

5543 Report - ggplotIntro

Yuheng Cui

10/16/2021

Contents

1	Introduction	1
1.1	Context of the project	2
2	Design of the shiny app and how to use it	2
2.1	Gamification	2
2.2	Why choosing shiny app	2
2.3	Why designing in this way	3
3	Challenges	18
3.1	Text display issue	18
3.2	Comparison to the solutions	18
3.3	Pop-up windows	22
3.4	Modulising the shiny app through <code>source()</code>	23
3.5	Making up questions and finding data sets	23
3.6	Make the shiny app as package	23
3.7	Some improvement	23
	References	23

1 Introduction

This report will introduce R package `ggplotIntro` and how I develop the package. I choose R for three main reasons: first, R is free and open-sourced; second, people can do basic analyses without advanced coding or programming knowledges; third, also the most important point, R has hundreds of packages. The `ggplot2` package is a very useful and basic package. Myint et al. (2020) suggests that basic R and `ggplot2` can make plots rated similarly on many characteristics, but "... ggplot2 graphics were generally perceived by students to be slightly more clearer overall with respect to presentation of a scientific relationship." `ggplot2` is so representative because it shows how R can draw plots and it always the first package when a person starts learning R. If a new learner knows how to use `ggplot2` package, the person can do basic data plotting and analysis. `ggplotIntro` Package is mainly used for new `ggplot2` package learner. Targeted group is new R learner. New R learners are defined as people who never used R before and have little knowledge

about programming. I understand the difficulties for new learners because I learned accounting for my bachelor degree and I still remember how I struggled when I started learning R – those lines of code look like puzzles to me. For anyone who wants to learn programming languages, not only R, two most important characteristics are self-motivation and self-learning. Because R is open-sourced, there are many websites can teach how to learn R, such as *stackoverflow* and *RStudio Community*. For most of problems and issues, we can find answers by googling. But I know a realistic issue for most new learners – learning programming, especially at beginning, it's boring and difficult. For myself, at beginning, I always copy and paste sample code and make minor changes to see how output changes. It takes me a while to understand how to read documentation of packages and functions. Most of university students do not have problems about self-learning, but self-motivation could be an issue. As I said, programming is boring at beginning, so new learners need motivation. Motivation can be internal or external. My project is aiming to provide **external motivation**.

1.1 Context of the project

Due to COVID-19, there is no companies needs interns now. First time I get in touch with my supervisor was in week 7. In first two meetings, we just have brief talk about the project and I drew a simple design sheet about the project. So, I started to work on the project since week 8.

2 Design of the shiny app and how to use it

2.1 Gamification

The `ggplotIntro` should be interesting and help new learner cultivate interests in R. Interest was recognised as an independent factor and a desired outcome in the process of learning (Schiefele, 1991). Lepper (1988) proposed four general ways to increase intrinsic motivation, two out of four could be satisfied through gamification – promoting students' sense of control and providing challenging activities. In the shiny app, `shinyAce` editor provide similar environment as R studio, and the difficulty levels are increasing across sections.

So, I studied how to make learning progress interesting. My supervisor and I both agreed that gamifying learning progress could be a good option. Educational games have been successfully used to teach a number of school subjects (Corbett et al., 2001). When a person receives positive feedback from learning/gaming, he/she is likely to be motivated and wants to learn/play more. The game versions of scoreboard were much more enjoyable the normal version, and were more acceptable (Flatla et al., 2011) So, the project must have a score system. A pop-up window will send a congratulatory message when the user gives correct answer or send a message containing comforting words and tips on solving questions. Pop-up window is feedback when learners submitting their answers, and the aim of pop-up windows is making people feel more ownership and purpose when engaging with tasks (Pavlus, 2010). I believe a learner will be confident when he/she got high scores in exercises.

2.2 Why choosing shiny app

At first, my plan was to build a package like `learnr` – use `shiny_prerendered` Rmd file to make exercises. The advantage of this plan is good example and template already existed. I can do my project by following the template. But I prefer to use shiny app because I want to make the project more interactive and more like games. Gamification is very important in my project, because I want to show new learners that make plots through R is interesting, funny and easy. If this project just asks people doing exercises, in conventional way, the project will be just like most online tutorials.

2.3 Why designing in this way

2.3.1 First tab in the shiny app

Figure 1 is an overview of my project. Left hand side is the list of contents, and right hand side is the content. The list of contents is in this order because I believe it is a common process when people draw plots. In other words, when people get access to a data set and want to draw plots for analysis, first thing first is understanding the data – such as, dimension, variable types, and missing values of the data. I introduce how to use `?` to read documentation of data sets built inside the packages. And use `summary()` to have an overview of the data set. This is the basis of data analysis and very crucial because it is unlikely to draw good plots if you know nothing about the data. I also take screenshots of `mtcars` documentation and `summary` as examples in first tab.

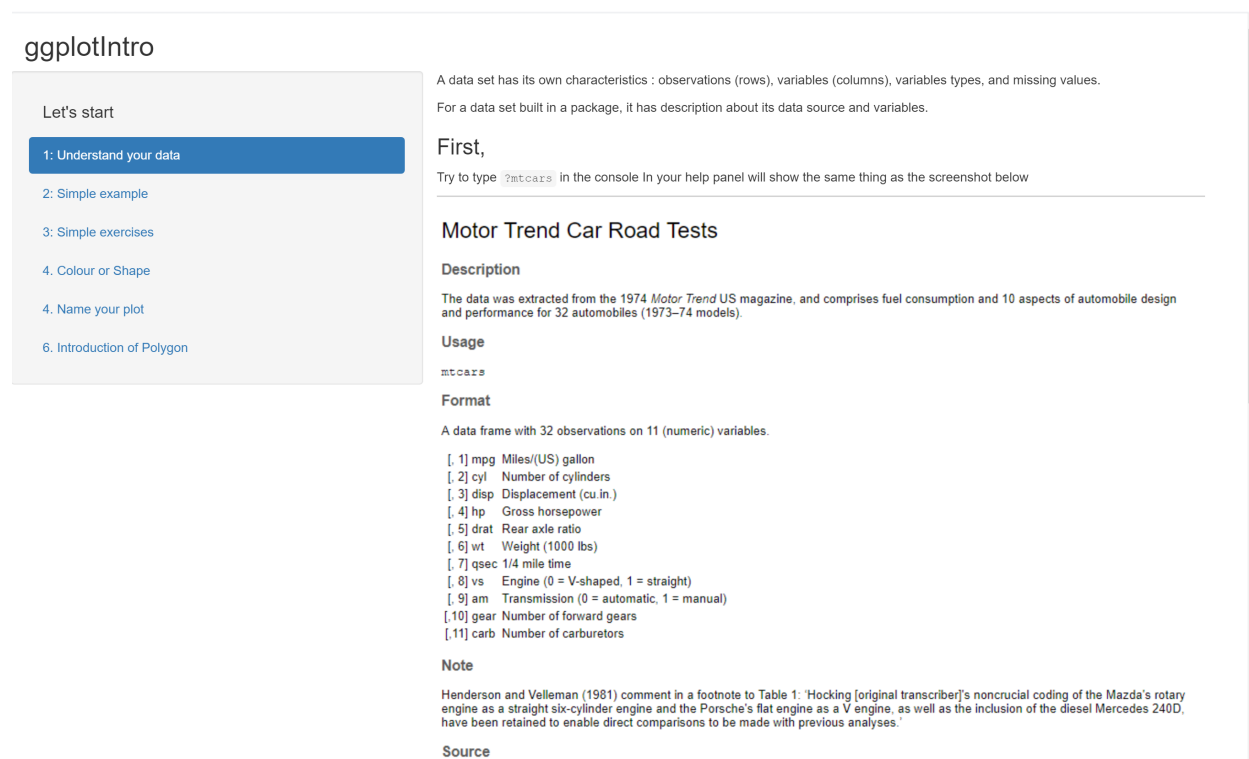


Figure 1: Section 1 screenshot

2.3.2 Second tab in the shiny app

In second tab, I let learners play around with the code. Learners can change variables and plot types and see how the code and the plot change when they select different variables and plot types. I use `mtcars` as illustration example because it is a small data set with all numeric variables. One goal of this tab is to tell learners that it is not very hard to use `ggplot2` package to draw plots, and coding is very understandable and organised stuff. At the end of second tab, I paste a link to `ggplot2` website. On the website, more `geom_` functions are introduced. I do not want to send too much contents to new learners because this shiny app is just a start point of learning R. Another goal of second tab is to show basic `ggplot2` code to learners. Knowing the basic code is enough to draw the plot. And this leads learners to third tab of the shiny app.

