Data Transformation

September 6, 2022

Unit 1: R for data mining	**Lecture 1:** Intro to modern data mining
Unit 2: Prediction fundamentals	**Lecture 2:** Data visualization
Unit 3: Regression-based methods	**Lecture 3:** Data transformation
Unit 4: Tree-based methods	**Lecture 4:** Data wrangling
Unit 5: Deep learning	**Lecture 5:** Unit review and quiz in class

1 Introduction

Today we'll continue with exploratory data analysis, focusing on **data transformation** using the `dplyr` package.

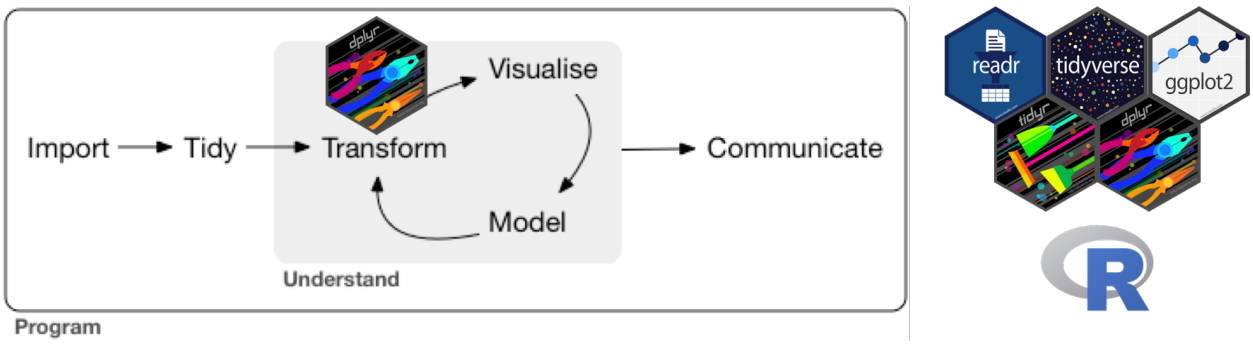


Figure 1: Data transformation (adapted from R4DS Chapter 1).

Let's load the `tidyverse` packages.

```
library(tidyverse)
```

Recall the `diamonds` dataset.

```
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
```

```
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
## # i Use `print(n = ...)` to see more rows
```

In addition to plotting these data, we might want to explore them by transforming them in various ways:

- Choose a subset of observations (rows) based on various criteria (`filter()`).
- Choose a subset of variables (columns) by their names or other criteria (`select()`).
- Reorder the rows (`arrange()`).
- Create new variables as functions of existing variables (`mutate()`).
- Collapse many values down to a single summary (`summarise()`).

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation. These functions can be strung together in sequences using the pipe (`%>%`), which is built into the `tidyverse`.

2 Isolating data

2.1 `filter()`

A **`filter`** operation subsets the observations (rows) of the data based on a certain logical condition:

```
# subset to diamonds with price at least $10,000
filter(diamonds, price >= 10000)
```

```
## # A tibble: 5,223 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  1.51 Good      H     VS2      64      59 10000  7.25  7.19  4.62
##  2  1.7  Ideal     J     VS2      60.5    58 10002  7.73  7.74  4.68
##  3  1.03 Ideal     E     VVS2     60.6    59 10003  6.5   6.53  3.95
##  4  1.23 Very Good G     VVS2     60.6    55 10004  6.93  7.02  4.23
##  5  1.25 Ideal     F     VS2      61.6    55 10006  6.93  6.96  4.28
##  6  2.01 Very Good I     SI2      61.4    63 10009  8.19  7.96  4.96
##  7  1.21 Very Good F     VS1      62.3    58 10009  6.76  6.85  4.24
##  8  1.51 Premium   I     VS2      59.9    60 10010  7.42  7.36  4.43
##  9  1.01 Fair      D     SI2      64.6    58 10011  6.25  6.2   4.02
## 10  1.05 Ideal     F     VVS2     60.5    55 10011  6.67  6.58  4.01
## # ... with 5,213 more rows
## # i Use `print(n = ...)` to see more rows
```

Commonly used **comparison operators** are `==` (equal), `!=` (not equal), `<=` (less than or equal), `<` (less than), `>=` (greater than or equal), `>` (greater than), `%in%` (in). Note that `%in%` is usually employed to check whether a categorical variable belongs to a set of values, e.g. `cut %in% c("Very Good", "Ideal")`.

