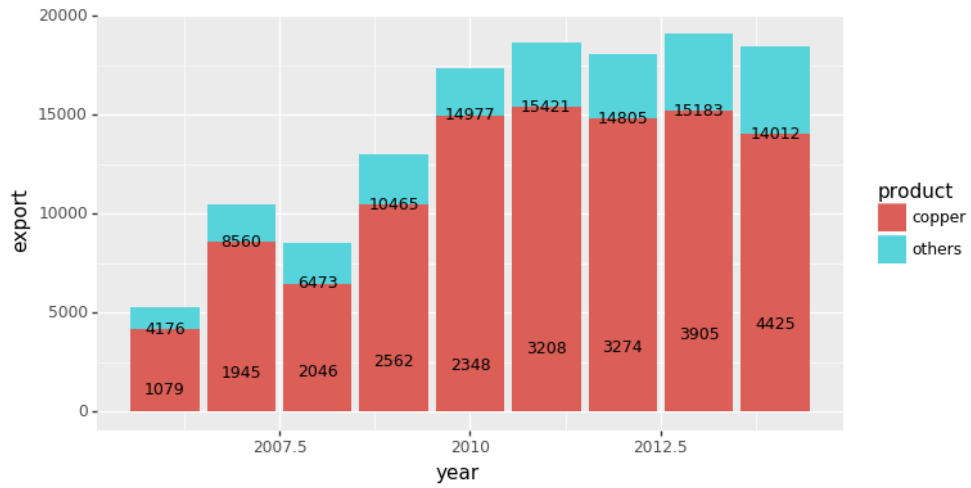
```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
)
p3
```



3.4 Adding data labels

To label the bars according to some variable in the data, we add another geom - `geom_text()` - and add `label="export"` to its `aes()` argument. We can also set the size of the labels in this geom.

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(label="export"), size=9)
)
p3
```

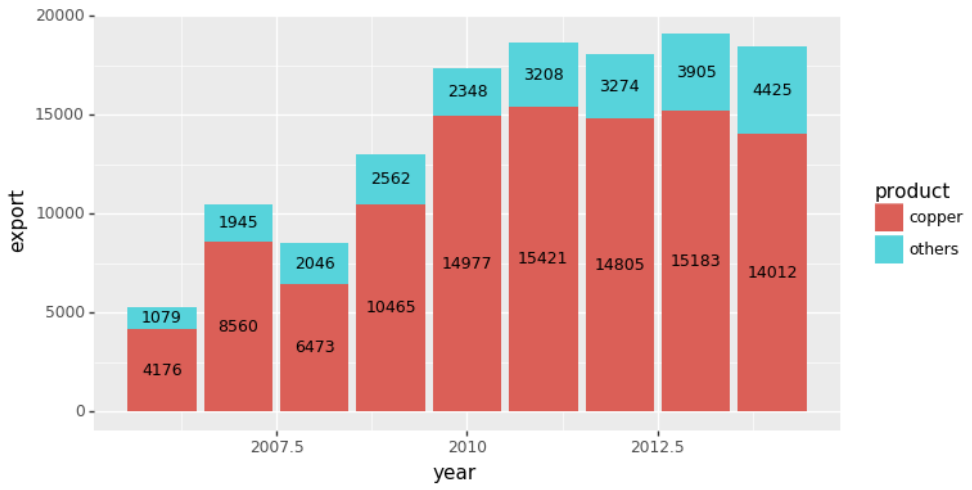


3.5 Adjusting data labels position

To adjust the position of the data labels from the default placement, we use both the `groupby` and `transform` functions on the data, and create a new variable called `pos`. This variable marks the position at the centre of each bar and can be used to specify the position of the labels by assigning it to the y-argument in `geom_text(aes())`.

```
copper["pos"] = copper[["year", "export"]].groupby("year").\
                transform(pd.Series.cumsum)
copper["pos"] = copper["pos"] - 0.5 * copper["export"]
```

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
)
p3
```

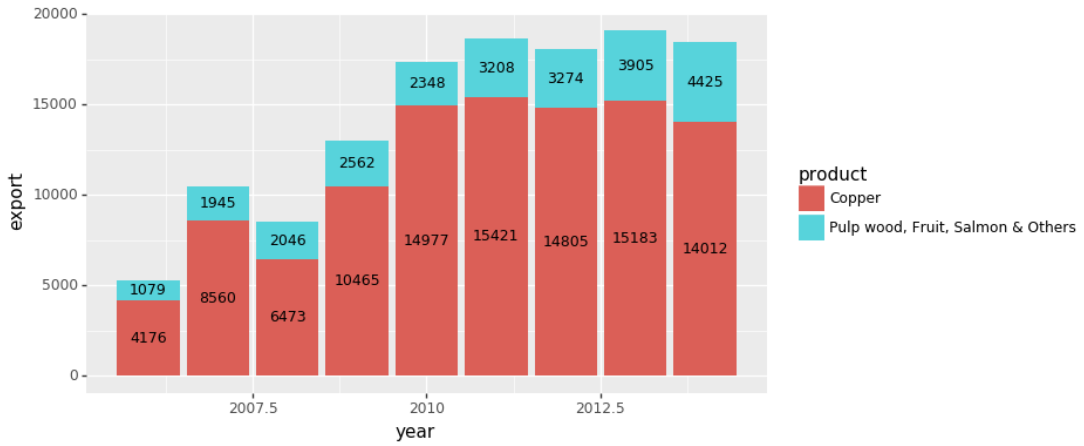


3.6 Changing group labels

To change the names of the groups in the legend, we need to rename the levels of the `product` variable. We can do this by creating a dictionary called `prod_name`, create another dictionary within *that*, and put in the values we want to change as key-value pairs. We then pass this dictionary to the `replace` method, making sure to set the `inplace` argument to `True` to make sure we overwrite the previous values of the column.

```
prod_name = {
    "product": {
        "copper": "Copper",
        "others": "Pulp wood, Fruit, Salmon & Others"
    }
}

copper.replace(prod_name, inplace=True)
```

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
)
p3
```



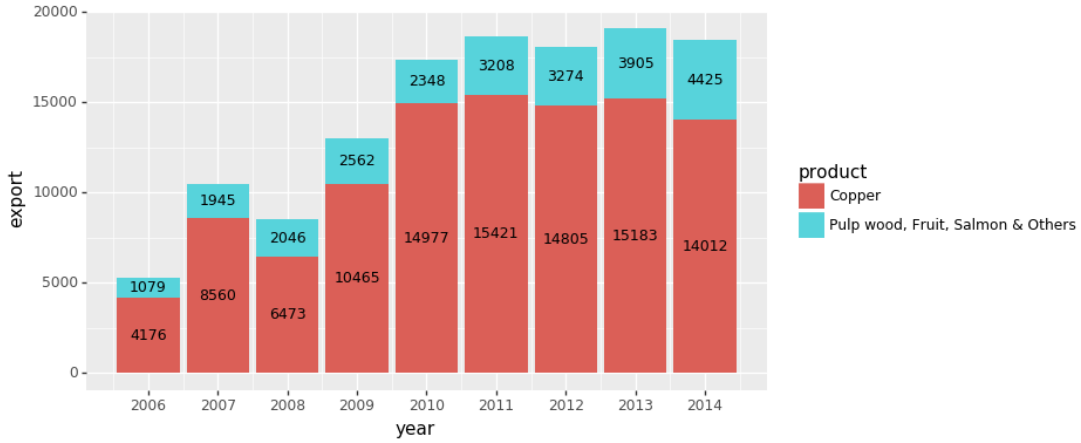
3.7 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, if we also wanted to change the y-axis we could use the `scale_y_continuous` option. Here we will change the x-axis to every year, rather than every 2 years. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[2] which generates a sequence from your selected start, stop and step values respectively. Note that because of Python's indexing, you need to set the `stop` argument to be one number more than your desired maximum when using `np.arange`.

Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis (passed as a list), although we haven't done so in this plot.

[2] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html`

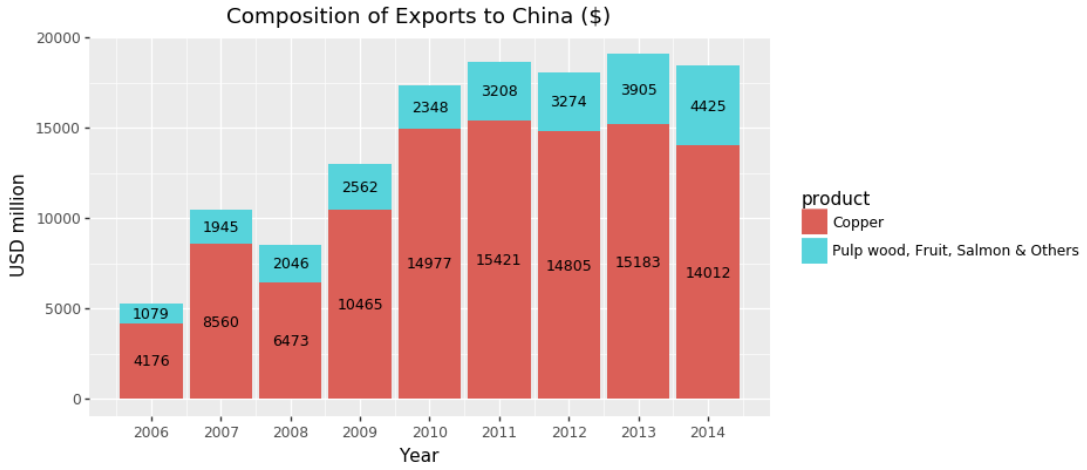
```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
)
p3
```



3.8 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
)
p3
```

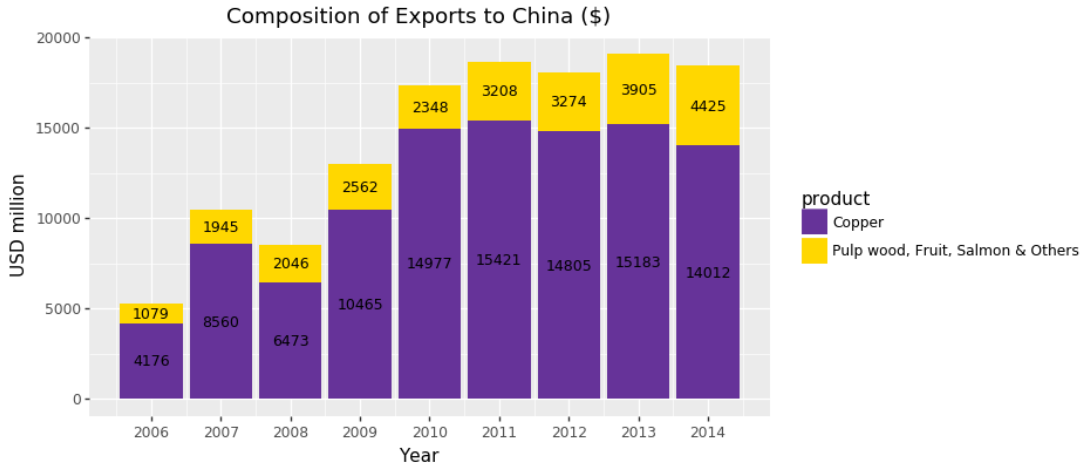


3.9 Adjusting the colour palette

There are a few options for adjusting the colour. The most simple is to change each level to a named colour using `scale_fill_manual`. `plotnine` uses the colour palette utilised by `matplotlib`, and the full set of named colours recognised by `ggplot` is here[3]. Let's try changing our bar areas to `rebeccapurple` and `gold`.

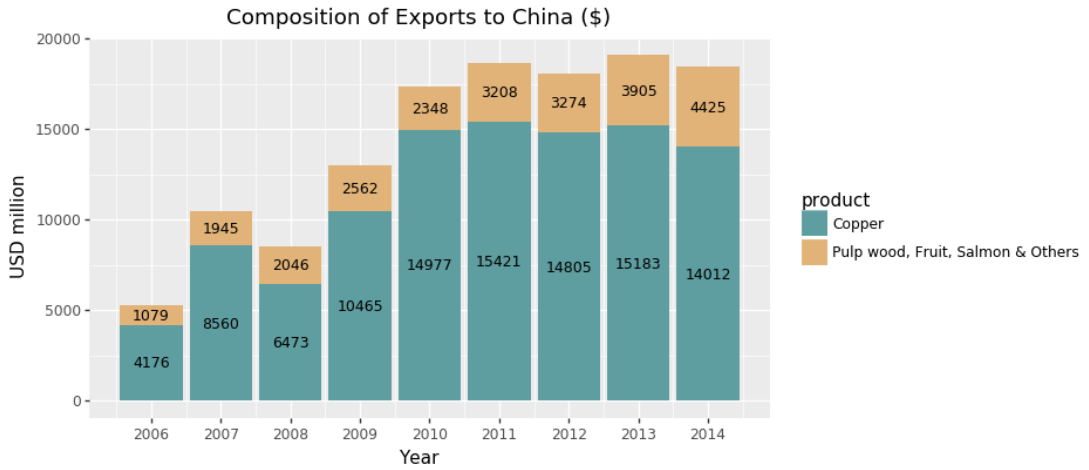
```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["rebeccapurple", "gold"])
)
p3
```

[3] https://matplotlib.org/examples/color/named_colors.html



We can also change the colours using specific HEX codes. Let's change the areas to `#5F9EA0` and `#E1B378`.

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["#5F9EA0", "#E1B378"])
)
p3
```



An alternative to using manual colours is to use the schemes from ColorBrewer[4]. Here we have used the `scale_colour_brewer` option with the qualitative scale `Accent`. More information on using `scale_colour_brewer` is here[5].

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
)
p3
```

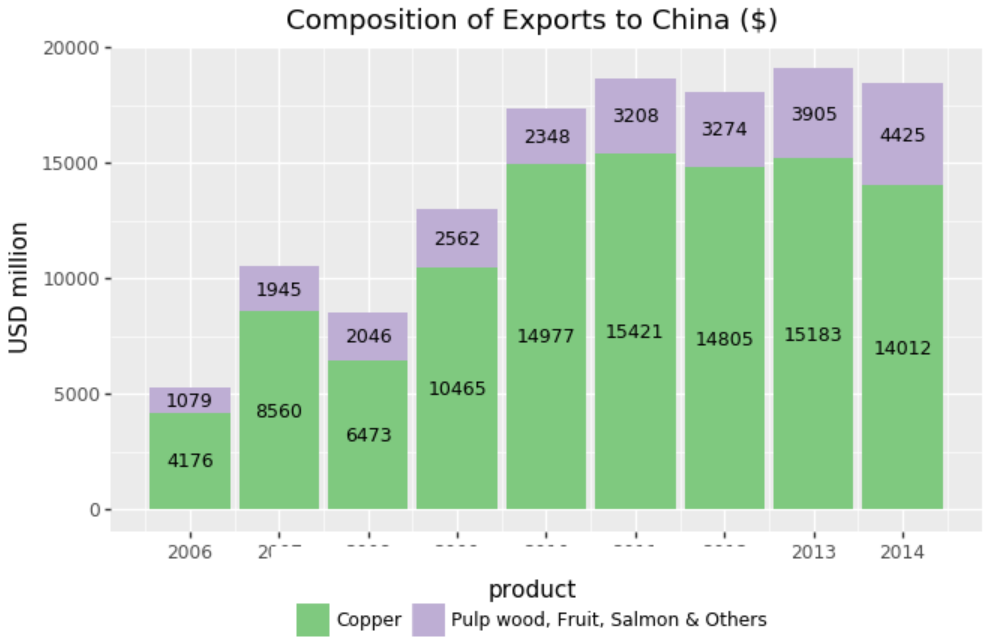
3.10 Adjusting the legend

To adjust the position of the legend from the default spot of right of the graph, we add the `theme` option and specify the `legend_position="bottom"` argument. We can also change the legend shape using the `legend_direction="horizontal"` argument. Finally, we can centre the legend title position using the argument `legend_title_align="center"`.

[4] `http://colorbrewer2.org/`

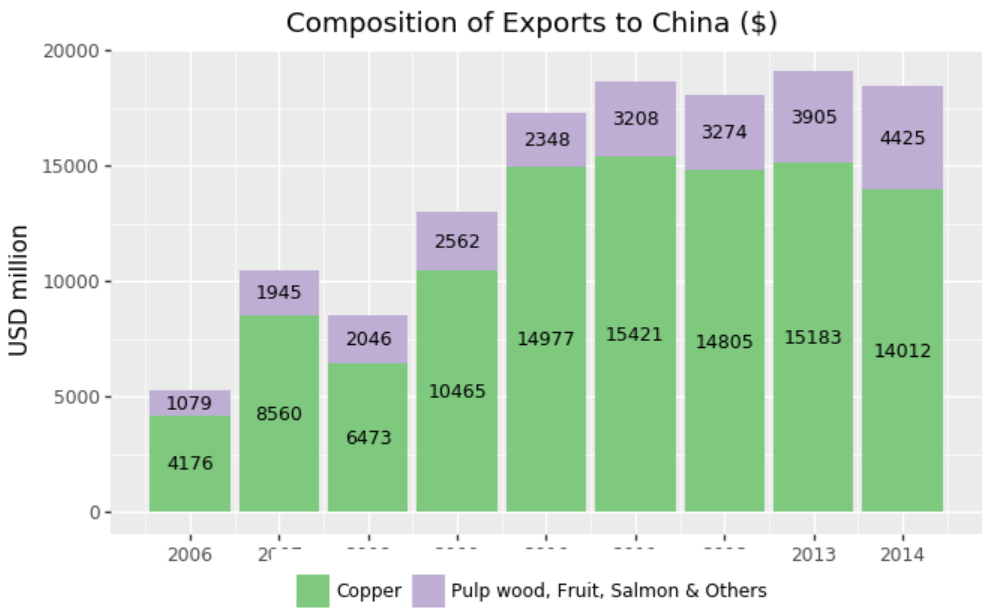
[5] `http://plotnine.readthedocs.io/en/stable/generated/plotnine.scales.scale_color_brewer.html#plotnine.scales.scale_color_brewer`

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
    )
)
p3
```



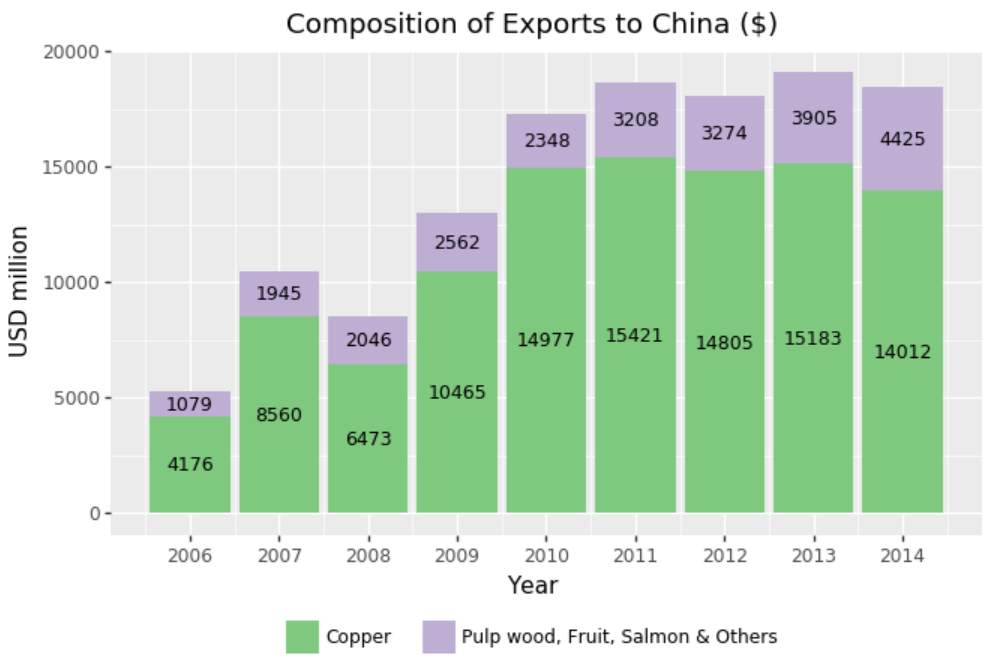
Let's also get rid of the legend title. We can do this by adding `legend_title` with the argument `element_blank()` to `theme`. If we instead wanted to change the name of the title, we could add the argument `name="Product"` to the `scale_fill_brewer()` option. Note that you can also do this with the `scale_fill_manual()` option we used above.

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
    )
)
p3
```



Unfortunately, the legend is now sitting over our x-axis. We can fix this using the `legend_box_spacing=0.4` option within `theme`. This allows us to move the legend down as much as needed. The legend is also looking a little squashed. We can space it out a bit more using the `legend_entry_spacing_x=15` argument.

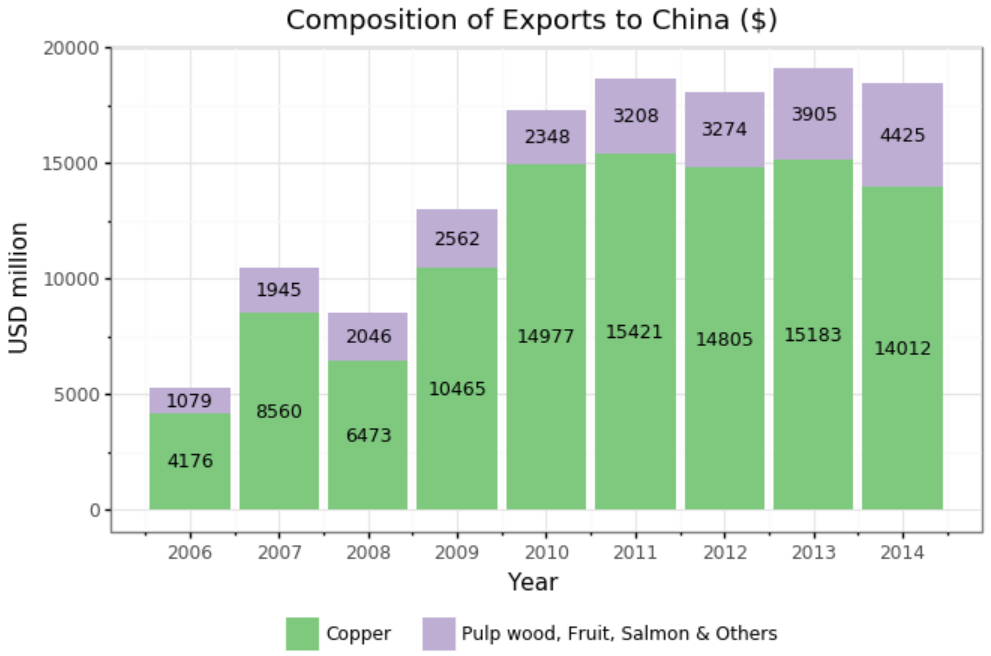
```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
    )
)
p3
```



3.11 Using the white theme

We can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after `ggplot()`.

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=9)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme_bw()
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
    )
)
p3
```



3.12 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[6]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we

[6] `xkcd.com/1350/xkcd-Regular.otf`

want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[7]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)
label_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

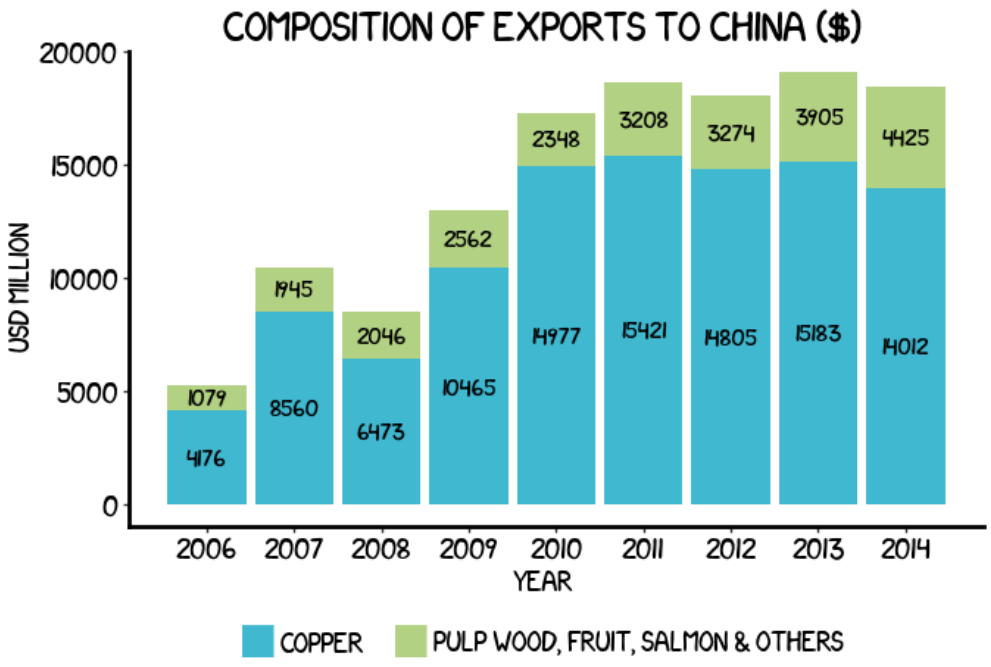
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- To change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`;
- Finally, in order to change the font of the labels within `geom_text`, we pass it the `family="xkcd"` argument. This will only work if you have the xkcd font installed on your computer.

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"),
                family="xkcd", size=10)
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
```

[7] https://matplotlib.org/api/font_manager_api.html

```
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["#40b8d0", "#b2d183"])
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.5,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        legend_key=element_blank(),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p3
```



3.13 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[8]). Below we've applied `theme_538()`, which approximates graphs in the FiveThirtyEight website. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[9] and 'Decima Mono Pro'[10] in `legend_text`, `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
legend_text = fm.FontProperties(fname=agr)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
legend_text.set_size(10)
axis_text.set_size(12)
body_text.set_size(10)


p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"), size=10,
                family="DecimaMonoPro")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year") + ylab("USD million")
    + scale_fill_manual(["#FF2700", "#008FD5"]) + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        legend_position="bottom", legend_direction="horizontal",
        legend_box_spacing=0.5, legend_title=element_blank(),
        legend_text=element_text(fontproperties=legend_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text))
)
p3
```

[8] http://plotnine.readthedocs.io/en/stable/api.html#themes
[9] https://commercialtype.com/catalog/atlas/atlas_grotesk
[10] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

3.14 Creating your own theme

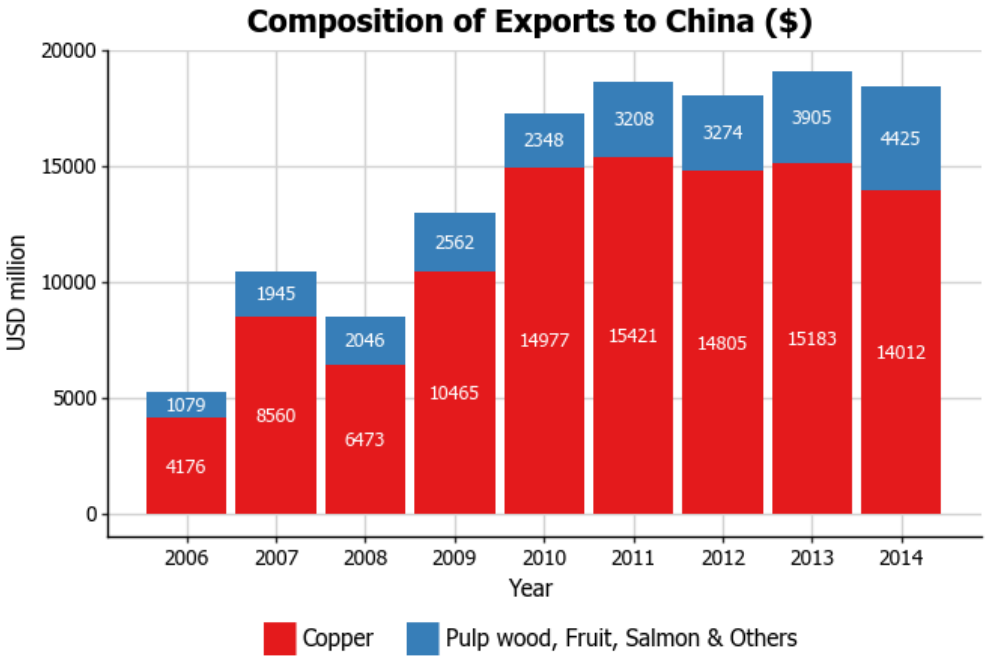
Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

We've also changed our colour scheme to another ColorBrewer theme, the quantitative scale `Set1`.

With all of these customisations, we now finally have the graph we presented at the beginning of this chapter.

```
p3 = (
    ggplot(copper, aes("year", "export", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(y="pos", label="export"),
                size=9, colour="white")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Set1")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p3
```

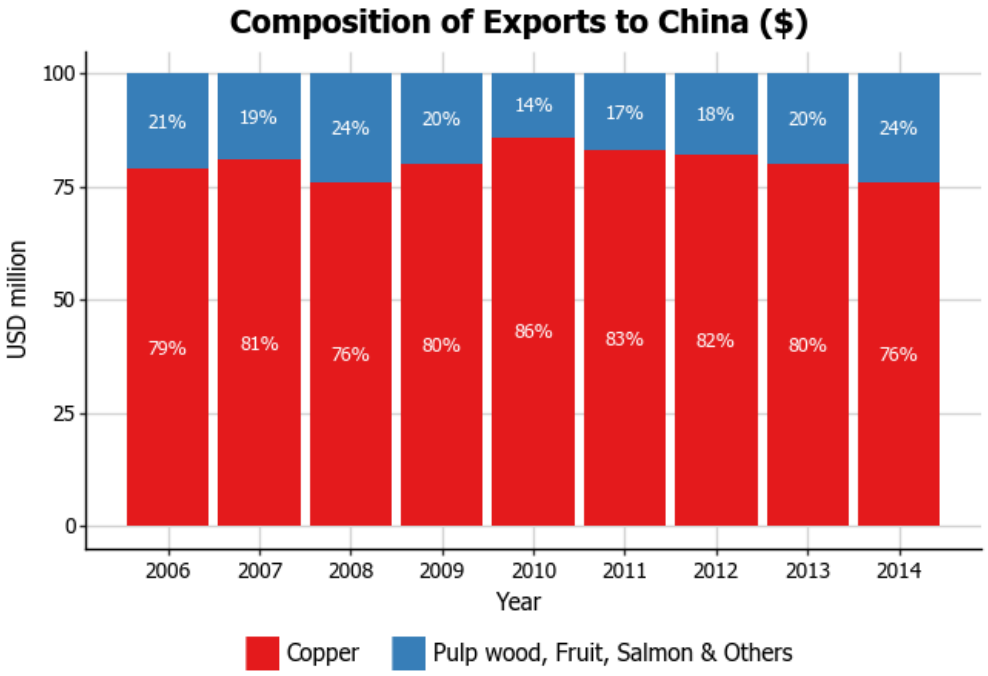

Composition of Exports to China ($)
USD million
20000
15000
10000
5000
0
2006
2007
2008
2009
2010
2011
2012
2013
2014
Year
1079
4176
1945
8560
2046
6473
2562
10465
2348
14977
3208
15421
3274
14805
3905
15183
4425
14012
Copper
Pulp wood, Fruit, Salmon & Others

Chapter 4

Stacked bar plots

4.1 Introduction

In this chapter, we will work towards creating the stacked bar plot below. We will take you from a basic stacked bar plot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from pandas import DataFrame
```

For this chapter, we'll be using a trade dataset put together by the book's authors. The exact sample we've used in this chapter can be downloaded from here[1]. You can load the data into Python like so:

```
copper = pd.read_csv("https://git.io/JecIS")
```

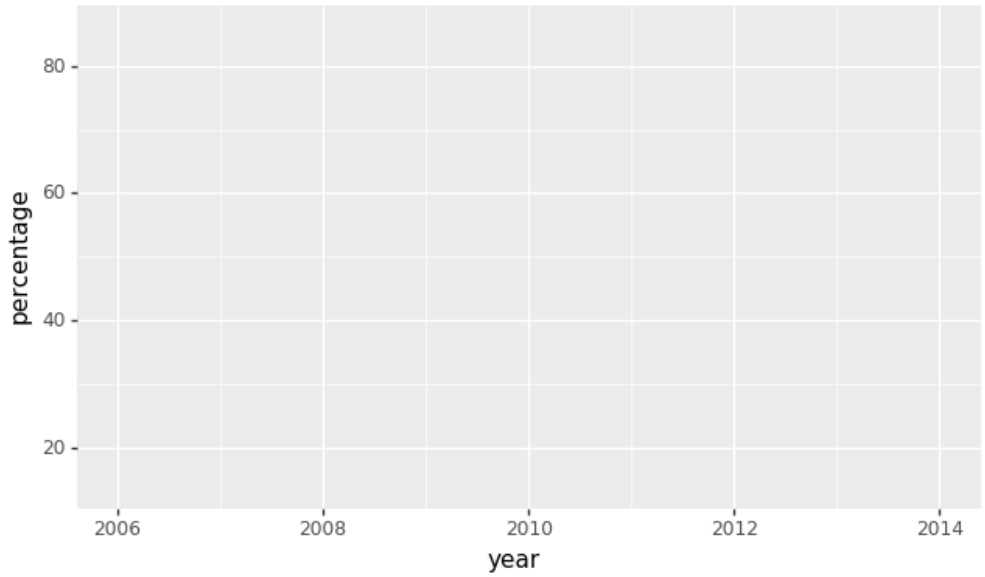
4.2 Basic ggplot structure

In order to initialise a stacked bar plot we tell `ggplot` that `copper` is our data, and specify that our x-axis plots the `year` variable, our y-axis plots the `percentage` variable, and our grouping variable is called `product`. You may have noticed that we put our variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `year` to the x-axis and `percentage` to the y-axis. We'll see how it handles `product` in a minute.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

[1] `https://git.io/JecIS`

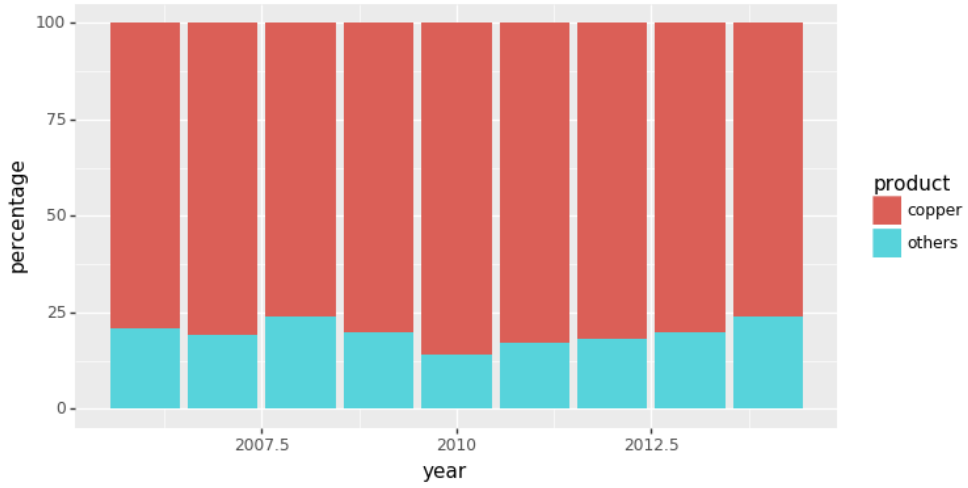
```
p4 = ggplot(copper, aes("year", "percentage", fill="product"))
p4
```



4.3 Basic stacked bar plot

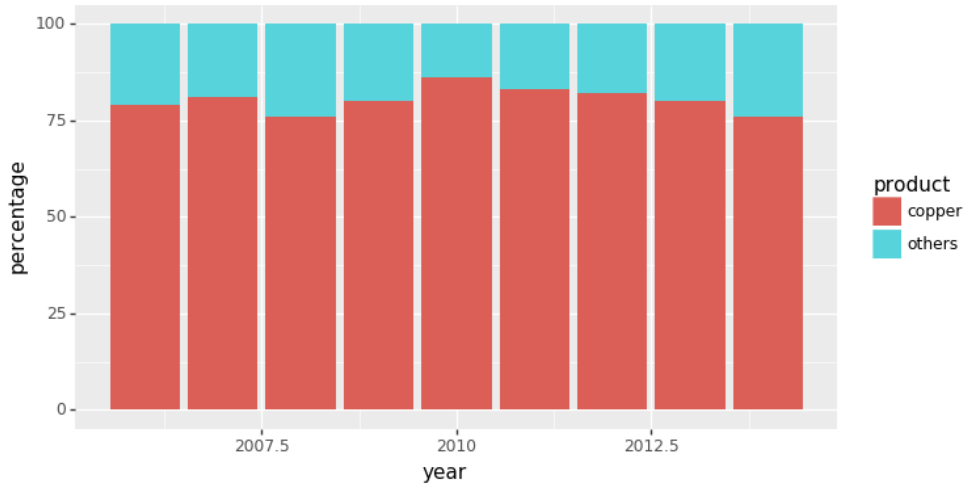
We can do this using `geoms`. In the case of a stacked bar plot (or any bar plot, really!), we use the `geom_col()` geom. (We could have also used `geom_bar(stat="identity")`, but `geom_col()` is just a bit cleaner.)

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col()
)
p4
```



From now and ongoing we will stack in the opposite order. Adding the argument `position=position_stack(reverse=True)` to `geom_col()` allows us to do that.

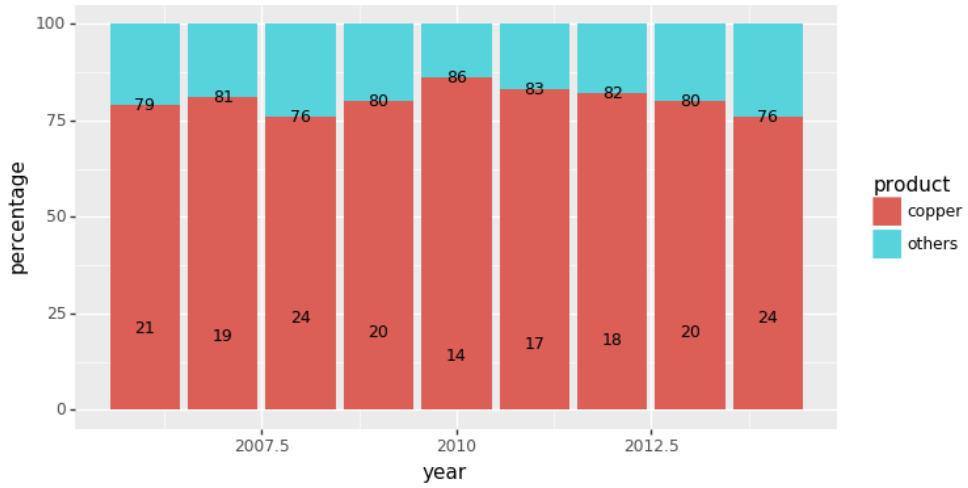
```
p4 = (ggplot(copper, aes("year", "percentage", fill="product"))
      + geom_col(position=position_stack(reverse=True))
)
p4
```



4.4 Adding data labels

To label the bars according to some variable in the data, we add another geom - `geom_text()` - and add `label="percentage"` to its `aes()` argument. We can also set the size of the labels in this geom.

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(copper, aes(label="percentage"), size=9)
)
p4
```

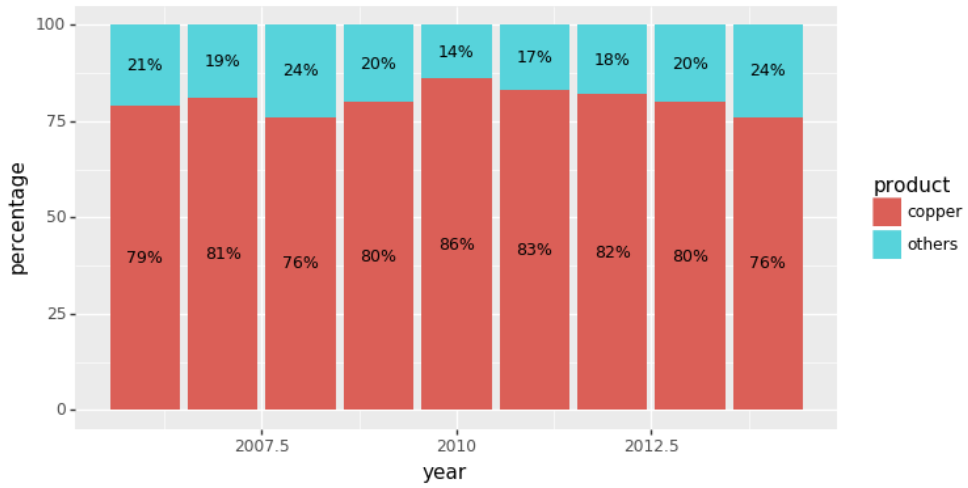


4.5 Adjusting data labels position

To adjust the position of the data labels from the default placement, we use both the `groupby` and `transform` functions on the data, and create a new variable called `pos`. This variable marks the position at the centre of each bar and can be used to specify the position of the labels by assigning it to the y-argument in `geom_text(aes())`.

We can also add a percentage symbol to our labels by adding the argument `format_string="{}%"` to `geom_text()`.

```
copper["pos"] = (copper[["year", "percentage"]].groupby("year").\
                 transform(pd.Series.cumsum))
copper["pos"] = copper["pos"] - 0.5 * copper["percentage"]

p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
)
p4
```

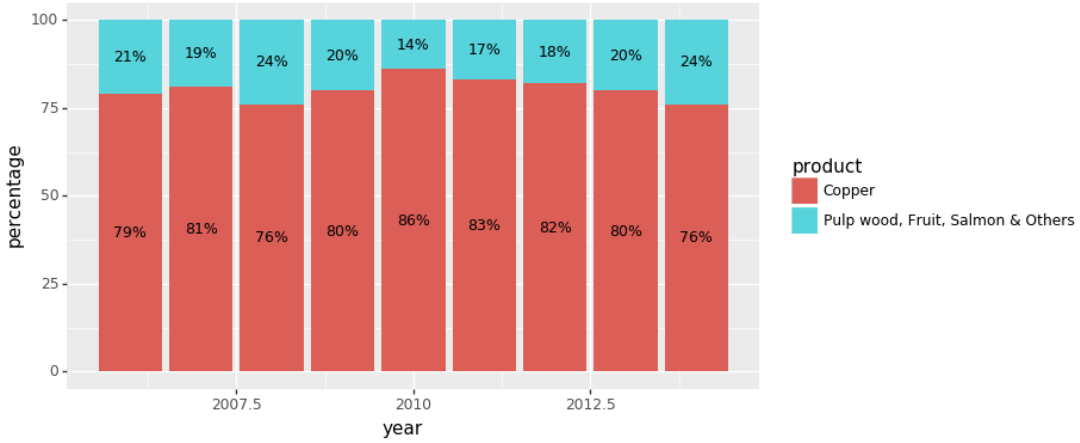


4.6 Changing group labels

To change the names of the groups in the legend, we need to rename the levels of the `product` variable. We can do this by creating a dictionary called `prod_name`, create another dictionary within *that*, and put in the values we want to change as key-value pairs. We then pass this dictionary to the `replace` method, making sure to set the `inplace` argument to `True` to make sure we overwrite the previous values of the column.

```
prod_name = {
    "product": {"copper": "Copper",
                "others": "Pulp wood, Fruit, Salmon & Others"}
}
```

```
copper.replace(prod_name, inplace=True)

p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
)
p4
```



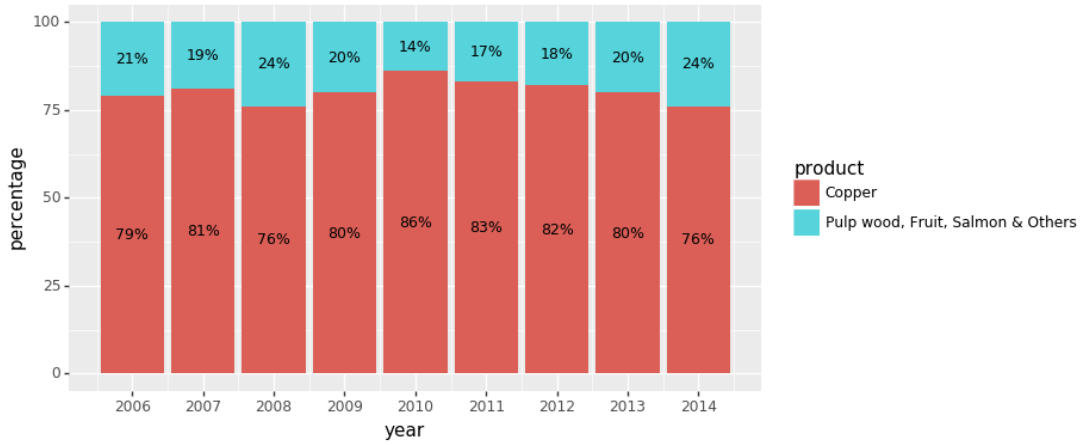
4.7 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, if we also wanted to change the y-axis we could use the `scale_y_continuous` option. Here we will change the x-axis to every year, rather than every 2 years. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[2] which generates a sequence from your selected start, stop and step values respectively. Note that because of Python's indexing, you need to set the `stop` argument to be one number more than your desired maximum when using `np.arange`.

Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis (passed as a list), although we haven't done so in this plot.

[2] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html`

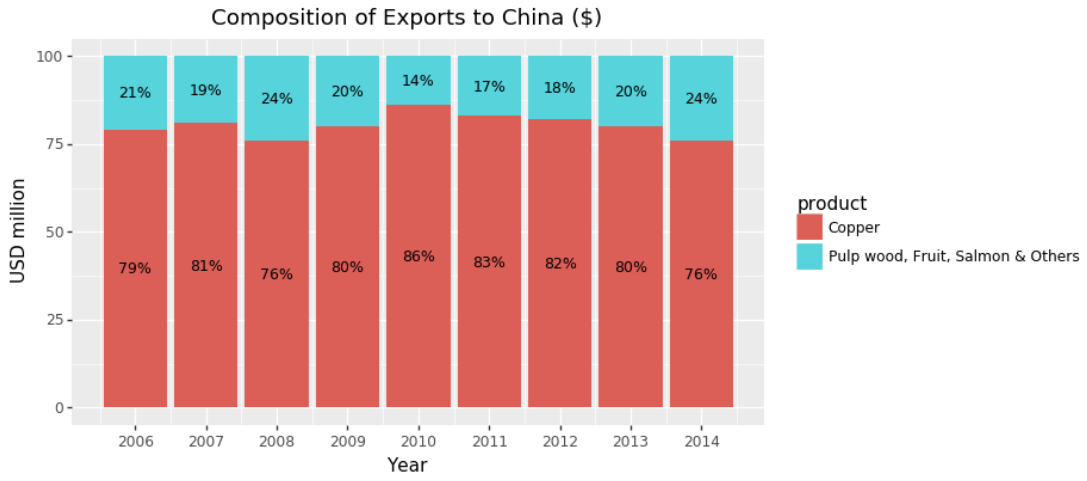
```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
)
p4
```



4.8 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
)
p4
```

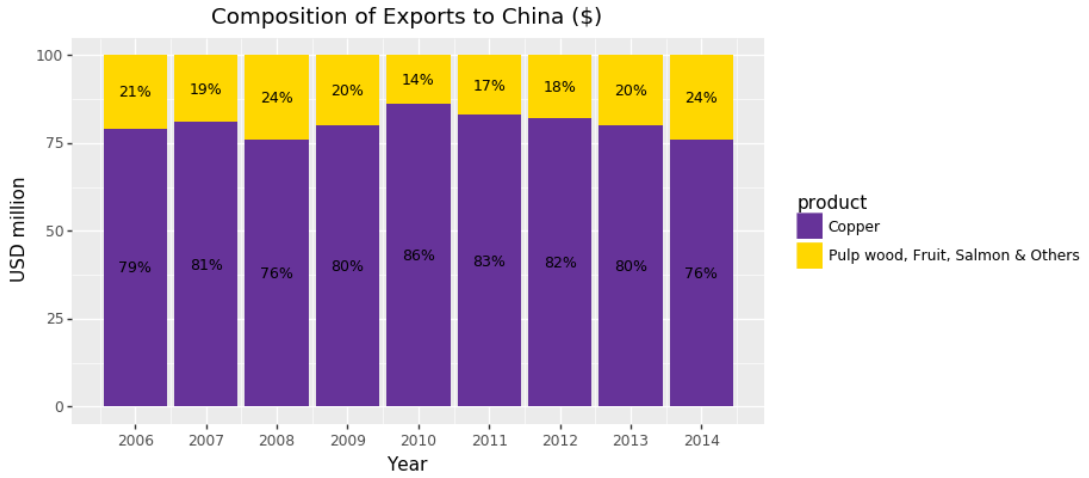


4.9 Adjusting the colour palette

There are a few options for adjusting the colour. The most simple is to change each level to a named colour using `scale_fill_manual`. `plotnine` uses the colour palette utilised by `matplotlib`, and the full set of named colours recognised by `ggplot` is here[3]. Let's try changing our bar areas to `rebeccapurple` and `gold`.

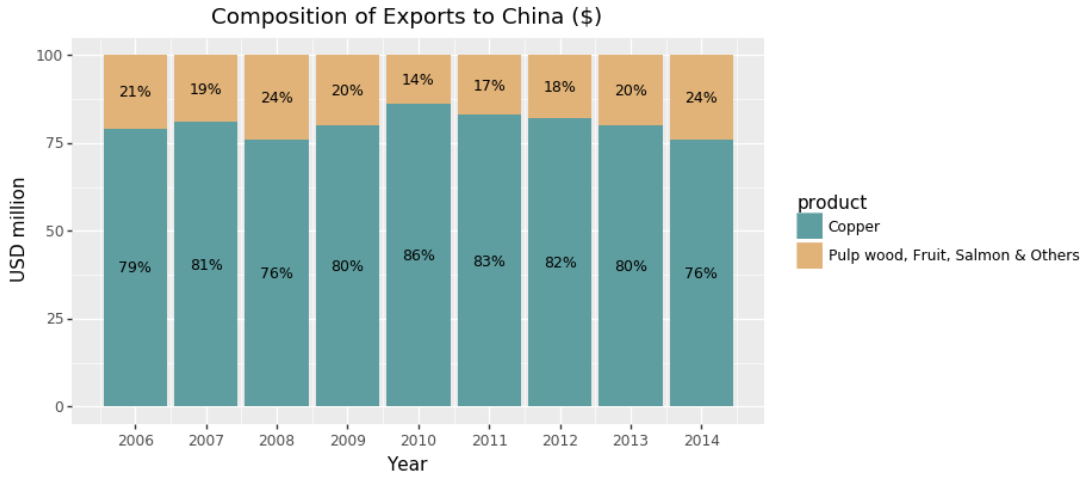
```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["rebeccapurple", "gold"])
)
p4
```

[3] https://matplotlib.org/examples/color/named_colors.html



We can also change the colours using specific HEX codes. Let's change the areas to `#5F9EA0` and `#E1B378`.

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["#5F9EA0", "#E1B378"])
)
p4
```

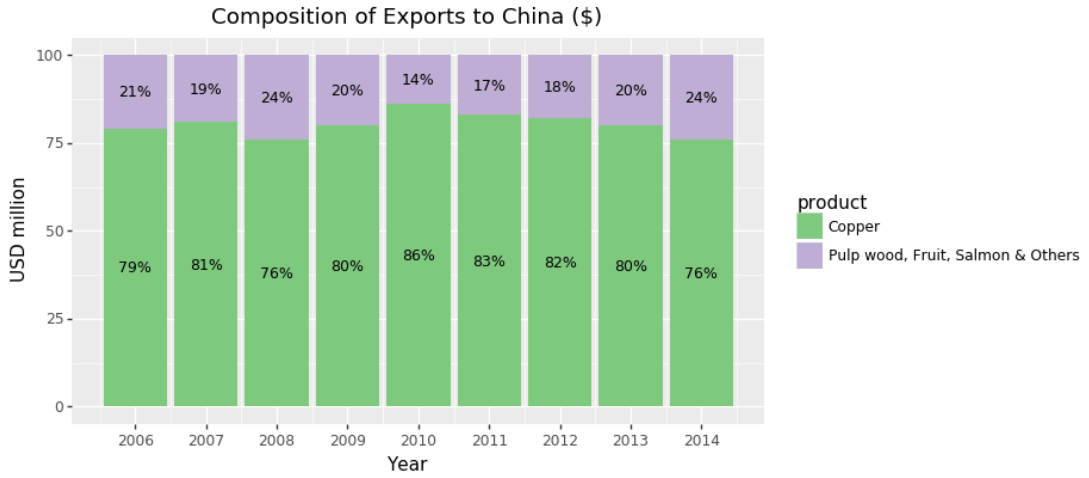


Another option is changing the colours using the schemes from ColorBrewer[4]. Here we have used the `scale_fill_brewer` option with the qualitative scale `Accent`. More information on using `scale_fill_brewer` is here[5].

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
)
p4
```

[4] http://colorbrewer2.org/

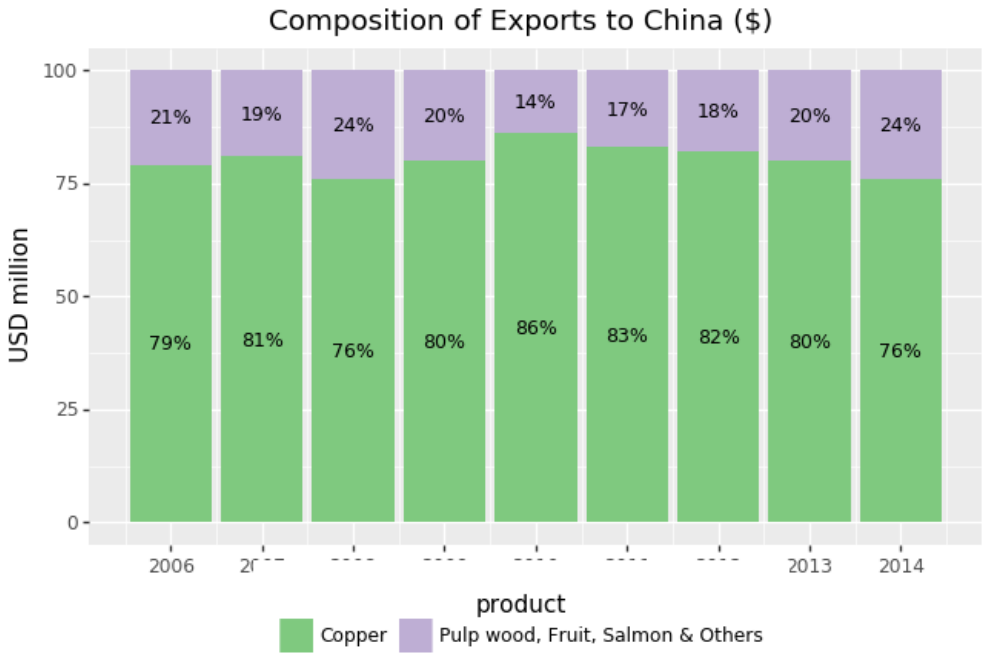
[5] http://plotnine.readthedocs.io/en/stable/generated/plotnine.scales.scale_color_brewer.html#plotnine.scales.scale_color_brewer



4.10 Adjusting the legend

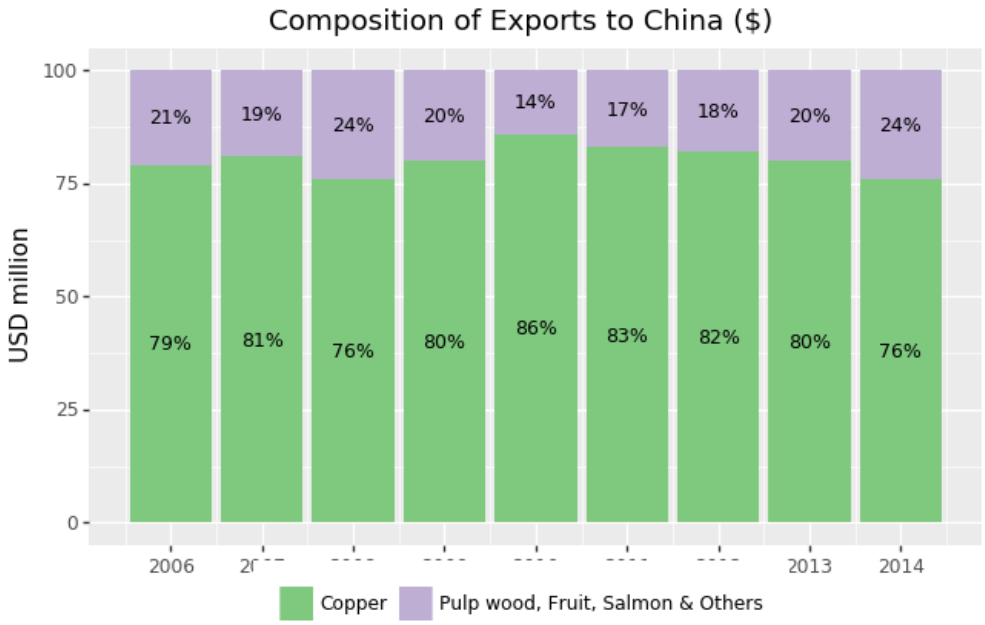
To adjust the position of the legend from the default spot of right of the graph, we add the `theme` option and specify the `legend_position="bottom"` argument. We can also change the legend shape using the `legend_direction="horizontal"` argument. Finally, we can centre the legend title position using the argument `legend_title_align="center"`.

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
    )
)
p4
```



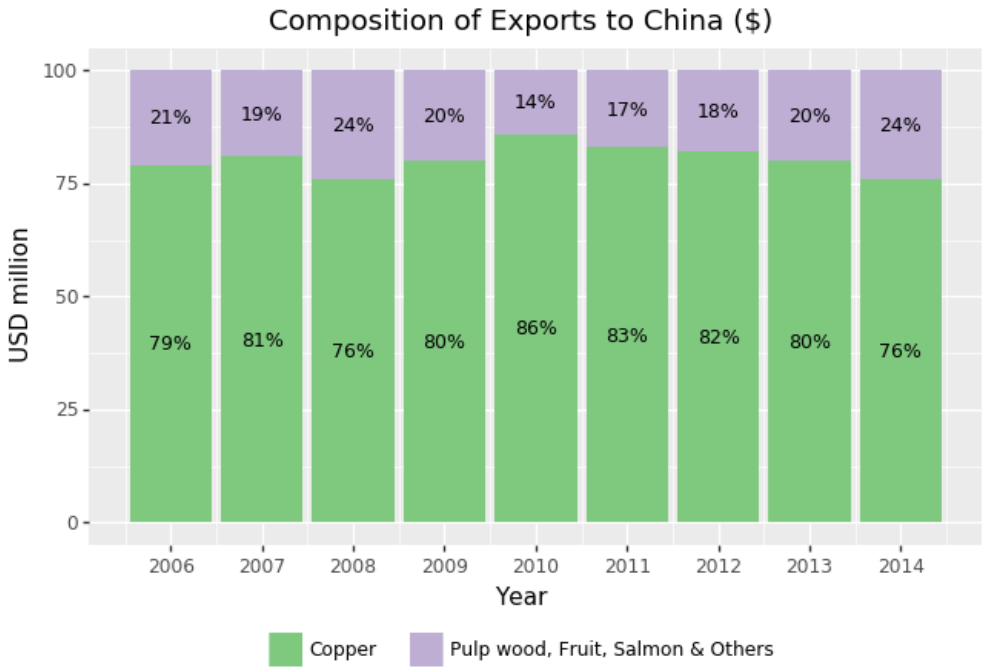
Let's also get rid of the legend title. We can do this by adding `legend_title` with the argument `element_blank()` to `theme`. If we instead wanted to change the name of the title, we could add the argument `name="Product"` to the `scale_fill_brewer()` option. Note that you can also do this with the `scale_fill_manual()` option we used above.

