Introduction to Empirical Analysis in R

Anubhav Jha

2025-09-10

Contents

1 Setting up R and Rstudio

1.1 Installing R and RStudio

This guide will help you install **R** and **RStudio** on your computer.

1.1.1 1. Install R

R is a free, open-source programming language for statistical computing and data analysis.

1. **Go to the R Project website:**
   - https://cran.r-project.org/
2. **Choose your operating system:**
   - **Windows:** Click "Download R for Windows" → "base" → "Download R-x.x.x for Windows" (where `x.x.x` is the latest version).

- **macOS:** Click "Download R for macOS" and download the latest `.pkg` file for your macOS version.
- **Linux:** Click "Download R for Linux" and choose your distribution (Debian, Ubuntu, Fedora, etc.).

3. **Run the installer:**
   - Open the downloaded file and follow the default installation options.

1.1.2 2. Install RStudio Desktop

RStudio is an IDE (Integrated Development Environment) that makes using R much easier.

1. **Go to the RStudio download page:**
   - https://posit.co/download/rstudio-desktop/
2. **Download the free RStudio Desktop (Open Source Edition):**
   - Scroll down to **Download RStudio Desktop for [your OS]**.
   - Choose the installer for your operating system.
3. **Install RStudio:**
   - Open the downloaded file and follow the installation steps.

1.1.3 3. Open RStudio and Test Your Installation

1. Open **RStudio** from your Applications menu or Start Menu.
2. In the **Console** pane (usually bottom left), type:

2 Setting up packages in R

We will be using the following packages. To install them on your R please use the following codes:

- First we create a vector of package names that we need to load/install
- We recover the list of packages that are already installed in your R installation
- We check which packages are not present in your R by using the %in% operator and install the missing ones using install.packages()
  - Difference between in and %in% is important: in iterates a variables x over the values of vector X, while %in% checks if value of x is equal to any value within the vector
- We load all packages using library(). To load all packages within a vector.
  - We use lapply(). lapply(vector,func,...) is a function that sends all elements of vector to func() as arguments. Usable with function of one argument only.
  - lapply() is faster than for-loops due to something called multi-threading (google it in your spare time)

```
# This code will be displayed
# but the output will not be shown
# when you run the code you will see output do not panic
# Step 1: Vector of required packages
```

```r
# Step 1: Vector of required packages
packages <- c(
  "tidyverse",
  "knitr",
  "kableExtra",
  "fixest",
  "AER",
  "sandwich",
  "lmtest",
  "gapminder",
  "scales",
  "ggrepel"
)

# Step 2: Recover the list/vector of packages you already have installed
installed <- rownames(installed.packages())

# Step 3: Install any missing packages
for (pkg in packages) {
  if (!(pkg %in% installed)) {
    print(paste("Missing",pkg," installation in prgoress...","\n"))
    install.packages(pkg)
  }
}

# Step 3: Load the packages
# Load packages
lapply(packages, library, character.only = TRUE)
```

3 Using Tidyverse for cleaning and summarizing Data

First we load all packages that are needed for the analysis. I will introduce and provide some basic examples for important packages as we move along.

```r
# Core
library(tidyverse)
library(knitr)
library(kableExtra)

# Models
library(fixest)     # OLS, FE, clustering, IV in one syntax
library(AER)        # ivreg() for IV/2SLS (classic)
library(sandwich)   # robust/cluster vcov for lm()
library(lmtest)     # coeftest()
```

Load a package called "mtcars". This is a dataset available on all versions of R and it is a good practising dummy dataset. Please do not try to come up with research ideas using this dataset.

```r
data("mtcars")
```

The above code has loaded mtcars in a dataframe with the name mtcars istelf. Type glimpse(mtcars) or head(mtcars). You should see the variables and some entries of the dataset.

- glimpse(mtcars)

```r
glimpse(mtcars)
```

```
## Rows: 32
## Columns: 11
## $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8,~
## $ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8,~
## $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, 16~
## $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, 180~
## $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, 3.92,~
## $ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, 3.~
## $ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, 18~
## $ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,~
## $ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,~
## $ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3,~
## $ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2,~
```

- head(mtcars)

```r
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

3.1 Creating new variables

Now, let's create some new variables and learn how we can use important **tidyverse** functions to clean data (also called *data wrangling* in industry).

We will focus on two variables: `disp` and `cyl`.
- `cyl` is a discrete variable, which we will treat as such.
- `disp` is a continuous variable.

Both are numeric, but `cyl` can take only finitely many values, while `disp` can take any value in a continuous range.

Let's try creating the following variables:

- **`bin_disp`**: takes the value `0` or `1` depending on whether `disp <= mean(disp)` or not.
  To achieve this, we will use several functions and a specific coding style:

- **First**, note the use of the pipe operator `%>%`. This operator sends object `x` into function `fun` to evaluate and return the output.
  - `x %>% fun()` is equivalent to `fun(x)`.
  - If a function `funq` has the syntax `funq(x, a, b, c)`, then `x %>% funq()` returns an error, while `x %>% funq(a, b, c)` works correctly.
  - We typically use this operator when cleaning data frames.
- **Second**, the function `mutate()` is used to edit or create new variables (columns) in data frames, and is very flexible.
- **Third**, the function `if_else(condition, value_if_true, value_if_false)`:
  - This function checks whether a condition holds and returns user-defined values accordingly.
  - The condition can be evaluated over a vector; in that case, the function returns a vector with `value_if_true` where the condition holds and `value_if_false` where it does not.
  - There is another function called `ifelse(condition, value_if_true, value_if_false)`, but it behaves differently: it returns an error if the condition is `NA`, whereas `if_else()` returns `NA`. Sometimes you want an error, and sometimes you want a missing value, depending on the purpose.

```
df <- mtcars %>%
  mutate(bin_disp=if_else(disp>mean(disp),1,0))
head(df)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb bin_disp
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4        0
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4        0
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1        0
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1        1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2        1
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1        0
```

- **`within_bin_disp`**: takes the value `0` or `1` depending on whether `disp <=` the *within-group* mean of `disp`, where a group is defined by `cyl`.
  - To evaluate this within groups, we need to calculate `mean()` for each group.
  - A brute-force method would be to create mini-data frames for each value of `cyl`, calculate the means, and then execute `if_else()` for each of those mini-data frames—this is tedious.
  - Instead, we use `group_by()`, which seamlessly creates these mini-data frames internally. `mutate()` can then be applied to each group separately (or in parallel) without the user

having to write extra code, apart from specifying the grouping variable(s). You can define groups using multiple variables.
  - Always, ungroup the dataframe when using `mutate()`. Don't need to ungroup when using `summarise()`.

```
df <- df %>%
  group_by(cyl) %>%
  mutate(grp_bin_disp=if_else(disp>mean(disp),1,0)) %>%
  ungroup()
head(df)
```

```
## # A tibble: 6 x 13
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb bin_disp
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>
## 1  21       6   160   110  3.9   2.62  16.5     0     1     4     4        0
## 2  21       6   160   110  3.9   2.88  17.0     0     1     4     4        0
## 3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1        0
## 4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1        1
## 5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2        1
## 6  18.1     6   225   105  2.76  3.46  20.2     1     0     3     1        0
## # i 1 more variable: grp_bin_disp <dbl>
```

This dataset is not particularly interesting, but in a real-world example, `cyl` could represent *caste* and `disp` could represent *income*.
- Then, `bin_disp` would capture "elites" in the overall population.
- `within_bin_disp` would capture "elites" within each caste.

3.2 Loading a Dataset and creating new variables

Load the dataset from a CSV file We use `readr` library to load Revenue data called Rev_df: `Rev_df <- read_csv(file = "C:/Users/anubh/Dropbox/UG Emp IO/My Slides/R_markdownfiles/revenue_producti`

Create a new variable y1:

- y1 is defined as the log of Revenue measure 1 minus the log of industry price index
- This transformation is often used to normalize revenue by a price index

```
Rev_df <- Rev_df %>% mutate(y1 = log(R1) - log(P_index))
```

```
library(readr)
Rev_df<-read_csv(file = "C:/Users/anubh/Dropbox/UG Emp IO/My Slides/R_markdownfiles/revenue_pro
```

```
## Rows: 4000 Columns: 11
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (11): firm_id, t, industry, L, K, tfp, Y, P_index, P2, R1, R2
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
Rev_df<-Rev_df %>%
  mutate(y1=log(R1)-log(P_index))
```

We will come-back to this data during lecture-3, the codebook is as followed make yourselves familiar with it:

3.2.1 Variables

- **`firm_id`**
  Firm identifier (1–100).
- **`t`**
  Time period indicator (1–10).
- **`industry`**
  Industry identifier (1–5).
- **`L`**
  Labor input.
- **`K`**
  Capital input.
- **`tfp`**
  Firm-time total factor productivity (TFP).
- **`Y`**
  Output from the Cobb–Douglas production function.
- **`P_index`**
  Industry–time base price index.
- **`P2`**
  Firm-level price under isoelastic demand.
- **`R1`**
  Revenue under price specification 1.
- **`R2`**
  Revenue under price specification 2.

3.2.2 Dataset Dimensions

- 100 firms $\times$ 10 periods $=$ 1000 observations
- 5 industries

3.3 Functions Used in the Summary Statistics Pipeline

3.3.1 %>% (Pipe operator from dplyr)

- Passes the result of one function to the next.
- Improves readability compared to nested calls.

```
1:5 %>%
  mean()
```

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

3.3.2 select() and all_of()

- `select()` chooses columns from a data frame.
- `all_of()` ensures that only variables listed in a character vector are selected.

```
df <- data.frame(a = 1:3, b = 4:6, c = 7:9)
vars <- c("a", "c")

df %>% select(all_of(vars))
```

```
##   a c
## 1 1 7
## 2 2 8
## 3 3 9
```

3.3.3 summarise() and across()

- `summarise()` reduces data to summary values.
- `across()` applies a set of functions across multiple variables at once.

