Contents

This blog along with all the necessary data are available in different formats (pdf, html, Rmd, and md) on DataSharp's GitHub.

---

1. Tour de France Records: Data Challenges and Cycling Trivia Facts

Our Tour de France blog series returns for a second episode in which we will learn how to manipulate strings to uncover Tour de France trivia facts.

In our last post, we practiced loading real-world data, asking simple questions, and dealing with the inevitable quirks of messy datasets.

Today, we're gaining more confidence and taking things further.

We will use the same **Tour de France dataset** to tackle *more complex, layered questions.* These will require you to think critically, combine columns, process text, and validate your assumptions carefully.

This post will show you how to:

- **Transform messy columns** into clean, usable numbers.
- **Extract structured information** from unstructured text.
- **Evaluate results mindfully** to uncover richer and more accurate insights.

If you're ready to move from "basic data handling" to "practical analysis that actually answers questions," this post is for you.

1.1 Flashbash: What We Discovered Last Time

In our first exploration of the Tour de France dataset, we uncovered some surprising insights despite asking seemingly simple questions:

- Only **65 different riders** have won the Tour de France.
- In contrast, **917 different riders or teams** have won at least one Tour de France stage.
- **Three riders** managed to win the Tour de France *without ever winning a single stage* in their career.

The process by which we revealed these findings highlighted how even basic data questions can reveal inconsistencies, typos, and quirks in real-world datasets, **reminding us that cleaning and validating data is part of the analysis, not just an annoying pre-step**.

1.2 What We're Tackling Next

In this post, we're levelling up.

We will explore more layered questions such as:

- Which city has been visited the most in Tour history?
- What are the smallest and largest time gaps between the winner and the runner-up?
- Who has finished last the most times?

These questions will stretch our data skills further, as we will:

```
Practise data cleaning to prepare our data for robust analysis.
Combine information across multiple columns.
```

Ready? Let's dive in.

2. Setting The Scene

2.1 Installing / Loading Your Packages

As we saw in previous posts, R's default base package contains a lot of functionalities. However, it is common to need extra functionalities that we can load with packages. This analysis is no different. Today, we will use three packages: `readr`, `stringr`, and `stringi`.

Installing packages in R is straightforward:

```
if(!require('readr')) install.packages('readr')
if(!require('stringr')) install.packages('stringr')
if(!require('stringi')) install.packages('stringi')
```

Loading packages is commonly done with the `library(package_name)` function. However, it only works if the package is already installed on your machine. The code above is a bit longer, but it tries to load the package with `require()` and if the package is not found, it will first install it.

2.2 Loading the Data

If you've been following along, you're already familiar with the structure of our Tour de France dataset and how to load it into R. If not, check out the previous post for a step-by-step walkthrough.

For this analysis, we will once again load our three tables:

```
finishers <- read.csv('./data/tdf_finishers.csv')
stages <- read.csv('./data/tdf_stages.csv')
tours <- readr::read_csv('./data/tdf_tours.csv', locale=readr::locale(encoding="UTF-8"))
```

```
## Rows: 109 Columns: 6
## -- Column specification ----------------------------------------------------------------------------
## Delimiter: ","
## chr (3): Dates, Stages, Distance
## dbl (3): Year, Starters, Finishers
##
## 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.
```

2.3 Cleaning The Data

In the previous post, we identified many inconsistencies in the data but we didn't fix them. We worked around them, using logic and critical thinking. In particular, we found different name spelling across different tables:

- `"Maurice De Waele (BEL)"` vs `"Maurice Dewaele (BEL)"`
- `"Chris Froome (GBR)"` vs `"Chris Froome (UK)"`

To avoid unnenessary complications in our analyses below, I wanted to fix these problems first. Unfortunately, I immediately encountered a new problem. See for yourself:

```
head(finishers[finishers$Year==2013, ])
```

```
##      Year Rank                   Rider        Time              Team
## 8340 2013    1       Chris Froome (UK) 83h 56' 40"          Team Sky
## 8341 2013    2    Nairo Quintana (COL)    + 4' 20"      Movistar Team
## 8342 2013    3 Joaquim Rodríguez (ESP)    + 5' 04"      Team Katusha
## 8343 2013    4  Alberto Contador (ESP)    + 6' 27"      Saxo-Tinkoff
## 8344 2013    5   Roman Kreuziger (CZE)    + 7' 27"      Saxo-Tinkoff
## 8345 2013    6     Bauke Mollema (NED)   + 11' 42" Belkin Pro Cycling
```

```
finishers[finishers$Rider == "Chris Froome (UK)", ]
```

```
## [1] Year  Rank  Rider Time  Team
## <0 rows> (or 0-length row.names)
```