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
    )
)
p4
```



Unfortunately, the legend is now sitting over our x-axis. We can fix this using the `legend_box_spacing=0.4` option within `theme`. This allows us to move the legend down as much as needed. The legend is also looking a little squashed. We can space it out a bit more using the `legend_entry_spacing_x=15` argument.

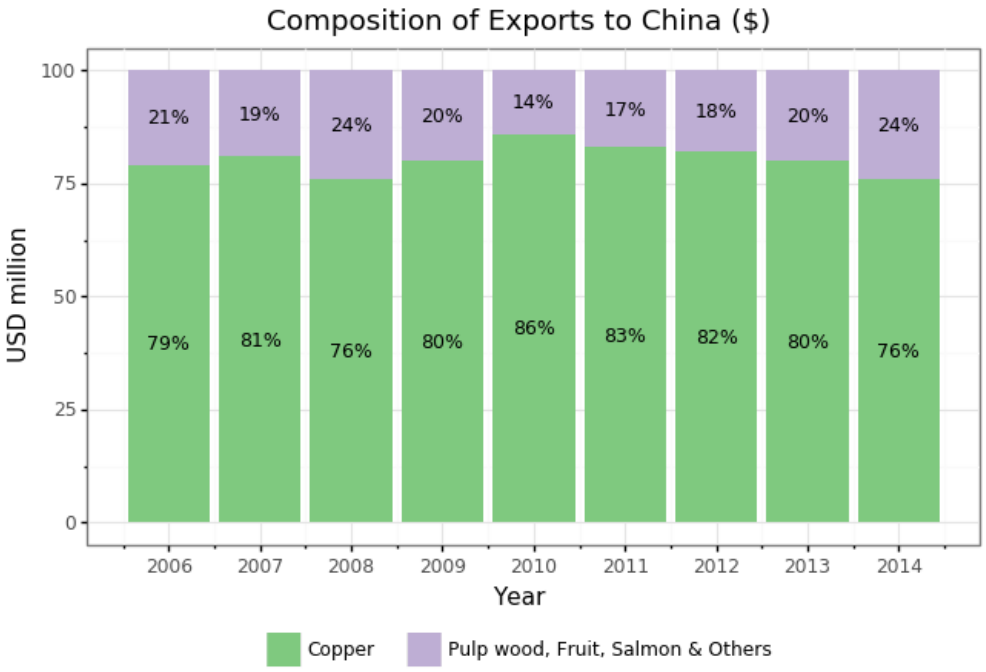
```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_title=element_blank(),
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
    )
)
p4
```



4.11 Using the white theme

We can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after `ggplot()`.

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(aes(y="pos", label="percentage"), size=9,
                format_string="{}%")
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year") + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme_bw()
    + theme(
        legend_position="bottom", legend_direction="horizontal",
        legend_title_align="center", legend_title=element_blank(),
        legend_box_spacing=0.4, legend_entry_spacing_x=15,
    )
)
p4
```



4.12 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[6]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we

[6] `xkcd.com/1350/xkcd-Regular.otf`

want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[7]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight()` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

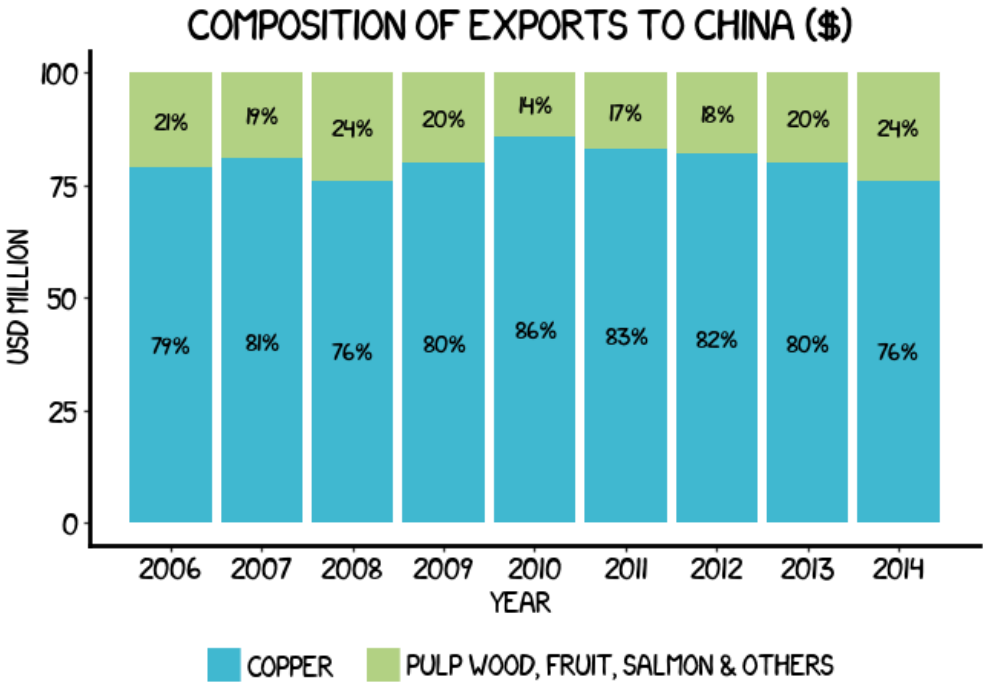
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- To change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`;
- Finally, in order to change the font of the labels within `geom_text`, we pass it the `family="xkcd"` argument. This will only work if you have the xkcd font installed on your computer.

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(
        aes(y="pos", label="percentage"), size=10, family="xkcd",
        format_string="{}%"
    )
```

[7] https://matplotlib.org/api/font_manager_api.html

```
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["#40b8d0", "#b2d183"])
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.5,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        legend_key=element_blank(),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p4
```



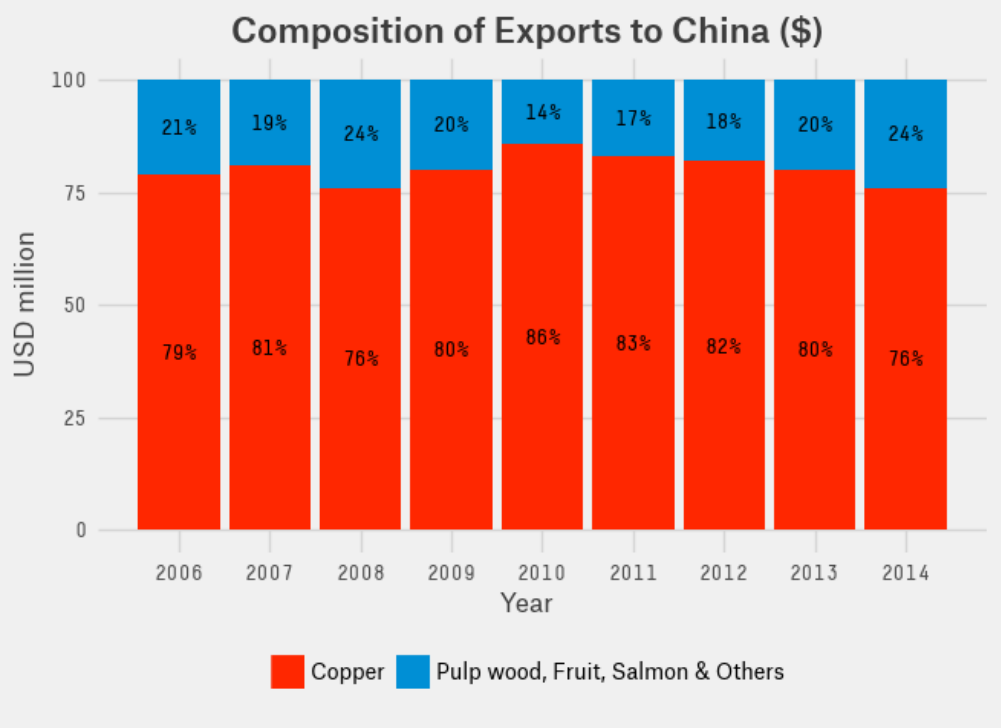
4.13 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[8]). Below we've applied `theme_538()`, which approximates graphs in the nice FiveThirtyEight website. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[9] and 'Decima Mono Pro'[10] in `legend_text`, `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
legend_text = fm.FontProperties(fname=agr)
axis_text = fm.FontProperties(fname=agr)
```

[8] http://plotnine.readthedocs.io/en/stable/api.html#themes
[9] https://commercialtype.com/catalog/atlas/atlas_grotesk
[10] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
legend_text.set_size(10)
axis_text.set_size(12)
body_text.set_size(10)


p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(
        aes(y="pos", label="percentage"),
        size=10,
        format_string="{}%",
        family="DecimaMonoPro",
    )
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_manual(["#FF2700", "#008FD5"])
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.5,
        legend_title=element_blank(),
        legend_text=element_text(fontproperties=legend_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p4
```



4.14 Creating your own theme

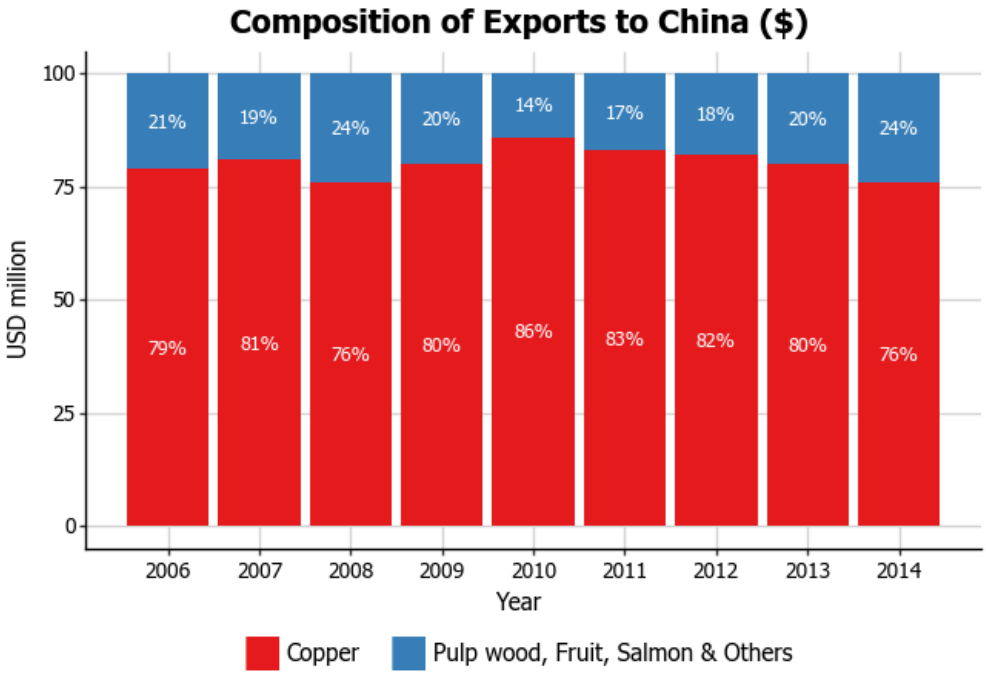
Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

We've also changed our colour scheme to another ColorBrewer theme, the quantitative scale `Set1`.

With all of these customisations, we now finally have the graph we presented at the beginning of this chapter.

```
p4 = (
    ggplot(copper, aes("year", "percentage", fill="product"))
    + geom_col(position=position_stack(reverse=True))
    + geom_text(
        aes(y="pos", label="percentage"), size=9, colour="white",
        format_string="{}%"
    )
    + scale_x_continuous(breaks=np.arange(2006, 2015, 1))
    + ggtitle("Composition of Exports to China ($)")
    + xlab("Year")
    + ylab("USD million")
    + scale_fill_brewer(type="qual", palette="Set1")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
        legend_title=element_blank(),
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p4
```

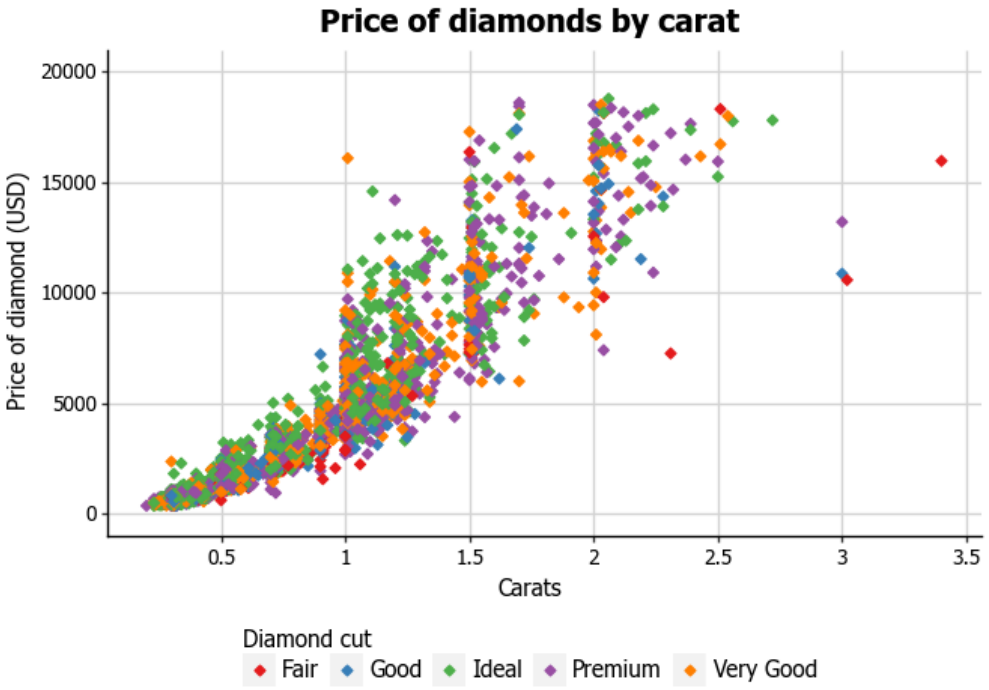

Composition of Exports to China ($)
USD million
100
75
50
25
0
21%
19%
24%
20%
14%
17%
18%
20%
24%
79%
81%
76%
80%
86%
83%
82%
80%
76%
2006
2007
2008
2009
2010
2011
2012
2013
2014
Year
Copper
Pulp wood, Fruit, Salmon & Others

Chapter 5

Scatterplots

5.1 Introduction

In this chapter, we will work towards creating the scatterplot below. We will take you from a basic scatterplot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from plotnine import data
from pandas import DataFrame
```

For this example, we'll be using the `diamonds` dataset from plotnine's `data` module. The details of this dataset are here[1]. As this is a large dataset for the purposes of graphing, we'll take a sample of 3,000 rows to make it a bit nicer for visualisation.

```
diamonds = data.diamonds
sdiamond = diamonds.sample(3000)
```

The exact sample we've used in this chapter can be downloaded from here[2], or you can create your own extract of the data using the code above. You can load the sample data into Python like so:

```
sdiamond = pd.read_csv("https://git.io/JecIN")
```

We will now convert the `clarity` variable from a categorical variable into a numeric variable. We can do this by first creating a new variable called `nclarity`. We then create a dictionary called `clarity_nums`, create another dictionary within *that*, and put in the values we want to change as key-value pairs. We then pass this dictionary to the `replace` method, making sure to set the `inplace` argument to `True` to make sure we overwrite the previous values of the column.

[1] `http://plotnine.readthedocs.io/en/stable/generated/plotnine.data.diamonds.html`
[2] `https://git.io/JecIN`

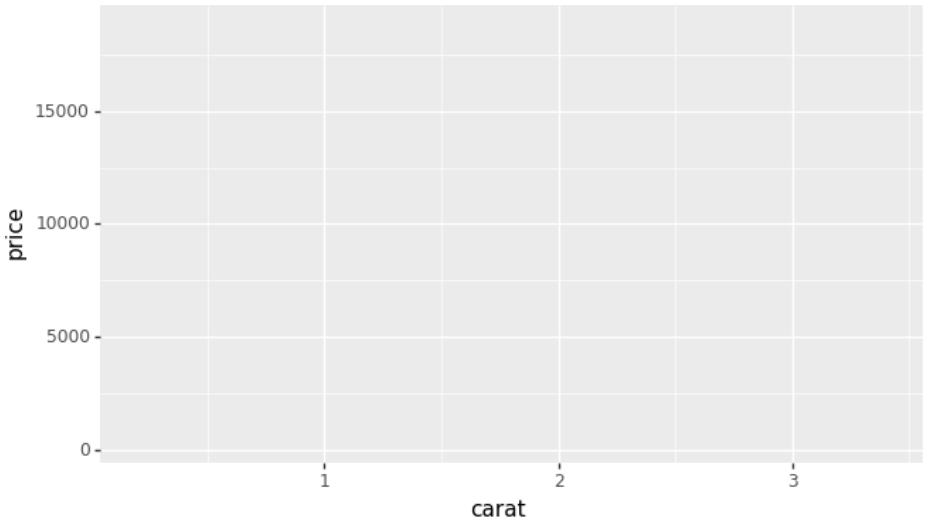
```
sdiamond["nclarity"] = sdiamond["clarity"]
clarity_nums = {
    "nclarity": {
        "I1": 8,
        "SI1": 7,
        "SI2": 6,
        "VS1": 5,
        "VS2": 4,
        "VVS1": 3,
        "VVS2": 2,
        "IF": 1,
    }
}
sdiamond.replace(clarity_nums, inplace=True)
```

5.2 Basic ggplot structure

In order to initialise a scatterplot we tell ggplot that `sdiamond` is our data, and specify that our x-axis plots the `carat` variable and our y-axis plots the `price` variable. You may have noticed that we put our x- and y-variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `carat` to the x-axis and `price` to the y-axis.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

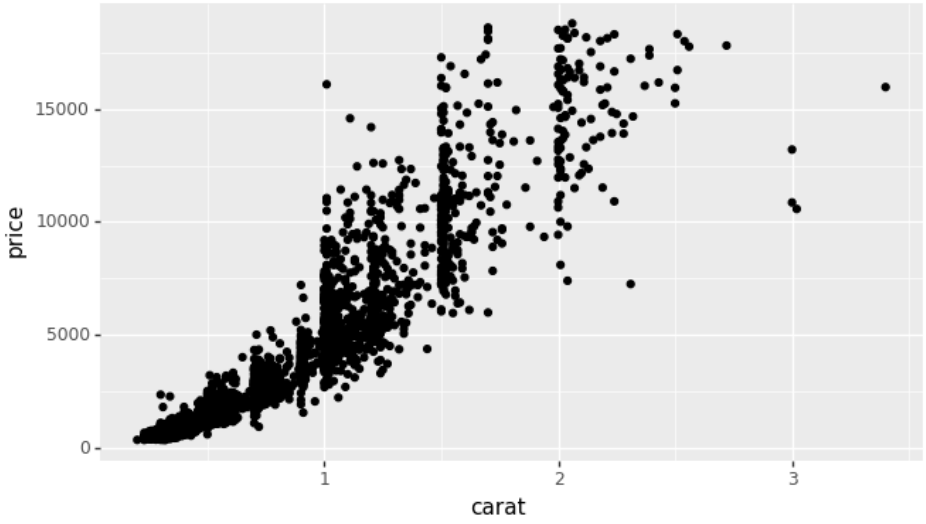
```
p5 = ggplot(sdiamond, aes("carat", "price"))
p5
```



5.3 Basic scatterplot

We can do this using `geoms`. In the case of a scatterplot, we use the `geom_point()` geom.

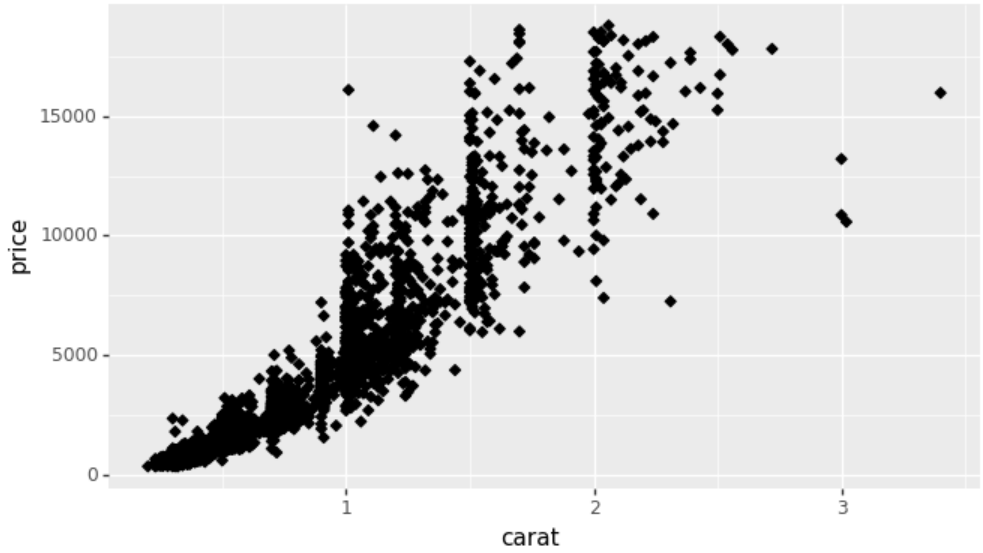
```
p5 = ggplot(sdiamond, aes("carat", "price")) + geom_point()
p5
```



5.4 Changing the shape of the data points

Perhaps we want the data points to be a different shape than a solid circle. We can change these by adding the `shape` argument to `geom_point`. The shape arguments for `plotnine` are the same as those available in `matplotlib`, and are therefore a little more limited than those in R's implementation of `ggplot2`. Nonetheless, there is a good range of options. The allowed arguments are here[3]. Let's change all of the markers to - what else - diamonds, using the argument `"D"`.

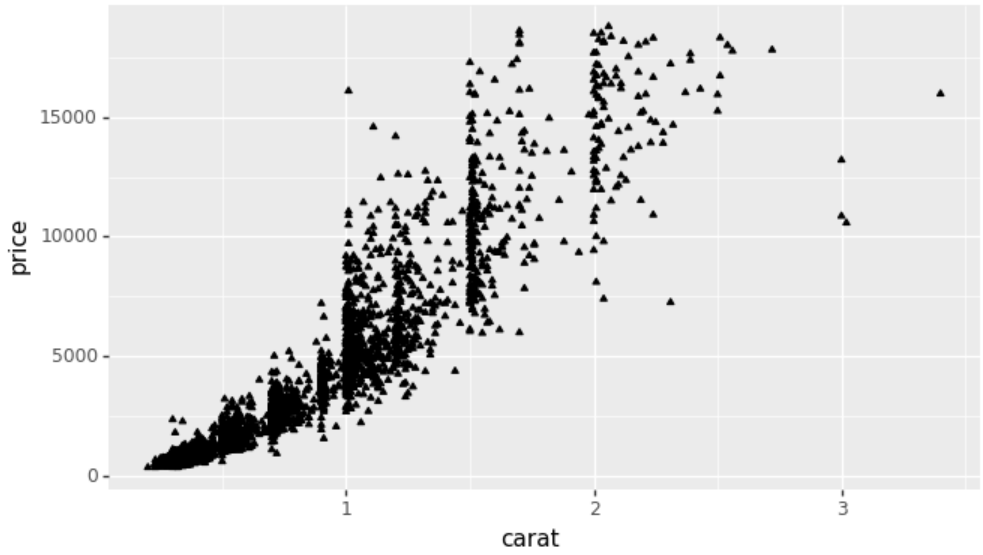
```
p5 = (
    ggplot(sdiamond, aes("carat", "price"))
      + geom_point(shape="D")
)
p5
```



We can also change the shape by passing a tuple to the `shape` argument, which will render a polygon. The tuple has 3 arguments (`numsides`, `style`, `angle`), and the parameters for these can be seen here[4]. In this example, we've created a 3-sided regular polygon (i.e., a triangle!) by passing the tuple `(3,0)`.

[3] https://matplotlib.org/api/markers_api.html
[4] https://matplotlib.org/api/markers_api.html

```
p5 = (
    ggplot(sdiamond, aes("carat", "price"))
    + geom_point(shape=(3, 0))
)
p5
```



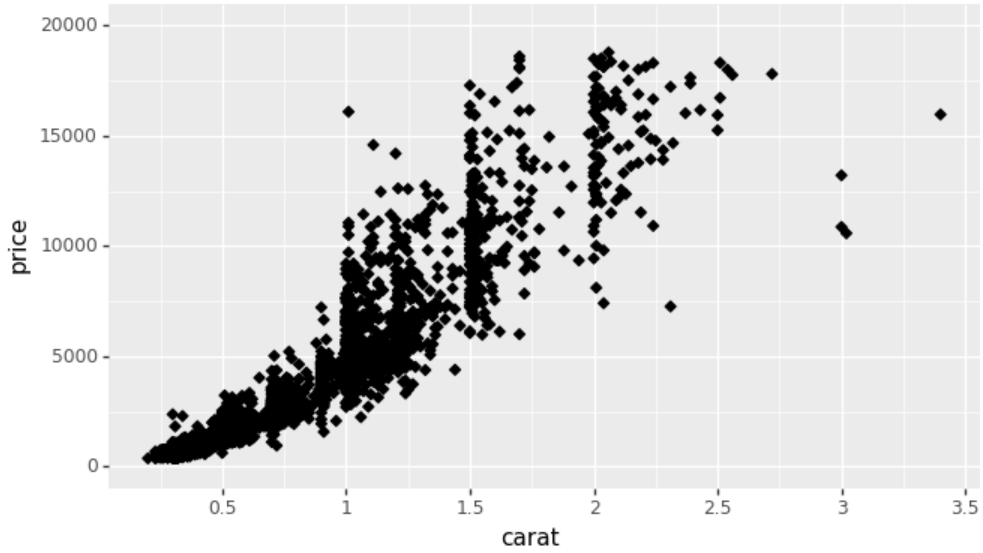
5.5 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, to change the y-axis we use the `scale_y_continuous` option. Here we will change the x-axis to every 0.5 carats, rather than 1, and change the range of the y-axis from 0 to 20,000. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[5] which generates a sequence from your selected start, stop and step values respectively. Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis.

We'll also go back to the diamond shape going forward.

[5] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html`

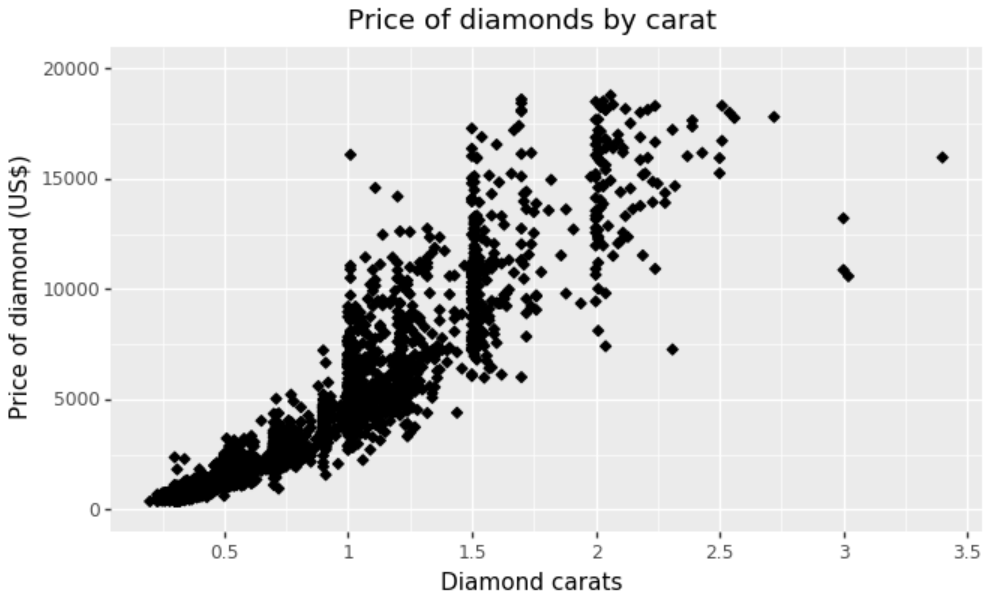
```
p5 = (
    ggplot(sdiamond, aes("carat", "price"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
)
p5
```



5.6 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p5 = (
    ggplot(sdiamond, aes("carat", "price"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p5
```

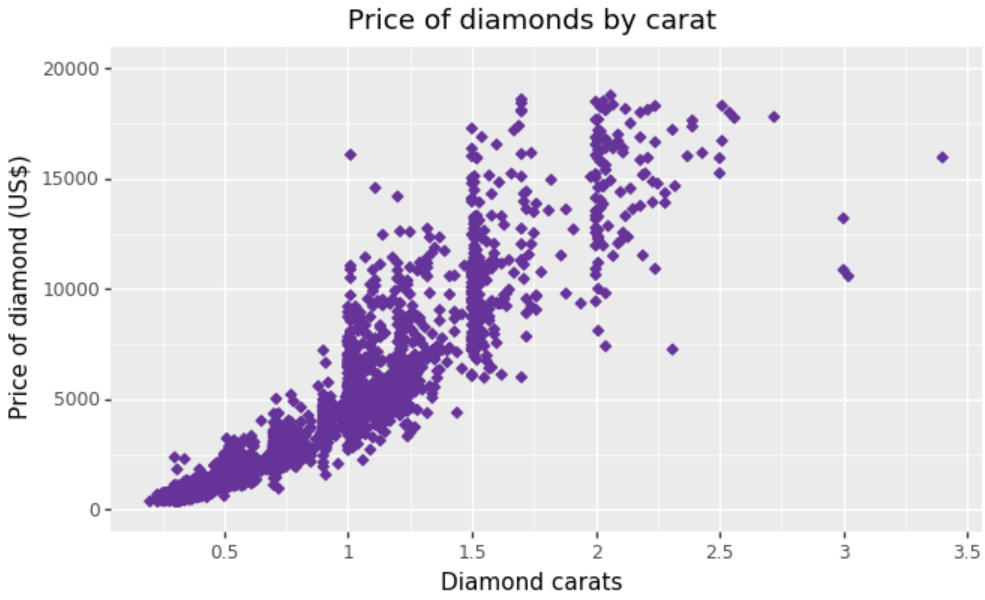


5.7 Adjusting the colour palette

There are a few options for adjusting the colour. The most simple is to make every point one fixed colour. As with shape, `plotnine` uses the colour palette utilised by `matplotlib`. The full set of named colours recognised by `ggplot` is here[6]. Let's try changing our shapes to `rebeccapurple`.

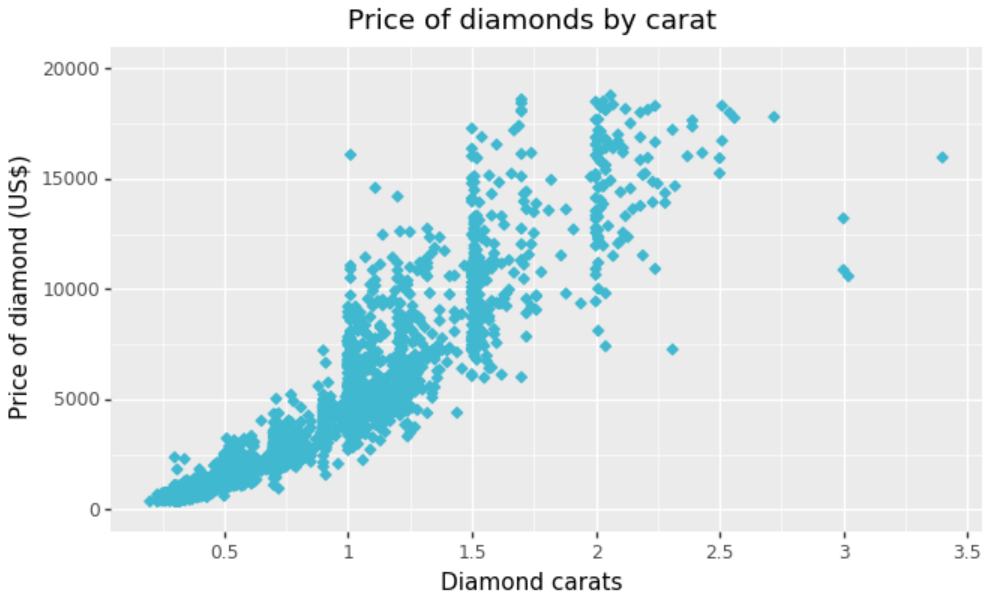
```
p5 = (
    ggplot(sdiamond, aes("carat", "price"))
    + geom_point(shape="D", colour="rebeccapurple")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p5
```

[6] https://matplotlib.org/examples/color/named_colors.html



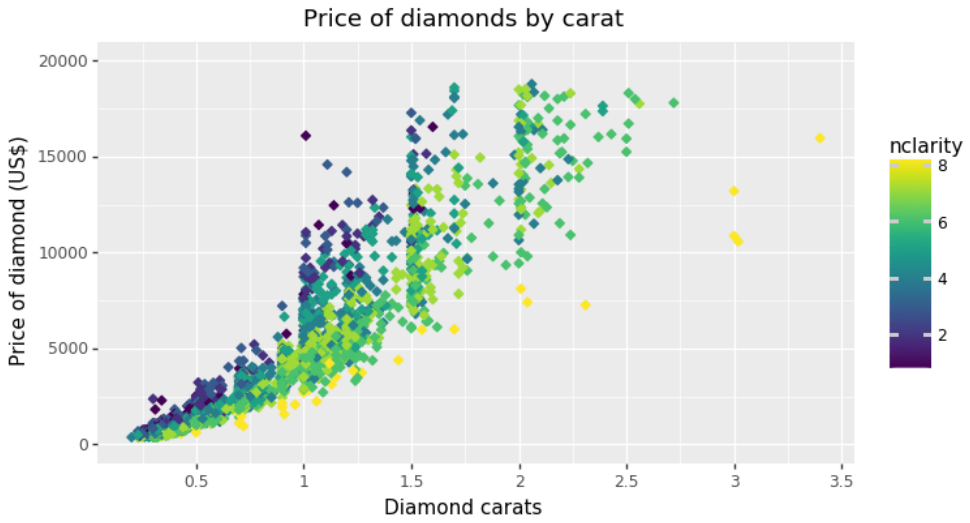
You can change the colours using specific HEX codes instead. Here we have made the shapes `'#40b8d0'` (vivid cyan).

```
p5 = (
    ggplot(sdiamond, aes("carat", "price"))
    + geom_point(shape="D", colour="#40b8d0")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p5
```



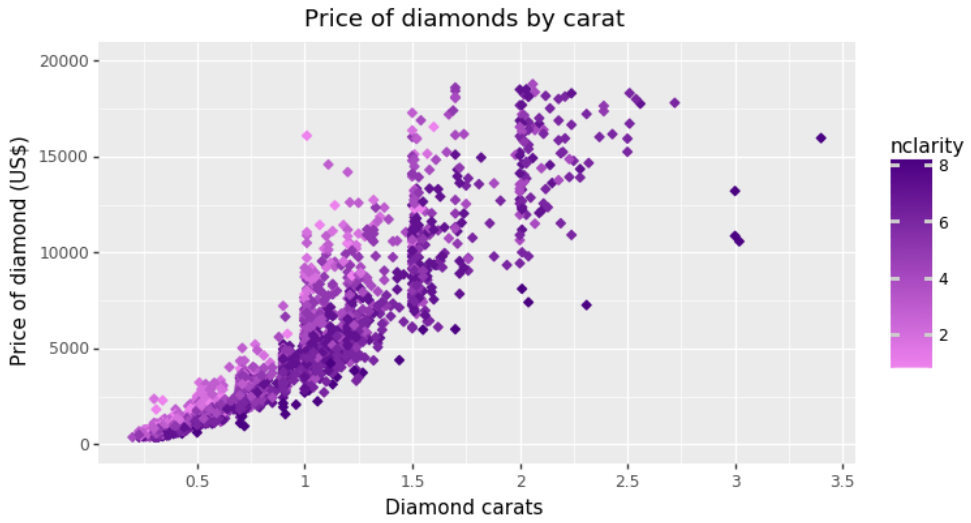
You can also change the colour of the data points according to the levels of another variable. This can be done either as a continuous gradient, or as a levels of a categorical variable. Let's change the colour by the values of our continuous clarity variable that we created at the beginning of this chapter, `nclarity`, by adding it to our aesthetic mapping, `aes`:

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="nclarity"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p5
```



We can change the gradient's colours by adding the `scale_fill_continuous` option. The `low` and `high` arguments specify the range of colours the gradient should transition between.

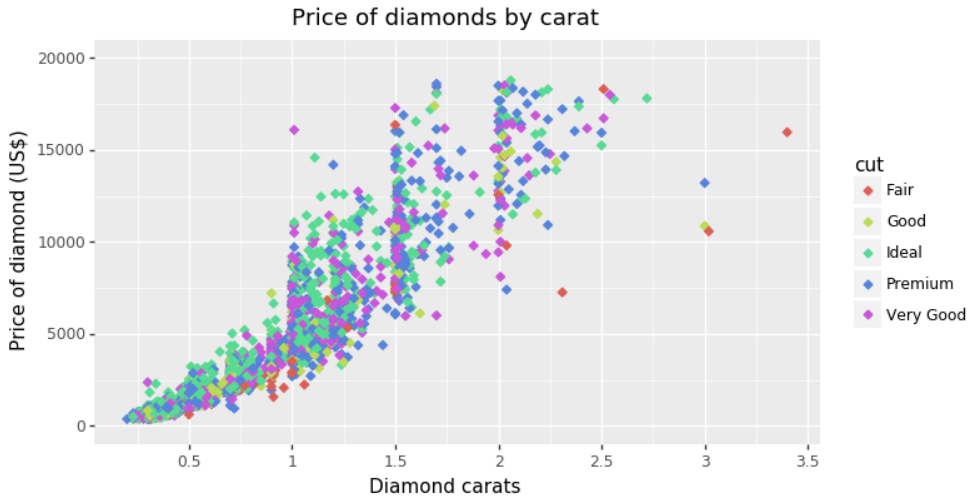
```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="nclarity"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_gradient(low="violet", high="indigo")
)
p5
```



We can see that higher diamond prices correspond with a higher clarity.

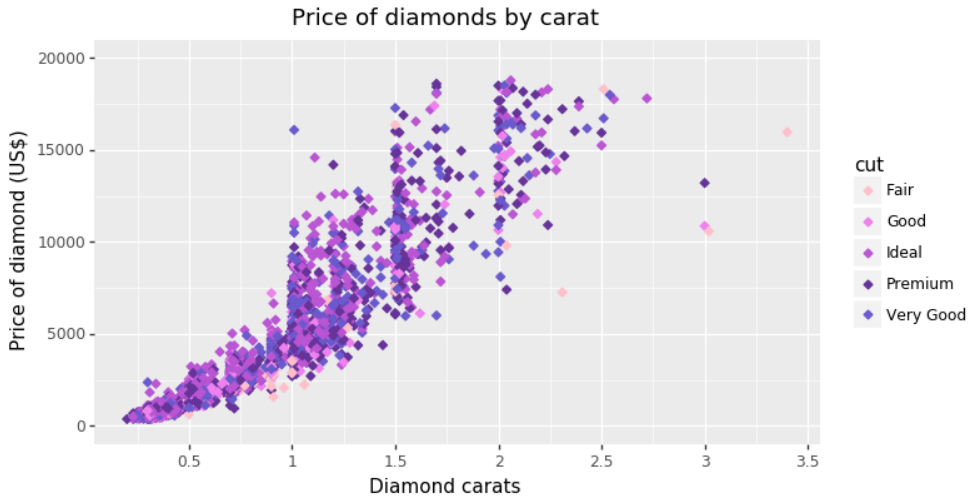
Let's now change the colours of the data points by a categorical variable, `cut`.

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p5
```



Again, we can change the colours of these data points, this time using `scale_fill_manual`.

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_manual(
        values=["pink", "violet", "mediumorchid",
                "rebeccapurple", "slateblue"]
    )
)
p5
```

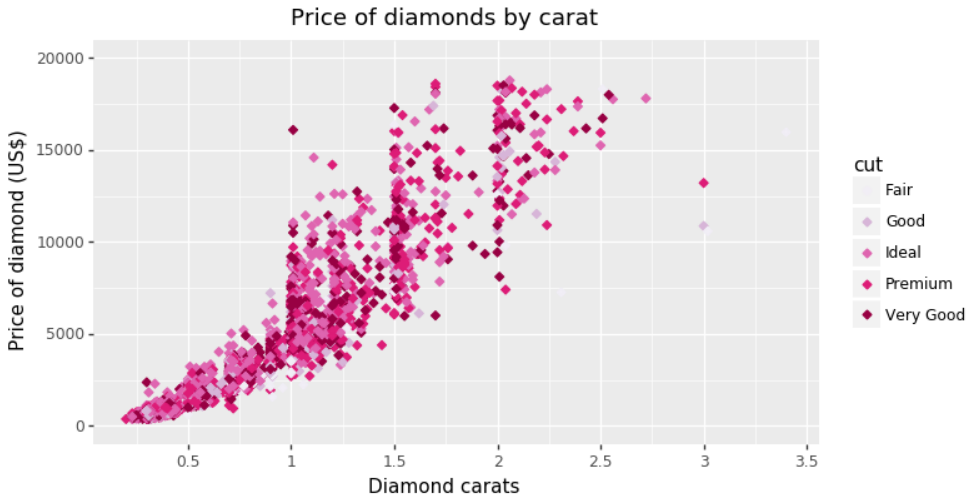


We can also change the manual colours using the schemes from ColorBrewer[7]. Here we have used the `scale_colour_brewer` option with the sequential scale `PuRd`. More information on using `scale_colour_brewer` is here[8].

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_brewer(type="seq", palette="PuRd")
)
p5
```

[7] http://colorbrewer2.org/

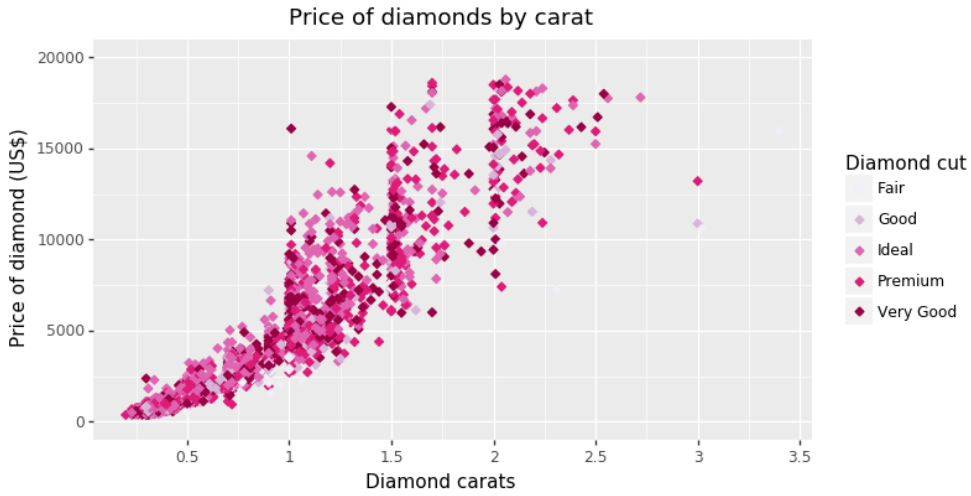
[8] http://plotnine.readthedocs.io/en/stable/generated/plotnine.scales.scale_color_brewer.html#plotnine.scales.scale_color_brewer



5.8 Adjusting the legend

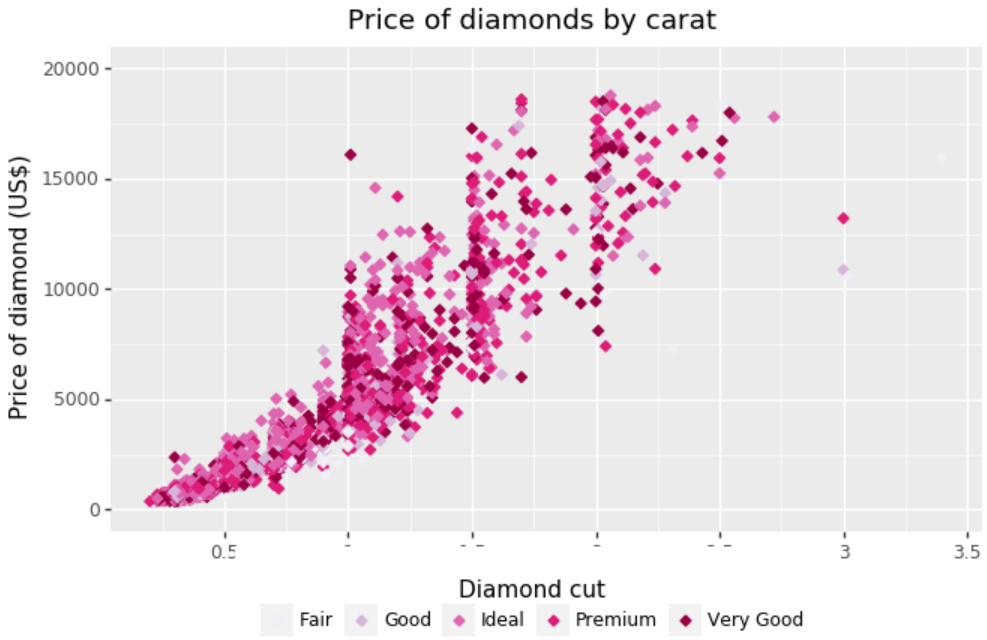
To change the title of the legend, we can add the argument `name="Diamond cut"` to the `scale_colour_brewer()` option. Note that you can also do this with both the `scale_colour_gradient()` and `scale_colour_manual()` options we used above.

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_brewer(type="seq", palette="PuRd",
                          name="Diamond cut")
)
p5
```



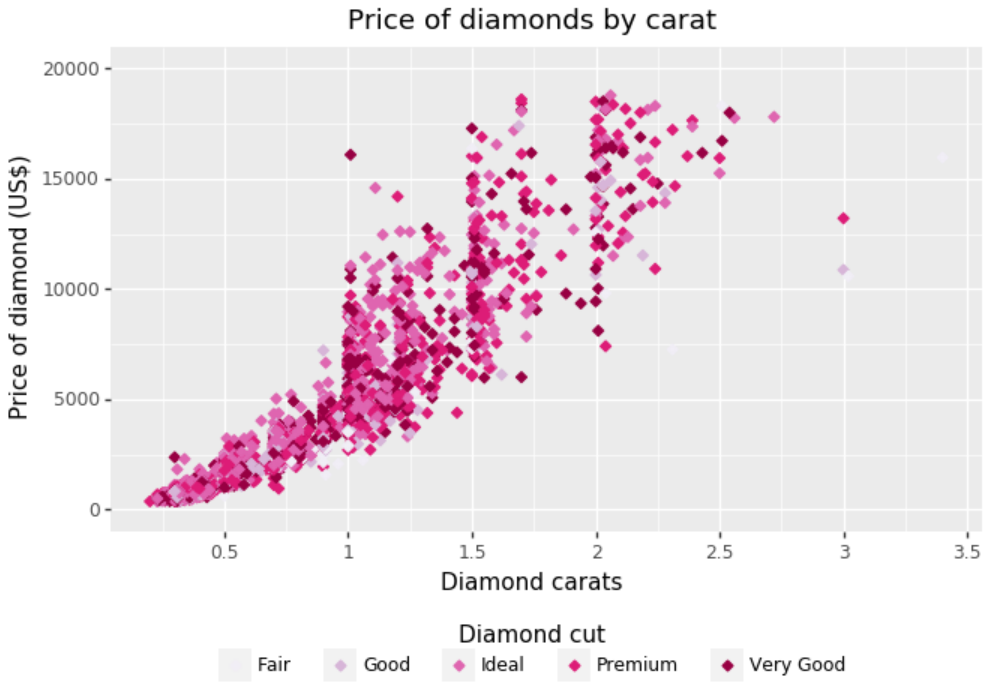
To adjust the position of the legend from the default spot of right of the graph, we add the `theme` option and specify the `legend_position = "bottom"` argument. We can also change the legend shape using the `legend_direction = "horizontal"` argument. Finally, we can centre the legend title position using the argument `'center'` with the parameter `legend_title_align`.

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_brewer(type="seq", palette="PuRd",
                          name="Diamond cut")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
    )
)
p5
```



Unfortunately, the legend is now sitting over our x-axis. We can fix this using the `legend_box_spacing = 0.4` option within `theme`. This allows us to move the legend down as much as needed. The legend is also looking a little squashed. We can space it out a bit more using the `legend_entry_spacing_x = 15` argument.

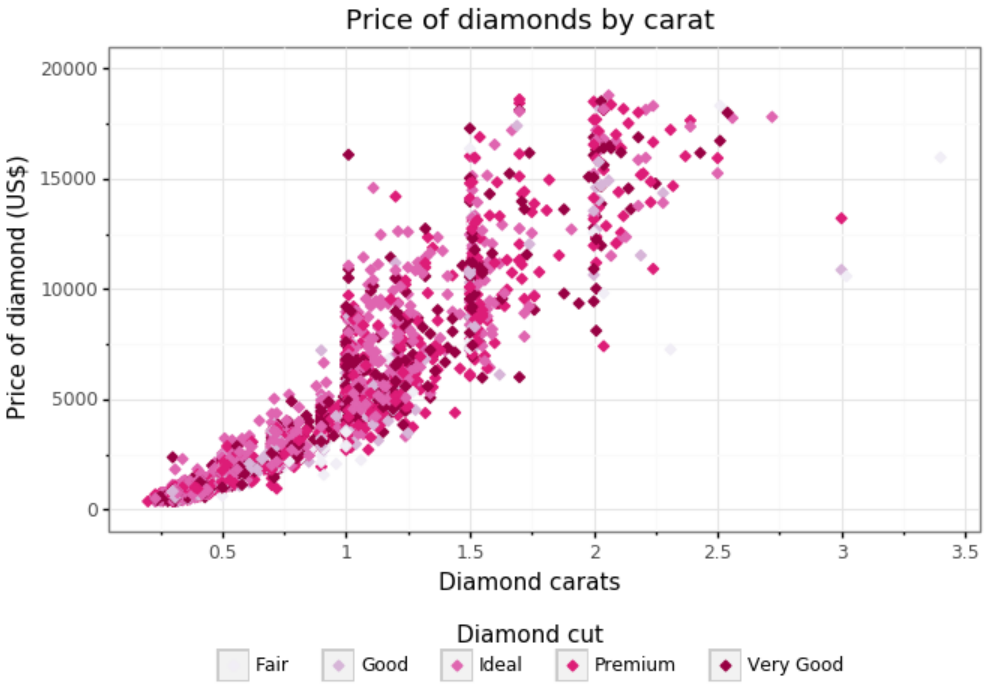
```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_brewer(type="seq", palette="PuRd",
                          name="Diamond cut")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=15,
    )
)
p5
```



5.9 Using the white theme

As explained in the previous chapters, we can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after `ggplot()`. As you can see, we can further tweak the graph using the `theme` option, which we've used so far to change the legend.

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats") + ylab("Price of diamond (US$)")
    + scale_colour_brewer(type="seq", palette="PuRd", name="Diamond cut")
    + theme_bw()
    + theme(
        legend_position="bottom", legend_direction="horizontal",
        legend_title_align="center", legend_box_spacing=0.4,
        legend_entry_spacing_x=15))
p5
```



5.10 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[9]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we

[9] `xkcd.com/1350/xkcd-Regular.otf`

want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[10]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight()` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

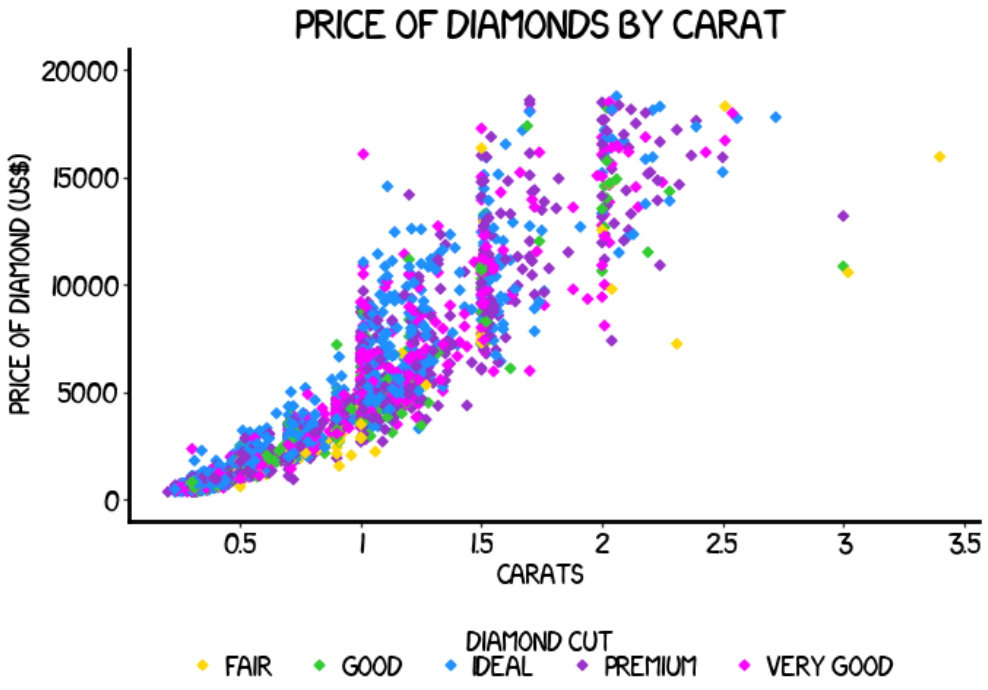
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_manual(
        values=["gold", "limegreen", "dodgerblue",
                "darkorchid", "magenta"],
```

[10] https://matplotlib.org/api/font_manager_api.html

```
        name="Diamond cut",
    )
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.5,
        legend_entry_spacing_x=15,
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        legend_key=element_blank(),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p5
```



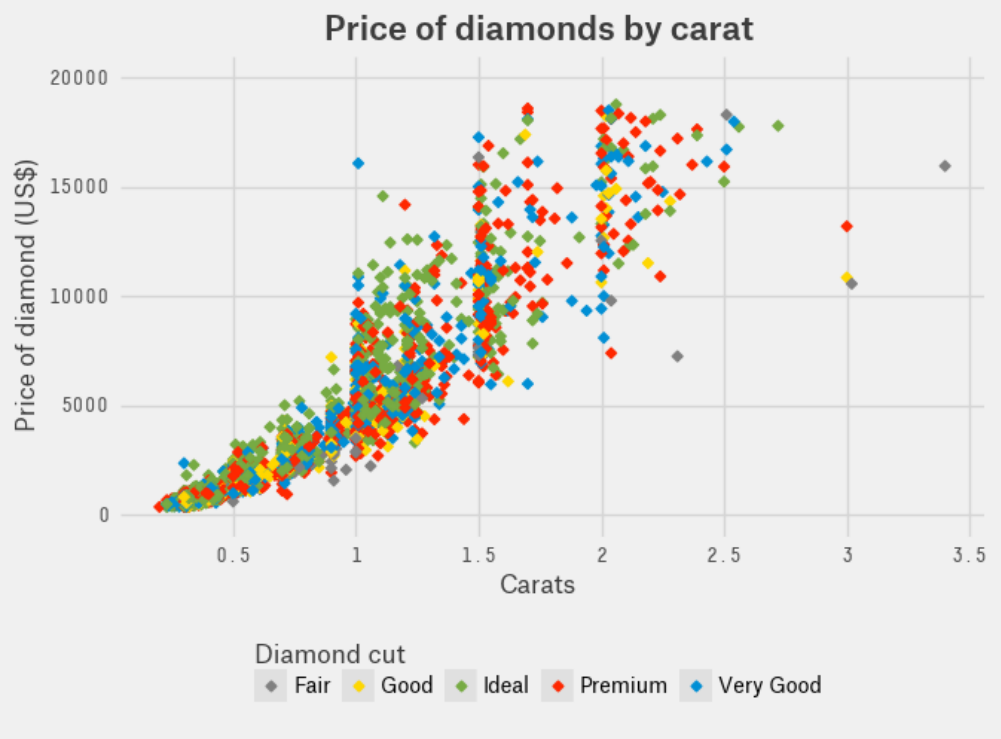
5.11 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[11]). Below we've applied `theme_538()`, which approximates graphs in the nice FiveThirtyEight website. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[12] and 'Decima Mono Pro'[13] in `legend_title`, `legend_text`, `axis_title`, `plot_title`, `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
legend_text = fm.FontProperties(fname=agr)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
legend_text.set_size(10)
axis_text.set_size(12)
body_text.set_size(10)


p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_manual(
        values=["grey", "gold", "#77AB43", "#FF2700", "#008FD5"],
        name="Diamond cut"
    )
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        legend_position="bottom",
        legend_direction="horizontal",
```

[11] http://plotnine.readthedocs.io/en/stable/api.html#themes
[12] https://commercialtype.com/catalog/atlas/atlas_grotesk
[13] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
        legend_box_spacing=0.5,
        legend_title=element_text(fontproperties=axis_text),
        legend_text=element_text(fontproperties=legend_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p5
```



5.12 Creating your own theme

Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

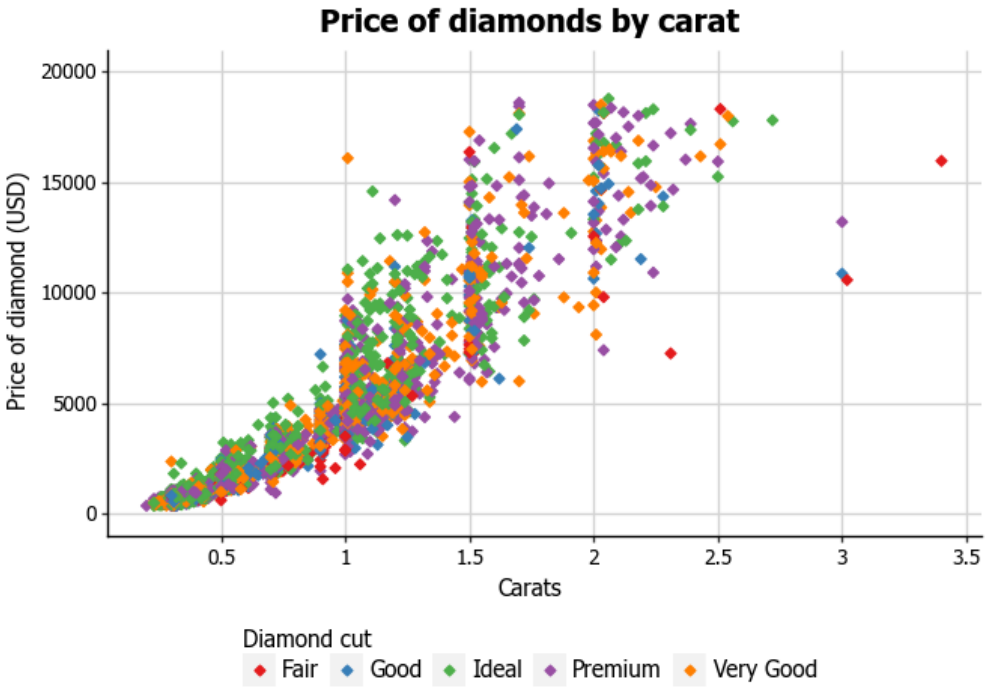
- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;

- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

We've also changed our colour scheme to another ColorBrewer theme, the quantitative scale `Set1`.

With all of these customisations, we now finally have the graph we presented at the beginning of this chapter.