```
df <- data.frame(x = 1:5, y = 6:10)

df %>%
  summarise(across(everything(), mean))
```

```
##   x y
## 1 3 8
```

```
# Returns mean of x and y
```

3.3.4 Statistical Functions for summary statistics

Each function computes a statistic for the chosen variable:
- `mean()` → average
- `sd()` → standard deviation
- `median()` → median
- `quantile(..., 0.25)` → 25th percentile
- `quantile(..., 0.75)` → 75th percentile

- `min()` → minimum value
- `max()` → maximum value
- `sum(!is.na())` → number of non-missing observations

```
x <- c(1, 2, 3, NA, 5)

mean(x, na.rm = TRUE)    # 2.75
```

```
## [1] 2.75
```

```
sd(x, na.rm = TRUE)      # 1.707
```

```
## [1] 1.707825
```

```
quantile(x, 0.25, na.rm = TRUE) # 1.75
```

```
##  25%
## 1.75
```

3.3.5 `.names` Argument in `across()`

- Controls naming of new columns.
- `"{.col}__{.fn}"` means: variable name (`.col`) + double underscore `__` + function name (`.fn`).

```
df <- data.frame(x = 1:3)

df %>% summarise(across(x, list(Mean = mean, SD = sd),
                        .names = "{.col}__{.fn}"))
```

```
##   x__Mean x__SD
## 1       2     1
```

3.3.6 Understanding `pivot_longer()` (from tidyr)

The function `pivot_longer()` reshapes data from **wide format** (values spread across many columns) to **long format** (values stacked in one column with identifiers).

3.3.6.1 Key Arguments:

- **`cols`** → Which columns to reshape.
- **`names_to`** → Name(s) of the new variable(s) that will store the old column names.
- **`values_to`** → Name of the new variable that stores the values.
- **`names_sep`** → How to split column names into multiple parts.
- **`.value`** inside **`names_to`** → Treat part of the column name as a new column heading.

3.3.6.2 Example 1: Basic reshaping

- Columns `math` and `science` are stacked into one column `score`.
- A new column `subject` stores which variable the score came from.

```
# Example 1: Basic reshaping
library(tidyr)
df <- data.frame(id = 1:2, math = c(80,70), science = c(90,85))

df %>% pivot_longer(cols = c(math, science),
                    names_to = "subject",
                    values_to = "score")
```

```
## # A tibble: 4 x 3
##      id subject score
##   <int> <chr>   <dbl>
## 1     1 math       80
## 2     1 science    90
## 3     2 math       70
## 4     2 science    85
```

3.3.6.3 Example 2: Splitting with `names_sep`

- Column names like `L__Mean` and `L__SD` contain both a variable name and a statistic.
- `names_sep = "__"` tells R to split them at the double underscore.
- The result is a `Variable` column and a `Statistic` column.

```
# Example 2: Splitting column names with names_sep
df <- data.frame(id = 1, L__Mean = 2.5, L__SD = 0.5)

df %>% pivot_longer(cols = -id,
                    names_to = c("Variable", "Statistic"),
                    names_sep = "__")
```

```
## # A tibble: 2 x 4
##      id Variable Statistic value
##   <dbl> <chr>    <chr>     <dbl>
## 1     1 L        Mean        2.5
## 2     1 L        SD          0.5
```

3.3.6.4 Example 3: Using `.value` in `names_to`

- When `.value` is used, part of the name becomes **a new column** instead of a row value.
- Columns `Mean` and `SD` are created, while `Variable` stores the variable name (`L`).

```
# Example 3: Using .value in names_to
df <- data.frame(L__Mean = 2.5, L__SD = 0.5)

df %>% pivot_longer(cols = everything(),
                    names_to = c("Variable", ".value"),
                    names_sep = "__")
```

```
## # A tibble: 1 x 3
##   Variable  Mean    SD
##   <chr>    <dbl> <dbl>
## 1 L          2.5   0.5
```

3.3.6.5 Why It Matters in Our Case

After computing summary statistics, our data frame has columns like `L__Mean`, `L__StdDev`, etc.
Using `pivot_longer()` with `names_sep="__"` and `.value` gives us a clean table where:
- Rows correspond to variables (`L`, `K`, `Y`, etc.).
- Columns correspond to summary statistics (`Mean`, `StdDev`, `Median`, etc.).

3.3.7 `kable()` (from knitr)

- Prints tables in a publication-style format for LaTeX, HTML, or Word output.
- `digits = 3` rounds numbers to three decimal places.
- `caption = "Summary Statistics"` adds a title above the table.

```
df <- data.frame(Variable = "x", Mean = 2, SD = 1)
kable(df, digits = 2, caption = "Example Table")
```

Table 1: Example Table

Variable	Mean	SD
x	2	1

3.3.8 Workflow Summary

1. Select relevant variables.
2. Compute summary statistics with `across()`.
3. Rename results safely with `__` separator.

4. Reshape into long format using `pivot_longer()`.

5. Print the results as a clean table with `kable()`.

```r
# Variables to keep (exclude firm_id, t, industry)
vars_to_keep <- c("L", "K", "tfp", "Y", "P_index", "P2", "R1", "R2")

# Compute summary stats with a safe separator
summary_stats <- Rev_df %>%
  select(all_of(vars_to_keep)) %>%
  summarise(across(
    everything(),
    list(
      Mean   = ~mean(., na.rm = TRUE),
      StdDev = ~sd(., na.rm = TRUE),
      Median = ~median(., na.rm = TRUE),
      P25    = ~quantile(., 0.25, na.rm = TRUE),
      P75    = ~quantile(., 0.75, na.rm = TRUE),
      Min    = ~min(., na.rm = TRUE),
      Max    = ~max(., na.rm = TRUE),
      N      = ~sum(!is.na(.))
    ),
    .names = "{.col}__{.fn}"   # <- double underscore!
  ))

# Reshape cleanly
stats_table <- summary_stats %>%
  pivot_longer(everything(),
               names_to = c("Variable", ".value"),
               names_sep = "__")

# Print
kable(stats_table, digits = 3, caption = "Summary Statistics")
```

Table 2: Summary Statistics

Variable	Mean	StdDev	Median	P25	P75	Min	Max	N
L	383.802	1887.017	49.269	11.643	192.244	0.043	77569.734	4000
K	440.183	2307.040	56.112	12.519	240.839	0.061	85194.812	4000
tfp	0.000	0.298	-0.001	-0.196	0.199	-0.939	1.026	4000
Y	11352.848	89677.326	1054.548	230.676	4359.855	0.574	4764450.394	4000
P_index	1.525	1.072	1.394	0.513	2.464	0.247	4.178	4000
P2	0.208	0.125	0.175	0.123	0.256	0.022	1.137	4000
R1	24158.802	237392.892	1195.020	203.599	6402.333	0.183	12654693.728	4000
R2	699.329	2509.836	185.360	58.570	536.016	0.636	105477.656	4000

3.3.9 Exporting Summary Statistics Table to LaTeX

Functions and Options

- **`kable()` (from knitr)**
  Generates a formatted table for LaTeX, HTML, or Word.
  - `digits = 3`: rounds numbers to three decimal places.
  - `caption = "Summary Statistics"`: adds a caption above the table.
  - `format = "latex"`: forces output to LaTeX code.
  - `booktabs = TRUE`: uses `\toprule`, `\midrule`, and `\bottomrule` for professional-quality horizontal lines.
- **`kable_styling()` (from kableExtra)**
  Adds LaTeX styling options.
  - `latex_options = c("striped", "hold_position")`
    - `"striped"`: alternating shaded rows for readability.
    - `"hold_position"`: prevents LaTeX from floating the table away from where it appears in the document.
- **`writeLines()`**
  Saves the generated LaTeX table into a file.
  - First argument: the LaTeX code string.
  - Second argument: the file name (`"summary_statistics.tex"`).

Workflow 1. Create the table with `kable()`.
2. Style it with `kable_styling()`.
3. Save the LaTeX code to a `.tex` file using `writeLines()`.
4. In your LaTeX document, you can include the table with: `\input{summary_statistics.tex}`

```
# Save LaTeX table to a file
latex_table <- kable(stats_table,
                     digits = 3,
                     caption = "Summary Statistics",
                     format = "latex",
                     booktabs = TRUE) %>%
  kable_styling(latex_options = c("striped", "hold_position"))

# Write to .tex file
writeLines(latex_table, "summary_statistics.tex")
```

4 Running OLS in R

First load the dataset again

```
library(readr)
Rev_df<-read_csv(file = "C:/Users/anubh/Dropbox/UG Emp IO/My Slides/R_markdownfiles/revenue_pr
```

```
## Rows: 4000 Columns: 11
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (11): firm_id, t, industry, L, K, tfp, Y, P_index, P2, R1, R2
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
Rev_df<-Rev_df %>%
  mutate(y1=log(R1)-log(P_index))
```

4.1 Ideal Regression:

We implement the two solutions that we studied in Lecture 2. The revenue data already has Y, L, and K. In addition to this it has log(A) as well which is called tfp. First we run the ideal regression:

$$y_{it} = \alpha_L l_{it} + \alpha_K k_{it} + tfp_{it} + e_{it}$$

We use the function `feols()` from the **fixest** package to estimate the production function regression. #### Syntax In R, the most basic regressions are written in the form:

`outcome ~ regressor1 + regressor2 + ...`

- `log(Y)` → dependent variable: logarithm of output.
- `log(L)` → logarithm of labor input.
- `log(K)` → logarithm of capital input.
- `tfp` → total factor productivity, already expressed as log(A).
- `+` → each variable is included separately as a regressor.

The model we run is:

$$\log(Y_{it}) = \alpha_L \cdot \log(L_{it}) + \alpha_K \cdot \log(K_{it}) + tfp_{it} + e_{it}.$$

`data = Rev_df` Specifies that all variables (`Y`, `L`, `K`, `tfp`) are taken from the data frame `Rev_df`. Without this, R would look for variables in the global environment.

Object Assignment `ideal_reg <- feols(...)`

- Saves the regression output to the object `ideal_reg`.
- This object can be used later for displaying results, extracting coefficients, or running hypothesis tests.

Displaying Results with ***`etable()`***

- `etable(ideal_reg)` prints regression results in a clean, publication-style table.

- `digits = 3` ensures values are rounded to three decimal places.

Interpretation - Coefficient on `log(L)` → output elasticity of labor.
- Coefficient on `log(K)` → output elasticity of capital.
- Coefficient on `tfp` → direct effect of productivity (log A) on output.
- Residual `e_it` → captures unobserved factors affecting production.