We can see there is at least one entry for "Chris Froome (UK)" in the `finishers` table. However, when I try to capture it with a boolean expression, nothing appears. **Why is that now?**

> It took me some time to identify the root cause, but I eventually found an unsual character, ***hidden* non-ASCII characters**, that looks like a space but is not a space – if you look at the table above carefully, you'll notive the two "spaces" between the first and last names, and the last name and country code do not have the same width. That's what causes the mismatches between the strngs. But how can we see these more clearly?

Since I had no idea how to do that in R (I actually used Python to load the data and discover the non-ASCII characers), I had to Google it for this post. Here is the solution I found. It is not perfect, but it does the job:

```
tools::showNonASCII(head(finishers[finishers$Year==2013, 'Rider']))
```

```
## 1: Chris Froome<c2><a0>(UK)
## 2: Nairo Quintana<c2><a0>(COL)
## 3: Joaquim Rodr<c3><ad>guez<c2><a0>(ESP)
## 4: Alberto Contador<c2><a0>(ESP)
## 5: Roman Kreuziger<c2><a0>(CZE)
## 6: Bauke Mollema<c2><a0>(NED)
```

We can now clearly see that the space between the first and last name is a proper space, while the "space" between the name and the country code is something else! Since the odds of findings more non-ASCII characters elsewhere in the table are high, the simplest solution for us to move forward with our analysis is to batch-replace all the non-ASCII characters by a similar ASCII characters (*e.g.* é by e, ù by u, weird space by regular space, etc.)

Note: Keep in mind that Data Scientists are not expected to know everything. Our strength comes from our process to finding adequate solutions. A big part of the job is to search for solutions to similar problems and to adapt them to your situation.

Naturally, someone had that problem before and developed a solution for it. I found it in the `stringi` package, which has a function that force-converts all characters to ASCII. The `stringi::stri_trans_general()` will thus allow me to put encode all my strings similarly using standard characters.

```r
finishers$Rider <- stringi::stri_trans_general(finishers$Rider, "latin-ascii")
stages$Winner <- stringi::stri_trans_general(stages$Winner, "latin-ascii")
```

We can check the names are now looking correct.

```r
tools::showNonASCII(head(finishers[finishers$Year==2013, 'Rider']))
```

Empty result, meaning there are no fields with non-ASCII characters anymore. Everything was changed to ASCII characters. Yay!

We can further validate the changes by trying our first test again. We now get a match:

```r
finishers[finishers$Rider == "Chris Froome (UK)", ]
```

```
##      Year Rank             Rider        Time     Team
## 8340 2013    1 Chris Froome (UK) 83h 56' 40" Team Sky
```

We can now fix the remaining name inconsistencies from the `finishers` and `stages` tables:

```r
finishers[finishers$Rider == "Chris Froome (UK)", 'Rider'] <- "Chris Froome (GBR)"
stages[stages$Winner == 'Maurice Dewaele (BEL)', 'Winner'] <- 'Maurice De Waele (BEL)'
```

And, as good practice habit, we always check that the data were really modified.

```r
finishers[finishers$Rider == "Chris Froome (UK)", ]
```

```
## [1] Year  Rank  Rider Time  Team
## <0 rows> (or 0-length row.names)
```

Note: Since I modified the Rider `columns` from `finishers`, I also add to modify the `Winner` column from `stages` to ensure that the pairings would still be there, even for those without spelling issues.

3. Now the Fun Begins: Exploring Tour de France Records

3.1 Which City Has Been Visited Most Often?

In this question, we aim to find the most visited city in Tour de France history. Here, "visited" means either the starting city or the finishing city of a stage, as recorded in our `stages` table.

This is a fun way to practice string splitting, counting, and sorting in R while building your instinct for checking results critically.