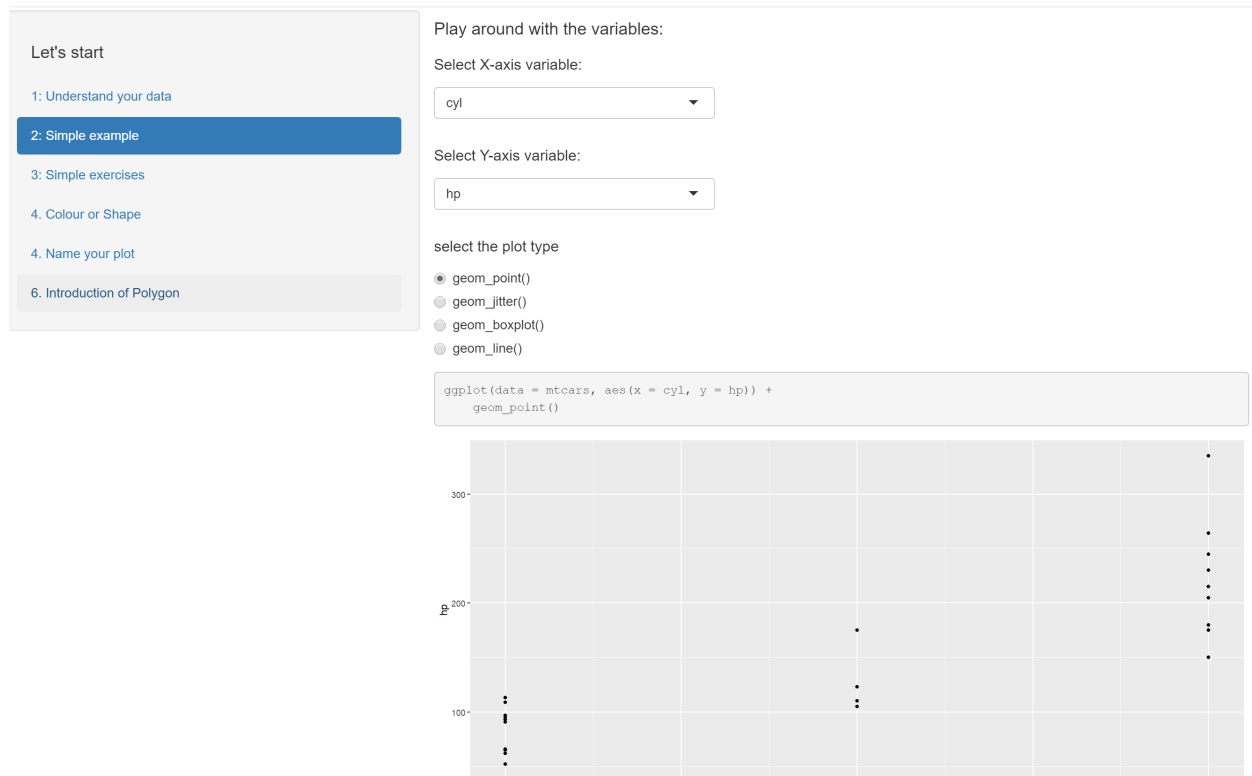


Figure 2: Section 2 screenshot

2.3.3 Third tab in the shiny app

In third tab, I make up three very easy questions for learners. If learners read the first two tabs, they can complete all three questions in five minutes. The screenshot below shows the tab layout. The questions are all designed as fill-in-blanks style (Figure 3). I believe this style of questions is the most acceptable questions for new learners. There are three reasons. First, this kind of questions is easy and won't take too much time. Because this project focuses on beginner, it is not reasonable to make it very long and hard. If exercises are long, users would bother to try them; and if exercises are hard, it would make beginners lose confidence. Second, fill-in-blank questions are easier for me to make comparison to the solution. I will discuss this point later in the 3.2 section. Third, fill-in-blanks questions can help new learners form a good coding style. When lines of code are extraordinarily long, good coding style and necessary comments are crucial. For writer him- or her- self, good coding style can make debug and review more easily. It is common that when we try to review our work which is done a few years or several months ago, we even don't understand what we were doing at that time. So, good coding style and comments can help us remember. For other people who want to read the code, good coding style can make the code more readable. Spinellis (2003) mentioned "... programming usually is a team-based activity, and writing code that others can easily decipher has become a necessity." So, good coding style can also ensure other team members to continue your works.

After complete the question, learners can click **Submit** button to see whether they are correct (Figure 3). If they are wrong, pop-up window will be like Figure 4. It may contain a hint message. The hint message can guide learners to solve the question. When your answer is wrong, there will be no score added in the tab.

If they are correct, the pop-up window will be like Figure 5. And when the question is correctly answered, one score will be added.

Users can use the **Solution** button to see the solution to the question (Figure 6).

And users can click the below-the-folder button to check how many questions that they correctly answer in

Let's understand your data set.

```
1  
2 ...{r}  
3 diamonds  
4 ...  
5
```

Submit

diamonds

```
## # A tibble: 53,940 x 10  
##   carat cut      color clarity depth table price      x      y  
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl>  
## 1  0.23 Ideal     E      SI2     61.5    55   326   3.95   3.98  
## 2  0.21 Premium  E      SI1     59.8    61   326   3.89   3.84  
## 3  0.23 Good     E      VS1     56.9    65   327   4.05   4.07  
## 4  0.29 Premium  I      VS2     62.4    58   334   4.2    4.23  
## 5  0.31 Good     J      SI2     63.3    58   335   4.34   4.35  
## 6  0.24 Very Good J      VVS2    62.8    57   336   3.94   3.96  
## 7  0.24 Very Good I      VVS1    62.3    57   336   3.95   3.98  
## 8  0.26 Very Good H      SI1     61.9    55   337   4.07   4.11  
## 9  0.22 Fair     E      VS2     65.1    61   337   3.87   3.78  
## 10 0.23 Very Good H      VS1     59.4    61   338   4      4.05  
## # ... with 53,930 more rows, and 1 more variable: z <dbl>
```

Solution

Remember: you can always use `?diamonds` to read more information about the data set.

