Transform data by using dplyr

2023-03-01

## Explore data by using R

As a data scientist, your role is principally to explore, analyze, and visualize data. There’s a wide range of tools and programming languages that you can use to do this. One of the most popular approaches is to use Jupyter notebooks, like this one, and the R language.

R is a flexible, intuitive, and elegant programming language that’s used in a wide range of industries, from banking to insurance, pharmaceutical, genetics, telecommunications, marketing, and healthcare. We think R is a great place to start your data-science adventure for the following reasons:

* It’s natively designed to support data science.
* The diverse and welcoming R community. Just to name a few: [R-Ladies](http://r-ladies.org/), [Minority in R](https://medium.com/@doritolay/introducing-mir-a-community-for-underrepresented-users-of-r-7560def7d861), and [the #rstats twitter community](https://twitter.com/search?q=%23rstats).
* R has a massive set of packages for data wrangling, statistical modeling, and machine learning. If you get stuck on a problem, there’s a high likelihood that someone has already resolved it and you can learn from or build on their work.

In this notebook, you’ll explore some of these packages and apply basic techniques to analyze data. This exercise isn’t intended to be a comprehensive R programming exercise or even a deep dive into data analysis. Rather, it’s intended as a crash course in some of the common ways in which data scientists use R to work with data.

**Note**: If you’ve never used the Jupyter notebooks environment before, here are a few things you should be aware of:

* Notebooks are made up of *cells*. Some cells, such as this one, contain *Markdown* text, and others, such as the next one, contain code.
* You can run each code cell by using the **Run** button. The **Run** button is displayed when you hover over the cell.
* The output from each code cell is displayed immediately below the cell.
* Even though the code cells can be run individually, some variables that are used in the code are global to the notebook. This means that you should run all of the code cells *in order*. There might be dependencies between code cells, so if you skip a cell, subsequent cells might not run correctly.

### Tibbles

Tibbles, or data frames, are both one of the most important concepts in R and one of the most common and useful storage structures for data analysis in R. As such, it’s a good idea to start with learning and working with tibbles, because it immediately pays off in both data transformation and visualization.

You’ll work through other data structures in R as you progress through the course.

Tibbles are provided by the tibble package, which is part of the core tidyverse. So, let’s take a trip into the tidyverse!

# Load the packages in the tidyverse into the current R session  
library (tidyverse)

## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──  
## ✔ dplyr 1.1.0 ✔ readr 2.1.4  
## ✔ forcats 1.0.0 ✔ stringr 1.5.0  
## ✔ ggplot2 3.4.1 ✔ tibble 3.1.8  
## ✔ lubridate 1.9.2 ✔ tidyr 1.3.0  
## ✔ purrr 1.0.1   
## ── Conflicts ────────────────────────────────────────── tidyverse\_conflicts() ──  
## ✖ dplyr::filter() masks stats::filter()  
## ✖ dplyr::lag() masks stats::lag()  
## ℹ Use the ]8;;http://conflicted.r-lib.org/conflicted package]8;; to force all conflicts to become errors

From the startup message, you can see that you’ve loaded or attached a bunch of packages with the tibble included. The message also shows *Conflicts*, which are functions from the tidyverse that conflict with other functions in base R or other packages you might have loaded. Don’t worry about these for now.

Good job! This means that you can now create your first tibble of student data by using tibble(), as shown in the following code:

# Build a tibble of student data  
df\_students <- tibble(  
   
 # Student names  
 name = c('Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',  
 'Frederic', 'Jimmie', 'Rhonda', 'Giovanni',  
 'Francesca', 'Rajab', 'Naiyana', 'Kian', 'Jenny',  
 'Jakeem','Helena','Ismat','Anila','Skye','Daniel',  
 'Aisha'),  
   
 # Study hours  
 study\_hours = c(10.0, 11.5, 9.0, 16.0, 9.25, 1.0, 11.5, 9.0,  
 8.5, 14.5, 15.5, 13.75, 9.0, 8.0, 15.5, 8.0,  
 9.0, 6.0, 10.0, 12.0, 12.5, 12.0),  
   
 # Grades  
 grade = c(50, 50, 47, 97, 49, 3, 53, 42, 26,  
 74, 82, 62, 37, 15, 70, 27, 36, 35,  
 48, 52, 63, 64)  
)  
  
# Print the tibble  
df\_students

## # A tibble: 22 × 3  
## name study\_hours grade  
## <chr> <dbl> <dbl>  
## 1 Dan 10 50  
## 2 Joann 11.5 50  
## 3 Pedro 9 47  
## 4 Rosie 16 97  
## 5 Ethan 9.25 49  
## 6 Vicky 1 3  
## 7 Frederic 11.5 53  
## 8 Jimmie 9 42  
## 9 Rhonda 8.5 26  
## 10 Giovanni 14.5 74  
## # … with 12 more rows

Yes! There goes your first tibble. You might have noticed that you created the tibble from individual elements: name, study\_hours, and grade. These elements are called vectors, and you create them by using c(), which is short for combine. The length of each vector in a tibble must be the same, a property that gives tibbles their rectangular structure.

