Contents

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

---

1. Start Here: Building Data Confidence

Data analysis can feel intimidating, especially when you're working with real-world datasets that are messy, inconsistent, and rarely behave like the polished examples you find in tutorials. But this is where the real learning **and the real fun** begins.

In this post, we'll practice extracting information from data using simple, robust R commands. I kept things simple and practical, focusing on thinking critically with the data.

Each part builds on the previous one, helping you develop structured, confident workflows.

Who is this post for? This post is for beginners who want to learn practical data manipulation and sharpen their instincts for extracting information from complex datasets. It will also challenge those who already know R but haven't yet practiced systematic data interrogation.

Topics covered in this post:

- Loading datasets with non-standard characters.
- Extracting information from single columns.
- Conditional subsetting of vectors and data frames.
- Mindful result evaluation.

2. Meet the Data: The *Tour de France* Dataset

Before diving into any data analysis, it's essential to take the time to **look at the data with your own eyes**. This step is often skipped, but it's where many problems—and insights—first reveal themselves. **Make sure you see the data for how they are and not how you think they are.**

Ask yourself:

- What do these data represent?
- What is measured, and how?
- What are the units and scales?
- Are there quirks in formatting or naming?

In this post, we'll use historical data from the *Tour de France* to practice this mindset. The dataset includes information on stages, winners, distances, and participation across more than a century of races. Each Tour de France is composed of several stages, each stage having a uniquer winner. The cumulated time across all stages define the overall winner of the Tour de France. This dataset a rich, real-world example. Although it looks quite simple (see tables below), it is full of typos and inconsistencies, making it ideal for sharpening your data intuition.

All data used here come from this Kaggle datasets. I encourage you to download the data and to run the commands yourself. But I want you to think about what you are doing; do not just run code.

We will work with three tables:

Table 1: `tdf_tours.csv`: Information for each *Tour de France* edition, including year, dates, number of stages, total distance (km/mi), and the number of starters and finishers.

Year	Dates	Stages	Distance	Starters	Finishers
1903	1–19 July 1903	6	2,428 km (1,509 mi)	60	21
1904	2–24 July 1904	6	2,428 km (1,509 mi)	88	15
1905	9–30 July 1905	11	2,994 km (1,860 mi)	60	24
1906	4–29 July 1906	13	4,637 km (2,882 mi)	82	14
1907	8 July – 4 August 1907	14	4,488 km (2,789 mi)	93	33

Table 2: `tdf_finishers.csv`: Lists all riders who finished each *Tour de France*, along with their final rank, team, and time (expressed as a time difference from the winner for non-winners).

Year	Rank	Rider	Time	Team
1903	1	Maurice Garin (FRA)	94h 33' 14"	La Française
1903	2	Lucien Pothier (FRA)	+ 2h 59' 21"	La Française
1903	3	Fernand Augereau (FRA)	+ 4h 29' 24"	La Française
1903	4	Rodolfo Muller (ITA)	+ 4h 39' 30"	La Française
1903	5	Jean Fischer (FRA)	+ 4h 58' 44"	La Française

Table 3: `tdf_stages.csv`: Contains one row per stage of the *Tour de France* from 1903 to 2022, detailing start and end cities, distance, stage type (km/mi), and the stage winner.

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

Before we move on, take a moment to **skim these tables slowly**. Notice the variety of data types: numbers mixed with units, dates with different formats, rider names with embedded country codes. These are the kinds of details that will shape how we clean, transform, and analyse these data later.

Look for potential challenges and opportunities:

- Which columns might need cleaning before analysis?
- Which columns are categorical vs. numerical?
- Are there inconsistencies that might require your attention?

> **Key takeaway:** Before writing a single line of code, ALWAYS spend time **looking at your dataset and understanding it**. Identify how your different tables are linked to each other. This will help you anticipate challenges and guide your analysis.

3. Let's Get to Work

Now that you're familiar with the dataset, let's start **interrogating the data**. We will begin by practicing simple R commands to extract meaningful information, exploring how to count unique values, subset conditionally, and evaluate results mindfully. This will build a solid foundation before we move on to string manipulation and data visualisation in the next posts.

3.1 Installing / Loading Your Packages

R's default base package contains a lot of functionalities. However, it is more than likely that you will have to add new functionalities by loading what is called "packages". This analysis is no different. Installing packages in R is straightforward:

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

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.