```
p5 = (
    ggplot(sdiamond, aes("carat", "price", colour="cut"))
    + geom_point(shape="D")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Carats")
    + ylab("Price of diamond (USD)")
    + scale_colour_brewer(type="qual", palette="Set1",
                          name="Diamond cut")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.4,
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p5
```

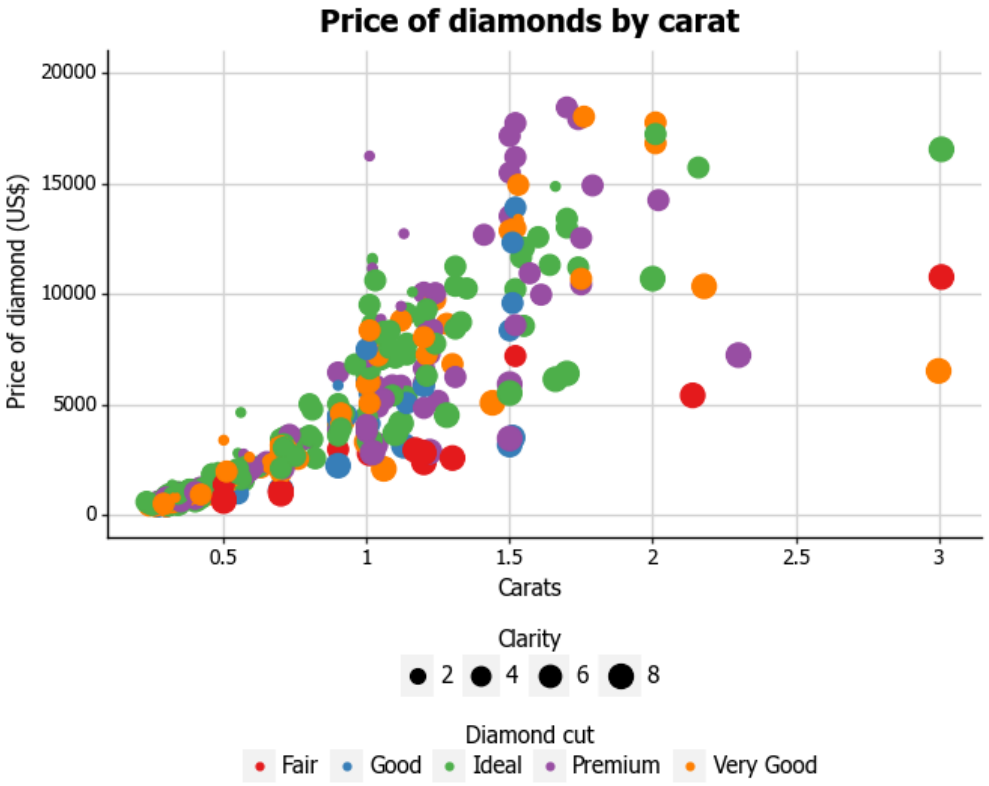

Price of diamonds by carat
Price of diamond (USD)
20000
15000
10000
5000
0
0.5
1
1.5
2
2.5
3
3.5
Carats
Diamond cut
Fair
Good
Ideal
Premium
Very Good

Chapter 6

Weighted scatterplots

6.1 Introduction

In this chapter, we will work towards creating the weighted scatterplot below. We will take you from a basic scatterplot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from plotnine import data
from pandas import DataFrame
```

For this example, we'll be using the `diamonds` dataset from plotnine's `data` module. The details of this dataset are here[1].

```
diamonds = data.diamonds
```

We will now convert the `clarity` variable from a categorical variable into a numeric variable. We can do this by first creating a new variable called `nclarity`. We then create a dictionary called `clarity_nums`, create another dictionary within *that*, and put in the values we want to change as key-value pairs. We then pass this dictionary to the `replace` method, making sure to set the `inplace` argument to `True` to make sure we overwrite the previous values of the column.

```
diamonds["nclarity"] = diamonds["clarity"]
clarity_nums = {
    "nclarity": {"I1": 8,
                 "SI1": 7,
                 "SI2": 6,
                 "VS1": 5,
                 "VS2": 4,
                 "VVS1": 3,
                 "VVS2": 2,
                 "IF": 1}
}
diamonds.replace(clarity_nums, inplace=True)
```

[1] `http://plotnine.readthedocs.io/en/stable/generated/plotnine.data.diamonds.html`

As this is quite a large dataset, we'll limit it to just a few levels of clarity. This will make it easier to see what a weighted scatterplot can do.

```
sdiamond = diamonds[diamonds["nclarity"].isin([8, 5, 1])]
len(sdiamond)
```

Finally, as this is still quite a dense dataset even after restricting it to a few levels, we'll also take a sample of 500 datapoints to use for visualising.

```
sdiamond = sdiamond.sample(500)
```

The exact sample we've used in this chapter can be downloaded from here[2], or you can create your own extract of the data using the code above. You can load our sample data into Python like so:

```
sdiamond = pd.read_csv("https://git.io/JecIp")
```

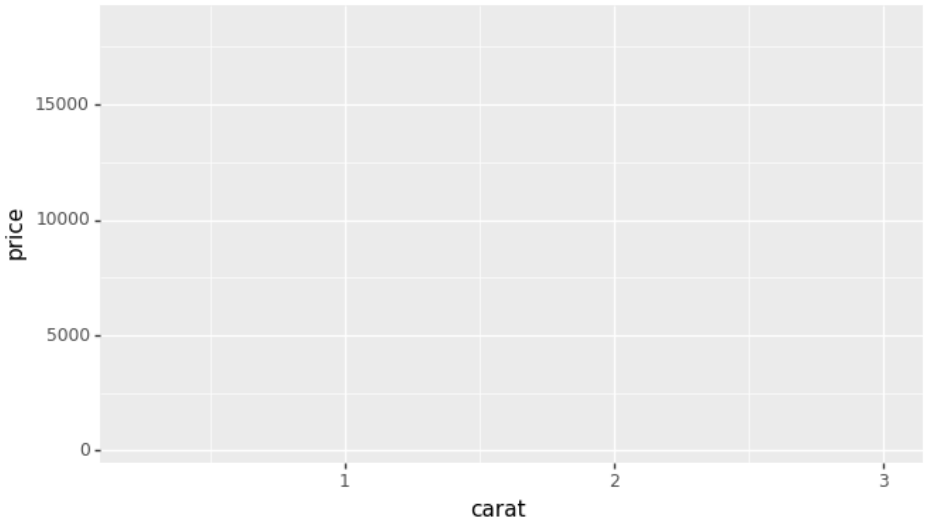
6.2 Basic ggplot structure

In order to initialise a weighted scatterplot we tell ggplot that `sdiamond` is our data, and specify that our x-axis plots the `carat` variable and our y-axis plots the `price` variable. You may have noticed that we put our x- and y-variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `carat` to the x-axis and `price` to the y-axis.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

```
p6 = ggplot(sdiamond, aes("carat", "price"))
p6
```

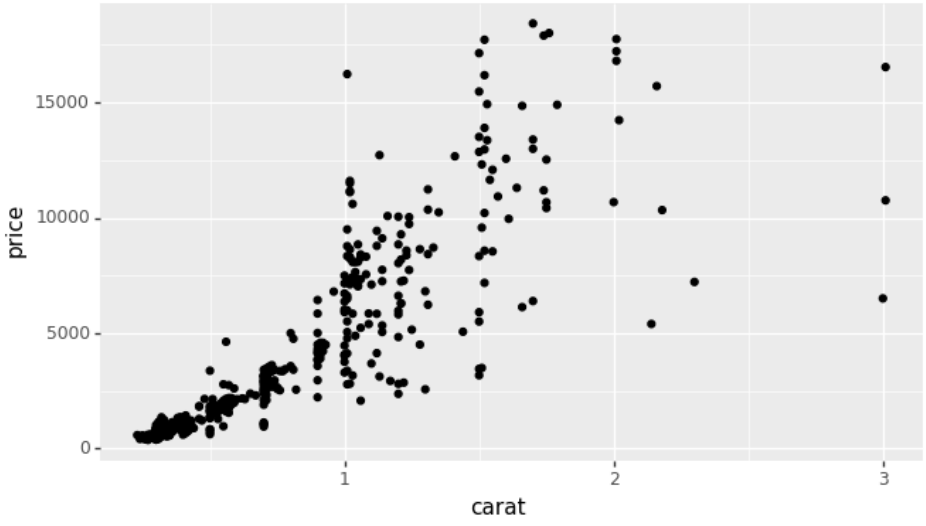
[2] `https://git.io/JecIp`



6.3 Basic weighted scatterplot

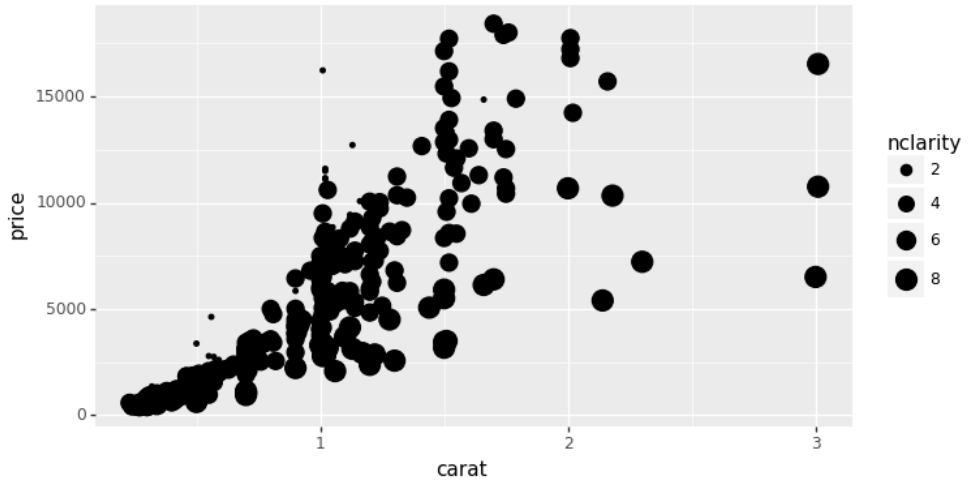
We can do this using `geoms`. In the case of a scatterplot, we use the `geom_point()` geom.

```
p6 = ggplot(sdiamond, aes("carat", "price")) + geom_point()
p6
```



In order to turn this into a weighted scatterplot, we simply add the size argument to `ggplot(aes())`. In this case, we want to weight the points by the clarity (`nclarity`) variable.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity"))
      + geom_point()
)
p6
```



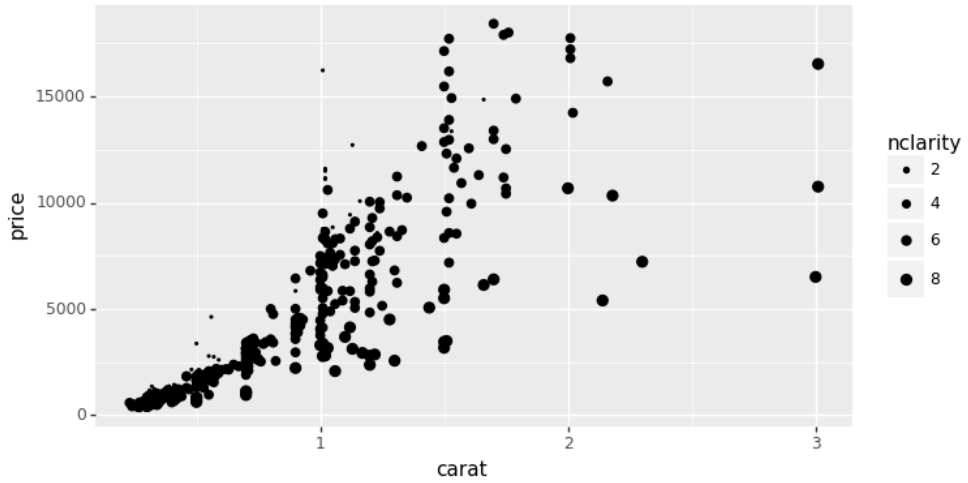
You can already see an interesting pattern emerge, where diamonds with more clarity tend to be worth more. Now let's make it beautiful!

6.4 Changing the shape of the data points

Perhaps we want the data points to be a different shape than a large circle. We can change these by adding the `shape` argument to `geom_point`. The shape arguments for `plotnine` are the same as those available in `matplotlib`, and are therefore a little more limited than those in R's implementation of `ggplot2`. Nonetheless, there is a good range of options. The allowed arguments are here[3]. Let's change all of the markers to a smaller version of a circle (called a `point` in the `matplotlib` documentation) using the argument `"."`.

[3] `https://matplotlib.org/api/markers_api.html`

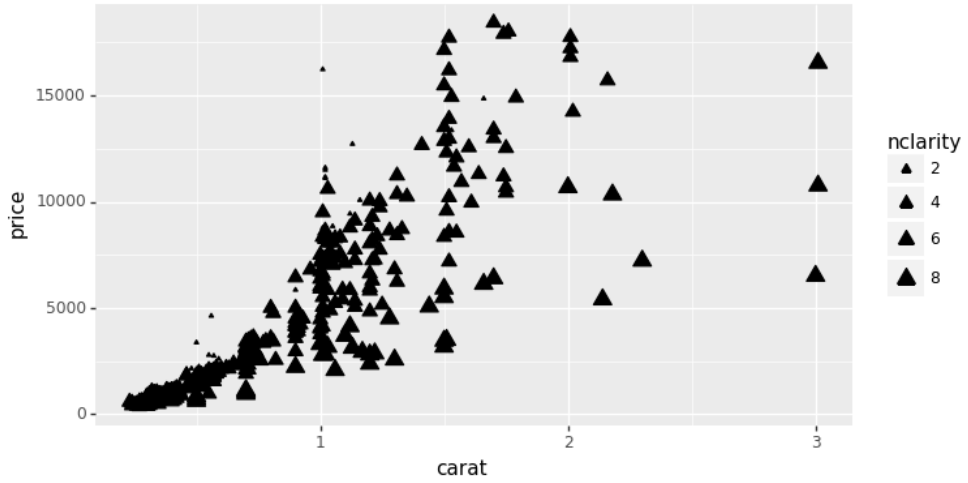
```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity"))
    + geom_point(shape=".")
)
p6
```



We can also change the shape by passing a tuple to the `shape` argument, which will render a polygon. The tuple has 3 arguments (`numsides`, `style`, `angle`), and the parameters for these can be seen here[4]. In this example, we've created a 3-sided regular polygon (i.e., a triangle) by passing the tuple `(3,0)`.

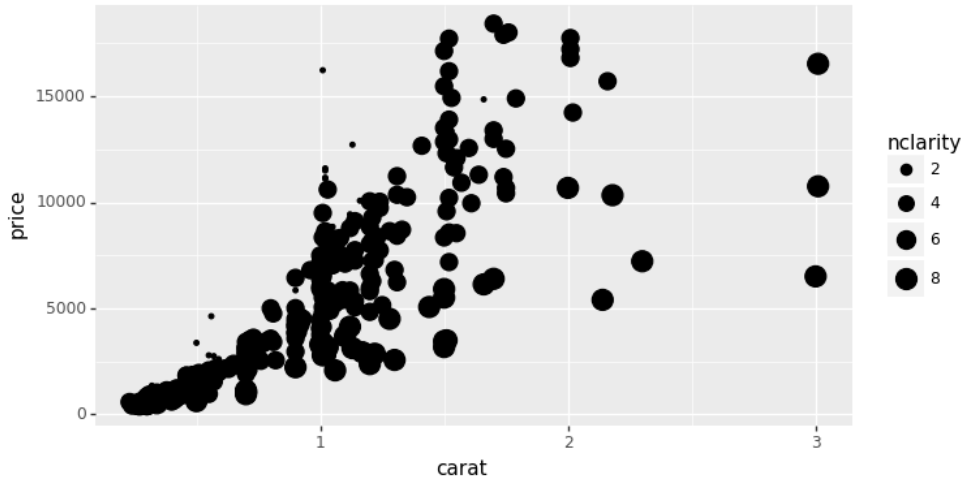
```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity"))
      + geom_point(shape=(3, 0))
)
p6
```

[4] `https://matplotlib.org/api/markers_api.html`



However, the default circle shape will work best with a weighted scatterplot, so we will go back to using this. We can do this using the argument `"o"`.

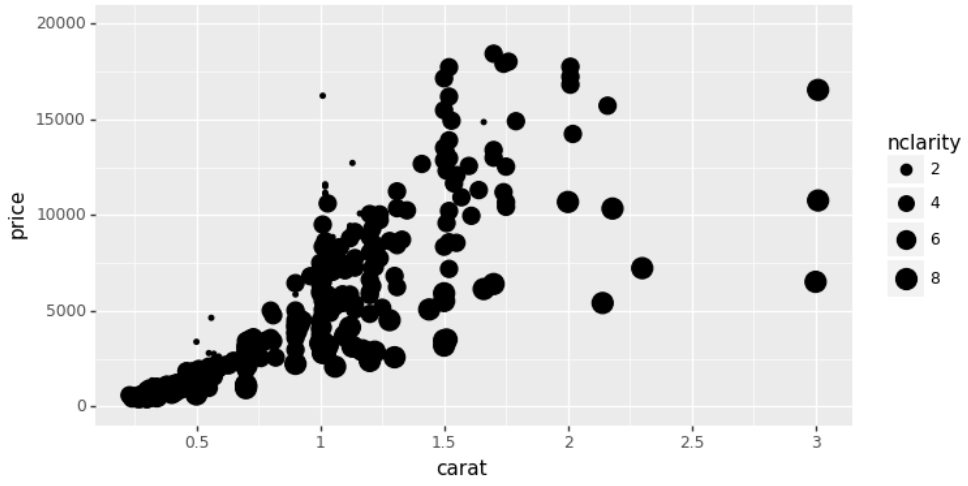
```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity"))
    + geom_point(shape="o")
)
p6
```



6.5 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, to change the y-axis we use the `scale_y_continuous` option. Here we will change the x-axis to every 0.5 carats, rather than 1, and change the range of the y-axis from 0 to 20,000. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[5] which generates a sequence from your selected start, stop and step values respectively. Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
)
p6
```

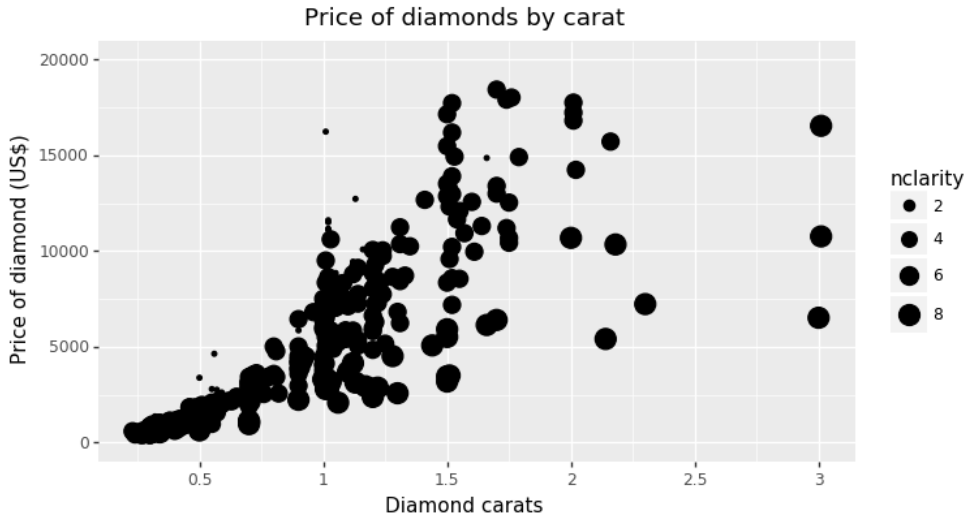


[5] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html`

6.6 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p6
```

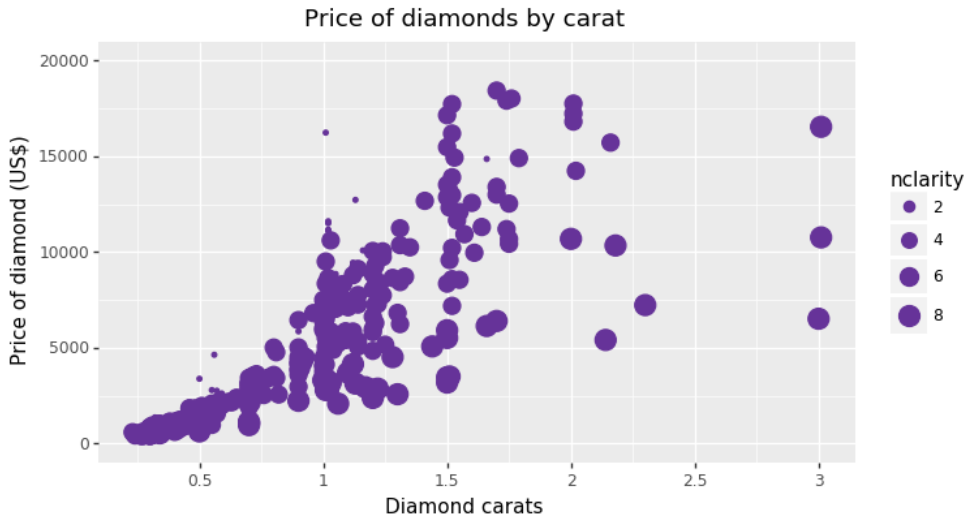


6.7 Adjusting the colour palette

There are a few options for adjusting the colour. The most simple is to make every point one fixed colour. As with shape, `plotnine` uses the colour palette utilised by `matplotlib`. The full set of named colours recognised by `ggplot` is here[6]. Let's try changing our shapes to `rebeccapurple`.

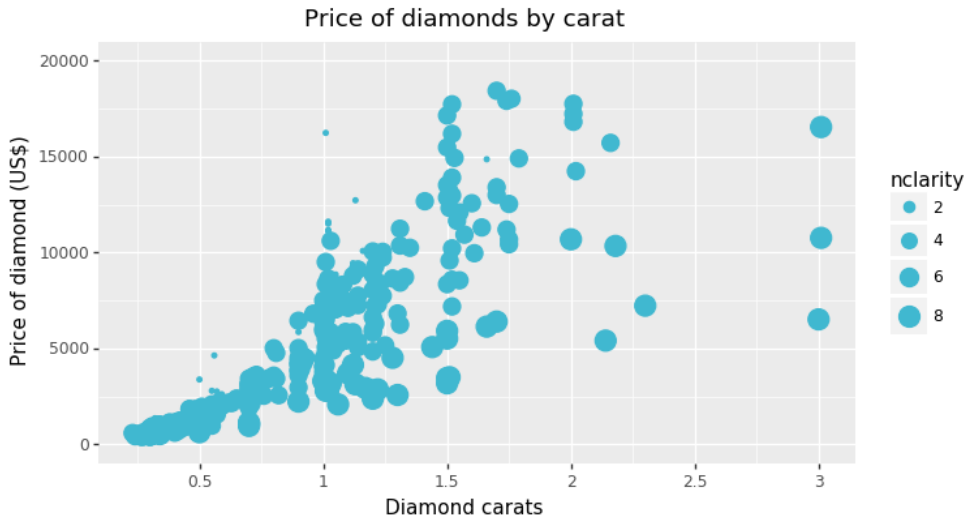
[6] `https://matplotlib.org/examples/color/named_colors.html`

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity"))
    + geom_point(shape="o", colour="rebeccapurple")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p6
```



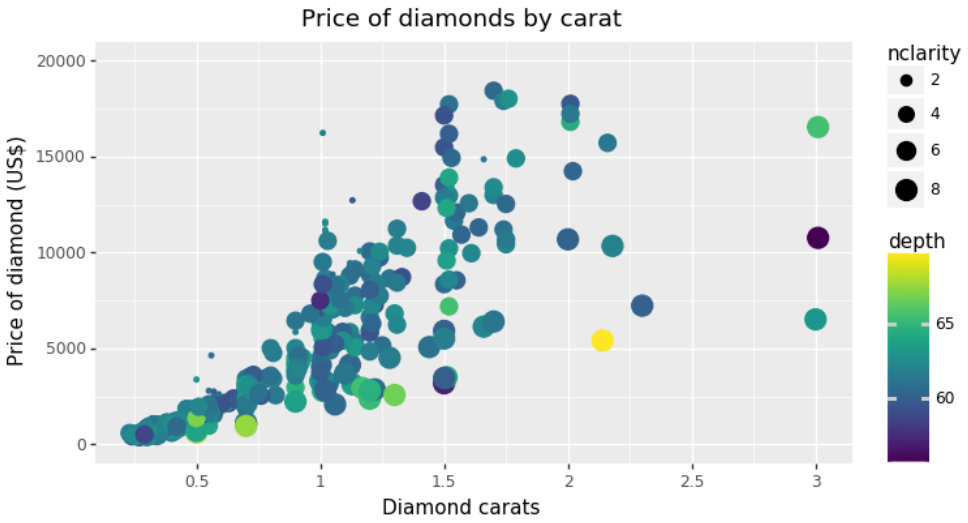
You can change the colours using specific HEX codes instead. Here we have made the shapes `'#40b8d0'` (vivid cyan).

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity"))
    + geom_point(shape="o", colour="#40b8d0")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p6
```



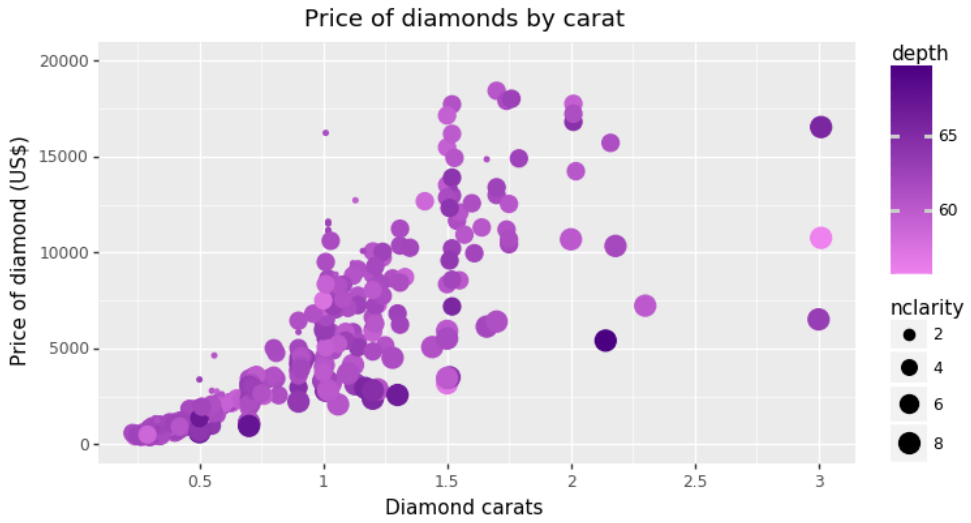
You can also change the colour of the data points according to the levels of another variable. This can be done either as a continuous gradient, or as a levels of a factor variable. Let's change the colour by the values a continuous variable, `depth`, by adding it to our aesthetic mapping, `aes`:

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="depth"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p6
```



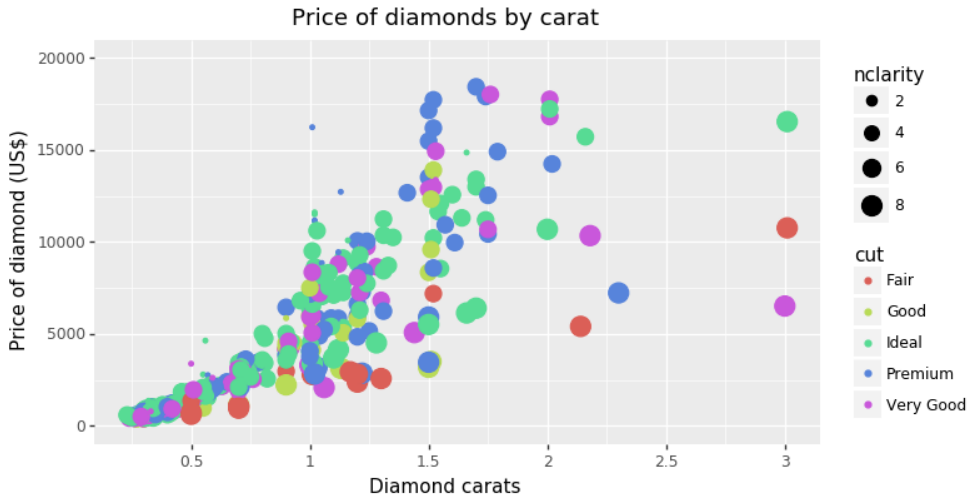
We can change the gradient's colours by adding the `scale_colour_gradient` option. The `low` and `high` arguments specify the range of colours the gradient should transition between.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="depth"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_gradient(low="violet", high="indigo")
)
p6
```



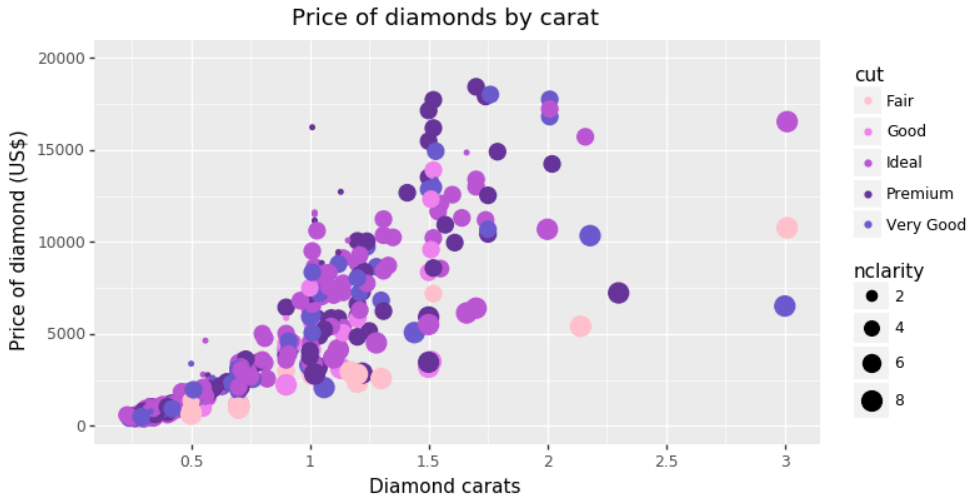
Let's now change the colours of the data points by a categorical variable, `cut`.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
)
p6
```



Again, we can change the colours of these data points, this time using `scale_colour_manual`.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_manual(
        values=["pink", "violet", "mediumorchid",
                "rebeccapurple", "slateblue"]
    )
)
p6
```

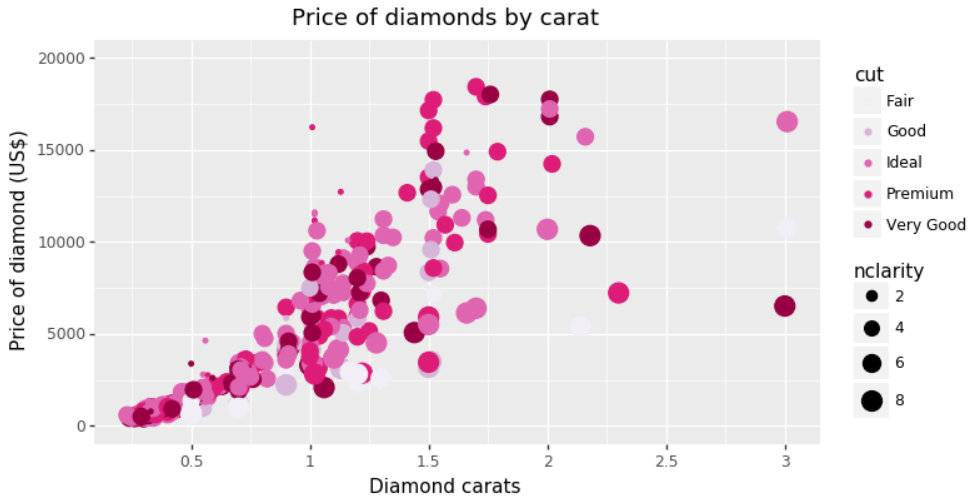


We can also change the manual colours using the schemes from ColorBrewer[7]. Here we have used the `scale_colour_brewer` option with the sequential scale `PuRd`. This gives us a nice transition between purple and red colour points. More information on using `scale_colour_brewer` is here[8].

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity", colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_brewer(type="seq", palette="PuRd")
)
p6
```

[7] http://colorbrewer2.org/

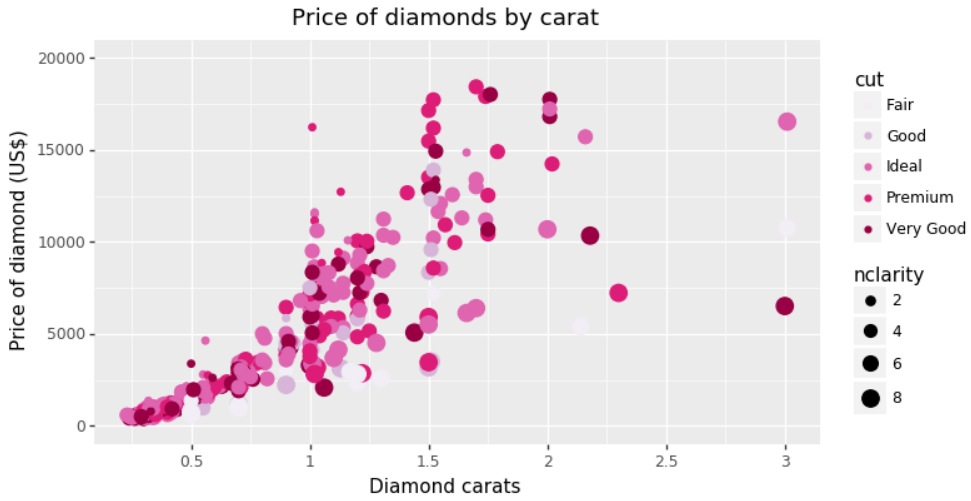
[8] http://plotnine.readthedocs.io/en/stable/generated/plotnine.scales.scale_color_brewer.html#plotnine.scales.scale_color_brewer



6.8 Adjusting the size of the data points

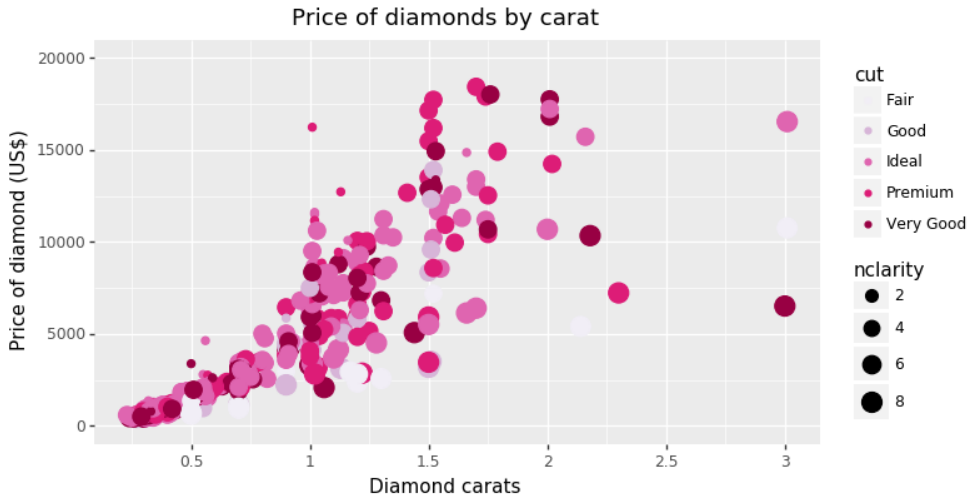
The default size of the the data points in a weighted scatterplot is mapped to the radius of the plots. If we want the data points to be proportional to the value of the weighting variable (e.g., a diamond with a clarity of 0 would have a value of 0), we need to use the `scale_size_area` option.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_brewer(type="seq", palette="PuRd")
    + scale_size_area(max_size=5)
)
p6
```



Alternatively, we can adjust the size of the data points by using `scale_size` and specifying a desired range.

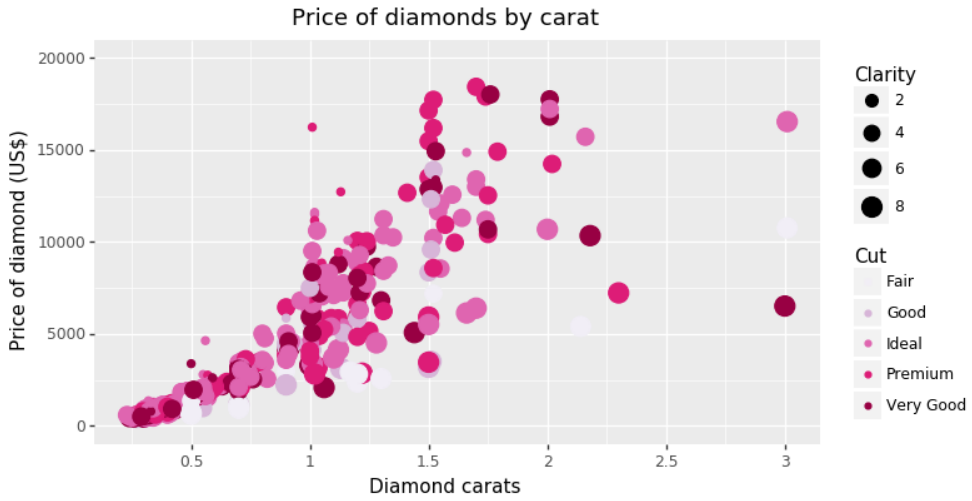
```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_colour_brewer(type="seq", palette="PuRd")
    + scale_size(range=(2, 6))
)
p6
```



6.9 Adjusting the legend

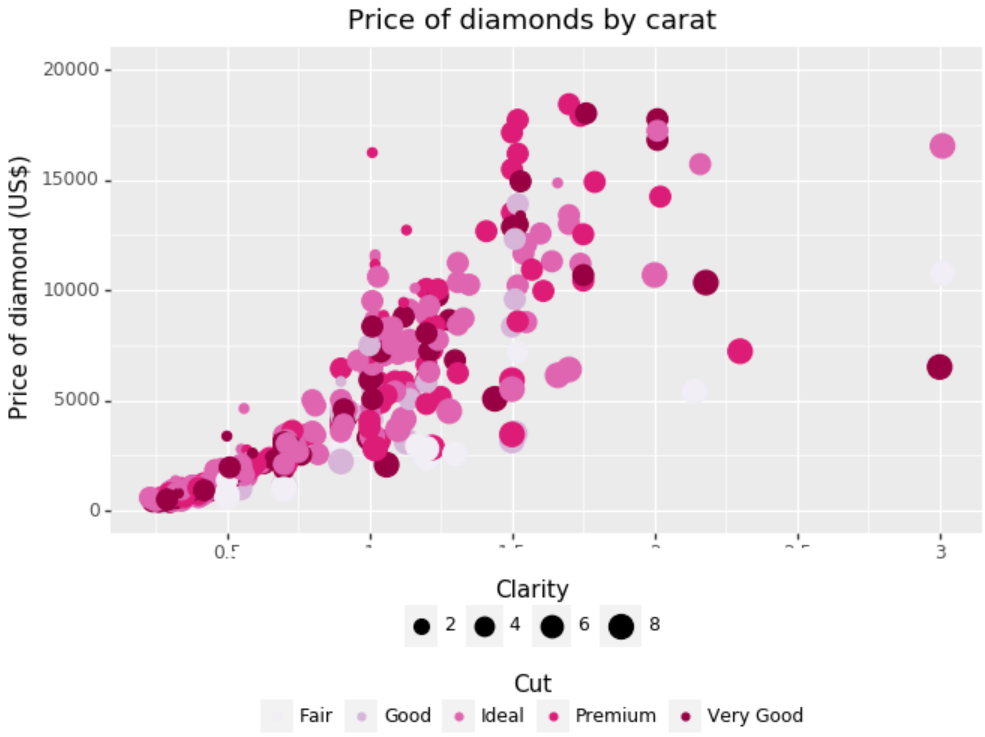
To change the title of the legend, we can use the `labs` option. In order to change the `nclarity` legend, you use the same argument you used in `aes` when you initialised your chart. As you can see, we've used the `size` argument to change the `Clarity` legend title. As we've used `scale_colour_brewer` to change the colour theme of `cut`, we have to add the `name="Cut"` argument to this method to change its name in the legend.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_size(range=(2, 6))
    + scale_colour_brewer(type="seq", palette="PuRd", name="Cut")
    + labs(size="Clarity")
)
p6
```



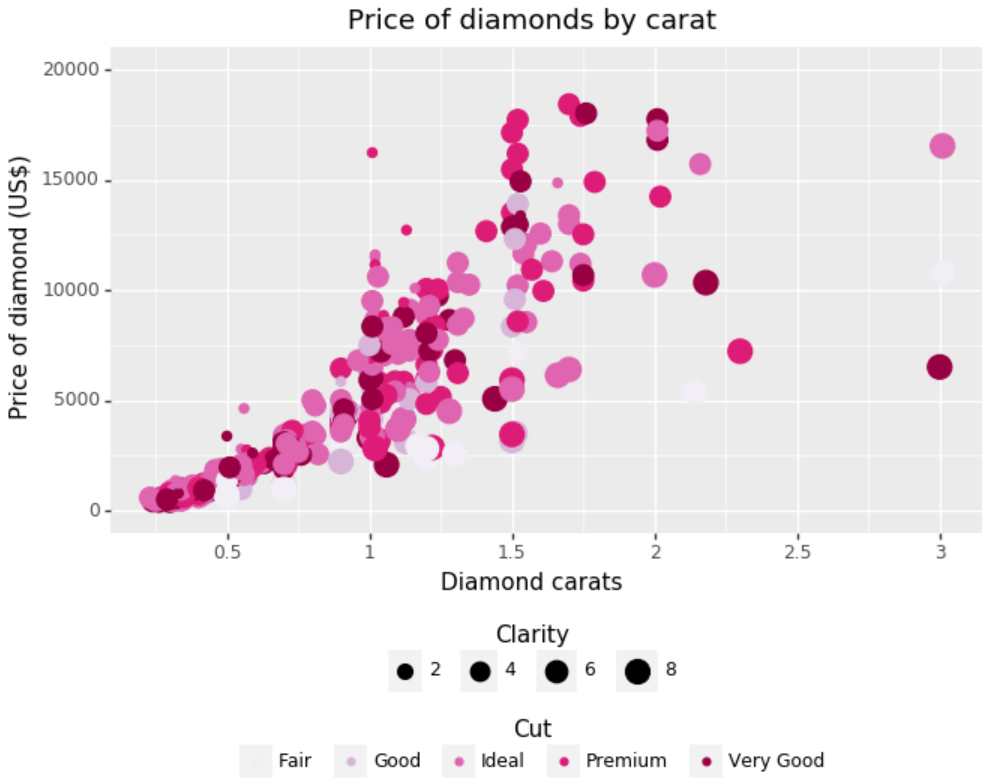
To adjust the position of the legends from the default spot of right of the graph, we add the `theme` option and specify the `legend_position="bottom"` argument. We can also change the legend shapes using the `legend_direction="horizontal"` argument. Finally, we can centre the legends' title positions using the argument `legend_title_align="center"`.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_size(range=(2, 6))
    + scale_colour_brewer(type="seq", palette="PuRd", name="Cut")
    + labs(size="Clarity")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
    )
)
p6
```



Unfortunately, the legends are now sitting over our x-axis. We can fix this using the `legend_box_spacing=0.4` option within `theme`. This allows us to move the legends down as much as needed. The legends are also looking a little squashed. We can space it out a bit more using the `legend_entry_spacing_x=10` argument.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats") + ylab("Price of diamond (US$)")
    + scale_size(range=(2, 6))
    + scale_colour_brewer(type="seq", palette="PuRd", name="Cut")
    + labels(size="Clarity")
    + theme(
        legend_position="bottom", legend_direction="horizontal",
        legend_title_align="center", legend_box_spacing=0.4,
        legend_entry_spacing_x=10))
p6
```

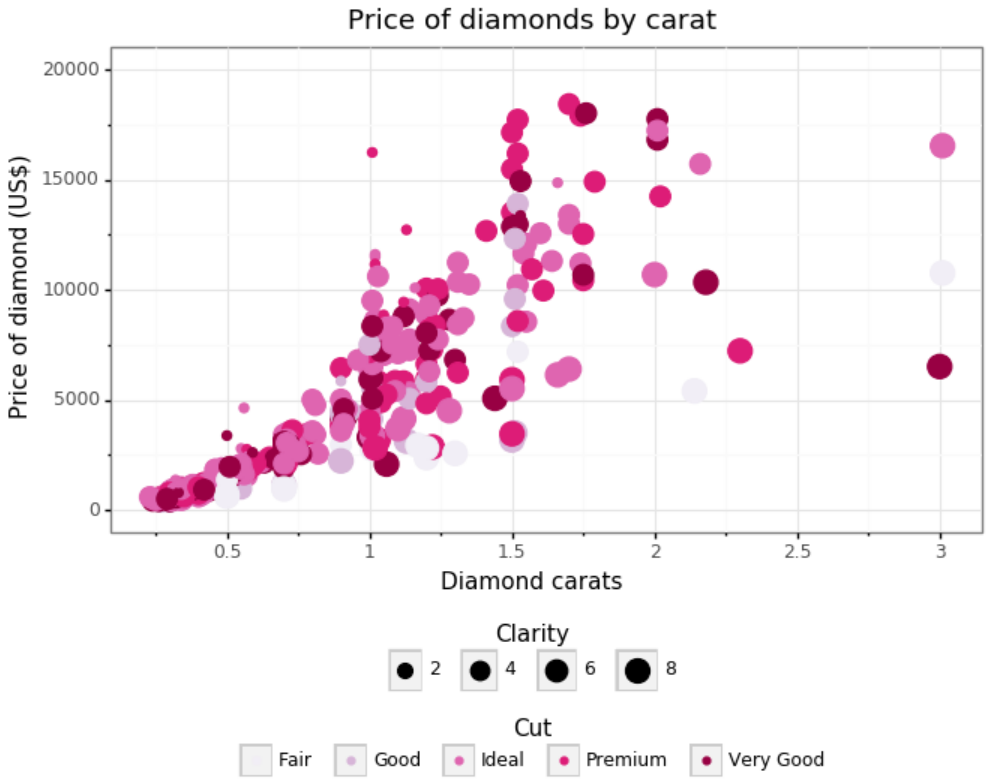


6.10 Using the white theme

As explained in the previous chapters, we can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after `ggplot()`. As you can see, we can further tweak the graph using the `theme` option, which we've used so far to change the legend.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Diamond carats")
    + ylab("Price of diamond (US$)")
    + scale_size(range=(2, 6))
    + scale_colour_brewer(type="seq", palette="PuRd", name="Cut")
```

```
    + labs(size="Clarity")
    + theme_bw()
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        legend_entry_spacing_x=10,
    )
)
p6
```



6.11 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead

created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[9]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[10]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight()` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

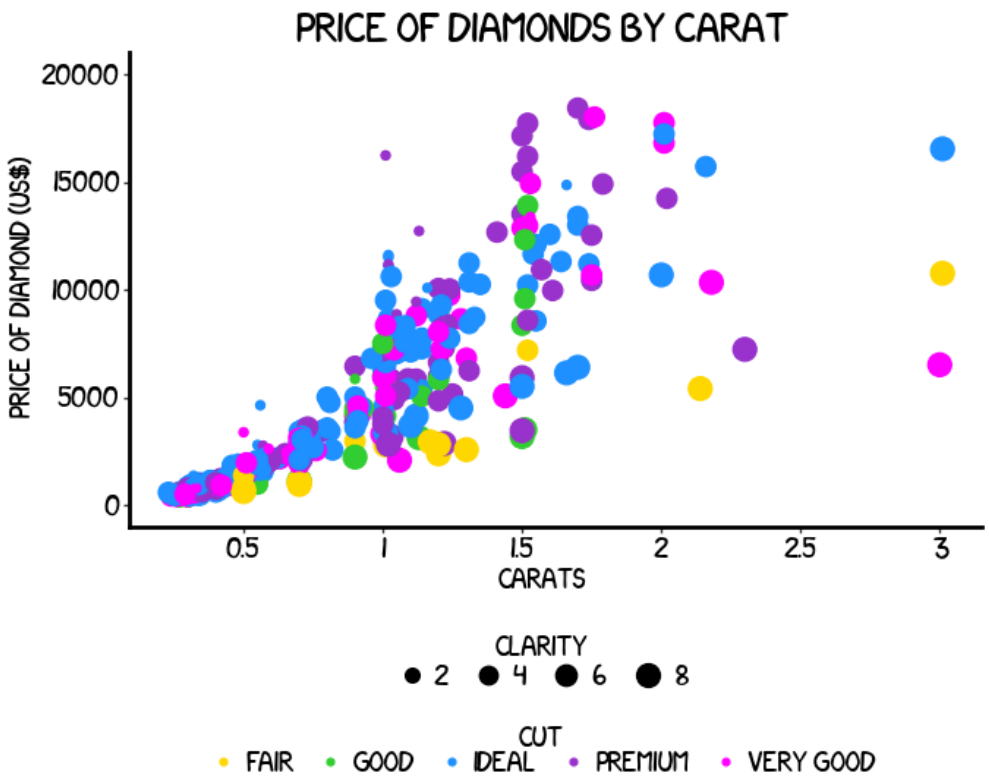
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;

[9] `xkcd.com/1350/xkcd-Regular.otf`
[10] `https://matplotlib.org/api/font_manager_api.html`

- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Carats")
    + ylab("Price of diamond (US$)")
    + scale_size(range=(2, 6))
    + labs(size="Clarity", colour="Cut")
    + scale_colour_manual(
        values=["gold", "limegreen", "dodgerblue",
                "darkorchid", "magenta"]
    )
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.5,
        legend_entry_spacing_x=10,
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        legend_key=element_blank(),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p6
```



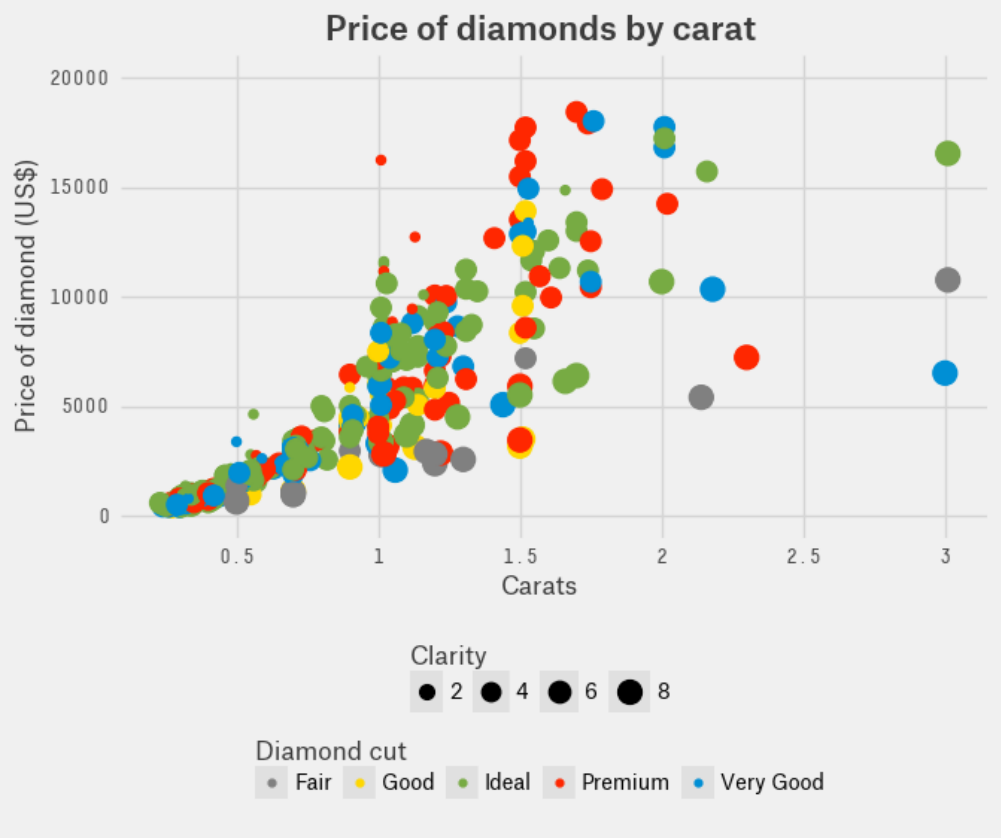
6.12 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[11]). Below we've applied `theme_538()`, which approximates graphs on the website FiveThirtyEight. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[12] and 'Decima Mono Pro'[13] in `legend_title`, `legend_text`, `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
```

[11] http://plotnine.readthedocs.io/en/stable/api.html#themes
[12] https://commercialtype.com/catalog/atlas/atlas_grotesk
[13] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
title_text = fm.FontProperties(fname=agm)
legend_text = fm.FontProperties(fname=agr)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
legend_text.set_size(10)
axis_text.set_size(12)
body_text.set_size(10)


p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Carats")
    + ylab("Price of diamond (US$)")
    + scale_size(range=(2, 6))
    + labs(size="Clarity", colour="Cut")
    + scale_colour_manual(
        values=["grey", "gold", "#77AB43", "#FF2700", "#008FD5"],
        name="Diamond cut"
    )
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.5,
        legend_title=element_text(fontproperties=axis_text),
        legend_text=element_text(fontproperties=legend_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p6
```



6.13 Creating your own theme

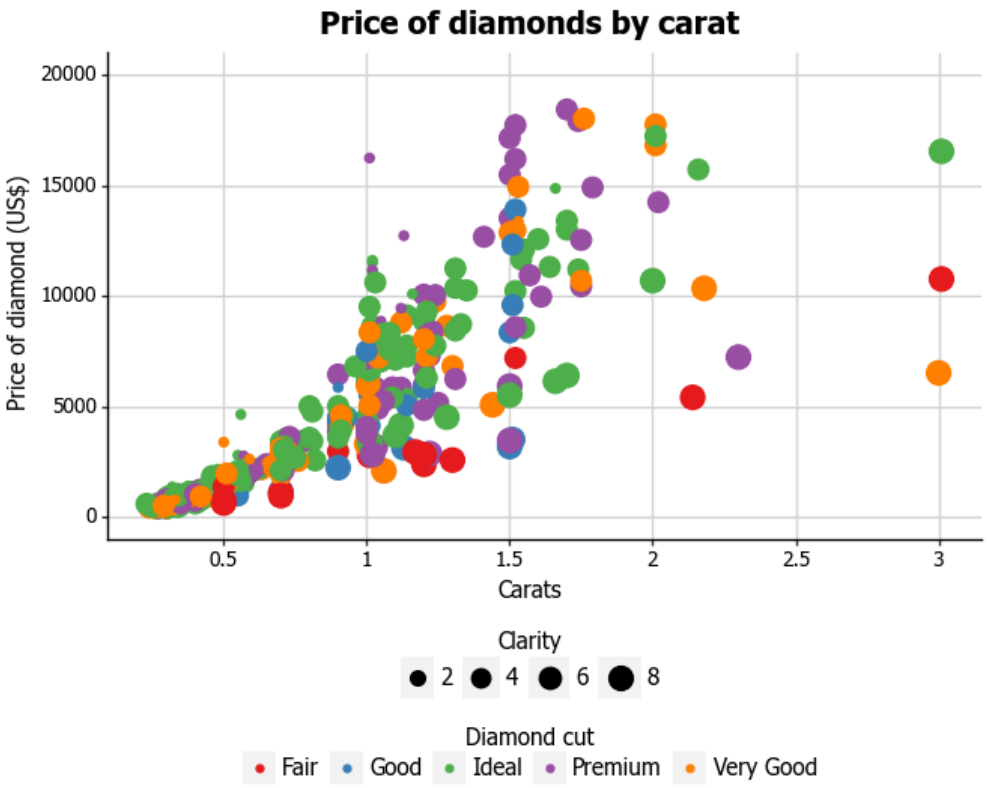
Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

We've also changed our colour scheme to another ColorBrewer theme, the quantitative scale `Set1`.