```
# Load required package
library(fixest)

# Create logged variables (small letter counterparts)
Rev_df <- Rev_df %>%
  mutate(
    log_Y = log(Y),
    log_L = log(L),
    log_K = log(K)
  )

# Ideal regression:
# log(Y_it) = _L * log(L_it) + _K * log(K_it) + tfp_it + e_it
ideal_reg <- feols(log_Y ~ log_L + log_K + tfp, data = Rev_df)

# Display regression results
etable(ideal_reg, digits = 3, title = "Ideal Regression: Cobb-Douglas with TFP")
```

```
##                         ideal_reg
## Dependent Var.:             log_Y
##
## Constant          3.02*** (0.016)
## log_L            0.602*** (0.003)
## log_K            0.394*** (0.003)
## tfp               1.03*** (0.026)
## ________________ _________________
## S.E. type                     IID
## Observations                4,000
## R2                        0.99182
## Adj. R2                   0.99181
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Now suppose we observe only the revenue (given by R1) and not Y.

Do the following two things, first calculate `log_y2=log(R1) - log(P_index)`

```
Rev_df<-Rev_df %>%
  mutate(log_r1 = log(R1),
         log_y2=log(R1) - log(P_index))
```

Then run the following regression

$$log_y2 = \alpha_L \cdot l_{it} + \alpha_K \cdot k_{it} + tfp_{it} + e_{it}.$$

```
ideal_r1_reg <- feols(log_y2 ~ log_L + log_K + tfp, data = Rev_df)

# Display regression results
etable(ideal_reg, ideal_r1_reg, digits = 3)
```

```
##                        ideal_reg     ideal_r1_reg
## Dependent Var.:            log_Y           log_y2
##
## Constant         3.02*** (0.016)  3.01*** (0.018)
## log_L           0.602*** (0.003) 0.603*** (0.003)
## log_K           0.394*** (0.003) 0.394*** (0.003)
## tfp              1.03*** (0.026)  1.02*** (0.029)
## ---------------- ---------------- ----------------
## S.E. type                   IID              IID
## Observations              4,000            4,000
## R2                      0.99182          0.98981
## Adj. R2                 0.99181          0.98981
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Suppose we do not observe Industry index. Then we need industry fixed effects. For that `fixest`'s regression formula syntax is as followed

```
outcome ~ regressor1 + regressor2 + ... | FE1 + FE2 + ...
```

```
ideal_r1_reg2 <- feols(log_r1 ~ log_L + log_K + tfp | industry, data = Rev_df)

# Display regression results
etable(ideal_reg, ideal_r1_reg, ideal_r1_reg2, digits = 3)
```

```
##                        ideal_reg     ideal_r1_reg    ideal_r1_reg2
## Dependent Var.:            log_Y           log_y2           log_r1
##
## Constant         3.02*** (0.016)  3.01*** (0.018)
## log_L           0.602*** (0.003) 0.603*** (0.003) 0.597*** (0.020)
## log_K           0.394*** (0.003) 0.394*** (0.003) 0.387*** (0.020)
## tfp              1.03*** (0.026)  1.02*** (0.029)  1.11*** (0.179)
## Fixed-Effects:   ---------------- ---------------- ----------------
## industry                    No               No              Yes
## ---------------- ---------------- ---------------- ----------------
## S.E. type                   IID              IID     by: industry
## Observations              4,000            4,000            4,000
## R2                      0.99182          0.98981          0.98661
## Within R2                    --               --          0.98029
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
wrong_r1_reg2 <- feols(log_r1 ~ log_L + log_K +  tfp, data = Rev_df)

# Display regression results
etable(ideal_reg, ideal_r1_reg, ideal_r1_reg2, wrong_r1_reg2, digits = 3)
```

```
##                         ideal_reg     ideal_r1_reg    ideal_r1_reg2
## Dependent Var.:             log_Y           log_y2           log_r1
##
## Constant          3.02*** (0.016)  3.01*** (0.018)
## log_L            0.602*** (0.003) 0.603*** (0.003) 0.597*** (0.020)
## log_K            0.394*** (0.003) 0.394*** (0.003) 0.387*** (0.020)
## tfp               1.03*** (0.026)  1.02*** (0.029)  1.11*** (0.179)
## Fixed-Effects:  ----------------- ----------------- -----------------
## industry                       No               No              Yes
## --------------- ----------------- ----------------- -----------------
## S.E. type                     IID              IID     by: industry
## Observations                4,000            4,000            4,000
## R2                        0.99182          0.98981          0.98661
## Within R2                      --               --          0.98029
##
##                      wrong_r1_reg2
## Dependent Var.:             log_r1
##
## Constant         -0.354*** (0.027)
## log_L             0.822*** (0.005)
## log_K              1.06*** (0.005)
## tfp               -4.25*** (0.043)
## Fixed-Effects:  ------------------
## industry                       No
## --------------- ------------------
## S.E. type                     IID
## Observations                4,000
## R2                        0.98307
## Within R2                      --
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

All feasible regressions: when tfp is not observable

```
wrong_y_reg3<- feols(log_Y ~ log_L + log_K , data = Rev_df)
wrong_r1_reg3 <- feols(log_y2 ~ log_L + log_K , data = Rev_df)
wrong_r1_reg4 <- feols(log_r1 ~ log_L + log_K| industry , data = Rev_df)
wrong_r1_reg5 <- feols(log_r1 ~ log_L + log_K , data = Rev_df)

# Display regression results
etable(ideal_reg, wrong_y_reg3, wrong_r1_reg3, wrong_r1_reg4, wrong_r1_reg5, digits = 3)
```

```
##                         ideal_reg     wrong_y_reg3    wrong_r1_reg3
## Dependent Var.:             log_Y           log_Y           log_y2
```

```
##
## Constant          3.02*** (0.016)  2.45*** (0.008)  2.45*** (0.009)
## log_L            0.602*** (0.003) 0.672*** (0.002) 0.673*** (0.003)
## log_K            0.394*** (0.003) 0.468*** (0.002) 0.468*** (0.003)
## tfp               1.03*** (0.026)
## Fixed-Effects:  ---------------- ---------------- ----------------
## industry                      No               No               No
## --------------- ---------------- ---------------- ----------------
## S.E. type                    IID              IID              IID
## Observations               4,000            4,000            4,000
## R2                       0.99182          0.98864          0.98672
## Within R2                     --               --               --
##
##                     wrong_r1_reg4    wrong_r1_reg5
## Dependent Var.:            log_r1           log_r1
##
## Constant                              1.99*** (0.022)
## log_L            0.689*** (0.014) 0.532*** (0.006)
## log_K            0.479*** (0.014) 0.748*** (0.006)
## tfp
## Fixed-Effects:  ---------------- ----------------
## industry                     Yes               No
## --------------- ---------------- ----------------
## S.E. type            by: industry              IID
## Observations                4,000            4,000
## R2                        0.98651          0.94230
## Within R2                 0.98014               --
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

To export etable into latex we use the following arguments. Set `tex=TRUE` and provide path to where you wish to save the `.tex` file using the argument `file=[pathname]`.

```r
# Run regressions
wrong_y_reg3  <- feols(log_Y ~ log_L + log_K , data = Rev_df)
wrong_r1_reg3 <- feols(log_y2 ~ log_L + log_K , data = Rev_df)
wrong_r1_reg4 <- feols(log_r1 ~ log_L + log_K | industry , data = Rev_df)
wrong_r1_reg5 <- feols(log_r1 ~ log_L + log_K , data = Rev_df)

