UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Advanced Econometrics: Statistical Learning

A (very) brief introduction to the tidyverse

Martin C. Arnold

Winter 2020

You may access these slides online at Netlify (https://tidyintro.netlify.app/) via any browser.

A PDF version is also available on Moodle.

🚀 Please load the tidyverse!🚀

tibble

- `tibble` is the main data structure used in the `tidyverse`. `tibble()` creates a new tibble from scratch.

```
new_tibble   <- tibble(x = 1:3, y = letters[1:3])
```

- `as_tibble()` creates a tibble from an existing object (e.g. a dataframe).

```
a_dataframe  <- data.frame(x = 1:3, y = letters[1:3])
a_tibble     <- as_tibble(a_dataframe)
```

- Everything that works for data frames also works with tibbles because

```
class(a_tibble)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Visualization using `ggplot2`

ggplot2

`ggplot2` in a nutshell:

- R package for data visualization
- implementation of the Grammar of Graphics in R

Some interesting links:

- R for Data Science
- ggplot2: Elegant Graphics for Data Analysis
- R Graphics Cookbook
- Learn the basics @ DataCamp

ggplot2 – a basic example

Exercise: scatterplot

```
library(ggplot2)
data("diamonds")
```

Base R:

```
plot(x = diamonds$carat, y = diamonds$price, pch = 20)
```

ggplot2:

```
ggplot(data = ___, mapping = aes(x = ___, y = ___)) +
  geom_point()
```

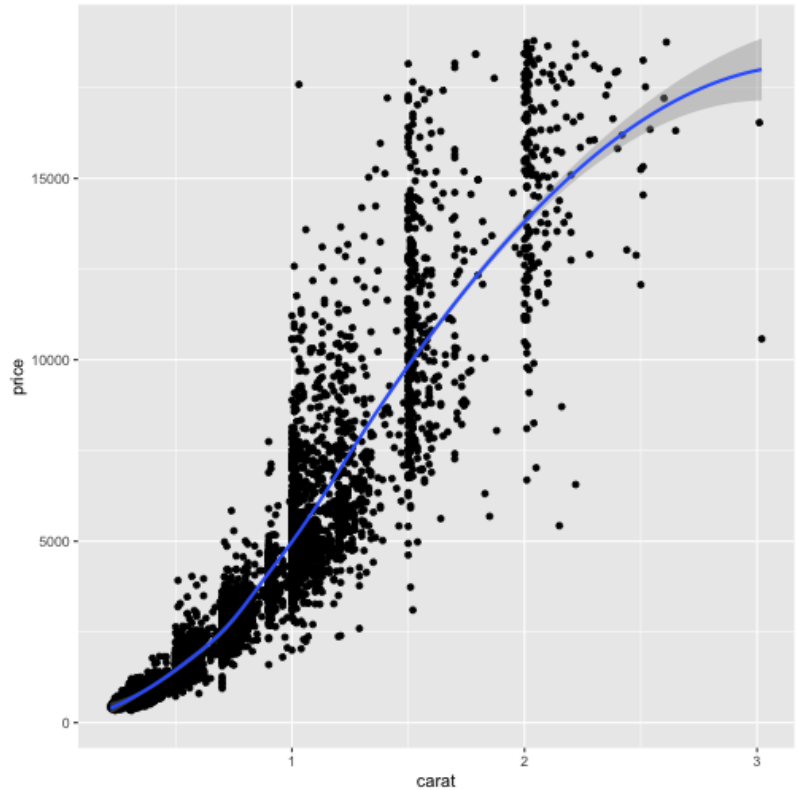
ggplot2 – a basic example – ctd.

Exercise: multiple layers

Add an additional `geom` that performs **lo**cally **e**stimated **s**catterplot **s**moothing.