With all of these customisations, we now finally have the graph we presented at the beginning of this chapter.

```
p6 = (
    ggplot(sdiamond, aes("carat", "price", size="nclarity",
                         colour="cut"))
    + geom_point(shape="o")
    + scale_x_continuous(breaks=np.arange(0, 4, 0.5))
    + scale_y_continuous(limits=(0, 20000))
    + ggtitle("Price of diamonds by carat")
    + xlab("Carats")
    + ylab("Price of diamond (US$)")
    + scale_size(range=(2, 6))
    + labs(size="Clarity", colour="Cut")
    + scale_colour_brewer(type="qual", palette="Set1",
                          name="Diamond cut")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p6
```

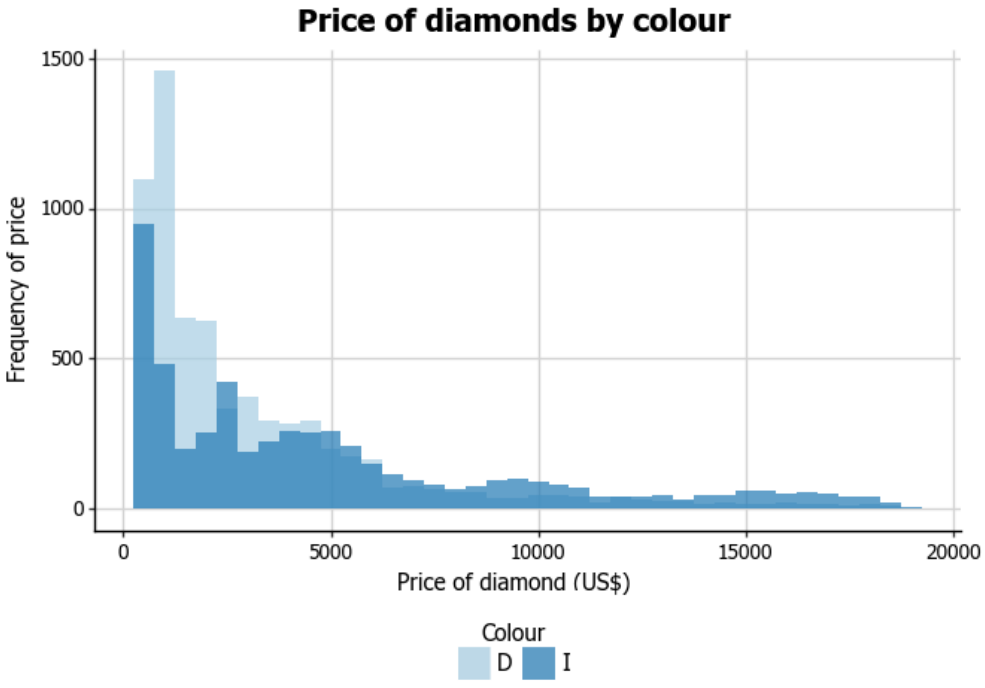

Price of diamonds by carat
Price of diamond (US$)
20000
15000
10000
5000
0
0.5
1
1.5
2
2.5
3
Carats
Clarity
2
4
6
8
Diamond cut
Fair
Good
Ideal
Premium
Very Good

Chapter 7

Histograms

7.1 Introduction

In this chapter, we will work towards creating the histogram below. We will take you from a basic histogram and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs;
- `numpy` to do some basic numeric calculations in our graphing; and
- `scipy.stats` to calculate a normal distribution curve.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd
import scipy.stats as st

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from plotnine import data
from pandas import DataFrame
```

For this example, we'll be using the diamonds dataset from plotnine's data module. The details of this dataset are here[1].

```
diamonds = data.diamonds
```

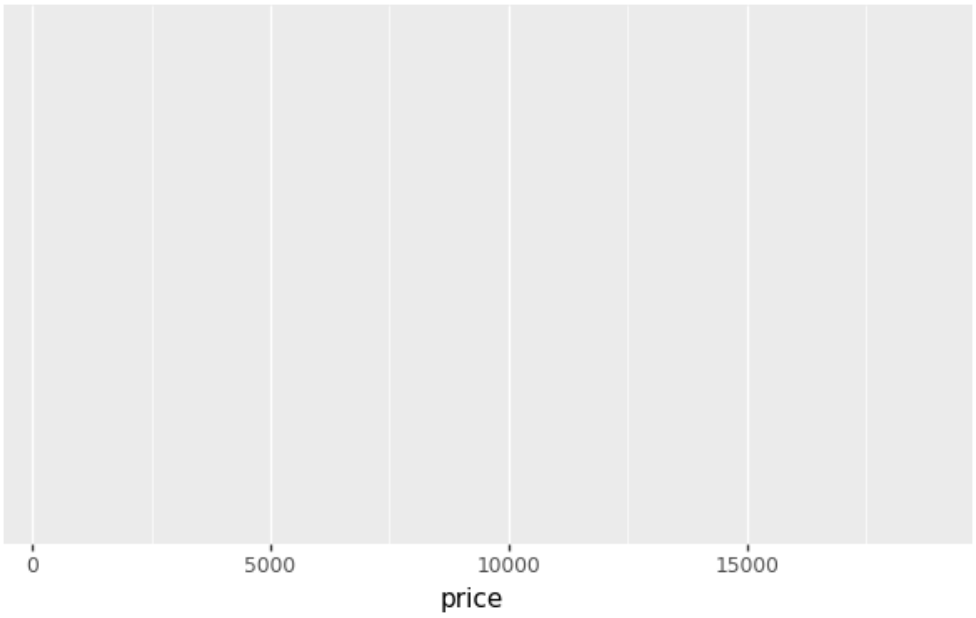
7.2 Basic ggplot structure

In order to initialise a scatterplot we tell ggplot that `diamonds` is our data, and specify that we want to plot the `price` variable (by argument position, this is assigned to the x-axis). You may have noticed that we put our x- and y-variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `price` to the x-axis and the y-axis is currently blank.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

[1] `http://plotnine.readthedocs.io/en/stable/generated/plotnine.data.diamonds.html`

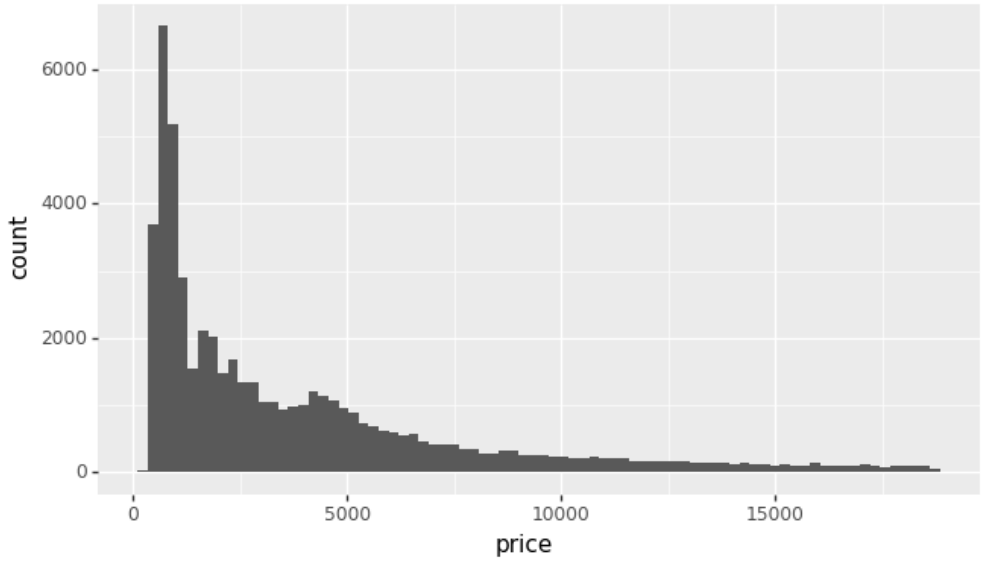
```
p7 = ggplot(diamonds, aes("price"))
p7
```



7.3 Basic histogram

We can do this using `geoms`. In the case of a histogram, we use the `geom_histogram()` geom.

```
p7 = ggplot(diamonds, aes("price")) + geom_histogram()
p7
```

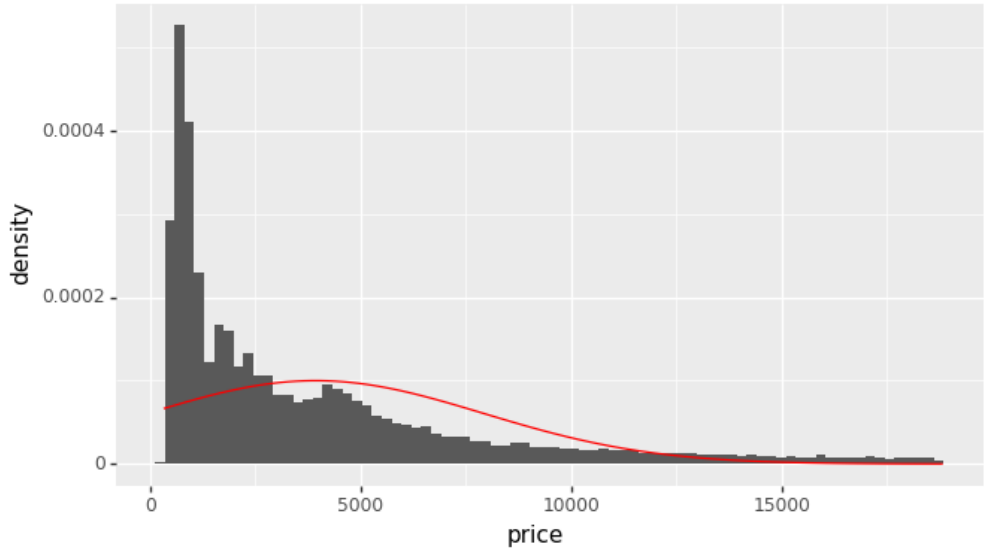


7.4 Adding a normal density curve

We can overlay a normal density function curve on top of our histogram to see how closely (or not) it fits a normal distribution. In this case, we can see it deviates from a normal distribution, showing marked positive skew. In order to overlay the function curve, we first need to import `scipy.stats`. We can then add the option `stat_function(fun=st.norm.pdf)` from this package. We need to tell this function what the mean and standard deviation of the distribution should be, using the `loc` and `scale` arguments. We can use the `pandas` functions `mean()` and `std()` on our price variable in order to set these arguments. If you have missing data, make sure you pass the argument `skipna=True` to the mean and sd parameters. Finally, you can change the colour using the `colour="red"` argument. We will discuss how to customise colours further below.

One further change we must make to display the normal curve correctly is adding `aes(y='..density..')` to the `geom_histogram` option. Note that the normal density curve will not work if you are using the frequency rather than the density, which we are changing in our next step.

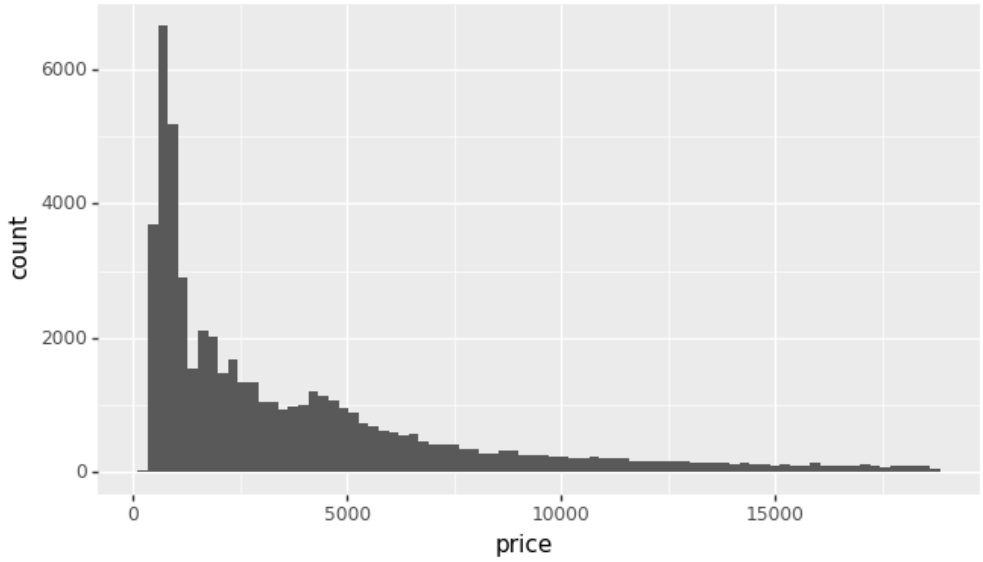
```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(y="..density.."))
    + stat_function(
        fun=st.norm.pdf,
        args=dict(loc=diamonds["price"].mean(),
                  scale=diamonds["price"].std()),
        colour="red",
    )
)
p7
```



7.5 Changing from density to frequency

Let's go back to the basic plot and lose the function curve. To change the y-axis from density to frequency, we add the `aes(y="..count..")` option to `geom_histogram`. You might have noticed that this is the default option for histograms in `plotnine`.

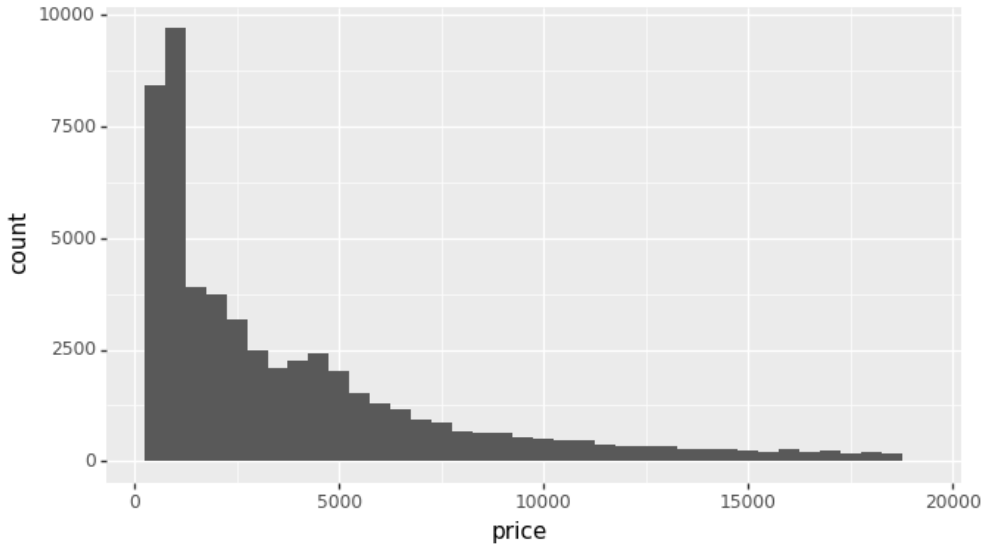
```
p7 = (
    ggplot(diamonds, aes("price"))
      + geom_histogram(aes(y="..count.."))
)
p7
```



7.6 Adjusting binwidth

To change the binwidth, we add a `binwidth` argument to `geom_histogram`. In this case, we will make binwidth 500 units of the `price` variable - that is, in bins of $500.

```
p7 = (
    ggplot(diamonds, aes("price"))
      + geom_histogram(aes(y="..count.."),
                       binwidth=500)
)
p7
```

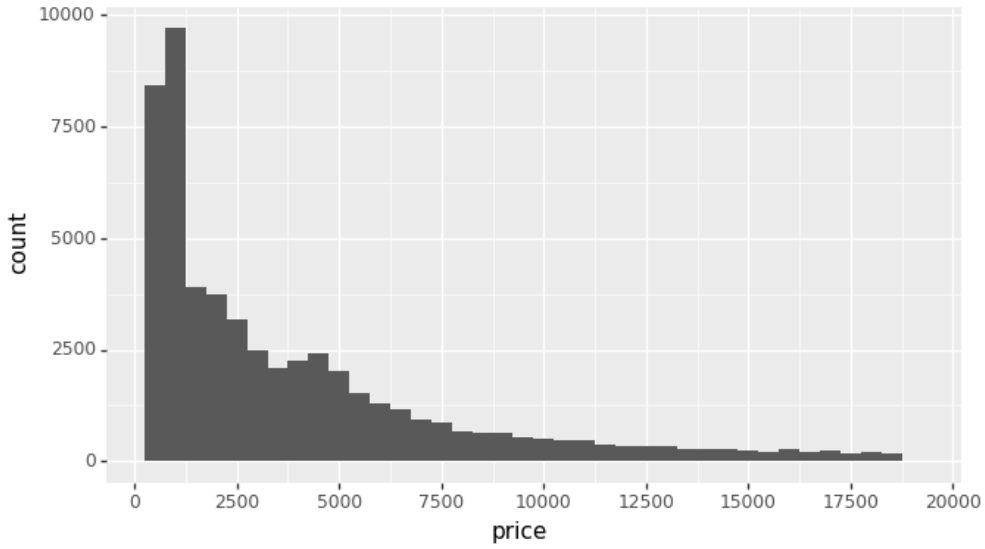


7.7 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, if we wanted to change the y-axis we could use the `scale_y_continuous` option. Here we will change the x-axis to every $2500, rather than every $5000. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[2] which generates a sequence from your selected start, stop and step values respectively. Alternatively if you just want to change the minimum and maximum values you can use the `limits` argument to define these (passing these values as a list).

```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(y="..count.."), binwidth=500)
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
)
p7
```

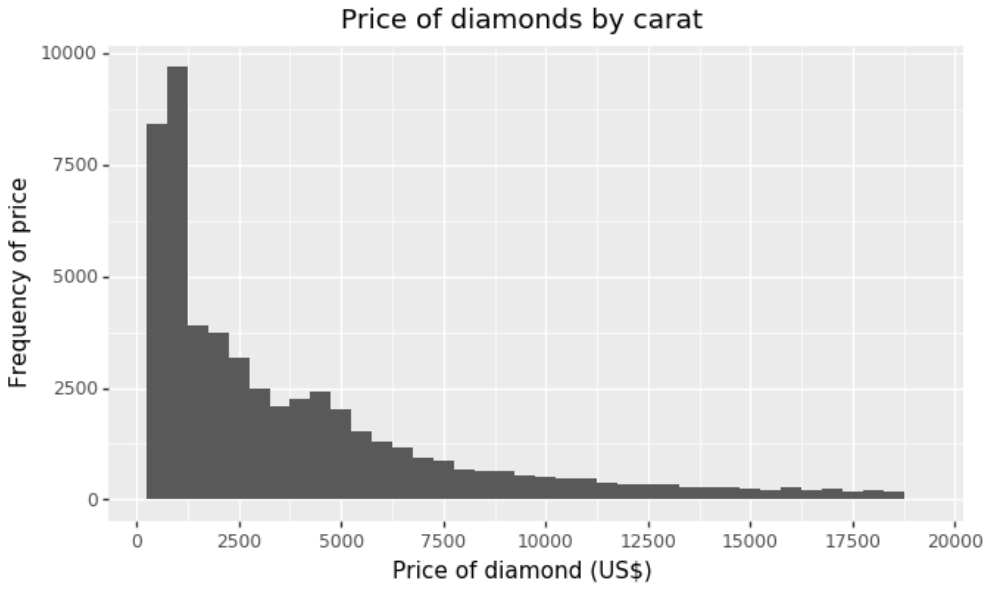
[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html



7.8 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(y="..count.."), binwidth=500)
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
)
p7
```

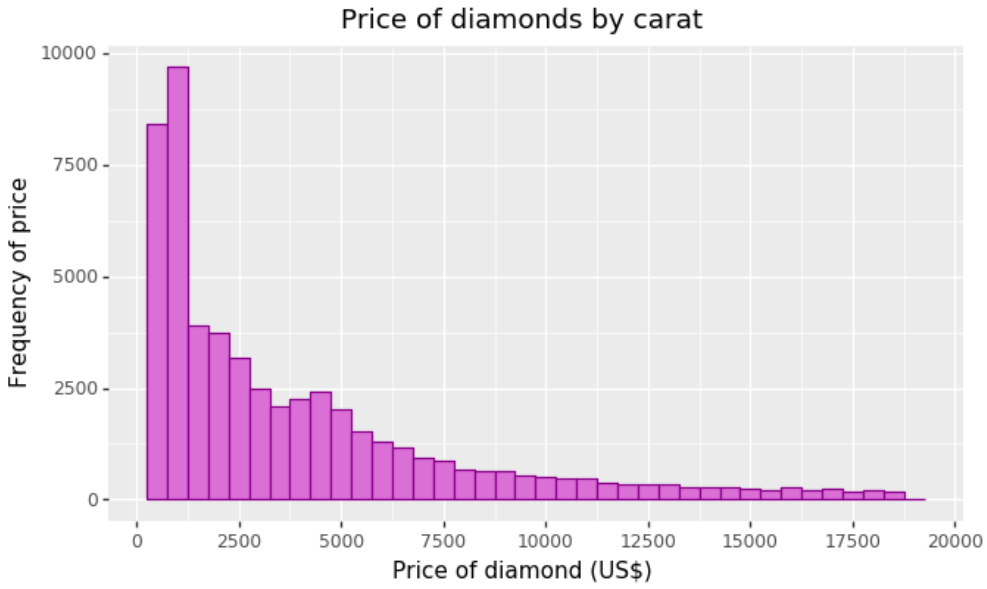


7.9 Adjusting the colour palette

There are a few options for adjusting the colour. The most simple is to make every bar one fixed colour. `plotnine` uses the colour palette utilised by `matplotlib`, and the full set of named colours recognised by `ggplot` is here[3]. Let's try changing our bars to two shades of purple. To do so, we need to add both the `fill` and `colour` options to `geom_histogram`. Of course, to make the bars a single colour, you would just assign the same colour name to both `fill` and `colour`.

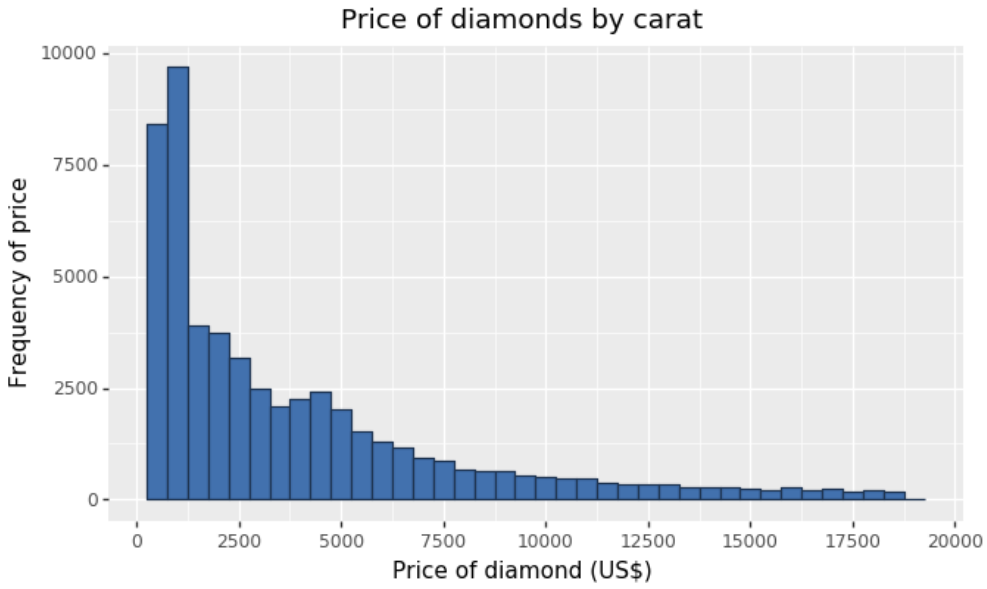
```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(
        aes(y="..count.."), binwidth=500,
        colour="darkmagenta", fill="orchid"
    )
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
)
p7
```

[3] https://matplotlib.org/examples/color/named_colors.html



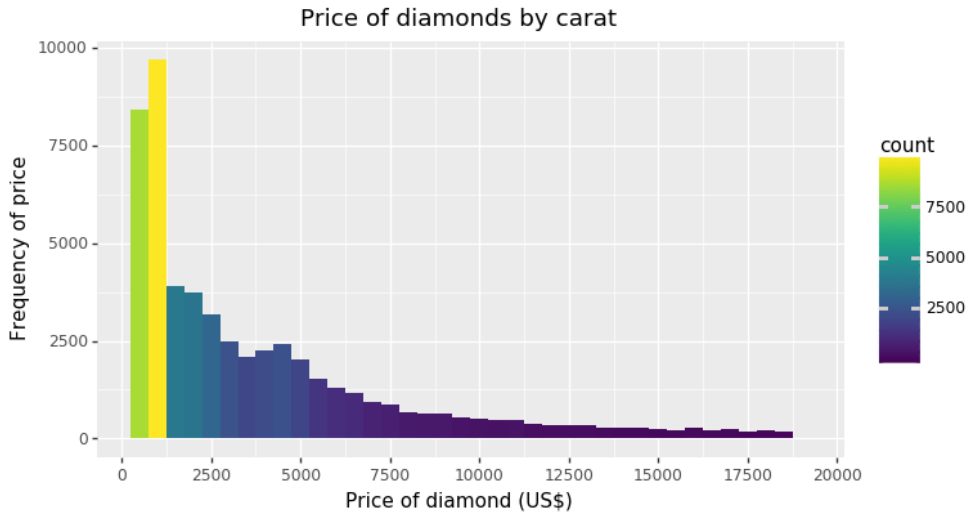
If you want to go beyond the options in the list above, you can also specify exact HEX colours by including them as a string preceded by a hash, e.g., `"#FFFFFF"`. Below, we have called two shades of blue for the fill and lines using their HEX codes.

```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(y="..count.."), binwidth=500,
                     colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
)
p7
```



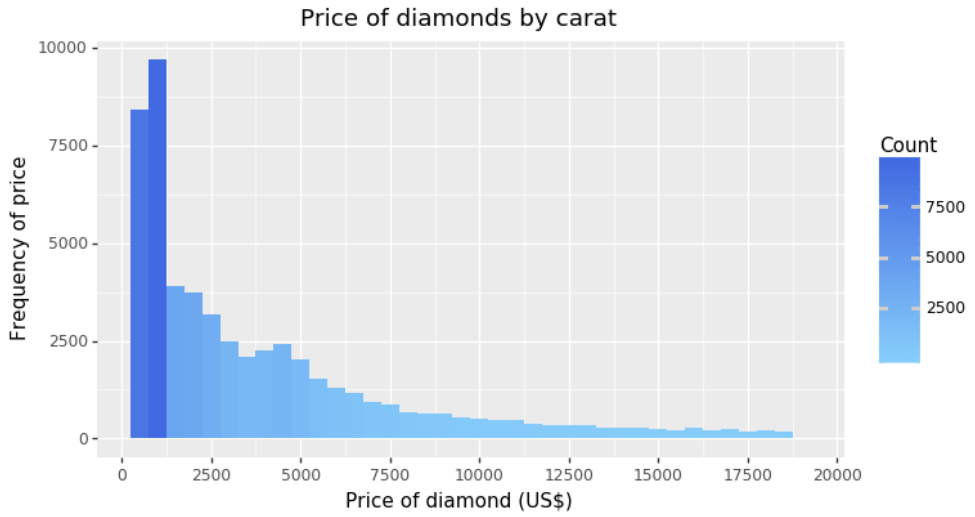
You can also add a gradient to your colour scheme that varies according to the frequency of the values. Below is the default gradient colour scheme. In order to do this, you can see we have changed the `aes(y="..count..")` argument in `geom_histogram` to `aes(fill="..count..")`.

```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(fill="..count.."), binwidth=500)
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
)
p7
```



You can customise the gradient by changing the anchoring colours for high and low. To do so, we have added the option `scale_fill_gradient` to the plot with the arguments `Count` (the name of the legend), `low` (the colour for the least frequent values) and `high` (the colour for the most frequent values).

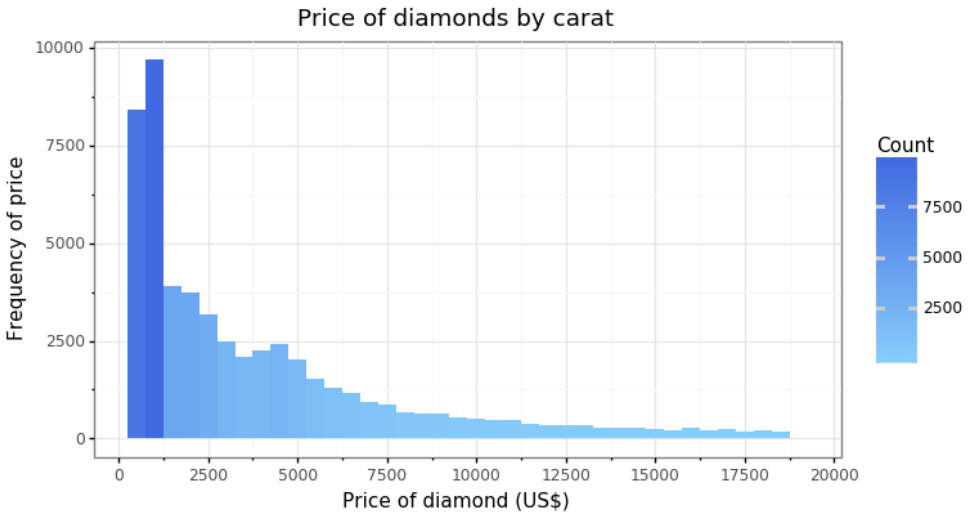
```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(fill="..count.."), binwidth=500)
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
    + scale_fill_gradient(name="Count", low="lightskyblue",
                          high="royalblue")
)
p7
```



7.10 Using the white theme

We can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after `ggplot()`.

```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(fill="..count.."), binwidth=500)
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
    + scale_fill_gradient(name="Count", low="lightskyblue",
                          high="royalblue")
    + theme_bw()
)
p7
```



7.11 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[4]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[5]). For the title, we'll change the font to size 18 and make it bold using

[4] `xkcd.com/1350/xkcd-Regular.otf`
[5] `https://matplotlib.org/api/font_manager_api.html`

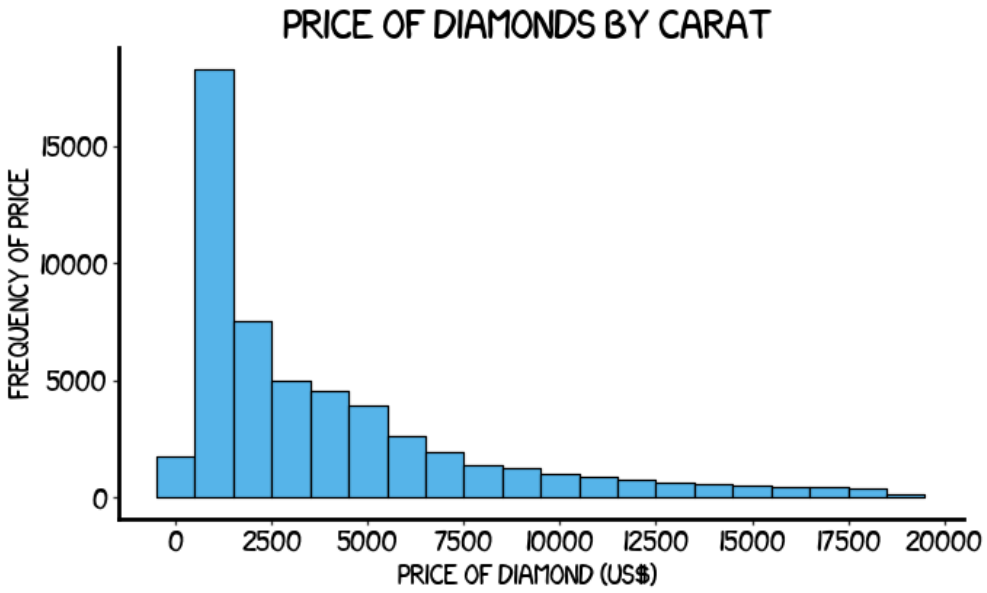
the `set_size()` and `set_weight()` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")
body_text.set_size(12)
```

In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(y="..count.."), binwidth=1000,
                     colour="black", fill="#56B4E9")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)") + ylab("Frequency of price")
    + theme(
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black")))
p7
```



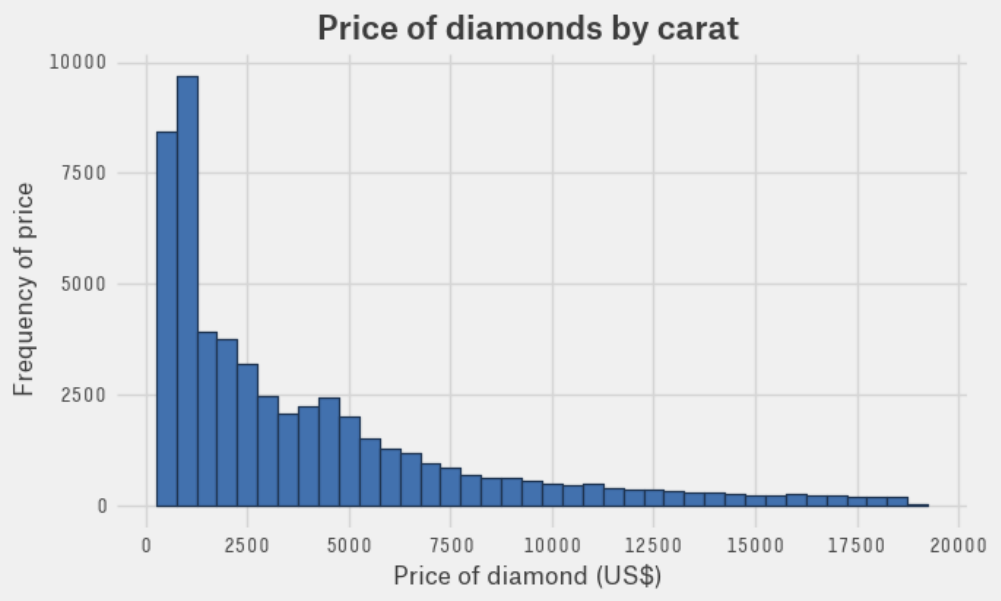
7.12 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[6]). Below we've applied `theme_538()`, which approximates graphs on the website FiveThirtyEight. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[7] and 'Decima Mono Pro'[8] in `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
```

[6] http://plotnine.readthedocs.io/en/stable/api.html#themes
[7] https://commercialtype.com/catalog/atlas/atlas_grotesk
[8] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
axis_text.set_size(12)
body_text.set_size(10)

p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(y="..count.."), binwidth=500,
                     colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p7
```

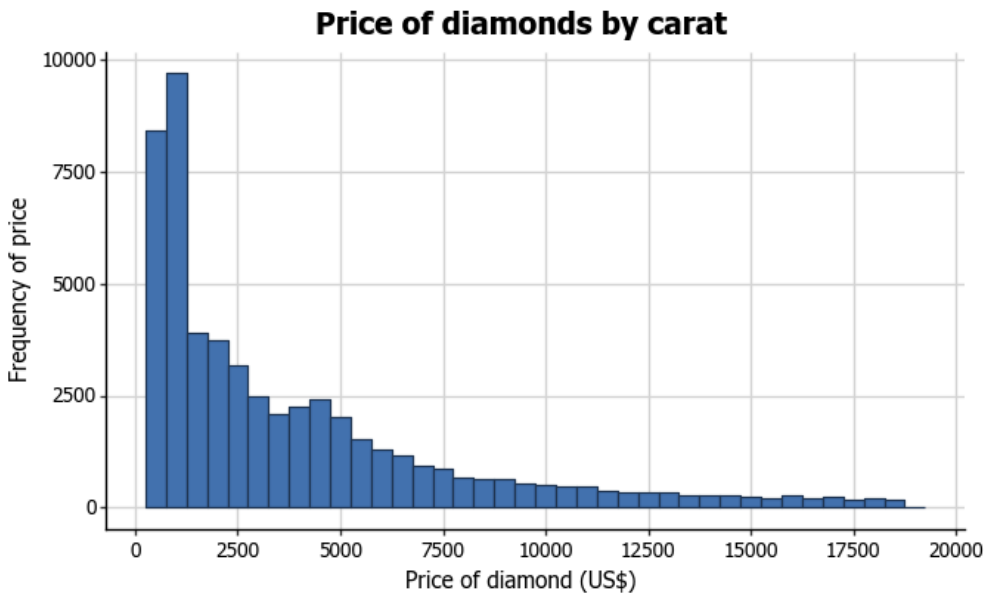


7.13 Creating your own theme

Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

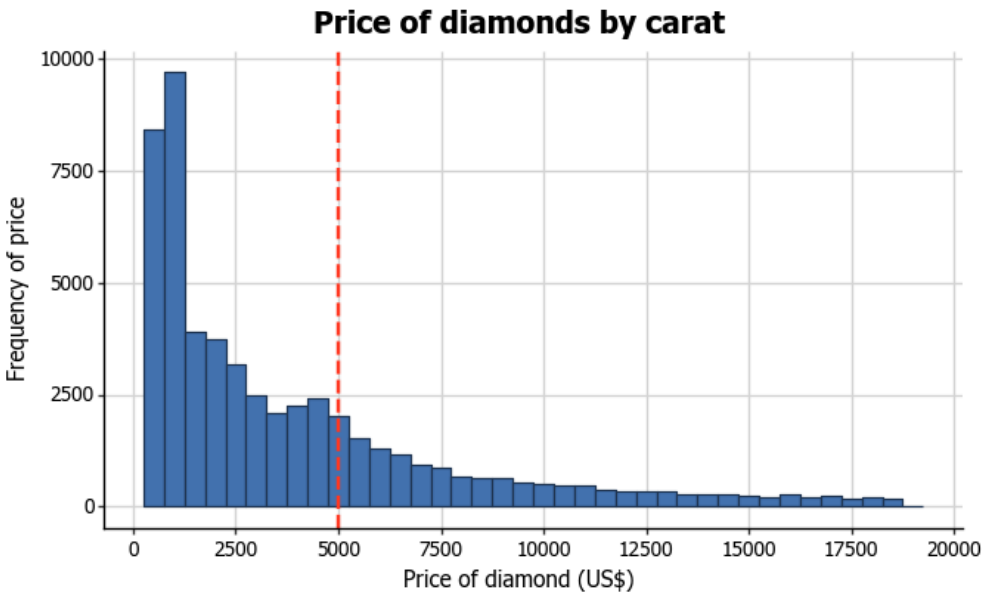
```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(y="..count.."), binwidth=500,
                     colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)") + ylab("Frequency of price")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10)))
p7
```



7.14 Adding lines

Let's say that we want to add a cutoff value to the chart (at $5000). We add the `geom_vline` option to the chart, and specify where it goes on the x-axis using the `xintercept` argument. We can customise how it looks using the `colour` and `linetype` arguments in `geom_vline`. (In the the same way, horizontal lines can be added using the `geom_hline`.)

```
p7 = (
    ggplot(diamonds, aes("price"))
    + geom_histogram(aes(y="..count.."), binwidth=500,
                     colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
    + geom_vline(xintercept=5000, size=1, colour="#FF3721",
                 linetype="dashed")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10)
    )
)
p7
```



7.15 Multiple histograms

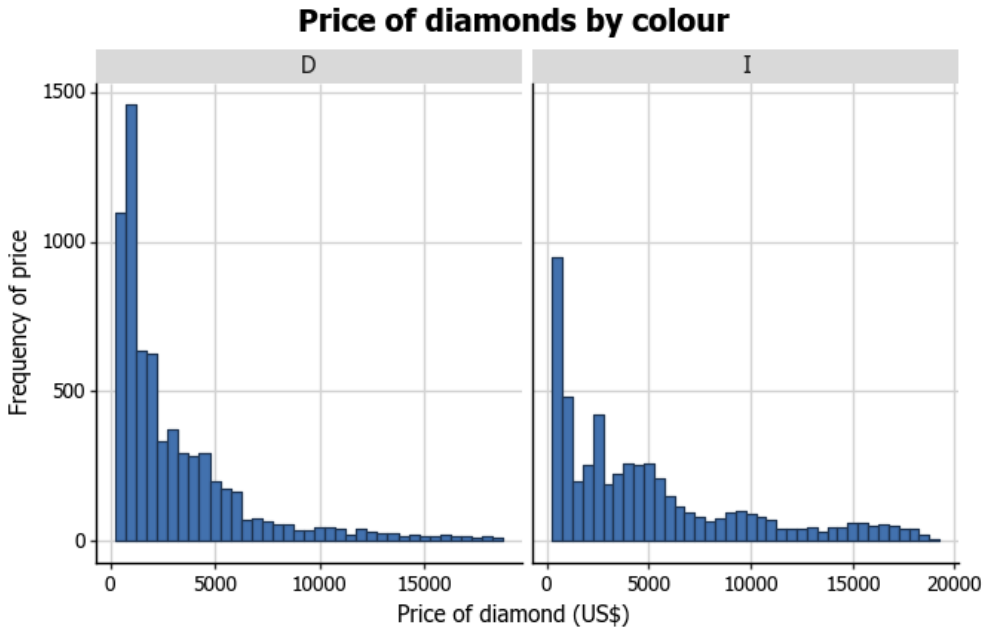
You can also easily create multiple histograms by the levels of another variable. There are two options, in separate (panel) plots, or in the same plot.

In order to create panel plots, we first need to do a little data wrangling. In order to make the graphs a bit clearer, we've only kept diamonds with either a "D" or "I" colour in a new dataset, `diamonds_trimmed`. We also need to reset the levels of `cut`, by casting it as a string and then as a category using the `astype` method.

In order to produce a panel plot by colour, we add the `facet_grid(".~color")` option to the plot. The additional `scales=free` argument in `facet_grid` means that the y-axes of each plot do not need to be the same. As you can see, we've also changed the `breaks` back to 5000 in `scale_x_continuous` as the x-axis was looking a little crowded.

```
diamonds_trimmed = diamonds[diamonds["color"].isin(["D", "I"])]
diamonds_trimmed["color"] = diamonds_trimmed["color"].\
                            astype("str").astype("category")
```

```
p7 = (
    ggplot(diamonds_trimmed, aes("price"))
    + geom_histogram(aes(y="..count.."), binwidth=500,
                     colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 5000))
    + ggtitle("Price of diamonds by colour")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
    + facet_grid(".~color", scales="free")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p7
```

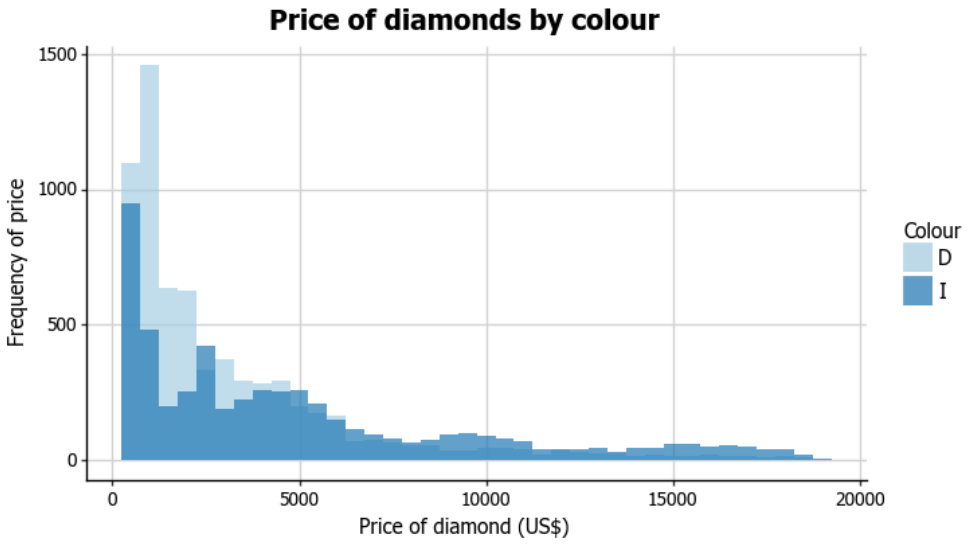


Alternatively, we can plot the two diamond colours on the same plot. Firstly, in the `ggplot` function, we add a `fill=cut` argument to `aes`. Secondly, in or-

der to more clearly see the graph, we add two arguments to the `geom_histogram` option, `position="identity"` and `alpha=0.75`. This controls the position and transparency of the curves respectively. Finally, you can customise the colours of the histograms by adding `scale_fill_brewer` to the plot. This[9] blog post describes the available packages. This article is for use in R, but the options are the same in `plotnine` as the original `ggplot2`, with the difference that you need to specify which group the colour scheme belongs to using the `type` argument: qualitative (`qual`), sequential (`seq`) and diverging (`div`).

```
p7 = (
    ggplot(diamonds_trimmed, aes("price", fill="color"))
    + geom_histogram(aes(y="..count.."), binwidth=500,
                     position="identity", alpha=0.70)
    + scale_x_continuous(breaks=np.arange(0, 22500, 5000))
    + ggtitle("Price of diamonds by colour")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
    + scale_fill_brewer(type="qual", palette="Paired",
                        name="Colour")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p7
```

[9] http://moderndata.plot.ly/create-colorful-graphs-in-r-with-rcolorbrewer-and-plotly/



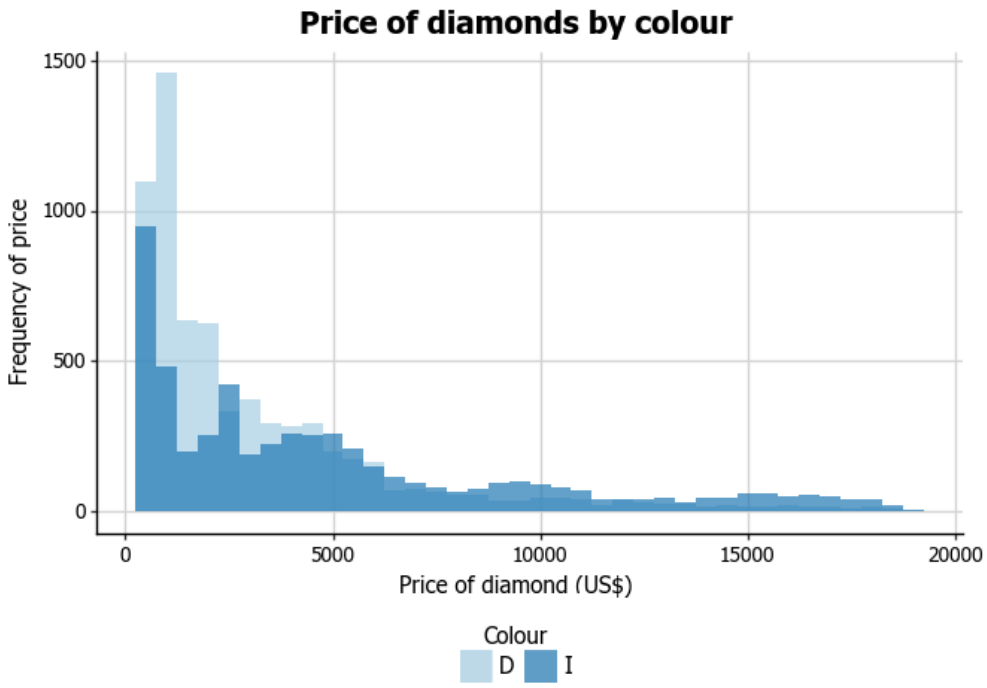
7.16 Formatting the legend

Finally, we can format the legend. Firstly, we can change the position by adding the `legend_position="bottom"` argument to the `theme` option, which moves the legend under the plot. We can change the orientation of the legend to horizontal by then adding `legend_direction="horizontal"` to `theme`. We can also centre the legend by adding `legend_title_align="center"`. We can adjust the legend position using `legend_box_spacing=0.4`. Finally, we can fix the title text by adding the `labs(colour="Diamond colour")` option to the plot.

With all of these customisations, we now have the graph we presented at the beginning of this chapter.

```
p7 = (
    ggplot(diamonds_trimmed, aes("price", fill="color"))
    + geom_histogram(aes(y="..count.."), binwidth=500,
                     position="identity", alpha=0.70)
    + scale_x_continuous(breaks=np.arange(0, 22500, 5000))
    + ggtitle("Price of diamonds by colour")
    + xlab("Price of diamond (US$)")
    + ylab("Frequency of price")
    + labs(fill="Diamond colour")
    + scale_fill_brewer(type="qual", palette="Paired",
                        name="Colour")
    + theme(
        legend_position="bottom",
```

```
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p7
```

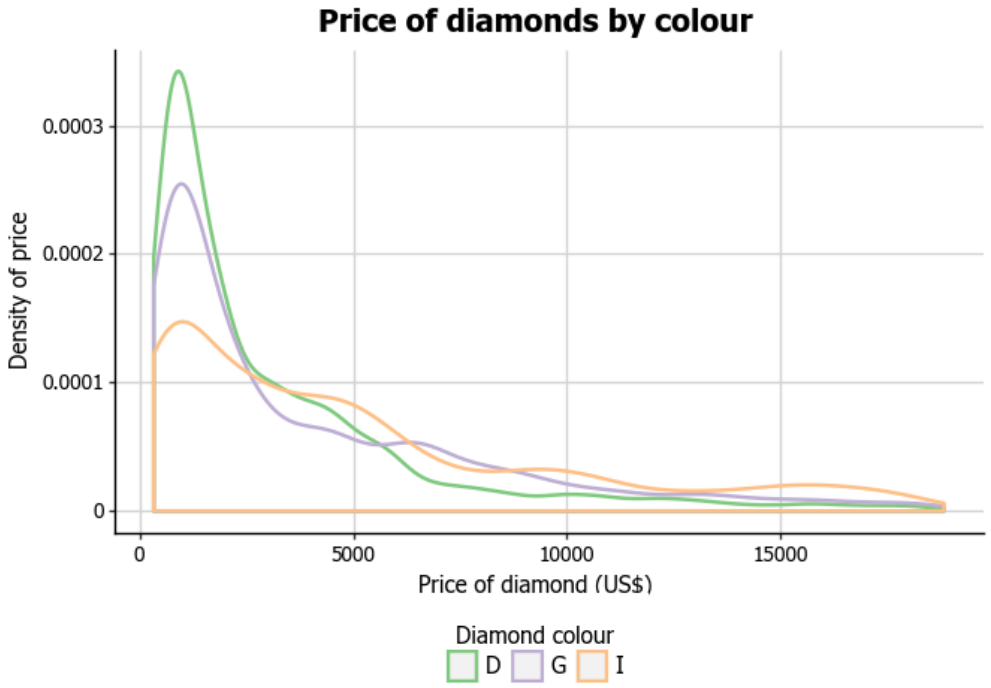


Chapter 8

Density plots

8.1 Introduction

In this chapter, we will work towards creating the density plot below. We will take you from a basic density plot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from plotnine import data
from pandas import DataFrame
```

For this example, we'll be using the diamonds dataset from plotnine's data module. The details of this dataset are here[1].

```
diamonds = data.diamonds
```

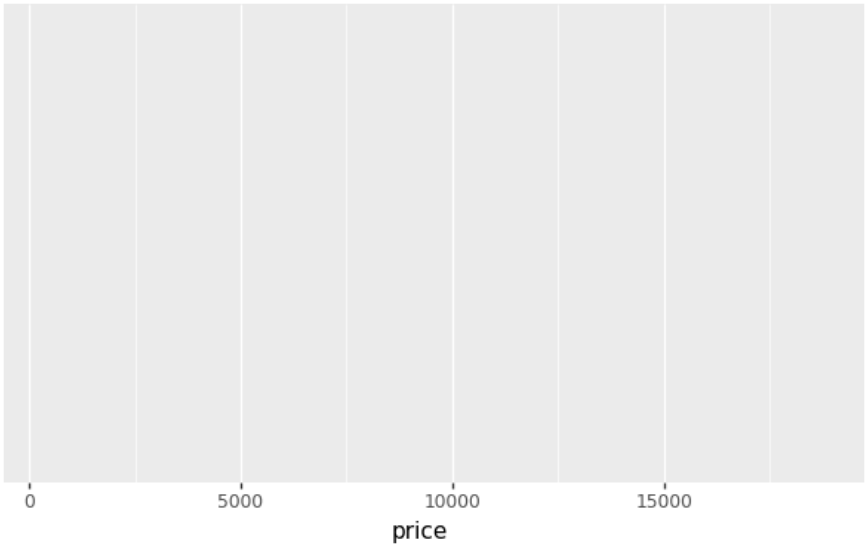
8.2 Basic ggplot structure

In order to initialise a density plot we tell ggplot that `diamonds` is our data, and specify that we want to plot the `price` variable (by argument position, this is assigned to the x-axis). You may have noticed that we put our x- and y-variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `price` to the x-axis, and for now, has left the y-axis blank.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

```
p8 = ggplot(diamonds, aes("price"))
p8
```

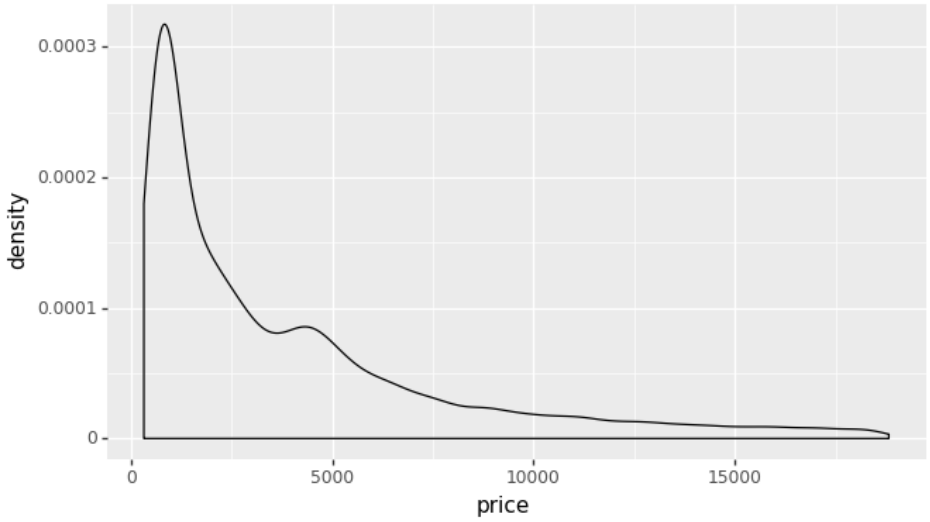
[1] `http://plotnine.readthedocs.io/en/stable/generated/plotnine.data.diamonds.html`



8.3 Basic density plot

We can do this using `geoms`. In the case of a density plot, we use the `geom_density()` geom.

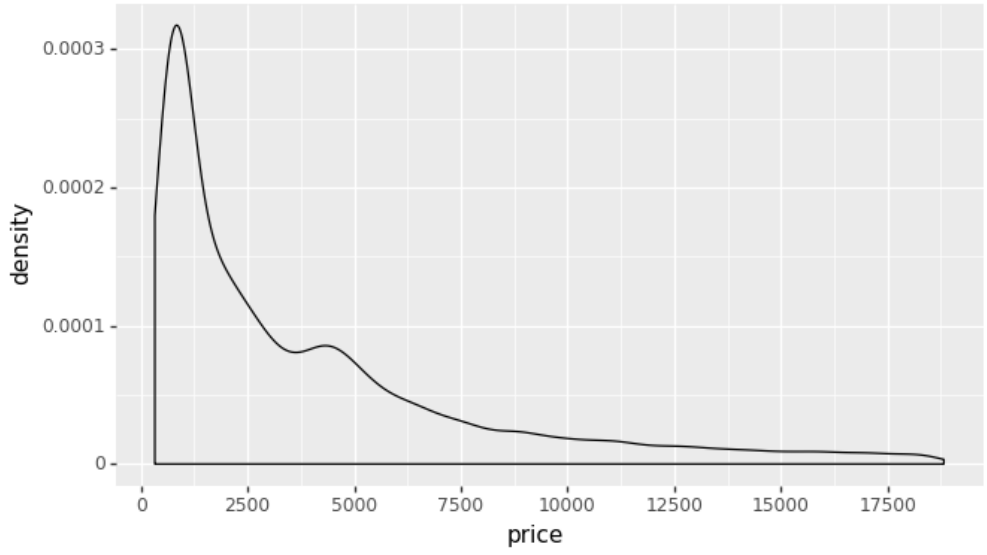
```
p8 = ggplot(diamonds, aes("price")) + geom_density()
p8
```



8.4 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, to change the y-axis we use the `scale_y_continuous` option. Here we will change the x-axis to every $2500, rather than every $5000. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[2] which generates a sequence from your selected start, stop and step values respectively. Alternatively if you just want to change the minimum and maximum values you can use the `limits` argument to define these.

```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density()
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
)
p8
```

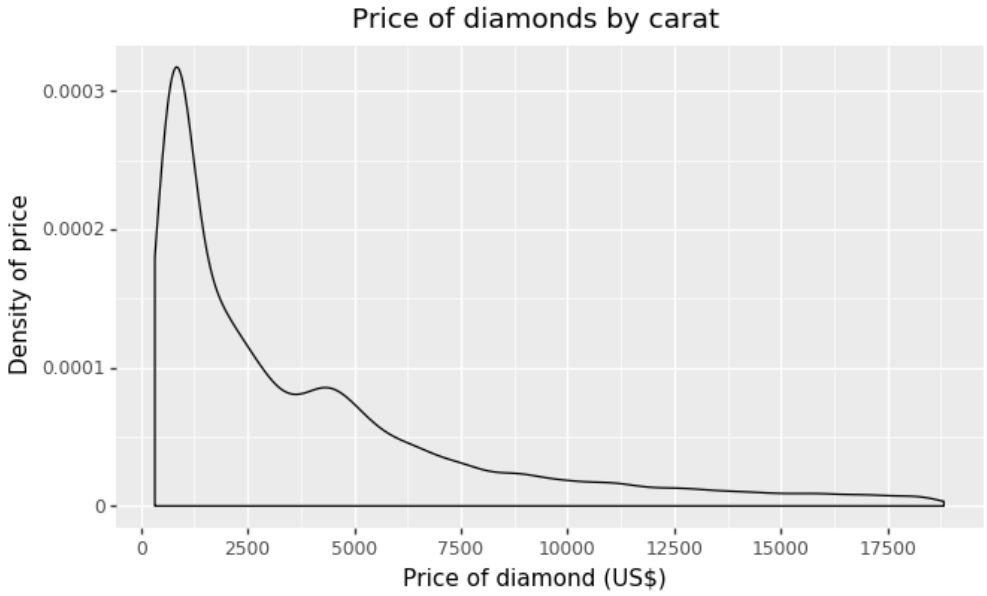


[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html

8.5 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density()
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
)
p8
```

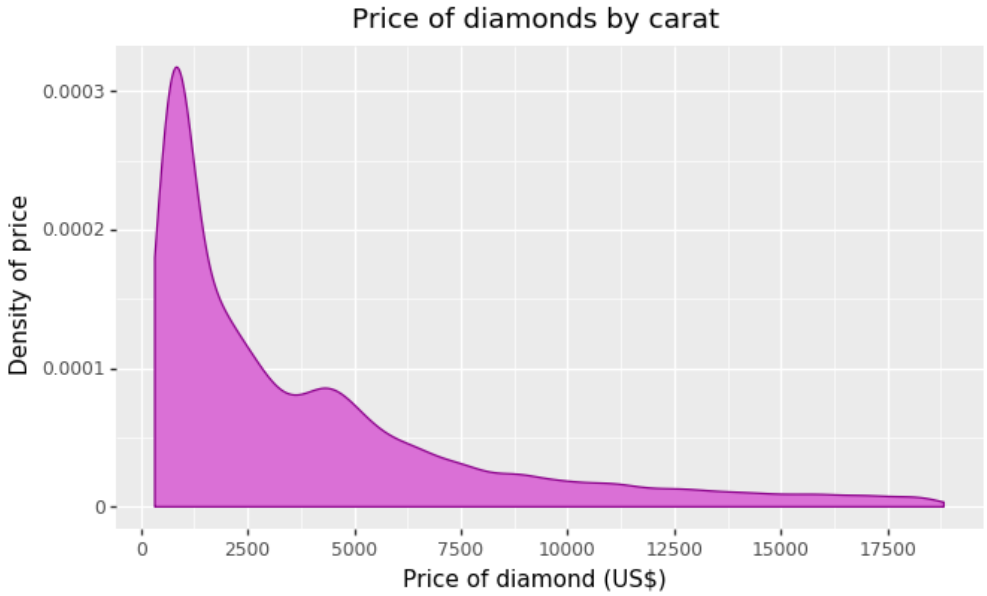


8.6 Adjusting the colour palette

There are a few options for adjusting the colour. The first is to change colours by name. `plotnine` uses the colour palette utilised by `matplotlib`, and the full

set of named colours recognised by `ggplot` is here[3]. Let's try changing our bars to two shades of purple. To do so, we need to add both the `fill` and `colour` options to `geom_density`. Of course, to make the bars a single colour, you would just assign the same colour name to both `fill` and `colour`.