4. Where We All Begin: Loading The Data

There are many ways to load data into your R environment. Since our files are plain CSVs, we'll start simply:

```
finishers <- read.csv('./data/tdf_finishers.csv')
head(finishers)
```

```
##   Year Rank                    Rider         Time         Team
## 1 1903    1      Maurice Garin (FRA)  94h 33' 14" La Française
## 2 1903    2     Lucien Pothier (FRA) + 2h 59' 21" La Française
## 3 1903    3   Fernand Augereau (FRA) + 4h 29' 24" La Française
## 4 1903    4 Rodolfo Muller[27] (ITA) + 4h 39' 30" La Française
## 5 1903    5       Jean Fischer (FRA) + 4h 58' 44" La Française
## 6 1903    6       Marcel Kerff (BEL) + 5h 52' 24"
```

I use the `head()` function to display the first six rows of the dataset. Everything is looking good.

```
stages <- read.csv('./data/tdf_stages.csv')
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)
```

Also fine. Now let's try the last file:

```
stages <- read.csv('./data/tdf_tours.csv')
```

```
## Error in type.convert.default(data[[i]], as.is = as.is[i], dec = dec,  :
##  invalid multibyte string at '<96>19 <4a>uly 1903'
```

Oops. Welcome to real-world data, where the first "easy step" breaks immediately.

What does this error mean? Usually, it indicates there's a strange character in your file that your computer can't read properly. By default, computers only "understand" standard keyboard characters, and unusual characters can cause issues during import.

> **Tip:** R error messages are often cryptic, and it's normal to feel stuck when reading them for the first time(s). Don't forget you can always Google the exact error message or ask your favorite AI tool for help. Data analysis is not about knowing everything; it is about knowing how to solve problems.

When I opened the file in Excel, I found odd values like *"1ñ19 July 1903"* in the Dates column. This unexpected *ñ* character likely wasn't intentional and may result from moving files between operating systems (Windows, Mac, Linux) or even different versions of the same OS.

How to move forward? We need to read the file using function with a character encoding able to understand what this *ñ* character is. There are many encoding system and the first one you should try is UTF-8.

```
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.
```

```
head(tours)
```

```
## # A tibble: 6 x 6
##    Year Dates                          Stages Distance          Starters Finishers
##   <dbl> <chr>                          <chr>  <chr>                <dbl>     <dbl>
## 1  1903 "1\x9619 July 1903"            6      "2,428\xa0km (1,~       60        21
## 2  1904 "2\x9624 July 1904"            6      "2,428\xa0km (1,~       88        15
## 3  1905 "9\x9630 July 1905"            11     "2,994\xa0km (1,~       60        24
## 4  1906 "4\x9629 July 1906"            13     "4,637\xa0km (2,~       82        14
## 5  1907 "8 July \x96 4 August 1907"    14     "4,488\xa0km (2,~       93        33
## 6  1908 "13 July \x96 9 August 1908"   14     "4,497\xa0km (2,~      112        36
```

You'll notice I used the `read_csv()` function of the `readr` package instead of the base R's `read.csv()`. Why not use this function always? You could! It is a matter of preference. Trying `read.csv()` first is usually a bit faster. Another difference is that the result of `read.csv()` is a data frame, while the result of `readr::read_csv()` is a tibble. Both behave similarly and differences are too marginal to impact our goals here. Take-home: try `read.csv()` first and if you get an encoding error, try `readr::read_csv()`.

Another handy alternative is using the `import()` function from the `rio` package, which can load nearly any file type (text, binary, with or without encoding quirks) and return a clean dataset. How it does it? I have no idea! It's like a tiny bit of magic for your workflow. However, since `rio` is less used, if you plan to share your code, it's often safer to use `read.csv()` or `read_csv()`. But it is a good trick to know. Here it is:

```
tours <- rio::import('./data/tdf_tours.csv')
head(tours)
```

```
##   Year                      Dates Stages                 Distance Starters
## 1 1903          1\x9619 July 1903      6 2,428\xa0km (1,509\xa0mi)       60
## 2 1904          2\x9624 July 1904      6 2,428\xa0km (1,509\xa0mi)       88
## 3 1905          9\x9630 July 1905     11 2,994\xa0km (1,860\xa0mi)       60
## 4 1906          4\x9629 July 1906     13 4,637\xa0km (2,881\xa0mi)       82
## 5 1907  8 July \x96 4 August 1907     14 4,488\xa0km (2,789\xa0mi)       93
## 6 1908 13 July \x96 9 August 1908     14 4,497\xa0km (2,794\xa0mi)      112
##   Finishers
## 1        21
## 2        15
## 3        24
## 4        14
## 5        33
## 6        36
```

Both methods will give you the same data, with differences in appearance due to how tibbles (`readr`) and data frames (`rio`) are displayed in your console.