Figure 3: Section 3 - Question 3 screenshot

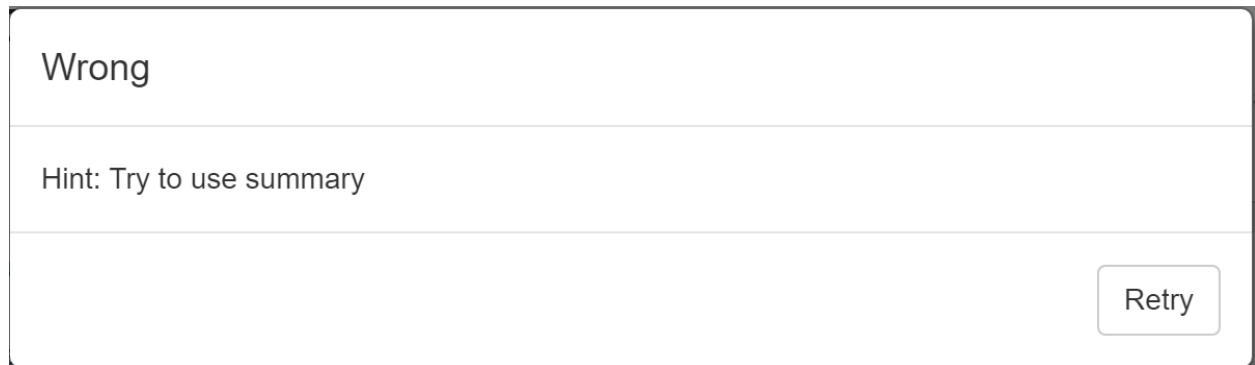


Figure 4: When answer is wrong, pop-up window will be like this

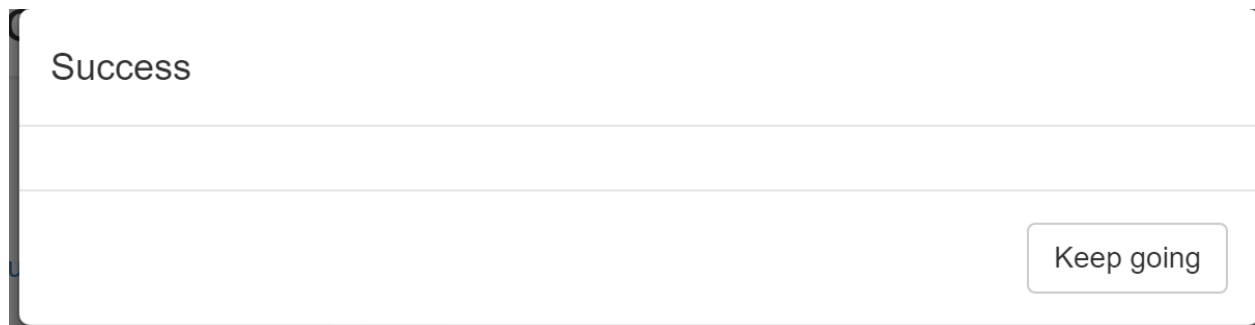


Figure 5: When answer is correct, pop-up window will be like this



Figure 6: Solution

the section. Users will have two different pop-up dialog: upper window in Figure 7 will appear when users do not answer all questions correctly, lower window in Figure 7 will appear is when all questions are solved correctly.

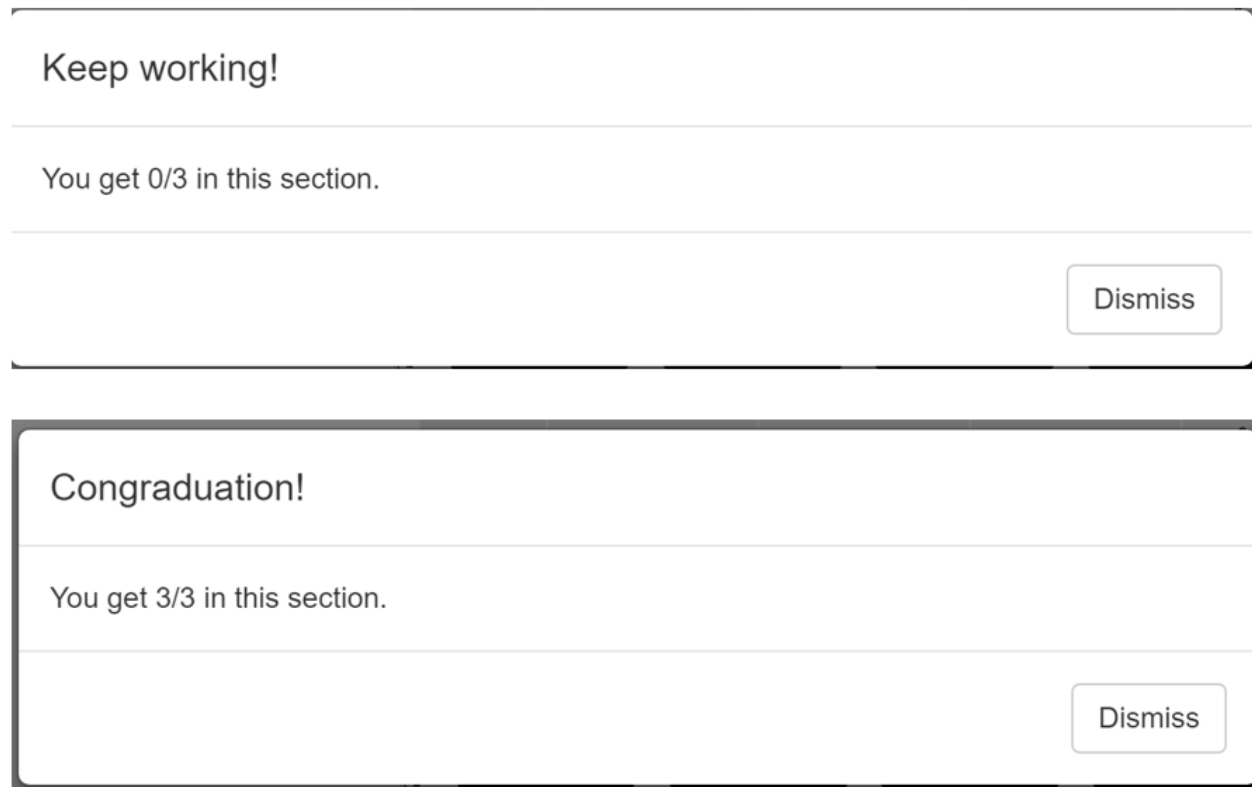


Figure 7: Score system will show two types of outcome

In this section, I use `diamonds` data set, because `diamonds` data set contains numeric and categorical variables. It is difficult to find a basic data set contains both types of variables. But `diamonds` has a huge disadvantage – it has 53940 observations. If I use all `diamonds` observations to make plot, it will take one minute to render the plot. So, I use `head()` to extract first 1000 rows in `diamonds` to reduce the time. The code chunk below is what I did for Q2. I think `head()` is reasonable because the goal of the project is not analysing data but helping new learners understand and get familiar with code.

```
ggplot(data = head(diamonds,1000), aes(x = ___, y = ___)) +  
  geom_point()
```

In Q3, I talk a little about the weakness of scatterplots. And after that, I briefly introduce `geom_jitter()` and `geom_boxplot()`. In this project, I try my best to prevent overwhelming new learners. But `geom_jitter()` and `geom_boxplot()` are commonly used in practice, so, I insist introducing them.

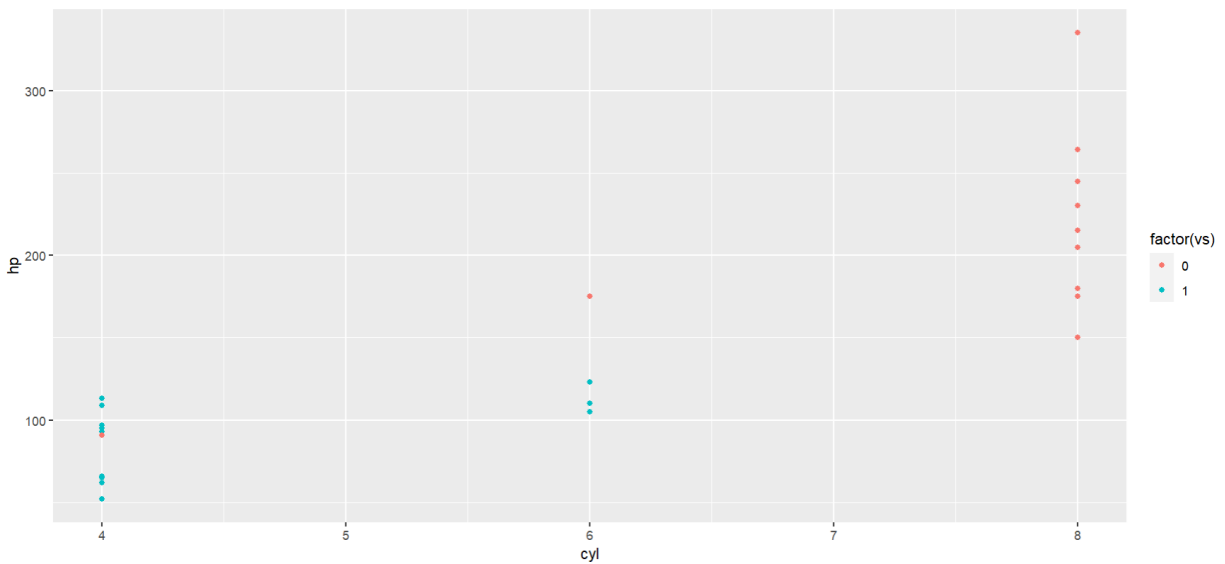
2.3.4 Fourth tab in the shiny app

In fourth section, I introduce how to use colours and shapes in data visualisation. Stone (2006) suggests colour can enhance and clarify a presentation. 2D plots are more understandable than 3D plots. Colour and/or shape will be additional dimension(s) in the plot. In this section, I mainly introduce **colour**. Colour is more commonly used than shape in data visualisation. People's eyes cannot detect the differences if there are more than four types of shapes in a plot. Colour can represent categorical variables, low to high