# Export regression results to LaTeX
etable(
  ideal_reg, wrong_y_reg3, wrong_r1_reg3, wrong_r1_reg4, wrong_r1_reg5,
  digits = 3,
  tex = TRUE,
  file = "C:/Users/anubh/Dropbox/UG Emp IO/My Slides/R_markdownfiles/regression_revenue_results
)
```

5 Using ggplot2 and tidyverse to visualize data

5.1 Data loading and preparation

- `data(gapminder)`
  Loads the `gapminder` dataset into the environment (from the **gapminder** package).
- `as_tibble()` **(from** `tibble`**)**
  Converts a data frame (or similar) to a **tibble**: a modern, print-friendly data frame.
- `filter()` **(from** `dplyr`**)**
  Keeps rows that satisfy a logical condition.
  Example here: `filter(country == "India")` keeps only India's observations.
- `readr` **functions such as** `read_csv()`: Used for loading datasets stored in your harddrive. We will discuss this while discussing instrument-variable estimation as we need it there.

5.2 Plotting with ggplot2

- `ggplot(data, aes(...))`
  Initializes a ggplot object with a dataset and aesthetic mappings (`aes`).
  In the snippets:
  - `aes(lifeExp)` maps the x-axis to life expectancy.
  - `aes(gdpPercap)` maps the x-axis to GDP per capita.
- `geom_histogram()`
  Draws a histogram of the x variable.
  - `aes(y = after_stat(density))`: rescales the histogram bars to show **density** rather than counts.
  - `bins = 30`: sets the number of bins.
  - `alpha = 0.6`: sets transparency (0 = fully transparent, 1 = opaque).
- `geom_density(linewidth = 1)`
  Overlays a kernel density estimate curve.
  - `linewidth`: thickness of the density line.
- `scale_x_continuous(labels = scales::dollar_format(prefix = "$"))`
  Formats the x-axis tick labels as currency (e.g., `$1,234`). Useful when plotting `gdpPercap`.
- `labs(title = ..., x = ..., y = ...)`
  Adds labels to the plot:
  - `title`: the plot title (here built with `paste0`).
  - `x`, `y`: axis labels.
- `paste0()` **(base R)**
  Concatenates strings **without** separators. Used to dynamically build titles (e.g., inserting `ref_year`).

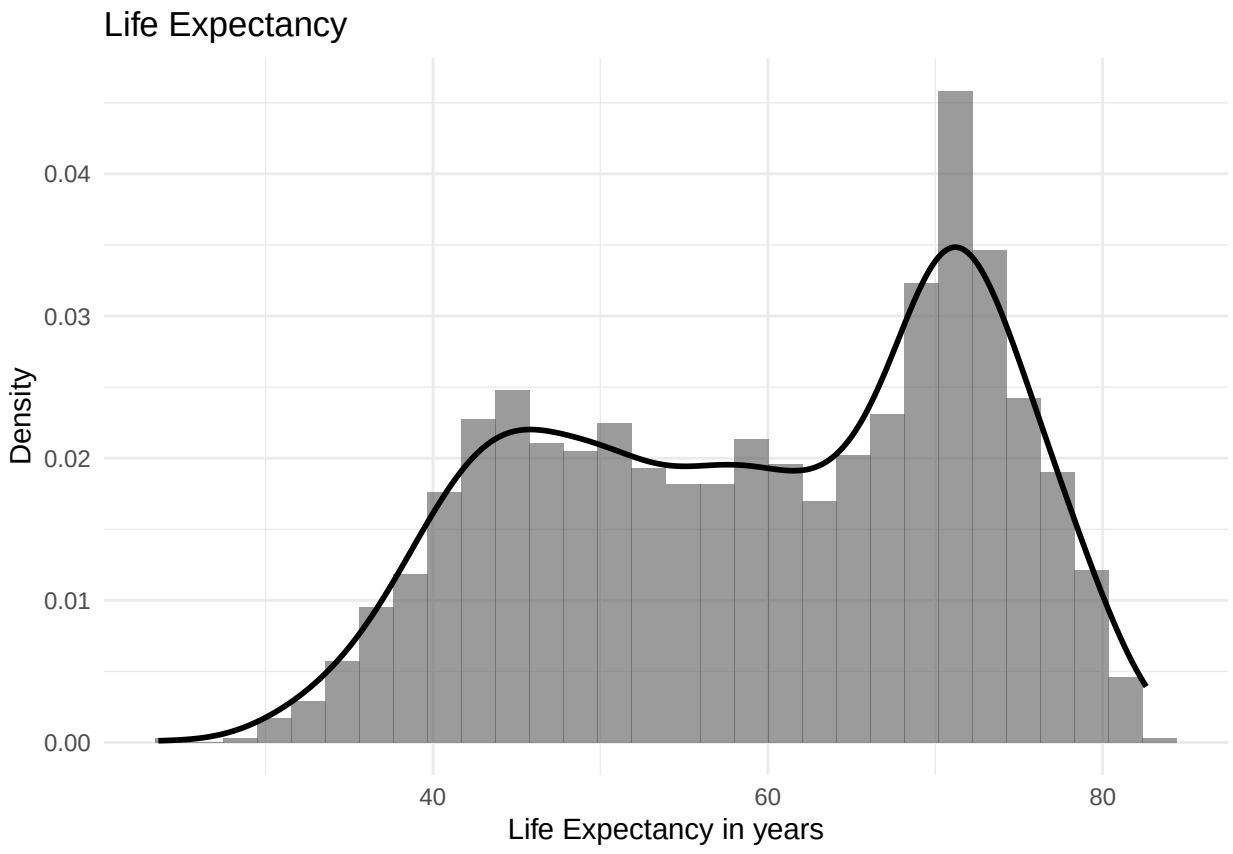
- `theme_minimal()`
  Applies a clean, minimal theme to the plot.

5.2.1 Plot objects

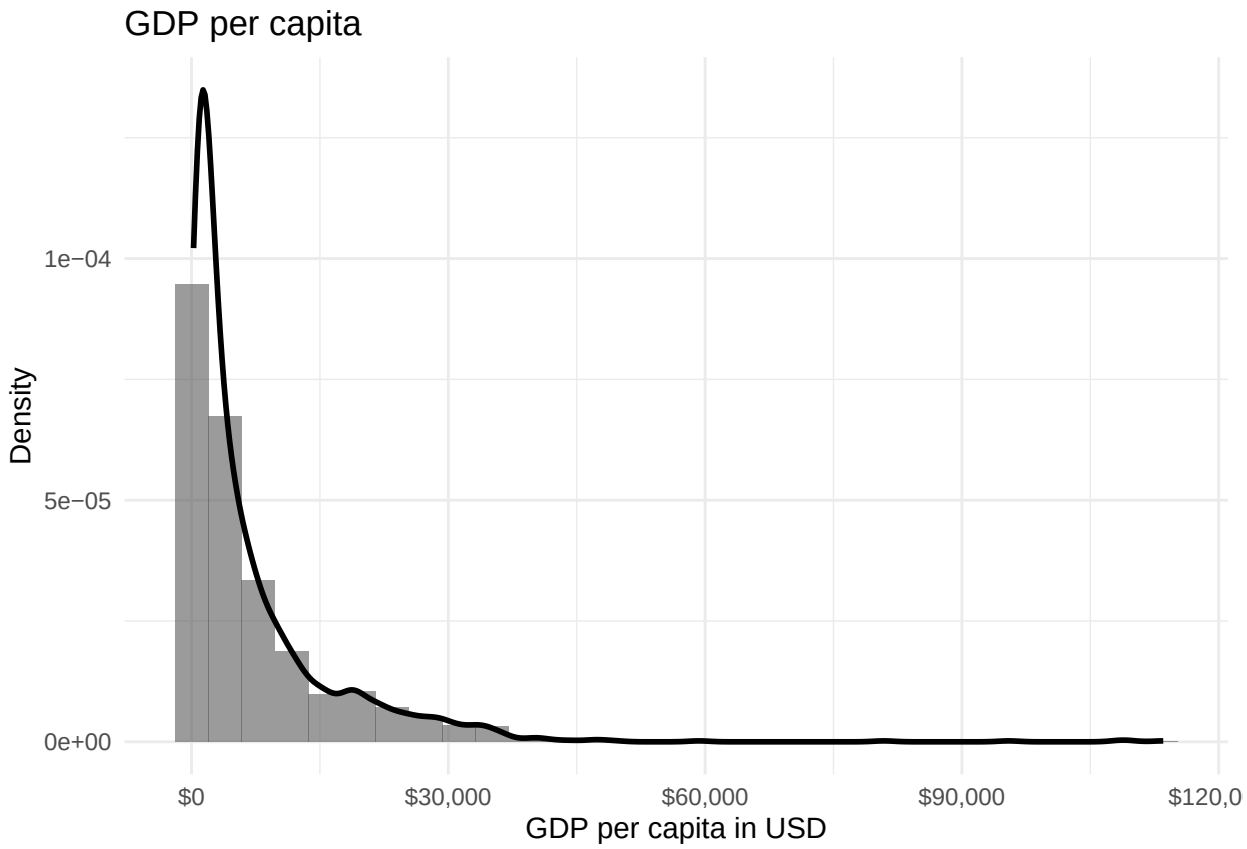
- **Assigning plots to variables**
  - `p_life_expectancy <- ggplot(...) + ...` stores a plot object in `p_life_expectancy`.
  - Printing the object name (`p_life_expectancy`) renders the plot in the output.

```
data(gapminder)
df <- gapminder %>% as_tibble()

# Convenience subsets
india   <- df %>% filter(country == "India")
```

```
# Overall: histogram + density, with formatting x-axis
p_life_expectancy <- ggplot(df, aes(lifeExp)) +
  geom_histogram(aes(y = after_stat(density)), bins = 30, alpha = 0.6) +
  geom_density(linewidth = 1) +
  labs(title = paste0("Life Expectancy"),
       x = "Life Expectancy in years", y = "Density") +
  theme_minimal()

p_life_expectancy
```



```r
p_gdp_percapita <- ggplot(df, aes(gdpPercap)) +
  geom_histogram(aes(y = after_stat(density)), bins = 30, alpha = 0.6) +
  geom_density(linewidth = 1) +
  scale_x_continuous(labels = scales::dollar_format(prefix = "$")) +
  labs(title = paste0("GDP per capita"),
       x = "GDP per capita in USD", y = "Density") +
  theme_minimal()