Logical conditions can be combined using **boolean** operators, including `&` (and), `|` (or), and `!` (not). For example:

```
# subset to diamonds with price at least $10,000 AND clarity VVS1 or IF
filter(diamonds, price >= 10000 & clarity %in% c("VVS1", "IF"))
```

```
## # A tibble: 415 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  1.01 Very Good F     VVS1     62.9    57 10019  6.35  6.41  4.01
##  2  1.02 Very Good E     IF       61.7    60 10029  6.38  6.52  3.98
```

```
##  3  1.03 Very Good F     IF       62.8    57 10032  6.4   6.47  4.04
##  4  1    Very Good F     IF       63.2    63 10046  6.26  6.24  3.95
##  5  1.11 Ideal     G     IF       61.2    54 10053  6.71  6.73  4.11
##  6  1    Ideal     F     VVS1     62.3    53 10058  6.37  6.43  3.99
##  7  1.09 Premium   G     IF       61.3    58 10065  6.64  6.6   4.06
##  8  1.11 Very Good F     VVS1     62.5    59 10069  6.59  6.63  4.13
##  9  1.16 Ideal     G     IF       62.3    55 10082  6.79  6.73  4.21
## 10  1.16 Ideal     G     IF       62      57 10082  6.73  6.7   4.16
## # ... with 405 more rows
## # i Use `print(n = ...)` to see more rows
```

Exercise: Filter `diamonds` to those with ideal cut and at least 3 carats. How many such diamonds are there?

2.2 select()

A **select** operation subsets the columns of the data, for example based on their names:

```
# select columns corresponding to the "4 C's"
select(diamonds, carat, cut, color, clarity)
```

```
## # A tibble: 53,940 x 4
##    carat cut       color clarity
##    <dbl> <ord>     <ord> <ord>
##  1  0.23 Ideal     E     SI2
##  2  0.21 Premium   E     SI1
##  3  0.23 Good      E     VS1
##  4  0.29 Premium   I     VS2
##  5  0.31 Good      J     SI2
##  6  0.24 Very Good J     VVS2
##  7  0.24 Very Good I     VVS1
##  8  0.26 Very Good H     SI1
##  9  0.22 Fair      E     VS2
## 10  0.23 Very Good H     VS1
## # ... with 53,930 more rows
## # i Use `print(n = ...)` to see more rows
```

The `select()` function comes with helper functions, such as the following:

- `-` selects all columns except the given ones, e.g. `select(diamonds, -carat)`
- `:` selects columns between the given ones, e.g. `select(diamonds, carat:clarity)`
- `contains` selects columns containing a given string, e.g. `select(diamonds, contains("c"))`
- `starts_with` selects columns starting with a given string, e.g. `select(diamonds, starts_with("c"))`
- `ends_with` selects columns ending with a given string, e.g. `select(diamonds, ends_with("t"))`

Exercise: Select all columns except `x`, `y`, `z`.

2.3 arrange()

An **arrange** operation sorts the rows of the data frame according to one of its variables:

```
arrange(diamonds, carat)       # sort diamonds by carat (ascending)
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1   0.2 Premium   E     SI2      60.2    62   345  3.79  3.75  2.27
##  2   0.2 Premium   E     VS2      59.8    62   367  3.79  3.77  2.26
##  3   0.2 Premium   E     VS2      59      60   367  3.81  3.78  2.24
```

```
##  4   0.2 Premium   E     VS2      61.1    59   367  3.81  3.78  2.32
##  5   0.2 Premium   E     VS2      59.7    62   367  3.84  3.8   2.28
##  6   0.2 Ideal     E     VS2      59.7    55   367  3.86  3.84  2.3
##  7   0.2 Premium   F     VS2      62.6    59   367  3.73  3.71  2.33
##  8   0.2 Ideal     D     VS2      61.5    57   367  3.81  3.77  2.33
##  9   0.2 Very Good E     VS2      63.4    59   367  3.74  3.71  2.36
## 10   0.2 Ideal     E     VS2      62.2    57   367  3.76  3.73  2.33
## # ... with 53,930 more rows
## # i Use `print(n = ...)` to see more rows
```

```
arrange(diamonds, desc(carat)) # sort diamonds by carat (descending)
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  5.01 Fair      J     I1       65.5    59 18018 10.7  10.5   6.98
##  2  4.5  Fair      J     I1       65.8    58 18531 10.2  10.2   6.72
##  3  4.13 Fair      H     I1       64.8    61 17329 10     9.85  6.43
##  4  4.01 Premium   I     I1       61      61 15223 10.1  10.1   6.17
##  5  4.01 Premium   J     I1       62.5    62 15223 10.0   9.94  6.24
##  6  4    Very Good I     I1       63.3    58 15984 10.0   9.94  6.31
##  7  3.67 Premium   I     I1       62.4    56 16193  9.86  9.81  6.13
##  8  3.65 Fair      H     I1       67.1    53 11668  9.53  9.48  6.38
##  9  3.51 Premium   J     VS2      62.5    59 18701  9.66  9.63  6.03
## 10  3.5  Ideal     H     I1       62.8    57 12587  9.65  9.59  6.03
## # ... with 53,930 more rows
## # i Use `print(n = ...)` to see more rows
```