Wow, that was already a journey, wasn't it? Loading datasets is often treated as trivial, so most tutorials use clean, pre-processed data. It's only when you're working with your own real-world data that you discover the dark side of the moon.

But we made it back stronger, and now we can move on to the fun part: analysing the data.

> **Key takeaway:** Don't underestimate the challenge of loading your data AND checking they are properly loaded. If you think it will take five minutes, you will most likely "lose" time here. The unexpected always happens when using real data. I usually block out 30 to 60 minutes for loading and checking my data. If it goes smoothly, great, I get that time back. If it doesn't, at least I avoid unnecessary stress and can keep your workflow focused.

5. What Can We Learn From These Data?

5.1 How Many Stage Winners in History?

This simple question requires us to look closely at the `stages` table, specifically at the `Winner` column. However, we immediately notice that this information isn't unique, as many riders have won multiple stages throughout their careers.

To find out how many distinct riders have won at least one stage, we extract the `Winner` column using the `$` symbol and apply the `unique()` function. This function takes a vector as input (*i.e.*, the column from the data frame) and returns a new vector containing only the unique names, removing duplicates. Once we have this vector, here called `stage_winners`, we can use `length()` to count the number of unique stage winners.

```
stage_winners <- unique(stages$Winner)
length(stage_winners)
```

```
## [1] 917
```

917 distinct riders have won at least one Tour de France stage.

Key takeaway: Simple questions often require careful data preparation before answering. Always check whether your data contains duplicates when counting "unique" entities, as failing to do so will lead to incorrect conclusions. Keep in mind that typos may bias your estimation.

5.2 How Many Unique Tour de France Winners in History?

At first glance, this question looks similar to the previous one, right? Yet our code will differ due to the structure of the data.

If the `tours` table had a `Winner` column, we could have used the same approach as in Q1. However, the `tours` table does not provide winner names. Instead, the complete rankings for each Tour de France are in the `finishers` table.

In the `finishers` table, a Tour de France winner is defined as a `Rider` whose `Rank` is 1. This means we need to combine information from two columns to get our answer: we need to identify the riders with `Rank` equal to 1, extract their names, and then count the number of unique names, just as we did in Q1.

There are several ways to subset a data frame in R. One common approach is to select rows based on their row numbers using a vector of indexes, like this:

```
finishers[c(1, 2, 4, 7), ]
```

```
##   Year Rank                    Rider         Time         Team
## 1 1903    1      Maurice Garin (FRA)  94h 33' 14" La Française
## 2 1903    2     Lucien Pothier (FRA) + 2h 59' 21" La Française
## 4 1903    4 Rodolfo Muller[27] (ITA) + 4h 39' 30" La Française
## 7 1903    7     Julien Lootens (BEL) + 8h 31' 08"    Brennabor
```

But this requires knowing the row indexes, which we don't in this case. We could do it by hand, but the `finishers` table has 9895 rows, which would take us ages AND would be prone to errors. For this analysis, it's more advantageous to use a logical condition to extract the rows we need.

We need to identify all the rows where Rank is equal to 1. It can be done as follow:

```
rows_winners <- finishers$Rank == 1
head(rows_winners)
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

where the `==` operator compares each value in the `Rank` column to `1` and returns `TRUE` when the value is `1`, and `FALSE` otherwise. This creates a logical (boolean) vector that we can inject into our data frame to filter rows, allowing us to select only the columns of interest for those matching conditions.