You might also have noticed other abbreviations below the column names. These abbreviations describe the type of column, for example:

* chr stands for character vectors, or strings.
* dbl stands for doubles, or real numbers that might have a floating point standard.

You’ll encounter more abbreviations as you progress through this adventure.

### Load a data frame from a file

We constructed a data frame from some existing vectors, which we typed by hand. However, typing them manually could invite typos and other errors. In many real-world scenarios, data is loaded from sources such as files. You can then ask R to read the file and store the contents as an object.

Let’s replace the data frame of student grades with the contents of a CSV file.

# Read a CSV file into a tibble  
students <- read\_csv(file = "https://raw.githubusercontent.com/MicrosoftDocs/ml-basics/master/data/grades.csv")

## Rows: 24 Columns: 3  
## ── Column specification ────────────────────────────────────────────────────────  
## Delimiter: ","  
## chr (1): Name  
## dbl (2): StudyHours, Grade  
##   
## 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.

# Print the first 10 rows of the data  
slice\_head(students, n = 10)

## # A tibble: 10 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Dan 10 50  
## 2 Joann 11.5 50  
## 3 Pedro 9 47  
## 4 Rosie 16 97  
## 5 Ethan 9.25 49  
## 6 Vicky 1 3  
## 7 Frederic 11.5 53  
## 8 Jimmie 9 42  
## 9 Rhonda 8.5 26  
## 10 Giovanni 14.5 74

You use read\_csv() to import a CSV file into a tibble. With slice\_head(), you can return the first n rows of a tibble.

slice\_head() is actually a variant of the function slice. By using slice(), you can select rows based on their integer location. If you want to narrow down to rows at integer positions 5 to 10, here’s one way to do it:

#slice(students, n = 5:10)

Indexing in R starts at 1.

### Explore tibbles by using dplyr

Now that you have some data, you can begin to solve some of the common data manipulation challenges:

#### Filter rows by using dplyr::filter()

The filter() function is used to create a subset of the original data, which contains rows that satisfy your conditions.

Let’s say you’re particularly interested in focusing on Jenny’s performance. To do so, you filter the *students* data to keep only rows where the entry for *Name* exactly matches *“Jenny”*.

filter(students, Name == "Jenny")

## # A tibble: 1 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Jenny 15.5 70

Perhaps you also want to focus on Giovanni’s performance. This means that you want to keep rows where the name matches Jenny *or* Giovanni. You can approach the problem by using the %in% operator, followed by a vector of values to look for a match, as shown here:

filter(students, Name %in% c("Jenny", "Giovanni"))

## # A tibble: 2 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Giovanni 14.5 74  
## 2 Jenny 15.5 70

Here, you’re asking filter() to select all rows where the observations in the Name column match those provided in the vector of values.

Sometimes, you want to retain rows whose conditions are met in multiple columns. Let’s say you want to find the students who studied for more than 12 hours *and* got a grade of more than 80. In this case, a row should be retained only if both of those conditions are met.

You can express an *and* statement within filter() in either of the following ways:

* By using a comma between conditions:

filter(students, StudyHours > 12, Grade > 80)

## # A tibble: 2 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Rosie 16 97  
## 2 Francesca 15.5 82

* By using an ampersand (&) between conditions:

filter(students, StudyHours > 12 & Grade > 80)

## # A tibble: 2 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Rosie 16 97  
## 2 Francesca 15.5 82

#### About the pipe operator (%>%)

The pipe operator (%>%) is used to perform operations in logical sequence by passing an object forward into a function or call expression. You can think of the pipe operator as saying “and then” in your code.

The pipe operator is one of the functions in R that will really help you to work intuitively with the data, by enabling you to translate what you have in mind to actual code.

For example, let’s say you want to filter for the student named Bill. You can think of it like this:

You’re telling R to take the *students* data frame *and then* *filter* the rows that contain the name **Bill**. This can be translated into code like this:

# Take the students data frame AND THEN filter for the Name "Bill"  
students %>%   
 filter(Name == "Bill")

## # A tibble: 1 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Bill 8 NA

Such a superpower, right?