numeric values, and diverging values. Colour can also represent variables with many levels. However, how to effectively use colour in data visualisation is a big topic, I can only introduce the basis of colour in data visualisation. At the end of introduction part, I use `mtcars` as example to show learners how useful colour is in data visualisation (Figure 8).

Example

```
ggplot(data = mtcars, aes(x = cyl, y = hp, colour = factor(vs))) +  
  geom_point()
```



Here, we use colour to represent the types of engine. 0 is V-shaped engine, while 1 is straight engine.

Figure 8: Section 4 screenshot

After introduction of colour and shapes, I give two simple exercises. I make Q4 and Q5 in order to differentiate `colour` and `fill` argument in ggplot. I use `mtcars` in Q4 while using `txhousing` in Q5. I introduce bar chart in Q5. There are too many plot types, thus, I must choose some most frequently used types. Based on my personal experience, `geom_point()`, `geom_line()` and `geom_bar()` are top-3 commonly used geom in ggplot. Q4 and Q5 tells learners that `colour` should be used in scatter plot while `fill` should be used in bar plot. If `colour` is used in Q5, the pop-up window will tell him/her `fill` should be used here (Figure 9).

Wrong

Try to use fill to replace colour.

Retry

Figure 9: The pop-up window for Q4

2.3.5 Fifth tab in the shiny app

In fifth section, I introduce how to make labels in graphs. In previous section, I introduce how to make basic plots; and when a raw plot is done, plot makers should label the plot, including x- and y-axis names and units, plot title, the data source (if applicable), and legend title (if applicable). Without proper labels, plot itself will be less readable and understandable. For example, length can be measured in many different units, such as meter, inch, kilometer, etc.. And in previous section, in `mtcars`, `cyl` and `hp` are x- and y-axis, respectively. They are meaningless for readers because they have little knowledge about the data set. We must use the full name of the variables with their units to represent coordinates. So, I make one question in this section (code shown below). There is only one question because I think labelling is easy.

```
ggplot(mtcars, aes(x = cyl, y = hp)) +  
  geom_point() +  
  ___(___ = "Relationship between Gross horsepower and Number of Cylinders for 32 automobiles in 1974",  
    ___ = "Number of Cylinders",  
    ___ = "Gross horsepower")
```

2.3.6 Final tab in the shiny app

The final section is harder than previous sections – I introduce Polygon. I struggle for a while because I am not sure whether I should introduce Polygon to new learners. When I learned Polygon, I thought it was difficult. But Polygon is a simple way to draw maps, and mapping is an important part in data visualisation – John Snow’s cholera map (Figure 10) is well-known in data visualisation history. And personally, I like mapping in data visualisation.

In this section, I introduce two steps of drawing a map (Figure 11). First step is “get map data” and second step is “draw the map.” Here, I introduce the simplest way to get map data. And in both steps, I give comments to help learners understand the code.

I also use `geom_point()` to draw the map. The aim is to show why we should use Polygon (Figure 12).

Next, I use `who_covid` data set and draw map for it. `who_covid` (Figure 13) is a data set of `ggplotIntro` package, data wrangling part will not appear in the formal shiny app within the package. `who_covid` data is downloaded from **WHO** website.

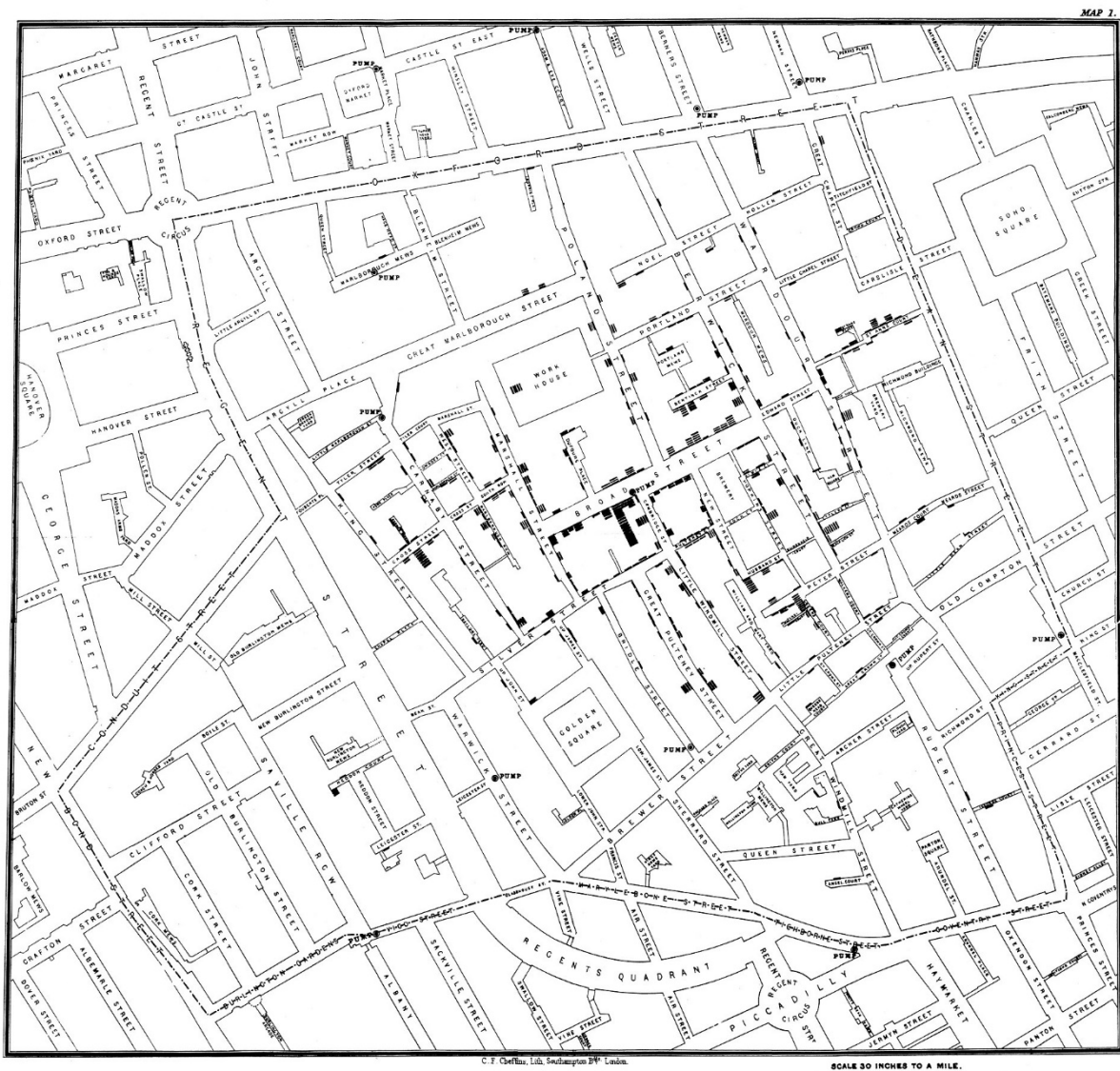
Figure 14 shows three steps of mapping. First step is to rename two countries to make two data sets can be merged. Second step is to merge `who_covid` and map data. Third step is to use `geom_polygon()` to draw the map.