Exercise: Arrange diamonds in decreasing order of their length. How long is the longest diamond?

3 Deriving information

3.1 `mutate()`

A **`mutate`** operation adds another column as a function of existing columns:

```
# add column that is the price per carat of each diamond
mutate(diamonds, price_per_carat = price/carat)
```

```
## # A tibble: 53,940 x 11
##    carat cut       color clarity depth table price     x     y     z price_per~1
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>       <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43       1417.
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31       1552.
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31       1422.
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63       1152.
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75       1081.
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48       1400
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47       1400
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53       1296.
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49       1532.
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39       1470.
## # ... with 53,930 more rows, and abbreviated variable name 1: price_per_carat
## # i Use `print(n = ...)` to see more rows
```

Some useful functions to use with mutate are arithmetic operators (`+`, `-`, `*`, `/`, `^`) or logical comparisons (`<`,

<=, >, >=, !=). For example,

```
# add column that indicates whether a diamond's price per carat is at least $10k
mutate(diamonds, fancy_diamond = price/carat > 10000)
```

```
## # A tibble: 53,940 x 11
##    carat cut       color clarity depth table price     x     y     z fancy_dia~1
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <lgl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43 FALSE
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31 FALSE
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31 FALSE
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63 FALSE
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75 FALSE
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48 FALSE
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47 FALSE
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53 FALSE
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49 FALSE
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39 FALSE
## # ... with 53,930 more rows, and abbreviated variable name 1: fancy_diamond
## # i Use `print(n = ...)` to see more rows
```

Note that `fancy_diamond` is a logical variable.

Complex combinations of existing variable can be obtained with `mutate()` via `if_else()` and `case_when()`. For example:

```
# use if_else() if you have two cases
mutate(diamonds,
  good_value =
    if_else(
      condition = carat > 2, # check whether carat > 2
      true = price < 5000,   # if so, good value if cheaper than $5k
      false = price < 1000   # if not, good value if cheaper than $1k
    )
)
```

```
## # A tibble: 53,940 x 11
##    carat cut       color clarity depth table price     x     y     z good_value
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <lgl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43 TRUE
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31 TRUE
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31 TRUE
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63 TRUE
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75 TRUE
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48 TRUE
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47 TRUE
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53 TRUE
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49 TRUE
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39 TRUE
## # ... with 53,930 more rows
## # i Use `print(n = ...)` to see more rows
```

```
# use case_when() if you have more than two cases
mutate(diamonds,
  value =
    case_when(
      carat > 2 & price < 5000 ~ "good", # if carat > 2 and price < 5000, then good
```

```
      carat > 1 & price < 2500 ~ "ok",   # if carat > 1 and price < 2500, then ok
      TRUE                     ~ "bad"   # otherwise, bad
    )
)
```

```
## # A tibble: 53,940 x 11
##    carat cut       color clarity depth table price     x     y     z value
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43 bad
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31 bad
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31 bad
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63 bad
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75 bad
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48 bad
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47 bad
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53 bad
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49 bad
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39 bad
## # ... with 53,930 more rows
## # i Use `print(n = ...)` to see more rows
```

Exercise: Add a variable called `good_color` that is `TRUE` if the color is `D`, `E`, `F`, `G` and `FALSE` otherwise.

3.2 summarise()

A **summarise** operation calculates summary statistics combining all rows of the data:

```
# find the number of "fancy" diamonds (price per carat at least $10000),
summarise(diamonds, num_fancy_diamonds = sum(price/carat > 10000))
```

```
## # A tibble: 1 x 1
##   num_fancy_diamonds
##                <int>
## 1                617
```

Useful summary functions are `sum()`, `mean()`, `median()`, `min()` `max()` `var()`, `sd()` for numeric variables and `any()`, `all()`, `sum()`, `mean()` for logical variables. The function `n()` takes no arguments and calculates the number of observations (rows) in the data.