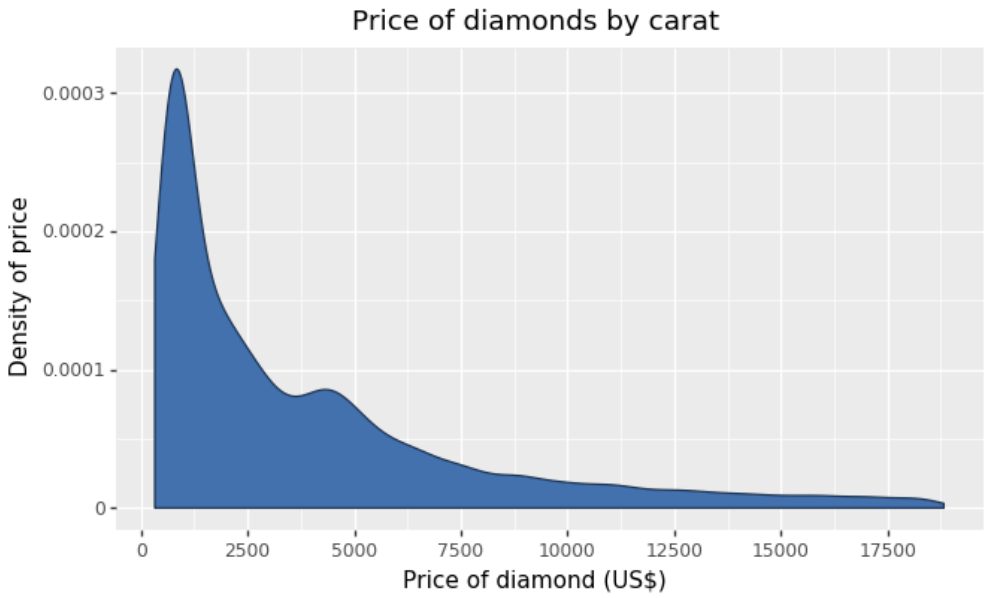
```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density(colour="darkmagenta", fill="orchid")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
)
p8
```



If you want to go beyond the options in the list above, you can also specify exact HEX colours by including them as a string preceded by a hash, e.g., `"#FFFFFF"`. Below, we have called two shades of blue for the fill and lines using their HEX codes.

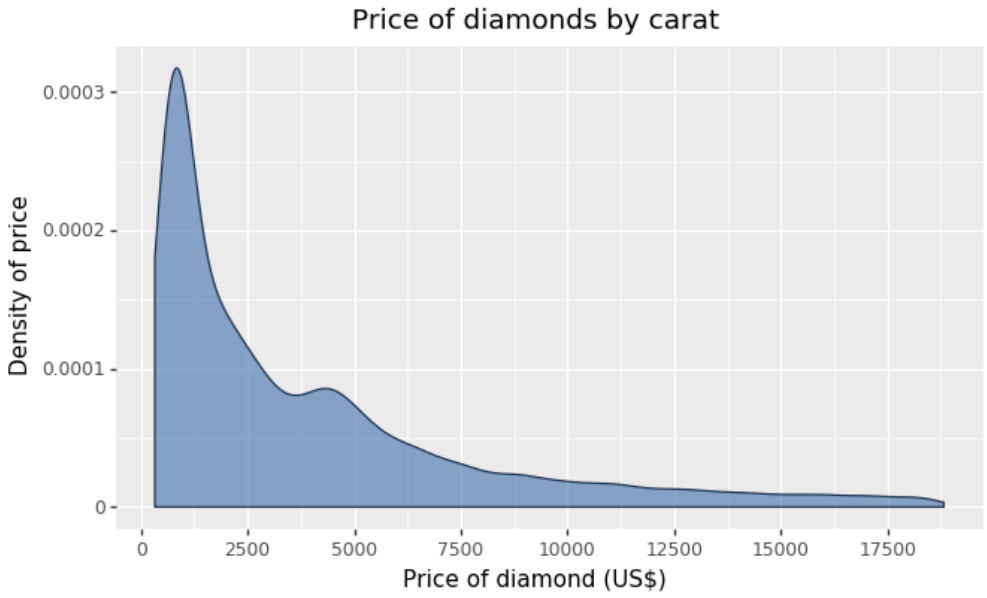
[3] `https://matplotlib.org/examples/color/named_colors.html`

```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density(colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
)
p8
```



You can also specify the degree of transparency in the density fill area using the argument `alpha` in `geom_density`. This ranges from 0 to 1.

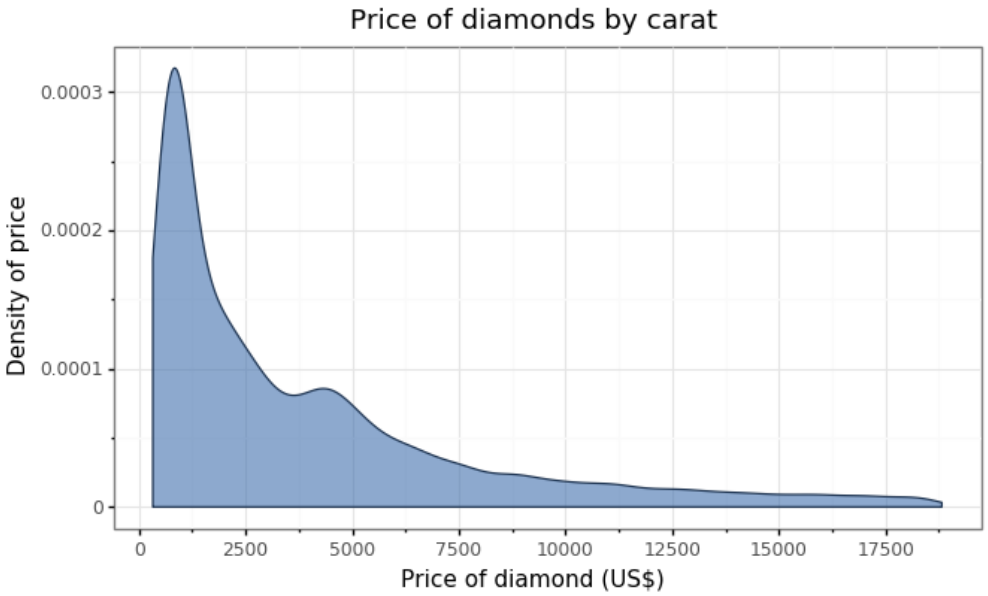
```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density(colour="#1F3552", fill="#4271AE",
                   alpha=0.6)
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
)
p8
```



8.7 Using the white theme

We can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after our current code.

```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density(colour="#1F3552", fill="#4271AE",
                   alpha=0.6)
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + theme_bw()
)
p8
```



8.8 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[4]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

[4] `xkcd.com/1350/xkcd-Regular.otf`

We can then call methods on these objects (the list of available methods is here[5]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight()` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

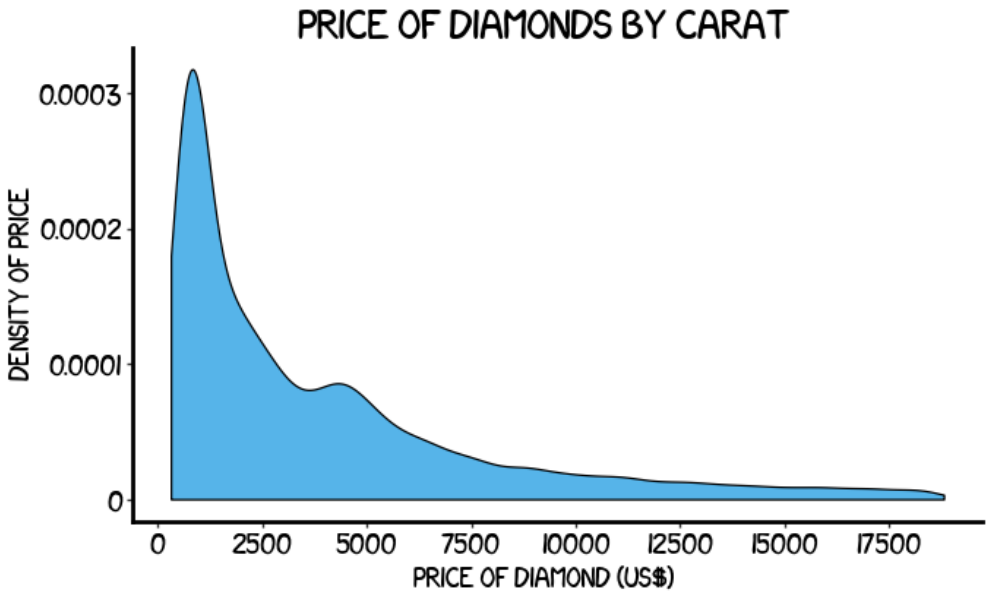
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density(colour="black", fill="#56B4E9")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + theme(
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
```

[5] https://matplotlib.org/api/font_manager_api.html

```
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p8
```



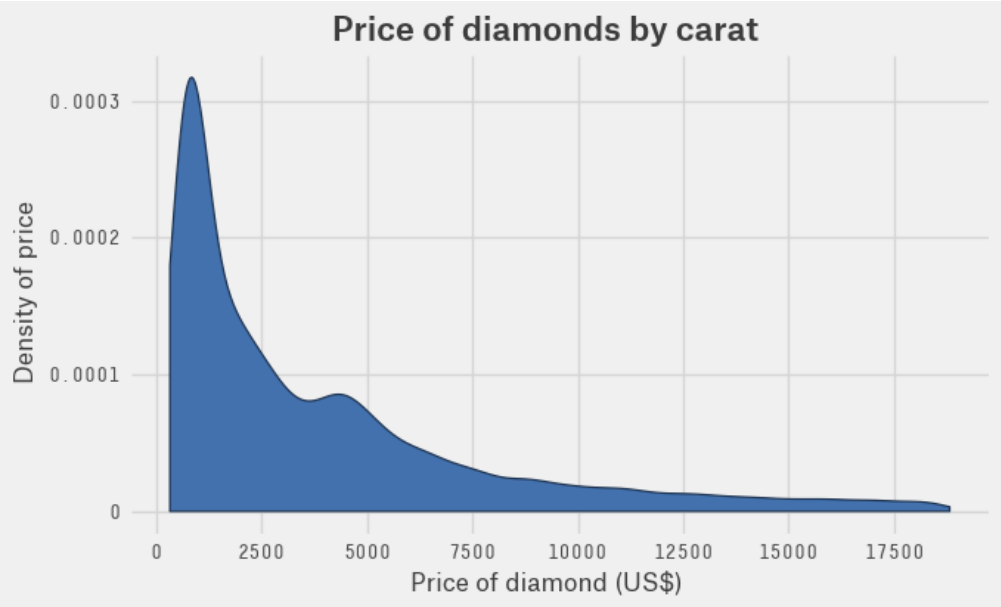
8.9 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[6]). Below we've applied `theme_538()`, which approximates graphs on the website FiveThirtyEight. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[7] and 'Decima Mono Pro'[8] in `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
```

[6] http://plotnine.readthedocs.io/en/stable/api.html#themes
[7] https://commercialtype.com/catalog/atlas/atlas_grotesk
[8] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
axis_text.set_size(12)
body_text.set_size(10)


p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density(colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds by carat")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p8
```

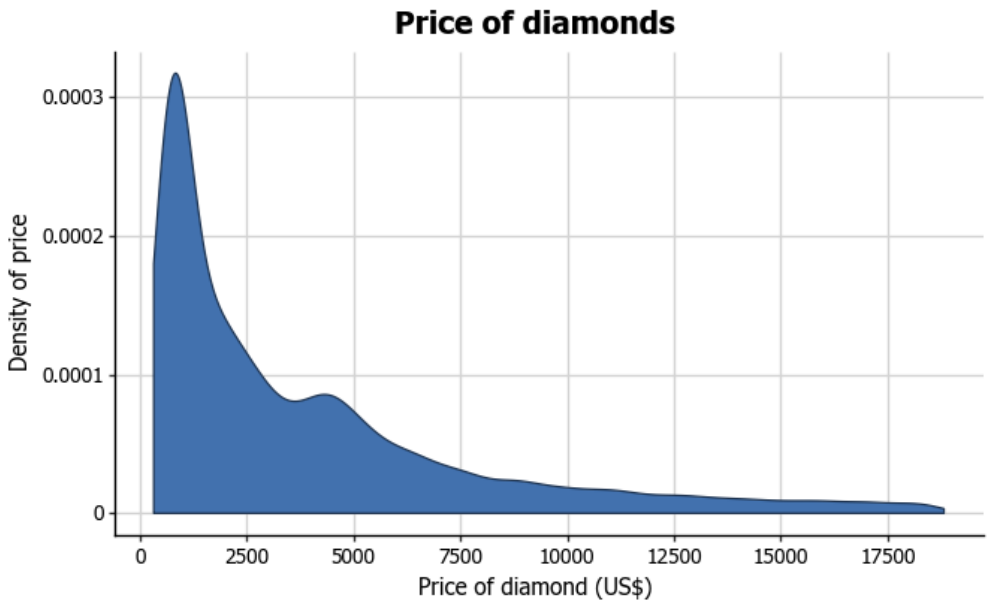


8.10 Creating your own theme

Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density(colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p8
```

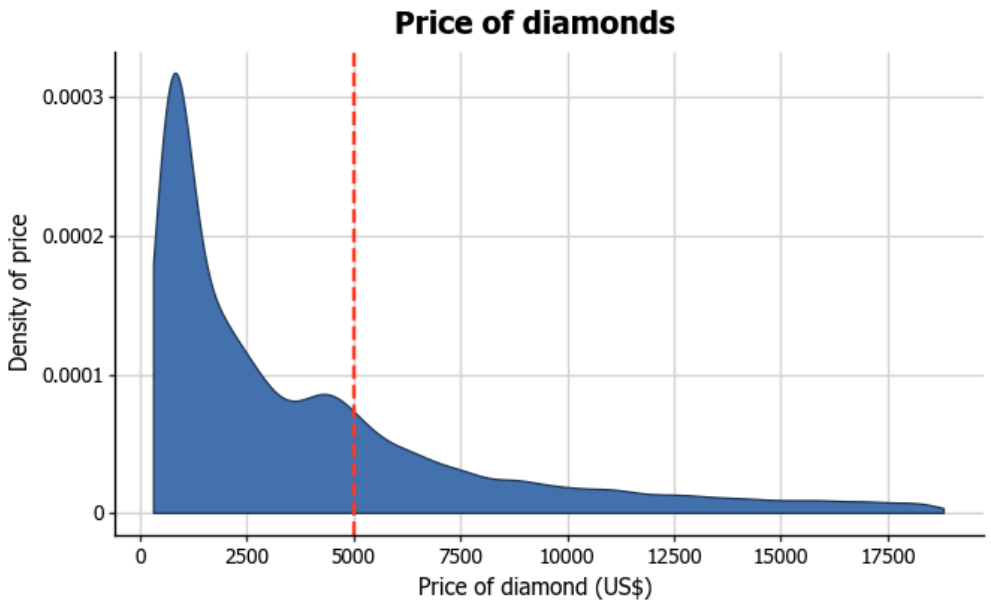


8.11 Adding lines

Let's say that we want to add a cutoff value to the chart (at $5000). We add the `geom_vline` geom to the chart, and specify where it goes on the x-axis using the `xintercept` argument. We can customise how it looks using the `colour` and `linetype` arguments in `geom_vline`. (In the the same way, horizontal lines can be added using the geom `geom_hline`.)

```
p8 = (
    ggplot(diamonds, aes("price"))
    + geom_density(colour="#1F3552", fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(0, 22500, 2500))
    + ggtitle("Price of diamonds")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + geom_vline(xintercept=5000, size=1, colour="#FF3721",
                 linetype="dashed")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
```

```
                                face="bold"),
            text=element_text(family="Tahoma", size=11),
            axis_text_x=element_text(colour="black", size=10),
            axis_text_y=element_text(colour="black", size=10),
        )
)
p8
```



8.12 Multiple density plots

You can also easily create multiple density plots by the levels of another variable. There are two options, in separate (panel) plots, or in the same plot.

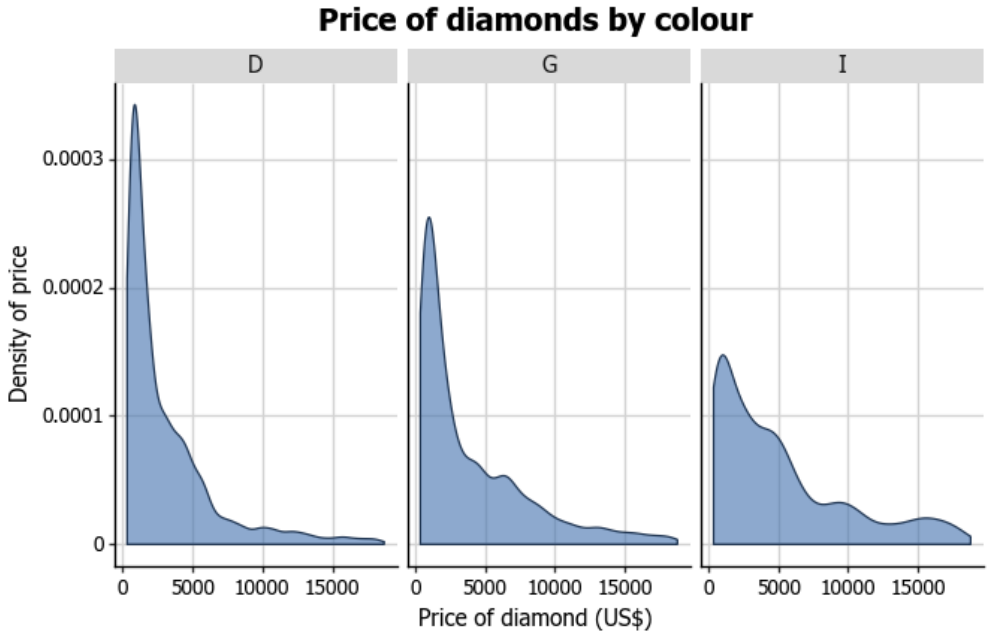
In order to create panel plots, we first need to do a little data wrangling. In order to make the graphs a bit clearer, we've only kept diamonds with either a "D", "G" or "I" colour in a new dataset, `diamonds_trimmed`. We also need to reset the levels of `colour`, by casting it as a string and then a category using the `astype` method.

In order to produce a panel plot by colour, we add the `facet_grid(".~color")` option to the plot. The additional `scales = free` argument in `facet_grid` means that the y-axes of each plot do not need to be the same. As you can see, we've

also changed the `breaks` back to 5000 in `scale_x_continuous` as the x-axis was looking a little crowded.

```
diamonds_trimmed = diamonds[diamonds["color"].isin(["D", "G", "I"])]
diamonds_trimmed["color"] = diamonds_trimmed["color"].\
                            astype("str").astype("category")
```

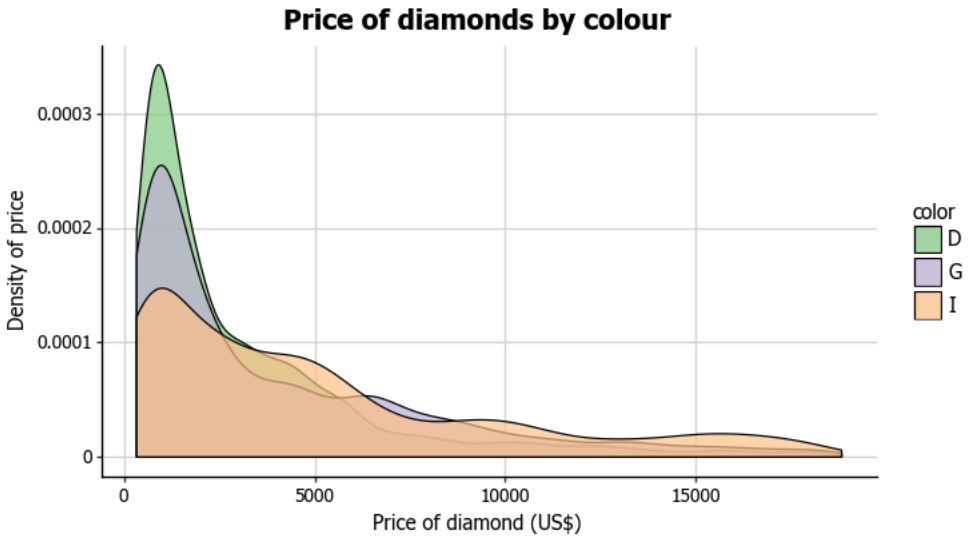
```
p8 = (
    ggplot(diamonds_trimmed, aes("price"))
    + geom_density(colour="#1F3552", fill="#4271AE", alpha=0.6)
    + scale_x_continuous(breaks=np.arange(0, 22500, 5000))
    + ggtitle("Price of diamonds by colour")
    + xlab("Price of diamond (US$)") + ylab("Density of price")
    + facet_grid(".~color", scales="free")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10)))
p8
```



Alternatively, we can plot the three diamond colours on the same plot. Firstly, in the `ggplot` function, we add a `fill=cut` argument to `aes`. Secondly, in order to more clearly see the graph, we add two arguments to the `geom_density()` geom, `position="identity"` and `alpha=0.70`. This controls the position and transparency of the curves respectively. Finally, you can customise the colours of the density plots by adding `scale_fill_brewer` to the plot. This[9] blog post describes the available packages. This article is for use in R, but the options are the same in `plotnine` as the original `ggplot2`, with the difference that you need to specify which group the colour scheme belongs to using the `type` argument: qualitative (`qual`), sequential (`seq`) and diverging (`div`).

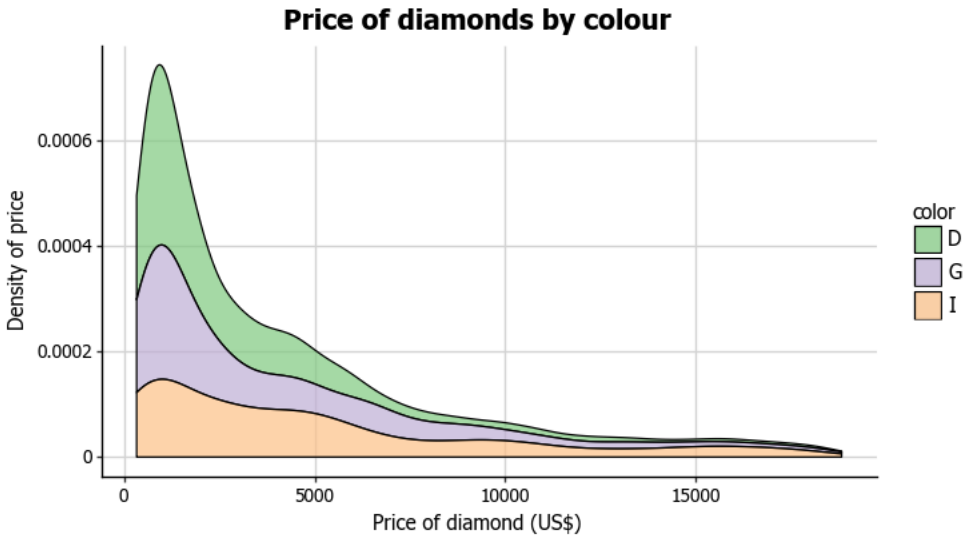
```
p8 = (
    ggplot(diamonds_trimmed, aes("price", fill="color"))
    + geom_density(position="identity", alpha=0.70)
    + scale_x_continuous(breaks=np.arange(0, 22500, 5000))
    + ggtitle("Price of diamonds by colour")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p8
```

[9] http://moderndata.plot.ly/create-colorful-graphs-in-r-with-rcolorbrewer-and-plotly/



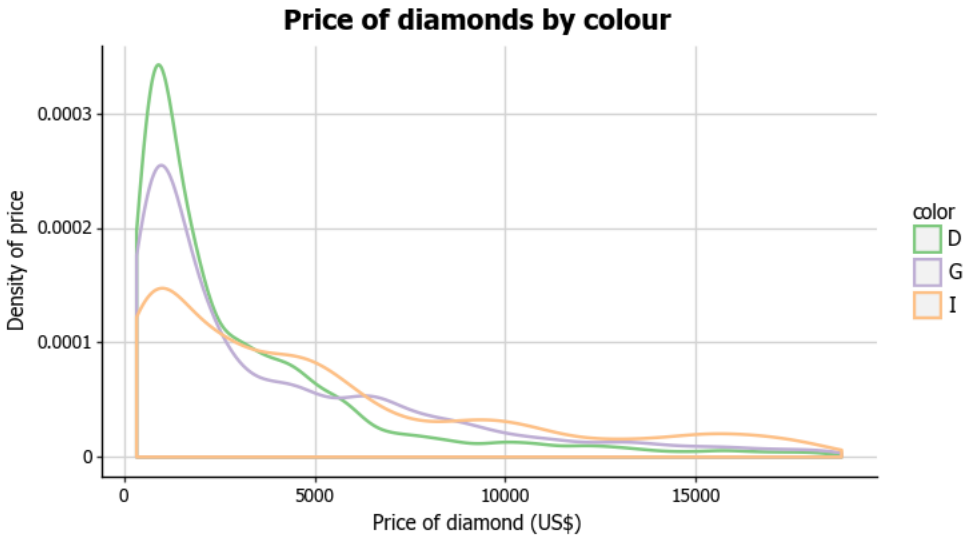
These densities are a little hard to see. One way we can make it easier to see them is to stack the densities on top of each other. To do so, we swap `position="stack"` for `position="identity"` in `geom_density`.

```
p8 = (
    ggplot(diamonds_trimmed, aes("price", fill="color"))
    + geom_density(position="stack", alpha=0.70)
    + scale_x_continuous(breaks=np.arange(0, 22500, 5000))
    + ggtitle("Price of diamonds by colour")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + scale_fill_brewer(type="qual", palette="Accent")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p8
```



Another way to make it a little easier to see the densities is by dropping out the fill. To do this need a few changes. We need to swap the option `fill=color` in `ggplot` for `colour=color`. We add the `fill=None` to `geom_density()`, and we've also added `size=1` to make it easier to see the lines. We need to change `position` back to `identity` in `geom_density()`. Finally, we change the `scale_fill_brewer()` option for `scale_colour_brewer()`.

```
p8 = (
    ggplot(diamonds_trimmed, aes("price", colour="color"))
    + geom_density(position="identity", fill=None, size=1)
    + scale_x_continuous(breaks=np.arange(0, 22500, 5000))
    + ggtitle("Price of diamonds by colour")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + scale_colour_brewer(type="qual", palette="Accent")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p8
```



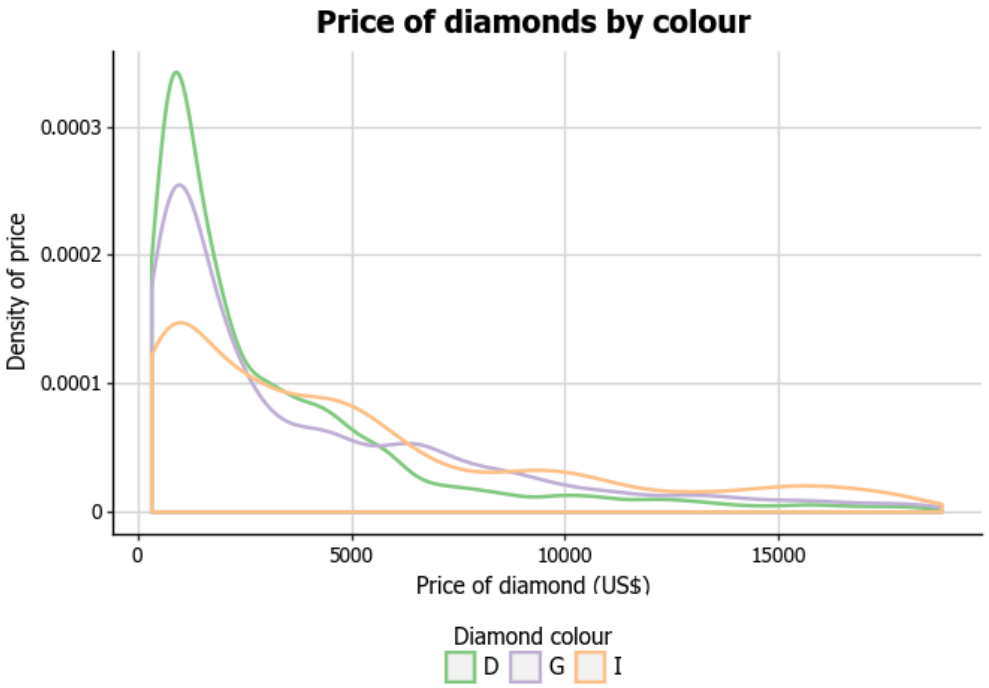
8.13 Formatting the legend

Finally, we can format the legend. Firstly, we can change the position by adding the `legend_position="bottom"` argument to the `theme` option, which moves the legend under the plot. We can change the orientation of the legend to horizontal by then adding `legend_direction="horizontal"` to `theme`. We can also centre the legend by adding `legend_title_align="center"`. We can adjust the legend position using `legend_box_spacing=0.4`. Finally, we can fix the title text by adding the `labs(colour="Diamond colour")` option to the plot.

With all of these customisations, we now have the graph we presented at the beginning of this chapter.

```
p8 = (
    ggplot(diamonds_trimmed, aes("price", colour="color"))
    + geom_density(position="identity", fill=None, size=1)
    + scale_x_continuous(breaks=np.arange(0, 22500, 5000))
    + ggtitle("Price of diamonds by colour")
    + xlab("Price of diamond (US$)")
    + ylab("Density of price")
    + labs(colour="Diamond colour")
    + scale_colour_brewer(type="qual", palette="Accent")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
```

```
        legend_box_spacing=0.4,
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p8
```

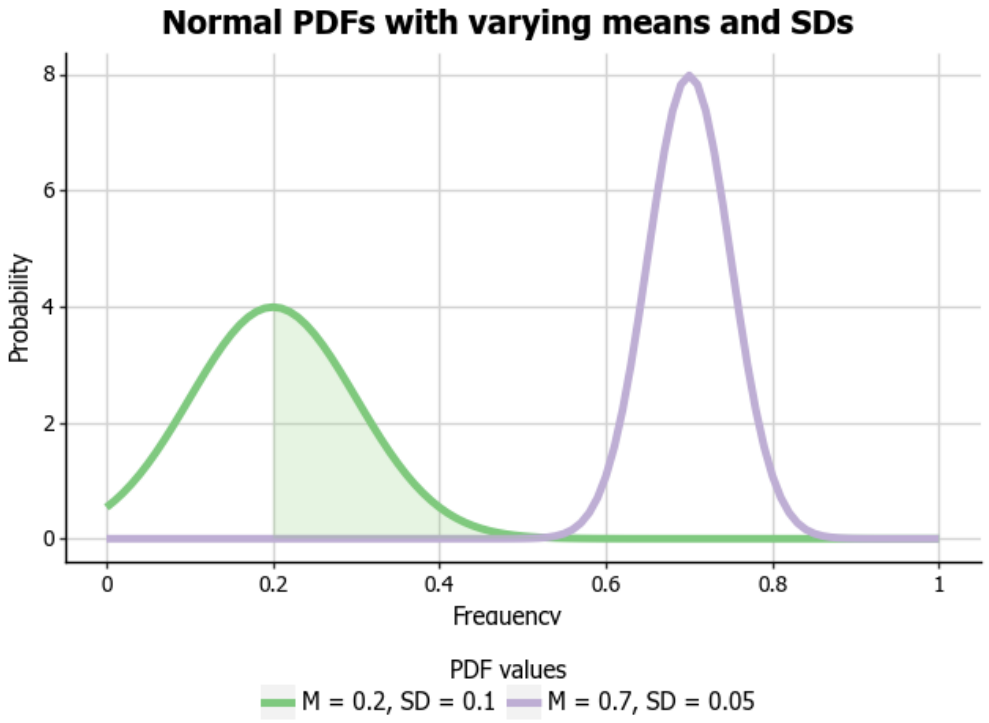


Chapter 9

Function plots

9.1 Introduction

In this chapter, we will work towards creating the function plot below. We will take you from a basic function plot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs;
- `numpy` to do some basic numeric calculations in our graphing; and
- `scipy.stats` to calculate our probability functions.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

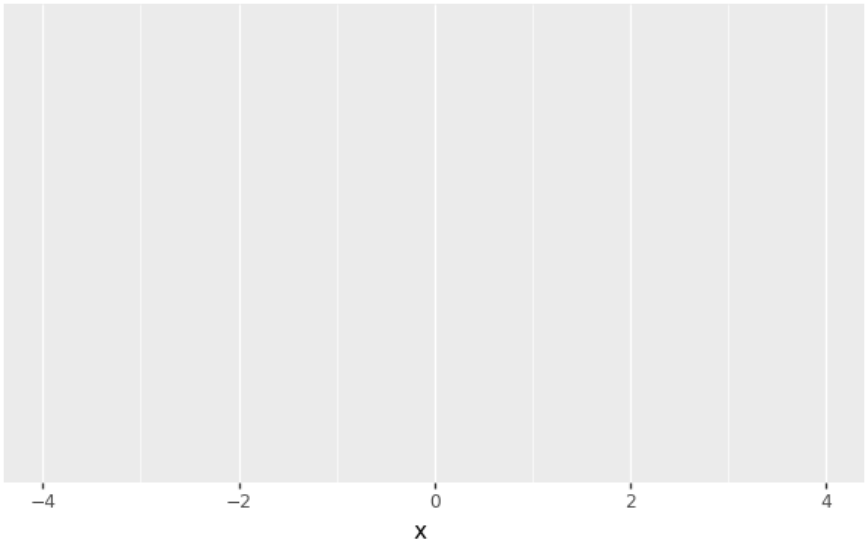
```
import pandas as pd
import numpy as np
import scipy.stats as st

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from pandas import DataFrame
```

9.2 Basic ggplot structure

In order to initialise our function plot we pass a DataFrame to `ggplot` with the x-coordinates of -4 to 4. You may have noticed that we put our x-variable inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell `ggplot` how we want to visually represent it.

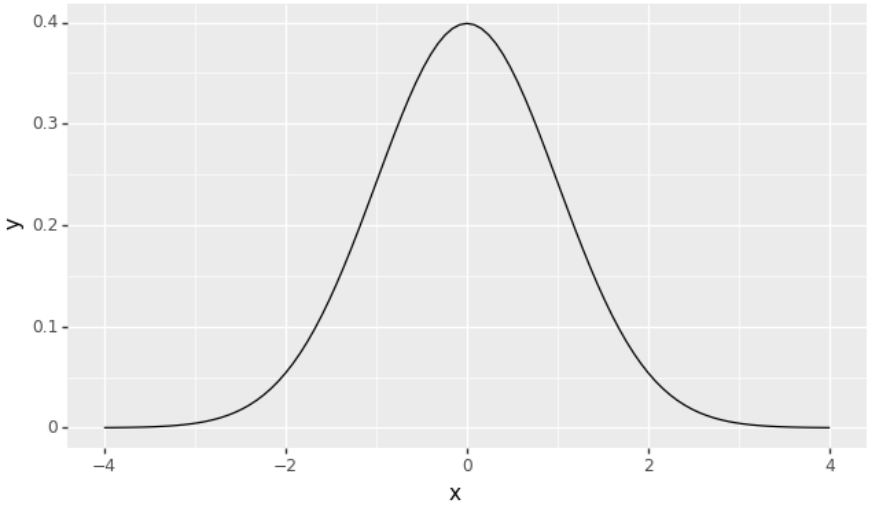
```
p9 = ggplot(DataFrame({"x": [-4, 4]}), aes("x"))
p9
```



9.3 Plotting a normal curve

In order to create a normal curve, we add the `stat_function()` option and add `st.norm.pdf` to the function argument to make it a normal curve.

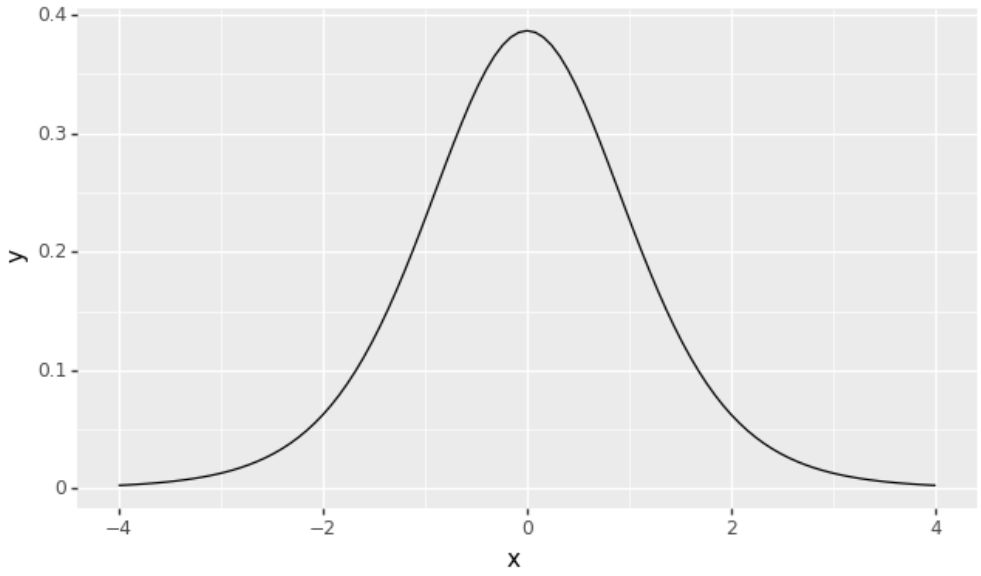
```
p9 = (ggplot(DataFrame({"x": [-4, 4]}), aes("x"))
      + stat_function(fun=st.norm.pdf))
p9
```



9.4 Plotting a t-curve

`stat_function` can draw a range of continuous probability density functions[1], including t (`t`), F (`f`) and Chi-square (`chi2`) PDFs. Here we will plot a t-distribution. As the shape of the t-distribution changes depending on the sample size (indicated by the degrees of freedom, or df), we need to specify our df value as part of defining our curve, using the `args` argument and passing the degrees of freedom in a dictionary.

```
p9 = (
    ggplot(DataFrame({"x": [-4, 4]}), aes("x"))
    + stat_function(fun=st.t.pdf, args=dict(df=8))
)
p9
```

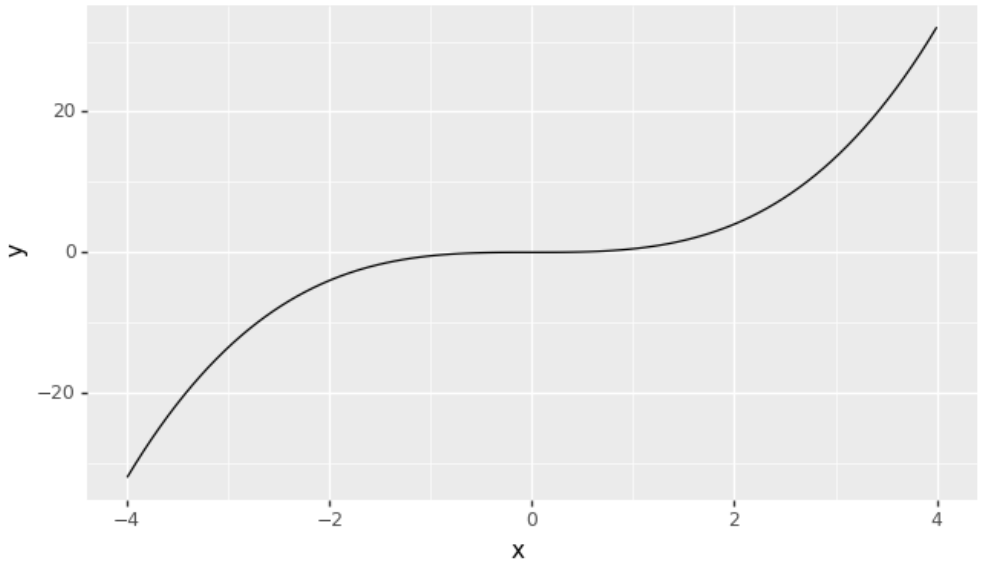


9.5 Plotting your own function

You can also draw your own function, as long as it takes the form of a formula that converts an x-value into a y-value. Here we have plotted a curve that returns y-values that are the cube of x times a half:

[1] https://en.wikipedia.org/wiki/Probability_density_function

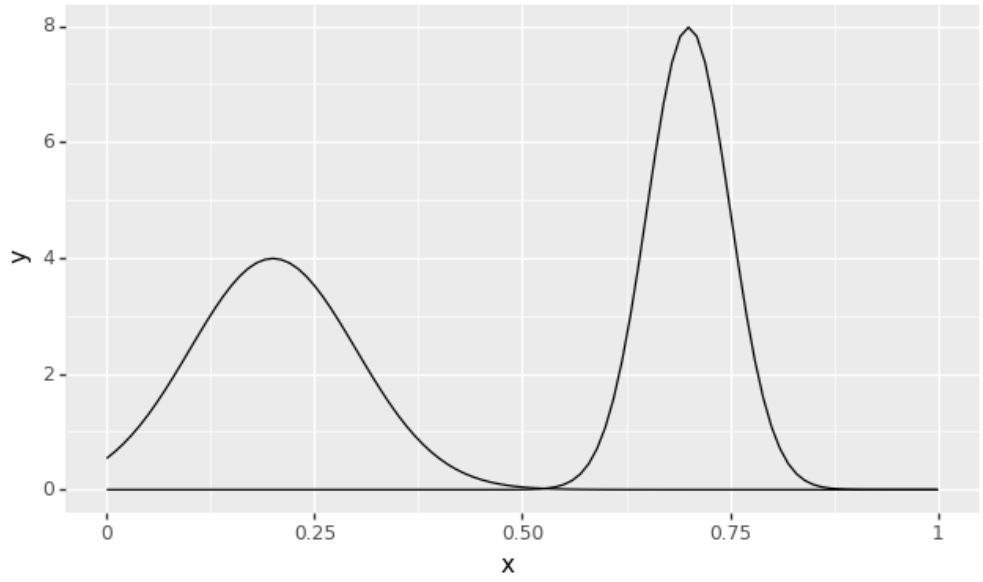
```
def cubeFunc(x):
    return x ** 3 * 0.5


p9 = (
    ggplot(DataFrame({"x": [-4, 4]}), aes("x"))
      + stat_function(fun=cubeFunc)
)
p9
```



9.6 Plotting multiple functions on the same graph

You can plot multiple functions on the same graph by simply adding another `stat_function()` for each curve. Here we have plotted two normal curves on the same graph, one with a mean of 0.2 and a standard deviation of 0.1, and one with a mean of 0.7 and a standard deviation of 0.05. (Note that the `st.norm.pdf` function has a default mean of 0 and a default standard deviation of 1, which is why we didn't need to explicitly define them in the first normal curve we plotted above.) You can also see we've changed the range of the x-axis to between 0 and 1.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(fun=st.norm.pdf,
                    args=dict(loc=0.2, scale=0.1))
    + stat_function(fun=st.norm.pdf,
                    args=dict(loc=0.7, scale=0.05))
)
p9
```

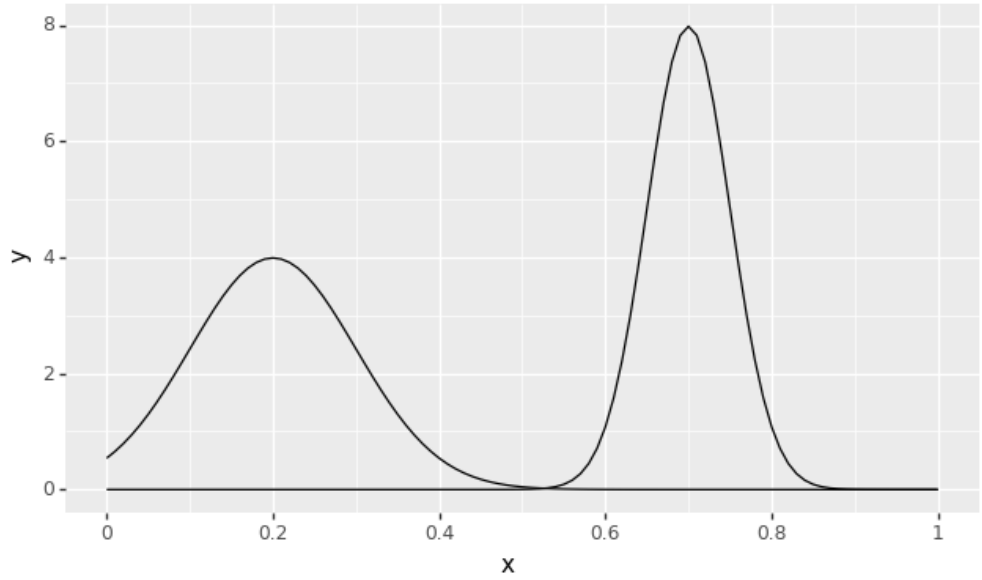


9.7 Adjusting the axis scales

The next thing we will change is the axis ticks. Let's make the x-axis ticks appear at every 0.2 units rather than 0.25. We can do this by adding the option `scale_x_continuous` and telling it where you want the tick marks to be. Instead of typing in the whole list manually, you can use numpy's `arange` function[2] which generates a sequence from your selected start, stop and step values respectively. (Note that because Python's way of doing indexing, we need to set the stop value to just beyond our desired end value.) We also can ensure that the x-axis begins and ends where we want by also adding the argument `limits = [0, 1]` to `scale_x_continuous`.

[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(fun=st.norm.pdf,
                    args=dict(loc=0.2, scale=0.1))
    + stat_function(fun=st.norm.pdf,
                    args=dict(loc=0.7, scale=0.05))
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
)
p9
```

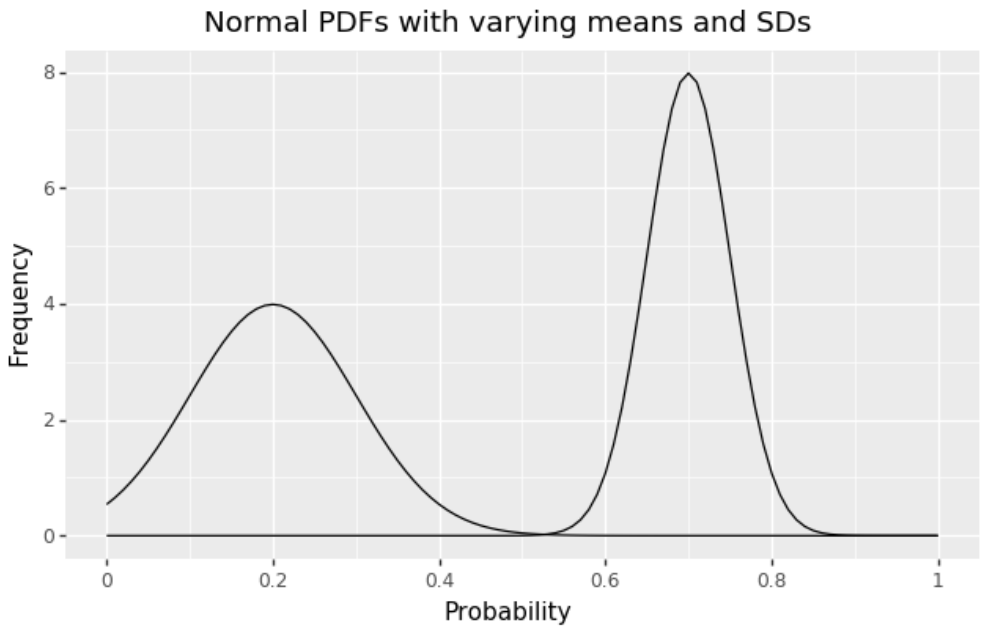


9.8 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `lab` option and the `x` and `y` options respectively.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(fun=st.norm.pdf,
                    args=dict(loc=0.2, scale=0.1))
    + stat_function(fun=st.norm.pdf,
                    args=dict(loc=0.7, scale=0.05))
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
```

```
                          limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
)
p9
```



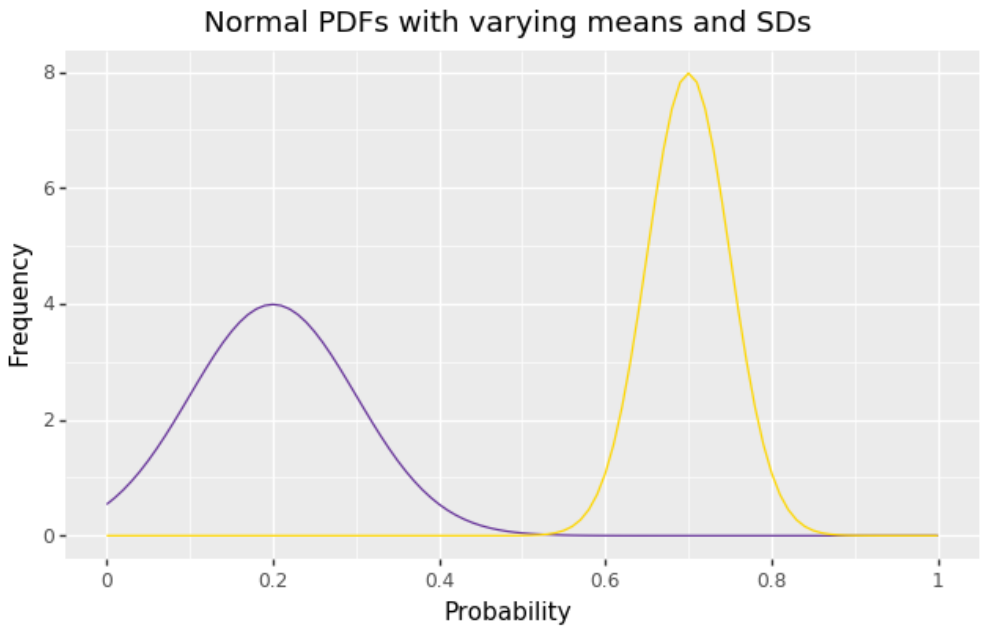
9.9 Changing the colour of the function lines

There are a couple of different ways to change the colours of the function lines. The first is using the colour names. `plotnine` uses the colour palette utilised by `matplotlib`, and the full set of named colours recognised by `ggplot` is here[3]. Let's try changing the line colour of one function to `rebeccapurple` and the other to `gold` by adding these names to the `colour` arguments in `stat_function()`.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        fun=st.norm.pdf, args=dict(loc=0.2, scale=0.1),
        colour="rebeccapurple"
    )
```

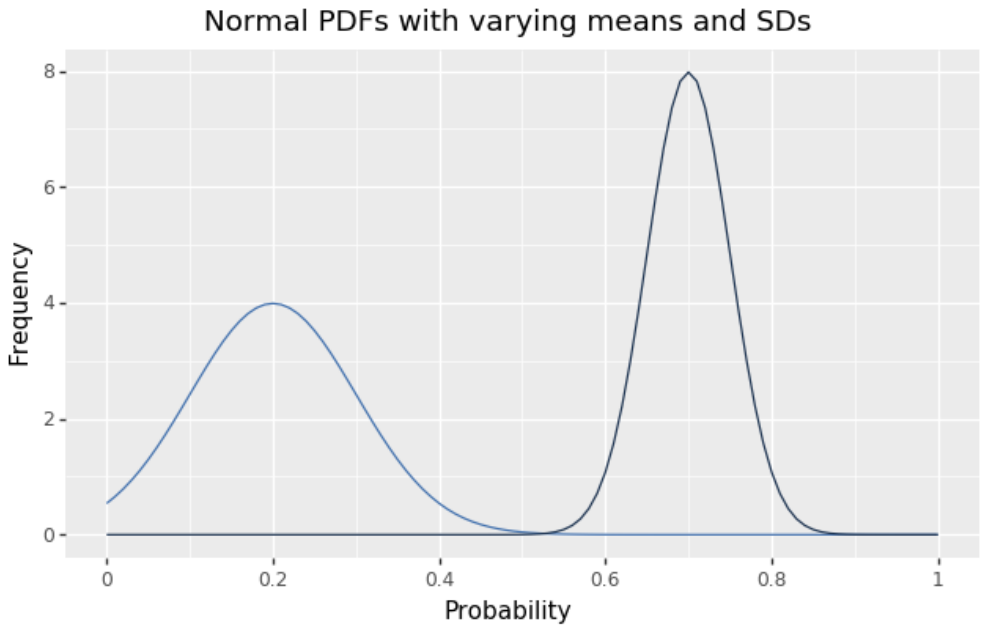
[3] https://matplotlib.org/examples/color/named_colors.html

```
    + stat_function(fun=st.norm.pdf,
                    args=dict(loc=0.7, scale=0.05),
                    colour="gold")
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
)
p9
```



If you want to go beyond the options in the list above, you can also specify exact HEX colours by including them as a string preceded by a hash, e.g., `"#FFFFFF"`. Below, we have called two shades of blue for the lines using their HEX codes.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(fun=st.norm.pdf, args=dict(loc=0.2, scale=0.1),
                    colour="#4271AE")
    + stat_function(fun=st.norm.pdf, args=dict(loc=0.7, scale=0.05),
                    colour="#1F3552")
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2), limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
)
p9
```

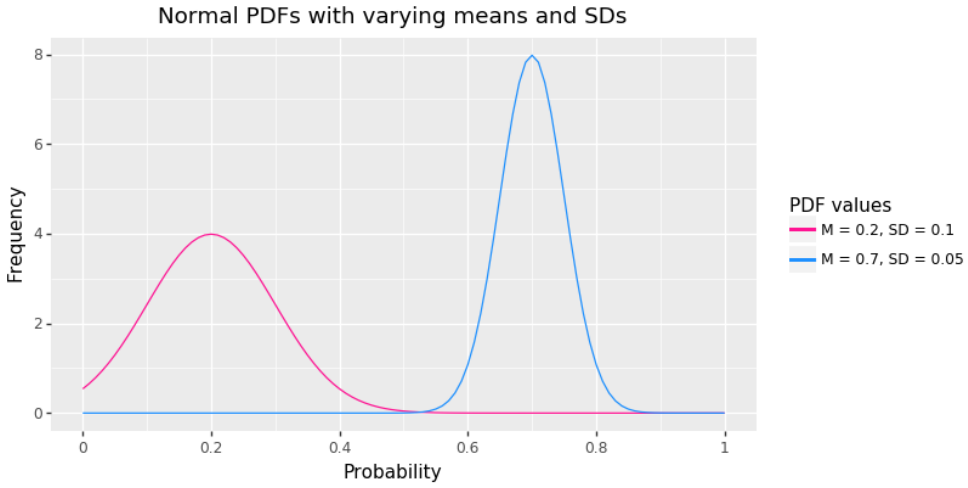


9.10 Adding a legend

As we have added two separate commands to plot the two function curves, `ggplot` does not automatically recognise that it needs to create a legend. We can make a legend by swapping out the `colour` argument in each of the `stat_function` commands for `aes(colour=)`, and assigning it the name of the group we want to appear in the legend. (Note that you need to put the name you pass to `aes(colour=)` in quotation marks, otherwise `plotnine` will try to interpret it as a `pandas` column name.) We also need to add the `scale_colour_manual` command to make the legend appear, and also assign colours and a title.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        aes(colour='"M = 0.2, SD = 0.1"'),
        fun=st.norm.pdf,
        args=dict(loc=0.2, scale=0.1)
    )
    + stat_function(
        aes(colour='"M = 0.7, SD = 0.05"'),
        fun=st.norm.pdf,
        args=dict(loc=0.7, scale=0.05),
    )
```

```
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
    + scale_colour_manual(name="PDF values",
                          values=["deeppink", "dodgerblue"])
)
p9
```

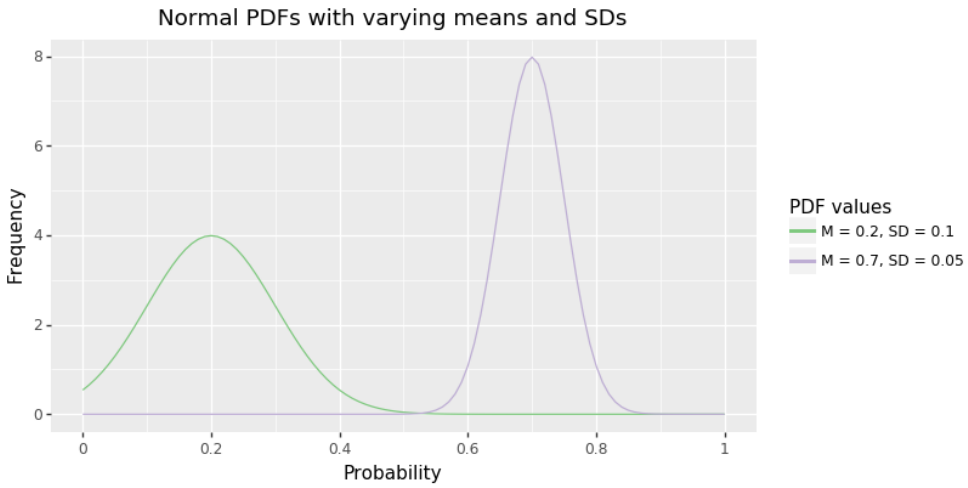


If you want to use one of the automatic brewer palettes, you can swap `scale_colour_manual` for `scale_colour_brewer`, and call your favourite Brewer colour scheme. More information on using `scale_colour_brewer` is here[4]. You can also assign the title to the legend by adding the argument `name="PDF values"` to this method.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        aes(colour='"M = 0.2, SD = 0.1"'),
        fun=st.norm.pdf,
        args=dict(loc=0.2, scale=0.1)
    )
    + stat_function(
        aes(colour='"M = 0.7, SD = 0.05"'),
        fun=st.norm.pdf,
        args=dict(loc=0.7, scale=0.05),
    )
```

[4] `http://plotnine.readthedocs.io/en/stable/generated/plotnine.scales.scale_color_brewer.html#plotnine.scales.scale_color_brewer`