In this section, there are two questions – Q7 and Q8. Q7 (Figure 15) is easy – it is almost same to the example above, except `deaths_cumulative_total` being used. Q7 is just warm up. When people first learn Polygon, they may confuse about the arguments within `aes`, such as, `group`, `fill`, `longitude` and `latitude`. So, I intentionally make this simple question.

Q8 (Figure 16) is little bit harder than Q7, because learners must fill in all blanks. I use a new data set here. GDP data is downloaded from **World Bank** website. Data wrangling is difficult for new learners, so I do all data wrangling part and merge it with map data. The merged data is called `GDP_clean`. If Q8 is answered wrongfully, hint message will pop up (Figure 17).

2.3.7 About pipe operator (`%>%`)

In early version of shiny app, I frequently used pipe operator. In third meeting with Emi (my supervisor), she told me I should try to avoid using pipe operator, because it could be confusing to new learners. This remind me when I started learning R, I was also confused about pipe operator. So, I took her advice, and remove most of the pipe operators in the shiny app. However, in last two sections, in data wrangling part, the code would be too complex if not using pipe operators. But I think data wrangling part is not the main focus of the project, and I give enough comments in those parts by commenting after the lines of code.



`Polygon(geom_polygon())` is a simple way to draw maps – it draws boundaries for different regions.

Step 1: get map data

Use `ggplot2::map_data()` to get the wanted map data.

```
# install.packages('map') # map package is required
world <- map_data('world')

head(world)
```

```
##           long      lat group order region subregion
## 1 -69.89912 12.45200     1     1  Aruba      <NA>
## 2 -69.89571 12.42300     1     2  Aruba      <NA>
## 3 -69.94219 12.43853     1     3  Aruba      <NA>
## 4 -70.00415 12.50049     1     4  Aruba      <NA>
## 5 -70.06612 12.54697     1     5  Aruba      <NA>
## 6 -70.05088 12.59707     1     6  Aruba      <NA>
```

We can see what map data looks like. Different countries and regions have different `group` number. If you want to get map data other than `world`, try `?map_data`.

Step 2: draw the map

```
ggplot(world, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = 'white', # This is the colour within countries' boundaries
              colour = 'grey' # This is background colour
              ) +
  coord_quickmap() # This is map projections, there are many other types of map projections
```

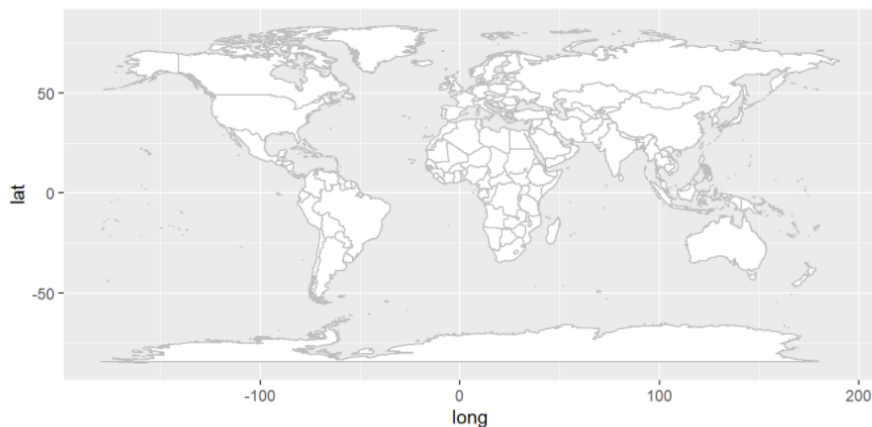


Figure 11: Section 6 - Introduction

What if we use `geom_point()`, instead?

```
ggplot(world, aes(long, lat, group = group)) +  
  geom_point(size = 0.25) +  
  coord_quickmap()
```

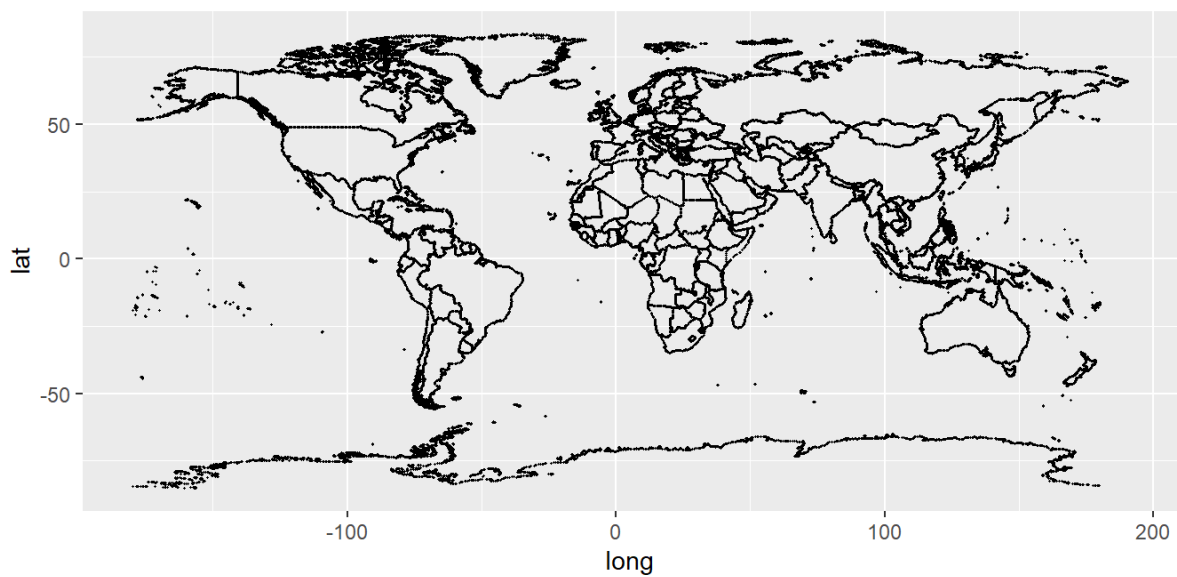


Figure 12: Section 6 - comparing scatter plot to polygon

Let's see what we can do for `who_covid` data

`who_covid` records COVID-19 cases and deaths in the countries of the world. Try to use `?who_covid` to read more information about the data.

```
head(ggplotIntro::who_covid)
```

```
## # A tibble: 6 x 6
##   name                who_region cases_cumulativ~ cases_c
umulativ~ deaths_cumulati~
##   <chr>                <chr>                <dbl>
##   <dbl>                <dbl>
## 1 Global                <NA>                232075351
##   2977.                4752988
## 2 United States of America Americas                42648573
##   12885.                683243
## 3 India                South-Eas~                33697581
##   2442.                447373
## 4 Brazil                Americas                21351972
##   10045.                594443
## 5 The United Kingdom    Europe                7701719
##   11345.                136208
## 6 Russian Federation    Europe                7464708
##   5115.                205531
## # ... with 1 more variable: deaths_cumulative_total_per_100000
##   _population <dbl>
```

Figure 13: Section 6 - who covid data set

First, rename USA and UK, because in world data United States of America is called USA and United Kingdom is called UK.

```
covid <-  
  mutate(ggplotIntro::who_covid, name = case_when(name == "United  
States of America" ~ "USA",  
                                                    name == "United Kingdom" ~ "UK",  
                                                    TRUE ~ name))
```

Second, merge two data set into one data set.

```
covid_map <- left_join(covid[-1,], # -1 is to deselect Global data  
                      world,  
                      by = c("name" = "region") # select name and region to join by  
                      )
```

Finally, draw the map.

```
ggplot(data = covid_map, aes(x = long, y = lat, group = group, fill = cases_cumulative_total)) +  
  geom_polygon(colour = 'white') +  
  coord_quickmap()
```