Step 1: Look at the Data Before coding, let's check how the information is structured:

```r
head(stages)
```

```
##   Year       Date Stage                Course        Distance
## 1 1903 1903-07-01     1         Paris to Lyon 467 km (290 mi)
## 2 1903 1903-07-05     2     Lyon to Marseille 374 km (232 mi)
## 3 1903 1903-07-08     3 Marseille to Toulouse 423 km (263 mi)
## 4 1903 1903-07-12     4  Toulouse to Bordeaux 268 km (167 mi)
## 5 1903 1903-07-13     5    Bordeaux to Nantes 425 km (264 mi)
## 6 1903 1903-07-18     6       Nantes to Paris 471 km (293 mi)
##                     Type                    Winner
## 1            Plain stage         Maurice Garin (FRA)
## 2 Stage with mountain(s) Hippolyte Aucouturier (FRA)
## 3            Plain stage Hippolyte Aucouturier (FRA)
## 4            Plain stage        Charles Laeser (SUI)
## 5            Plain stage         Maurice Garin (FRA)
## 6            Plain stage         Maurice Garin (FRA)
```

You see the `Course` column contains entries like "Paris to Lyon", "Lyon to Marseille", etc. We need to cut these strings using the " to " keyboard and to keep the two outputs.

Step 2: Splitting City Names We need to separate the starting and finishing cities from each Course. We can use:

- `stringr::str_split()` to split the strings using " to " as the separator.
- `simplify = TRUE` to return a matrix with two columns: the start and end cities.

```r
city_names <- stringr::str_split(stages$Course, ' to ', simplify=TRUE)
head(city_names)
```

```
##      [,1]        [,2]
## [1,] "Paris"     "Lyon"
## [2,] "Lyon"      "Marseille"
## [3,] "Marseille" "Toulouse"
## [4,] "Toulouse"  "Bordeaux"
## [5,] "Bordeaux"  "Nantes"
## [6,] "Nantes"    "Paris"
```

Tip: Pay attention to the spaces around " to ". If you forget them, you'll get city names like"Paris " or " Lyon", which will trip you up later. `"Paris "` (space after) and `" Paris"` (space before) are different, even for the smartest computers.

Step 3: Counting and Sorting Let's count how often each city appears as a start or end city:

- Use `table()` to count each city's occurrences across the two columns.
- Use `sort()` the rank the counts by decreseasing order.
- Use `head(..., n=20)` to show the top 20.

```r
head(sort(table(city_names), decreasing=TRUE), n=20)
```

```
## city_names
##                  Bordeaux          Pau        Paris       Luchon         Metz 
##          136          130          127          104           94           76 
##     Grenoble         Nice    Perpignan     Briançon    Marseille         Caen 
##           75           73           69           67           67           65 
##      Bayonne  Montpellier        Brest       Nantes     Toulouse      Belfort 
##           61           58           56           53           52           51 
##      Roubaix          Gap 
##           49           48
```

Step 4: Checking Results Critically The most visited city (136 visits) is … a city without name!

While we might first think there is another issue data, let's shift our mindset and look at this as if if was *the right answer*.

Let's look for patterns in our data:

```r
sum(city_names[, 1] == '') ## Counting how many start cities don't have a name
```

```
## [1] 0
```

```r
sum(city_names[, 2] == '') ## Counting how many arrival cities don't have a name
```

```
## [1] 136
```

Turns out, all the blank names are arrival cities. Let's examine them further:

```r
head(stages[city_names[, 2] == '', ])
```

```
##     Year       Date Stage                      Course        Distance
## 782 1955 1955-07-07    1b                      Dieppe  12.5 km (8 mi)
## 807 1956 1956-07-08    4a Circuit de Rouen-Les-Essarts  15.1 km (9 mi)
## 829 1957 1957-06-29    3a  Circuit de la Prairie, Caen  15 km (9.3 mi)
## 843 1957 1957-07-12   15b     Montjuïc circuit (Spain) 9.8 km (6.1 mi)
## 858 1958 1958-07-03     8                  Châteaulin   46 km (29 mi)
## 889 1959 1959-07-10    15                 Puy de Dôme  12 km (7.5 mi)
##                      Type                      Winner
## 782       Team time trial                 Netherlands
## 807 Individual time trial            Charly Gaul (LUX)
## 829       Team time trial                      France
## 843 Individual time trial       Jacques Anquetil (FRA)
## 858 Individual time trial            Charly Gaul (LUX)
## 889   Mountain time trial Federico Bahamontes (ESP)
```

If we look at the `Course` column, we can see that only one city name is indicated, which supports the results we got. Our string splitting worked as expected.