```
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
    + scale_colour_brewer(type="qual", palette="Accent",
                          name="PDF values")
)
p9
```

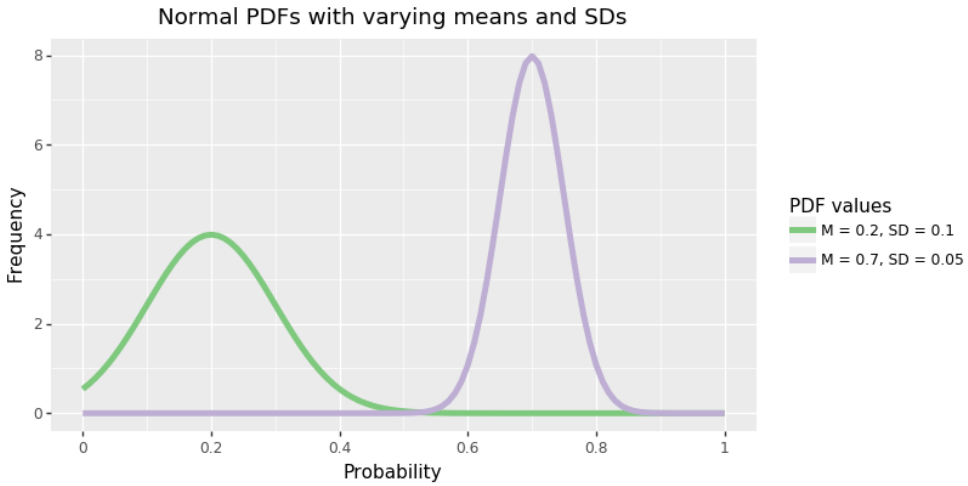


9.11 Changing the size of the lines

As you can see, the lines are a little difficult to see. You can make them thicker (or thinner) using the argument `size` argument within `stat_function()`. Here we have changed the thickness of each line to size 2.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        aes(colour='"M = 0.2, SD = 0.1"'),
        fun=st.norm.pdf,
        args=dict(loc=0.2, scale=0.1),
        size=2,
    )
    + stat_function(
        aes(colour='"M = 0.7, SD = 0.05"'),
        fun=st.norm.pdf,
        args=dict(loc=0.7, scale=0.05),
```

```
        size=2,
    )
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
    + scale_colour_brewer(type="qual", palette="Accent",
                          name="PDF values")
)
p9
```

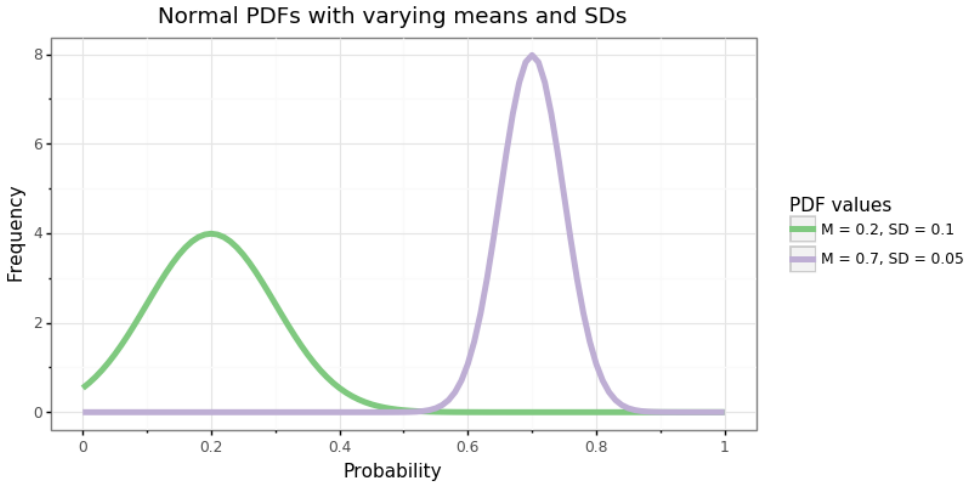


9.12 Using the white theme

As explained in the previous chapters, we can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after `ggplot()`. As you can see, we can further tweak the graph using the `theme` option, which we've used so far to change the legend.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        aes(colour='"M = 0.2, SD = 0.1"'),
        fun=st.norm.pdf,
        args=dict(loc=0.2, scale=0.1),
        size=2,
    )
    + stat_function(
```

```
        aes(colour='"M = 0.7, SD = 0.05"'),
        fun=st.norm.pdf,
        args=dict(loc=0.7, scale=0.05),
        size=2,
    )
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
    + scale_colour_brewer(type="qual", palette="Accent",
                          name="PDF values")
    + theme_bw()
)
p9
```



9.13 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[5]. Once you have it, you can load it

[5] `xkcd.com/1350/xkcd-Regular.otf`

into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[6]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight()` methods. Similarly, we'll change the body text to size 12.

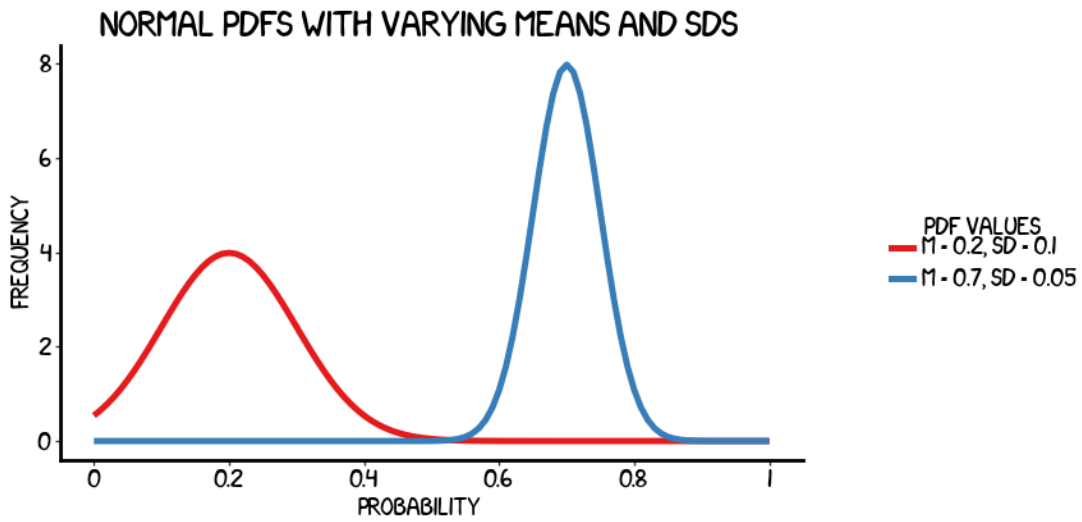
```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

[6] `https://matplotlib.org/api/font_manager_api.html`

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        aes(colour='"M = 0.2, SD = 0.1"'), fun=st.norm.pdf,
        args=dict(loc=0.2, scale=0.1), size=2,
    )
    + stat_function(
        aes(colour='"M = 0.7, SD = 0.05"'), fun=st.norm.pdf,
        args=dict(loc=0.7, scale=0.05), size=2,
    )
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
    + scale_colour_brewer(type="qual", palette="Set1",
                          name="PDF values")
    + theme(
        legend_direction="vertical",
        legend_title_align="center",
        legend_box_spacing=0.5,
        legend_entry_spacing_x=10,
        legend_key=element_blank(),
        axis_line_x=element_line(size=2,
                                 colour="black"),
        axis_line_y=element_line(size=2,
                                 colour="black"),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p9
```



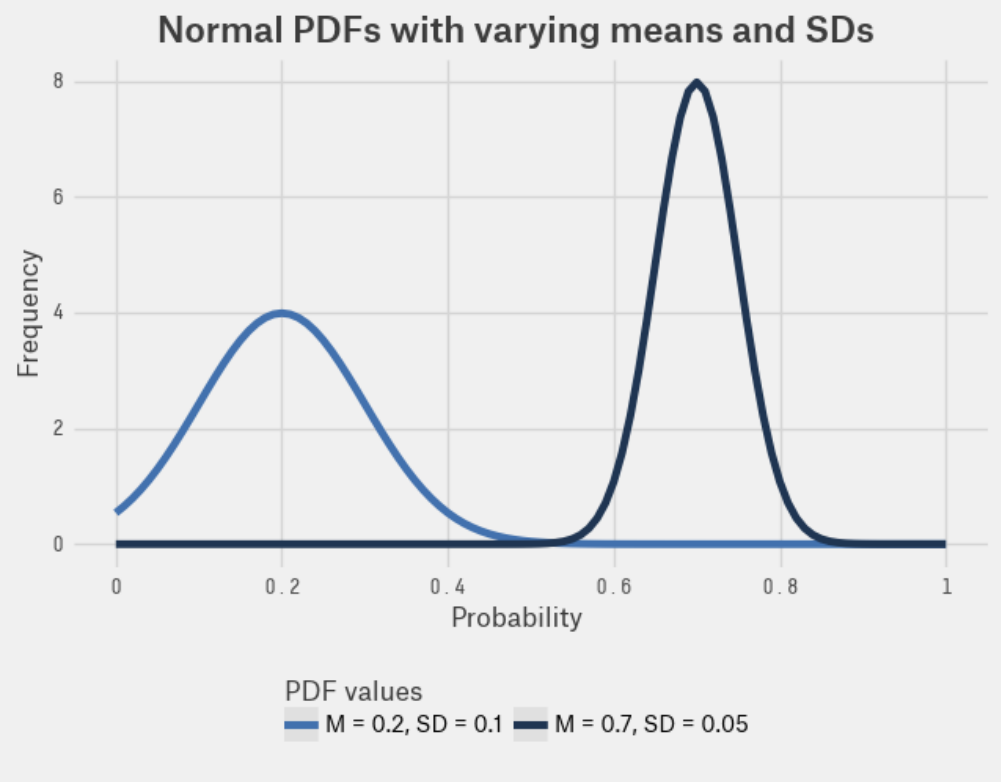
9.14 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[7]). Below we've applied `theme_538()`, which approximates graphs on the website FiveThirtyEight. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[8] and 'Decima Mono Pro'[9] in `axis_title`, `legend_title`, `legend_text`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
legend_text = fm.FontProperties(fname=agr)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
legend_text.set_size(10)
```

[7] http://plotnine.readthedocs.io/en/stable/api.html#themes
[8] https://commercialtype.com/catalog/atlas/atlas_grotesk
[9] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
axis_text.set_size(12)
body_text.set_size(10)

p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        aes(colour='"M = 0.2, SD = 0.1"'), fun=st.norm.pdf,
        args=dict(loc=0.2, scale=0.1), size=2,
    )
    + stat_function(
        aes(colour='"M = 0.7, SD = 0.05"'), fun=st.norm.pdf,
        args=dict(loc=0.7, scale=0.05), size=2,
    )
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
    + scale_colour_manual(name="PDF values",
                          values=["#4271AE", "#1F3552"])
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        legend_position="bottom",
        legend_direction="horizontal",
        legend_box_spacing=0.5,
        legend_title=element_text(fontproperties=axis_text),
        legend_text=element_text(fontproperties=legend_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p9
```



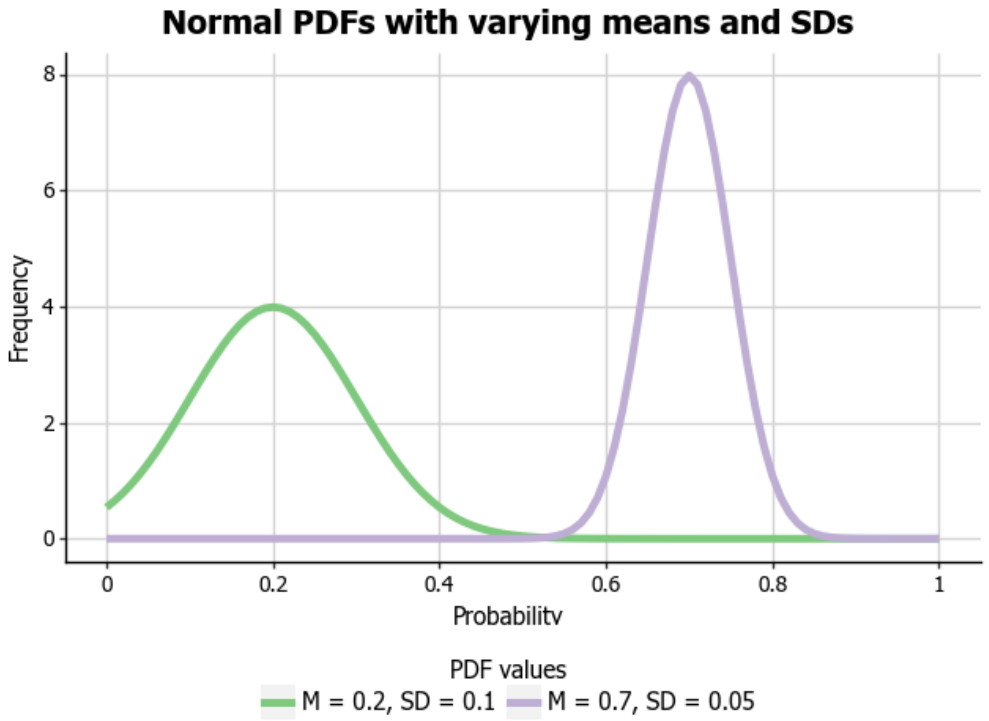
9.15 Creating your own theme

Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        aes(colour='"M = 0.2, SD = 0.1"'), fun=st.norm.pdf,
        args=dict(loc=0.2, scale=0.1), size=2,
```

```
    )
    + stat_function(
        aes(colour='"M = 0.7, SD = 0.05"'), fun=st.norm.pdf,
        args=dict(loc=0.7, scale=0.05), size=2,
    )
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
    + scale_colour_brewer(type="qual", palette="Accent",
                          name="PDF values")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p9
```



9.16 Adding areas under the curve

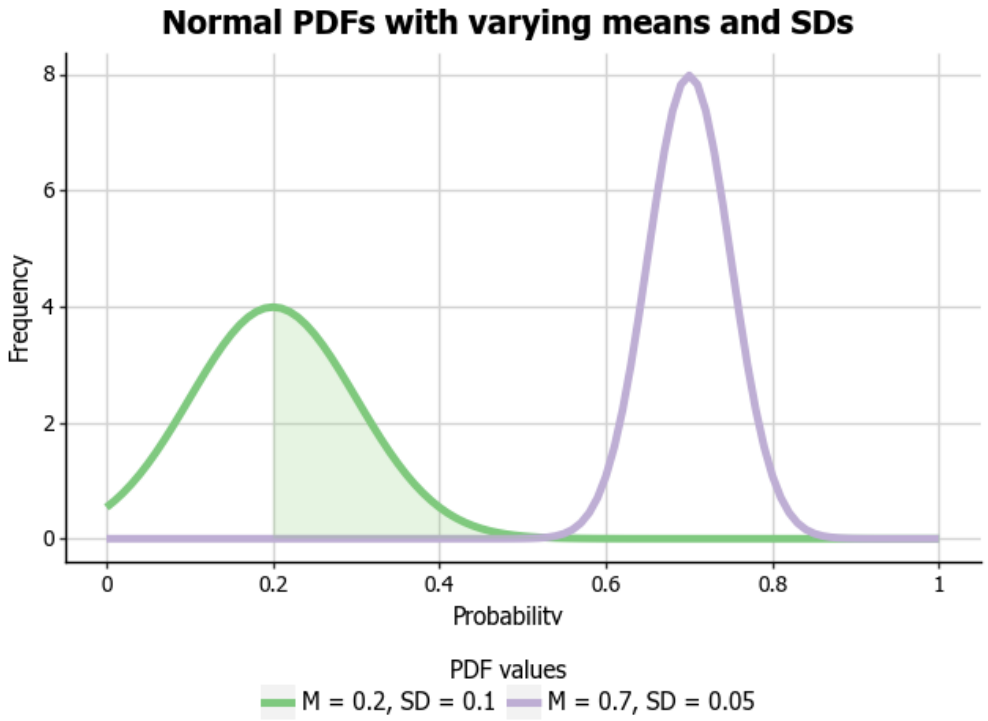
If we want to shade an area under the curve, we first need to create an additional `DataFrame`. This `DataFrame` contains all of the x-values for the part of the curve we want to shade, and then generates all of the relevant y-values from this distribution. For example, let's say that we only want to shade our first normal distribution function from the median upwards. We can create a `DataFrame` which iterates through all of the values between the median (0.2) to an upper bound (0.6), and extracts the matching y-values from `st.norm.pdf(i, loc=0.2, scale=0.1)`.

```
d1 = DataFrame({
    "x": 0.2,
    "y": st.norm.pdf(0.2, loc=0.2, scale=0.1)
}, index=[0])
for i in np.arange(0.21, 0.6, 0.01):
    d1 = d1.append(
        DataFrame({
            "x": i,
```

```
            "y": st.norm.pdf(i, loc=0.2, scale=0.1)
        }, index=[0])
    )
d1 = d1.reset_index(drop=True)
```

We can then add this DataFrame to the plot as an area curve (`geom_area(d1, aes("x", "y"), fill="#84CA72", alpha=0.2)`). You can see that we've made it the same colour as the function line, but made it slightly transparent using the `alpha` argument.

```
p9 = (
    ggplot(DataFrame({"x": [-0, 1]}), aes("x"))
    + stat_function(
        aes(colour='"M = 0.2, SD = 0.1"'), fun=st.norm.pdf,
        args=dict(loc=0.2, scale=0.1), size=2,
    )
    + stat_function(
        aes(colour='"M = 0.7, SD = 0.05"'), fun=st.norm.pdf,
        args=dict(loc=0.7, scale=0.05), size=2,
    )
    + geom_area(d1, aes("x", "y"), fill="#84CA72",
                alpha=0.2)
    + scale_x_continuous(breaks=np.arange(0, 1.1, 0.2),
                         limits=[0, 1])
    + ggtitle("Normal PDFs with varying means and SDs")
    + labs(x="Probability", y="Frequency")
    + scale_colour_brewer(type="qual", palette="Accent",
                          name="PDF values")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p9
```



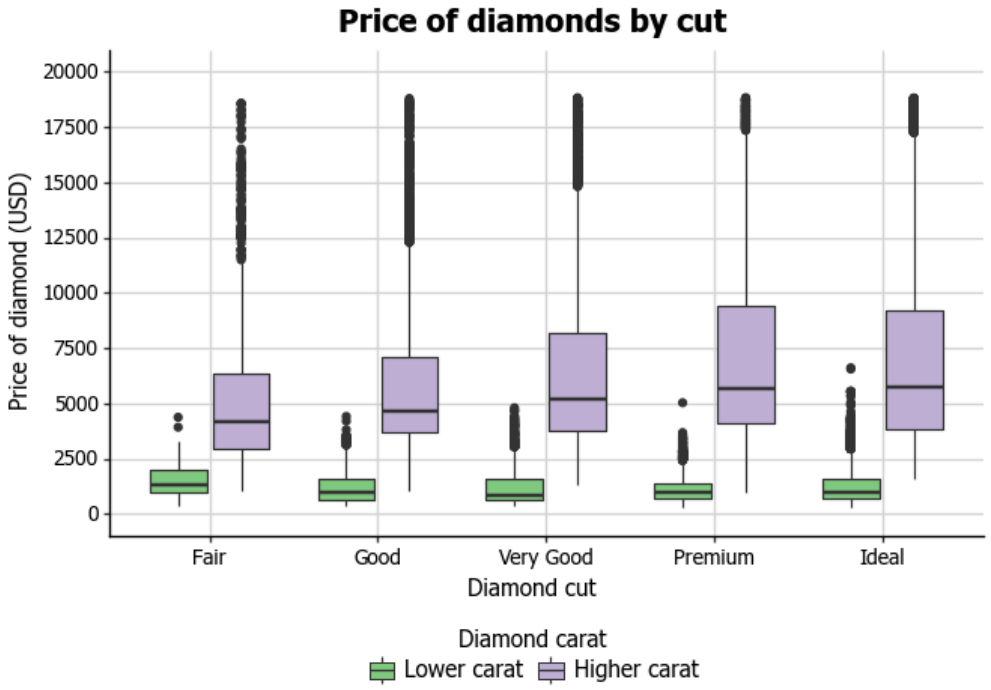
With this addition, you can see we have now recreated the chart at the beginning of this chapter.

Chapter 10

Boxplots

10.1 Introduction

In this chapter, we will work towards creating the boxplot below. We will take you from a basic boxplot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from plotnine import data
from pandas import DataFrame
```

We then need to load in the data, as below.

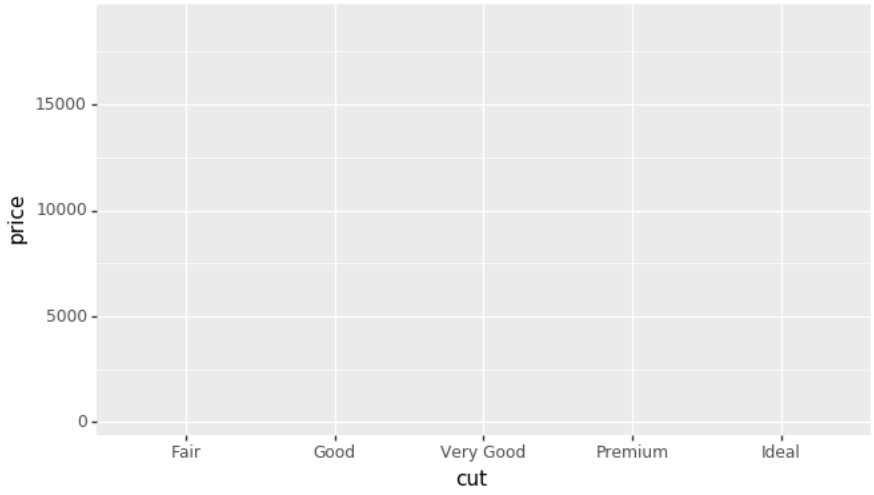
```
diamonds = data.diamonds
```

10.2 Basic ggplot structure

In order to initialise a boxplot we tell ggplot that `diamonds` is our data, and specify that our x-axis plots the `cut` variable and our y-axis plots the `price` variable. You may have noticed that we put our variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `cut` to the x-axis and `price` to the y-axis.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

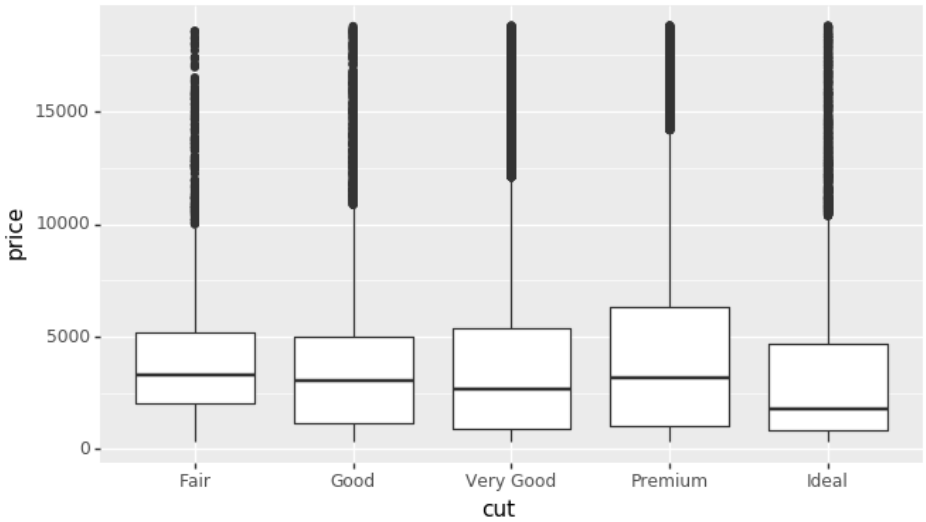
```
p10 = ggplot(diamonds, aes("cut", "price"))
p10
```



10.3 Basic boxplot

We can do this using `geoms`. In the case of a boxplot, we use the `geom_boxplot()` geom.

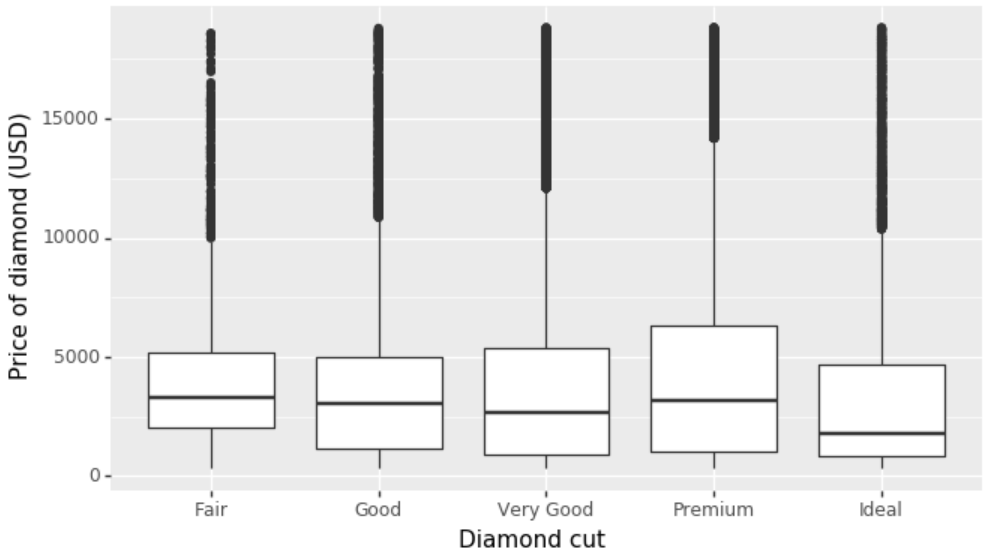
```
p10 = ggplot(diamonds, aes("cut", "price")) + geom_boxplot()
p10
```



10.4 Customising axis labels

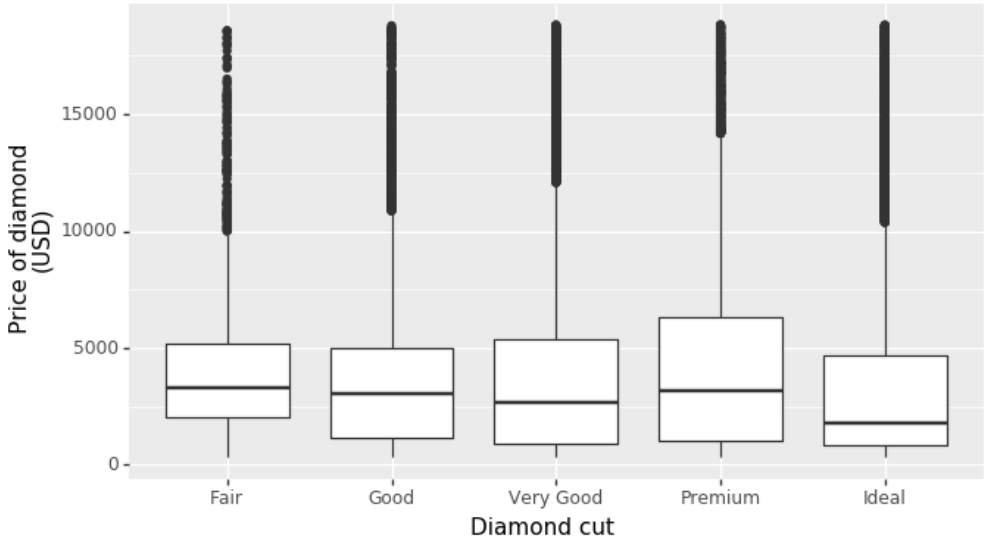
In order to change the axis labels, we have used the `xlab` and `ylab` options. In each, we add the desired name as an argument.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot()
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
)
p10
```



`ggplot` also allows for the use of multiline names (in both axes and titles). Here, we've changed the y-axis label so that it goes over two lines using the `\n` character to break the line.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot()
    + xlab("Diamond cut")
    + ylab("Price of diamond\n(USD)")
)
p10
```



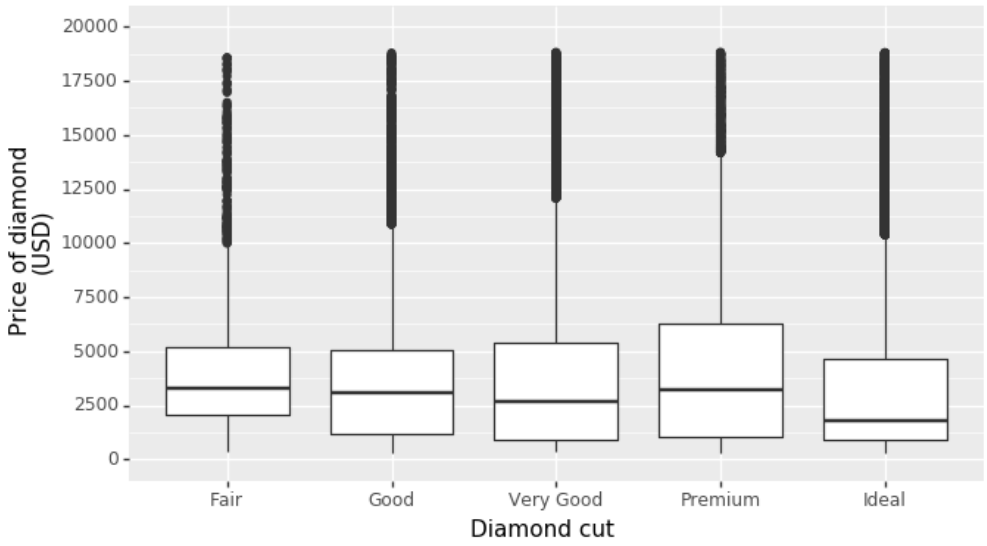
10.5 Changing axis ticks

To change the x-axis tick marks, we can use the `scale_x_continuous` option. Similarly, to change the y-axis we can use the `scale_y_continuous` option. Here we will change the y-axis to every $2500 rather than the default of $5000. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[1] which generates a sequence from your selected start, stop and step values respectively. Note that because of Python's indexing, you need to set the `stop` argument to be one number more than your desired maximum.

Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis. We've also included this in our `scale_y_continuous` option, increasing the maximum value to $20000.

```
p10 = (
    ggplot(diamonds, aes("cut", "price")) + geom_boxplot()
    + xlab("Diamond cut") + ylab("Price of diamond\n(USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500), limits=[0, 20000])
)
p10
```

[1] https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html



10.6 Adding a title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot()
    + xlab("Diamond cut")
    + ylab("Price of diamond\n(USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
)
p10
```

10.7 Changing the colour of the boxes

To change the line and fill colours of the box plot, we add a valid colour to the `colour` and `fill` arguments in `geom_boxplot()`. `plotnine` uses the colour palette utilised by `matplotlib`, and the full set of named colours recognised by `ggplot` is here[2]. Let's try changing our box lines and fills to `rebeccapurple` and `lightskyblue` respectively.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(colour="rebeccapurple", fill="lightskyblue")
    + xlab("Diamond cut")
    + ylab("Price of diamond\n(USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
)
p10
```

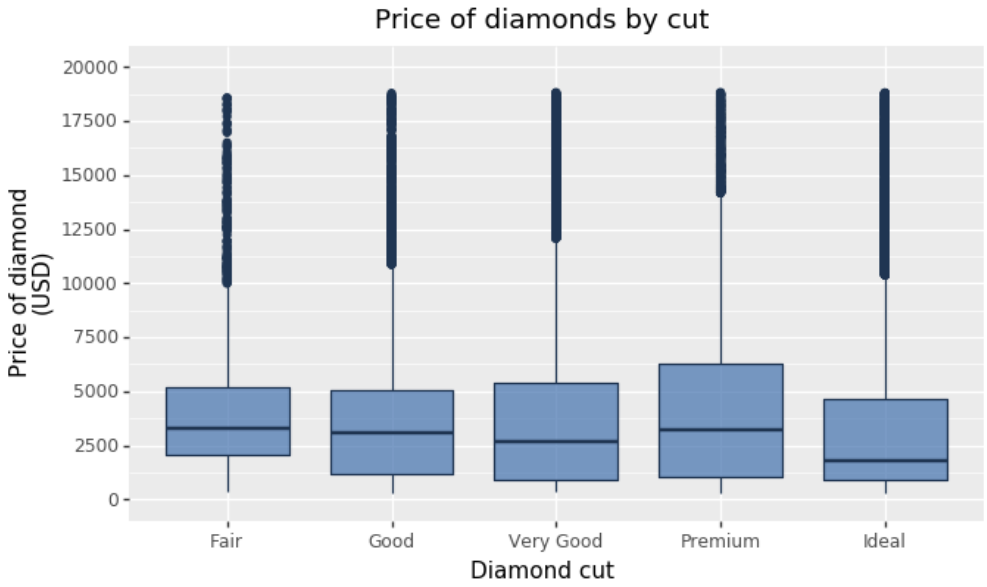
[2] https://matplotlib.org/examples/color/named_colors.html

If you want to go beyond the options in the list above, you can also specify exact HEX colours by including them as a string preceded by a hash, e.g., `"#FFFFFF"`. Below, we have called two shades of blue for the fill and lines using their HEX codes.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(colour="#1F3552", fill="#4271AE")
    + xlab("Diamond cut")
    + ylab("Price of diamond\n(USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
)
p10
```

You can also specify the degree of transparency in the box fill area using the argument `alpha` in `geom_boxplot()`. This ranges from 0 to 1.

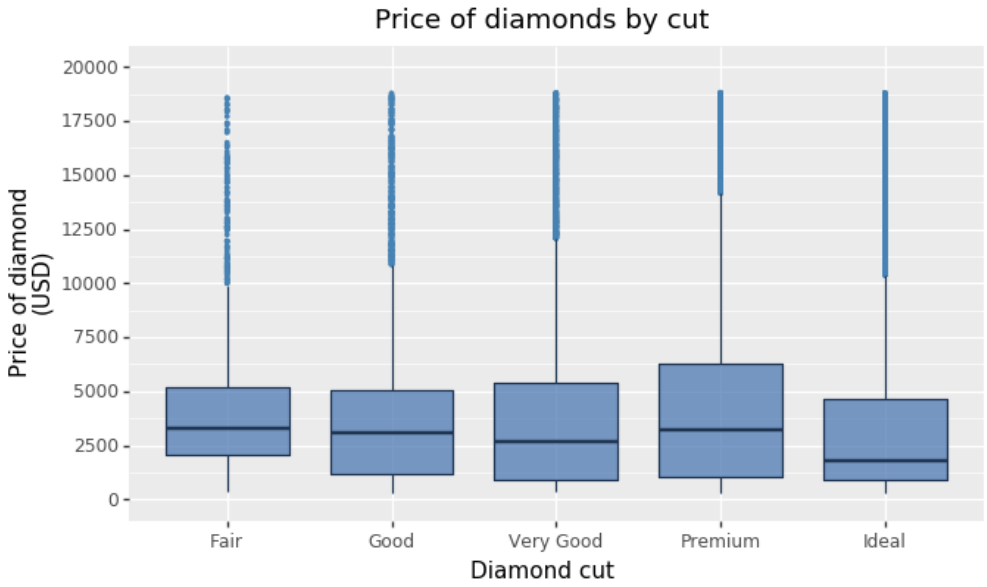
```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(colour="#1F3552", fill="#4271AE",
                   alpha=0.7)
    + xlab("Diamond cut")
    + ylab("Price of diamond\n(USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
)
p10
```



Finally, you can change the appearance of the outliers as well, using the arguments `outlier.colour` and `outlier.shape` in `geom_boxplot` to change the colour and shape respectively. The shape arguments for `plotnine` are the same as those available in `matplotlib`, and are therefore a little more limited than those in R's implementation of `ggplot2`. Nonetheless, there is a good range of options. The allowed arguments are here[3]. Here we will make the outliers small solid circles (using `outlier.shape="."`) and make them coloured `steelblue` (using `outlier.colour="steelblue"`).

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(
        colour="#1F3552",
        fill="#4271AE",
        alpha=0.7,
        outlier_shape=".",
        outlier_colour="steelblue",
    )
    + xlab("Diamond cut")
    + ylab("Price of diamond\n(USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
)
p10
```

[3] https://matplotlib.org/api/markers_api.html



10.8 Using the white theme

As explained in the previous chapters, we can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()`.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(
        colour="#1F3552",
        fill="#4271AE",
        alpha=0.7,
        outlier_shape=".",
        outlier_colour="steelblue",
    )
    + xlab("Diamond cut")
    + ylab("Price of diamond\n(USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
    + theme_bw()
)
p10
```

10.9 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[4]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

[4] `xkcd.com/1350/xkcd-Regular.otf`

We can then call methods on these objects (the list of available methods is here[5]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

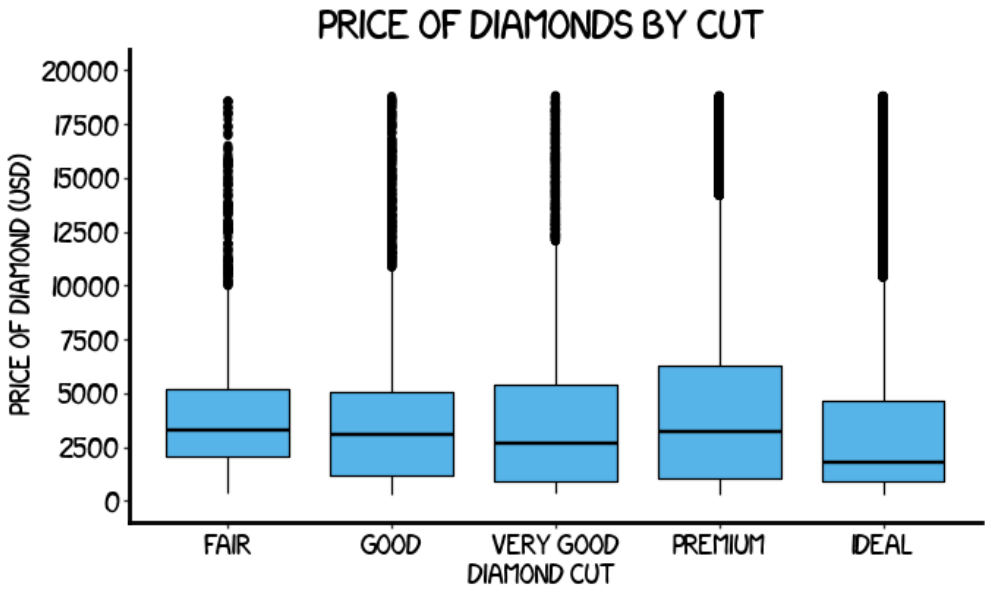
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(colour="black", fill="#56B4E9")
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
    + theme(
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
```

[5] https://matplotlib.org/api/font_manager_api.html

```
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p10
```



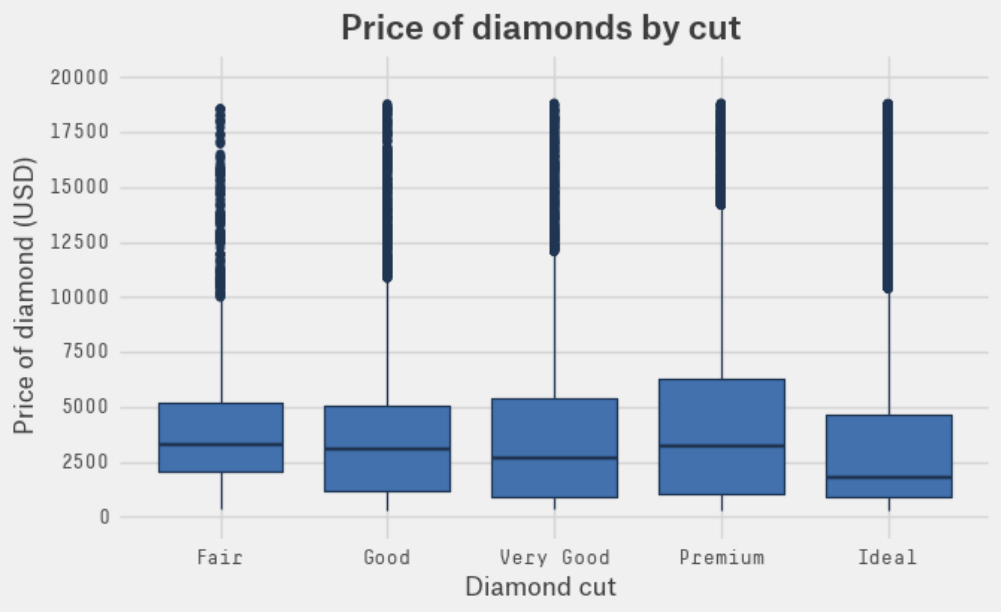
10.10 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[6]). Below we've applied `theme_538()`, which approximates graphs in the nice FiveThirtyEight website. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[7] and 'Decima Mono Pro'[8] in `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
```

[6] http://plotnine.readthedocs.io/en/stable/api.html#themes
[7] https://commercialtype.com/catalog/atlas/atlas_grotesk
[8] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
axis_text.set_size(12)
body_text.set_size(10)

p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(colour="#1F3552", fill="#4271AE")
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p10
```

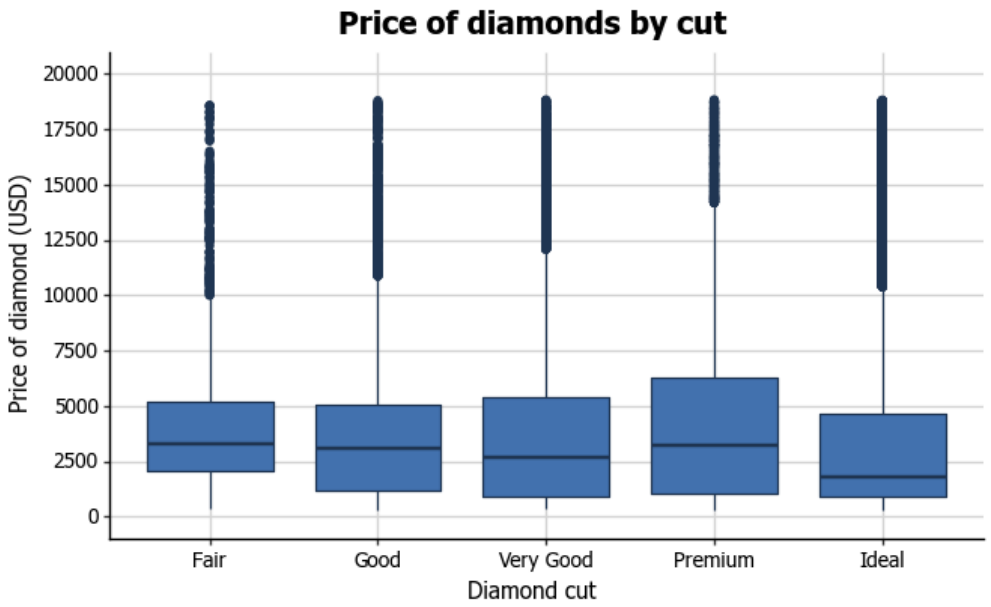


10.11 Creating your own theme

Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(colour="#1F3552", fill="#4271AE")
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p10
```



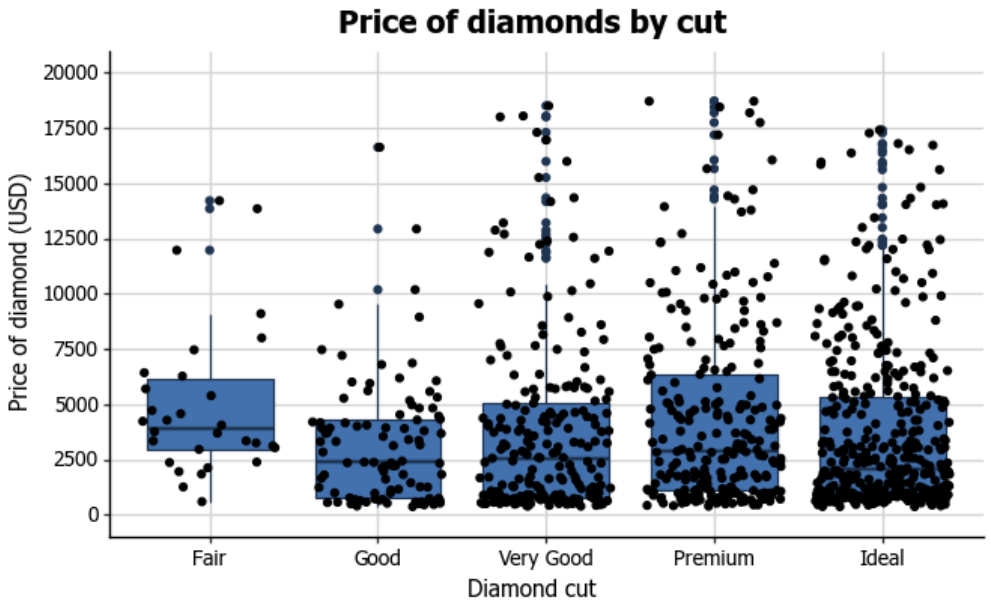
10.12 Boxplot extras

An extra feature you can add to boxplots is to overlay all of the points for that group on each boxplot in order to get an idea of the sample size of the group. This can be achieved using by adding the `geom_jitter()` option. As `diamonds` is such a large dataset, we'll first take a small sample to illustrate this.

```
diamonds_sample = diamonds.sample(1000)
```

```
p10 = (
    ggplot(diamonds_sample, aes("cut", "price"))
    + geom_boxplot(colour="#1F3552", fill="#4271AE")
    + geom_jitter()
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
```

```
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p10
```

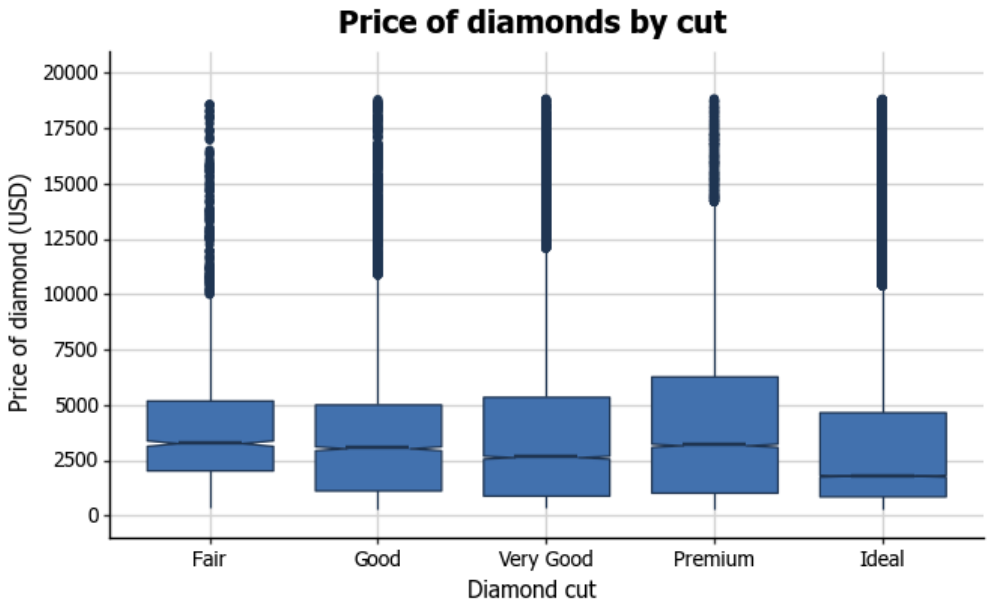


We can see that the `Fair` group has a smaller sample than the other categories, indicating that it may not give as reliable information as the other cut types.

Another thing you can do with your boxplot is add a notch to the box where the median sits to give a clearer visual indication of how the data are distributed within the IQR. You achieve this by adding the argument `notch=True` to the `geom_boxplot()` geom.

```
p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(colour="#1F3552", fill="#4271AE",
                   notch=True)
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
```

```
    + ggtitle("Price of diamonds by cut")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p10
```



10.13 Grouping by another variable

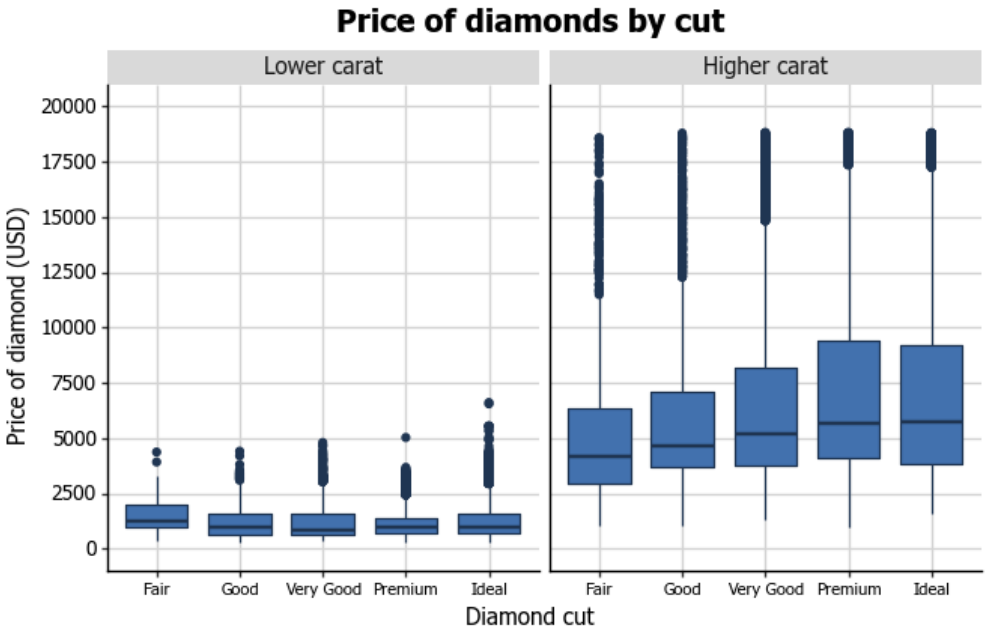
You can also easily group boxplots by the levels of another variable. There are two options, in separate (panel) plots, or in the same plot.

We first need to do a little data wrangling. To create our grouping variable, we'll median-split `carat` so that this is categorical, and made it into a new la-

belled factor variable called `carat_c`.

In order to produce a panel plot by this categorical carat variable, we add the `facet_grid(".~carat_c")` option to the plot. Note that unlike in R's `ggplot`, you need to include the arguments in `facet_grid` in quote marks.

```
diamonds["carat_c"] = pd.qcut(
    diamonds["carat"], 2, labels=["Lower carat", "Higher carat"]
)


p10 = (
    ggplot(diamonds, aes("cut", "price"))
    + geom_boxplot(colour="#1F3552", fill="#4271AE")
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=8),
        axis_text_y=element_text(colour="black", size=10),
    )
    + facet_grid(". ~ carat_c")
)
p10
```



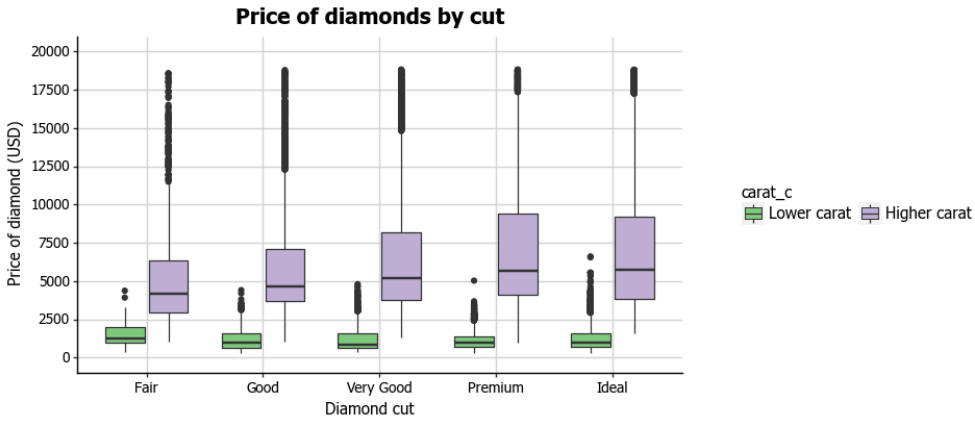
In order to plot the two `carat` levels in the same plot, we need to add a couple of things. Firstly, in the `ggplot` function, we add a `fill=carat_c` argument to `aes`. Secondly, we change the manual colours using the schemes from ColorBrewer[9]. Here we have used the `scale_fill_brewer` option with the quantitative scale `Accent`. More information on using `scale_colour_brewer` is here[10].

```
p10 = (
    ggplot(diamonds, aes("cut", "price", fill="carat_c"))
    + geom_boxplot()
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
                         limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
    + theme(
        legend_direction="horizontal",
        legend_box_spacing=0.4,
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
```

[9] http://colorbrewer2.org/

[10] http://plotnine.readthedocs.io/en/stable/generated/plotnine.scales.scale_color_brewer.html#plotnine.scales.scale_color_brewer

```
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
    + scale_fill_brewer(type="qual", palette="Accent")
)
p10
```

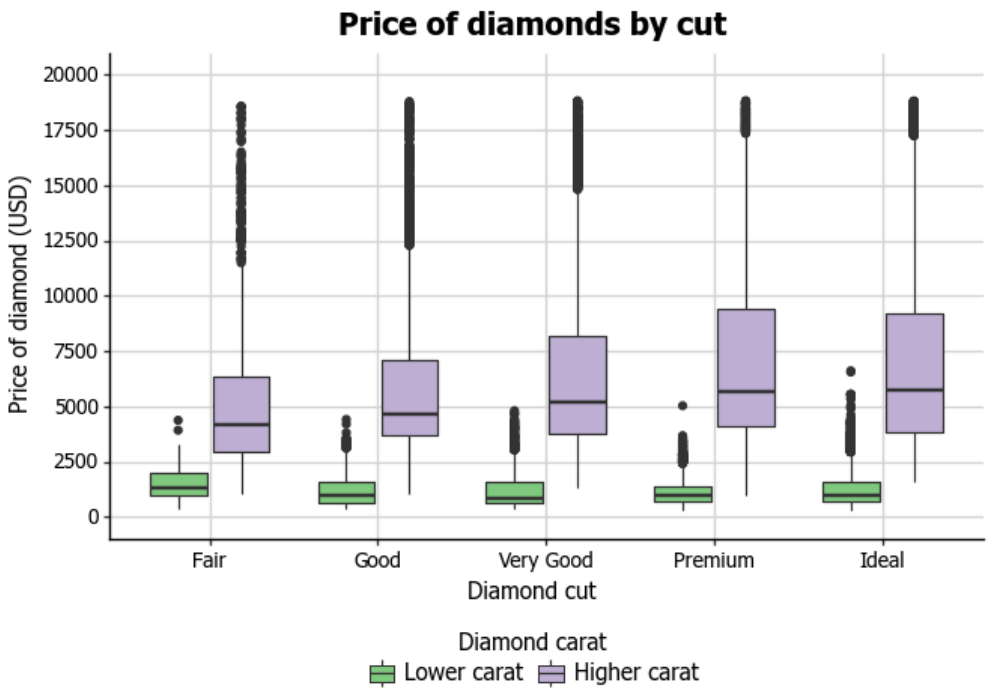


10.14 Formatting the legend

Finally, we can format the legend. Firstly, we can change the position by adding the `legend_position="bottom"` argument to the `theme` option, which moves the legend under the plot. We can change the orientation of the legend to horizontal by then adding `legend_direction="horizontal"` to `theme`. We can also centre the legend by adding `legend_title_align="center"`. We can adjust the legend position using `legend_box_spacing=0.4`. We can get rid of the grey background behind the legend keys using `legend_key=element_blank()`. Lastly, we can fix the title by adding the `name="Diamond carat"` argument to the `scale_fill_brewer` option.

```
p10 = (
    ggplot(diamonds, aes("cut", "price", fill="carat_c"))
    + geom_boxplot()
    + xlab("Diamond cut")
    + ylab("Price of diamond (USD)")
    + scale_y_continuous(breaks=np.arange(0, 20001, 2500),
```

```
                        limits=[0, 20000])
    + ggtitle("Price of diamonds by cut")
    + theme(
        legend_position="bottom",
        legend_direction="horizontal",
        legend_title_align="center",
        legend_box_spacing=0.4,
        legend_key=element_blank(),
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
    + scale_fill_brewer(type="qual", palette="Accent",
                        name="Diamond carat")
)
p10
```



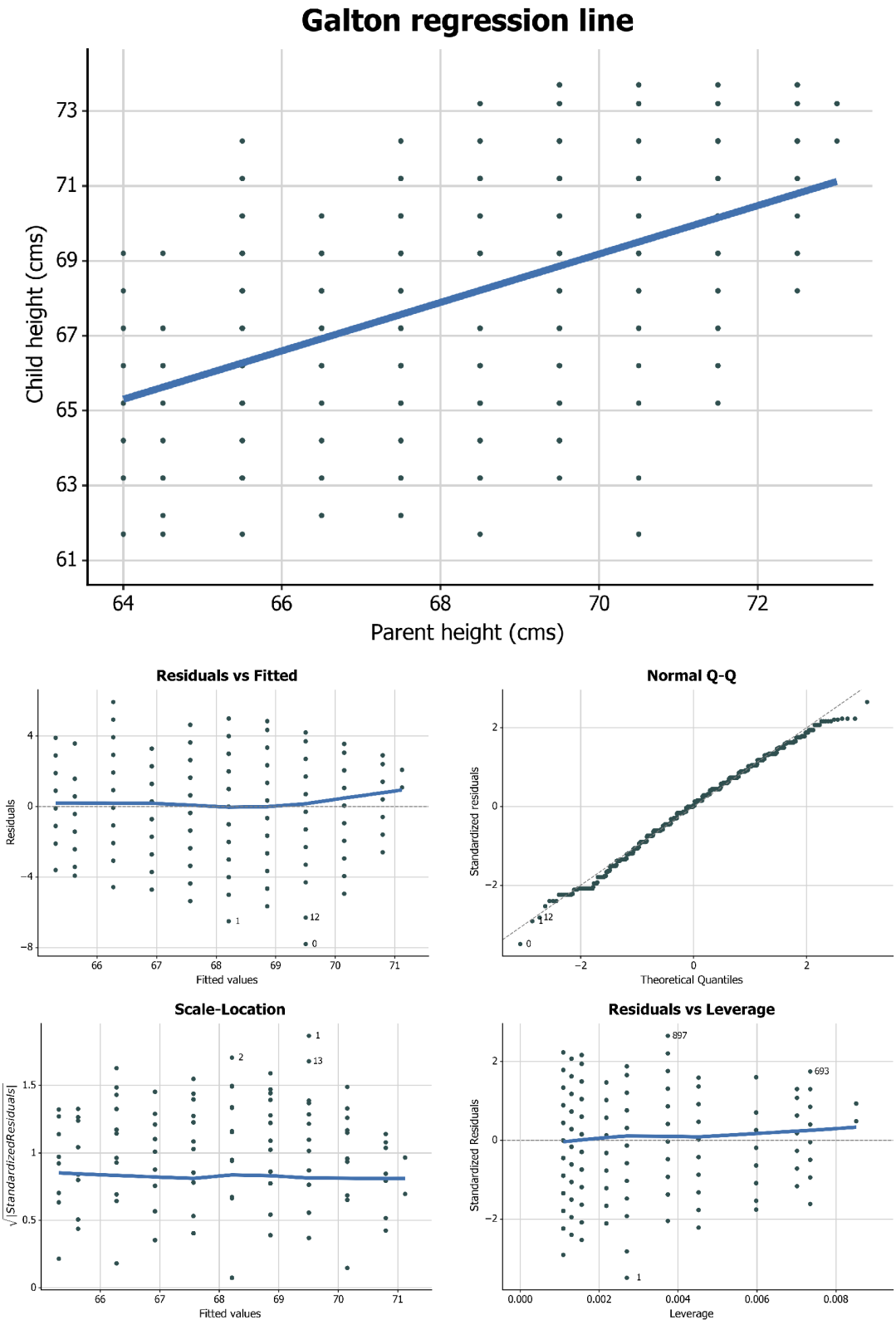
Chapter 11

Linear regression plots

11.1 Introduction

In this chapter we will explain how to create regression plots using the Galton parent-child height dataset, created by Francis Galton himself. Galton was a 19th century scientist, who among other achievements developed the concept of regression modelling in order to explore his data on the relationship between parents' and childrens' heights.