p_gdp_percapita
```



5.3 Plotting histogram and densities by group

1. `facet_grid(rows ~ cols, scales = "free_y")` - **Purpose:** Creates a grid of subplots based on the levels of one or two categorical variables.

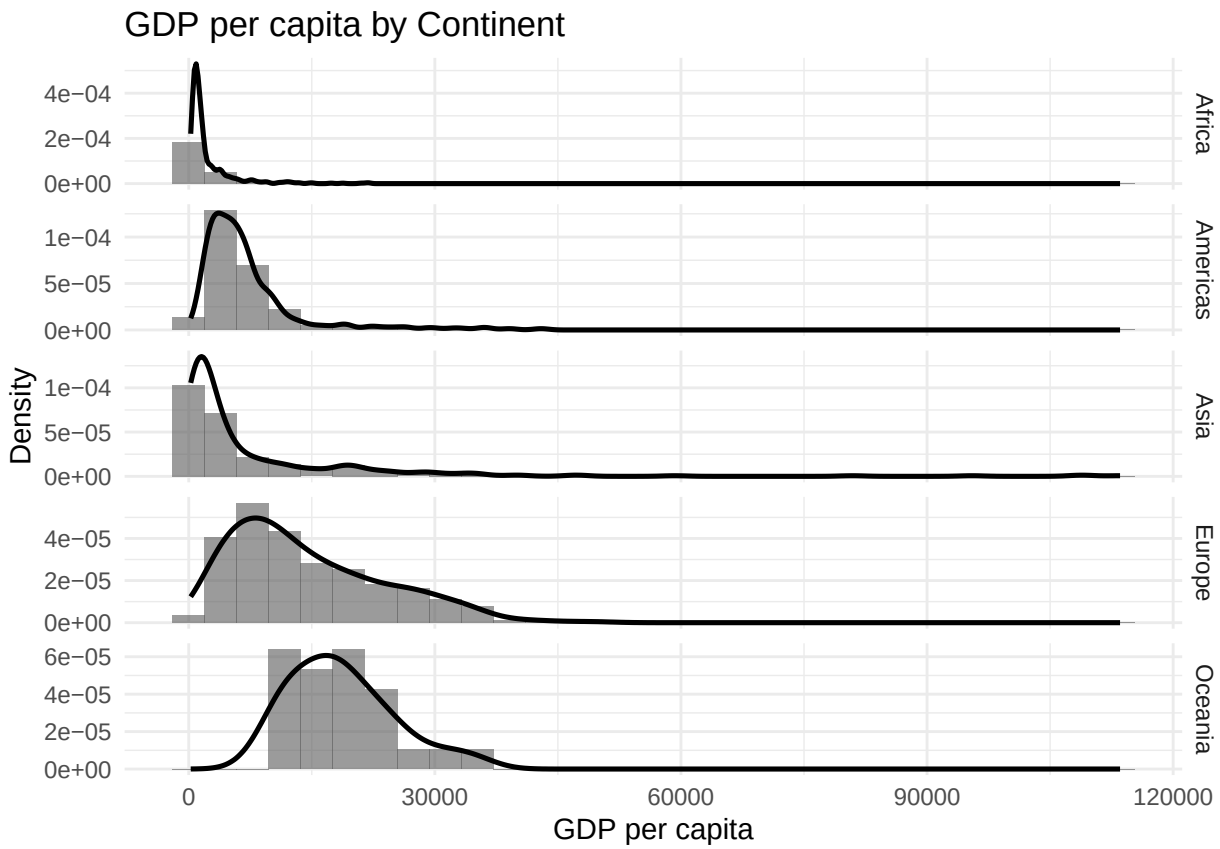
- **Usage in this code:**
  - `facet_grid(continent ~ .)` → Rows represent continents, columns are not used (`.` placeholder).
  - `facet_grid(continent ~ year)` → Rows represent continents, columns represent years.
- `scales = "free_y"` allows each facet (subplot) to have its own y-axis scale. This is useful when the density or counts differ greatly across groups.

2. `df[df$year %in% c(2007, 1952), ]` - **Purpose:** Base R syntax for subsetting rows of a data

frame.

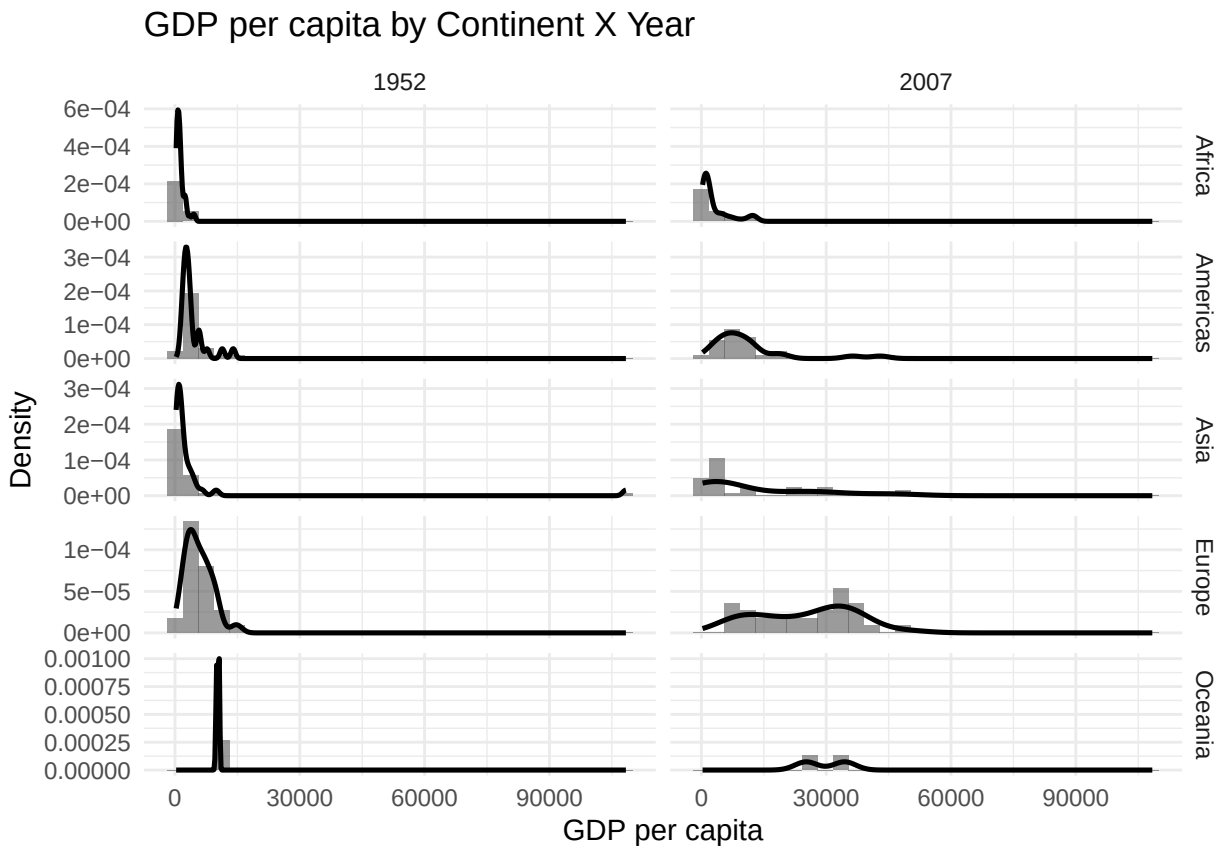
- **Explanation:**
  - `df$year %in% c(2007, 1952)` returns a logical vector `TRUE` if the `year` is 2007 or 1952.
  - `df[ ... , ]` keeps rows where the condition is `TRUE`.
  - This is equivalent to `filter(df, year %in% c(2007, 1952))` from **dplyr**.

```r
# By continent: facets
p_by_group <- ggplot(df, aes(gdpPercap)) +
  geom_histogram(aes(y = after_stat(density)), bins = 30, alpha = 0.6) +
  geom_density(linewidth = 0.9) +
  facet_grid(continent ~ ., scales = "free_y")  +
  labs(title = paste0("GDP per capita by Continent"),
       x = "GDP per capita", y = "Density") +
  theme_minimal()

p_by_group
```




```r
# By continent x year: facets
p_by_group_year <- ggplot(df[df$year %in% c(2007,1952),], aes(gdpPercap)) +
  geom_histogram(aes(y = after_stat(density)), bins = 30, alpha = 0.6) +
  geom_density(linewidth = 0.9) +
  facet_grid(continent ~ year, scales = "free_y")  +
  labs(title = paste0("GDP per capita by Continent X Year"),
       x = "GDP per capita", y = "Density") +
```

```
  theme_minimal()

