

# Data Transformation

September 5, 2023

**Unit 1:** R for data mining

**Unit 2:** Prediction fundamentals

**Unit 3:** Regression-based methods

**Unit 4:** Tree-based methods

**Unit 5:** Deep learning

**Lecture 1:** Intro to modern data mining

**Lecture 2:** Data visualization

**Lecture 3:** Data transformation

**Lecture 4:** Data wrangling

**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.

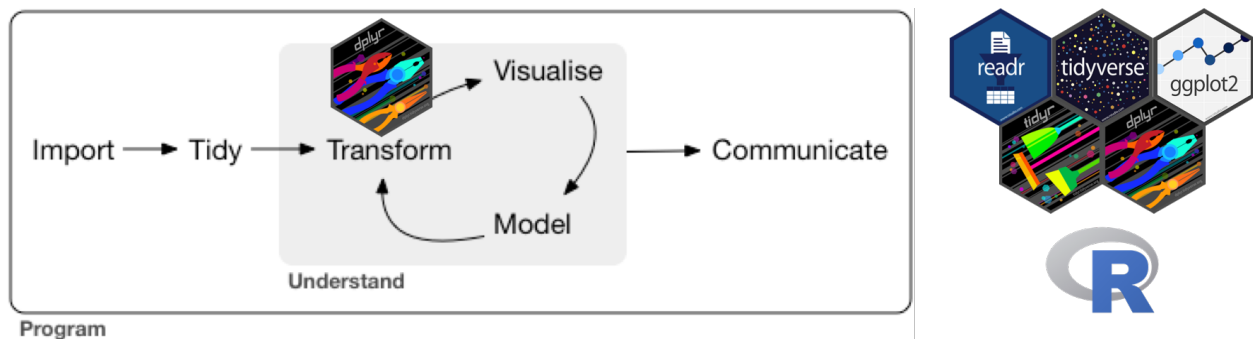


Figure 1: Data transformation (adapted from R4DS Chapter 1).

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
## # i 53,930 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 (`summarize()`).

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
## # i 5,213 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
## # i 405 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
## # i 53,930 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"))`
- **where** selects columns based on their type, e.g. `where(is.numeric)` selects all numeric columns

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
## # i 53,930 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
## # i 53,930 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_carat
##   <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 Prem~ 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 Prem~ 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~ J      VVS2      62.8  57  336 3.94 3.96 2.48      1400
## 7 0.24 Very~ I      VVS1      62.3  57  336 3.95 3.98 2.47      1400
## 8 0.26 Very~ 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~ H      VS1      59.4  61  338 4    4.05 2.39      1470.
## # i 53,930 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_diamond
##   <dbl> <ord>    <ord> <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl> <lgl>
```

```
##      <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 G~ J      VVS2     62.8  57    336  3.94  3.96  2.48 FALSE
## 7  0.24 Very G~ I      VVS1     62.3  57    336  3.95  3.98  2.47 FALSE
## 8  0.26 Very G~ 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 G~ H      VS1      59.4  61    338  4     4.05  2.39 FALSE
## # i 53,930 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
## # i 53,930 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
## # i 53,930 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 summarize()

A `summarize` operation calculates summary statistics combining all rows of the data:

```
# find the number of "fancy" diamonds (price per carat at least $10000),
summarize(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 `summarize()`:

```
# find the number of "fancy" diamonds (price per carat at least $10000),
# as well as the mean price of a diamond
summarize(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 `summarize` to determine if there are any diamonds of at least one carat that cost less than \$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:

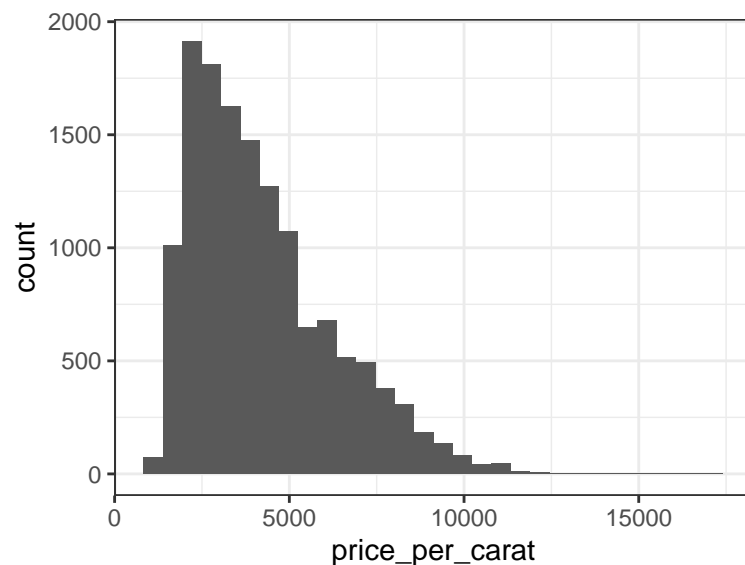
```
diamonds |>
  filter(cut == "Premium") |>      # pipe in the data
  mutate(price_per_carat = price/carat) |> # restrict to premium cut diamonds
  arrange(desc(price_per_carat))      # add price_per_carat variable
                                      # sort based on price_per_carat
```

```
## # A tibble: 13,791 x 11
##   carat cut    color clarity depth table price     x     y     z price_per_carat
##   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>         <dbl>
## 1  1.07 Prem~ D      IF      60.9   58 18279  6.67  6.57  4.03         17083.
## 2  1.01 Prem~ D      IF      61.6   56 16234  6.46  6.43  3.97         16073.
## 3  1.02 Prem~ D      IF      61.5   60 15370  6.52  6.45  3.99         15069.
## 4  1.04 Prem~ D      IF      60.6   56 15671  6.6   6.54  3.98         15068.
## 5  1.02 Prem~ D      IF      61.5   60 15231  6.45  6.52  3.99         14932.
## 6  1.21 Prem~ D      VVS1   60.1   59 17192  6.96  6.88  4.16         14208.
## 7  1.31 Prem~ D      VVS1   62.8   55 17496  7.01  6.95  4.38         13356.
## 8  1.34 Prem~ E      IF      61.8   58 17663  7.15  7.08  4.4          13181.
## 9  1.2  Prem~ D      VVS1   62.1   59 15686  0     0     0          13072.
## 10 1.28 Prem~ E      IF      59.8   59 15806  7.1   7.07  4.24         12348.
## # i 13,781 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
  summarize(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
  summarize(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
## # i 30 more rows
```

Note that the resulting data are still grouped based on `cut`. This is because each call to `summarize()` 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
  summarize(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
```



```
## # i 30 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 `summarize()`.

### 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
## # i 53,930 more rows
```

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

```
max_prices <- diamonds |>           # pipe in the data
  group_by(cut) |>                  # group by cut
  summarize(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 Applying the same operation across multiple variables (`across()`)

In this section, we will learn how to either `summarize()`, `mutate()`, or `filter()` based on several columns at the same time.

### 5.1 `summarize()` based on many columns

Suppose we want to find the median value for each numeric variable in `diamonds`. Here is one way we could do this with `summarize()`:

```
diamonds |>
  summarize(
    carat = median(carat),
    depth = median(depth),
    table = median(table),
    price = median(price),
    x = median(x),
    y = median(y),
    z = median(z)
  )
```

```
## # A tibble: 1 x 7
##   carat depth table price      x      y      z
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1   0.7  61.8   57  2401   5.7   5.71   3.53
```

However, it is best to avoid copying and pasting code. The `across()` function allows us to summarize several variable in the same way:

```
diamonds |>
  summarize(across(
    .cols = c(carat, depth, table, price, x, y, z),
    .fns = median
  ))
```

```
## # A tibble: 1 x 7
##   carat depth table price      x      y      z
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1   0.7  61.8   57  2401   5.7   5.71   3.53
```