In this tutorial, we will work towards creating the trend line and diagnostics plots below. We will take you from a basic regression plot and explain all the customisations we add to the code step-by-step. We'll also explain how to create all of the regression diagnostic plots below using `plotnine`.


Galton regression line
Child height (cms)
Parent height (cms)
Residuals vs Fitted
Residuals
Fitted values
Normal Q-Q
Standardized residuals
Theoretical Quantiles
Scale-Location
√|StandardizedResiduals|
Fitted values
Residuals vs Leverage
Standardized Residuals
Leverage

The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs;
- `numpy` to do some basic numeric calculations in our graphing;
- `statsmodels.api` to create the regression model and get the variables needed for our diagnostic plots; and
- the `matplotlib` packages `pyplot`, `gridspec` and `image` to combine all of our plots into one image.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import matplotlib.image as mpimg

import plotnine
plotnine.options.figure_size=(7, 4.8)

from plotnine import *
from pandas import DataFrame
```

Next, we need to download our dataset. As this is an R dataset and is not included with the `plotnine` default datasets, we need to download it.

```
galton = pd.read_csv(
    "https://vincentarelbundock.github.io/Rdatasets/csv/HistData/Galton.csv",
    header=0,
    usecols=[1, 2],
)
```

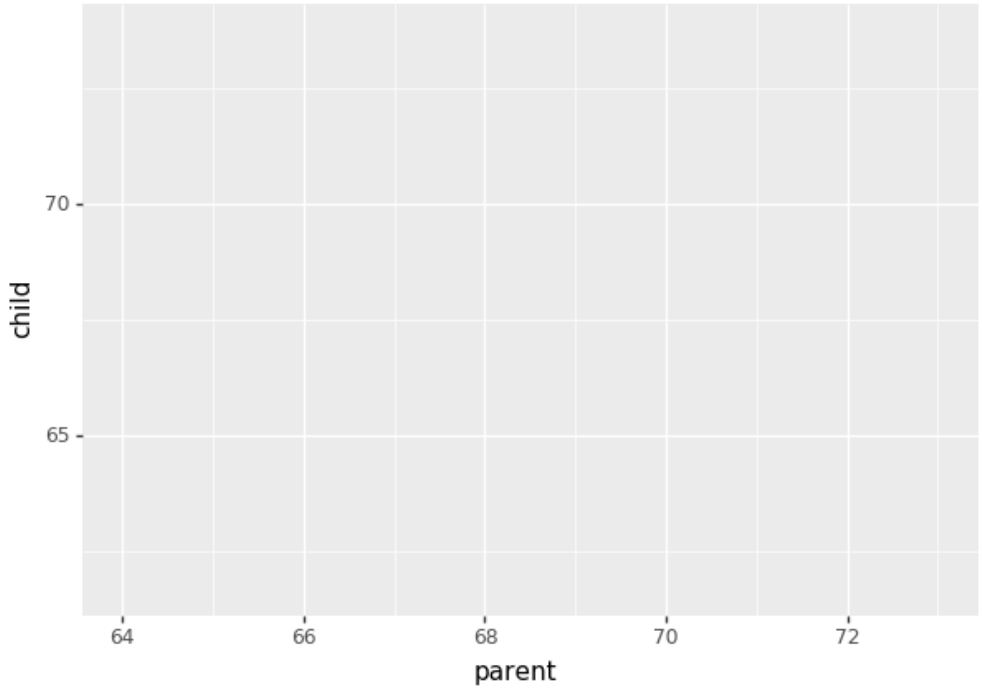
11.2 Basic ggplot structure

In order to initialise a regression plot we tell ggplot that `galton` is our data, and specify that our x-axis plots the `parent` variable and our y-axis plots the `child` variable. You may have noticed that we put our x- and y-variables inside a method called `aes`. This is short for aesthetic mappings, and determines how

the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `parent` to the x-axis and `child` to the y-axis.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

```
p11 = ggplot(galton, aes("parent", "child"))
p11
```



11.3 Trend line plot

We'll be estimating a model of the form $y_1 = \beta_0 + \beta_1 x_1 + e_1$ where x_1, y_1 is an observation of the height of a parent and their child.

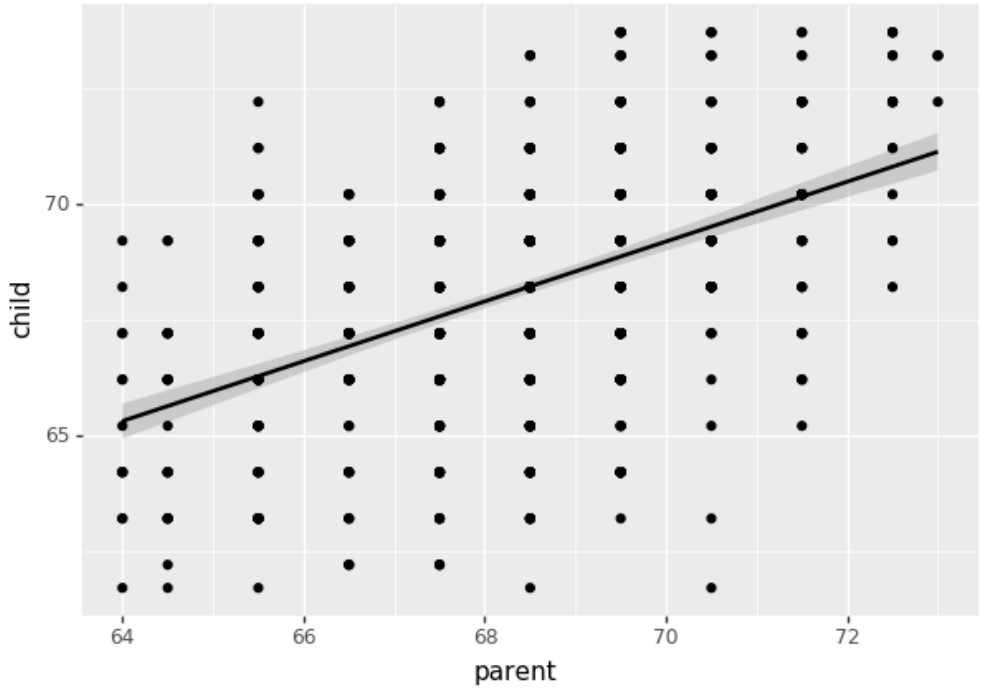
Before we plot the line, let's have a look at the model in a little more detail. We'll create our regression model using the `OLS` method from `statsmodels.api`.

```
X = galton["parent"]
y = galton["child"]
X = sm.add_constant(X)

model = sm.OLS(y, X).fit()
model.summary()
```

From this we can see that our regression slope should take the form of $y_1 = 23.94 + 0.65x_1 + e_1$: that is, it should have an x-intercept of 23.94 and a slope of 0.65. Also note that the adjusted R-squared is low (0.21), indicating that there is quite a lot of error in this model.

Let's now visualise it. We can do this by adding the option `geom_smooth(method="lm")` to the chart. We'll also plot the raw data to see how well the regression line fits, by adding the option `geom_point(shape="o")`.

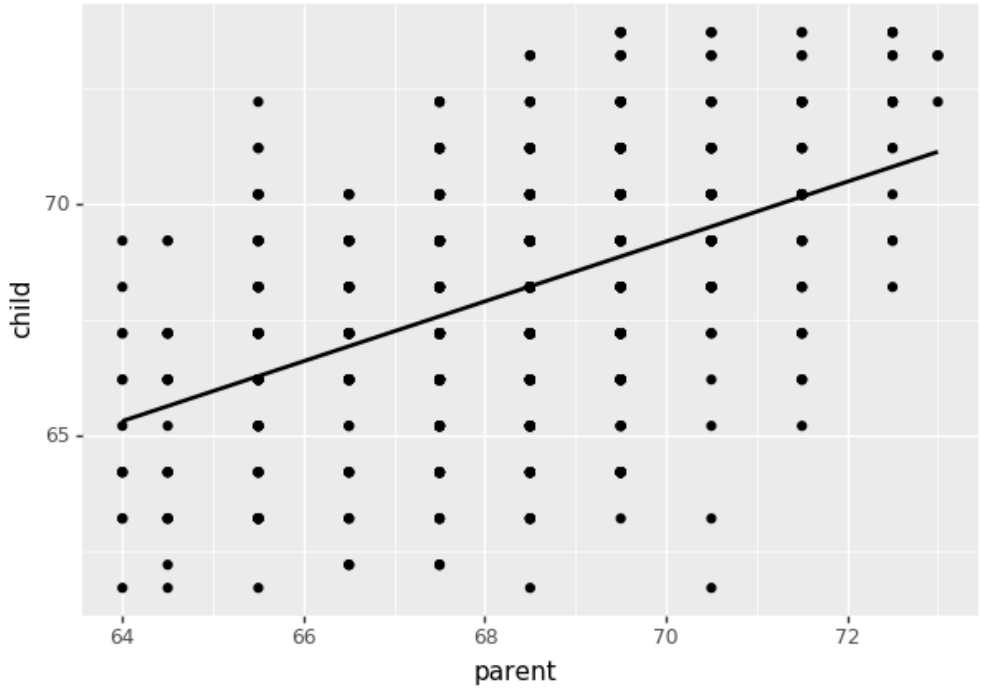
```
p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_smooth(method="lm")
    + geom_point(shape="o")
)
p11
```



Our plot reflects what the model has told us. Children's ages appear to increase by just over half a centimetre for every centimetre increase in parent's height, and the wide spread of the raw data shows why the adjusted R^2 is relatively low.

`geom_smooth` can be customized, for example, not to include the confidence region.

```
p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_smooth(method="lm", se=False)
    + geom_point(shape="o")
)
p11
```

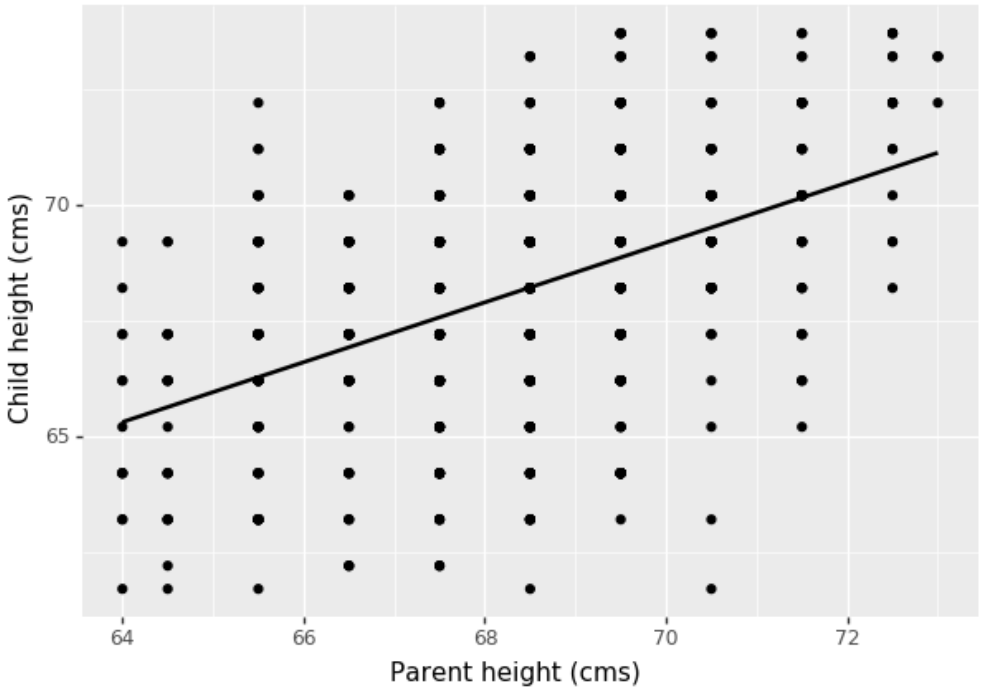


Before continuing it is a good idea to fix the axis labels and add a title.

11.4 Customising axis labels

We can change the text of the axis labels using the `xlab` and `ylab` options, with the names passed as a string in each.

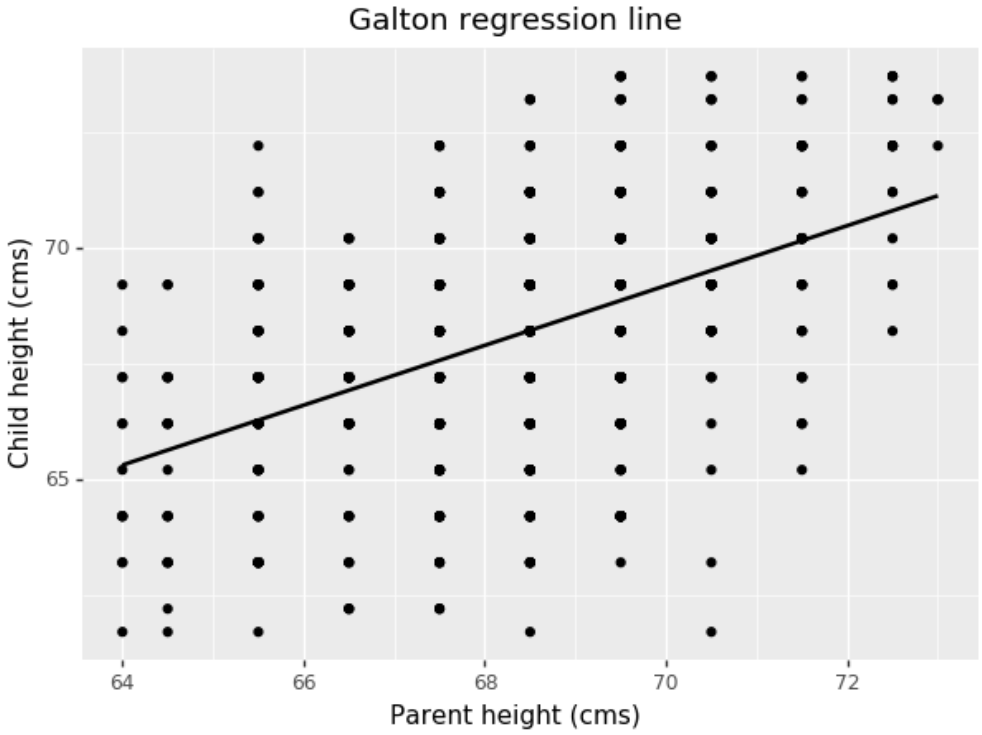
```
p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_smooth(method="lm", se=False)
    + geom_point(shape="o")
    + xlab("Parent height (cms)")
    + ylab("Child height (cms)")
)
p11
```



11.5 Adding a title

Similarly, we can add a title using the `labs` option.

```
p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_smooth(method="lm", se=False)
    + geom_point(shape="o")
    + xlab("Parent height (cms)")
    + ylab("Child height (cms)")
    + labs(title="Galton regression line")
)
p11
```



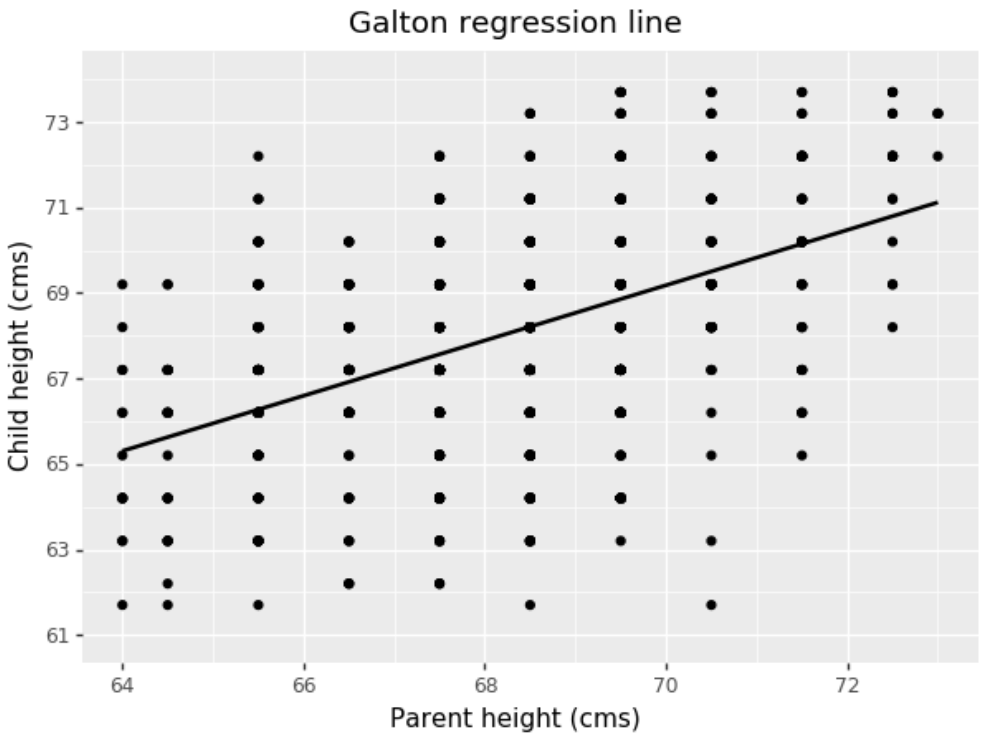
11.6 Adjusting the axis scales

To change the x-axis tick marks, we can use the `scale_x_continuous` option. Similarly, to change the y-axis we can use the `scale_y_continuous` option. Here we will change the y-axis to every 2cms to match the x-axis. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[1] which generates a sequence from your selected start, stop and step

[1] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html`

values respectively. Note that because of Python's indexing system, you need to set the maximum as one past your desired number. Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis.

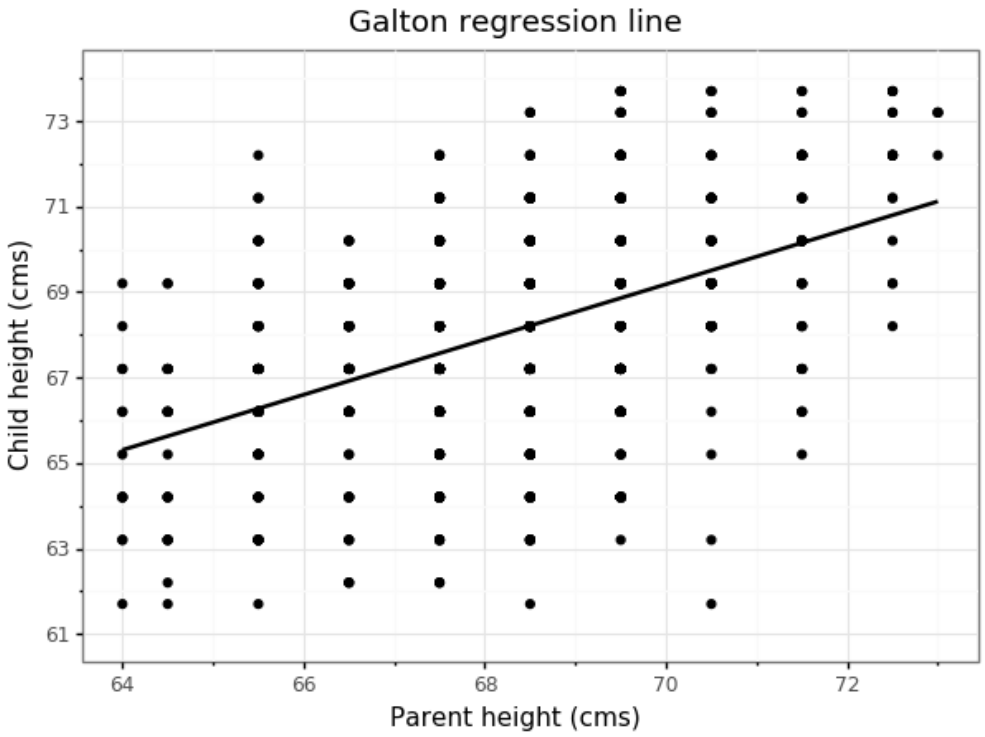
```
p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_smooth(method="lm", se=False)
    + geom_point(shape="o")
    + xlab("Parent height (cms)")
    + ylab("Child height (cms)")
    + labs(title="Galton regression line")
    + scale_y_continuous(breaks=np.arange(61, 74, 2),
                         limits=[61, 74])
)
p11
```



11.7 Using the white theme

As explained in the previous chapters, we can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()`.

```
p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_smooth(method="lm", se=False)
    + geom_point(shape="o")
    + xlab("Parent height (cms)")
    + ylab("Child height (cms)")
    + labs(title="Galton regression line")
    + scale_y_continuous(breaks=np.arange(61, 74, 2),
                         limits=[61, 74])
    + theme_bw()
)
p11
```



11.8 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[2]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

We can then call methods on these objects (the list of available methods is here[3]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

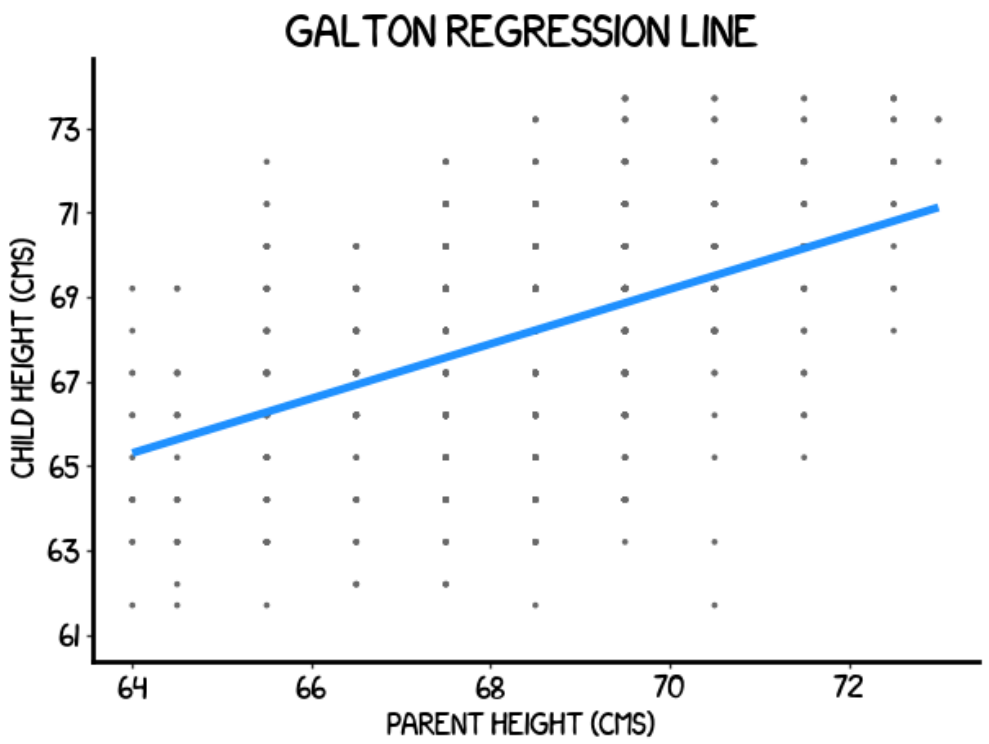
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;

[2] `xkcd.com/1350/xkcd-Regular.otf`
[3] `https://matplotlib.org/api/font_manager_api.html`

- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_point(shape=".", colour="dimgrey")
    + geom_smooth(method="lm", se=False, colour="dodgerblue",
                  size=2)
    + xlab("Parent height (cms)")
    + ylab("Child height (cms)")
    + labs(title="Galton regression line")
    + scale_y_continuous(breaks=np.arange(61, 74, 2),
                         limits=[61, 74])
    + theme(
        axis_line_x=element_line(size=2, colour="black"),
        axis_line_y=element_line(size=2, colour="black"),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p11
```



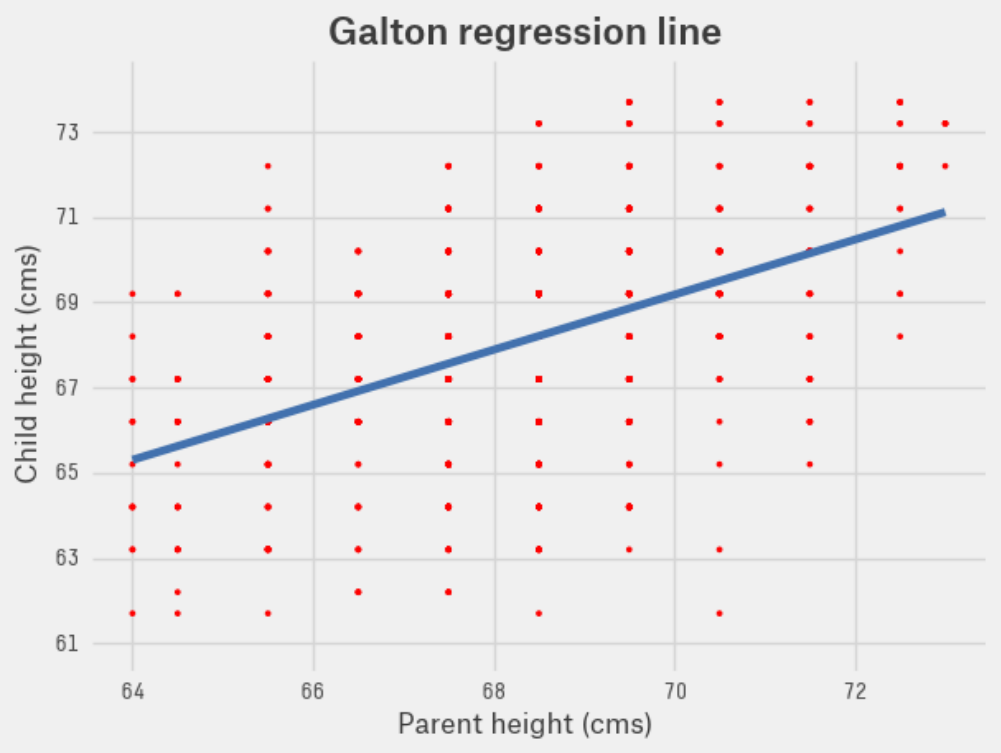
11.9 Using the 'Five Thirty Eight' theme

There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[4]). Below we've applied `theme_538()`, which approximates graphs in the nice FiveThirtyEight website. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[5] and 'Decima Mono Pro'[6] in `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
axis_text = fm.FontProperties(fname=agr)
```

[4] http://plotnine.readthedocs.io/en/stable/api.html#themes
[5] https://commercialtype.com/catalog/atlas/atlas_grotesk
[6] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/

```
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
axis_text.set_size(12)
body_text.set_size(10)


p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_point(shape=".", colour="red")
    + geom_smooth(method="lm", se=False, colour="#4271AE",
                  size=2)
    + xlab("Parent height (cms)")
    + ylab("Child height (cms)")
    + labs(title="Galton regression line")
    + scale_y_continuous(breaks=np.arange(61, 74, 2),
                         limits=[61, 74])
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p11
```



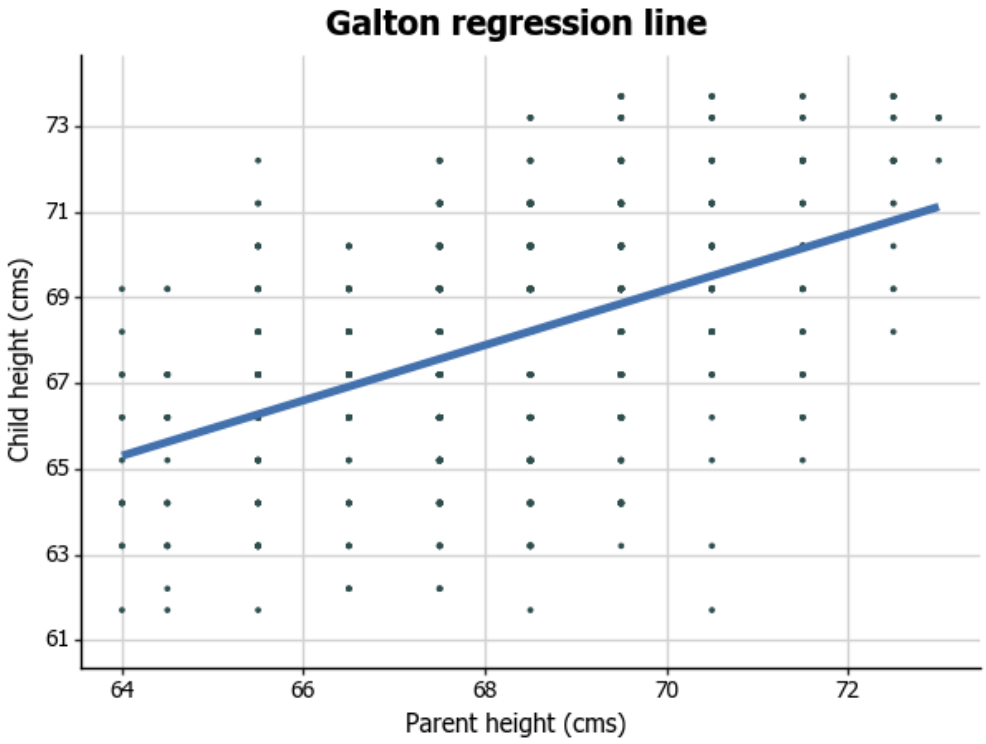
11.10 Creating your own theme

Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

```
p11 = (
    ggplot(galton, aes("parent", "child"))
    + geom_point(shape=".", colour="darkslategrey")
    + geom_smooth(method="lm", se=False, colour="#4271AE",
                  size=2)
    + xlab("Parent height (cms)")
```

```
    + ylab("Child height (cms)")
    + labs(title="Galton regression line")
    + scale_y_continuous(breaks=np.arange(61, 74, 2),
                         limits=[61, 74])
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p11
```



11.11 Regression diagnostics plots

An important part of creating regression models is evaluating how well they fit the data. In R, a range of diagnostic plots can be produced using packages like `ggfortify`. However, as `plotnine` doesn't yet have an equivalent package, we'll need to create these manually.

At the beginning of this chapter, we created a linear regression model object called `model`. We can get the values we need from this object to create all of our plots. We will extract:

- The model fitted values (`model_fitted_y`), the residuals (`model_residuals`) and the absolute residuals (`model_abs_resid`) for plotting the residuals against the fitted values;
- The normed residuals (`model_norm_residuals`) for creating the quantile-quantile plot;
- The absolute square root of the normed residuals (`model_norm_residuals_abs_sqrt`) plus the fitted values for creating the scale-location plot; and
- The leverage (`model_leverage`), Cook's distance (`model_cooks`) plus the normed residuals to create the leverage vs residuals plot.

To make it a bit easier to use these with `plotnine`, we then combine these into a `DataFrame`.

```
model_fitted_y = model.fittedvalues
model_residuals = model.resid
model_norm_residuals = model.get_influence().resid_studentized_internal
model_norm_residuals_abs_sqrt = np.sqrt(np.abs(model_norm_residuals))
model_abs_resid = np.abs(model_residuals)
model_leverage = model.get_influence().hat_matrix_diag
model_cooks = model.get_influence().cooks_distance[0]

diags = pd.concat(
    [
        model_fitted_y,
        model_residuals,
        pd.Series(model_norm_residuals),
        pd.Series(model_norm_residuals_abs_sqrt),
        model_abs_resid,
        pd.Series(model_leverage),
        pd.Series(model_cooks),
    ],
    axis=1,
)
```

```
diags.columns = [
    "fitted_y",
    "residuals",
    "normed_residuals",
    "abs_sqrt_normed_residuals",
    "abs_residuals",
    "leverage",
    "cooks",
]
```

We also need to calculate the theoretical quantiles for plotting our quantile-quantile graph. In order to do this we first make a `ProbPlot` object from `statsmodels.api` (the same package we used to make our regression model), using the `model_norm_residuals`. From this, we can extract the theoretical quantiles and add them to our `DataFrame`.

```
def make_qq(dd, x):

    QQ = sm.ProbPlot(model_norm_residuals)

    dd = dd.sort_values(x)
    dd["qq"] = QQ.theoretical_quantiles
    return dd


diags = make_qq(diags, "normed_residuals")
```

11.11.1 Residual vs fitted plot

The first plot we'll create is the residual vs fitted plot. The first thing we need to do is find the top 3 outliers for this chart: in this case, the absolute residuals.

```
outliers_1 = diags["abs_residuals"].isin(sorted(model_abs_resid,
↪  reverse=True)[:3])
```

This will be used to label the index of these points on the charts, just as you'll find in the R regression diagnostic plots. Let's have a look at what slice of the data these points represent.

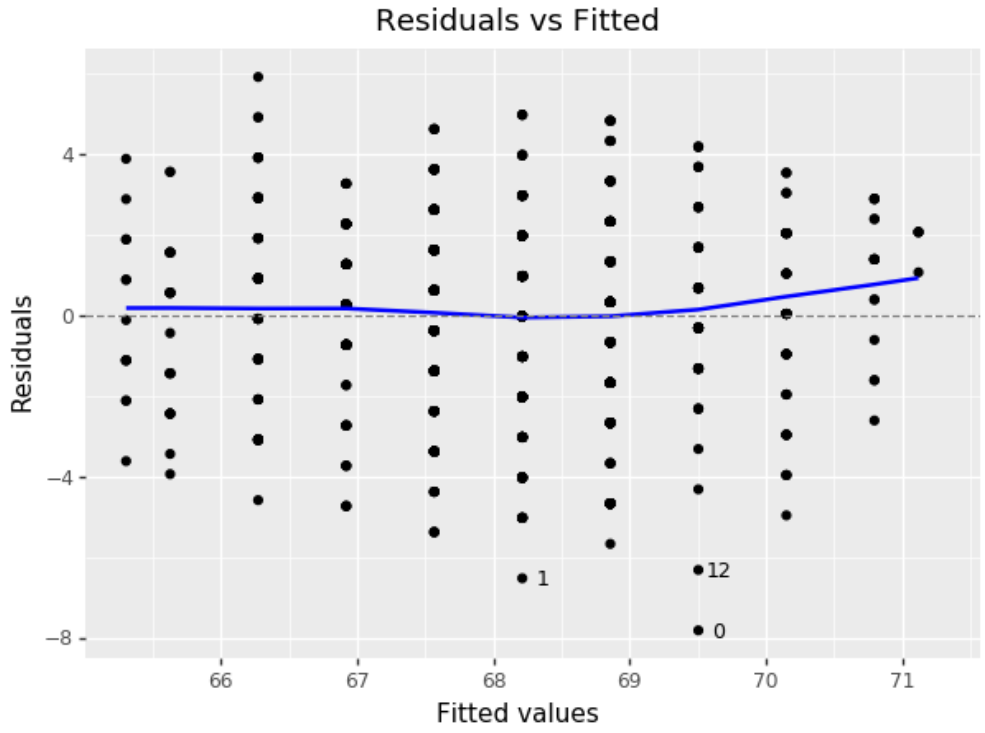
```
diags.loc[outliers_1, ["abs_residuals", "fitted_y", "residuals"]]
```

We can now create our plot. As you can see, we put the fitted values (`fitted_y`) variable on the x-axis and the residuals (`residuals`) on the y-axis. We then add

the points (`geom_point()`) and a lowess line (`stat_smooth(method="lowess")`) to the plot. We also add the reference line at y = 0 (`geom_hline(yintercept=0)`). Using the `outlier_1` variable we created above, we label the top 3 outliers by adding them to `geom_text(aes(label=np.where(outliers_1, outliers_1.index, ""))`. This means that when `outliers_1` is equal to `True`, the index of that point will be displayed, and otherwise that nothing will be displayed.

Finally, we finish off the plot by changing the x- and y-axis titles, adding a plot title and adjusting the scale on the x-axis ticks.

```
r1 = (
    ggplot(diags, aes("fitted_y", "residuals"))
    + geom_point()
    + stat_smooth(method="lowess", colour="blue")
    + geom_hline(yintercept=0, size=0.5, colour="grey",
                 linetype="dashed")
    + geom_text(
        aes(label=np.where(outliers_1, outliers_1.index, "")),
        size=9, nudge_x=0.15
    )
    + xlab("Fitted values")
    + ylab("Residuals")
    + labs(title="Residuals vs Fitted")
    + scale_x_continuous(breaks=np.arange(66, 72, 1))
)
r1
```



We now have a pretty close representation of the Residual vs Fitted plot produced by `ggfortify` in R. If you're comparing these charts between those created by R, you'll notice that the labels on the outliers differ by 1. This is because Python indexes from 0, rather than 1. In order to completely recreate the R plots, you can simply add 1 to the index label, like so: `outliers_1.index + 1`.

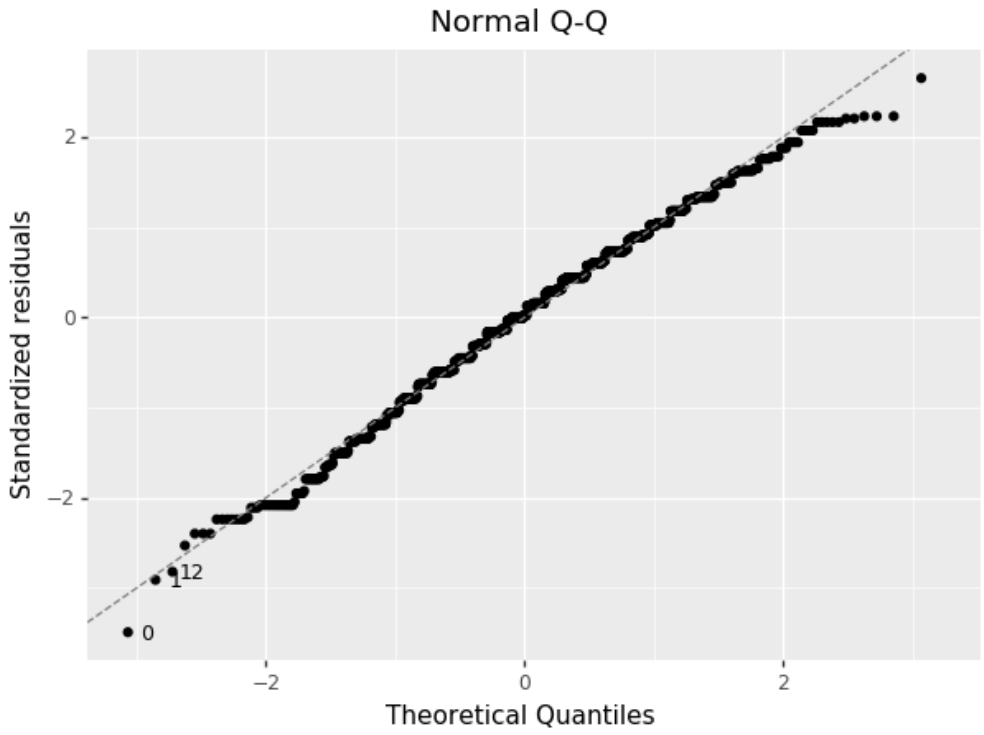
11.11.2 Normal Q-Q plot

Next, we create out Q-Q plot. Again, we find the top 3 outliers: this time for the normed residuals, and save them to `outliers_2`.

Next, we create a `ggplot` of the theoretical quantiles (`qq`) against the normed residuals (`normed_residuals`). We add the data as points (`geom_point`) and create the reference line using `geom_abline(intercept=0, slope=1)`. Again, we add the outliers using `geom_text()`, and finally, adjust the titles.

```
outliers_2 = abs(diags["normed_residuals"]).isin(
    sorted(np.abs(model_norm_residuals), reverse=True)[:3]
)
```

```
r2 = (
    ggplot(diags, aes("qq", "normed_residuals"))
    + geom_point()
    + geom_abline(intercept=0, slope=1, size=0.5,
                  colour="grey", linetype="dashed")
    + geom_text(
        aes(label=np.where(outliers_2, outliers_2.index, "")),
        size=9, nudge_x=0.15
    )
    + xlab("Theoretical Quantiles")
    + ylab("Standardized residuals")
    + labs(title="Normal Q-Q")
)
r2
```

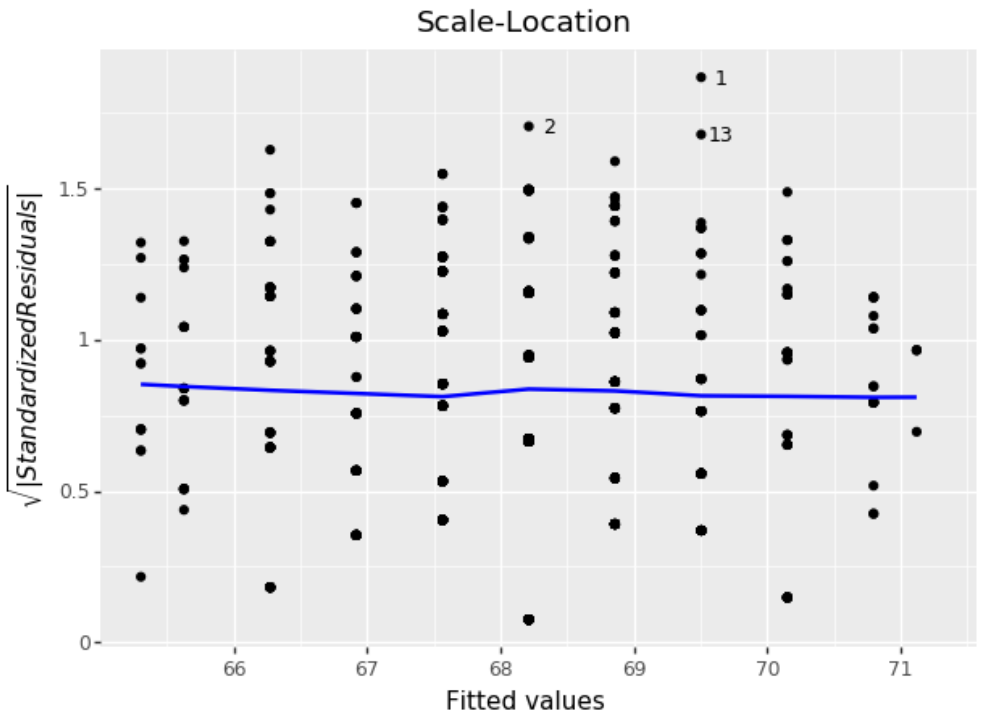


11.11.3 Scale-Location plot

A similar process is used for the Scale-Location plot. We simply plot the fitted values (`fitted_y`) against the absolute square root of the normed residuals (`abs_sqrt_normed_residuals`), add both the points and a lowess line, label the

top 3 outliers in the form of `abs_sqrt_normed_residuals`, and adjust the titles and x-axis ticks.

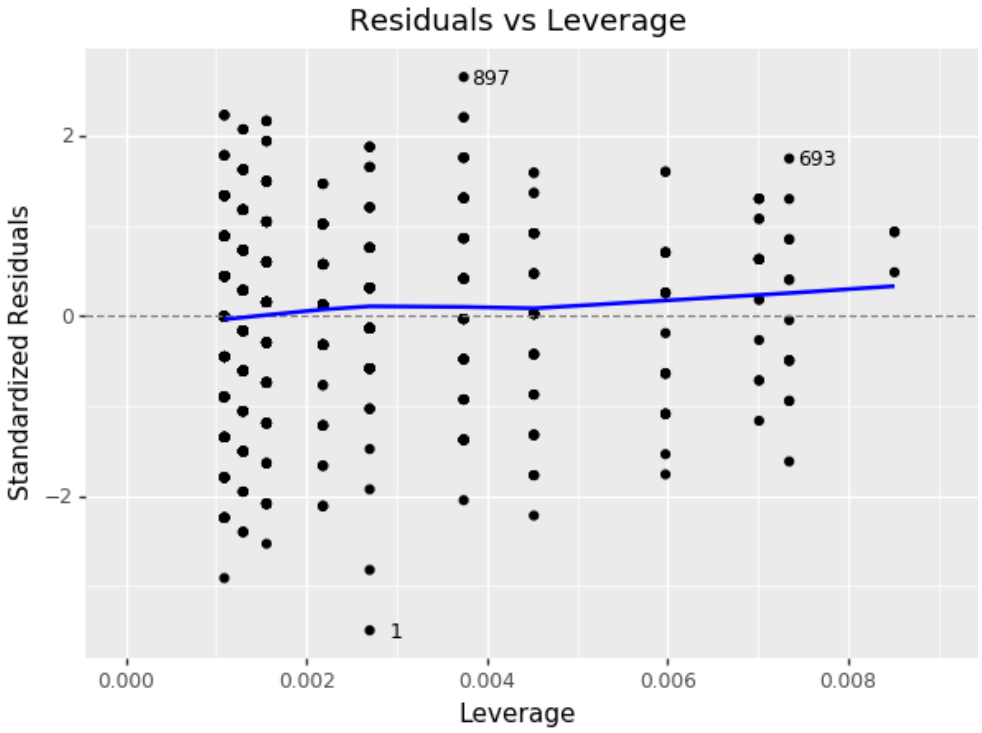
```
outliers_3 = diags["abs_sqrt_normed_residuals"].isin(
    sorted(model_norm_residuals_abs_sqrt, reverse=True)[:3]
)
r3 = (
    ggplot(diags, aes("fitted_y", "abs_sqrt_normed_residuals"))
    + geom_point()
    + stat_smooth(method="lowess", colour="blue")
    + geom_text(
        aes(label=np.where(outliers_3, outliers_3.index + 1, "")),
        size=9, nudge_x=0.15
    )
    + xlab("Fitted values")
    + ylab("$\sqrt{|Standardized Residuals|}$")
    + labs(title="Scale-Location")
    + scale_x_continuous(breaks=np.arange(66, 72, 1))
)
r3
```



11.11.4 Residuals vs Leverage plot

Finally, we do the same as with the Scale-Location plot for the Residuals vs Leverage plot. However, this time, we calculate the outliers as those values with the highest Cook's distance, rather than something that exists on the plot. We've also manually input the x-axis labels as a list under the `labels` argument in `scale_x_continuous`, in order to make sure that all numbers are displayed to 3 decimal places.

```
outliers_4 = diags["cooks"].isin(sorted(model_cooks, reverse=True)[:3])

r4 = (
    ggplot(diags, aes("leverage", "normed_residuals"))
    + geom_point()
    + geom_hline(yintercept=0, size=0.5, colour="grey",
                 linetype="dashed")
    + stat_smooth(method="lowess", colour="blue")
    + geom_text(
        aes(label=np.where(outliers_4, outliers_4.index + 1, "")),
        size=9,
        nudge_x=0.0003,
    )
    + xlab("Leverage")
    + ylab("Standardized Residuals")
    + labs(title="Residuals vs Leverage")
    + scale_x_continuous(
        breaks=np.arange(0.000, 0.009, 0.002),
        limits=[0.000, 0.009],
        labels=["0.000", "0.002", "0.004", "0.006", "0.008"],
    )
)
r4
```

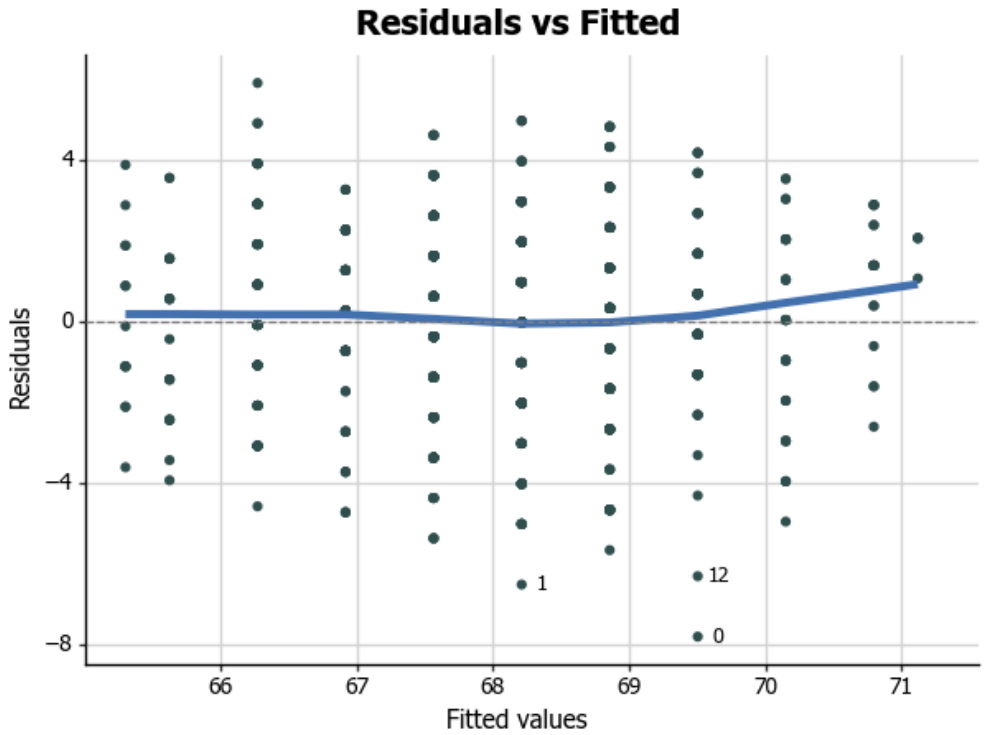


11.12 Customising the residual plots

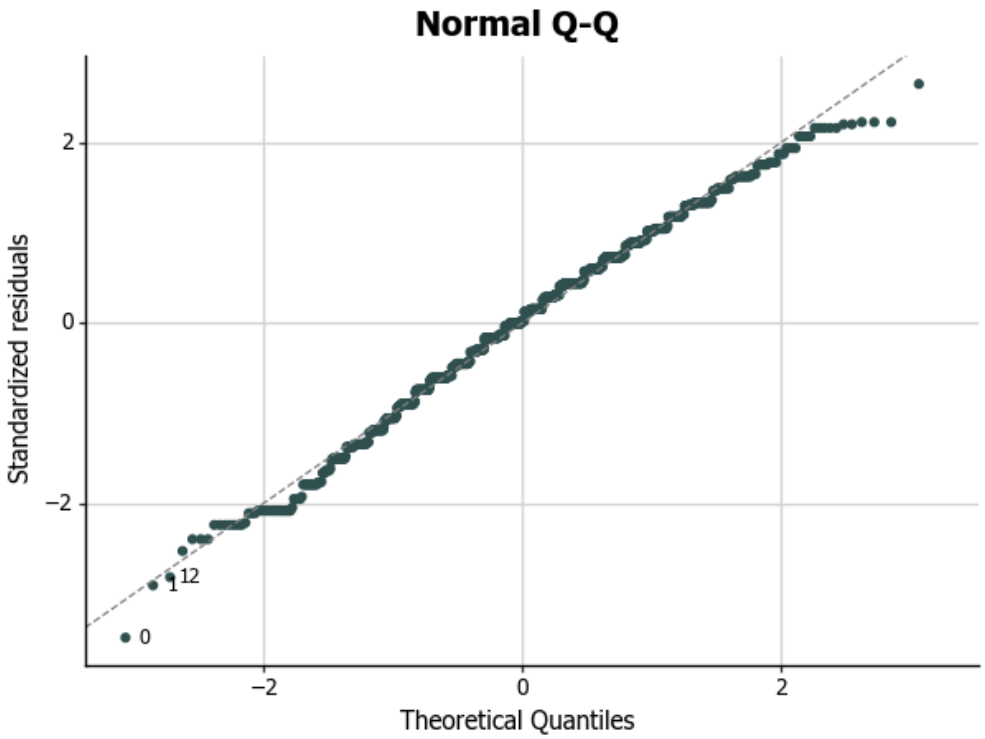
The same customisations that we added to our regression plot can be added to all of the diagnostic plots. This makes this `plotnine` implementation of regression plots extremely flexible. To give a sense of how many options you have, we will reproduce each of the diagnostic plots we created above using the custom theme we designed for our regression plot. Of course, you could change the appearance of these plots to any of the themes we've discussed in this chapter so far, or to your own custom theme.

```
outliers_1 = diags["abs_residuals"].isin(sorted(model_abs_resid,
↪  reverse=True)[:3])

r1 = (
    ggplot(diags, aes("fitted_y", "residuals"))
    + geom_point(colour="darkslategrey")
    + stat_smooth(method="lowess", colour="#4271AE", size=2)
    + geom_hline(yintercept=0, size=0.5, colour="grey",
                 linetype="dashed")
    + geom_text(
```

```
        aes(label=np.where(outliers_1, outliers_1.index, "")),
        size=9, nudge_x=0.15
    )
    + xlab("Fitted values")
    + ylab("Residuals")
    + labs(title="Residuals vs Fitted")
    + scale_x_continuous(breaks=np.arange(66, 72, 1))
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
r1
```

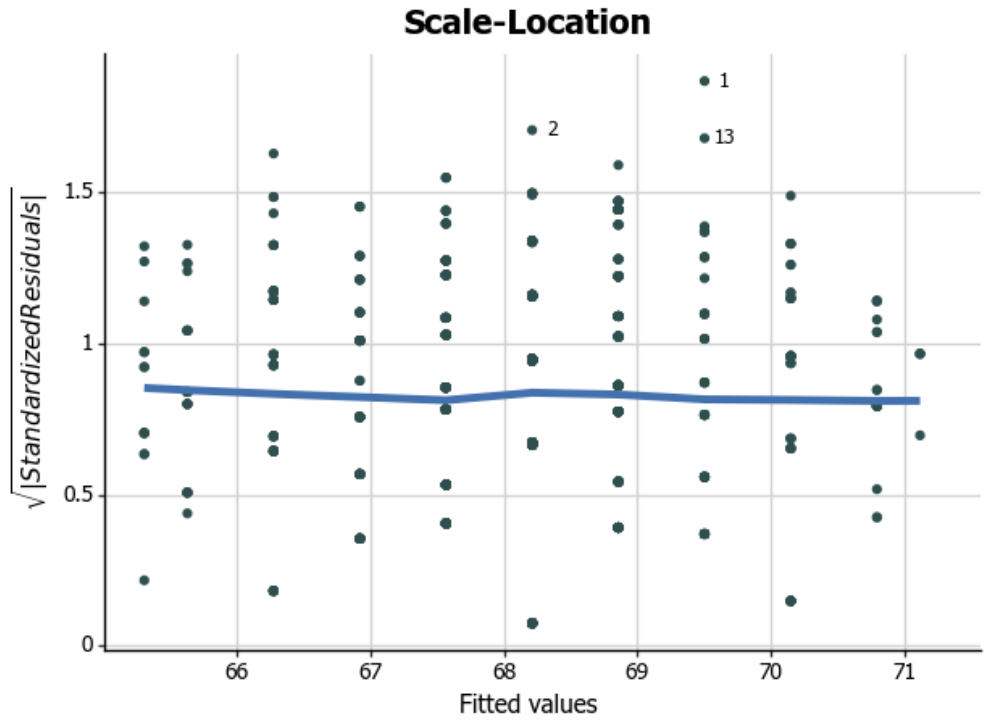


```
outliers_2 = abs(diags["normed_residuals"]).isin(
    sorted(np.abs(model_norm_residuals), reverse=True)[:3]
)
r2 = (
    ggplot(diags, aes("qq", "normed_residuals"))
    + geom_point(colour="darkslategrey")
    + geom_abline(intercept=0, slope=1, size=0.5, colour="grey",
                  linetype="dashed")
    + geom_text(
        aes(label=np.where(outliers_2, outliers_2.index, "")),
        size=9, nudge_x=0.15
    )
    + xlab("Theoretical Quantiles")
    + ylab("Standardized residuals")
    + labs(title="Normal Q-Q")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
r2
```



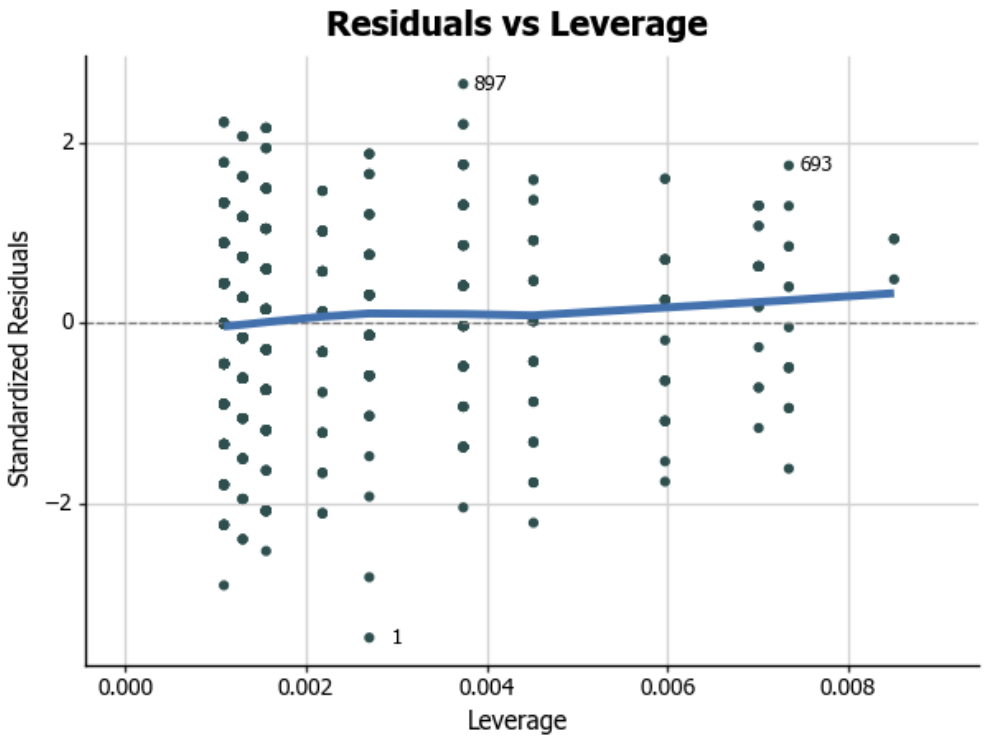

```
outliers_3 = diags["abs_sqrt_normed_residuals"].isin(
    sorted(model_norm_residuals_abs_sqrt, reverse=True)[:3]
)
r3 = (
    ggplot(diags, aes("fitted_y", "abs_sqrt_normed_residuals"))
    + geom_point(colour="darkslategrey")
    + stat_smooth(method="lowess", colour="#4271AE", size=2)
    + geom_text(
        aes(label=np.where(outliers_3, outliers_3.index + 1, "")),
        size=9, nudge_x=0.15
    )
    + xlab("Fitted values")
    + ylab("$\sqrt{|Standardized Residuals|}$")
    + labs(title="Scale-Location")
    + scale_x_continuous(breaks=np.arange(66, 72, 1))
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
```

```
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
r3
```




```
outliers_4 = diags["cooks"].isin(sorted(model_cooks, reverse=True)[:3])

r4 = (
    ggplot(diags, aes("leverage", "normed_residuals"))
    + geom_point(colour="darkslategrey")
    + geom_hline(yintercept=0, size=0.5, colour="grey",
                 linetype="dashed")
    + stat_smooth(method="lowess", colour="#4271AE", size=2)
    + geom_text(
        aes(label=np.where(outliers_4, outliers_4.index + 1, "")),
        size=9,
        nudge_x=0.0003,
    )
    + xlab("Leverage")
    + ylab("Standardized Residuals")
    + labs(title="Residuals vs Leverage")
    + scale_x_continuous(
        breaks=np.arange(0.000, 0.009, 0.002),
```

```
        limits=[0.000, 0.009],
        labels=["0.000", "0.002", "0.004", "0.006", "0.008"],
    )
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
r4
```



11.13 Combining the plots into a single plot

In order to completely reproduce the charts output by R we need to combine the diagnostic plots into a single chart. Unfortunately, as the implementation of combining charts in `matplotlib` is a bit clumsy the makers of `plotnine` have not yet created a way of doing this. A workaround is to save our charts to file as images, and then combine them using a `matplotlib` package called `gridspec`.

The first thing to do is save all of our plots to png files. We'll save them at a resolution of 400 DPI as this seems to give the clearest images once they're positioned in the chart.