When we look at the `Type` column, here we see a potential reason emerging! The "problematic" stages are time trials, which start and end from the same city. Let's verify that:

```r
table(stages[city_names[, 2] == '', 'Type'])
```

```
##
##            Flat stage   High mountain stage           Hilly stage
##                     2                     1                     2
## Individual time trial   Mountain time trial           Plain stage
##                    95                     1                    11
##       Team time trial
##                    24
```

Most of them are time trials, indeed, but not all. We can imagine that regular stages may also start and end in the same city, like the Toulouse - Toulouse stage that is happening as I write these lines in the 2025 edition. Such stage would probably be recorded as `Toulouse` if our dataset was up-to-date.

That brings us to a challenge for you:

> **Your turn:** Can you identify all the stages that started and ended in the same city but were not time trials? Share your solution in the comments below.

Step 5: Drawing Robust Conclusions Now that we understand the blank entry, let's revisit the actual results.

Bordeaux appears to be the most visited city, followed by Pau and Paris.

> **But... is that really true?**

Look closely and you'll notice that Paris appears in different forms, like "Paris" and "Paris (Champs-Élysées)". These inconsistencies can split your counts.

Let's identify and count the all possible occurrences of Bordeaux, Pau, and Paris. How? Do you remember `stringr::str_detect()` we discovered in the previous post?

```r
sum(stringr::str_detect(city_names, "Bordeaux"), na.rm=TRUE)
```

```
## [1] 130
```

```r
sum(stringr::str_detect(city_names, "Pau"), na.rm=TRUE)
```

```
## [1] 132
```

Looks like Bordeaux is consistent. In contrast, Pau shows up a few extra times. But are these extra occurrences real?

```r
table(city_names[stringr::str_detect(city_names, "Pau")])
```

```
##
##                       Pau                  Pauillac Saint-Paul-Trois-Châteaux
##                       127                         1                         4
```

Nope, they were **false positive**! So Bordeaux has been more visited than Pau. But how many times was Paris visited? With the data above, we already see that Paris was visited many more times than Bordeaux. Let's try to count how many more.

```r
table(city_names[stringr::str_detect(city_names, "Paris")])
```

```
##
##         Disneyland Paris   Le Touquet-Paris-Plage                     Paris
##                        3                        4                       104
##   Paris (Champs-Élysées) Paris (Eiffel Tower)[12]   Paris La Défense Arena
##                       48                        1                         1
```

```r
sum(stringr::str_detect(city_names, "Paris"), na.rm=TRUE)
```

```
## [1] 161
```

All names but `Le Touquet-Paris-Plage` correspond to Paris.

> When combined, **Paris was visited 157 times**, making it the most visited city in Tour de France history (as many of you probably expected!).

3.2 What Are the Smallest and Largest Time Gaps Between the Winner and the Runner-Up of the Tour?

This question takes us into the `Time` column of the `finishers` table. The challenge? The data are far from being clean or consistent:

- For the winner, the column contains their total time..
- For all other finishers, the column shows the time difference with the winner. This kind of data is usually called an "anomaly" or a delta from a reference value.

> In many analyses, you'd want to convert anomalies back to absolute values. But in our case, anomalies are exactly what we need! We want to find the smallest and largest anomalies for second-place finishers.

Step 1: Filtering the Data We are only interested in the time difference between riders ranked second and the winners. Therefore, we can create this subset of the finishers table:

```r
finishers_only2nd <- finishers[finishers$Rank == 2, ]
head(finishers_only2nd, n=15)
```

```
##     Year Rank                          Rider          Time            Team
## 2   1903    2           Lucien Pothier (FRA) + 2h 59' 21"    La Française
## 23  1904    2 Jean-Baptiste Dortignacq (FRA) + 2h 16' 14"
## 38  1905    2    Hippolyte Aucouturier (FRA)                Peugeot-Wolber
## 62  1906    2        Georges Passerieu (FRA)                       Peugeot
## 76  1907    2         Gustave Garrigou (FRA)                Peugeot-Wolber
## 109 1908    2           Francois Faber (LUX)                Peugeot-Wolber
## 145 1909    2         Gustave Garrigou (FRA)                        Alcyon
## 200 1910    2           Francois Faber (LUX)                        Alcyon
## 241 1911    2               Paul Duboc (FRA)                  La Française
## 269 1912    2        Eugene Christophe (FRA)                         Armor
## 310 1913    2         Gustave Garrigou (FRA)       +8' 37"         Peugeot
## 335 1914    2          Henri Pelissier (FRA)     + 1' 50" Peugeot-Wolber
## 389 1919    2            Jean Alavoine (FRA) + 1h 42' 54"
## 399 1920    2          Hector Heusghem (BEL)    + 57' 21"
## 421 1921    2          Hector Heusghem (BEL)    + 18' 36"
```

We can already see that the information is missing for the years 1905 to 1912. So we will have to get rid of these years before continuing.