```
id <- sample(1:nrow(diamonds), 5000)
diamonds <- diamonds[id, ]

ggplot(
  data = diamonds,
  mapping = aes(x = carat, y = price)
  ) +
  geom_point() +
  geom_smooth(___ = "___")
```



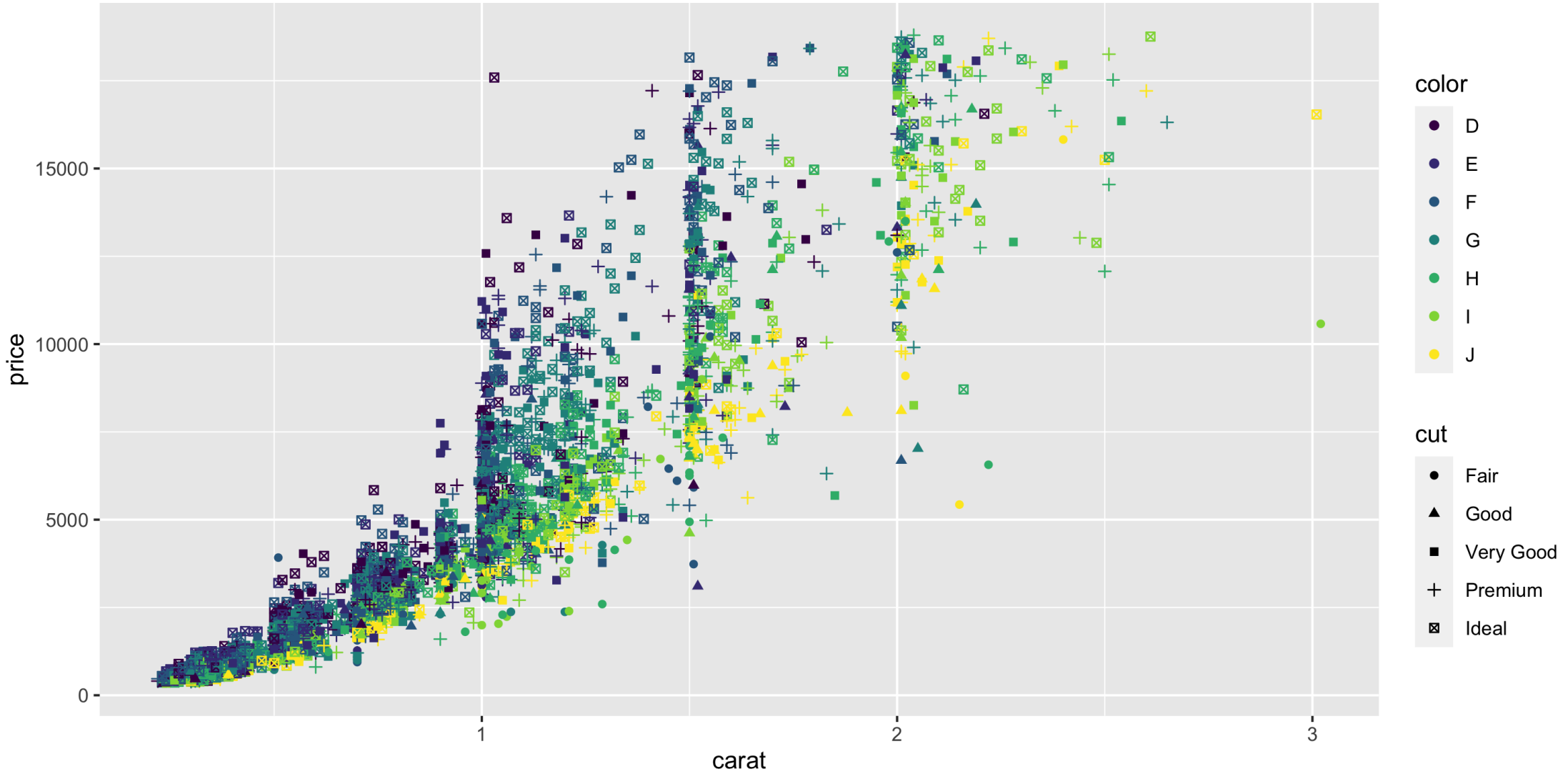
ggplot2 – aesthetics

Each geom has its own set of aesthetics. `geom_point()` requires `x` and `y`, see `?geom_point()`.

Let's add some additional mappings (we say that variables are *mapped* to aesthetics).