p_by_group_year
```



5.4 Plotting Time-Series using `geom_line()`

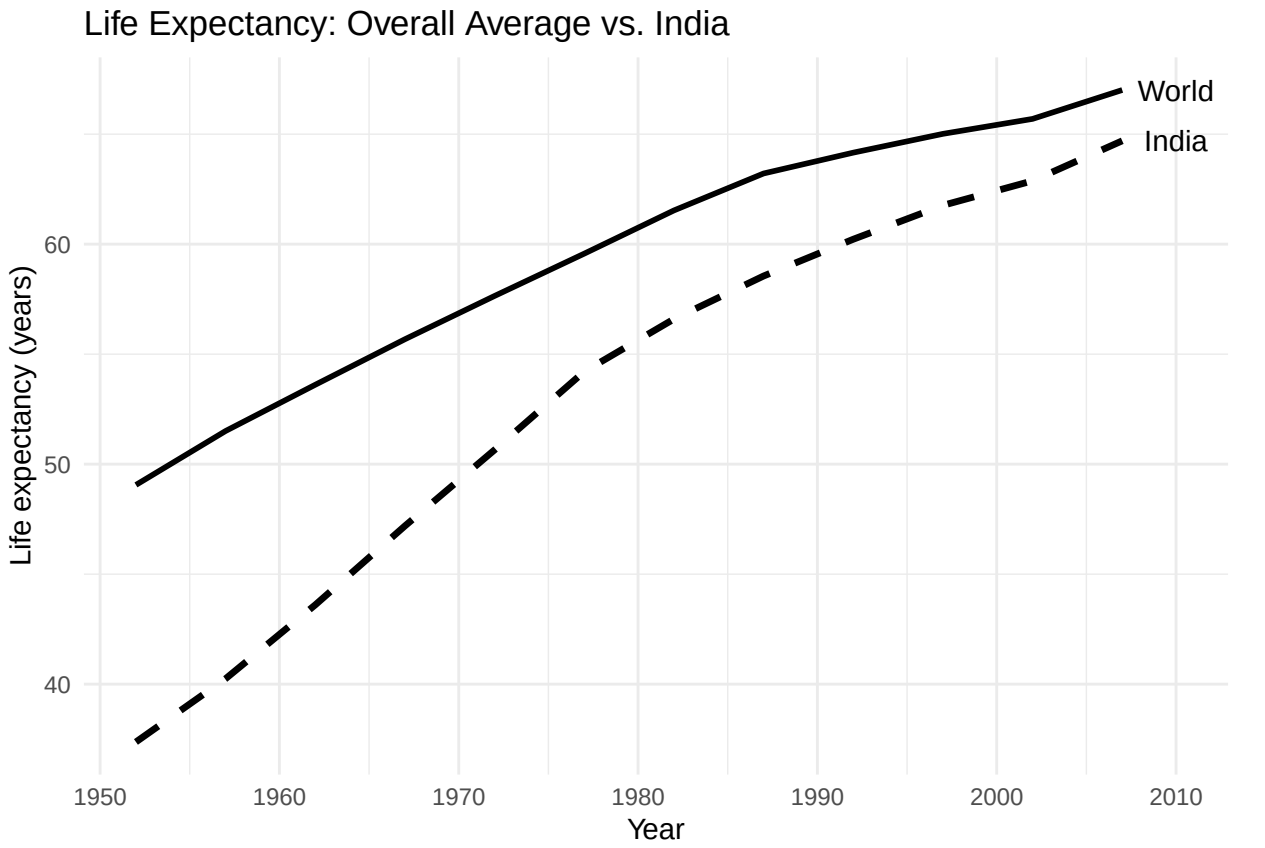
- `geom_line()`
  - Creates a line plot where data points are connected by straight lines.
  - Common arguments:
    - `data`: the dataframe to be plotted.
    - `aes(x, y)`: specifies the x-axis and y-axis variables.
    - `linewidth`: controls the thickness of the line.
    - `linetype`: changes the style of the line (e.g., `"solid"`, `"dashed"`, `"dotted"`).
    - `alpha`: controls the transparency of the line (0 = fully transparent, 1 = fully opaque).
  - In your code:
    - Used to draw life expectancy over time, both for the **world average** and for **India**

(with a dashed line for India).

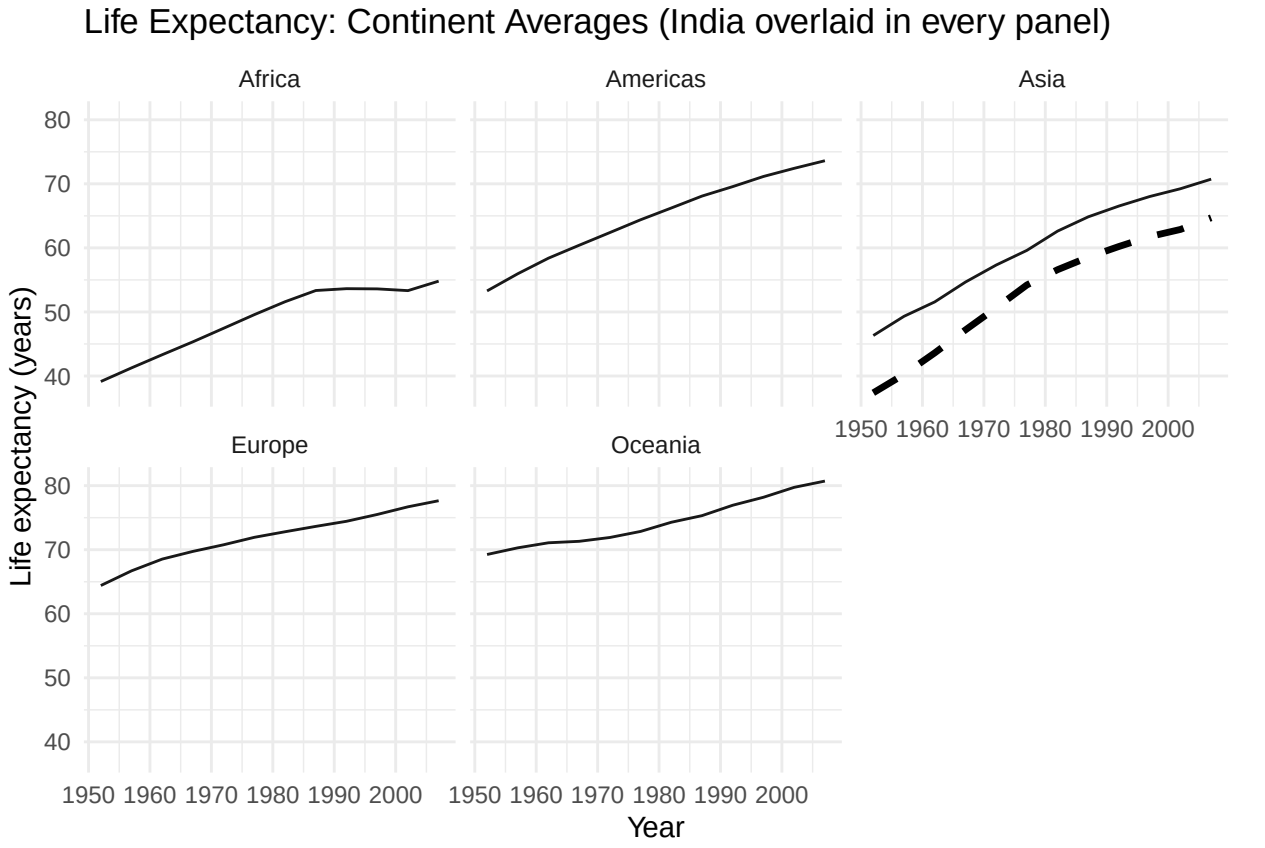
- `geom_text_repel()` (from the `ggrepel` package)
  - Places labels (`geom_text`) on a plot but with *repelling forces*, so labels do not overlap each other or the data points.
  - Common arguments:
    - `data`: the subset of data used for labeling.
    - `aes(x, y, label)`: specifies x and y positions along with the text label.
    - `nudge_x` / `nudge_y`: moves the label slightly away from the point for readability.
    - `direction`: controls the direction labels can move (e.g., `"x"`, `"y"`, `"both"`).
    - `segment.color`: color of the line segment connecting the text to the data point (set to `NA` if you don't want the segment).
  - In your code:
    - Used to annotate India and World at the **last available year** with clean, non-overlapping labels.
- `facet_wrap(~ continent)`
  - Splits the plot into multiple subplots ("facets") arranged in a grid, one for each level of the grouping variable (`continent` in this case).
  - Useful for showing the same type of visualization separately for different groups.
  - In your code:
    - Shows a separate line chart of life expectancy over time for each continent, while overlaying India's trajectory in all panels.

```
# Overall average per year
avg_overall <- gapminder %>% group_by(year) %>% summarise(lifeExp = mean(lifeExp, na.rm = TRUE)

# Average per continent per year
avg_by_cont <- gapminder %>% group_by(continent,year) %>% summarise(lifeExp = mean(lifeExp, na

# Plot: overall average + India
p_ts_overall <- ggplot() +
  geom_line(data = avg_overall, aes(x=year, y=lifeExp), linewidth = 1) +
  geom_line(data = india, aes(x=year, y=lifeExp), linewidth = 1.2,linetype="dashed") +
  ggrepel::geom_text_repel(
    data = india %>% filter(year == max(year)),
    aes(year, lifeExp, label = "India"),
    nudge_x = 3, direction = "y", segment.color = NA
  ) +
  ggrepel::geom_text_repel(
    data = avg_overall %>% filter(year == max(year)),
    aes(year, lifeExp, label = "World"),
```

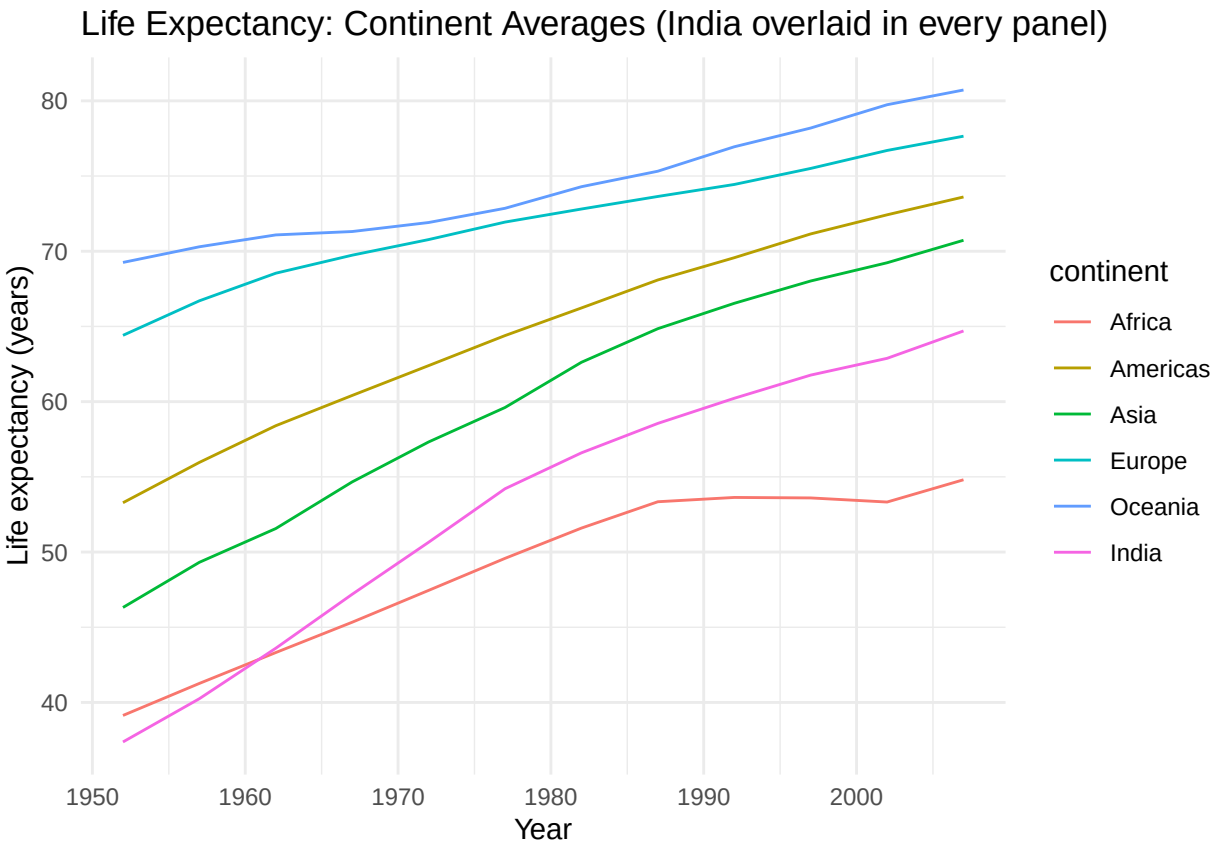
```
    nudge_x = 3, direction = "y", segment.color = NA
  ) +
  labs(title = "Life Expectancy: Overall Average vs. India",
       x = "Year", y = "Life expectancy (years)") +
  theme_minimal()

# Plot: by continent averages + India
p_ts_by_group <- ggplot() +
  geom_line(data = avg_by_cont, aes(x=year, y=lifeExp, group = continent), alpha = 0.9) +
  facet_wrap(~ continent) +
  geom_line(data = india, aes(year, lifeExp), linewidth = 1.2, linetype="dashed") +
  labs(title = "Life Expectancy: Continent Averages (India overlaid in every panel)",
       x = "Year", y = "Life expectancy (years)") +
  theme_minimal()

# Plot: by continent averages + India
p_ts_by_color <- ggplot() +
  geom_line(data = avg_by_cont, aes(x=year, y=lifeExp, color = continent)) +
  geom_line(data = india, aes(year, lifeExp, color="India")) +
  labs(title = "Life Expectancy: Continent Averages (India overlaid in every panel)",
       x = "Year", y = "Life expectancy (years)") +
  theme_minimal()

p_ts_overall
```


Life Expectancy: Overall Average vs. India
World
India
Life expectancy (years)
60
50
40
1950
1960
1970
1980
1990
2000
2010
Year

```
p_ts_by_group
```


Life Expectancy: Continent Averages (India overlaid in every panel)
Africa
Americas
Asia
Europe
Oceania
Life expectancy (years)
80
70
60
50
40
1950 1960 1970 1980 1990 2000
Year