> **Tour trivia** Between 1905 and 1912, the Tour riders were ranked by a point-based system instead of time, due to rampant cheating in the early editions. Curious about the juicy details? Check out the Wikipedia of the 1905 edition of the Tour de France.

Let's identify what kind of "empty" we're dealing with. Do we have empty strings, i.e. `""`, or `NA`/`NULL` values.

```r
finishers_only2nd[finishers_only2nd$Year == 1905, 'Time']
```

```
## [1] ""
```

Looks like these are empty strings (`""`), not NA or NULL. Let's remove them:

```r
nrow(finishers_only2nd)
```

```
## [1] 109
```

```r
finishers_only2nd <- finishers_only2nd[ ! finishers_only2nd$Time == "", ]
nrow(finishers_only2nd)
```

```
## [1] 101
```

We're now working with 101 editions of the Tour when time defined the final ranking.

Step 2: Cleaning the Time Data This is the critical, challenging step.

The `Time` anomalies are stored as complex strings that we need to convert to numbers. Unfortunately, this way of reporting time is non-standard. No built-in R function can handle these directly ... so we'll write our own!

Here's a custom function to turn these strings into total seconds, making comparisons and ranking between them straightforward. I will guide you through every step of the function below, but I want you to first take some time to read and try to understand it. I only used functions we previously encountered.

```r
convert_time_string <- function(time_string) {
    ## There are again some non-ASCII characters, so I am removing them
    clean_str <- stringi::stri_trans_general(time_string, "latin-ascii")

    ## Remove '+' prefix
    clean_str <- substr(clean_str, 2, nchar(time_string))

    ## Removing all spaces from string
    clean_str <- stringr::str_replace_all(clean_str, " ", "")

    # Settings everything to 0
    hours <- minutes <- seconds <- 0

    # Extracting hours
    if(stringr::str_detect(clean_str, 'h')) {
        cut_str <- stringr::str_split(clean_str, 'h', simplify=TRUE)
        hours <- as.numeric(cut_str[1,1])
        clean_str <- cut_str[1,2]
    }

    # Extracting minutes
    if(stringr::str_detect(clean_str, "'")) {
        cut_str <- stringr::str_split(clean_str, "'", simplify=TRUE)
        minutes <- as.numeric(cut_str[1,1])
        clean_str <- cut_str[1,2]
    }

    # Extracting seconds
    if(stringr::str_detect(clean_str, '\"')) {
        cut_str <- stringr::str_split(clean_str, '\"', simplify=TRUE)
        seconds <- as.numeric(cut_str[1,1])
        clean_str <- cut_str[1,2]
    }

    # Total time in seconds
    total_seconds <- hours * 3600 + minutes * 60 + seconds
    return(total_seconds)
}
```

How does this function work?

Let's take a closer look at what this function does, and why each step is needed. This will help you learn how to approach similar problems on your own.

We want to transform a time string like this: `+ 5h 49' 12"` into a clean number (5*3600+49*60+12 = 20952) representing the total number of seconds (which we can then compare, sort, etc.).

1. **time_string is our input**: This is one value from the Time column of our dataset. It can look like `+ 1h 2' 14"` or just `+ 42"` depending on the race. Not all time strings have entries for the hours, minutes, or seconds.
2. **We clean up the string with stri_trans_general()**: This dataset is full of non-ASCII characters and this column is no exception. My first attempts crashed because of the mixed presence of curly and straight quotes. While we could have fixed that before using the function, including it in the function body adds an extra security if we ever want to reuse this function in another context.
3. **We remove the "+" sign with substr()**: All our time strings begin with a +, but we don't need it. Using substr(), we grab everything after the first character. I used `substr()` here as a chance to show you a new tool, though `str_replace(clean_str, "+", "")` would also work.
4. **We remove all the spaces**: To make the parsing easier, we remove all whitespace, replacing them with an empty character. Note: `str_replace()` only removes the first space. Since we want to remove all of them, we use `str_replace_all()`.
5. **We initialise all time components to zero**: Not all time strings include hours or minutes, so we define `hours`, `minutes`, and `seconds` equal to 0. Step 8 requires hours, minutes, and seconds to have values. Setting them to 0 will have no effect on the results if our time string does not contain entry for these parameters.