```
ggplot(diamonds,
       aes(
         x = carat,
         y = price,
         color = color,
         shape =  cut)
       ) +
       geom_point() +
       geom_smooth(
         method = "loess"
      )
```

ggplot2 – aesthetics – ctd.



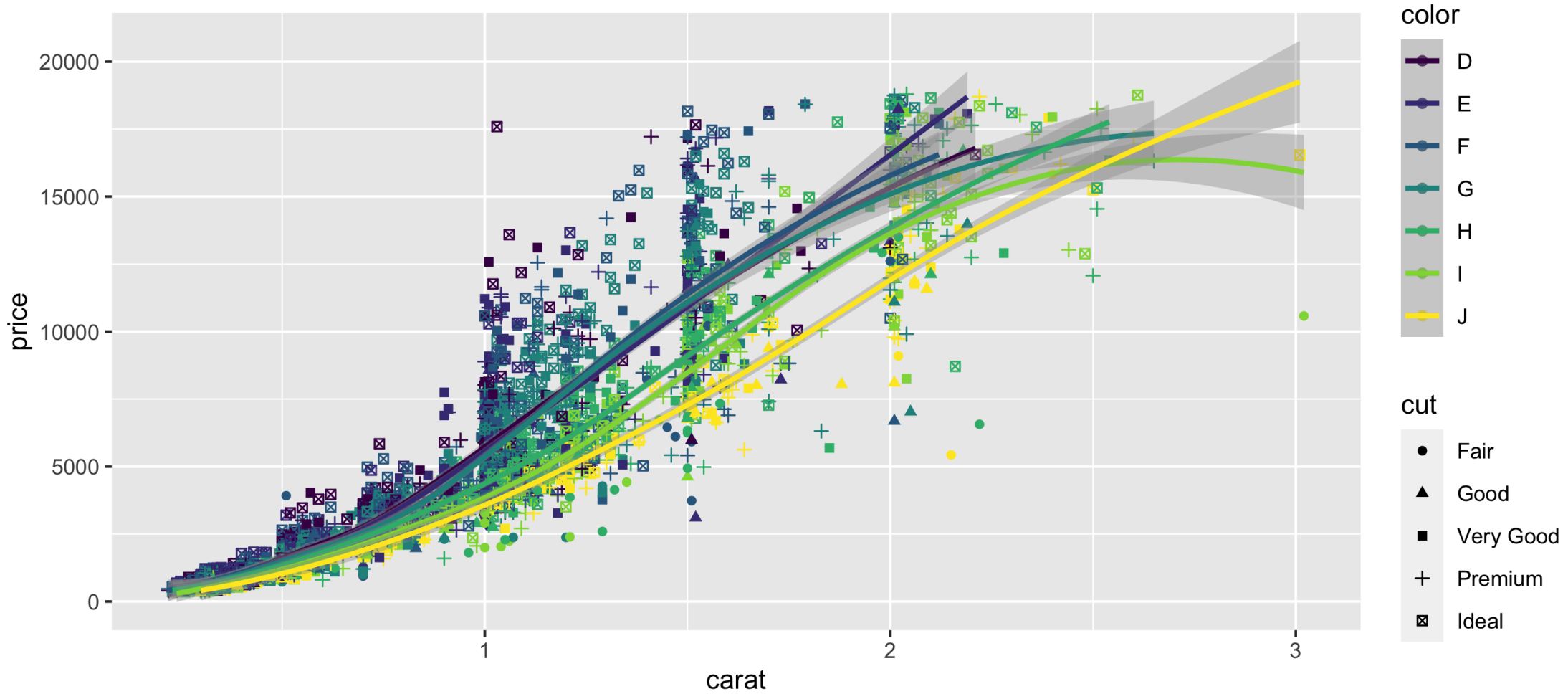
ggplot2 – aesthetics – ctd.

Let's add some loess regression lines:

```
ggplot(diamonds) +
       geom_point(aes(x = carat, y = price, color = color, shape =  cut)) +
       geom_smooth(
         aes(x = carat, y = price, color = color),
         method = "loess"
       )
```

ggplot2 – aesthetics – ctd.

Let's add some complexity:



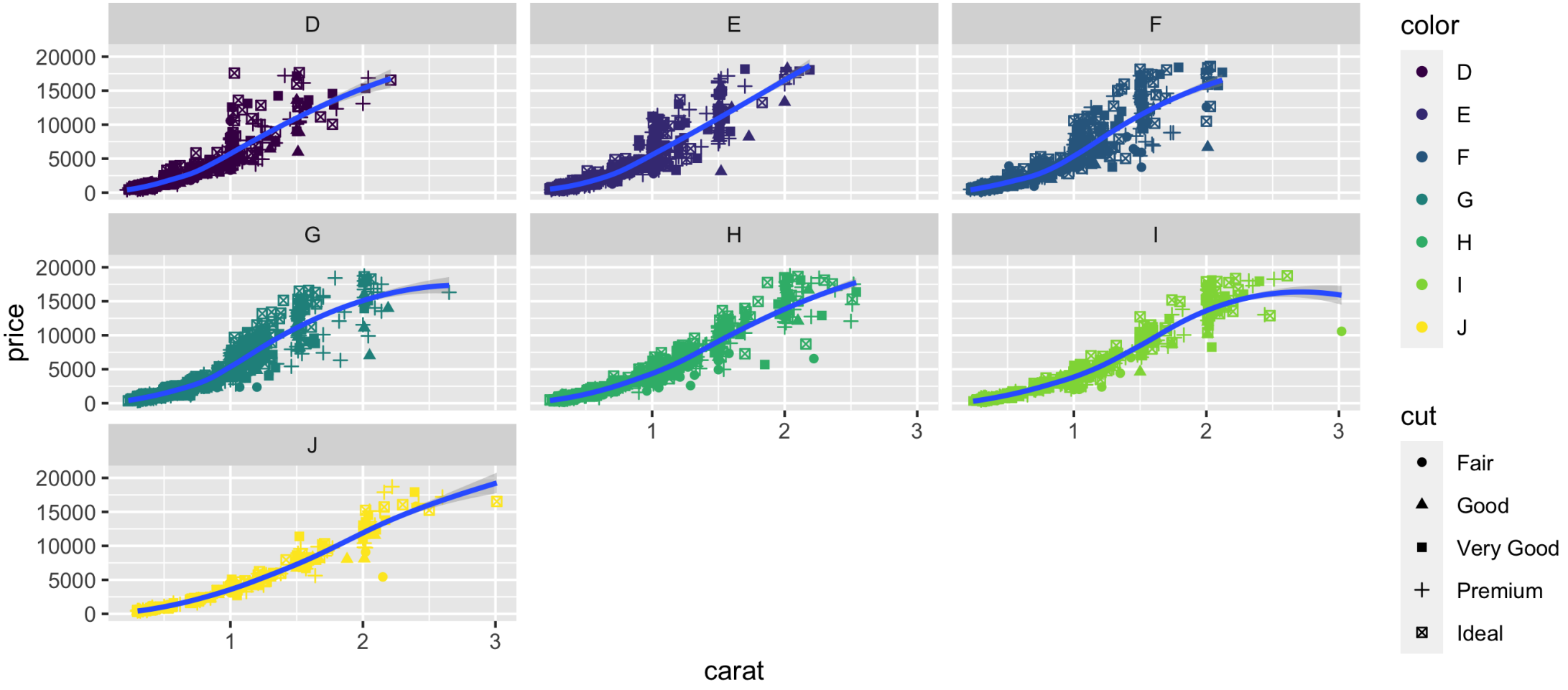
ggplot2 – aesthetics – ctd.

We may use facetting to get a less cluttered result:

```
ggplot(diamonds) +
       geom_point(aes(x = carat, y = price, color = color, shape =  cut)) +
       geom_smooth(
         aes(x = carat, y = price, group = color),
         method = "loess"
       ) +
       facet_wrap(~ color)
```

ggplot2 – aesthetics – ctd.

We may use facetting to get a less cluttered result:



ggplot2

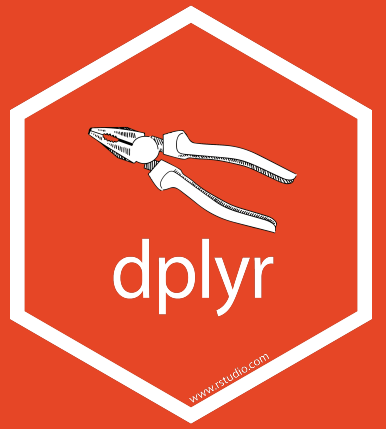
Exercises

1. Use the $mtcars$ data set and plot mpg vs. hp. Add a smoothing line to the plot.
2. Add a smoothing function to the plot for each number of cylinders.
3. Find out how to remove the confidence interval.
4. Use a simple linear regression model and a quadratic regression model for smoothing.

UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Data Wrangling using `dplyr` and `tidyr`



dplyr

- The `dplyr` package is the most important package of the `tidyverse` when it comes to data manipulation and transformation.
- *Data Wrangling* : (data.frame in → transform → data.frame out)
- *Database queries* : more verbose but also easier to understand than in base R
- `dplyr` is fast: the data.frame interface uses *c++*

`dplyr` – verbs

We will often use these verbs in together with the `%>%` operator:

- `select()` picks *variables* based on name
- `filter()` picks *observations* based on value(s)
- `mutate()` adds *variables* that are functions of existing ones
- `summarize()` reduces multiple *values* down to a single summary
- `arrange()` changes ordering of *observations*
- `group_by()` group *observations* by *levels / values*

dplyr — verbs — ctd.

Example: basic dplyr verbs — 1

1. Select variables `clarity`, `carat`, and `cut` from the `diamonds` dataset, then
2. filter for diamonds that have clarity rating `VS2`. Compute the average value of carat for the selected diamonds.
3. Compute the average value of `carat` for the selected diamonds.

```
selected <- select(diamonds, ___, ___, ___)

filtered <- filter(selected, ___ == "VS1")

summarized <- summarize(filtered, mean_carat = ___(___))
```

`dplyr` — verbs — ctd.

Example: basic `dplyr` verbs — 2

Repeat the previous exercise by combining everything using the `%>%` operator.