To clean this code up even further, we can use the fact that the `.cols` argument is compatible with the `select()` helper functions we discussed before. In this case, to select all numeric columns we can use `where()`:

```
diamonds |>
  summarize(across(.cols = where(is.numeric), .fns = median))
```

```
## # A tibble: 1 x 7
##   carat depth table price      x      y      z
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1   0.7  61.8   57  2401   5.7   5.71   3.53
```

### 5.2 `mutate()` based on many columns

Now, suppose we want to mutate, rather than summarize, several columns. Suppose we want to compute the square of each numeric column. For this, we can use `across()` again, except we have to define our own function to pass to `.fns`, which takes as input a numeric value `x` and outputs its square:

```
diamonds |>
  mutate(across(.cols = where(is.numeric), .fns = function(x) (x * x)))

## # 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 0.0529 Ideal    E     SI2    3782.  3025 106276  15.6  15.8  5.90
## 2 0.0441 Premium  E     SI1    3576.  3721 106276  15.1  14.7  5.34
## 3 0.0529 Good     E     VS1    3238.  4225 106929  16.4  16.6  5.34
## 4 0.0841 Premium  I     VS2    3894.  3364 111556  17.6  17.9  6.92
## 5 0.0961 Good     J     SI2    4007.  3364 112225  18.8  18.9  7.56
## 6 0.0576 Very Good J     VVS2    3944.  3249 112896  15.5  15.7  6.15
## 7 0.0576 Very Good I     VVS1    3881.  3249 112896  15.6  15.8  6.10
## 8 0.0676 Very Good H     SI1    3832.  3025 113569  16.6  16.9  6.40
## 9 0.0484 Fair     E     VS2    4238.  3721 113569  15.0  14.3  6.20
## 10 0.0529 Very Good H     VS1    3528.  3721 114244  16    16.4  5.71
## # i 53,930 more rows
```

If we want to keep the original variables while computing their absolute values, we also need to specify the `.names` argument of `across()`, which specifies how to name each newly created column:

```
diamonds |>
  mutate(
    across(
      .cols = where(is.numeric),
      .fns = function(x)(x * x),
      .names = "{.col}_square"
    )
  )

## # A tibble: 53,940 x 17
##   carat cut      color clarity depth table price      x      y      z carat_square
##   <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      0.0529
## 2 0.21 Premium  E     SI1    59.8   61   326  3.89  3.84  2.31      0.0441
## 3 0.23 Good     E     VS1    56.9   65   327  4.05  4.07  2.31      0.0529
## 4 0.29 Premium  I     VS2    62.4   58   334  4.2   4.23  2.63      0.0841
## 5 0.31 Good     J     SI2    63.3   58   335  4.34  4.35  2.75      0.0961
## 6 0.24 Very Go~ J     VVS2    62.8   57   336  3.94  3.96  2.48      0.0576
## 7 0.24 Very Go~ I     VVS1    62.3   57   336  3.95  3.98  2.47      0.0576
## 8 0.26 Very Go~ H     SI1    61.9   55   337  4.07  4.11  2.53      0.0676
## 9 0.22 Fair     E     VS2    65.1   61   337  3.87  3.78  2.49      0.0484
## 10 0.23 Very Go~ H     VS1    59.4   61   338  4     4.05  2.39      0.0529
## # i 53,930 more rows
## # i 6 more variables: depth_square <dbl>, table_square <dbl>,
## #   price_square <int>, x_square <dbl>, y_square <dbl>, z_square <dbl>
```