6. **We extract hours if they exist**: We check if the string contains an "h". If yes, we split it. "5h49'12"" → ["5", "49'12""] We save the "5" as the number of hours and update `clean_str` with the second element to work on the remaining part.
7. **We extract minutes and seconds**: We apply the same idea, searching for "''' and """ to split the string for the minutes and then seconds.
8. **We convert everything to seconds**: I bring everything together by converting the hours and minutes to seconds, and summing everything.
9. **We return that total**: The `total_second` we compute gets returned by the function. That's the value we'll use in our analysis.

> **A Quick Reality Check** I'm showing you the final, polished version of the function here, but it didn't start that way. It took me almost an hour to get this function working properly. I started with a simple version that worked for one or two clean examples, then slowly added more cleaning and filtering to handle edge cases: "+" v "+", the absence of hours or minutes in some elements, the presence of non-ASCII characters (always them!). **That's how real coding works.** You don't have to write the perfect function in one go. You start with the ideal case, then adapt your code as your data shows you its quirks.

Now that our function works, we want to apply it to every value in the `Time` column.

In R, one great way to do this is with `sapply()`:

This applies `convert_time_string()` to every row of the column and returns a vector of results.

```
total_seconds=sapply(finishers_only2nd[, 'Time'], convert_time_string)
head(total_seconds)
```

```
## + 2h 59' 21" + 2h 16' 14"      +8' 37"     + 1' 50" + 1h 42' 54"    + 57' 21"
##        10761         8174          517          110         6174         3441
```

You can manually check that the number of seconds corresponds to the above Time string.

We can then append these results to our data table using `cbind()` (for column binding):

```
finishers_only2nd <- cbind(
                finishers_only2nd,
                "total_seconds"=total_seconds
            )
head(finishers_only2nd)
```

```
##     Year Rank                          Rider         Time           Team
## 2   1903    2          Lucien Pothier (FRA) + 2h 59' 21"   La Française
## 23  1904    2 Jean-Baptiste Dortignacq (FRA) + 2h 16' 14"
## 310 1913    2        Gustave Garrigou (FRA)      +8' 37"        Peugeot
## 335 1914    2         Henri Pelissier (FRA)     + 1' 50" Peugeot-Wolber
## 389 1919    2           Jean Alavoine (FRA) + 1h 42' 54"
## 399 1920    2         Hector Heusghem (BEL)    + 57' 21"
##     total_seconds
## 2           10761
## 23           8174
## 310           517
## 335           110
## 389          6174
## 399          3441
```

Now our table has a new column `total_seconds` ready for sorting, filtering, or whatever else we want to do.

Step 3: Answering the question

> What are the smallest and largest time gaps between a Tour de France winner and the second-place rider?

Let's compute the minimum and maximum values in our new total_seconds column:

```
print(paste("Minimal time between first and second riders:", min(finishers_only2nd$total_seconds), "sec
```

```
## [1] "Minimal time between first and second riders: 8 seconds."
```

```
print(paste("Maximal time between first and second riders:", max(finishers_only2nd$total_seconds), "sec
```

```
## [1] "Maximal time between first and second riders: 10761 seconds (or 2.99 hours)."
```

The verdict?

- The smallest gap was just **8 seconds**.
- The largest gap was almost 3 hours!

I can't decide which result is the most surprising.

But that 8-second margin sounds intense. Can we find out when that happened?

Can we identify which Tour de France that was?

This gives us the perfect excuse to introduce a handy function: `which.min()`.

- `min(x)` gives you the smallest value in a vector.
- `which.min(x)` gives you the position (or row index) of that smallest value in that vector.

And once you have the position, you can use it to extract the corresponding row from `finishers_only2nd`:

```
finishers_only2nd[which.min(finishers_only2nd$total_seconds), ]
```

```
##      Year Rank                Rider     Time                 Team total_seconds
## 4943 1989    2 Laurent Fignon (FRA) + 0' 08" Super U-Raleigh-Fiat             8
```

> The legendary Tour de France 1989, where American Greg LeMond snatched victory from Frenchman Laurent Fignon in the final Time Trial stage for 8 tiny seconds.

3.3 Who has finished last the most?

With this final question, let's honour those away from the spotlight.