```
diamonds %>%
  ___(___) %>%
  ___(___) %>%
  ___(___)
```

`dplyr` — verbs — ctd.

Exercises: `dplyr` verbs — 1

Chain the following operations using the `%>%` operator:

1. pick variables `name`, `vore`, `sleep_total`, `brainwt`, and `sleep_rem` from the `msleep` dataset
2. add a new variable `rem_share = sleep_rem / sleep_total`
3. save the result in `msleep_new`

`dplyr` — verbs — ctd.

Exercises: `dplyr` verbs — 2

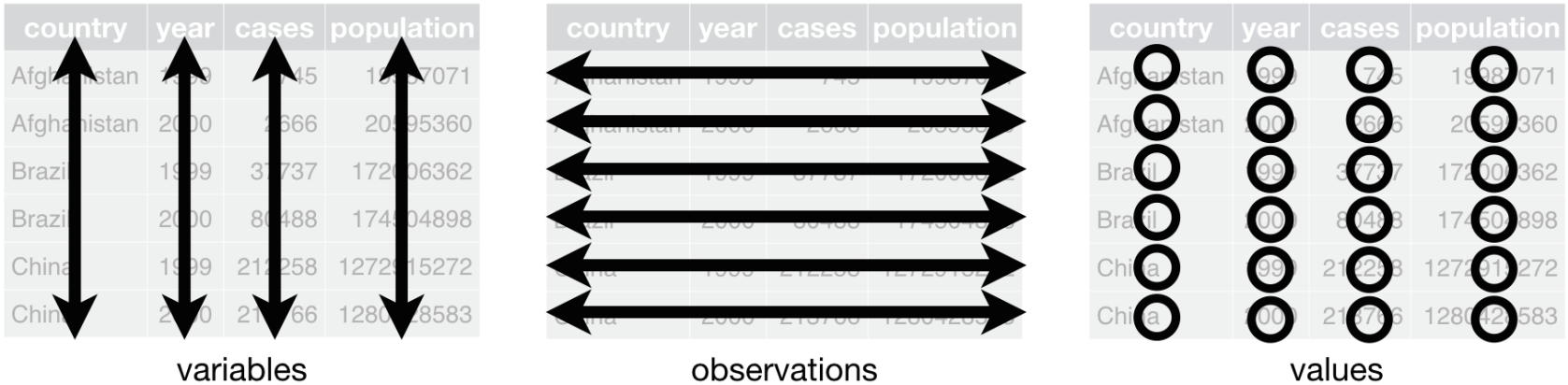
Chain the following operations using the `%>%` operator:

1. sort `msleep_new` from the previous exercise by `rem_share` in descending order
2. group by `vore`
3. drop all rows in `msleep_new` that contain missing values
4. compute the average of `rem_share` per group in `vore`

tidyr

`tidyr` in a nutshell:

A toolkit for getting data in the below format...



... or to convert to *wider* / *longer* formats.

tidyr – ctd.

Numeric values in the below table represent incidents of a rare lung desease.

Is the dataset tidy? Explain why (not).

```
dat <- tibble(
        country = c("Afghanistan", "Germany", "USA"),
        `1999` = c(155, 6, 20),
        `2000` = c(150, 5, 10)
       )
```

tidyr

We can make `dat` *longer* using `tidyr::pivot_longer()`:

```
dat %>% pivot_longer(cols = `1999`:`2000`, names_to = "year", values_to = "cases")
```

```
## # A tibble: 6 x 3
##   country     year  cases
##   <chr>       <chr> <dbl>
## 1 Afghanistan 1999    155
## 2 Afghanistan 2000    150
## 3 Germany     1999      6
## 4 Germany     2000      5
## 5 USA         1999     20
## 6 USA         2000     10
```

UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Iteration using `purrr`

Recap — iteration using loops

```
for(i in <index_set>) {
 # do something
}
```

Loops ...

- make it relatively cumbersome to harness the power of iteration
- are prone to typos/errors that are difficult to identify
- can be *overly flexible*

Iteration — loops vs. `purrr` functionals

Definition: functional

A functional takes a function (along with data) as input and returns a vector as output.

Prominent base R functionals are the `*apply()` functions.

Iteration — loops vs. `purrr` functionals

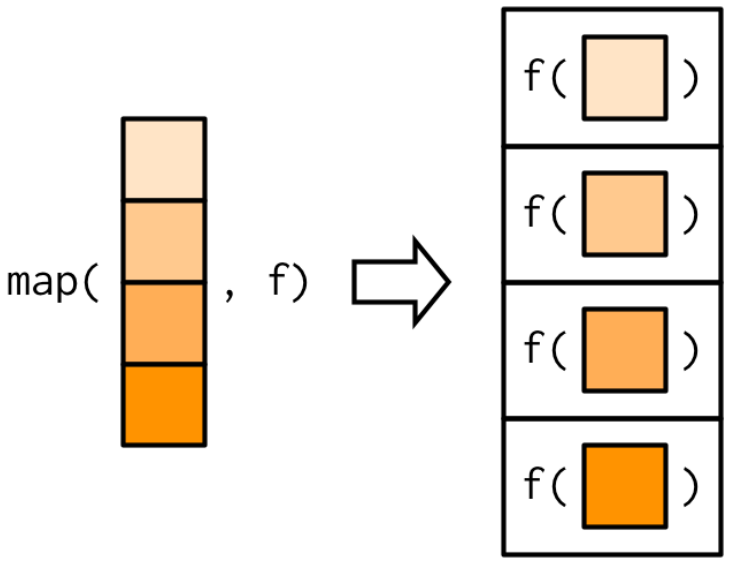
`purrr` functionals

- are tailored to perform specific iterations
- avoid inefficient or faulty iteration
- convey what kind of iteration is done and what kind of output to expect

Iteration with functionals — `purrr::map()`

`map()` takes a vector and a function as arguments. The result is a list of elements obtained by applying the function *element-wise* to the vector.

`map()` — vector in, list out:



Source: Wickham (2019)

Iteration with functionals – `purrr::map()` – ctd.

Example: `map()` – list in, vector out