```
p_ts_by_color
```



5.5 Scatter Plots and Fitted Lines for correlations

- `geom_point()`
  - Adds a scatter plot layer to the graph, plotting individual observations as points.
  - Common arguments:
    - `alpha`: controls point transparency (useful for reducing overplotting when there are many observations).
    - `data`: allows overlaying a different dataset (e.g., highlighting India on top of the overall scatter).
  - In your code:
    - Used to plot the relationship between GDP per capita and life expectancy.
    - Transparency (`alpha = 0.45`) makes the global scatter less cluttered, while India's points are highlighted separately.
- `geom_smooth()`
  - Adds a smoothed conditional mean (trend line) to the plot.
  - Common arguments:
    - `method`: specifies the smoothing method (e.g., `"lm"` for ordinary least squares regression, `"loess"` for local regression).

        * `se`: logical, whether to display the confidence interval around the fitted line (`FALSE` removes it).

        * `linewidth`: controls the thickness of the line.

    - In your code:
        * Fits an OLS regression line to visualize the relationship between GDP per capita and life expectancy.

        * Separate regression lines are drawn for each continent when `facet_wrap` is applied.

- **`scale_x_log10()`**
    - Transforms the x-axis to a base-10 logarithmic scale.

    - Useful when data spans several orders of magnitude, as in GDP per capita.

    - Can be combined with label formatting functions from the `scales` package.

    - In your code:
        * Used with `scales::dollar_format(prefix = "$")` to display GDP per capita on a log scale with dollar formatting, improving readability.
- **`facet_wrap(~ continent, scales = "free")`**
    - Creates multiple panels (facets) arranged in a grid, one for each continent.

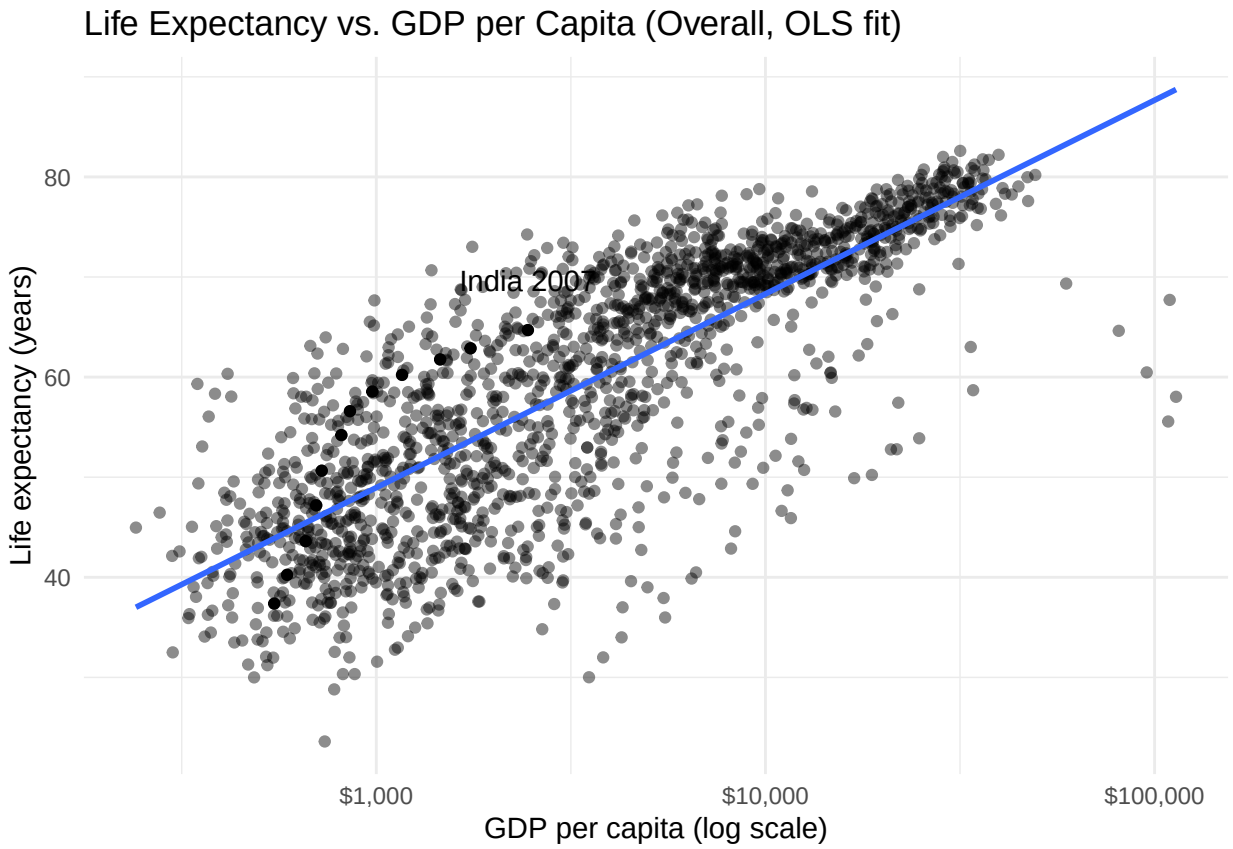
    - `scales = "free"` allows each facet to have its own axis scaling, which is useful when groups differ greatly in magnitude.

    - In your code:
        * Produces continent-specific scatter plots with their own OLS fit, while still overlaying India in each panel.

```
# Overall scatter with OLS fit; highlight India (all years)
p_scatter_overall <- ggplot(df, aes(x=gdpPercap, y=lifeExp)) +
  geom_point(alpha = 0.45) +
  geom_smooth(method = "lm", se = FALSE, linewidth = 1) +
  geom_point(data = india, aes(gdpPercap, lifeExp)) +
  ggrepel::geom_text_repel(
    data = india %>% filter(year == max(year)),
    aes(gdpPercap, lifeExp, label = paste(country, year)),
    nudge_y = 5, segment.color = NA
  ) +
  scale_x_log10(labels = scales::dollar_format(prefix = "$")) +
  labs(title = "Life Expectancy vs. GDP per Capita (Overall, OLS fit)",
       x = "GDP per capita (log scale)", y = "Life expectancy (years)") +
  theme_minimal()

# By continent, with continent-specific OLS fits; India shown in each panel
p_scatter_by_group <- ggplot(df, aes(x=gdpPercap, y=lifeExp)) +
  geom_point(alpha = 0.45) +
```

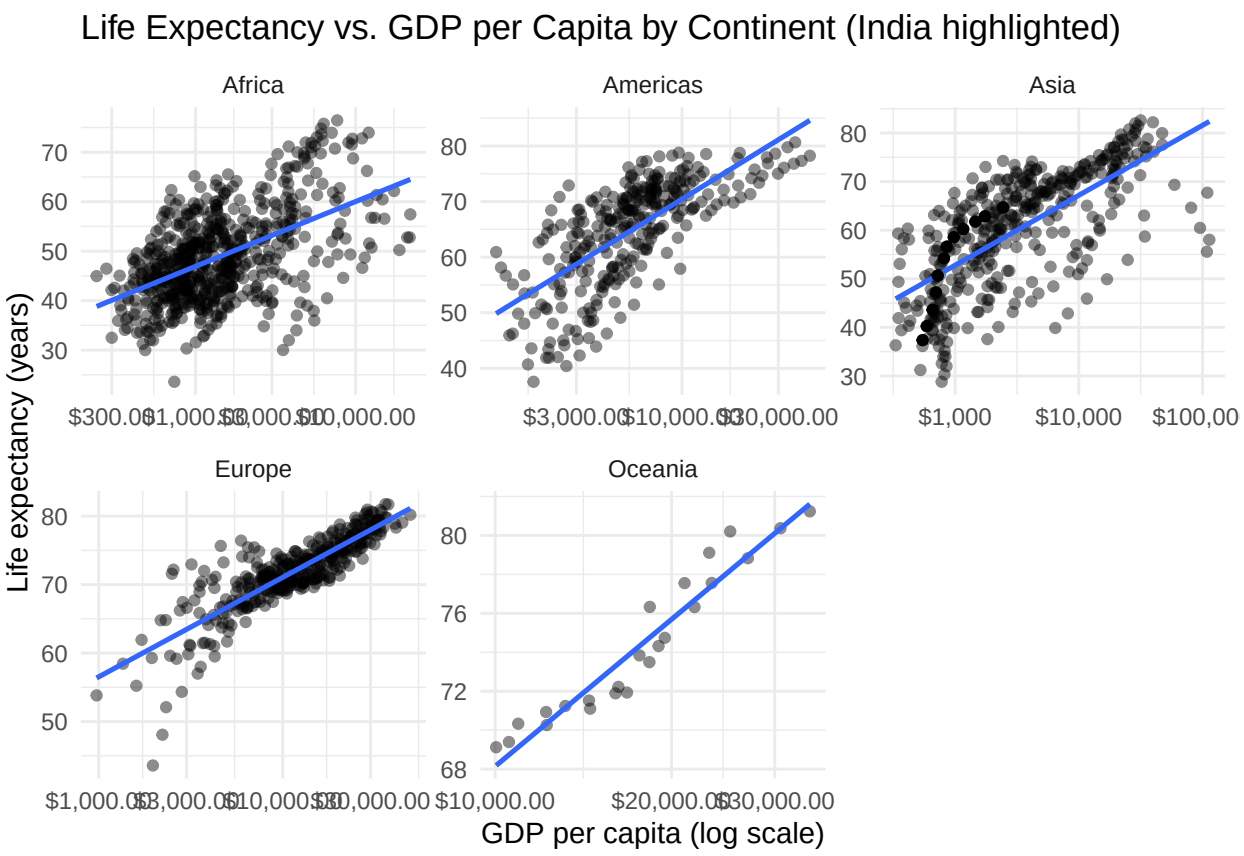
```
  geom_smooth(method = "lm", se = FALSE, linewidth = 0.9) +
  geom_point(data = india, aes(gdpPercap, lifeExp)) +
  facet_wrap(~ continent, scales = "free") +
  scale_x_log10(labels = scales::dollar_format(prefix = "$")) +
  labs(title = "Life Expectancy vs. GDP per Capita by Continent (India highlighted)",
       x = "GDP per capita (log scale)", y = "Life expectancy (years)") +
  theme_minimal()

p_scatter_overall
```

```
## `geom_smooth()` using formula = 'y ~ x'
```




```
p_scatter_by_group
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



5.5.1 Notes on Using `color = continent` for Overlaying Multiple Fits

- **`aes(color = continent)` inside `ggplot()` or within a geometry**
  - Tells `ggplot2` to map the variable `continent` to colors.
  - Each continent gets a distinct color, and both the plotted data (points, lines, or fitted curves) and the legend are generated automatically.
  - In the case of fitted lines (`geom_smooth(method = "lm")`), each continent gets its own regression line in a different color, overlaid in the same coordinate system.
  - This approach avoids faceting and instead layers group-specific fits in one plot, which can make cross-group comparisons easier.
- **Advantages of using `color = group_variable`**
  - Allows comparisons across groups in a single frame without splitting the plot into multiple facets.
  - The legend is automatically generated and linked to the colors, making it easy to identify which line or points belong to which group.
  - Works across multiple geoms at once (e.g., `geom_line`, `geom_point`, `geom_smooth`) so that the same legend entry applies consistently.
- **Caveats when using many categories**
  - When there are only a few groups (like continents), color differentiation works well and

is visually clear.