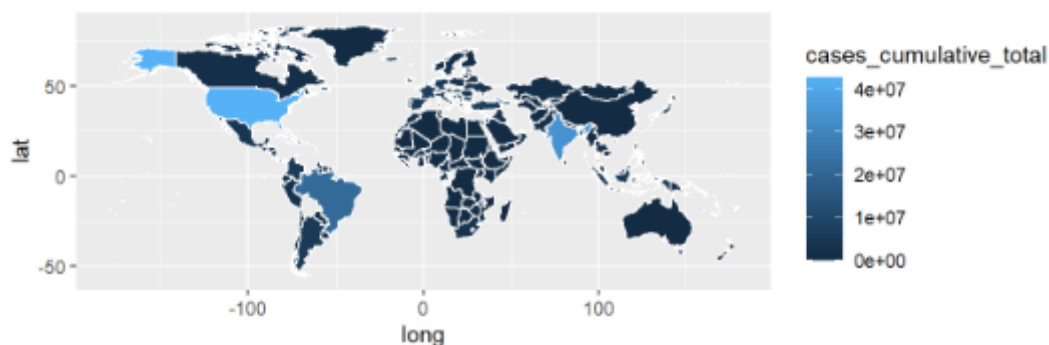


Figure 14: Section 6 - who covid example

Now, let try two simple exercises.

Q7: create a map that shows total deaths of each country. **Hint:** The variable you need is *deaths_cumulative_total*.

```
1 |  
2  
3  
4 ```{r}  
5 ggplot(data = covid_map, aes(x = __, y = __, group = __, fill = __)) +  
6   geom_polygon(colour = 'white') +  
7   coord_quickmap()  
8  
9 ```  
10
```

Submit

```
ggplot(data = covid_map, aes(x = __, y = __, group = __, fill = __)) +  
  geom_polygon(colour = 'white') +  
  coord_quickmap()
```

```
## Error: <text>:1:34: unexpected input  
## 1: ggplot(data = covid_map, aes(x = _  
##                                     ^
```

Solution

Figure 15: Section 6 - Q7

This is `GDP_clean` data.

```
## # A tibble: 6 x 8
##   country_name year   GDP long  lat group order subregion
##   <chr>         <dbl> <dbl> <dbl> <dbl> <dbl> <int> <chr>
## 1 Zimbabwe     2020 1128.  32.0 -21.7  1624 100896 <NA>
## 2 Zimbabwe     2020 1128.  31.9 -21.8  1624 100897 <NA>
## 3 Zimbabwe     2020 1128.  31.7 -22.0  1624 100898 <NA>
## 4 Zimbabwe     2020 1128.  31.6 -22.2  1624 100899 <NA>
## 5 Zimbabwe     2020 1128.  31.4 -22.3  1624 100900 <NA>
## 6 Zimbabwe     2020 1128.  31.3 -22.4  1624 100901 <NA>
```

Q8: create a map that shows GDP of each country.

```
1
2
3
4   ```{r}
5   ggplot(GDP_clean, aes(____, _____ = _____, _____ = _____)) +
6     geom_polygon(colour = 'white') +
7     coord_quickmap()
8
9   ```
10
```

Submit

Figure 16: Section 6 - Q8

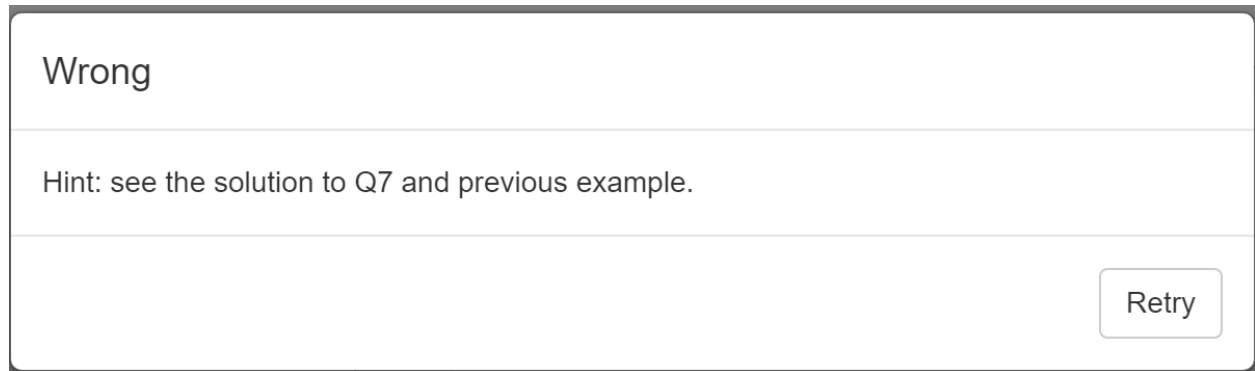


Figure 17: Section 6 - Hints for Q8

2.3.8 Happen coincidence

I don't want learners click the **Solution** button before at least trying the exercises. The aim of **Solution** button is to give learners some hints when they feel the exercises are too hard. In fact, in the shiny app, users cannot see the solutions before clicking **Submit** button. As the screenshot below showing, the **Solution** will keep loading and show nothing (Figure 18). I tried and waited for over two minutes, but it was still loading.

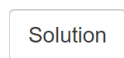


Figure 18: Loading Solution

After clicking **Submit** button, the solution will appear (as screenshot shown below)(Figure 19). I do not truly understand why this happens. I guess it may be sequential order between **Submit** and **Solution** buttons. That is, the content in **Solution** can be rendered only when users click **Submit** first.

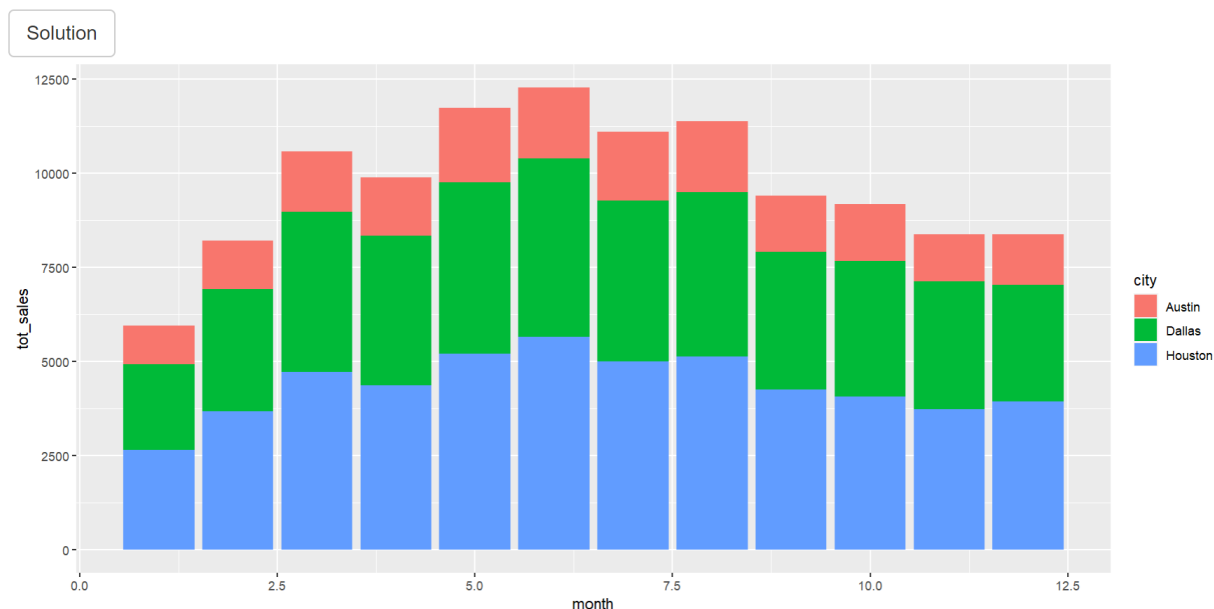


Figure 19: Solution output

3 Challenges

3.1 Text display issue

The first issue I met is part of the text in the shiny app does not display in right way. Two screenshots depict the issue. After I maximising the window of the shiny app, some paragraphs would be dislocated (Figure 20 and Figure 21). Later, I found the reason is the issue of the column width in the shiny app – when I write too much text in shiny app, the text will find a way to fill the full column width.