Cycling is a team sport: champions rely on their teammates to control the race, fetch bottles, and shield them from the wind. For many years, the last rider in the final ranking of the Tour de France, the so-called *lanterne rouge*, was even invited onto the final podium.

So let's do the same here by paying hommage to the loyal champion of champions.

Step 1: Understanding the Challenge Identifying the winner of each Tour de France is easy: we can use `Rank == 1`.

But for the last rider, it's trickier:

- The number of finishers varies each year.
- In 1903, the last rider was ranked 21st, while in 2022, the last rider was ranked 134th.

We need a flexible way to find the last rider for each edition.

Step 2: Using tapply() to Extract Last Riders **How to identify the list rider from each Tour de France edition?**

The *lanterne rouge* is defined as the rider with the largest rank **for each edition**.

When you hear **for each edition** or similar criteria, you should think of group filtering.

We need a function that does a *specific task* for *each elements of the group*.

The easiest way to apply a function to different subgroups within a dataset is with `tapply()`.

It takes:

- A vector of values to analyse (here, the names of finishers),
- A categorical vector to group by (here, the Year of the Tour),
- A function to apply to each subgroup (here, we can use `tail()` to get the last element since the riders are sroted from first to last),
- Additional arguments if needed (`n = 1` to get a single last name).

Since our data are ordered from first to last rider, `tail(..., n=1)` will return the last rider for each year, giving us exactly what we need.

```r
last_each_tour <- tapply(finishers$Rider, finishers$Year, tail, n=1)
head(last_each_tour)
```

```
##                        1903                        1904
##    "Arsene Millocheau (FRA)" "Antoine Deflotriere (FRA)"
##                        1905                        1906
##       "Clovis Lacroix (FRA)"    "Georges Bronchard (FRA)"
##                        1907                        1908
##      "Albert Chartier (FRA)"       "Henri Anthoine (FRA)"
```

```r
## Let's make these data look sharper
last_each_tour <- data.frame("Year" = names(last_each_tour), "Name" = last_each_tour)
head(last_each_tour)
```

```
##      Year                      Name
## 1903 1903   Arsene Millocheau (FRA)
## 1904 1904 Antoine Deflotriere (FRA)
## 1905 1905      Clovis Lacroix (FRA)
## 1906 1906   Georges Bronchard (FRA)
## 1907 1907     Albert Chartier (FRA)
## 1908 1908      Henri Anthoine (FRA)
```

Step 3: Counting Who Came Last Most Often Now, we simply count how often each name appears in our `last_each_tour` table to find the rider who finished last the most times in Tour de France history.

```
sort(table(last_each_tour$Name), decreasing=TRUE)[1]
```

```
## Wim Vansevenant (BEL)
##                     3
```

> Belgian rider Wim Vansevenant finished last in the Tour de France three times, making him the most frequent lanterne rouge in history. As he expressed it himself: "Lanterne rouge is not a position you go for, it comes for you." (Source Wikipedia).

4. Food for thought

1. **Real-world data is rarely clean or consistent.** From empty strings and strange characters to formatting quirks and typos, working with messy data is the norm, not the exception.
2. **When data looks strange, stop and investigate.** A city with no name? A time gap that looks suspicious? Dig deeper. It may be an issue with the data, but there could also be a logical explanation. But you must be curious enough to look.
3. **Use built-in tools to simplify your life.** Functions like `tapply()`, `which.min()`, or even `str_detect()` are powerful allies. I will try to showcase as many of these useful tricks in my posts, so stay alert for more (Even when the chosen topic may not appeal to you!).
4. **And when what you need doesn't exist, build it.** Sometimes no function will do exactly what you want. That's when writing your own becomes not just helpful, but empowering. It lets you tailor your analysis to your data, not the other way around.

> **Key takeaway:** Good data analysis isn't about knowing every trick. It's about staying curious, being resourceful, and feeling confident enough to build the tools you need when the situation calls for it.

5. What's Next?

Ready to keep sharpening your data skills?

In the next and final Tour de France post, we will dive into **dataset merging** and do basic **data visualisation**.

> **Stay tuned**, and if you have questions or examples you'd like us to cover in the next post, drop them in the comments below!

> **Your learning challenge until then:** Now that you know more about string manipulation, can you identify which country has won the most Tour de France? Is it France? Or is it another one? Share your solution in the comment section below!

This blog along with all the necessary data are available in different formats (pdf, html, Rmd, and md) on DataSharp's GitHub.