```
p11.save("main_plot.png", dpi=400)
r1.save(filename="diag1.png", dpi=400)
r2.save(filename="diag2.png", dpi=400)
r3.save(filename="diag3.png", dpi=400)
r4.save(filename="diag4.png", dpi=400)
```

We now need to place each of our charts on our combined plot. In order to do this, we create new axes subplot objects for each of the individual charts (axes objects in `matplotlib` are simply objects that have plotting methods). For each of these axes objects, we want to read in and display our chart images, and then get rid of all of the axis attributes (the axis lines, and the tick marks and labels). As we'll be repeating the same thing for each chart, let's first create a function that does all of these things. This function takes 3 arguments: the position of the chart and the name of the chart image file.

```
def placeSubplots(position, file):
    ax_i = plt.subplot(position)
    ax_i.imshow(mpimg.imread(file), aspect="equal")
    return plt.axis("off")
```

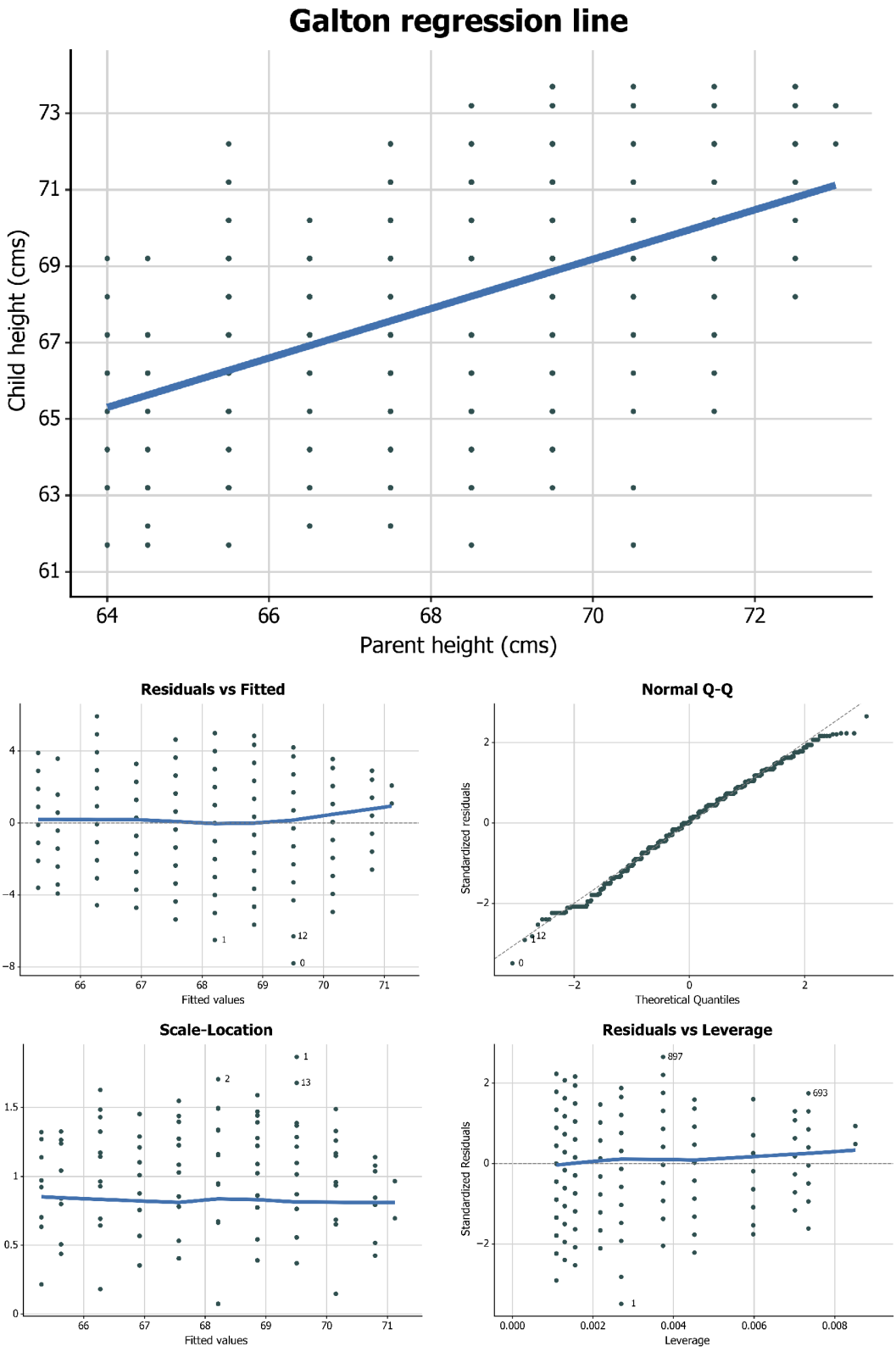
We then create a `matplotlib` figure of the size 5.75 x 8.5 inches, with a resolution of 400 DPI. We also overlay a `GridSpec` on top of the figure, which has 8 spots for us to position our charts (4×2). Finally, we'll reduce the space between all of the charts to give a nice tight positioning using `gs1.update(wspace=0.01, hspace=0.01)`.

We can now call our `placeSubplots` function to add all of the images to that chart. The position argument works like a coordinate system, with the ability to have ranges of the chart. So for the top chart, we want it to start at (0,0) and end at (2,2); hence, we set the argument as [0:2, 0:2]. For the others, we implicitly state that we only want them taking up one cell on the grid, so by setting

the argument to (for example) [2, 0] for the second subplot, we're saying we want it to start at (2,0) and end at (3,1).

With this final step of combining our charts, we now have the customised version of the regression and diagnostic plots that we saw at the beginning of this chapter.

```
fig = plt.figure(figsize=(5.75, 8.5), dpi=400)
gs1 = gridspec.GridSpec(4, 2)
gs1.update(wspace=0.01, hspace=0.01)

placeSubplots(gs1[0:2, 0:2], "main_plot.png")
placeSubplots(gs1[2, 0], "diag1.png")
placeSubplots(gs1[2, 1], "diag2.png")
placeSubplots(gs1[3, 0], "diag3.png")
placeSubplots(gs1[3, 1], "diag4.png")
plt.show()
```

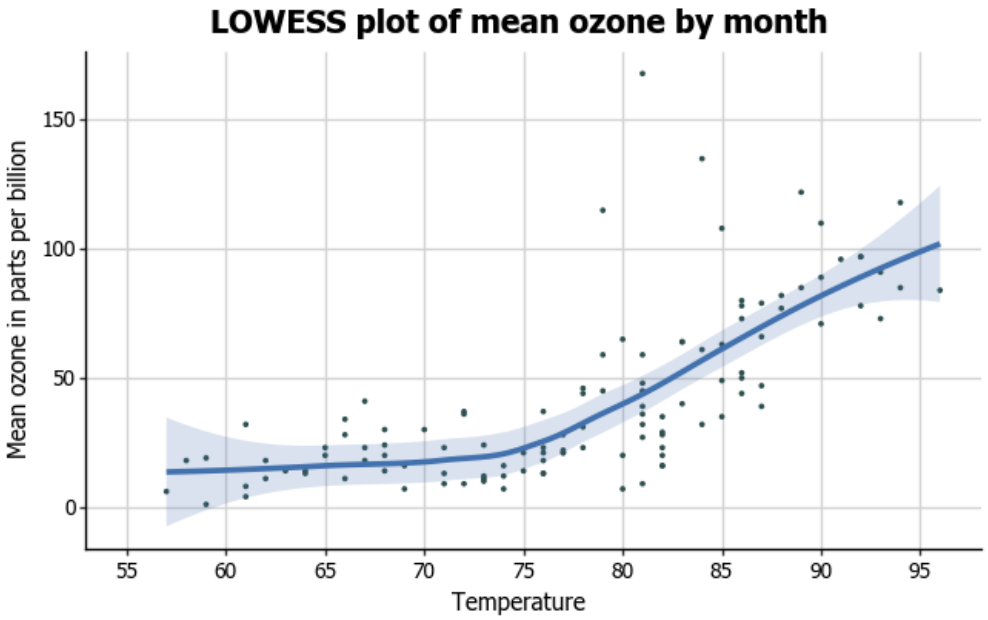

Galton regression line
Child height (cms)
Parent height (cms)
73
71
69
67
65
63
61
64
66
68
70
72
Residuals vs Fitted
Residuals
Fitted values
Normal Q-Q
Standardized residuals
Theoretical Quantiles
Scale-Location
√|StandardizedResiduals|
Fitted values
Residuals vs Leverage
Standardized Residuals
Leverage
897
693
1

Chapter 12

LOWESS plots

12.1 Introduction

This is the twelfth and final chapter. In this chapter, we will work towards creating the LOWESS plot below. We will take you from a basic LOWESS plot and explain all the customisations we add to the code step-by-step.



The first step is to import all of the required packages. For this we need:

- `pandas` and its `DataFrame` class to read in and manipulate our data;
- `plotnine` to get our data and create our graphs; and
- `numpy` to do some basic numeric calculations in our graphing.

We can also change the size of the plots using the `figure_size` function from `plotnine`. We have resized the plots in this chapter so they display a little more neatly.

```
import numpy as np
import pandas as pd

import plotnine
plotnine.options.figure_size=(7.5, 4.2)

from plotnine import *
from pandas import DataFrame
```

In this chapter, we'll be using the R dataset `airquality`. Details on this dataset are here[1]. As this is an R dataset and is not included with the `plotnine` default datasets, we need to download it.

```
airquality = pd.read_csv(
    "https://bit.ly/1r8B1wN", header=0,
    usecols=np.arange(1, 7, 1),
)
```

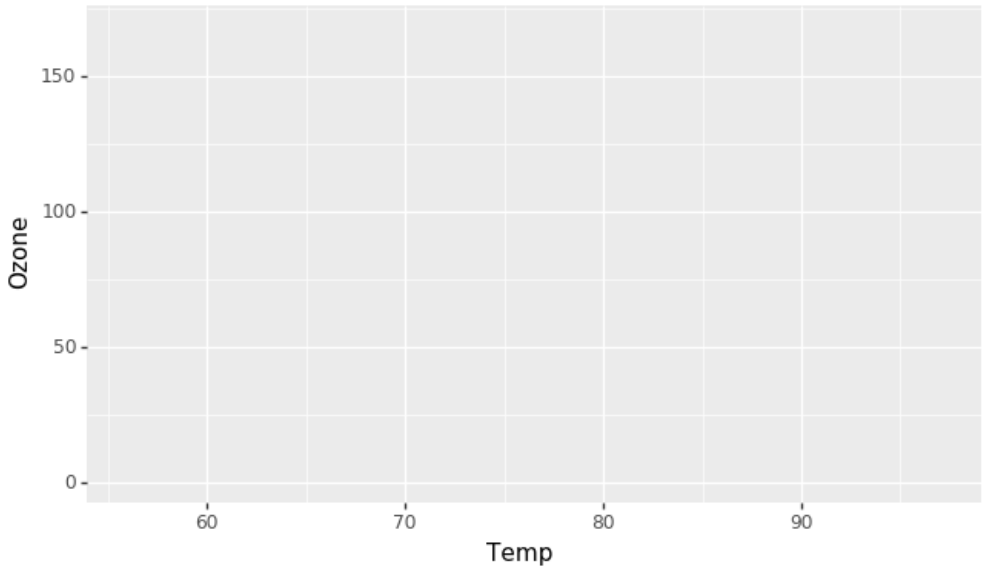
12.2 Basic ggplot structure

In order to initialise a LOWESS plot we tell ggplot that `airquality` is our data, and specify that our x-axis plots the `Temp` variable and our y-axis plots the `Ozone` variable. You may have noticed that we put our x- and y-variables inside a method called `aes`. This is short for aesthetic mappings, and determines how the different variables you want to use will be mapped to parts of the graph. As you can see below, `ggplot` has mapped `Temp` to the x-axis and `Ozone` to the y-axis.

You might have also noticed that there is nothing in the plot. In order to render our data, we need to tell ggplot how we want to visually represent it.

[1] `https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/airquality.html`

```
p12 = ggplot(airquality, aes("Temp", "Ozone"))
p12
```

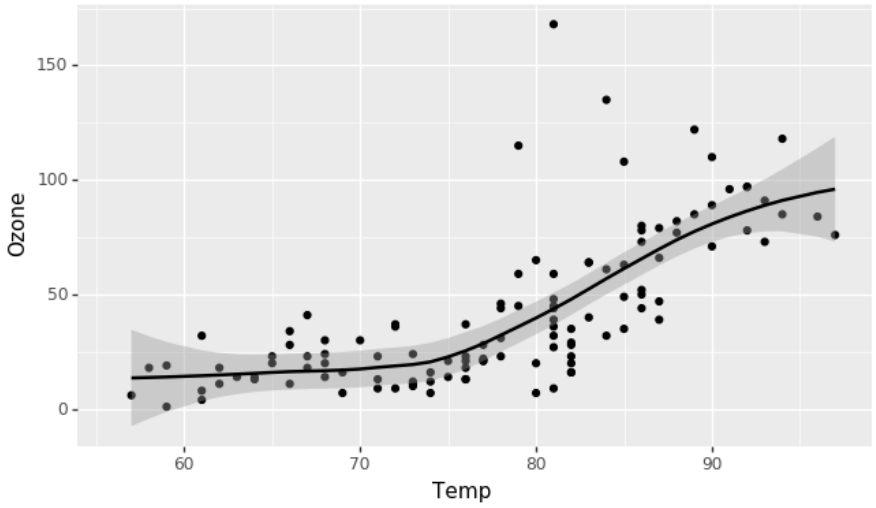


12.3 Creating a basic LOWESS plot, and what it can tell us about our data

We can add a LOWESS curve to the plot by adding the `geom_smooth(method="loess")` option. Note that the default for `stat_smooth` is to include the confidence interval.

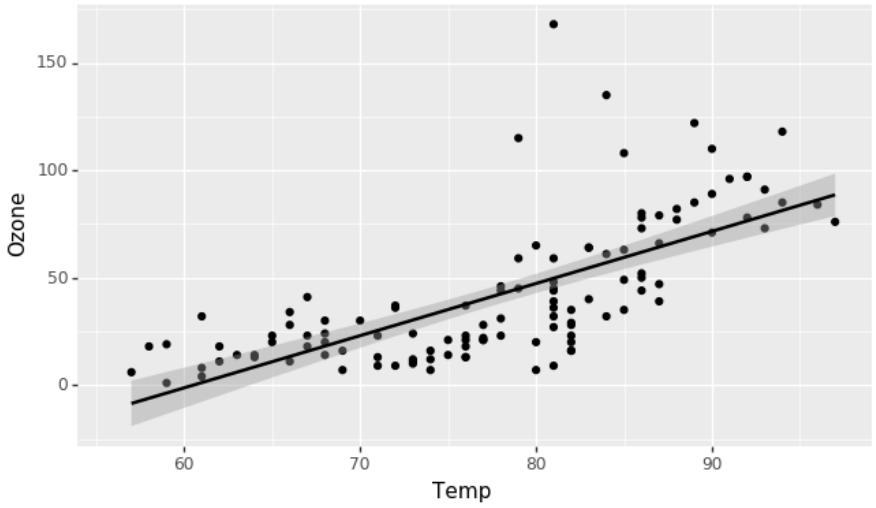
You can alternatively create a "simple" LOWESS chart (without a confidence interval) by using `method="lowess"`.

```
p12 = (ggplot(airquality, aes("Temp", "Ozone")) + geom_point()
       + geom_smooth(method="loess"))
p12
```



We can see that while the relationship between `Temp` and `Ozone` is fairly linear, the LOWESS plot is demonstrating there may be a threshold effect where ozone only starts increasing as temperatures pass around 75 degrees Fahrenheit. To assess whether this is the case, let's see how a linear fit between these variables looks.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
       + geom_point()
       + geom_smooth(method="lm")
)
p12
```



Let's now have a look at the amount of variance it explains in ozone levels by extracting the adjusted R^2 from the linear regression model between these two variables.

```
import statsmodels.formula.api as smf

model = smf.ols(formula="Ozone ~ Temp", data=airquality).fit()
model.rsquared_adj
```

You can see that the line comes away from the data at several points, which will have increased the error in the regression model and brought down the overall R^2. Let's see whether we can get a better result by fitting a quadratic model, which better reflects the curved line that the LOWESS plot gives.

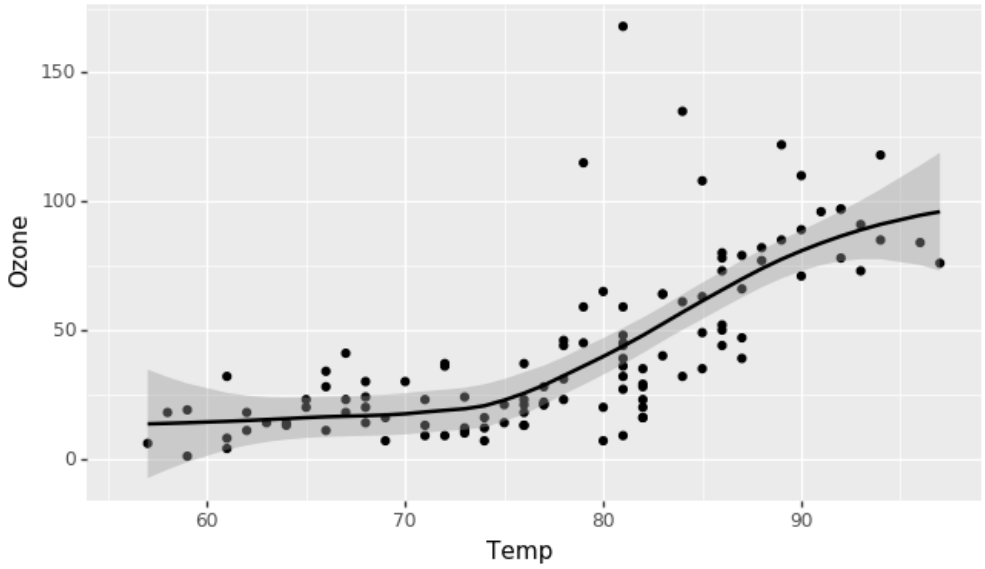
```
model = smf.ols(formula="Ozone ~ I(Temp**2) + Temp", data=airquality).fit()
model.rsquared_adj
```

You can see that we've managed to explain an additional 5% of variance in ozone levels by fitting a quadratic model rather than defaulting to a linear model. Using LOWESS plots to explore the relationships between your variables can therefore guide you in choosing the the right regression model in a fairly pain-free way.

12.4 Changing the width of the bins

An important part of fitting LOWESS curves is that you can change the number of bins that the x-axis is divided into by using the argument `n`. More bins smooth out the line more, while less make it closer to linear. The default number is 80, and here we will change it to 5 so you can see the difference.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(method="loess", n=5)
)
p12
```



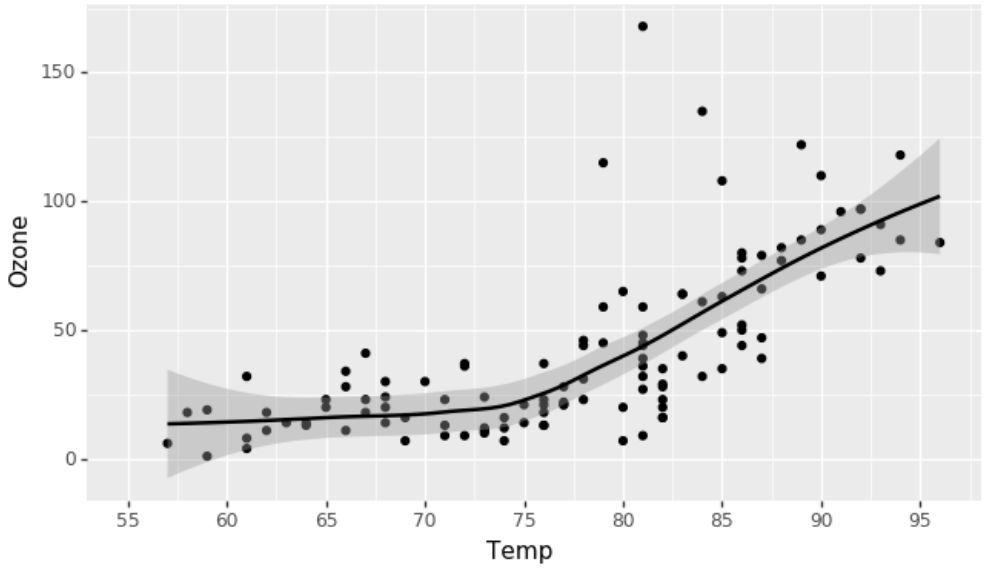
12.5 Adjusting the axis scales

To change the x-axis tick marks, we use the `scale_x_continuous` option. Similarly, to change the y-axis we use the `scale_y_continuous` option. Here we will change the x-axis to every 5 degrees, rather than 10, and change the range from 55 to 95. We can change the breaks using the `breaks` option, which takes a list of values as an argument. You can shortcut having to type in the whole list manually using numpy's `arange` function[2] which generates a sequence from your selected start, stop and step values respectively. Similarly, you can use the `limits` argument to define the minimum and maximum values of your axis.

We'll also go back to the diamond shape going forward.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(method="loess")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
)
p12
```

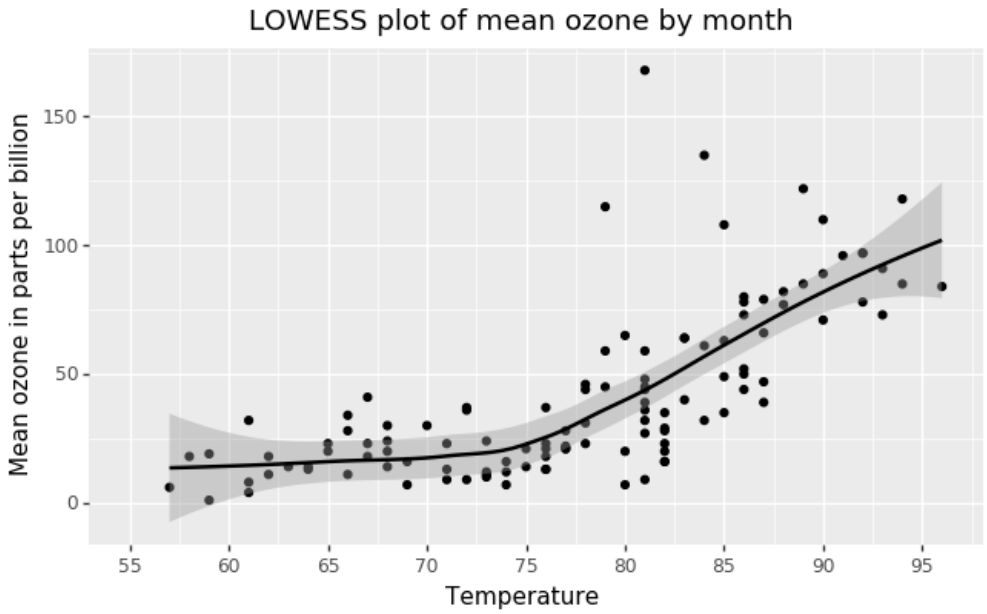
[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html



12.6 Adjusting axis labels & adding title

To add a title, we include the option `ggtitle` and include the name of the graph as a string argument. To change the axis names we similarly use the `xlab` and `ylab` arguments.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(method="loess")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```

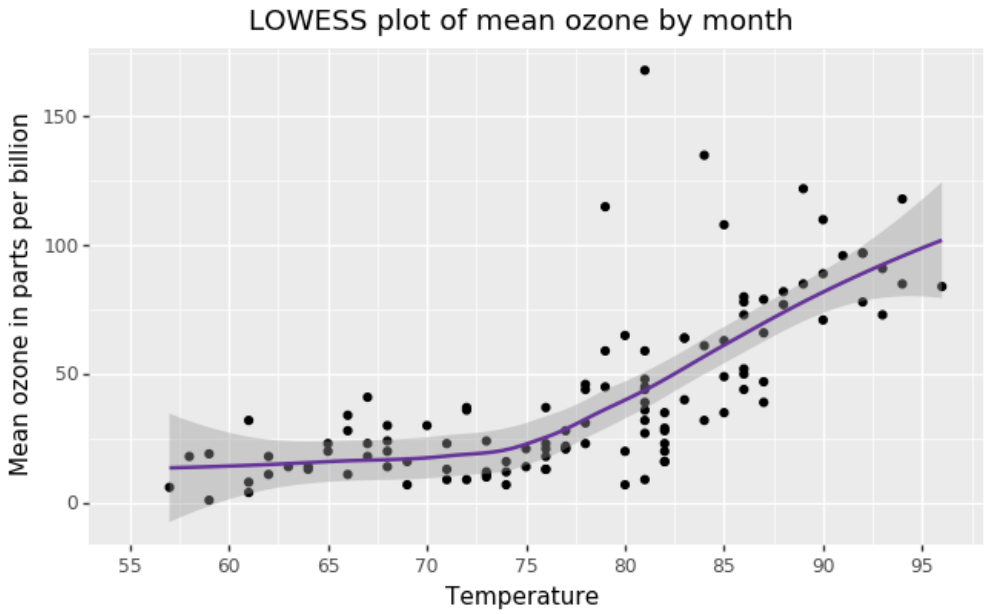


12.7 Changing the colour and size of the LOWESS curve

To change the colour of the LOWESS curve, we add a valid colour to the colour argument in `geom_smooth()`. `plotnine` uses the colour palette utilised by `matplotlib`. The full set of named colours recognised by `ggplot` is here[3]. Let's try changing our line to `rebeccapurple`.

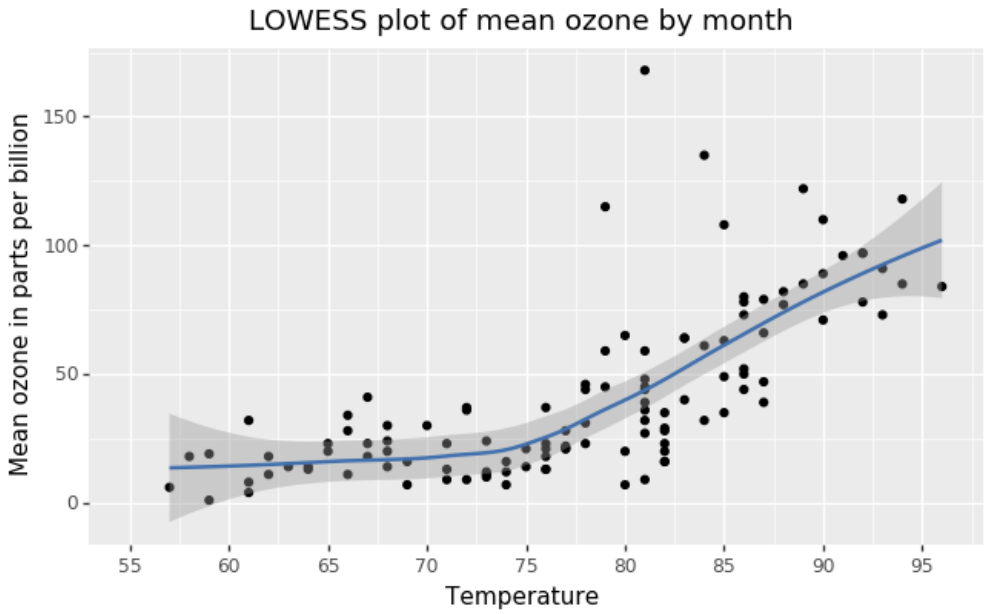
```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(method="loess", colour="rebeccapurple")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```

[3] https://matplotlib.org/examples/color/named_colors.html



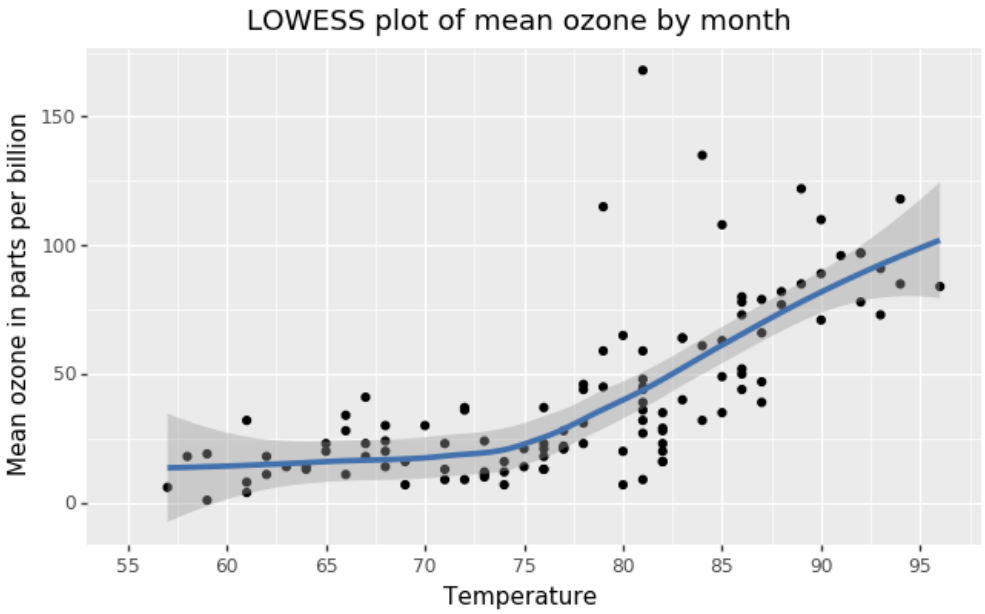
You can change the colours using specific HEX codes instead. Here we have made the line `"#4271AE"` (steel blue).

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(method="loess", colour="#4271AE")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```



We can also increase the thickness of the line using the size option in `geom_smooth()`.

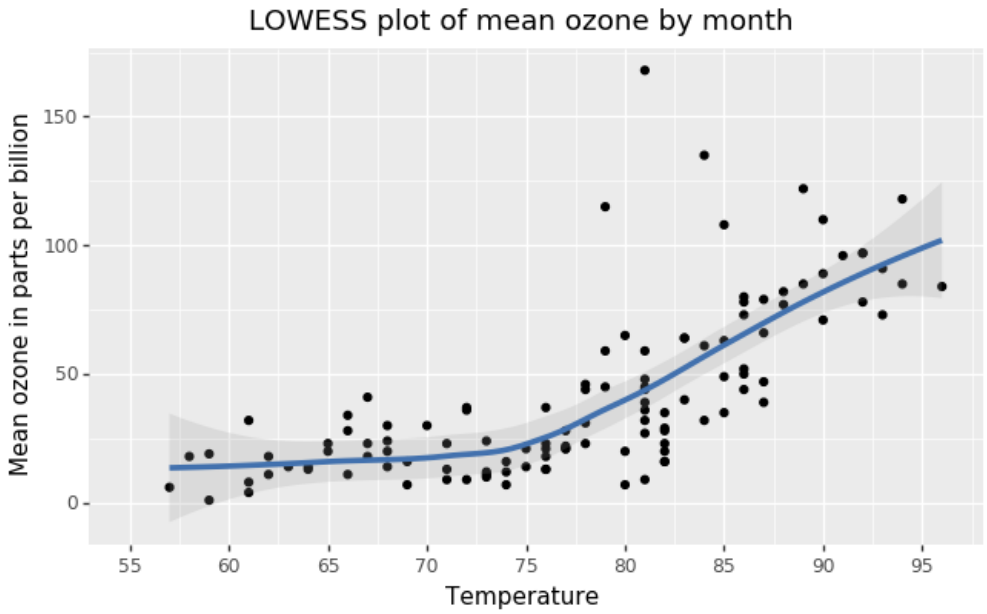
```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(method="loess", colour="#4271AE", size=1.5)
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```



12.8 Changing the appearance of the confidence interval

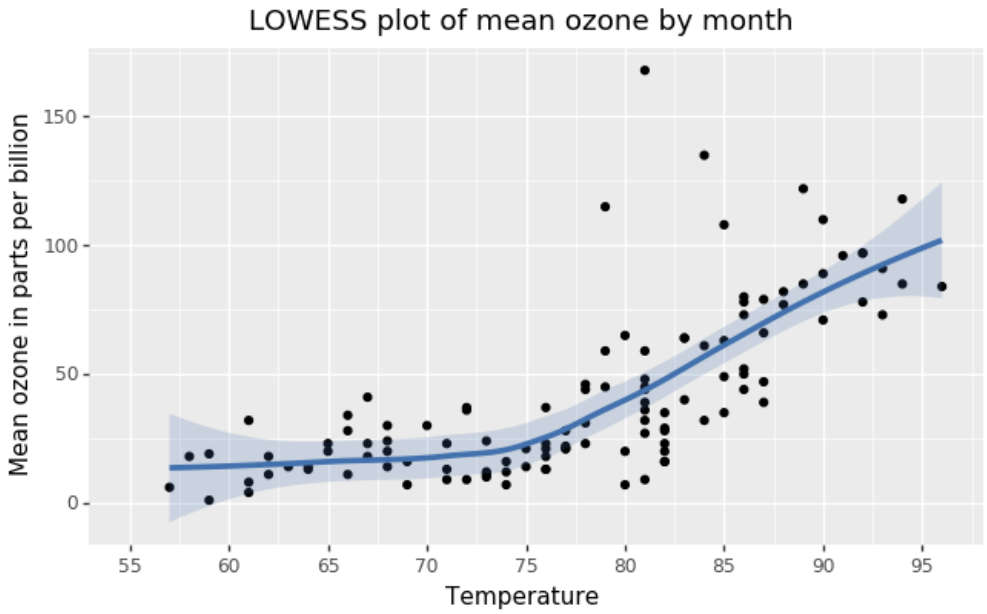
Additionally, we can alter how the confidence interval around the LOWESS curve looks. We can change the transparency using the argument `alpha` in `geom_smooth()`. This ranges from 0 to 1. Here we will increase the transparency of the confidence interval.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(method="loess", colour="#4271AE", size=1.5,
                  alpha=0.2)
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```



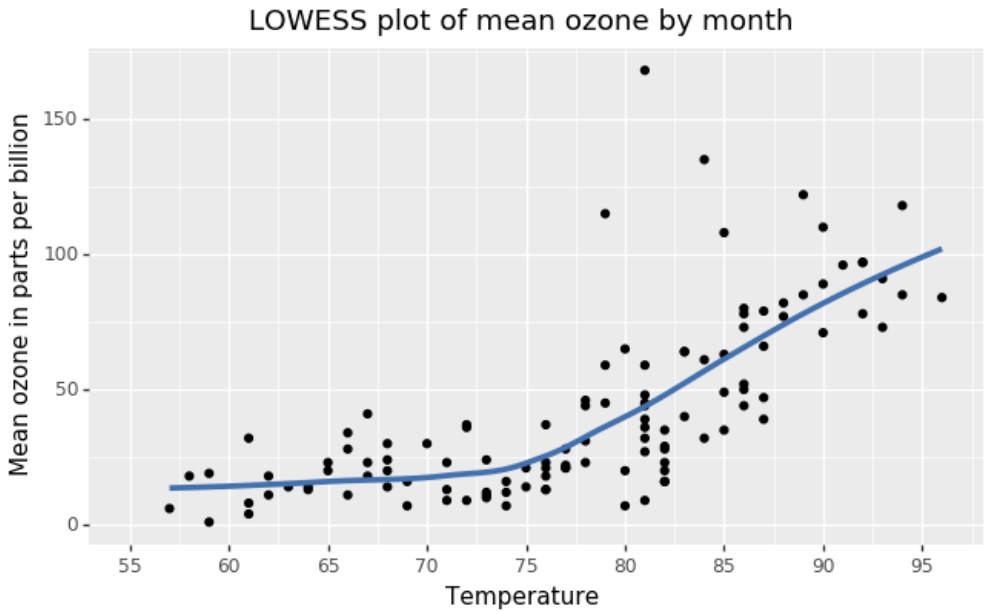
We can also change the colour of the confidence interval from the default grey using the argument `fill`, also within `geom_smooth()`. Let's change it to the same blue as our LOWESS curve.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(method="loess", colour="#4271AE",
                  size=1.5, alpha=0.2, fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```



Finally, you can also turn off the confidence altogether by adding the argument `se=False` to `geom_smooth()`.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point()
    + geom_smooth(
        method="loess", colour="#4271AE", size=1.5,
        alpha=0.2, fill="#4271AE", se=False
    )
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```

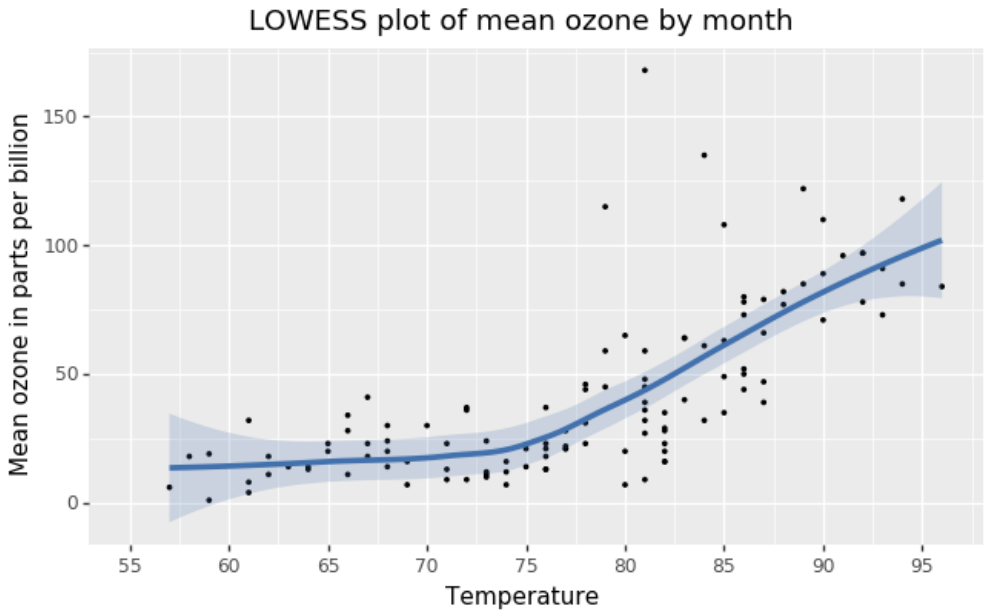


12.9 Changing the appearance of the scatterplot

Of course, the LOWESS curve is not the only part of this plot. We can also customise the appearance of the scatterplot underlying the curve. Perhaps we want the data points to be a different shape than a solid circle. We can change these by adding the `shape` argument to `geom_point`. The shape arguments for `plotnine` are the same as those available in `matplotlib`, and are therefore a little more limited than those in R's implementation of `ggplot2`. Nonetheless, there is a good range of options. The allowed arguments are here[4]. Let's change all of the markers to small circles using the argument `"."`.

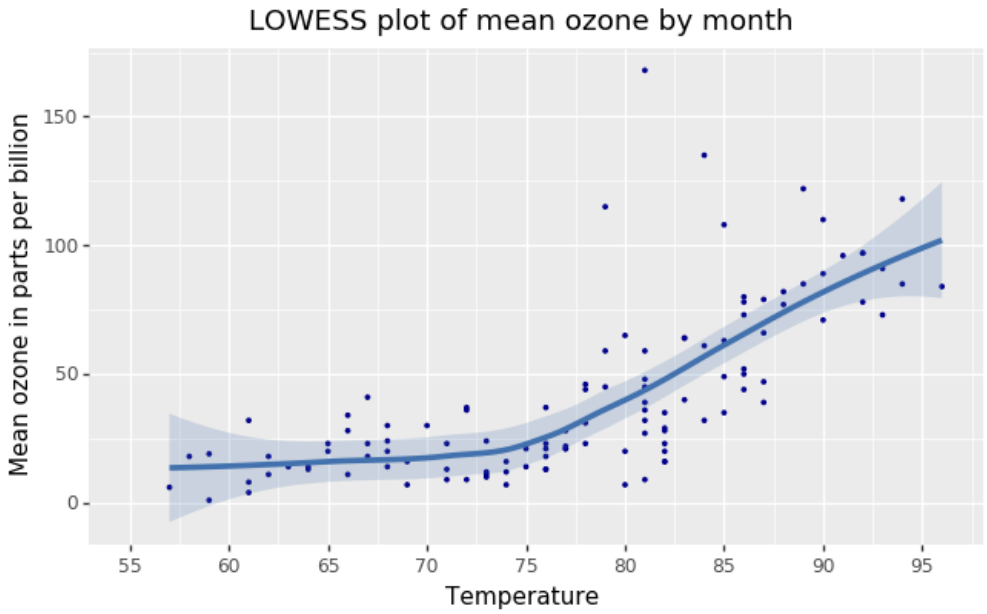
```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point(shape=".")
    + geom_smooth(method="loess", colour="#4271AE",  size=1.5, alpha=0.2,
                  fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(55, 96, 5), limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature") + ylab("Mean ozone in parts per billion")
)
p12
```

[4] https://matplotlib.org/api/markers_api.html



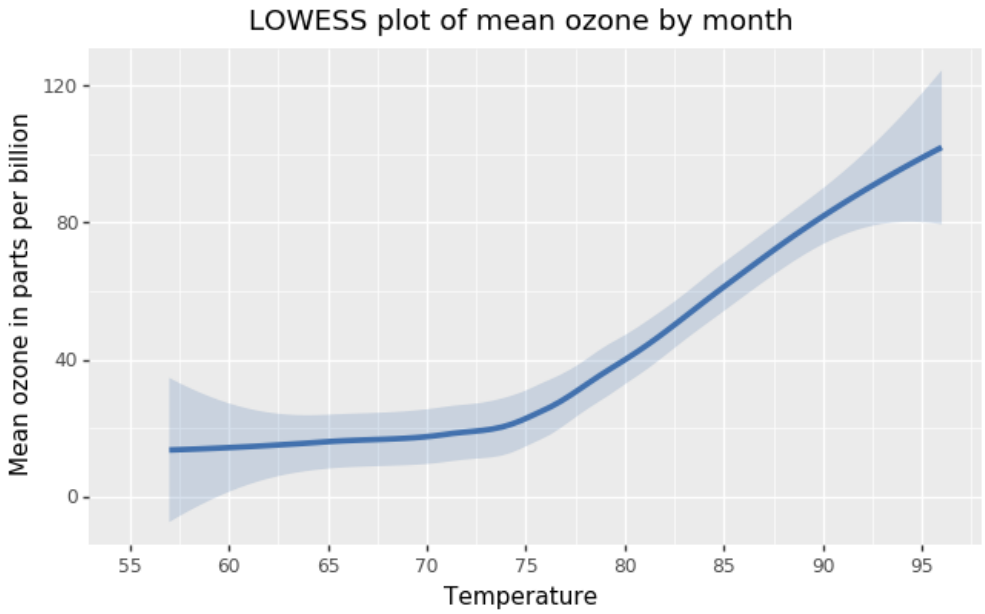
Similarly, we can also change the colours of the points by adding an `colour` argument to `geom_point()`.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point(shape=".", colour="darkblue")
    + geom_smooth(method="loess", colour="#4271AE",
                  size=1.5, alpha=0.2, fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```



You can also get rid of the points altogether by removing the `geom_point()` option.

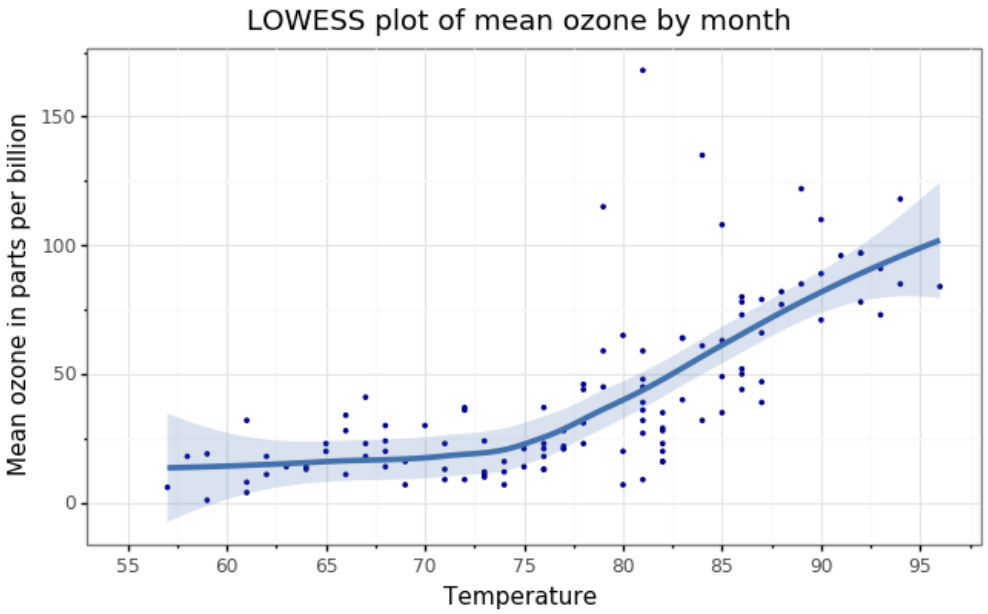
```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_smooth(method="loess", colour="#4271AE",
                  size=1.5, alpha=0.2, fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
)
p12
```



12.10 Using the white theme

As explained in the previous chapters, we can also change the overall look of the plot using themes. We'll start using a simple theme customisation by adding `theme_bw()` after `ggplot()`. As you can see, we can further tweak the graph using the `theme` option, which we've used so far to change the legend.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point(shape=".", colour="darkblue")
    + geom_smooth(method="loess", colour="#4271AE",
                  size=1.5, alpha=0.2, fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
    + theme_bw()
)
p12
```



12.11 Creating an XKCD style chart

Of course, you may want to create your own themes as well. `ggplot` allows for a very high degree of customisation, including allowing you to use imported fonts. `plotnine` already has a `theme_xkcd()` implementation, but we've instead created one from scratch to demonstrate how to use imported fonts and some of the other options in `theme` to tweak the overall look of the graph.

In order to create this chart, you first need to download the XKCD font, which Randall Munroe has kindly provided here[5]. Once you have it, you can load it into Python using the `matplotlib.font_manager` class.

```
import matplotlib.font_manager as fm

fpath = "path/to/file/xkcd-Regular.otf"
```

As this is an imported font, we can't change its size directly within the graph. Instead, we need to alter our imported font objects to change the size. As we want a different font size for the title and the body, we will create 2 different font objects, `title_text` and `body_set`.

[5] `xkcd.com/1350/xkcd-Regular.otf`

We can then call methods on these objects (the list of available methods is here[6]). For the title, we'll change the font to size 18 and make it bold using the `set_size()` and `set_weight` methods. Similarly, we'll change the body text to size 12.

```
# Create font objects
title_text = fm.FontProperties(fname=fpath)
body_text = fm.FontProperties(fname=fpath)

# Alter size and weight of font objects
title_text.set_size(18)
title_text.set_weight("bold")

body_text.set_size(12)
```

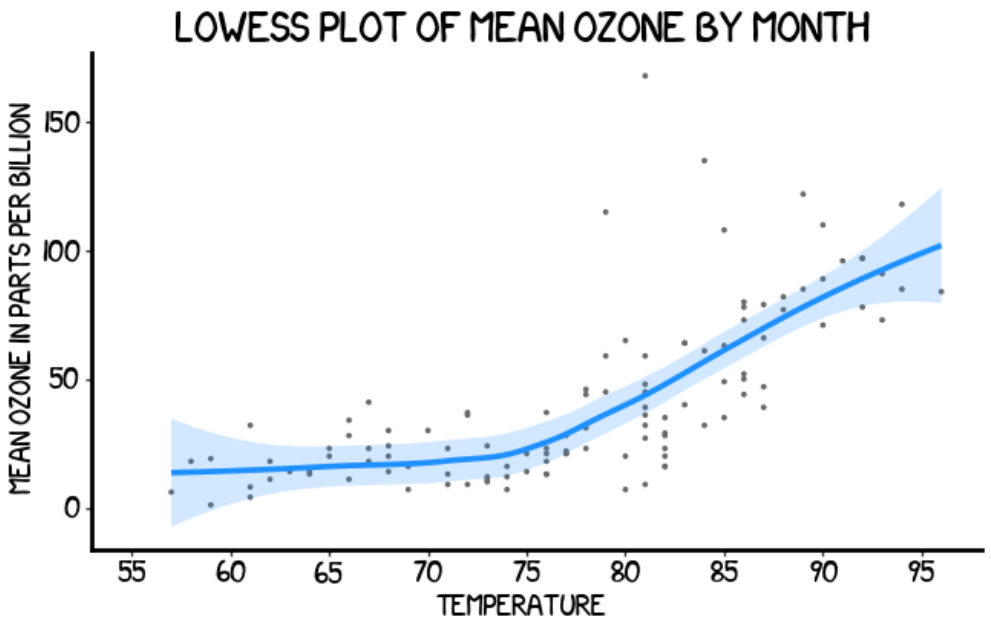
In order to get the plot to look more like the XKCD artstyle, we'll make a few more changes:

- Altering the values of `axis_line_x` and `axis_line_y` changes the thickness of the axis lines;
- Setting the argument of `legend_key` to `element_blank()` gets rid of the boxes around the legend;
- In order to get rid of the grid lines, we need to change the value of four parameters: `panel_grid_major`, `panel_grid_minor`, `panel_border` and `panel_background`;
- To use the XKCD font that we just imported, we need to change the values of both `plot_title` and `text`;
- Finally, to change the colour of the text to black (from its default grey), we change the values of `axis_text_x` and `axis_text_y`.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point(shape=".", colour="dimgrey")
    + geom_smooth(
        method="loess", colour="dodgerblue",
        size=1.5, alpha=0.2, fill="dodgerblue"
    )
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
    + theme(
        axis_line_x=element_line(size=2, colour="black"),
```

[6] https://matplotlib.org/api/font_manager_api.html

```
        axis_line_y=element_line(size=2, colour="black"),
        legend_key=element_blank(),
        panel_grid_major=element_blank(),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
        axis_text_x=element_text(colour="black"),
        axis_text_y=element_text(colour="black"),
    )
)
p12
```



12.12 Using the 'Five Thirty Eight' theme

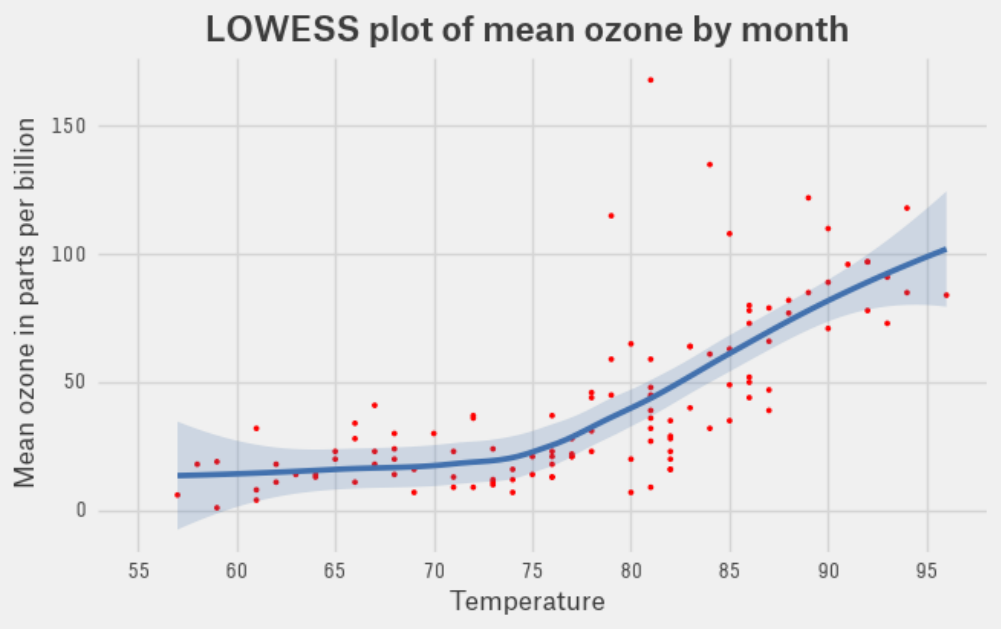
There are a wider range of pre-built themes available as part of the `ggplot` package (more information on these here[7]). Below we've applied `theme_538()`, which approximates graphs in the nice FiveThirtyEight website. As you can see, we've used the commercially available fonts 'Atlas Grotesk'[8] and 'Decima

[7] http://plotnine.readthedocs.io/en/stable/api.html#themes
[8] https://commercialtype.com/catalog/atlas/atlas_grotesk

Mono Pro'[9] in `axis_title`, `plot_title` and `text`. This is just to make the plots exactly like those on the site, and is entirely optional.

```
agm = "path/to/file/AtlasGrotesk-Medium.otf"
agr = "path/to/file/AtlasGrotesk-Regular.otf"
dp = "path/to/file/DecimaMonoPro.otf"

# Create font objects
title_text = fm.FontProperties(fname=agm)
axis_text = fm.FontProperties(fname=agr)
body_text = fm.FontProperties(fname=dp)

# Alter size and weight of font objects
title_text.set_size(16)
axis_text.set_size(12)
body_text.set_size(10)


p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point(shape=".", colour="red")
    + geom_smooth(method="loess", colour="#4271AE",
                  size=1.5, alpha=0.2, fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
                         limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
    + theme_538()
    + theme(
        axis_title=element_text(fontproperties=axis_text),
        plot_title=element_text(fontproperties=title_text),
        text=element_text(fontproperties=body_text),
    )
)
p12
```

[9] https://www.myfonts.com/fonts/tipografiaramis/decima-mono-pro/



12.13 Creating your own theme

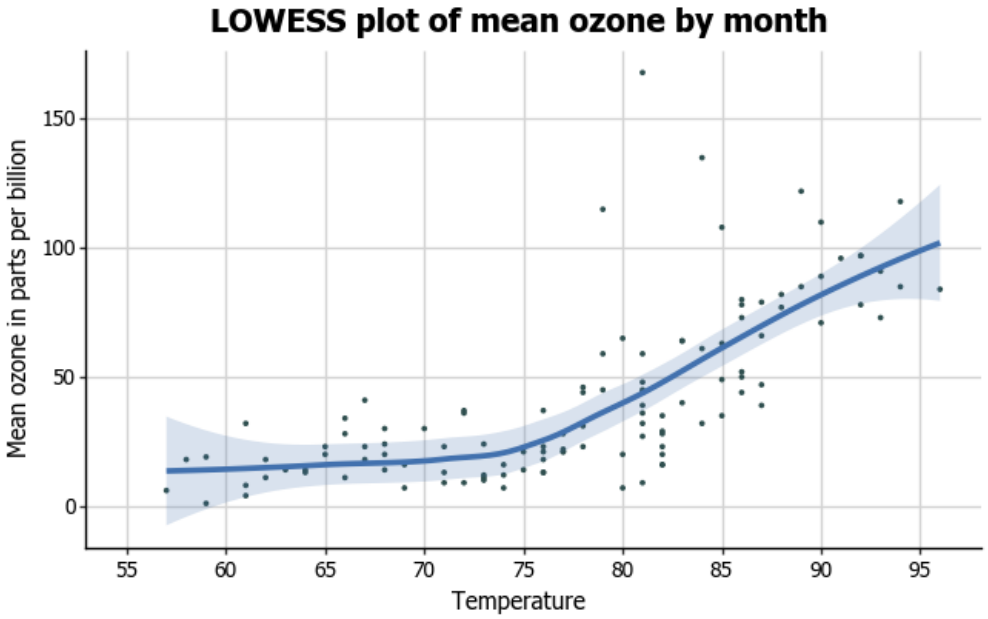
Now that we've explored some of the options available in plot customisation, we can now build our own completely customised graph:

- Changing the `size` and `colour` arguments of `axis_line` allows us to thicken the lines and change their colour to black;
- Changing the colour argument passed to `panel_grid_major` means that all of our major grid lines are now light grey;
- Similarly, we removed the minor grid lines and background by changing the arguments of `panel_grid_minor`, `panel_border` and `panel_background`;
- We've changed the font using the standard font Tahoma.

With all of these customisations, we now finally have the graph we presented at the beginning of this chapter.

```
p12 = (
    ggplot(airquality, aes("Temp", "Ozone"))
    + geom_point(shape=".", colour="darkslategrey")
    + geom_smooth(method="loess", colour="#4271AE",
                  size=1.5, alpha=0.2, fill="#4271AE")
    + scale_x_continuous(breaks=np.arange(55, 96, 5),
```

```
                          limits=[55, 96])
    + ggtitle("LOWESS plot of mean ozone by month")
    + xlab("Temperature")
    + ylab("Mean ozone in parts per billion")
    + theme(
        axis_line=element_line(size=1, colour="black"),
        panel_grid_major=element_line(colour="#d3d3d3"),
        panel_grid_minor=element_blank(),
        panel_border=element_blank(),
        panel_background=element_blank(),
        plot_title=element_text(size=15, family="Tahoma",
                                face="bold"),
        text=element_text(family="Tahoma", size=11),
        axis_text_x=element_text(colour="black", size=10),
        axis_text_y=element_text(colour="black", size=10),
    )
)
p12
```



Further reading

Plotnine

Hassan Kibirige. Plotnine: A Grammar of Graphics for Python[10]

Monash University. Making Plots With ggplot (aka plotnine)[11].

Emre Can. Emulating R regression plots in Python[12].

Python 3

Paul Gries, Jennifer Campbell & Jason Montojo. Practical Programming: An Introduction to Computer Science Using Python 3[13]. Pragmatic Bookshelf, 2013.

Pandas

Wes McKinney. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython[14]. O'Reilly Media, 2017.

Monash University. Data Analysis with Python[15].

[10] https://plotnine.readthedocs.io/en/stable/
[11] https://monashdatafluency.github.io/python-workshop-base/modules/plotting_with_ggplot/
[12] https://medium.com/@emredjan/emulating-r-regression-plots-in-python-43741952c034
[13] https://bit.ly/2OkSQqV
[14] https://amzn.to/2lrRafO
[15] https://monashdatafluency.github.io/python-workshop-base/modules/working_with_data/