At first, I tried to set specific column width for each paragraph, but it did not work. Thanks for Mitch’s advice, I used `includeMd` and `includeHTML` to solve the problem. I learned that it is better not to include too much text in a shiny app. Too much text will make the shiny app unnecessarily long, and it is difficult to design pretty layout in the shiny app. For instance, in my shiny app, I include many screenshots and plots. I must create ui and sever to show those screenshots or plots. So, my workload increases and later if there are too much plots the loading speed of the shiny app will be slow. But if I use Md file or HTML file (generated by Rmd file), I can easily make the layout and draw plots. In addition, because HTML files are prerendered, reading HTML files is faster than generating plots in the shiny app. Also, `includeHTML` is just one line of code, it saves spaces and shortens the length of the shiny app.

3.2 Comparison to the solutions

First, I tried to follow the structure of `dwexercise` package. But there are big difference between *shiny-prerendered Rmd file* and *shiny app*. If using *shiny-prerendered Rmd file*, with `learnr` package, it is very easy to create code chunks letting users fill the answers. If using shiny app, the structure will be totally different, because *shiny-prerendered Rmd file* supports users to run code while *ui* and *server* are both required in shiny app. Ui input can be problematic because some types of input would cause errors in *server* part. For instance, Figure 2 was the first-version plan – the questions provide multiple choices to learners. The advantage of this plan is easy to construct; the disadvantage are the sense of control being reduced and limited choices could limit learners thinking.

engine as a straight six-cylinder engine and the Porsche's flat engine as a V engine, as well as the have been retained to enable direct comparisons to be made with previous analyses."

Source

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, 37

Examples

```
require(graphics)
pairs(mtcars, main = "mtcars data", gap = 1/4)
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 1)
## possibly more meaningful, e.g., for summary() or bivariate plots:
mtcars2 <- within(mtcars, {
  vs <- factor(vs, labels = c("V", "S"))
  am <- factor(am, labels = c("automatic", "manual"))
  cyl <- ordered(cyl)
  gear <- ordered(gear)
  carb <- ordered(carb)
})
summary(mtcars2)
```

This step is important before drawing any plots, because different types of plots are suitable for different types of variable.

Second,

Use `skimr::skim()` or `summary()` function to have a first look of your data.

```
> summary(mtcars)
```

mpg	cyl	disp	hp	dr
Min. :10.40	Min. :4.000	Min. : 71.1	Min. : 52.0	Min.
1st Qu.:15.43	1st Qu.:4.000	1st Qu.:120.8	1st Qu.: 96.5	1st Qu.
Median :19.20	Median :6.000	Median :196.3	Median :123.0	Median
Mean :20.09	Mean :6.188	Mean :230.7	Mean :146.7	Mean
3rd Qu.:22.80	3rd Qu.:8.000	3rd Qu.:326.0	3rd Qu.:180.0	3rd Qu.
Max. :33.90	Max. :8.000	Max. :472.0	Max. :335.0	Max.

wt	qsec	vs	am	carb
Min. :1.513	Min. :14.50	Min. :0.0000	Min. :0.0000	Min.
1st Qu.:2.581	1st Qu.:16.89	1st Qu.:0.0000	1st Qu.:0.0000	1st Q
Median :3.325	Median :17.71	Median :0.0000	Median :0.0000	Medi
Mean :3.217	Mean :17.85	Mean :0.4375	Mean :0.4062	Mean
3rd Qu.:3.610	3rd Qu.:18.90	3rd Qu.:1.0000	3rd Qu.:1.0000	3rd Q
Max. :5.424	Max. :22.90	Max. :1.0000	Max. :1.0000	Max.


```
carb
Min. :1.000
1st Qu.:2.000
Median :2.000
Mean :2.812
3rd Qu.:4.000
Max. :8.000
```

Let's start from easy example

Figure 20: Text display - normal

engine as a straight six-cylinder engine and the Porsche's flat engine as a V engine, as well as the have been retained to enable direct comparisons to be made with previous analyses."

Source

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, 37

Examples

```
require(graphics)
pairs(mtcars, main = "mtcars data", gap = 1/4)
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 1)
## possibly more meaningful, e.g., for summary() or bivariate plots:
mtcars2 <- within(mtcars, {
  vs <- factor(vs, labels = c("V", "S"))
  am <- factor(am, labels = c("automatic", "manual"))
  cyl <- ordered(cyl)
  gear <- ordered(gear)
  carb <- ordered(carb)
})
summary(mtcars2)
```

This step is important before drawing any plots, because different types of plots are suitable for different types of variable.

Second,

Use `skimr::skim()` or `summary()` function to have a first look of your data.

```
> summary(mtcars)
```

mpg	cyl	disp	hp	dr
Min. :10.40	Min. :4.000	Min. : 71.1	Min. : 52.0	Min. :
1st Qu.:15.43	1st Qu.:4.000	1st Qu.:120.8	1st Qu.: 96.5	1st Qu.:
Median :19.20	Median :6.000	Median :196.3	Median :123.0	Median :
Mean :20.09	Mean :6.188	Mean :230.7	Mean :146.7	Mean :
3rd Qu.:22.80	3rd Qu.:8.000	3rd Qu.:326.0	3rd Qu.:180.0	3rd Qu.:
Max. :33.90	Max. :8.000	Max. :472.0	Max. :335.0	Max. :

wt	qsec	vs	am	carb
Min. :1.513	Min. :14.50	Min. :0.0000	Min. :0.0000	Min. :
1st Qu.:2.581	1st Qu.:16.89	1st Qu.:0.0000	1st Qu.:0.0000	1st Qu.:
Median :3.325	Median :17.71	Median :0.0000	Median :0.0000	Median :
Mean :3.217	Mean :17.85	Mean :0.4375	Mean :0.4062	Mean :
3rd Qu.:3.610	3rd Qu.:18.90	3rd Qu.:1.0000	3rd Qu.:1.0000	3rd Qu.:
Max. :5.424	Max. :22.90	Max. :1.0000	Max. :1.0000	Max. :


```
carb
Min. :1.000
1st Qu.:2.000
Median :2.000
Mean :2.812
3rd Qu.:4.000
Max. :8.000
```

Let's start from easy example

Figure 21: Text display - when maximising window

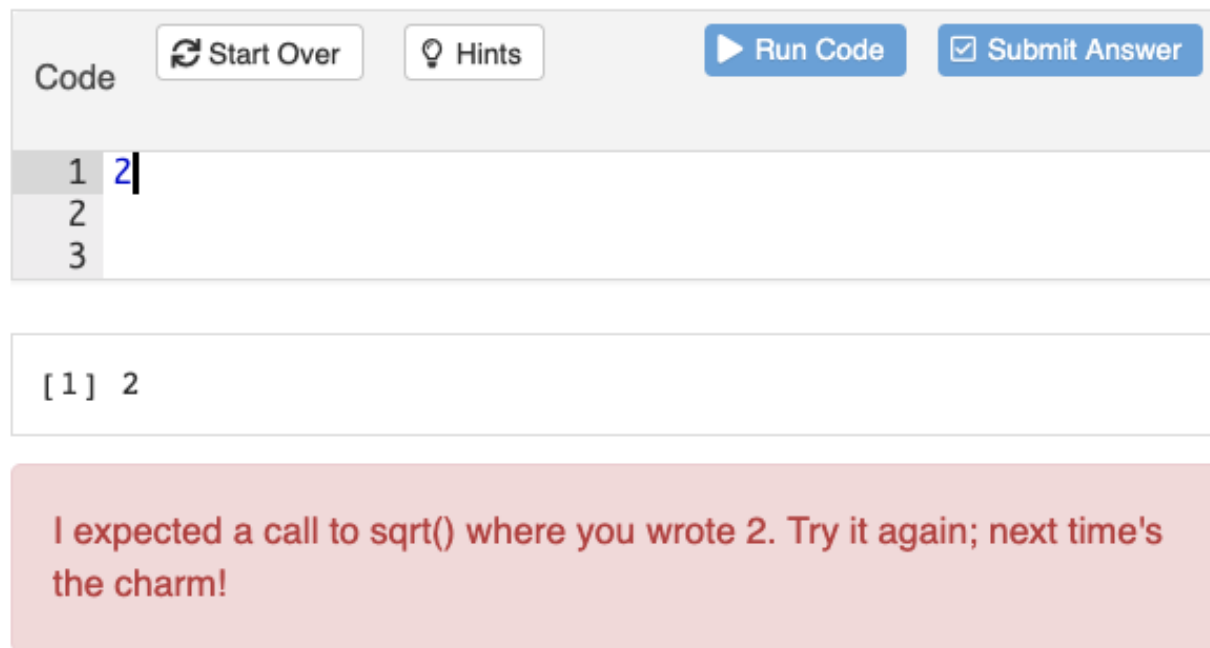
The second version was to let learners enter code freely. This version was similar to `dwexercise`, but in shiny it was more complicated. If a user wants to enter and run code in basic shiny, there are one options – `textInput`. But if using `textInput` it would generate two more problems. One way is to let users enter full code of `ggplot`. The main difficulty here is how to tell the right or wrong of the code. Different coding style and space between variables and arguments can make the comparison be difficult.