```
tour_winners <- finishers[rows_winners, "Rider"]
tour_winners
```

```
##   [1] "Maurice Garin (FRA)"       "Henri Cornet (FRA)"
##   [3] "Louis Trousselier (FRA)"   "René Pottier (FRA)"
##   [5] "Lucien Petit-Breton (FRA)" "Lucien Petit-Breton (FRA)"
##   [7] "François Faber (LUX)"      "Octave Lapize (FRA)"
##   [9] "Gustave Garrigou (FRA)"    "Odile Defraye (BEL)"
##  [11] "Philippe Thys (BEL)"       "Philippe Thys (BEL)"
##  [13] "Firmin Lambot (BEL)"       "Philippe Thys (BEL)"
##  [15] "Leon Scieur (BEL)"         "Firmin Lambot (BEL)"
##  [17] "Henri Pélissier (FRA)"     "Ottavio Bottecchia (ITA)"
##  [19] "Ottavio Bottecchia (ITA)"  "Lucien Buysse (BEL)"
##  [21] "Nicolas Frantz (LUX)"      "Nicolas Frantz (LUX)"
##  [23] "Maurice De Waele (BEL)"    "André Leducq (FRA)"
##  [25] "Antonin Magne (FRA)"       "André Leducq (FRA)"
##  [27] "Georges Speicher (FRA)"    "Antonin Magne (FRA)"
##  [29] "Romain Maes (BEL)"         "Sylvère Maes (BEL)"
##  [31] "Roger Lapébie (FRA)"       "Gino Bartali (ITA)"
##  [33] "Sylvère Maes (BEL)"        "Jean Robic (FRA)"
##  [35] "Gino Bartali (ITA)"        "Fausto Coppi (ITA)"
##  [37] "Ferdinand Kübler (SUI)"    "Hugo Koblet (SUI)"
##  [39] "Fausto Coppi (ITA)"        "Louison Bobet (FRA)"
##  [41] "Louison Bobet (FRA)"       "Louison Bobet (FRA)"
##  [43] "Roger Walkowiak (FRA)"     "Jacques Anquetil (FRA)"
##  [45] "Charly Gaul (LUX)"         "Federico Bahamontes (ESP)"
##  [47] "Gastone Nencini (ITA)"     "Jacques Anquetil (FRA)"
##  [49] "Jacques Anquetil (FRA)"    "Jacques Anquetil (FRA)"
##  [51] "Jacques Anquetil (FRA)"    "Felice Gimondi (ITA)"
##  [53] "Lucien Aimar (FRA)"        "Roger Pingeon (FRA)"
##  [55] "Jan Janssen (NED)"         "Eddy Merckx (BEL)"
##  [57] "Eddy Merckx (BEL)"         "Eddy Merckx (BEL)"
##  [59] "Eddy Merckx (BEL)"         "Luis Ocaña (ESP)"
##  [61] "Eddy Merckx (BEL)"         "Bernard Thévenet (FRA)"
##  [63] "Lucien Van Impe (BEL)"     "Bernard Thévenet (FRA)"
##  [65] "Bernard Hinault (FRA)"     "Bernard Hinault (FRA)"
##  [67] "Joop Zoetemelk (NED)"      "Bernard Hinault (FRA)"
##  [69] "Bernard Hinault (FRA)"     "Laurent Fignon (FRA)"
##  [71] "Laurent Fignon (FRA)"      "Bernard Hinault (FRA)"
##  [73] "Greg LeMond (USA)"         "Stephen Roche (IRE)"
##  [75] "Pedro Delgado (ESP)"       "Greg LeMond (USA)"
##  [77] "Greg LeMond (USA)"         "Miguel Induráin (ESP)"
##  [79] "Miguel Induráin (ESP)"     "Miguel Induráin (ESP)"
```

```
##  [81] "Miguel Induráin (ESP)"     "Miguel Induráin (ESP)"
##  [83] "Bjarne Riis (DEN)"         "Jan Ullrich (GER)"
##  [85] "Marco Pantani (ITA)"       "Lance Armstrong (USA)[a]"
##  [87] "Lance Armstrong (USA)[a]"  "Óscar Pereiro (ESP)"
##  [89] "Alberto Contador (ESP)"    "Carlos Sastre (ESP)"
##  [91] "Alberto Contador (ESP)"    "Andy Schleck (LUX)"
##  [93] "Cadel Evans (AUS)"         "Bradley Wiggins (GBR)"
##  [95] "Chris Froome (UK)"         "Vincenzo Nibali (ITA)"
##  [97] "Chris Froome (GBR)"        "Chris Froome (GBR)"
##  [99] "Chris Froome (GBR)"        "Geraint Thomas (GBR)"
## [101] "Egan Bernal (COL)"         "Tadej Pogačar (SLO)"
## [103] "Tadej Pogačar (SLO)"       "Jonas Vingegaard (DEN)"
```

Now we have a vector of Tour de France winners, but it still contains duplicates since some riders have won multiple times. To find the number of unique Tour de France winners, we can apply the same solution we used in Q1: use the `unique()` function to remove duplicates, and then use `length()` to count how many unique riders have won the Tour de France.