Look at what’s displayed for Bill’s grade: NA. This brings us to the next adventure.

#### Handling missing values

One of the most common issues that data scientists deal with is incomplete or missing data. R represents missing, or unknown values, with special sentinel value: NA (Not Available).

So how would you know that the data frame contains missing values?

* One way to quickly investigate whether your data contains any NAs is to use the function anyNA, which returns TRUE or FALSE.

anyNA(students)

## [1] TRUE

* Another way is to use the function is.na(), which indicates which individual elements are missing by displaying a logical TRUE.

is.na(students)

## Name StudyHours Grade  
## [1,] FALSE FALSE FALSE  
## [2,] FALSE FALSE FALSE  
## [3,] FALSE FALSE FALSE  
## [4,] FALSE FALSE FALSE  
## [5,] FALSE FALSE FALSE  
## [6,] FALSE FALSE FALSE  
## [7,] FALSE FALSE FALSE  
## [8,] FALSE FALSE FALSE  
## [9,] FALSE FALSE FALSE  
## [10,] FALSE FALSE FALSE  
## [11,] FALSE FALSE FALSE  
## [12,] FALSE FALSE FALSE  
## [13,] FALSE FALSE FALSE  
## [14,] FALSE FALSE FALSE  
## [15,] FALSE FALSE FALSE  
## [16,] FALSE FALSE FALSE  
## [17,] FALSE FALSE FALSE  
## [18,] FALSE FALSE FALSE  
## [19,] FALSE FALSE FALSE  
## [20,] FALSE FALSE FALSE  
## [21,] FALSE FALSE FALSE  
## [22,] FALSE FALSE FALSE  
## [23,] FALSE FALSE TRUE  
## [24,] FALSE TRUE TRUE

Okay, this gets the job done.

But with a larger data frame, you would find it inefficient and practically impossible to review each row and column individually.

* Another more intuitive way would be to get the sum of missing values for each column, like this:

colSums(is.na(students))

## Name StudyHours Grade   
## 0 1 2

This tells you that the last row has two NAs and the second last row has one missing value.

Great! This means that you can tell R to *filter* the rows where the sum of NAs is greater than 0:

students %>%   
 filter(rowSums(is.na(students)) > 0)

## # A tibble: 2 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Bill 8 NA  
## 2 Ted NA NA

Now that you’ve found the missing values, what can you do about them?

One common approach is to *impute* replacement values. For example, if the number of study hours is missing, you can assume that the student studied for an average amount of time and replace the missing value with the mean study hours. This raises the question, how do you modify existing columns?

#### Create and modify columns by using dplyr::mutate()

You can use mutate() to add or modify columns and preserve existing ones. Here’s how to replace the missing values found in the **StudyHours** column with the mean study hours:

# Replace NA in column StudyHours with the mean study hours  
students <- students %>%   
 mutate(StudyHours = replace\_na(StudyHours, mean(StudyHours, na.rm = TRUE)))  
  
# Print the data frame  
students

## # A tibble: 24 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Dan 10 50  
## 2 Joann 11.5 50  
## 3 Pedro 9 47  
## 4 Rosie 16 97  
## 5 Ethan 9.25 49  
## 6 Vicky 1 3  
## 7 Frederic 11.5 53  
## 8 Jimmie 9 42  
## 9 Rhonda 8.5 26  
## 10 Giovanni 14.5 74  
## # … with 14 more rows

na.rm = TRUE argument is added to exclude missing values

Awesome! You have just replaced the missing value in the StudyHour column with the mean study hours. In the process, you have also learned a new function: replace\_na(). tidyr::replace\_na replaces missing values with specified values.

[*tidyr*](https://tidyr.tidyverse.org/index.html) *is a part of the tidyverse, and its role is to help you tidy up messy data.*

Alternatively, it might be important to ensure that you use only data that you know to be absolutely correct. So let’s drop rows that contain missing values by using tidyr::drop\_na function.

# Drop NAs from our tibble  
students <- students %>%   
 drop\_na()  
  
# Print tibble  
students

## # A tibble: 22 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Dan 10 50  
## 2 Joann 11.5 50  
## 3 Pedro 9 47  
## 4 Rosie 16 97  
## 5 Ethan 9.25 49  
## 6 Vicky 1 3  
## 7 Frederic 11.5 53  
## 8 Jimmie 9 42  
## 9 Rhonda 8.5 26  
## 10 Giovanni 14.5 74  
## # … with 12 more rows

Double-check to ensure that you have no more missing values:

anyNA(students)

## [1] FALSE

Now that you’ve cleaned up the missing values, you’re ready to do more meaningful data exploration. Start by comparing the mean study hours and grades. This requires you to extract the numeric values of the individual column and pass them to the mean() function. $ and dplyr::pull() do exactly this.