More than one summary can be extracted in a single call to `summarise()`:

```
# find the number of "fancy" diamonds (price per carat at least $10000),
#  as well as the mean price of a diamond
summarise(diamonds,
          num_fancy_diamonds = sum(price/carat > 10000),
          mean_diamond_price = mean(price))
```

```
## # A tibble: 1 x 2
##   num_fancy_diamonds mean_diamond_price
##                <int>              <dbl>
## 1                617              3933.
```

Exercise: Use `summarise` to determine if there are any diamonds of at least one carat that cost less that $1000.

4 Multi-step transformations

4.1 The pipe (%>%)

When stringing together multiple `dplyr` verbs, the pipe `%>%` is extremely useful. The pipe passes the quantity on its left-hand side to the first argument of the function on the right hand side: `x %>% f(y)` is translated to `f(x,y)`. The first argument of all `dplyr` verbs is the data, so the pipe allows us to apply several operations to the data in sequence. For example:

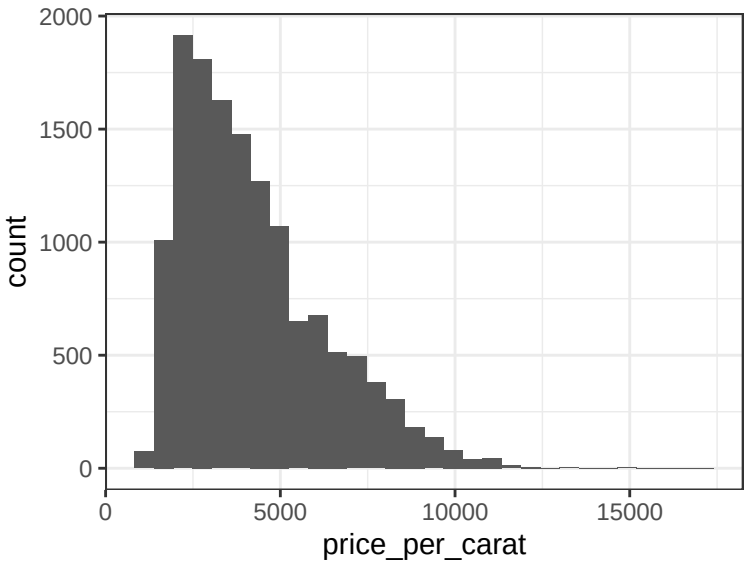
```
diamonds %>%                                    # pipe in the data
  filter(cut == "Premium") %>%                  # restrict to premium cut diamonds
  mutate(price_per_carat = price/carat) %>%     # add price_per_carat variable
  arrange(desc(price_per_carat))                # sort based on price_per_carat
```

```
## # A tibble: 13,791 x 11
##    carat cut     color clarity depth table price     x     y     z price_per_c~1
##    <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>         <dbl>
##  1  1.07 Premium D     IF       60.9    58 18279  6.67  6.57  4.03        17083.
##  2  1.01 Premium D     IF       61.6    56 16234  6.46  6.43  3.97        16073.
##  3  1.02 Premium D     IF       61.5    60 15370  6.52  6.45  3.99        15069.
##  4  1.04 Premium D     IF       60.6    56 15671  6.6   6.54  3.98        15068.
##  5  1.02 Premium D     IF       61.5    60 15231  6.45  6.52  3.99        14932.
##  6  1.21 Premium D     VVS1     60.1    59 17192  6.96  6.88  4.16        14208.
##  7  1.31 Premium D     VVS1     62.8    55 17496  7.01  6.95  4.38        13356.
##  8  1.34 Premium E     IF       61.8    58 17663  7.15  7.08  4.4         13181.
##  9  1.2  Premium D     VVS1     62.1    59 15686  0     0     0           13072.
## 10  1.28 Premium E     IF       59.8    59 15806  7.1   7.07  4.24        12348.
## # ... with 13,781 more rows, and abbreviated variable name 1: price_per_carat
## # i Use `print(n = ...)` to see more rows
```

The pipe can be used to pass data between different `tidyverse` packages, e.g. from `dplyr` to `ggplot2`:

```
diamonds %>%                                    # pipe in the data
  filter(cut == "Premium") %>%                  # restrict to premium cut diamonds
  mutate(price_per_carat = price/carat) %>%     # add price_per_carat variable
  ggplot() +                                    # start a ggplot
  geom_histogram(aes(x = price_per_carat))      # add a histogram
```

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



Exercise: Compute the mean price for diamonds of volume at least one carat.