```r
# input we want to iterate over
x <- 1:3

# function to be applied
Double <- function(z) 2 * z
```

1. Write a `for` loop that applies the function `double()` element-wise to the vector `x` and returns the result in a `list`.
2. Implement a solution using `map()`.

Iteration with functionals — `purrr::map()` — ctd.

Using *anonymous functions* with `map()` is very convenient:

`map()` — using anonymous functions:

```
# return absolute values of entries in x as list
map(x, function(x) abs(x))
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

We can write it even shorter using the *twiddle* operator `~`:

```
map(x, ~ abs(.))
```

Iteration with functionals — `purrr::map()` — ctd.

`map()` — using anonymous functions:

For clearity and brevity will often use `map()` together with the *pipe operator* `%>%`

```
x %>% map(~ abs(.))
```

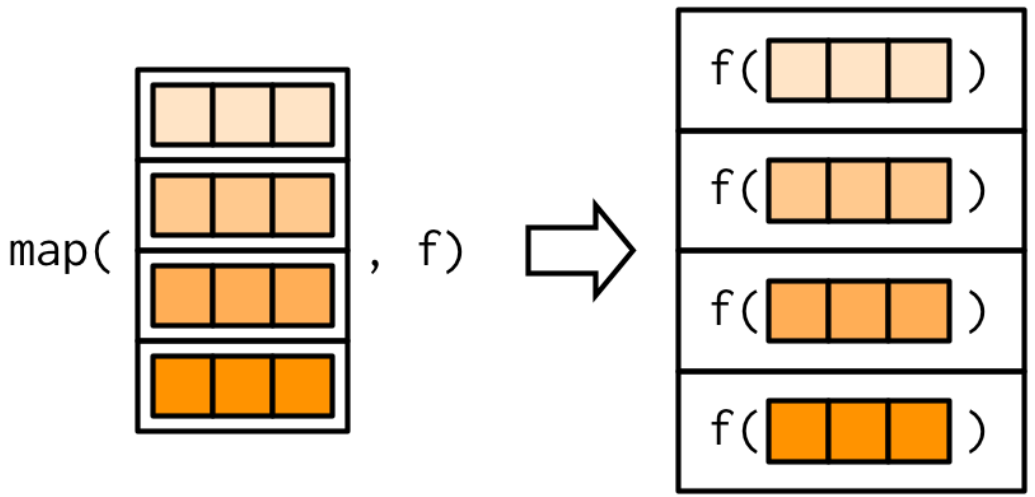
```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

`purrr::map_*()` — producing atomic vectors

`map()` has variants that are more handy if simpler data structures are required as output:

`map_lgl()`, `map_int()`, `map_dbl()`, and `map_chr()` return an atomic vector.

Example: `map_*()` — list in, vector out



Source: Wickham (2019)

`purrr::map_*()` – producing atomic vectors – ctd.

Exercise: standard deviations of numeric variables

Compute the sample standard deviation of all numeric variables in `mtcars` and obtain the result in a numeric vector.

```
___ %>%
  select_if(___) %>%
  map_dbl(___)
```

Thank You!