# Get the mean study hours using the accessor `$`  
mean\_study <- mean(students$StudyHours)  
  
# Get the mean grade using dplyr::pull  
mean\_grade <- students %>%   
 pull(Grade) %>%   
 mean()  
  
# Print the mean study hours and mean grade  
cat(  
 'Average weekly study hours: ', round(mean\_study, 2),  
 '\nAverage grade: ', round(mean\_grade, 2)  
)

## Average weekly study hours: 10.52   
## Average grade: 49.18

With this information, you might want to filter the data frame to find only the students who studied for more than the average number of hours.

# Get students who studied for more than the average number of hours  
students %>%   
 filter(StudyHours > mean\_study)

## # A tibble: 10 × 3  
## Name StudyHours Grade  
## <chr> <dbl> <dbl>  
## 1 Joann 11.5 50  
## 2 Rosie 16 97  
## 3 Frederic 11.5 53  
## 4 Giovanni 14.5 74  
## 5 Francesca 15.5 82  
## 6 Rajab 13.8 62  
## 7 Jenny 15.5 70  
## 8 Skye 12 52  
## 9 Daniel 12.5 63  
## 10 Aisha 12 64

Note that the filtered result retained the attributes of the original tibble and is itself a tibble, so you can work with its rows and columns as you would with any other tibble.

For example, how about finding the average grade for students who spent more than the average amount of study time.

# Mean grade of students who studied more than average hours  
students %>%   
 filter(StudyHours > mean\_study) %>%   
 pull(Grade) %>%   
 mean()

## [1] 66.7

Let’s assume that the passing grade for the course is 60.

You can use that information to add a new column to the data frame, indicating whether each student passed (TRUE/FALSE). Again, this calls for using dplyr::mutate(). Earlier, you saw how to use mutate to modify existing columns. You’ll now use mutate to add a new column.

The general structure of adding new columns is basically the same as before:

df %>% mutate(new\_column\_name = what\_it\_contains)

Let’s go forth and mutate!

# TRUE/FALSE column based on whether student passed or not  
students <- students %>%   
 mutate(Pass = Grade >= 60)  
  
# Print data frame  
students

## # A tibble: 22 × 4  
## Name StudyHours Grade Pass   
## <chr> <dbl> <dbl> <lgl>  
## 1 Dan 10 50 FALSE  
## 2 Joann 11.5 50 FALSE  
## 3 Pedro 9 47 FALSE  
## 4 Rosie 16 97 TRUE   
## 5 Ethan 9.25 49 FALSE  
## 6 Vicky 1 3 FALSE  
## 7 Frederic 11.5 53 FALSE  
## 8 Jimmie 9 42 FALSE  
## 9 Rhonda 8.5 26 FALSE  
## 10 Giovanni 14.5 74 TRUE   
## # … with 12 more rows

You’ve now added a column **Pass** of type lgl.

* lgl stands for logical, vectors that contain only TRUE or FALSE.

#### Grouped summaries

Data frames are designed for rectangular data, and you can use them to perform many of the kinds of data analytics operations that you can do in a relational database, such as grouping and aggregating tables of data. In R, you achieve this by using dplyr::group\_by() %>% summarize().

* dplyr::group\_by() changes the unit of analysis from the complete dataset to individual groups.
* dplyr::summarize() creates a new data frame for summary statistics that you’ve specified.

For example, you can use dplyr::group\_by() %>% summarize() to group the student data into groups based on the **Pass** column, and then you can find the mean study time and grade for the groups of students who passed and failed the course.

# Mean study time and grade for students who passed or failed the course  
students %>%   
 group\_by(Pass) %>%   
 summarise(mean\_study = mean(StudyHours), mean\_grade = mean(Grade))

## # A tibble: 2 × 3  
## Pass mean\_study mean\_grade  
## <lgl> <dbl> <dbl>  
## 1 FALSE 8.78 38   
## 2 TRUE 14.2 73.1

Let’s say that you have many numeric columns and still want to apply the mean function across them.

With dplyr::across, it’s easy to apply a function across multiple columns. To demonstrate, let’s apply across to the preceding example.