4.2 group_by()

Sometimes we'd like to apply transformations to groups of observations based on categorical variables in our data. For example, suppose we'd like to know the maximum diamond price for each value of `cut`. We can do the following:

```
diamonds %>%                              # pipe in the data
  group_by(cut) %>%                       # group by cut
  summarise(max_price = max(price))       # find the max price for each cut
```

```
## # A tibble: 5 x 2
##   cut       max_price
##   <ord>         <int>
## 1 Fair          18574
## 2 Good          18788
## 3 Very Good     18818
## 4 Premium       18823
## 5 Ideal         18806
```

We can group by multiple characteristics, e.g.:

```
diamonds %>%                              # pipe in the data
  group_by(cut, clarity) %>%              # group by both cut and clarity
  summarise(max_price = max(price))       # find the max price for each group
```

```
## `summarise()` has grouped output by 'cut'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 40 x 3
## # Groups:   cut [5]
##    cut   clarity max_price
##    <ord> <ord>       <int>
##  1 Fair  I1          18531
##  2 Fair  SI2         18308
##  3 Fair  SI1         18574
##  4 Fair  VS2         18565
```

```
##  5 Fair  VS1         17995
##  6 Fair  VVS2        16364
##  7 Fair  VVS1        12648
##  8 Fair  IF           3205
##  9 Good  I1          11548
## 10 Good  SI2         18788
## # ... with 30 more rows
## # i Use `print(n = ...)` to see more rows
```

Note that the resulting data are still grouped based on `cut`. This is because each call to `summarise()` peels off just one layer of grouping. We might want to `ungroup()` the resulting data for downstream use:

```
diamonds %>%                                # pipe in the data
  group_by(cut, clarity) %>%                # group by both cut and clarity
  summarise(max_price = max(price)) %>%     # find the max price for each group
  ungroup()                                 # remove grouping
```

```
## `summarise()` has grouped output by 'cut'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 40 x 3
##    cut   clarity max_price
##    <ord> <ord>       <int>
##  1 Fair  I1          18531
##  2 Fair  SI2         18308
##  3 Fair  SI1         18574
##  4 Fair  VS2         18565
##  5 Fair  VS1         17995
##  6 Fair  VVS2        16364
##  7 Fair  VVS1        12648
##  8 Fair  IF           3205
##  9 Good  I1          11548
## 10 Good  SI2         18788
## # ... with 30 more rows
## # i Use `print(n = ...)` to see more rows
```

A common type of grouped summary is to tabulate the number of values of a categorical variable. A shortcut for this is the `count()` function, e.g.:

```
count(diamonds, cut)
```

```
## # A tibble: 5 x 2
##   cut           n
##   <ord>     <int>
## 1 Fair       1610
## 2 Good       4906
## 3 Very Good 12082
## 4 Premium   13791
## 5 Ideal     21551
```

Exercise: Reproduce the output of `count(diamonds, cut)` via `group_by()` and `summarise()`.

4.3 Storing the transformed data

Note that applying various functions to `diamonds` does not actually change the data itself. We can check that, after all those operations, `diamonds` is still the same as it was in the beginning:

```
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
## # i Use `print(n = ...)` to see more rows
```

If we want to save the transformed data, we have the use the assignment operator, `<-`:

```
max_prices <- diamonds %>%           # pipe in the data
  group_by(cut) %>%                  # group by cut
  summarise(max_price = max(price))  # find the max price for each cut
max_prices
```

```
## # A tibble: 5 x 2
##   cut       max_price
##   <ord>         <int>
## 1 Fair          18574
## 2 Good          18788
## 3 Very Good     18818
## 4 Premium       18823
## 5 Ideal         18806
```

5 References:

- `dplyr` cheat sheet
- Work with Data tutorials
- R4DS Chapter 5

6 Exercises

Use `dplyr` to answer the following questions:

- What is the minimum diamond price in this dataset? See if you can find the answer in two different ways (i.e. using two different `dplyr` verbs).
- How many diamonds have length at least one and a half times their width?
- Among diamonds with colors D, E, F, G, what is the median number of carats for diamonds of each cut?