```
length(unique(tour_winners))
```

```
## [1] 66
```

66 different riders appear to have won at least one Tour de France.

But is that really the case?

A lot can go wrong when processing data, from coding mistakes (*Did I actually do the right thing?*) to issues in the input data itself. I cannot emphasise enough how important it is to **double-check your results**.

Let's go back to our last question. We identified 66 different names, but do these names actually represent 66 *different riders*?

Names in datasets are prone to errors: typos, inconsistent accents, or differences in formatting (like "Bernard Hinault" vs. "Hinault, Bernard") can cause the same rider to be counted multiple times.

To check, let's look at the **last 10 names** in our list. We can do this easily by using the `tail()` function, which is similar to the `head()` we used at the beginning:

```
last_10_winner_names <- tail(tour_winners, n=10)
last_10_winner_names
```

```
##  [1] "Chris Froome (UK)"      "Vincenzo Nibali (ITA)"  "Chris Froome (GBR)"
##  [4] "Chris Froome (GBR)"     "Chris Froome (GBR)"     "Geraint Thomas (GBR)"
##  [7] "Egan Bernal (COL)"      "Tadej Pogačar (SLO)"    "Tadej Pogačar (SLO)"
## [10] "Jonas Vingegaard (DEN)"
```

If we look closely, we can identify a name that appears twice with different spelling: `Chris Froome (UK)` and `Chris Froome (GBR)`. Given the difference between these two names, it is fair to assume they refer to the *same person*, but with the country recorded differently ("United Kingdom" vs. "Great Britain").

Assuming these two Chris Froome entries refer to the same person, **there are therefore 65 distinct Tour de France winners.**

This is still the case as the Tour 2025 just started – but will it still be the case in a couple of weeks?

> **Key takeaway:** This simple example highlights the importance of *always checking your results*. If you skip this step, small errors will slip through unnoticed, leading to incorrect results in your analysis.

> **Your turn:** I didn't check the list of stage winners for inconsistencies, but it's likely a few are hiding there as well. Why not try spotting them yourself? Look for misspellings, duplicate riders with different country codes, and inconsistent name formatting. If you find anything interesting, share it in the comments below! It's a great way to sharpen your data-checking reflexes and build confidence in your analytical workflow.

5.3 How Many Riders Won the Tour de France Without Ever Winning a Stage?

Now, this question is more interesting because it requires us to *combine information from two tables*: the `finishers` table (to identify Tour winners) and the `stages` table (to check if those winners ever won a stage).

In the previous questions, we calculated two vectors:

- `stage_winners`
- `tour_winners`

Now we need to **compare them** to identify which riders in `tour_winners` are **not** in `stage_winners`. As with most data analysis tasks, there are many routes you could take to answer this question.

One straightforward option is to use a `for` loop. Here's what the following code snippet does:

1. For every `rider` name in our `tour_winners` list,
2. Check whether that rider's name is `%in%` the `stage_winners` list,
3. Combine this boolean test with the `!` symbol (which inverts `TRUE`/`FALSE`) to check whether the rider is *not* in the `stage_winners` list,
4. If the rider is *not* in `stage_winners`, we `print` the rider's name.

```
for(rider in tour_winners) {
    if(! rider %in% stage_winners) {
        print(rider)
    }
}
```

```
## [1] "Henri Cornet (FRA)"
## [1] "Maurice De Waele (BEL)"
## [1] "Roger Walkowiak (FRA)"
## [1] "Chris Froome (UK)"
## [1] "Egan Bernal (COL)"
```

We can see five names.

Before exploring whether all these names are *truly* valid solutions to Q3, let me show you another, more compact solution that avoids using an explicit `for` loop. The `setdiff()` function takes two vectors and checks whether elements in the first vector are *not* found in the second vector.

```
GC_but_no_stages <- setdiff(tour_winners, stage_winners)
GC_but_no_stages
```