### 5.3 filter() based on many columns

Finally, Suppose we wish to find the diamonds that measure at least 6mm in each dimension. We could do this via the `&` operator:

```
diamonds |>
  filter(x > 6 & y > 6 & z > 6)

## # A tibble: 13 x 10
```

```
##      carat cut      color clarity depth table price      x      y      z
##      <dbl> <ord>      <ord> <ord>  <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  3.65 Fair      H      I1      67.1  53 11668  9.53  9.48  6.38
##  2  2      Premium H      SI2      58.9  57 12210  8.09  58.9  8.06
##  3  3.5  Ideal      H      I1      62.8  57 12587  9.65  9.59  6.03
##  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  2.01 Fair      G      SI2      65.6  56 15562  7.89  7.84  6.16
##  7  3.4  Fair      D      I1      66.8  52 15964  9.42  9.34  6.27
##  8  4      Very Good I      I1      63.3  58 15984  10.0  9.94  6.31
##  9  3.67 Premium    I      I1      62.4  56 16193  9.86  9.81  6.13
## 10  4.13 Fair      H      I1      64.8  61 17329  10      9.85  6.43
## 11  5.01 Fair      J      I1      65.5  59 18018  10.7  10.5  6.98
## 12  4.5  Fair      J      I1      65.8  58 18531  10.2  10.2  6.72
## 13  3.51 Premium    J      VS2      62.5  59 18701  9.66  9.63  6.03
```

We can make this code a bit cleaner using the `if_all()` function:

```
diamonds |>
  filter(if_all(.cols = x:z, .fns = function(w)(w > 6)))
```

```
## # A tibble: 13 x 10
##      carat cut      color clarity depth table price      x      y      z
##      <dbl> <ord>      <ord> <ord>  <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  3.65 Fair      H      I1      67.1  53 11668  9.53  9.48  6.38
##  2  2      Premium H      SI2      58.9  57 12210  8.09  58.9  8.06
##  3  3.5  Ideal      H      I1      62.8  57 12587  9.65  9.59  6.03
##  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  2.01 Fair      G      SI2      65.6  56 15562  7.89  7.84  6.16
##  7  3.4  Fair      D      I1      66.8  52 15964  9.42  9.34  6.27
##  8  4      Very Good I      I1      63.3  58 15984  10.0  9.94  6.31
##  9  3.67 Premium    I      I1      62.4  56 16193  9.86  9.81  6.13
## 10  4.13 Fair      H      I1      64.8  61 17329  10      9.85  6.43
## 11  5.01 Fair      J      I1      65.5  59 18018  10.7  10.5  6.98
## 12  4.5  Fair      J      I1      65.8  58 18531  10.2  10.2  6.72
## 13  3.51 Premium    J      VS2      62.5  59 18701  9.66  9.63  6.03
```

If we wanted to find diamonds that are at least 6mm in *any* dimension, we could use `if_any()`:

```
diamonds |>
  filter(if_any(.cols = x:z, .fns = function(w)(w > 6)))
```

```
## # A tibble: 22,443 x 10
##      carat cut      color clarity depth table price      x      y      z
##      <dbl> <ord>      <ord> <ord>  <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.86 Fair      E      SI2      55.1  69  2757  6.45  6.33  3.52
##  2  0.96 Fair      F      SI2      66.3  62  2759  6.27  5.95  4.07
##  3  0.81 Ideal      F      SI2      58.8  57  2761  6.14  6.11  3.6
##  4  0.9  Premium    I      VS2      63      58  2761  6.16  6.12  3.87
##  5  0.8  Ideal      F      SI2      59.9  59  2762  6.01  6.07  3.62
##  6  0.91 Premium    H      SI1      61.4  56  2763  6.09  5.97  3.7
##  7  0.91 Fair      H      SI2      64.4  57  2763  6.11  6.09  3.93
##  8  0.91 Fair      H      SI2      65.7  60  2763  6.03  5.99  3.95
##  9  0.8  Very Good F      SI2      61      57  2772  6.01  6.03  3.67
## 10  1.17 Very Good J      I1      60.2  61  2774  6.83  6.9  4.13
```

```
## # i 22,433 more rows
```

We can combine `if_all()` or `if_any()` with other logical functions.

Exercise: Find all diamonds that are at least 6mm in each direction *and* at least 4 carats.

## 6 References:

- [dplyr cheat sheet](#)
- [Work with Data tutorials](#)
- [R4DS](#) Chapters 4, 13, 14, 15, 17, and 27.2

## 7 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?