`gradethis`, `vdiffr` and `testthat` are three packages can make the comparison. `gradethis` is a package which is always used with `learnr`. It can compare learners' answers with solutions and gives messages to learners when answers are wrong. However, `gradethis` is compatible with shiny.

Here is a number. You can do great things with it, like this:

- Take the square root of the log of the number 2 .

Then click Submit Answer.



Code

Start Over Hints Run Code Submit Answer

```
1 2
2
3
```

[1] 2

I expected a call to `sqrt()` where you wrote 2. Try it again; next time's the charm!

Figure 22: `gradethis` package

Next plan is to use `vdiffr` which is a `testthat` extension for monitoring the appearance of R plots. To use this package, solution plots should be saved, so later they can be used to make the comparison. Using `vdiffr` would make the package too large because of too many SVG files saved in `_snap()` folder. `testthat` is not a good option, either. It is too complicated in shiny.

Learners must enter the code in one line if using `textInput`. This is a problem if the code is too long, such as in Polygon section. The sense of control is also reduced because learners can only know whether their answers are right or wrong. They don't know which part of the answer goes wrong. So, the project will not satisfying the goals of gamification and cultivating good coding style. Finally, `shinyAce` solved all these problems. `shinyAce` can create a code chunk which is same as the code chunk in **R studio**. It enables users to enter and run code in shiny app. Thus, even the answer is wrong, learners can see the output, which is important because there are two types of wrong answers. One is syntax error, the other is using wrong arguments

or wrong variables. This is a big disadvantage when using `textInput`. Only telling students that they are wrong could result in making students remember the right answers. Merely remembering “right answer” is harmful in learning programming. The plan is to make all exercises in the shiny app are fill-in-blanks style, and `shinyAce` is the simple way to fulfill the plan. Another reason of making fill-in-blanks exercises is relatively technical. As mentioned before, different coding style and space between variables and arguments can make the comparison be difficult. The fill-in-blanks questions are easy to make the comparison.

3.3 Pop-up windows

In early version of the shiny app, the plan was to create an UI similar to Figure 22. After a few days, as Section ?? mentioned, `gradethis`, `vdiffr` and `testthat` were not compatible with shiny. The next version of the plan was simpler than the first one – each question has a solution plot, and let users use eyes to compare their answers to the solution. This plan was abandoned soon because the shiny app would be interactive no longer. It also reduce the level of gamification. In this version of plan, the scoreboard would no longer exist because the comparison to the solution was done by users instead of the shiny app.

Later, `shinyAce` and fill-in-blanks questions were determined as the final version of the questions in the shiny app. The pop-up windows are built through `showModal` and `modalDialog`. The first version of pop-up windows was simple. It only contained one-word message – **Success** or **Wrong**. The second version differentiated the close buttons – **Keep going** was for **Success** while **Retry** was for **Wrong**. In final version, **Hint** buttons were removed, and hint messages were merged in pop-up windows. It has **three** advantages. First, it would not show hints to learners before trying the exercises. Second, it makes the layout of the shiny app look tidier and smarter. In early version, for each question, there were three buttons: **Submit**, **Hint** and **Solution**. Learners could read the hints at any time. Now, **Hint** buttons no longer exist, and the hint messages will only available to learners when learners submit wrong answers (Figure 9). Third, it reduces the length and complexity of the shiny script. Previously, in server part, there were 3-4 lines of code for hint message output; now, they were replaced by one line of code. In conclusion, pop-up windows increase the feeling of gamification and interaction, and also tidy the shiny script.

Chunk below shows the latest version of shiny script. The score and hint message are merged in one function.

```
observeEvent(input$eval8, {
  if (input$Q8 == q8sol) {
    ModalTitle = "Success"
    ModalText = NULL
    ModalFooter = tagList(
      modalButton("Keep going")
    )
    score_q8 <- 1 # Score
  } else {
    score_q8 <- 0 # Score
    ModalTitle = "Wrong"
    ModalText = "Hint: see the solution to Q7 and previous example." # Hint message
    ModalFooter = tagList(
      modalButton("Retry")
    )
  }
  showModal(modalDialog(
    title = ModalTitle,
    ModalText,
    footer = ModalFooter
  ))
})
```

3.4 Modulising the shiny app through source()

Previously, all sections were in `app.R` file. So, the `app.R` file was extremely long. It had three weaknesses. First, it took too much time when the developer wanted to make any changes in previous section because there were hundreds lines of code in `app.R` file. Second, it was unfriendly for debugging process. The debugging would be very difficult if the developer could not locate the wrongful code section. Third, the shiny app could be hardly maintained and further developed in the future.

Modulising the shiny app has two advantages. First, it can shorten the length of main shiny script. Second, it is convenient for developer and others adding more questions and section in the future. However, during modulisation process, every time restarting R and running the shiny app, the error messages, such as “q2sol not found,” appeared. At same time, the code chunks for exercises became blank boxes. The reason was all questions were saved as rds file and sourced by the shiny app, but exercise entries and solutions were excluded. The problems was solved through sourcing the whole question script.

3.5 Making up questions and finding data sets

It is tricky to find useful and interesting data sets and to make up questions based on the data.

Who_covid from WHO GDP from World Bank Data wrangling

Questions Difficulty level,

3.6 Make the shiny app as package

3.7 Some improvement

References

- Corbett, A. T., Koedinger, K. & Hadley, W. S. (2001). Cognitive tutors: From the research classroom to all classrooms. In *Technology enhanced learning* (pp. 215–240). Routledge.
- Flatla, D. R., Gutwin, C., Nacke, L. E., Bateman, S. & Mandryk, R. L. (2011). Calibration games: Making calibration tasks enjoyable by adding motivating game elements. *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, 403–412.
- Lepper, M. R. (1988). Motivational considerations in the study of instruction. *Cognition and Instruction*, 5(4), 289–309.
- Myint, L., Hadavand, A., Jager, L. & Leek, J. (2020). Comparison of beginning r students’ perceptions of peer-made plots created in two plotting systems: A randomized experiment. *Journal of Statistics Education*, 28(1), 98–108.
- Pavlus, J. (2010). The game of life.(cover story). *Scientific American*, 303(6), 43–44.
- Schiefele, U. (1991). Interest, learning, and motivation. *Educational Psychologist*, 26(3-4), 299–323.
- Spinellis, D. (2003). Reading, writing, and code: The key to writing readable code is developing good coding style. *Queue*, 1(7), 84–89.
- Stone, M. (2006). Choosing colors for data visualization. *Business Intelligence Network*, 2.