```
## [1] "Henri Cornet (FRA)"     "Maurice De Waele (BEL)" "Roger Walkowiak (FRA)"
## [4] "Chris Froome (UK)"      "Egan Bernal (COL)"
```

Unsurprisingly, we get the same five names as before.

But are these names *actually* correct?

We already know that `Chris Froome (UK)` is a problematic name. The absence of `Chris Froome (GBR)` in this list indicates that Chris Froome won stages, so we can confidently exclude `Chris Froome (UK)` from our suspect list, reducing it to **four names**.

There are two common types of typos we can easily explore at this stage:

1. **Upper and lower case inconsistencies** (e.g., "de", "De", "DE"),
2. **The use (or absence) of accents** or similar "decorations" on letters.

One name that immediately raises suspicion is **"Maurice De Waele (BEL)"**. Family names with particles (e.g., "de", "van", "von") are often spelled inconsistently across datasets.

A useful tool for checking potential case issues is the `tolower()` function (or its counterpart, `toupper()`), which standardises strings to lower (or upper) case before comparison. Additionally, instead of scanning the *entire* name, we can focus on a distinctive fragment using `str_detect()` from the `stringr` package.

> Tip: When comparing names across datasets, normalising the text using tolower() or toupper() and using partial matching with str_detect() are essential habits for catching subtle inconsistencies.

Here's how we can check if Maurice De Waele won stages under a different case or spelling. We look if any name in our list of stage winners contains the characters "waele" (lower case!).

```
stage_winners[stringr::str_detect(tolower(stage_winners), tolower('Waele'))]
```

```
## [1] "Maurice Dewaele (BEL)"
```

This returns a name that is very close to our suspect, and we can fairly assume these two names refer to the *same person*.

We can repeat this operation to check for potential inconsistencies with **Roger Walkowiak**, using either his first name or last name as search terms:

```
stage_winners[stringr::str_detect(tolower(stage_winners), tolower('Walk'))]
```

```
## character(0)
```

```
stage_winners[stringr::str_detect(tolower(stage_winners), tolower('roger'))]
```

```
## [1] "Roger Lapébie (FRA)"      "Roger Lambrecht (BEL)"
## [3] "Roger Lévêque (FRA)"      "Roger Hassenforder (FRA)"
## [5] "Roger Rivière (FRA)"      "Roger de Breuker (BEL)"
## [7] "Roger Pingeon (FRA)"      "Roger De Vlaeminck (BEL)"
## [9] "Michael Rogers (AUS)"
```

There are no riders with similar names that seem to have won stages. The same procedure can be repeated for the remaining two names. But neither Egan Bernal or Henri Cornet have ever won any Tour de France stages either.

> We can now confidently conclude that **three riders have won the Tour de France without ever winning a single stage**.

> **Key takeaway:** Data analysis often requires investigating your results carefully to validate your findings. Small inconsistencies, typos, or structural quirks in your data can distort your results if left unchecked.

6. Concluding Remarks

With this simple Tour de France dataset, we were able to practice the first principles of data analysis. When I decided to analyse this dataset, I was certain everything would go smoothly and I could focus more on showcasing more functions like `setdiff()`. However, despite the simplicity of the questions asked, this exercise has highlighted several very common issues (*e.g.* issues importing data, typos in names, duplicated entries). This post ended up highlighting how to navigate noisy data – This was not my initial goal but it was a better presentation of what data analysis truly is.

Using the Tour de France dataset, we've practised applying **first principles of data analysis** to real-world data. When I started this post, I expected everything to go smoothly so I could focus on showcasing more functions like `setdiff()`. Instead, even these simple questions revealed common challenges, such as issues with imports, typos in names, and duplicated entries.

Unexpectedly, this post became a practical lesson in **navigating noisy data**, which is a more accurate depiction of what real data analysis truly is.

6.1 Take-Home Messages

1. **Always look at your data; do not take results at face value.** Small inconsistencies can lead to large errors if left unchecked.
2. **Decompose your problems in smaller ones.** Identifying key milestones will help you reach your objective more consistently. Big objectives are hard to reach; small objectives are much easier.
3. **Be clever in how you calculate things.** Sometimes it is easier to calculate the *opposite* of what you want. Here, counting how many riders won a stage while winning the Tour was a simpler, clearer path than directly counting non-winners.
4. **There are many routes to reach your analytical goals.** Choose the one that feels most comfortable and understandable to you, especially while learning.

> **Key takeaway:** Good data analysis is not just about getting the numbers, but about understanding *why* they are correct, *how* you arrived at them, and *being mindful of the hidden details* in your data along the way.

6.2 What's Next?

Ready to keep sharpening your data skills?

In the next post, we will dive into **string manipulation**, i.e. learning how to clean and transform messy columns like `"285 km"` into usable numbers, extract country codes from rider names, and prepare your data for analysis. This is a *crucial skill* when working with real-world datasets, where "clean" data is the exception, not the rule.

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: Look for messy columns in your own projects or datasets, and think about how you might clean them using string manipulation techniques. It's a great warm-up before we jump in.

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