# Mean study time and grade for students who passed or failed the course  
students %>%   
 group\_by(Pass) %>%   
 summarise(across(where(is.numeric), mean))

## # A tibble: 2 × 3  
## Pass StudyHours Grade  
## <lgl> <dbl> <dbl>  
## 1 FALSE 8.78 38   
## 2 TRUE 14.2 73.1

What if you want to determine how many students passed or failed? You would have to group your data based on the Pass column and then do a tally for each group. dplyr::count() wraps all this to give you a nice grouped count like this:

# Grouped count for Pass column  
students %>%   
 count(Pass)

## # A tibble: 2 × 2  
## Pass n  
## <lgl> <int>  
## 1 FALSE 15  
## 2 TRUE 7

Pretty succinct, right?

#### Select columns with dplyr::select()

You might occasionally be faced with datasets with hundreds of columns, in which case you might want to narrow down to some columns of interest. By using select(), you can pick or exclude columns in a data frame.

Let’s say you want to pick only the **Name** and **StudyHours** columns. Here’s how you would approach the problem:

# Select the Name and StudyHours column  
students %>%   
 select(Name, StudyHours)

## # A tibble: 22 × 2  
## Name StudyHours  
## <chr> <dbl>  
## 1 Dan 10   
## 2 Joann 11.5   
## 3 Pedro 9   
## 4 Rosie 16   
## 5 Ethan 9.25  
## 6 Vicky 1   
## 7 Frederic 11.5   
## 8 Jimmie 9   
## 9 Rhonda 8.5   
## 10 Giovanni 14.5   
## # … with 12 more rows

In some scenarios, it might be more convenient to drop a specific column rather than select all the other columns. You do this by using the ! operator.

For example, let’s keep all but the **StudyHours** column:

# Keep all columns except the StudyHours column  
select(students, !StudyHours)

## # A tibble: 22 × 3  
## Name Grade Pass   
## <chr> <dbl> <lgl>  
## 1 Dan 50 FALSE  
## 2 Joann 50 FALSE  
## 3 Pedro 47 FALSE  
## 4 Rosie 97 TRUE   
## 5 Ethan 49 FALSE  
## 6 Vicky 3 FALSE  
## 7 Frederic 53 FALSE  
## 8 Jimmie 42 FALSE  
## 9 Rhonda 26 FALSE  
## 10 Giovanni 74 TRUE   
## # … with 12 more rows

Good job!

select also has a number of helpers that allow you to select variables based on their properties. For example, you might be interested only in columns where the observations are *numeric*. Here’s how to do it:

# Select numeric columns  
students %>%   
 select(where(is.numeric))

## # A tibble: 22 × 2  
## StudyHours Grade  
## <dbl> <dbl>  
## 1 10 50  
## 2 11.5 50  
## 3 9 47  
## 4 16 97  
## 5 9.25 49  
## 6 1 3  
## 7 11.5 53  
## 8 9 42  
## 9 8.5 26  
## 10 14.5 74  
## # … with 12 more rows

select() can make working with datasets with many variables more manageable.

#### Order rows by using dplyr::arrange()

Let’s wrap up this adventure by sorting the student data by Grade, in descending order, and then assigning the resulting sorted data frame to the variable name *students\_sorted*.

To do this, you’ll need to reach into dplyr and take one more verb: arrange(), which orders the rows of a data frame by column values.

# Create a data frame with the data sorted by Grade (descending)  
students\_sorted <- students %>%  
 # Sort by descending order  
 arrange(desc(Grade))  
  
# Print data frame  
students\_sorted

## # A tibble: 22 × 4  
## Name StudyHours Grade Pass   
## <chr> <dbl> <dbl> <lgl>  
## 1 Rosie 16 97 TRUE   
## 2 Francesca 15.5 82 TRUE   
## 3 Giovanni 14.5 74 TRUE   
## 4 Jenny 15.5 70 TRUE   
## 5 Aisha 12 64 TRUE   
## 6 Daniel 12.5 63 TRUE   
## 7 Rajab 13.8 62 TRUE   
## 8 Frederic 11.5 53 FALSE  
## 9 Skye 12 52 FALSE  
## 10 Dan 10 50 FALSE  
## # … with 12 more rows

To get help on any function (arrange(), for example), run the following command:

?arrange

## Summary

That’s it for now!

It’s rare in large, complex projects that data scientists manage to arrange their data as precisely as they’d like.

Fortunately, dplyr provides simple verbs, or functions that correspond to the most common data manipulation tasks, to help you translate your thoughts into code.

By using the six verbs that learned in this exercise (filter, arrange, select, mutate, group\_by, and summarise), you’re well on your way to solving the vast majority of data manipulation challenges.

In your next workbook, you’ll look at how to create graphs and explore data in additional interesting ways.

## Further reading

To learn more about the R packages you explored in this notebook, see:

* [Tidyverse packages](https://www.tidyverse.org/packages/)