  - For datasets with many categories, colors may become visually overwhelming or "ugly," making interpretation harder.

  - In those cases, faceting (`facet_wrap`, `facet_grid`) is often a better alternative, or one might combine color with other aesthetics like line type (`linetype = group`) or shape (`shape = group`).

- **Flexibility of legends**
  - Because color is a shared aesthetic, it generates one unified legend that applies to all geoms that use `color`.

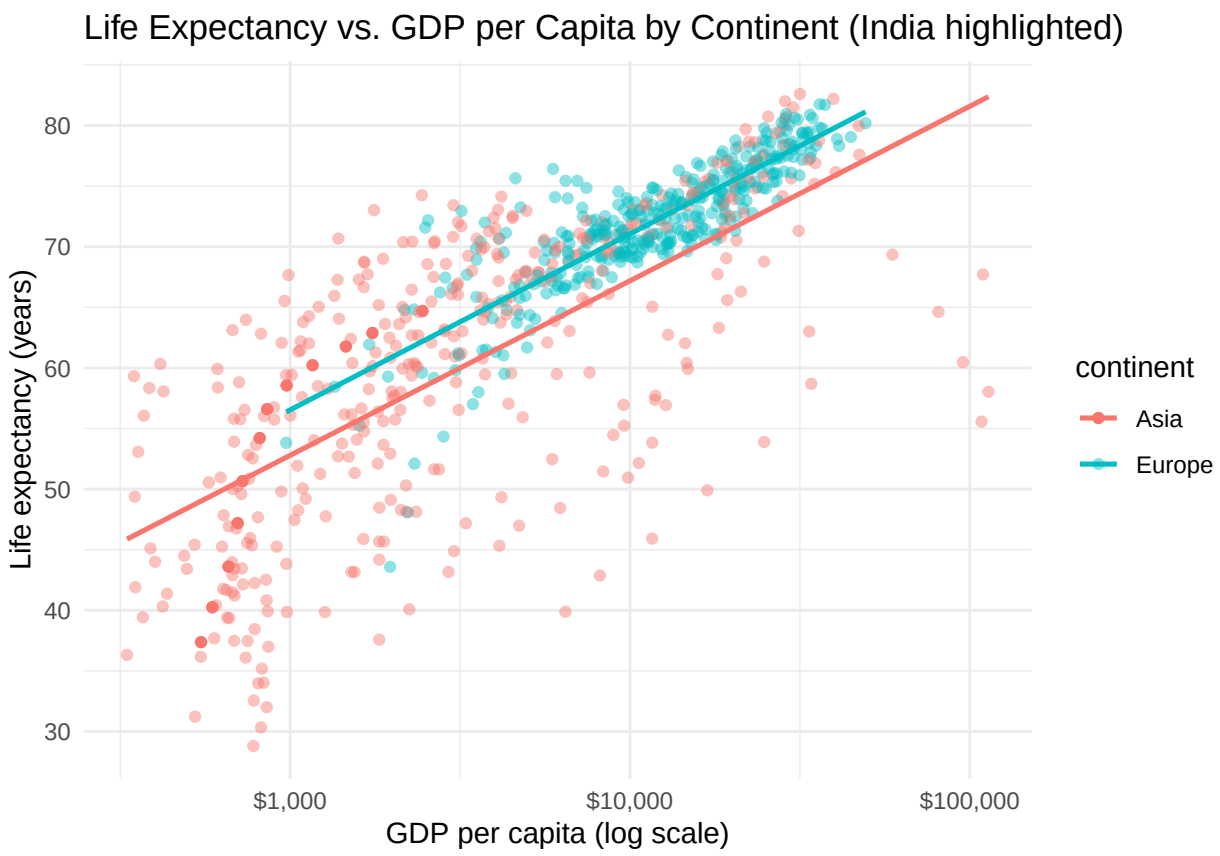
  - This means you can control how points, fitted lines, and even labels are tied to the same group identity in the legend.

  - You can also separately map other aesthetics (like line type) if you want multiple legends—for instance, one for `geom_line` style and another for `geom_smooth` style.

  - This flexibility is powerful for building informative and layered plots, where the same categorical variable is represented across different visual elements.

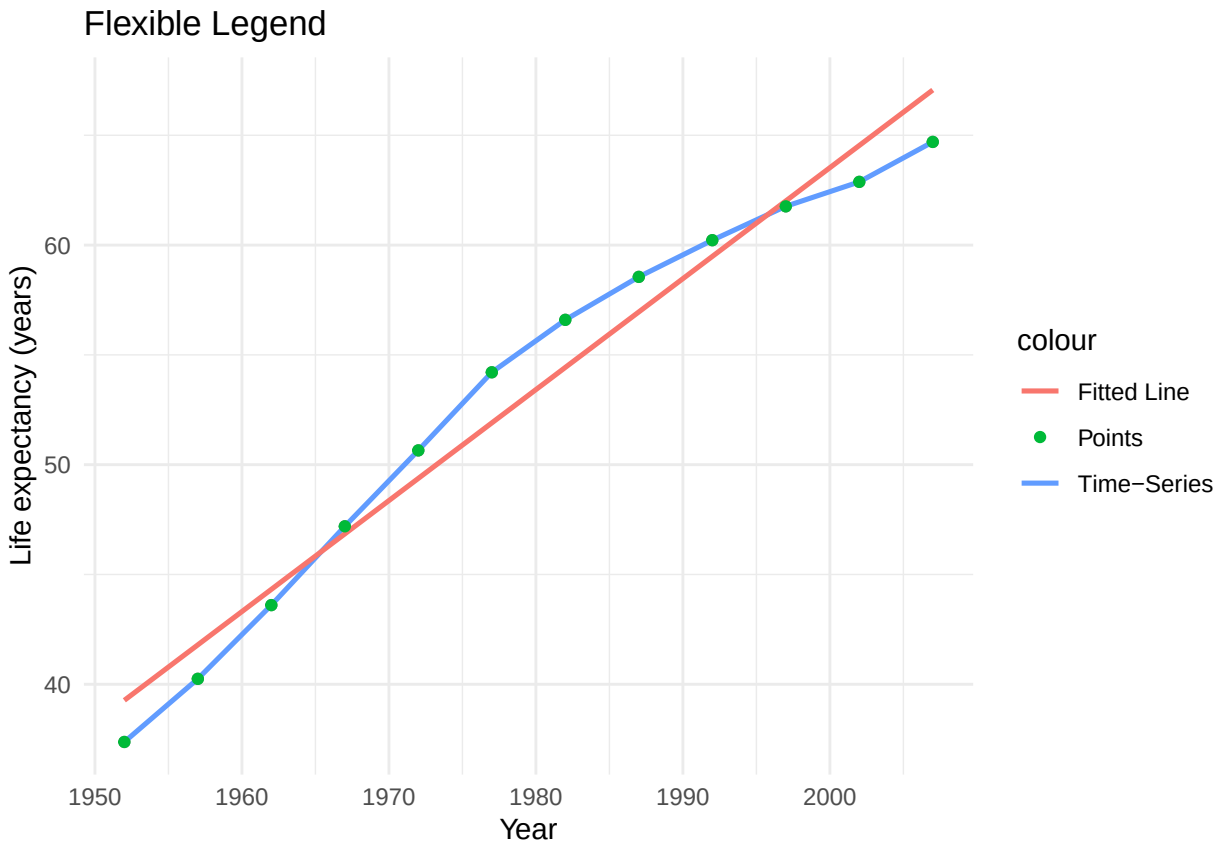
```
# By continent, with continent-specific OLS fits; India shown in each panel
p_scatter_by_color <- ggplot(df[df$continent %in% c("Europe","Asia"),], aes(x=gdpPercap, y=life
  geom_point(alpha = 0.45) +
  geom_smooth(method = "lm", se = FALSE, linewidth = 0.9) +
  geom_point(data = india, aes(gdpPercap, lifeExp)) +
  scale_x_log10(labels = scales::dollar_format(prefix = "$")) +
  labs(title = "Life Expectancy vs. GDP per Capita by Continent (India highlighted)",
       x = "GDP per capita (log scale)", y = "Life expectancy (years)") +
  theme_minimal()


# By continent, with continent-specific OLS fits; India shown in each panel
p_flexibile_legend <- ggplot(df[df$country %in% c("India"),], aes(x=year, y=lifeExp)) +
  geom_point(alpha = 0.45) +
  geom_line(aes(color="Time-Series"), linewidth = 0.9) +
  geom_smooth(aes(color="Fitted Line"),method = "lm", se = FALSE, linewidth = 0.9) +
  geom_point(data = india, aes(color="Points")) +
  labs(title = "Flexible Legend",
       x = "Year", y = "Life expectancy (years)") +
  theme_minimal()

p_scatter_by_color
```

```
## `geom_smooth()` using formula = 'y ~ x'
```




```
p_flexibile_legend
```

```
## `geom_smooth()` using formula = 'y ~ x'
```


Flexible Legend
Life expectancy (years)
60
50
40
1950
1960
1970
1980
1990
2000
Year
colour
Fitted Line
Points
Time−Series